Fundamentals of Data Structures

by Ellis Horowitz and Sartaj Sahni

PREFACE

For many years a data structures course has been taught in computer science programs. Often it is regarded as a central course of the curriculum. It is fascinating and instructive to trace the history of how the subject matter for this course has changed. Back in the middle1960's the course was not entitled Data Structures but perhaps List Processing Languages. The major subjects were systems such as SLIP (by J. Weizenbaum), IPL-V (by A. Newell, C. Shaw, and H. Simon), LISP 1.5 (by J. McCarthy) and SNOBOL (by D. Farber, R. Griswold, and I. Polonsky). Then, in 1968, volume I of the Art of Computer Programming by D. Knuth appeared. His thesis was that list processing was not a magical thing that could only be accomplished within a specially designed system. Instead, he argued that the same techniques could be carried out in almost any language and he shifted the emphasis to efficient algorithm design. SLIP and IPL-V faded from the scene, while LISP and SNOBOL moved to the programming languages course. The new strategy was to explicitly construct a representation (such as linked lists) within a set of consecutive storage locations and to describe the algorithms by using English plus assembly language.

Progress in the study of data structures and algorithm design has continued. Out of this recent work has come many good ideas which we believe should be presented to students of computer science. It is our purpose in writing this book to emphasize those trends which we see as especially valuable and long lasting.

The most important of these new concepts is the need to distinguish between the specification of a data structure and its realization within an available programming language. This distinction has been mostly blurred in previous books where the primary emphasis has either been on a programming language or on representational techniques. Our attempt here has been to separate out the specification of the data structure from its realization and to show how both of these processes can be successfully accomplished. The specification stage requires one to concentrate on describing the functioning of the data structure without concern for its implementation. This can be done using English and mathematical notation, but here we introduce a programming notation called axioms. The resulting implementation independent specifications valuable in two ways: (i) to help prove that a program which uses this data structure is correct and (ii) to prove that a particular implementation of the data structure is correct. To describe a data structure in a representation independent way one needs a syntax. This can be seen at the end of section 1.1 where we also precisely define the notions of data object and data structure.

This book also seeks to teach the art of analyzing algorithms but not at the cost of undue mathematical sophistication. The value of an implementation ultimately relies on its resource utilization: time and space. This implies that the student needs to be capable of analyzing these factors. A great many analyses have appeared in the literature, yet from our perspective most students don't attempt to rigorously analyze their programs. The data structures course comes at an opportune time in their training to advance and promote these ideas. For every algorithm that is given here we supply a simple, yet rigorous worst case analysis of its behavior. In some cases the average computing time is also

derived.

The growth of data base systems has put a new requirement on data structures courses, namely to cover the organization of large files. Also, many instructors like to treat sorting and searching because of the richness of its examples of data structures and its practical application. The choice of our later chapters reflects this growing interest.

One especially important consideration is the choice of an algorithm description language. Such a choice is often complicated by the practical matters of student background and language availability. Our decision was to use a syntax which is particularly close to ALGOL, but not to restrict ourselves to a specific language. This gives us the ability to write very readable programs but at the same time we are not tied to the idiosyncracies of a fixed language. Wherever it seemed advisable we interspersed English descriptions so as not to obscure the main pointof an algorithm. For people who have not been exposed to the IF-THEN-ELSE, WHILE, REPEAT- UNTIL and a few other basic statements, section 1.2 defines their semantics via flowcharts. For those who have only FORTRAN available, the algorithms are directly translatable by the rules given in the appendix and a translator can be obtained (see appendix A). On the other hand, we have resisted the temptation to use language features which automatically provide sophisticated data structuring facilities. We have done so on several grounds. One reason is the need to commit oneself to a syntax which makes the book especially hard to read by those as yet uninitiated. Even more importantly, these automatic featules cover up the implementation detail whose mastery remains a cornerstone of the course.

The basic audience for this book is either the computer science major with at least one year of courses or a beginning graduate student with prior training in a field other than computer science. This book contains more than one semester's worth of material and several of its chapters may be skipped without harm. The following are two scenarios which may help in deciding what chapters should be covered.

The first author has used this book with sophomores who have had one semester of PL/I and one semester of assembly language. He would cover chapters one through five skipping sections 2.2, 2.3, 3.2, 4.7, 4.11, and 5.8. Then, in whatever time was left chapter seven on sorting was covered. The second author has taught the material to juniors who have had one quarter of FORTRAN or PASCAL and two quarters of introductory courses which themselves contain a potpourri of topics. In the first quarter's data structure course, chapters one through three are lightly covered and chapters four through six are completely covered. The second quarter starts with chapter seven which provides an excellent survey of the techniques which were covered in the previous quarter. Then the material on external sorting, symbol tables and files is sufficient for the remaining time. Note that the material in chapter 2 is largely mathematical and can be skipped without harm.

The paradigm of class presentation that we have used is to begin each new topic with a problem, usually chosen from the computer science arena. Once defined, a high level design of its solution is made and each data structure is axiomatically specified. A tentative analysis is done to determine which operations are critical. Implementations of the data structures are then given followed by an attempt at verifying

that the representation and specifications are consistent. The finishedalgorithm in the book is examined followed by an argument concerning its correctness. Then an analysis is done by determining the relevant parameters and applying some straightforward rules to obtain the correct computing time formula.

In summary, as instructors we have tried to emphasize the following notions to our students: (i) the ability to define at a sufficiently high level of abstraction the data structures and algorithms that are needed; (ii) the ability to devise alternative implementations of a data structure; (iii) the ability to synthesize a correct algorithm; and (iv) the abilityto analyze the computing time of the resultant program. In addition there are two underlying currents which, though not explicitly emphasized are covered throughout. The first is the notion of writing nicely structured programs. For all of the programs contained herein we have tried our best to structure them appropriately. We hope that by reading programs with good style the students will pick up good writing habits. A nudge on the instructor's part will also prove useful. The second current is the choice of examples. We have tried to use those examples which prove a point well, have application to computer programming, and exhibit some of the brightest accomplishments in computer science.

At the close of each chapter there is a list of references and selected readings. These are not meant to be exhaustive. They are a subset of those books and papers that we found to be the most useful. Otherwise, they are either historically significant or develop the material in the text somewhat further.

Many people have contributed their time and energy to improve this book. For this we would like to thank them. We wish to thank Arvind [sic], T. Gonzalez, L. Landweber, J. Misra, and D. Wilczynski, who used the book in their own classes and gave us detailed reactions. Thanks are also due to A. Agrawal, M. Cohen, A. Howells, R. Istre, D. Ledbetter, D. Musser and to our students in CS 202, CSci 5121 and 5122 who provided many insights. For administrative and secretarial help we thank M. Eul, G. Lum, J. Matheson, S. Moody, K. Pendleton, and L. Templet. To the referees for their pungent yet favorable comments we thank S. Gerhart, T. Standish, and J. Ullman. Finally, we would like to thank our institutions, the University of Southern California and the University of Minnesota, for encouraging in every way our efforts to produce this book.

Ellis Horowitz

Sartaj Sahni

Preface to the Ninth Printing

We would like to acknowledge collectively all of the individuals who have sent us comments and corrections since the book first appeared. For this printing we have made many corrections and improvements.

October 198l

Ellis Horowitz

Sartaj Sahni

◀   ■   ▶

CHAPTER 1: INTRODUCTION

1.1 OVERVIEW

The field of *computer science* is so new that one feels obliged to furnish a definition before proceeding with this book. One often quoted definition views computer science as the *study of algorithms*. This study encompasses four distinct areas:

(i) *machines for executing algorithms*--this area includes everything from the smallest pocket calculator to the largest general purpose digital computer. The goal is to study various forms of machine fabrication and organization so that algorithms can be effectively carried out.

(ii) *languages for describing algorithms*--these languages can be placed on a continuum. At one end are the languages which are closest to the physical machine and at the other end are languages designed for sophisticated problem solving. One often distinguishes between two phases of this area: language design and translation. The first calls for methods for specifying the syntax and semantics of a language. The second requires a means for translation into a more basic set of commands.

(iii) *foundations of algorithms*--here people ask and try to answer such questions as: is a particular task accomplishable by a computing device; or what is the minimum number of operations necessary for any algorithm which performs a certain function? Abstract models of computers are devised so that these properties can be studied.

(iv) *analysis of algorithms*--whenever an algorithm can be specified it makes sense to wonder about its behavior. This was realized as far back as 1830 by Charles Babbage, the father of computers. An algorithm's behavior pattern or *performance profile* is measured in terms of the computing time and space that are consumed while the algorithm is processing. Questions such as the worst and average time and how often they occur are typical.

We see that in this definition of computer science, "algorithm" is a fundamental notion. Thus it deserves a precise definition. The dictionary's definition "any mechanical or recursive computational procedure" is not entirely satisfying since these terms are not basic enough.

Definition: An *algorithm* is a finite set of instructions which, if followed, accomplish a particular task. In addition every algorithm must satisfy the following criteria:

(i) *input:* there are zero or more quantities which are externally supplied;

(ii) *output:* at least one quantity is produced;

(iii) *definiteness:* each instruction must be clear and unambiguous;

(iv) *finiteness:* if we trace out the instructions of an algorithm, then for all cases the algorithm will terminate after a finite number of steps;

(v) *effectiveness:* every instruction must be sufficiently basic that it can in principle be carried out by a person using only pencil and paper. It is not enough that each operation be definite as in (iii), but it must also be feasible.

In formal computer science, one distinguishes between an algorithm, and a program. A program does not necessarily satisfy condition (iv). One important example of such a program for a computer is its operating system which never terminates (except for system crashes) but continues in a wait loop until more jobs are entered. In this book we will deal strictly with programs that always terminate. Hence, we will use these terms interchangeably.

An algorithm can be described in many ways. A natural language such as English can be used but we must be very careful that the resulting instructions are definite (condition iii). An improvement over English is to couple its use with a graphical form of notation such as flowcharts. This form places each processing step in a "box" and uses arrows to indicate the next step. Different shaped boxes stand for different kinds of operations. All this can be seen in figure 1.1 where a flowchart is given for obtaining a Coca-Cola from a vending machine. The point is that algorithms can be devised for many common activities.

Have you studied the flowchart? Then you probably have realized that it isn't an algorithm at all! Which properties does it lack?

Returning to our earlier definition of computer science, we find it extremely unsatisfying as it gives us no insight as to why the computer is revolutionizing our society nor why it has made us re-examine certain basic assumptions about our own role in the universe. While this may be an unrealistic demand on a definition even from a technical point of view it is unsatisfying. The definition places great emphasis on the concept of algorithm, but never mentions the word "data". If a computer is merely a means to an end, then the means may be an algorithm but the end is the transformation of data. That is why we often hear a computer referred to as a data processing machine. Raw data is input and algorithms are used to transform it into refined data. So, instead of saying that computer science is the study of algorithms, alternatively, we might say that computer science is the *study of data*:

(i) machines that hold data;

(ii) languages for describing data manipulation;

(iii) foundations which describe what kinds of refined data can be produced from raw data;

(iv) structures for representing data.

Figure 1.1: Flowchart for obtaining a Coca-Cola

There is an intimate connection between the structuring of data, and the synthesis of algorithms. In fact, a data structure and an algorithm should be thought of as a unit, neither one making sense without the other. For instance, suppose we have a list of n pairs of names and phone numbers $(a_1, b_1)(a_2, b_2), ..., (a_n, b_n)$, and we want to write a program which when given any name, prints that person's phone number. This task is called searching. Just how we would write such an algorithm critically depends upon how the names and phone numbers are stored or structured. One algorithm might just forge ahead and examine names, $a_1, a_2, a_3, ...$ etc., until the correct name was found. This might be fine in Oshkosh, but in Los Angeles, with hundreds of thousands of names, it would not be practical. If, however, we knew that the data was structured so that the names were in alphabetical order, then we could do much better. We could make up a second list which told us for each letter in the alphabet, where the first name with that letter appeared. For a name beginning with, say, S, we would avoid having to look at names beginning with other letters. So because of this new structure, a very different algorithm is possible. Other ideas for algorithms become possible when we realize that we can organize the data as we wish. We will discuss many more searching strategies in Chapters 7 and 9.

Therefore, computer science can be defined as the study of data, its representation and transformation by a digital computer. The goal of this book is to explore many different kinds of data objects. For each object, we consider the class of operations to be performed and then the way to represent this object so that these operations may be efficiently carried out. This implies a mastery of two techniques: the ability to devise alternative forms of data representation, and the ability to analyze the algorithm which operates on that structure . The pedagogical style we have chosen is to consider problems which have arisen often in computer applications. For each problem we will specify the data object or objects and what is to be accomplished. After we have decided upon a representation of the objects, we will give a complete algorithm and analyze its computing time. After reading through several of these examples you should be confident enough to try one on your own.

There are several terms we need to define carefully before we proceed. These include data structure, data object, data type and data representation. These four terms have no standard meaning in computer science circles, and they are often used interchangeably.

A *data type* is a term which refers to the kinds of data that variables may "hold" in a programming language. In FORTRAN the data types are INTEGER, REAL, LOGICAL, COMPLEX, and DOUBLE PRECISION. In PL/I there is the data type CHARACTER. The fundamental data type of SNOBOL is the character string and in LISP it is the list (or S-expression). With every programming language there is a set of built-in data types. This means that the language allows variables to name data of that type and

provides a set of operations which meaningfully manipulates these variables. Some data types are easy to provide because they are already built into the computer's machine language instruction set. Integer and real arithmetic are examples of this. Other data types require considerably more effort to implement. In some languages, there are features which allow one to construct combinations of the built-in types. In COBOL and PL/I this feature is called a STRUCTURE while in PASCAL it is called a RECORD. However, it is not necessary to have such a mechanism. All of the data structures we will see here can be reasonably built within a conventional programming language.

Data object is a term referring to a set of elements, say *D*. For example the data object *integers* refers to $D = \{0, \pm 1, \pm 2, ...\}$. The data object *alphabetic character strings of length less than thirty one* implies $D = \{'',' A',' B', ...,' Z',' AA', ...\}$. Thus, *D* may be finite or infinite and if *D* is very large we may need to devise special ways of representing its elements in our computer.

The notion of a data structure as distinguished from a data object is that we want to describe not only the set of objects, but the way they are related. Saying this another way, we want to describe the set of operations which may legally be applied to elements of the data object. This implies that we must specify the set of operations and show how they work. For integers we would have the arithmetic operations +, -, *, / and perhaps many others such as mod, ceil, floor, greater than, less than, etc. The data object integers plus a description of how +, -, *, /, etc. behave constitutes a data structure definition.

To be more precise lets examine a modest example. Suppose we want to define the data structure natural number (abbreviated natno) where natno = {0,1,2,3, ...} with the three operations being a test for zero addition and equality. The following notation can be used:

structure *NATNO*

1   **declare** *ZERO( )* $\Box$ *natno*

2   *ISZERO(natno)* $\Box$ *boolean*

3   *SUCC(natno)* $\Box$ *natno*

4   *ADD(natno, natno)* $\Box$ *natno*

5   *EQ(natno, natno)* $\Box$ *boolean*

6   **for all** *x, y* $\in$ *natno* **let**

7   *ISZERO(ZERO)* *::=* **true;** *ISZERO(SUCC(x))* *::=* **false**

8   *ADD(ZERO, y) :: = y, ADD(SUCC(x), y) :: =*

SUCC(ADD(x, y))

9   *EQ(x, ZERO) :: =* **if** *ISZERO(x)* **then true else false**

10   *EQ(ZERO, SUCC(y)) :: =* **false**

EQ(SUCC(x), SUCC(y)) :: = EQ(x, y)

11   **end**

end *NATNO*

In the declare statement five functions are defined by giving their names, inputs and outputs. ZERO is a constant function which means it takes no input arguments and its result is the natural number zero, written as ZERO. ISZERO is a boolean function whose result is either **true** or **false**. SUCC stands for successor. Using ZERO and SUCC we can define all of the natural numbers as: ZERO, l = SUCC (ZERO), 2 = SUCC(SUCC(ZERO)), 3 = SUCC(SUCC(SUCC(ZERO))), ... etc. The rules on line 8 tell us exactly how the addition operation works. For example if we wanted to add two and three we would get the following sequence of expressions:

ADD(SUCC(SUCC(ZERO)),SUCC(SUCC(SUCC(ZERO))))

which, by line 8 equals

SUCC(ADD(SUCC(ZERO),SUCC(SUCC(SUCC(ZERO)))))

which, by line 8 equals

SUCC(SUCC(ADD(ZERO,SUCC(SUCC(SUCC(ZERO))))))

which by line 8 equals

SUCC(SUCC(SUCC(SUCC(SUCC(ZERO)))))

Of course, this is not the way to implement addition. In practice we use bit strings which is a data structure that is usually provided on our computers. But however the ADD operation is implemented, it must obey these rules. Hopefully, this motivates the following definition.

Definition: A *data structure* is a set of domains ☐, a designated domain ☐, a set of functions ☐ and a

set of axioms ☐. The triple ☐ denotes the data structure *d* and it will usually be abbreviated by writing *d*.

In the previous example

☐

The set of axioms describes the semantics of the operations. The form in which we choose to write the axioms is important. Our goal here is to write the axioms in a representation independent way. Then, we discuss ways of implementing the functions using a conventional programming language.

An *implementation of a data structure d* is a mapping from *d* to a set of other data structures *e*. This mapping specifies how every object of *d* is to be represented by the objects of *e*. Secondly, it requires that every function of *d* must be written using the functions of the implementing data structures *e*. Thus we say that integers are represented by bit strings, boolean is represented by zero and one, an array is represented by a set of consecutive words in memory.

In current parlance the triple ☐ is referred to as an *abstract data type*. It is called abstract precisely because the axioms do not imply a form of representation. Another way of viewing the implementation of a data structure is that it is the process of refining an abstract data type until all of the operations are expressible in terms of directly executable functions. But at the first stage a data structure should be designed so that we know *what* it does, but not necessarily *how* it will do it. This division of tasks, called specification and implementation, is useful because it helps to control the complexity of the entire process.

1.2 SPARKS

The choice of an algorithm description language must be carefully made because it plays such an important role throughout the book. We might begin by considering using some existing language; some names which come immediately to mind are ALGOL, ALGOL-W, APL, COBOL, FORTRAN, LISP, PASCAL, PL/I, SNOBOL.

Though some of these are more preferable than others, the choice of a specific language leaves us with many difficulties. First of all, we wish to be able to write our algorithms without dwelling on the idiosyncracies of a given language. Secondly, some languages have already provided the mechanisms we wish to discuss. Thus we would have to make pretense to build up a capability which already exists. Finally, each language has its followers and its detractors. We would rather not have any individual rule us out simply because he did not know or, more particularly, disliked to use the language *X*.

Furthermore it is not really necessary to write programs in a language for which a compiler exists. Instead we choose to use a language which is tailored to describing the algorithms we want to write.

Using it we will not have to define many aspects of a language that we will never use here. Most importantly, the language we use will be close enough to many of the languages mentioned before so that a hand translation will be relatively easy to accomplish. Moreover, one can easily program a translator using some existing, but more primitive higher level language as the output (see Appendix A). We call our language SPARKS. Figure 1.2 shows how a SPARKS program could be executed on any machine.

Figure 1.2: Translation of SPARKS

Many language designers choose a name which is an acronym. But SPARKS was not devised in that way; it just appeared one day as Athena sprang from the head of Zeus. Nevertheless, computerniks still try to attach a meaning. Several cute ideas have been suggested, such as

Structured **P**rogramming: **A R**easonably **K**omplete **S**et

or

Smart **P**rogrammers **A**re **R**equired To **K**now **S**PARKS.

SPARKS contains facilities to manipulate numbers, boolean values and characters. The way to assign values is by the assignment statement

variable ☐ expression.

In addition to the assignment statement, SPARKS includes statements for conditional testing, iteration, input-output, etc. Several such statements can be combined on a single line if they are separated by a semi-colon. Expressions can be either arithmetic, boolean or of character type. In the boolean case there can be only one of two values,

true or **false.**

In order to produce these values, the logical operators

and, or, not

are provided, plus the relational operators

A conditional statement has the form

if *cond* **then** S_1 **if** *cond* **then** S_1

or

else S_2

where cond is a boolean expression and S_1, S_2 are arbitrary groups of SPARKS statements. If S_1 or S_2 contains more than one statement, these will be enclosed in square brackets. Brackets must be used to show how each **else** corresponds to one **if**. The meaning of this statement is given by the flow charts:

We will assume that conditional expressions are evaluated in "short circuit" mode; given the boolean expression (cond1 **or** cond2), if condl is true then cond2 is not evaluated; or, given (condl **and** cond2), if cond1 is false then cond2 is not evaluated.

To accomplish iteration, several statements are available. One of them is

while *cond* **do**

S

end

where *cond* is as before, *S* is as S_1 before and the meaning is given by

It is well known that all "proper" programs can be written using only the assignment, conditional and while statements. This result was obtained by Bohm and Jacopini. Though this is very interesting from a theoretical viewpoint, we should not take it to mean that this is the way to program. On the contrary, the more expressive our languages are, the more we can accomplish easily. So we will provide other statements such as a second iteration statement, the **repeat-until**,

repeat

S

until *cond*

which has the meaning

□

In contrast to the **while** statement, the **repeat-until** guarantees that the statements of *S* will be executed at least once. Another iteration statement is

loop

S

forever

which has the meaning

□

As it stands, this describes an infinite loop! However, we assume that this statement is used in conjunction with some test within *S* which will cause an exit. One way of exiting such a loop is by using a

go to *label*

statement which transfers control to "label." Label may be anywhere in the procedure. A more restricted form of the **go to** is the command

exit

which will cause a transfer of control to the first statement after the innermost loop which contains it. This looping statement may be a **while, repeat, for** or a **loop-forever. exit** can be used either conditionally or unconditionally, for instance

loop

S_1

if *cond* **then exit**

S_2

forever

which will execute as

☐

The last statement for iteration is called the **for-loop**, which has the form

for *vble* ☐ *start* **to** *finish* **by** *increment* **do**

S

end

Vble is a variable, while *start, finish* and *increment* are arithmetic expressions. A variable or a constant is a simple form of an expression. The clause "**by** *increment*" is optional and taken as +1 if it does not occur. We can write the meaning of this statement in SPARKS as

vble ☐ *start*

fin ☐ *finish*

incr ☐ *increment*

while (*vble* - *fin*) * *incr* ≤0 **do**

S

vble☐*vble* + *incr*

end

Another statement within SPARKS is the **case,** which allows one to distinguish easily between several alternatives without using multiple **if**-then-**else** statements. It has the form

☐

where the S_i, $1 \leq i \leq n + 1$ are groups of SPARKS statements. The semantics is easily described by the

following flowchart:

The **else** clause is optional.

A complete SPARKS procedure has the form

procedure NAME (parameter list)

S

end

A procedure can be used as a function by using the statement

return (*expr*)

where the value of *expr* is delivered as the value of the procedure. The *expr* may be omitted in which case a return is made to the calling procedure. The execution of an **end** at the end of procedure implies a return. A procedure may be invoked by using a **call** statement

call NAME (*parameter list*)

Procedures may call themselves, direct recursion, or there may be a sequence resulting in indirect recursion. Though recursion often carries with it a severe penalty at execution time, it remains all elegant way to describe many computing processes. This penalty will not deter us from using recursion. Many such programs are easily translatable so that the recursion is removed and efficiency achieved.

A complete SPARKS program is a collection of one or more procedures, the first one taken as the main program. All procedures are treated as external, which means that the only means for communication between them is via parameters. This may be somewhat restrictive in practice, but for the purpose of exposition it helps to list all variables explicitly, as either local or parameter. The association of actual to formal parameters will be handled using the call by reference rule. This means that at run time the address of each parameter is passed to the called procedure. Parameters which are constants or values of expressions are stored into internally generated words whose addresses are then passed to the procedure.

For input/output we assume the existence of two functions

read (*argument list*), **print** (*argument list*)

Arguments may be variables or quoted strings. We avoid the problem of defining a "format" statement as we will need only the simplest form of input and output.

The command **stop** halts execution of the currently executing procedure. Comments may appear anywhere on a line enclosed by double slashes, e.g.

//this is a comment//

Finally, we note that multi-dimensional arrays are available with arbitrary integer lower and upper bounds. An *n-dimensional array* A *with lower and upper bounds* l_i, u_i, *$1 \leq i \leq n$ may be declared by using the syntax* **declare** *$A(l_1:u_1, ...,l_n:u_n)$. We have avoided introducing the record or structure concept. These are often useful features and when available they should be used. However, we will persist in building up a structure from the more elementary array concept.* Finally, we emphasize that all of our variables are assumed to be of type INTEGER unless stated otherwise.

Since most of the SPARKS programs will be read many more times than they will be executed, we have tried to make the code readable. This is a goal which should be aimed at by everyone who writes programs. The SPARKS language is rich enough so that one can create a good looking program by applying some simple rules of style.

(i) Every procedure should carefully specify its input and output variables.

(ii) The meaning of variables should be defined.

(iii) The flow of the program should generally be forward except for normal looping or unavoidable instances.

(iv) Indentation rules should be established and followed so that computational units of program text can more easily be identified.

(v) Documentation should be short, but meaningful. Avoid sentences like "*i* is increased by one."

(vi) Use subroutines where appropriate.

See the book *The Elements of Programming Style* by Kernighan and Plauger for more examples of good rules of programming.

1.3 HOW TO CREATE PROGRAMS

Now that you have moved beyond the first course in computer science, you should be capable of

developing your programs using something better than the seat-of-the-pants method. This method uses the philosophy: write something down and then try to get it working. Surprisingly, this method is in wide use today, with the result that an average programmer on an average job turns out only between five to ten lines of correct code per day. We hope your productivity will be greater. But to improve requires that you apply some discipline to the process of creating programs. To understand this process better, we consider it as broken up into five phases: requirements, design, analysis, coding, and verification.

(i) *Requirements*. Make sure you understand the information you are given (the input) and what results you are to produce (the output). Try to write down a rigorous description of the input and output which covers all cases.

You are now ready to proceed to the design phase. Designing an algorithm is a task which can be done independently of the programming language you eventually plan to use. In fact, this is desirable because it means you can postpone questions concerning *how* to represent your data and *what* a particular statement looks like and concentrate on the order of processing.

(ii) *Design*. You may have several data objects (such as a maze, a polynomial, or a list of names). For each object there will be some basic operations to perform on it (such as print the maze, add two polynomials, or find a name in the list). Assume that these operations already exist in the form of procedures and write an algorithm which solves the problem according to the requirements. Use a notation which is natural to the way you wish to describe the order of processing.

(iii) *Analysis*. Can you think of another algorithm? If so, write it down. Next, try to compare these two methods. It may already be possible to tell if one will be more desirable than the other. If you can't distinguish between the two, choose one to work on for now and we will return to the second version later.

(iv) *Refinement and coding*. You must now choose representations for your data objects (a maze as a two dimensional array of zeros and ones, a polynomial as a one dimensional array of degree and coefficients, a list of names possibly as an array) and write algorithms for each of the operations on these objects. The order in which you do this may be crucial, because once you choose a representation, the resulting algorithms may be inefficient. Modern pedagogy suggests that all processing which is independent of the data representation be written out first. By postponing the choice of how the data is stored we can try to isolate what operations depend upon the choice of data representation. You should consider alternatives, note them down and review them later. Finally you produce a complete version of your first program.

It is often at this point that one realizes that a much better program could have been built. Perhaps you should have chosen the second design alternative or perhaps you have spoken to a friend who has done it better. This happens to industrial programmers as well. If you have been careful about keeping track of your previous work it may not be too difficult to make changes. One of the criteria of a good design is

that it can absorb changes relatively easily. It is usually hard to decide whether to sacrifice this first attempt and begin again or just continue to get the first version working. Different situations call for different decisions, but we suggest you eliminate the idea of working on both at the same time. If you do decide to scrap your work and begin again, you can take comfort in the fact that it will probably be easier the second time. In fact you may save as much debugging time later on by doing a new version now. This is a phenomenon which has been observed in practice.

The graph in figure 1.3 shows the time it took for the same group to build 3 FORTRAN compilers (A, B and C). For each compiler there is the time they estimated it would take them and the time it actually took. For each subsequent compiler their estimates became closer to the truth, but in every case they underestimated. Unwarrented optimism is a familiar disease in computing. But prior experience is definitely helpful and the time to build the third compiler was less than one fifth that for the first one.

Figure 1.3: History of three FORTRAN compilers

(v) *Verification*. Verification consists of three distinct aspects: program proving, testing and debugging. Each of these is an art in itself. Before executing your program you should attempt to prove it is correct. Proofs about programs are really no different from any other kinds of proofs, only the subject matter is different. If a correct proof can be obtained, then one is assured that for all possible combinations of inputs, the program and its specification agree. Testing is the art of creating sample data upon which to run your program. If the program fails to respond correctly then debugging is needed to determine what went wrong and how to correct it. One proof tells us more than any finite amount of testing, but proofs can be hard to obtain. Many times during the proving process errors are discovered in the code. The proof can't be completed until these are changed. This is another use of program proving, namely as a methodology for discovering errors. Finally there may be tools available at your computing center to aid in the testing process. One such tool instruments your source code and then tells you for every data set: (i) the number of times a statement was executed, (ii) the number of times a branch was taken, (iii) the smallest and largest values of all variables. As a minimal requirement, the test data you construct should force every statement to execute and every condition to assume the value true and false at least once.

One thing you have forgotten to do is to document. But why bother to document until the program is entirely finished and correct ? Because for each procedure you made some assumptions about its input and output. If you have written more than a few procedures, then you have already begun to forget what those assumptions were. If you note them down with the code, the problem of getting the procedures to work together will be easier to solve. The larger the software, the more crucial is the need for documentation.

The previous discussion applies to the construction of a single procedure as well as to the writing of a large software system. Let us concentrate for a while on the question of developing a single procedure which solves a specific task. This shifts our emphasis away from the management and integration of the

various procedures to the disciplined formulation of a single, reasonably small and well-defined task. The design process consists essentially of taking a proposed solution and successively refining it until an executable program is achieved. The initial solution may be expressed in English or some form of mathematical notation. At this level the formulation is said to be abstract because it contains no details regarding how the objects will be represented and manipulated in a computer. If possible the designer attempts to partition the solution into logical subtasks. Each subtask is similarly decomposed until all tasks are expressed within a programming language. This method of design is called the *top-down* approach. Inversely, the designer might choose to solve different parts of the problem directly in his programming language and then combine these pieces into a complete program. This is referred to as the *bottom-up* approach. Experience suggests that the top-down approach should be followed when creating a program. However, in practice it is not necessary to unswervingly follow the method. A look ahead to problems which may arise later is often useful.

Underlying all of these strategies is the assumption that a language exists for adequately describing the processing of data at several abstract levels. For this purpose we use the language SPARKS coupled with carefully chosen English narrative. Such an algorithm might be called pseudo-SPARKS. Let us examine two examples of top-down program development.

Suppose we devise a program for sorting a set of $n \geq 1$ distinct integers. One of the simplest solutions is given by the following

"from those integers which remain unsorted, find the smallest and place it next in the sorted list"

This statement is sufficient to construct a sorting program. However, several issues are not fully specified such as where and how the integers are initially stored and where the result is to be placed. One solution is to store the values in an array in such a way that the i-th integer is stored in the i-th array position, $A(i)$ $1 \leq i \leq n$. We are now ready to give a second refinement of the solution:

```
for i  1 to n do

examine A(i) to A(n) and suppose the

smallest integer is at A(j); then

interchange A(i) and A(j).

end
```

Note how we have begun to use SPARKS pseudo-code. There now remain two clearly defined subtasks: (i) to find the minimum integer and (ii) to interchange it with $A(i)$. This latter problem can be solved by the code

t □ A(i); A(i) □ A(j); A(j) □ t

The first subtask can be solved by assuming the minimum is $A(i)$, checking $A(i)$ with $A(i + 1)$, $A(i + 2)$, ... and whenever a smaller element is found, regarding it as the new minimum. Eventually $A(n)$ is compared to the current minimum and we are done. Putting all these observations together we get

procedure *SORT(A,n)*

1   **for** *i* □ 1 **to** *n* **do**

2   *j* □ *i*

3   **for** *k* □ *j* + 1 *to* n *do*

4   **if** *A(k) < A(j)* **then** *j* □ *k*

5   **end**

6   *t* □ *A(i); A(i)* □ *A(j); A(j)* □ *t*

7   **end**

end *SORT*

The obvious question to ask at this point is: "does this program work correctly?"

Theorem: Procedure SORT (A,n) correctly sorts a set of $n \geq 1$ distinct integers, the result remains in A $(1:n)$ such that $A(1) < A(2) < ... < A(n)$.

Proof: We first note that for any i, say $i = q$, following the execution of lines 2 thru 6, it is the case that $A(q) \leq A(r)$, $q < r \leq n$. Also, observe that when i becomes greater than q, $A(1 .. q)$ is unchanged. Hence, following the last execution of these lines, (i.e., $i = n$), we have $A(1) \leq A(2) \leq ... \leq A(n)$.

We observe at this point that the upper limit of the **for**-loop in line 1 can be changed to $n - 1$ without damaging the correctness of the algorithm.

From the standpoint of readability we can ask if this program is good. Is there a more concise way of describing this algorithm which will still be as easy to comprehend? Substituting **while** statements for the **for** loops doesn't significantly change anything. Also, extra initialization and increment statements would be required. We might consider a FORTRAN version using the ANSI language standard

```
IF (N. LE. 1) GO TO 100

NM1 = N - 1

DO 101 I = 1, NM1

J = I

JP1 = J + 1

DO 102 K = JP1, N

IF (A(K).LT.A(J)) J = K

102     CONTINUE

T = A(I)

A(I) = A(J)

A(J) = T

101 CONTINUE

100 CONTINUE
```

FORTRAN forces us to clutter up our algorithms with extra statements. The test for $N = 1$ is necessary because FORTRAN DO-LOOPS always insist on executing once. Variables NM1 and JP1 are needed because of the restrictions on lower and upper limits of DO-LOOPS.

Let us develop another program. We assume that we have $n \geq 1$ distinct integers which are already sorted and stored in the array $A(1:n)$. Our task is to determine if the integer x is present and if so to return j such that $x = A(j)$; otherwise return $j = 0$. By making use of the fact that the set is sorted we conceive of the following efficient method:

"let A(mid) be the middle element. There are three possibilities. Either x < A(mid) in which case x can only occur as A(1) to A(mid - 1); or x > A(mid) in which case x can only occur as A(mid + 1) to A(n): or x = A(mid) in which case set j to mid and return. Continue in this way by keeping two pointers, lower and upper, to indicate the range of elements not yet tested."

At this point you might try the method out on some sample numbers. This method is referred to as *binary search*. Note how at each stage the number of elements in the remaining set is decreased by about one half. We can now attempt a version using SPARKS pseudo code.

```
procedure BINSRCH(A,n,x,j)

initialize lower and upper

while there are more elements to check do

let A(mid) be the middle element

case

: x > A(mid): set lower to mid + 1

: x < A(mid): set upper to mid - 1

: else: found

end

end

not found

end BINSRCH
```

The above is not the only way we might write this program. For instance we could replace the **while** loop by a **repeat-until** statement with the same English condition. In fact there are at least six different binary search programs that can be produced which are all correct. There are many more that we might produce which would be incorrect. Part of the freedom comes from the initialization step. Whichever version we choose, we must be sure we understand the relationships between the variables. Below is one complete version.

```
procedure BINSRCH (A,n,x,j)

1      lower □ 1; upper □ n

2      while lower ≤ upper do
```

```
3        mid □ └(lower + upper) / 2┘

4        case

5          : x > A(mid): lower □ mid + 1

6          : x < A(mid): upper □ mid - 1

7          : else: j □ mid; return

8        end

9      end

10     j □ 0
```

end

To prove this program correct we make assertions about the relationship between variables before and after the **while** loop of steps 2-9. As we enter this loop and as long as *x* is not found the following holds:

lower ≤ upper **and** A *(lower)* ≤ x ≤ A *(upper)* **and** SORTED *(A*, n*)*

Now, if control passes out of the **while** loop past line 9 then we know the condition of line 2 is false

*lowe*r > *upper.*

This, combined with the above assertion implies that *x* is not present.

Unfortunately a complete proof takes us beyond our scope but for those who wish to pursue program proving they should consult our references at the end of this chapter. An analysis of the computing time for BINSRCH is carried out in section 7.1.

Recursion

We have tried to emphasize the need to structure a program to make it easier to achieve the goals of readability and correctness. Actually one of the most useful syntactical features for accomplishing this is the procedure. Given a set of instructions which perform a logical operation, perhaps a very complex and long operation, they can be grouped together as a procedure. The procedure name and its parameters

are viewed as a new instruction which can be used in other programs. Given the input-output specifications of a procedure, we don't even have to know how the task is accomplished, only that it is available. This view of the procedure implies that it is invoked, executed and returns control to the appropriate place in the calling procedure. What this fails to stress is the fact that procedures may call themselves (direct recursion) before they are done or they may call other procedures which again invoke the calling procedure (indirect recursion). These recursive mechanisms are extremely powerful, but even more importantly, many times they can express an otherwise complex process very clearly. For these reasons we introduce recursion here.

Most students of computer science view recursion as a somewhat mystical technique which only is useful for some very special class of problems (such as computing factorials or Ackermann's function). This is unfortunate because any program that can be written using assignment, the **if-then-else** statement and the **while** statement can also be written using assignment, **if-then-else** and recursion. Of course, this does not say that the resulting program will necessarily be easier to understand. However, there are many instances when this will be the case. When is recursion an appropriate mechanism for algorithm exposition? One instance is when the problem itself is recursively defined. Factorial fits this category, also binomial coefficients where

☐

can be recursively computed by the formula

☐

Another example is reversing a character string, $S = 'x_1 \ldots x_n'$ where SUBSTRING (S,i,j) is a function which returns the string $x_i \ldots x_j$ for appropriately defined i and j and $S \parallel T$ stands for concatenation of two strings (as in PL/I). Then the operation REVERSE is easily described recursively as

procedure *REVERSE(S)*

n ☐ *LENGTH(S)*

if *n* = 1 **then return** *(S)*

else return *(REVERSE(SUBSTRING(S,2,n))*

∥ *SUBSTRING(S,1,1))*

end *REVERSE*

If this looks too simple let us develop a more complex recursive procedure. Given a set of $n \geq 1$ elements the problem is to print all possible permutations of this set. For example if the set is $\{a,b,c\}$, then the set of permutations is $\{(a, b,c), (a,c,b), (b,a,c), (b,c,a), (c,a,b), (c,b,a)\}$. It is easy to see that given n elements there are n ! different permutations. A simple algorithm can be achieved by looking at the case of four elements (a,b,c,d). The answer is obtained by printing

(i) a followed by all permutations of (b,c,d)

(ii) b followed by all permutations of (a,c,d)

(iii) c followed by all permutations of (b,a,d)

(iv) d followed by all permutations of (b,c,a)

The expression "followed by all permutations" is the clue to recursion. It implies that we can solve the problem for a set with n elements if we had an algorithm which worked on n - 1 elements. These considerations lead to the following procedure which is invoked by **call** PERM(A,*1,n*). A is a character string e.g. A ='abcd', and INTERCHANGE (A,k,i) exchanges the k-th character of A with the i-th character of A.

```
procedure PERM(A,k,n)

if k = n then [print (A); return]

B ☐ A

for i ☐ k to n do

call INTERCHANGE(A,k,i)

call PERM(A,k + 1,n)

A ☐ B

end

end PERM
```

Try this algorithm out on sets of length one, two, and three to insure that you understand how it works. Then try to do one or more of the exercises at the end of this chapter which ask for recursive procedures.

Another time when recursion is useful is when the data structure that the algorithm is to operate on is recursively defined. We will see several important examples of such structures, especially lists in section 4.9 and binary trees in section 5.4. Another instance when recursion is invaluable is when we want to describe a backtracking procedure. But for now we will content ourselves with examining some simple, iterative programs and show how to eliminate the iteration statements and replace them by recursion. This may sound strange, but the objective is not to show that the result is simpler to understand nor more efficient to execute. The main purpose is to make one more familiar with the execution of a recursive procedure.

Suppose we start with the sorting algorithm presented in this section. To rewrite it recursively the first thing we do is to remove the **for** loops and express the algorithm using assignment, **if**-then-**else** and the **go**-to **statement.**

```
procedure SORT(A,n)

i ← 1

Ll: if i ≤ n - 1          // for i ← 1 to n - 1 do//

then [j ← i; k ← j + 1

L2: if k ≤ n        //for k ← j + 1 to n do//

then [if A(k) < A(j)

then j ← k

k ← k + 1; go to L2]

t ← A(i); A(i) ← A(j); A(j) ← t

i ← i + 1; go to L1]

end SORT
```

Now every place where we have a label we introduce a procedure whose parameters are the variables which are already assigned a value at that point. Every place where a "**go to** label" appears, we replace that statement by a call of the procedure associated with that label. This gives us the following set of three procedures.

```
procedure SORT(A,n)

call SORTL1(A,n,1)

end SORT

procedure SORTL1(A,n,i)

if i ≤ n - 1

then [j ☐ i; call MAXL2(A,n,j,i + 1)

t ☐ A(i); A(i) ☐ A(j); A(j) ☐ t

call SORTL1(A,n,i + 1)]

end SORTL1

procedure MAXL2(A,n,j,k)

if k ≤ n

then [if A(k) < A(j) then j ☐ k

call MAXL2(A,n,j,k + 1)]

end MAXL2
```

We can simplify these procedures somewhat by ignoring SORT(*A*,*n*) entirely and begin the sorting operation by **call** SORTL1(*A*,*n*,1). Notice how SORTL1 is directly recursive while it also uses procedure MAXL2. Procedure MAXL2 is also directly reculsive. These two procedures use eleven lines while the original iterative version was expressed in nine lines; not much of a difference. Notice how in MAXL2 the fourth parameter *k* is being changed. The effect of increasing *k* by one and restarting the procedure has essentially the same effect as the **for** loop.

Now let us trace the action of these procedures as they sort a set of five integers

☐

When a procedure is invoked an implicit branch to its beginning is made. Thus a recursive call of a

program can be made to simulate a **go to** statement. The parameter mechanism of the procedure is a form of assignment. Thus placing the argument $k + 1$ as the fourth parameter of MAXL2 is equivalent to the statement $k \leftarrow k + 1$.

In section 4.9 we will see the first example of a recursive data structure, the list. Also in that section are several recursive procedures, followed in some cases by their iterative equivalents. Rules are also given there for eliminating recursion.

1.4 HOW TO ANALYZE PROGRAMS

One goal of this book is to develop skills for making evaluative judgements about programs. There are many criteria upon which we can judge a program, for instance:

(i) Does it do what we want it to do?

(ii) Does it work correctly according to the original specifications of the task?

(iii) Is there documentation which describes how to use it and how it works?

(iv) Are subroutines created in such a way that they perform logical sub-functions?

(v) Is the code readable?

The above criteria are all vitally important when it comes to writing software, most especially for large systems. Though we will not be discussing how to reach these goals, we will try to achieve them throughout this book with the programs we write. Hopefully this more subtle approach will gradually infect your own program writing habits so that you will automatically strive to achieve these goals.

There are other criteria for judging programs which have a more direct relationship to performance. These have to do with computing time and storage requirements of the algorithms. Performance evaluation can be loosely divided into 2 major phases: (a) a priori estimates and (b) a posteriori testing. Both of these are equally important.

First consider a priori estimation. Suppose that somewhere in one of your programs is the statement

$x \leftarrow x + 1$.

We would like to determine two numbers for this statement. The first is the amount of time a single execution will take; the second is the number of times it is executed. The product of these numbers will be the total time taken by this statement. The second statistic is called the *frequency count,* and this may

vary from data set to data set. One of the hardest tasks in estimating frequency counts is to choose adequate samples of data. It is impossible to determine exactly how much time it takes to execute any command unless we have the following information:

(i) the machine we are executing on:

(ii) its machine language instruction set;

(iii) the time required by each machine instruction;

(iv) the translation a compiler will make from the source to the machine language.

It is possible to determine these figures by choosing a real machine and an existing compiler. Another approach would be to define a hypothetical machine (with imaginary execution times), but make the times reasonably close to those of existing hardware so that resulting figures would be representative. Neither of these alternatives seems attractive. In both cases the exact times we would determine would not apply to many machines or to any machine. Also, there would be the problem of the compiler, which could vary from machine to machine. Moreover, it is often difficult to get reliable timing figures because of clock limitations and a multi-programming or time sharing environment. Finally, the difficulty of learning another machine language outweighs the advantage of finding "exact" fictitious times. All these considerations lead us to limit our goals for an a priori analysis. Instead, we will concentrate on developing only the frequency count for all statements. The anomalies of machine configuration and language will be lumped together when we do our experimental studies. Parallelism will not be considered.

Consider the three examples of Figure 1.4 below.

```
      .                                        for i □ 1 to n do

      .            for i □ 1 to n do

      .                                        for j □ 1 to n do

x □ x + 1                x □ x + 1

      .                                              x □ x + 1

      .            end

      .                                        end
```

end

(a)   (b)   (c)

Figure 1.4: Three simple programs for frequency counting.

In program (a) we assume that the statement $x \leftarrow x + 1$ is not contained within any loop either explicit or implicit. Then its frequency count is one. In program (b) the same statement will be executed n times and in program (c) n^2 times (assuming $n \geq 1$). Now 1, n, and n^2 are said to be different and increasing orders of magnitude just like 1, 10, 100 would be if we let $n = 10$. In our analysis of execution we will be concerned chiefly with determining the order of magnitude of an algorithm. This means determining those statements which may have the greatest frequency count.

To determine the order of magnitude, formulas such as

often occur. In the program segment of figure 1.4(c) the statement $x \leftarrow x + 1$ is executed

Simple forms for the above three formulas are well known, namely,

In general

To clarify some of these ideas, let us look at a simple program for computing the n-th Fibonacci number. The Fibonacci sequence starts as

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

Each new term is obtained by taking the sum of the two previous terms. If we call the first term of the sequence F_0 then $F_0 = 0$, $F_1 = 1$ and in general

$F_n = F_{n-1} + F_{n-2}, n \geq 2$.

The program on the following page takes any non-negative integer n and prints the value F_n.

```
1     procedure FIBONACCI

2        read (n)

3-4      if n < 0 then [print ('error'); stop]

5-6      if n = 0 then [print ('0'); stop]

7-8      if n = 1 then [print ('1'); stop]

9        fnm2 ☐ 0; fnm 1 ☐ 1

10       for i ☐ 2 to n do

11           fn ☐ fnm1 + fnm2

12           fnm2 ☐ fnm1

13           fnm1 ☐ fn

14       end

15       print (fn)

16     end FIBONACCI
```

The first problem in beginning an analysis is to determine some reasonable values of *n*. A complete set would include four cases: $n < 0$, $n = 0$, $n = 1$ and $n > 1$. Below is a table which summarizes the frequency counts for the first three cases.

Step	n < 0	n = 0	n = 1
1	1	1	1
2	1	1	1
3	1	1	1

4	1	0	0
5	0	1	1
6	0	1	0
7	0	0	1
8	0	0	1
9-15	0	0	0

These three cases are not very interesting. None of them exercises the program very much. Notice, though, how each **if** statement has two parts: the **if** condition and the **then** clause. These may have different execution counts. The most interesting case for analysis comes when $n > 1$. At this point the **for** loop will actually be entered. Steps 1, 2, 3, 5, 7 and 9 will be executed once, but steps 4, 6 and 8 not at all. Both commands in step 9 are executed once. Now, for $n \geq 2$ how often is step 10 executed: not n - 1 but n times. Though 2 to n is only n - 1 executions, remember that there will be a last return to step 10 where i is incremented to $n + 1$, the test $i > n$ made and the branch taken to step 15. Thus, steps 11, 12, 13 and 14 will be executed n - 1 times but step 10 will be done n times. We can summarize all of this with a table.

Step	Frequency	Step	Frequency
1	1	9	2
2	1	10	n
3	1	11	n-1
4	0	12	n-1
5	1	13	n-1
6	0	14	n-1
7	1	15	1

```
8          0          16          1
```

Figure 1.5: Execution Count for Computing F$_n$

Each statement is counted once, so step 9 has 2 statements and is executed once for a total of 2. Clearly, the actual time taken by each statement will vary. The **for** statement is really a combination of several statements, but we will count it as one. The total count then is $5n + 5$. We will often write this as $O(n)$, ignoring the two constants 5. This notation means that the order of magnitude is proportional to n.

The notation $f(n) = O(g(n))$ (read as f of n equals big-oh of g of n) has a precise mathematical definition.

Definition: $f(n) = O(g(n))$ iff there exist two constants c and n_o such that $|f(n)| \leq c|g(n)|$ for all $n \geq n_o$.

$f(n)$ will normally represent the computing time of some algorithm. When we say that the computing time of an algorithm is $O(g(n))$ we mean that its execution takes no more than a constant times $g(n)$. n is a parameter which characterizes the inputs and/or outputs. For example n might be the number of inputs or the number of outputs or their sum or the magnitude of one of them. For the Fibonacci program n represents the magnitude of the input and the time for this program is written as $T(\text{FIBONACCI}) = O(n)$.

We write $O(1)$ to mean a computing time which is a constant. $O(n)$ is called linear, $O(n^2)$ is called quadratic, $O(n^3)$ is called cubic, and $O(2^n)$ is called exponential. If an algorithm takes time $O(\log n)$ it is faster, for sufficiently large n, than if it had taken $O(n)$. Similarly, $O(n \log n)$ is better than $O(n^2)$ but not as good as $O(n)$. These seven computing times, $O(1)$, $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$, $O(n^3)$, and $O(2^n)$ are the ones we will see most often throughout the book.

If we have two algorithms which perform the same task, and the first has a computing time which is $O(n)$ and the second $O(n^2)$, then we will usually take the first as superior. The reason for this is that as n increases the time for the second algorithm will get far worse than the time for the first. For example, if the constant for algorithms one and two are 10 and 1/2 respectively, then we get the following table of computing times:

```
 n   10n     n²/2

----------------

 1    10     1/2

 5    50     12-1/2

10   100      50
```

```
15   150   112-1/2

20   200      200

25   250   312-1/2

30   300      450
```

For $n \leq 20$, algorithm two had a smaller computing time but once past that point algorithm one became better. This shows why we choose the algorithm with the smaller order of magnitude, but we emphasize that this is not the whole story. For small data sets, the respective constants must be carefully determined. In practice these constants depend on many factors, such as the language and the machine one is using. Thus, we will usually postpone the establishment of the constant until after the program has been written. Then a performance profile can be gathered using real time calculation.

Figures 1.6 and 1.7 show how the computing times (counts) grow with a constant equal to one. Notice how the times $O(n)$ and $O(n \log n)$ grow much more slowly than the others. For large data sets, algorithms with a complexity greater than $O(n \log n)$ are often impractical. An algorithm which is exponential will work only for very small inputs. For exponential algorithms, even if we improve the constant, say by 1/2 or 1/3, we will not improve the amount of data we can handle by very much.

Given an algorithm, we analyze the frequency count of each statement and total the sum. This may give a polynomial

$$P(n) = c_k n^k + c_{k-1} n^{k-1} + \ldots + c_1 n + c_o$$

where the c_i are constants, $c_k \neq 0$ and n is a parameter. Using big-oh notation, $P(n) = O(n^k)$. On the other hand, if any step is executed 2^n times or more the expression

$$c2^n + P(n) = O(2^n).$$

Another valid performance measure of an algorithm is the space it requires. Often one can trade space for time, getting a faster algorithm but using more space. We will see cases of this in subsequent chapters.

☐

Figure 1.6: Rate of Growth of Common Computing Time Functions

```
log₂n   n   nlog₂n   n²   n³   2ⁿ
```
$\log_2 n \quad n \quad n\log_2 n \quad n^2 \quad n^3 \quad 2^n$

```
-------------------------------------------------------

   0      1      0       1      1       2

   1      2      2       4      8       4

   2      4      8      16     64      16

   3      8     24      64    512     256

   4     16     64     256   4096   65536

   5     32    160    1024  32768   2, 147, 483, 648
```

Figure 1.7: Values for Computing Functions

We end this chapter with a problem from recreational mathematics which uses many of the SPARKS features that have been discussed. A magic square is an n x n matrix of the integers 1 to n^2 such that the sum of every row, column and diagonal is the same. For example, if n = 5 we have

```
15    8    1   24   17

16   14    7    5   23

22   20   13    6    4

 3   21   19   12   10

 9    2   25   18   11
```

where the common sum is 65. When n is odd *H*. Coxeter has given a simple rule for generating a magic square:

"Start with 1 in the middle of the top row; then go up and left assigning numbers in increasing order to empty squares; if you fall off the square imagine the same square as tiling the plane and continue; if a square is occupied, move down instead and continue."

The magic square above was formed using this rule. We now write a SPARKS program for creating an n X n magic square for n odd.

```
procedure MAGIC(square, n)

//for n odd create a magic square which is declared as an array//

//square (0: n - 1, 0: n - 1)//

//(i,j) is a square position. 2 ≤ key ≤ n² is integer valued.//

if n is even then [print ('input error'); stop]

SQUARE ☐ 0

square (0,(n - 1)/2) ☐ 1;          //store 1 in middle of first row//

key ☐ 2; i ☐ 0; j ☐ (n - 1)/2        //i,j are current position//

while key ≤ n² do

(k,l) ☐ ((i - 1) mod n, (j - 1) mod n)         //look up and left//

if square (k,l) ≠ 0

then i ☐ (i + 1) mod n         //square occupied, move down//

else (i,j) ☐ (k,l)    //square (k,l) needs to be assigned//

square (i,j) ☐ key       //assign it a value//

key ☐ key + 1

end

print (n, square)      //output result//

end MAGIC
```

MAGIC is a complete SPARKS procedure. The statement (i,j) ☐ (k,l) is a shorthand way of writing i ☐ $k; j$ ☐ l. It emphasizes that the variables are thought of as pairs and are changed as a unit. The

reserved word **mod** computes the nonnegative remainder and is a built in function. The magic square is represented using a two dimensional array having n rows and n column. For this application it is convenient to number the rows (and columns) from zero to $n - 1$ rather than from one to n. Thus, when the program "falls off the square" the **mod** operator sets i and/or j back to zero or $n - 1$.

The **while** loop is governed by the variable *key* which is an integer variable initialized to 2 and increased by one each time through the loop. Thus each statement within the **while** loop will be executed no more than $n^2 - 1$ times and hence the computing time for MAGIC is $O(n^2)$. Since there are n^2 positions in which the algorithm must place a number, we see that $O(n^2)$ is the best bound an algorithm could have.

REFERENCES

For a discussion of algorithms and how to analyze them see

The Art of Computer Programming: Fundamental Algorithms, by D. E. Knuth, vol. 1, chapter 1, 2-nd edition, Addison-Wesley, 1973.

For a discussion of good programming techniques see

Structured Programming by O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, Academic Press, 1972.

The Elements of Programming Style by B. W. Kernighan and P. J. Plauger, McGraw-Hill, 1974.

ACM Computing Surveys, Special Issue: Programming, vol. 6, no. 4, December, 1974.

For a discussion of tools and procedures for developing very large software systems see

Practical Strategies for Developing Large Software Systems, by E. Horowitz, Addison-Wesley, May, 1975.

For a discussion of the more abstract formulation of data structures see

"Toward an understanding of data structures" by J. Earley, *CACM*, vol. 14, no. 10, October, 1971, pp. 617-627.

"Another look at data," by G. Mealy, *Proc. AFIPS Fall Joint Computer Conference,* vol. 31, 1967, pp. 525-534.

For a further discussion of program proving see

"Assigning meaning to programs," by R. W. Floyd, *Proc. of a Symposium in Applied Mathematics,* vol. 19, J. T. Schwartz, ed., American Mathematical Society, Providence, 1967, pp. 19-32.

"An interactive program verification system," by D. I. Good, R. L. London, W. W. Bledsoe, *IEEE Transactions on Software Engineering*, SE-1, vol. 1, March, 1975, pp. 59-67.

EXERCISES

1. Look up the word *algorithm* or its older form *algorism* in the dictionary.

2. Consider the two statements: (i) Is $n = 2$ the largest value of n for which there exists positive integers x, y and z such that $x^n + y^n = z^n$ has a solution; (ii) Store 5 divided by zero into X and go to statement 10. Both do not satisfy one of the five criteria of an algorithm. Which criteria do they violate?

3. Describe the flowchart in figure 1.1 by using a combination of SPARKS and English. Can you do this without using the **go to**? Now make it into an algorithm.

4. Discuss how you would actually represent the list of name and telephone number pairs in a real machine. How would you handle people with the same last name.

5. Write FORTRAN equivalents of the **while, repeat-until, loop-forever** and **for** statements of SPARKS.

6. Can you think of a clever meaning for S.P.A.R.K.S.? Concentrate on the letter *K* first.

7. Determine the frequency counts for all statements in the following two SPARKS program segments:

```
1 for i □ 1 to n           1 i □ 1

2   for j □ 1 to i         2 while i ≤ n do

3     for k □ 1 to j       3   x □ x + 1

4       x □ x + 1          4   i □ i +1

5     end                  5 end

6   end
```

7 **end**

    (a)   (b)

8. Horner's Rule is a means for evaluating a polynomial $A(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$ at a point x_0 using a minimum number of multiplications. The rule is:

$$A(x) = (\ldots ((a_n . x_0 + a_{n-1}) x_0 + \ldots + a_1) x_0 + a_0$$

Write a SPARKS program to evaluate a polynomial using Horner's Rule. Determine how many times each statement is executed.

9. Given n boolean variables $x_1, \ldots, x_n$ we wish to print all possible combinations of truth values they can assume. For instance, if $n = 2$, there are four possibilities: true, true; true, false; false, true; false, false. Write a SPARKS program to accomplish this and do a frequency count.

10. Compare the two functions n^2 and $2^n/4$ for various values of n. Determine when the second becomes larger than the first.

11. Write a SPARKS program which prints out the integer values of x, y, z in nondecreasing order. What is the computing time of your method?

12. Write a SPARKS procedure which searches an array $A(1:n)$ for the element x. If x occurs, then set j to its position in the array else set j to zero. Try writing this without using the **go to** statement.

13. One useful facility we might add to SPARKS is the ability to manipulate character strings. If x, y are variables of type character, then we might like to implement the procedures:

(i) $z \square$ CONCAT(x,y) which concatenates a copy of string y to the end of a copy of string x and assigns the resulting string to z. Strings x and y remain unchanged.

(ii) $z \square$ SUBSTR(x,i,j) which copies to z the i-th to the j-th character in string x with appropriate definitions for $j = 0$, $i > j$, etc. String x is unchanged.

(iii) $z \square$ INDEX(x,y) which searches string x for the first occurrence of string y and sets z to its starting position in x or else zero.

Implement these procedures using the array facility.

14. Write a SPARKS procedure which is given an argument STRING, whose value is a character string

of length *n*. Copy STRING into the variable FILE so that every sequence of blanks is reduced to a single blank. The last character of STRING is nonblank.

15. Design a program that counts the number of occurrences of each character in the string STRING of length *n*. Represent your answer in the array ANS(1:*k*,1:2) where ANS(*i*,l) is the *i*-th character and ANS(*i*,2) is the number of times it occurs in STRING.

16. Trace the action of the procedure below on the elements 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 searching for l, 3, 13 and 21.

```
i □ 1; j □ n

repeat k □ (i + j)/2

if A(k) ≤ x then i □ k + 1

else j □ k - 1

until i > j
```

What is the computing time for this segment in terms of *n*?

17. Prove by induction:

□

□

□

18. List as many rules of style in programming that you can think of that you would be willing to follow yourself.

19. Using the notation introduced at the end of section 1.1, define the structure Boolean with operations AND, OR, NOT, IMP and EQV (equivalent) using only the **if-then-else** statement. e.g. NOT (*X*) :: = **if *X* then false else true.**

20. Give a version of a binary search procedure which initializes lower to zero and upper to *n* + l.

21. Take any version of binary search, express it using assignment, **if-then-else** and **go to** and then give

an equivalent recursive program.

22. Analyze the computing time of procedure SORT as given in section 1.3.

23. Write a recursive procedure for computing the binomial coefficient ▢ as defined in section 1.3 where ▢. Analyze the time and space requirements of your algorithm.

24. Ackermann's function $A(m,n)$ is defined as follows:

▢

This function is studied because it grows very fast for small values of m and n. Write a recursive procedure for computing this function. Then write a nonrecursive algorithm for computing Ackermann's function.

25. (Tower of Hanoi) There are three towers and sixty four disks of different diameters placed on the first tower. The disks are in order of decreasing diameter as one scans up the tower. Monks were reputedly supposed to move the disks from tower 1 to tower 3 obeying the rules: (i) only one disk can be moved at any time; (ii) no disk can be placed on top of a disk with smaller diameter. Write a recursive procedure which prints the sequence of moves which accomplish this task.

26. Write an equivalent recursive version of procedure MAGIC as given in section 1.4.

27. The *pigeon hole principle* states that if a function f has n distinct inputs but less than n distinct outputs then there exists two inputs a, b such that $a \neq b$ and $f(a) = f(b)$. Give an algorithm which finds the values a, b for which the range values are equal.

28. Given n, a positive integer determine if n is the sum of all of its divisors; i.e. if n is the sum of all t such that $1 \leq t < n$ and t divides n.

29. Consider the function $F(x)$ defined by

$F(x)$ ▢ **if** *even(x)* **then** $x/2$ **else** $F(F(3x + 1))$

Prove that $F(x)$ terminates for all integers x. (Hint: consider integers of the form $(2i + 1) 2^k - 1$ and use induction.)

30. If S is a set of n elements the *powerset* of S is the set of all possible subsets of S. For example if $S = (a,b,c,)$ then POWERSET(S) = {( ), (a), (b), (c), (a,b), (a,c), (b,c), (a,b,c)}. Write a recursive procedure to compute powerset (S).

◀   ■   ▶

CHAPTER 2: ARRAYS

2.1 AXIOMATIZATION

It is appropriate that we begin our study of data structures with the array. The array is often the only means for structuring data which is provided in a programming language. Therefore it deserves a significant amount of attention. If one asks a group of programmers to define an array, the most often quoted saying is: a *consecutive set of memory locations.* This is unfortunate because it clearly reveals a common point of confusion, namely the distinction between a data structure and its representation. It is true that arrays are almost always implemented by using consecutive memory, but not always. Intuitively, an array is a set of pairs, index and value. For each index which is defined, there is a value associated with that index. In mathematical terms we call this a correspondence or a mapping. However, as computer scientists we want to provide a more functional definition by giving the operations which are permitted on this data structure. For arrays this means we are concerned with only two operations which retrieve and store values. Using our notation this object can be defined as:

structure *ARRAY(value, index)*

declare *CREATE( )* □ *array*

RETRIEVE(array,index) □ *value*

STORE(array,index,value) □ *array;*

for all *A* ∈ *array, i,j* ∈ *index, x* ∈ *value* **let**

RETRIEVE(CREATE,i) :: = **error**

RETRIEVE(STORE(A,i,x),j) :: =

if *EQUAL(i,j)* **then** *x* **else** *RETRIEVE(A,j)*

end

end *ARRAY*

The function CREATE produces a new, empty array. RETRIEVE takes as input an array and an index, and either returns the appropriate value or an error. STORE is used to enter new index-value pairs. The

second axiom is read as "to retrieve the *j*-th item where *x* has already been stored at index *i* in *A* is equivalent to checking if *i* and *j* are equal and if so, *x,* or search for the *j*-th value in the remaining array, *A*." This axiom was originally given by J. McCarthy. Notice how the axioms are independent of any representation scheme. Also, *i* and *j* need not necessarily be integers, but we assume only that an EQUAL function can be devised.

If we restrict the index values to be integers, then assuming a conventional random access memory we can implement STORE and RETRIEVE so that they operate in a constant amount of time. If we interpret the indices to be *n*-dimensional, $(i_1, i_2, ..., i_n)$, then the previous axioms apply immediately and define *n*-dimensional arrays. In section 2.4 we will examine how to implement RETRIEVE and STORE for multi-dimensional arrays using consecutive memory locations.

2.2 ORDERED LISTS

One of the simplest and most commonly found data object is the ordered or linear list. Examples are the days of the week

(MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,

SATURDAY, SUNDAY)

or the values in a card deck

(2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King, Ace)

or the floors of a building

(basement, lobby, mezzanine, first, second, third)

or the years the United States fought in World War II

(1941, 1942, 1943, 1944, 1945).

If we consider an ordered list more abstractly, we say that it is either empty or it can be written as

$(a_1, a_2, a_3, ..., a_n)$

where the a_i are atoms from some set *S*.

There are a variety of operations that are performed on these lists. These operations include:

(i) find the length of the list, n;

(ii) read the list from left-to-right (or right-to-left);

(iii) retrieve the i-th element, $\boxed{}$;

(iv) store a new value into the i-th position, $\boxed{}$;

(v) insert a new element at position $\boxed{}$ causing elements numbered $i, i + 1, ..., n$ to become numbered $i + 1, i + 2, ..., n + 1$;

(vi) delete the element at position $\boxed{}$ causing elements numbered $i + 1, ..., n$ to become numbered $i, i + 1, ..., n - 1$.

See exercise 24 for a set of axioms which uses these operations to abstractly define an ordered list. It is not always necessary to be able to perform all of these operations; many times a subset will suffice. In the study of data structures we are interested in ways of representing ordered lists so that these operations can be carried out efficiently.

Perhaps the most common way to represent an ordered list is by an array where we associate the list element a_i with the array index i. This we will refer to as a *sequential mapping,* because using the conventional array representation we are storing a_i and $a_{i + 1}$ into consecutive locations i and $i + 1$ of the array. This gives us the ability to retrieve or modify the values of random elements in the list in a constant amount of time, essentially because a computer memory has random access to any word. We can access the list element values in either direction by changing the subscript values in a controlled way. It is only operations (v) and (vi) which require real effort. Insertion and deletion using sequential allocation forces us to move some of the remaining elements so the sequential mapping is preserved in its proper form. It is precisely this overhead which leads us to consider nonsequential mappings of ordered lists into arrays in Chapter 4.

Let us jump right into a problem requiring ordered lists which we will solve by using one dimensional arrays. This problem has become the classical example for motivating the use of list processing techniques which we will see in later chapters. Therefore, it makes sense to look at the problem and see why arrays offer only a partially adequate solution. The problem calls for building a set of subroutines which allow for the manipulation of symbolic polynomials. By "symbolic," we mean the list of coefficients and exponents which accompany a polynomial, e.g. two such polynomials are

$A(x) = 3x^2 + 2x + 4$ and $B(x) = x^4 + 10x^3 + 3x^2 + 1$

For a start, the capabilities we would include are the four basic arithmetic operations: addition, subtraction, multiplication, and division. We will also need input and output routines and some suitable format for preparing polynomials as input. The first step is to consider how to define polynomials as a computer structure. For a mathematician a polynomial is a sum of terms where each term has the form ax^e; x is the variable, a is the coefficient and e is the exponent. However this is not an appropriate definition for our purposes. When defining a data object one must decide what functions will be available, what their input is, what their output is and exactly what it is that they do. A complete specification of the data structure polynomial is now given.

structure *POLYNOMIAL*

declare *ZERO( )* $\Box$ *poly; ISZERO(poly)* $\Box$ *Boolean*

COEF(poly,exp) $\Box$ *coef;*

ATTACH(poly,coef,exp) $\Box$ *poly*

REM(poly,exp) $\Box$ *poly*

SMULT(poly,coef,exp) $\Box$ *poly*

ADD(poly,poly) $\Box$ *poly; MULT(poly,poly)* $\Box$ *poly;*

for all *P,Q,* $\in$ *poly c,d,* $\in$ *coef e,f* $\in$ *exp* **let**

REM(ZERO,f) :: = ZERO

REM(ATTACH(P,c,e),f) :: =

if *e = f* **then** *REM(P,f)* **else** *ATTACH(REM(P,f),c,e)*

ISZERO(ZERO) :: = **true**

ISZERO(ATTACH(P,c,e)):: =

if *COEF(P,e) = - c* **then** *ISZERO(REM(P,e))* **else false**

COEF(ZERO,e) :: = 0

COEF(ATTACH(P,c,e),f) :: =

```
if e = f then c + COEF(P,f) else COEF(P,f)

SMULT(ZERO,d,f) :: = ZERO

SMULT(ATTACH(P,c,e),d,f) :: =

ATTACH(SMULT(P,d,f),c    d,e + f)

ADD(P,ZERO):: = P

ADD(P,ATTACH(Q,d,f)) :: = ATTACH(ADD(P,Q),d,f)

MULT(P,ZERO) :: = ZERO

MULT(P,ATTACH(Q,d,f)) :: =

ADD(MULT(P,Q),SMULT(P,d,f))
```

end

end *POLYNOMIAL*

In this specification every polynomial is either ZERO or constructed by applying ATTACH to a polynomial. For example the polynomial $P = 10x - 12x^3 - 10x + 0x^2$ is represented by the string

ATTACH(ATTACH(ATTACH(ATTACH(ZERO, 10,1),-12,3),

- 10,1),0,2).

Notice the absense of any assumptions about the order of exponents, about nonzero coefficients, etc. These assumptions are decisions of representation. Suppose we wish to remove from *P* those terms having exponent one. Then we would write REM(*P,* 1) and by the axioms the above string would be transformed into

ATTACH(REM(ATTACH(ATTACH(ATTACH(ZERO,10,1), - 12,3),

- 10,1),1),0,2)

which is transformed into

ATTACH(REM(ATTACH(ATTACH(ZERO,10,1), - 12,3),1),0,2)

which becomes

ATTACH(ATTACH(REM(ATTACH(ZERO,10,1),1) - 12,3),0,2)

which becomes

ATTACH(ATTACH(REM(ZERO,1), - 12,3),0,2)

which becomes finally

ATTACH(ATTACH(ZERO, - 12,3),0,2)

or $- 12x^3 + 0x^2$.

These axioms are valuable in that they describe the meaning of each operation concisely and without implying an implementation. Note how trivial the addition and multiplication operations have become.

Now we can make some representation decisions. Exponents should be unique and in decreasing order is a very reasonable first decision. This considerably simplifies the operations ISZERO, COEF and REM while ADD, SMULT and MULT remain unchanged. Now assuming a new function EXP (poly) ☐ exp which returns the leading exponent of poly, we can write a version of ADD which is expressed more like program, but is still representation independent.

//C = A + B where A,B are the input polynomials//

C ☐ *ZERO*

while not *ISZERO(A)* **and not** *ISZERO(B)* **do**

case

: *EXP(A) < EXP(B):*

C ☐ *ATTACH(C,COEF(B,EXP(B)),EXP(B))*

B ☐ *REM(B,EXP(B))*

`:EXP(A) = EXP(B):`

`C ☐ ATTACH(C,COEF(A,EXP(A))+COEF(B,EXP(B)),EXP(A))`

`A ☐ REM(A,EXP(A)); B ☐ REM(B,EXP(B))`

`:EXP(A) > EXP(B):`

`C ☐ ATTACH(C,COEF(A,EXP(A)),EXP(A))`

`A ☐ REM(A,EXP(A))`

end

end

insert any remaining terms in A or B into C

The basic loop of this algorithm consists of merging the terms of the two polynomials, depending upon the result of comparing the exponents. The **case** statement determines how the exponents are related and performs the proper action. Since the tests within the **case** statement require two terms, if one polynomial gets exhausted we must exit and the remaining terms of the other can be copied directly into the result. With these insights, suppose we now consider the representation question more carefully.

A general polynomial $A(x)$ can be written as

$$a_n x^n + a_{n-1} x^{n-1} + ... + a_1 x + a_0$$

where $a_n \neq 0$ and we say that the degree of A is n. Then we can represent $A(x)$ as an ordered list of coefficients using a one dimensional array of length $n + 2$,

$$A = (n, a_n, a_{n-1}, ..., a_1, a_0).$$

The first element is the degree of A followed by the $n + 1$ coefficients in order of decreasing exponent. This representation leads to very simple algorithms for addition and multiplication. We have avoided the need to explicitly store the exponent of each term and instead we can deduce its value by knowing our position in the list and the degree.

But are there any disadvantages to this representation? Hopefully you have already guessed the worst one, which is the large amount of wasted storage for certain polynomials. Consider $x^{1000} + 1$, for

instance. It will require a vector of length 1002, while 999 of those values will be zero. Therefore, we are led to consider an alternative scheme.

Suppose we take the polynomial $A(x)$ above and keep only its nonzero coefficients. Then we will really have the polynomial

$$b_{m-1} x^{em-1} + b_{m-2}x^{em-2} + ... + b_o x^{eo}$$

(1)

where each b_i is a nonzero coefficient of A and the exponents e_i are decreasing ☐. If all of A's coefficients are nonzero, then $m = n + 1$, $e_i = i$, and $b_i = a_i$ for ☐. Alternatively, only a_n may be nonzero, in which case $m = 1$, $b_o = a_n$, and $e_0 = n$. In general, the polynomial in (1) could be represented by the ordered list of length $2m + 1$,

$$(m, e_{m-1}, b_{m-1}, e_{m-2}, b_{m-2}, ..., e_0, b_0).$$

The first entry is the number of nonzero terms. Then for each term there are two entries representing an exponent-coefficient pair.

Is this method any better than the first scheme? Well, it certainly solves our problem with $x^{1000} + 1$, which now would be represented as (2,1000,1,0,1). Basic algorithms will need to be more complex because we must check each exponent before we handle its coefficient, but this is not too serious. As for storage, this scheme could be worse than the former. For example, $x^4 + 10x^3 + 3x^2 + 1$ would have the two forms

(4,1,10,3,0,1) or (4,4,1,3,10,2,3,0,1).

In the worst case, scheme 2 requires less than twice as much storage as scheme 1 (when the degree $= n$ and all $n + 1$ coefficients are nonzero). But scheme 1 could be much more wasteful, as in the case of $x^{1000} + 1$, where it needs more than 200 times as many locations. Therefore, we will prefer representation scheme 2 and use it.

Let us now write a procedure in SPARKS for adding two polynomials represented as in scheme 2.

```
procedure PADD(A,B,C)

//A(1:2m + 1), B(1:2n + 1), C(1:2(m + n) + 1)//
```

```
1     m ← A(1); n ← B(1)

2     p ← q ← r ← 2

3     while p ≤ 2m and q ≤ 2n do

4         case            //compare exponents//

:A(p) = B(q): C(r + 1) ← A(p + 1) + B(q + 1)

//add coefficients//

if C(r + 1) ≠ 0

then [C(r) ← A(p); r ← r + 2]

//store exponent//

p ← p + 2; q ← q + 2     //advance to next

terms//

:A(p) < B(q): C(r + 1) ← B(q + 1); C(r) ← B(q)

//store new term//

q ← q + 2; r ← r + 2      //advance to next

term//

:A(p) > B(q): C(r + 1) ← A(p + 1); C(r) ← A(p)

//store new term//

p ← p + 2; r ← r + 2    //advance to next

term//

end
```

```
end

5      while p ← 2m do   //copy remaining terms of A//

C(r) ← A(p); C(r + 1) ← A(p + 1)

p ← p + 2 ; r ← r + 2

end

6      while q ← 2n do   //copy remaining terms of B//

C(r) ← B(q); C(r + 1) ← B(q + 1)

q ← q + 2; r ← r + 2

end

7      C(1) ← r/2 - 1           //number of terms in the sum//

end PADD
```

As this is one of our first complex algorithms written in SPARKS, suppose we point out some features. The procedure has parameters which are polynomial (or array) names, and hence they are capitalized. Three pointers (*p,q,r*) are used to designate a term in *A, B,* or *C*.

Comments appear to the right delimited by double slashes. The basic iteration step is governed by a **while** loop. Blocks of statements are grouped together using square brackets. Notice how closely the actual program matches with the original design. The code is indented to reinforce readability and to reveal more clearly the scope of reserved words. This is a practice you should adopt in your own coding. Statement two is a shorthand way of writing

$r ← 2; q ← r; p ← q$

Let us now analyze the computing time of this algorithm. It is natural to carry out this analysis in terms of *m* and *n*, the number of nonzero terms in A and B respectively. The assignments of lines 1 and 2 are made only once and hence contribute $O(1)$ to the overall computing time. If either $n = 0$ or $m = 0$, the **while** loop of line 3 is not executed.

In case neither *m* nor *n* equals zero, the **while** loop of line 3 is entered. Each iteration of this **while** loop requires $O(1)$ time. At each iteration, either the value of *p* or *q* or both increases by 2. Since the iteration terminates when either *p* or *q* exceeds 2*m* or 2*n* respectively, the number of iterations is bounded by *m* + *n* - 1. This worst case is achieved, for instance, when $A(x) = \sum^n_{i=0} x^{2^i}$ and $B(x) = \sum^n_{i=0} x^{2^i+1}$ Since none of the exponents are the same in *A* and *B*, $A(p) \neq B(q)$. Consequently, on each iteration the value of only one of *p* or *q* increases by 2. So, the worst case computing time for this **while** loop is $O(n + m)$. The total computing time for the **while** loops of lines 5 and 6 is bounded by $O(n + m)$, as the first cannot be iterated more than *m* times and the second more than *n* . Taking the sum of all of these steps, we obtain $O(n + m)$ as the asymptotic computing time of this algorithm.

This example shows the array as a useful representational form for ordered lists. Returning to the abstract object--the ordered list--for a moment, suppose we generalize our problem and say that it is now required to represent a variable number of lists where the size of each may vary. In particular we now have the *m* lists

$$(a_{11}, a_{12}, ..., a_{1_{n1}}), (a_{2l}, a_{22}, ..., a_{2_{n2}}), ..., (a_{m1}, a_{m2}, ..., a_{mn_m})$$

where n_i, the size of the *i*-th list, is an integer greater than or equal to zero.

A two dimensional array could be a poor way to represent these lists because we would have to declare it as $A(m, \max\{n_1, ..., n_m\})$, which might be very wasteful of space. Instead we might store them in a one dimensional array and include a front(*i*) and rear(*i*) pointer for the beginning and end of each list. This only requires $2m + n_1 + n_2 + ... + n_m$ locations rather than *m* times $\max\{n_1, ..., n_m\}$ locations. But the one dimensional array presents problems when we try to insert an item in list *i* and there is no more room unless we move the elements of list *i* + 1 and perhaps list *i* + 2, ...,list *m* to the right.

To make this problem more concrete, let us return to the ordered list of polynomials represented using the second scheme. Suppose in addition to PADD, we have also written procedures which subtract, multiply, and divide two polynomials: PSUB, PMUL, and PDIV. We are making these four procedures available to any user who wants to manipulate polynomials. This hypothetical user may have many polynomials he wants to compute and he may not know their sizes.

He would include these subroutines along with a main procedure he writes himself. In this main program he needs to declare arrays for all of his polynomials (which is reasonable) and to declare the maximum size that every polynomial might achieve (which is harder and less reasonable). If he declares the arrays too large, much of that space will be wasted. Consider the main routine our mythical user might write if he wanted to compute the Fibonacci polynomials. These are defined by the recurrence relation

where $F_0(x) = 1$ and $F_1(x) = x$. For example

$$F_2(x) = x \boxed{} F_1(x) + F_0(x) = x^2 + 1.$$

Suppose the programmer decides to use a two dimensional array to store the Fibonacci polynomials, the exponents and coefficients of $F_i(x)$ being stored in the i-th row. For example $F(2,*) = (2,2,1,0,1)$ implies $F(2,1) = 2, F(2,2) = 2, F(2,3) = 1, F(2,4) = 0, F(2,5) = 1$ and is the polynomial $x^2 + 1$. Then the following program is produced.

```
procedure MAIN

declare F(0:100,203),TEMP(203)

read (n)

if n > 100 then [print ('n too large') stop]

F(0,*) ☐ (1,0,1)          //set F₀ = 1x⁰//

F(1,*) ☐ (1,1,1)          //set F₁ = 1x¹//

for i ☐ 2 to n do

call PMUL(F(1,1),F(i - 1,1),TEMP(1))     //TEMP=x ☐

Fᵢ₋(x)//

call PADD(TEMP(1),F(i - 2,1),F(i,1))     //Fᵢ = TEMP +

Fᵢ₋₂//

//TEMP is no longer needed//

end

for i ☐ 0 to n do

call PPRINT(F(i,1))        //polynomial print routine//
```

end

end *MAIN*

The author of this procedure has declared $101 * 203 = 20,503$ locations to hold the Fibonacci polynomials, which is about twice as much as is actually needed. A much greater saving could be achieved if $F_i(x)$ were printed as soon as it was computed in the first loop. Then by storing all polynomials in a single array, 1000 locations would be more than adequate. However, storing several polynomials in a single array alters the way one creates new polynomials and complicates matters if they are to be destroyed and the space reused.

This example reveals other limitations of the array as a means for data representation. The array is usually a homogeneous collection of data which will not allow us to intermix data of different types. Exponents and coefficients are really different sorts of numbers, exponents usually being small, non-negative integers whereas coefficients may be positive or negative, integer or rational, possibly double, triple or multiple precision integers or even other polynomials. Different types of data cannot be accommodated within the usual array concept. Ideally, we would like a representation which would:

(i) require the programmer only to name his polynomial variables and declare one maximum size for the entire working space;

(ii) provide a system which would automatically maintain all polynomials until the number of terms exceeds the work space;

(iii) allow the programmer to use different representations for different parts of the data object.

Let's pursue the idea of storing all polynomials in a single array called POLY. Let the polynomials be

$A(x) = 2x + 3, B(x) = x^2 + 5x + 3, C(x) = 3x^{10} + 9x^4$

POLY: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 ... max

Then the name of a polynomial is never the array POLY, but a simple variable whose value is a pointer into the place in POLY where it begins. For instance, in the above case we might have $A = 1$, $B = 6$, and $C = 13$. Also we need a pointer to tell us where the next free location is, as above where *free* = 18.

If we made a call to our addition routine, say PADD (A,B,D), then it would take the polynomials starting at POLY(A) and POLY(B) and store the result starting at POLY$(free)$. If the result has k terms, then D

 free and *free* *free* + 2*k* + 1. Now we have localized all storage to one array. As we create polynomials, *free* is continually incremented until it tries to exceed *max*. When this happens must we quit? We must unless there are some polynomials which are no longer needed. There may be several such polynomials whose space can be reused. We could write a subroutine which would compact the remaining polynomials, leaving a large, consecutive free space at one end. But this may require much data movement. Even worse, if we move a polynomial we must change its pointer. This demands a sophisticated compacting routine coupled with a disciplined use of names for polynomials. In Chapter 4 we will see an elegant solution to these problems.

2.3 SPARSE MATRICES

A matrix is a mathematical object which arises in many physical problems. As computer scientists, we are interested in studying ways to represent matrices so that the operations to be performed on them can be carried out efficiently. A general matrix consists of *m* rows and *n* columns of numbers as in figure 2.1.

Figure 2.1: Example of 2 matrices

The first matrix has five rows and three columns, the second six rows and six columns. In general, we write *m* X *n* (read *m* by *n*) to designate a matrix with *m* rows and *n* columns. Such a matrix has *mn* elements. When *m* is equal to *n*, we call the matrix *square*.

It is very natural to store a matrix in a two dimensional array, say $A(1:m, 1:n)$. Then we can work with any element by writing $A(i,j)$; and this element can be found very quickly, as we will see in the next section. Now if we look at the second matrix of figure 2.1, we see that it has many zero entries. Such a matrix is called *sparse*. There is no precise definition of when a matrix is sparse and when it is not, but it is a concept which we can all recognize intuitively. Above, only 8 out of 36 possible elements are nonzero and that is sparse! A sparse matrix requires us to consider an alternate form of representation. This comes about because in practice many of the matrices we want to deal with are large, e.g., 1000 X 1000, but at the same time they are sparse: say only 1000 out of one million possible elements are nonzero. On most computers today it would be impossible to store a full 1000 X 1000 matrix in the memory at once. Therefore, we ask for an alternative representation for sparse matrices. The alternative representation will explicitly store only the nonzero elements.

Each element of a matrix is uniquely characterized by its row and column position, say *i,j*. We might then store a matrix as a list of 3-tuples of the form

(*i,j*,value).

Also it might be helpful to organize this list of 3-tuples in some way, perhaps placing them so that the row numbers are increasing. We can go one step farther and require that all the 3-tuples of any row be stored so that the columns are increasing. Thus, we might store the second matrix of figure 2.1 in the array $A(0:t,1:3)$ where $t = 8$ is the number of nonzero terms.

```
        1,   2,    3

        -----------

A(0,   6,   6,     8

  (1,   1,   1,    15

  (2,   1,   4,    22

  (3,   1,   6,   -15

  (4,   2,   2,    11

  (5,   2,   3,     3

  (6,   3,   4,    -6

  (7,   5,   1,    91

  (8,   6,   3,    28
```

Figure 2.2: Sparse matrix stored as triples

The elements $A(0,1)$ and $A(0,2)$ contain the number of rows and columns of the matrix. $A(0,3)$ contains the number of nonzero terms.

Now what are some of the operations we might want to perform on these matrices? One operation is to compute the transpose matrix. This is where we move the elements so that the element in the i,j position gets put in the j,i position. Another way of saying this is that we are interchanging rows and columns. The elements on the diagonal will remain unchanged, since $i = j$.

The transpose of the example matrix looks like

```
        1,   2,    3
```

```
         -------------

B(0,   6,   6,      8

 (1,   1,   1,     15

 (2,   1,   5,     91

 (3,   2,   2,     11

 (4,   3,   2,      3

 (5,   3,   6,     28

 (6,   4,   1,     22

 (7,   4,   3,     -6

 (8,   6,   1,    -15
```

Since *A* is organized by row, our first idea for a transpose algorithm might be

for *each row* i **do**

take element (i,j,val) and

store it in (j,i,val) of the transpose

end

The difficulty is in not knowing where to put the element $(j,i,$val$)$ until all other elements which precede it have been processed. In our example of figure 2.2, for instance, we have item

(1,1,15) which becomes (1,1,15)

(1,4,22) which becomes (4,1,22)

(1,6, - 15) which becomes (6,1, - 15).

If we just place them consecutively, then we will need to insert many new triples, forcing us to move

elements down very often. We can avoid this data movement by finding the elements in the order we want them, which would be as

```
for all elements in column j do

place element (i,j,val) in position (j,i,val)

end
```

This says find all elements in column 1 and store them into row 1, find all elements in column 2 and store them in row 2, etc. Since the rows are originally in order, this means that we will locate elements in the correct column order as well. Let us write out the algorithm in full.

```
procedure TRANSPOSE (A,B)

// A is a matrix represented in sparse form//

// B is set to be its transpose//

1    (m,n,t) □ (A(0,l),A(0,2),A(0,3))

2    (B(0,1),B(0,2),B(0,3)) □ (n,m,t)

3    if t □ 0 then return      //check for zero matrix//

4    q □ 1       //q is position of next term in B//

5    for col □ 1 to n do     //transpose by columns//

6      for p □ 1 to t do      //for all nonzero terms do//

7         if A(p,2) = col      //correct column//

8              then [(B(q,1),B(q,2),B(q,3)) □    //insert next term
of B//

9                     (A(p,2),A(p,1),A(p,3))
```

```
10                      q □ q + 1]

11        end

12     end

13  end TRANSPOSE
```

The above algorithm makes use of (lines 1, 2, 8 and 9) the vector replacement statement of SPARKS. The statement

(a,b,c) □ (d,e,f)

is just a shorthand way of saying

a □ d; b □ e; c □ f.

It is not too difficult to see that the algorithm is correct. The variable q always gives us the position in B where the next term in the transpose is to be inserted. The terms in B are generated by rows. Since the rows of B are the columns of A, row i of B is obtained by collecting all the nonzero terms in column i of A. This is precisely what is being done in lines 5-12. On the first iteration of the **for** loop of lines 5-12 all terms from column 1 of A are collected, then all terms from column 2 and so on until eventually, all terms from column n are collected.

How about the computing time of this algorithm! For each iteration of the loop of lines 5-12, the **if** clause of line 7 is tested t times. Since the number of iterations of the loop of lines 5-12 is n, the total time for line 7 becomes nt. The assignment in lines 8-10 takes place exactly t times as there are only t nonzero terms in the sparse matrix being generated. Lines 1-4 take a constant amount of time. The total time for the algorithm is therefore $O(nt)$. In addition to the space needed for A and B, the algorithm requires only a fixed amount of additional space, i.e. space for the variables m, n, t, q, col and p.

We now have a matrix transpose algorithm which we believe is correct and which has a computing time of $O(nt)$. This computing time is a little disturbing since we know that in case the matrices had been represented as two dimensional arrays, we could have obtained the transpose of a n X m matrix in time $O(nm)$. The algorithm for this takes the form:

```
for j □ 1 to n do

for i □ 1 to m do
```

```
B(j,i) □ A(i,j)

end


end
```

The $O(nt)$ time for algorithm TRANSPOSE becomes $O(n^2 m)$ when t is of the order of nm. This is worse than the $O(nm)$ time using arrays. Perhaps, in an effort to conserve space, we have traded away too much time. Actually, we can do much better by using some more storage. We can in fact transpose a matrix represented as a sequence of triples in time $O(n + t)$. This algorithm, FAST__TRANSPOSE, proceeds by first determining the number of elements in each column of A. This gives us the number of elements in each row of B. From this information, the starting point in B of each of its rows is easily obtained. We can now move the elements of A one by one into their correct position in B.

```
procedure FAST--TRANSPOSE(A,B)

//A is an array representing a sparse m X n matrix with t nonzero

terms. The transpose is stored in B using only O(t + n)

operations//

declare S(1:n),T(1:n);        //local arrays used as pointers//

1        (m,n,t) □ (A(0,1),A(0,2),A(0,3))

2        (B(0,1),B(0,2),B(0,3)) □ (n,m,t)     //store dimensions of

transpose//

3        if t □ 0 then return  //zero matrix//

4        for i □ 1 to n do S(i) □ 0 end

5        for i □ 1 to t do   //S(k) is the number of//

6          S(A(i,2)) □ S(A(i,2)) + 1      //elements in row k of B//

7        end
```

```
8       T(1) □ 1

9       for i □ 2 to n do                    //T(i) is the starting//

10          T(i) □ T(i - 1) + S(i - 1)      //position of row i in

B//

11      end

12      for i □ 1 to t do        //move all t elements of A to B//

13          j □ A(i,2)              //j is the row in B//

14          (B(T(j),1),B(T(j),2),B(T(j),3))□          //store in
triple//

15              (A(i,2),A(i,1),A(i,3))

16          T(j) □ T(j) + 1        //increase row j to next spot//

17      end

18  end FAST--TRANSPOSE
```

The correctness of algorithm FAST--TRANSPOSE follows from the preceding discussion and the observation that the starting point of row i, $i > 1$ of B is $T(i - 1) + S(i - 1)$ where $S(i - 1)$ is the number of elements in row i - 1 of B and $T(i - 1)$ is the starting point of row i - 1. The computation of S and T is carried out in lines 4-11. In lines 12-17 the elements of A are examined one by one starting from the first and successively moving to the t-th element. $T(j)$ is maintained so that it is always the position in B where the next element in row j is to be inserted.

There are four loops in FAST--TRANSPOSE which are executed n, t, n - 1, and t times respectively. Each iteration of the loops takes only a constant amount of time, so the order of magnitude is $O(n + t)$. The computing time of $O(n + t)$ becomes $O(nm)$ when t is of the order of nm. This is the same as when two dimensional arrays were in use. However, the constant factor associated with FAST__TRANSPOSE is bigger than that for the array algorithm. When t is sufficiently small compared to its maximum of nm, FAST__TRANSPOSE will be faster. Hence in this representation, we save both space and time! This was not true of TRANSPOSE since t will almost always be greater than max$\{n,m\}$ and $O(nt)$ will

therefore always be at least $O(nm)$. The constant factor associated with TRANSPOSE is also bigger than the one in the array algorithm. Finally, one should note that FAST__TRANSPOSE requires more space than does TRANSPOSE. The space required by FAST__TRANSPOSE can be reduced by utilizing the same space to represent the two arrays S and T.

If we try the algorithm on the sparse matrix of figure 2.2, then after execution of the third **for** loop the values of S and T are

```
        (1)   (2)   (3)   (4)   (5)   (6)

S  =     2     1     2     2     0     1

T  =     1     3     4     6     8     8
```

$S(i)$ is the number of entries in row i of the transpose. $T(i)$ points to the position in the transpose where the next element of row i is to be stored.

Suppose now you are working for a machine manufacturer who is using a computer to do inventory control. Associated with each machine that the company produces, say MACH(1) to MACH(m), there is a list of parts that comprise each machine. This information could be represented in a two dimensional table

```
           PART(1)   PART(2)   PART(3)    ...    PART(n)


           ----------------------------------------

MACH(1)  |    0,        5,        2,      ...,      0

MACH(2)  |    0,        0,        0,      ...,      3

MACH(3)  |    1,        1,        0,      ...,      8

   .     |    .         .         .                 .

   .     |    .         .         .                 .

   .     |    .         .         .                 .

MACH(m)  |    6,        0,        0,      ...,      7


         |
```

```
|              array MACHPT(m,n)
```

The table will be sparse and all entries will be non-negative integers. MACHPT(i,j) is the number of units of PART(j) in MACH(i). Each part is itself composed of smaller parts called microparts. This data will also be encoded in a table whose rows are PART(1) to PART(n) and whose columns are MICPT(1) to MICPT(p). We want to determine the number of microparts that are necessary to make up each machine.

Observe that the number of MICPT(j) making up MACH(i) is

MACHPT(i,1) * MICPT(1,j) + MACHPT(i,2) * MICPT(2,j)

+ ... + MACHPT(i,n) * MICPT(n,j)

where the arrays are named MACHPT(m,n) and MICPT(n,p). This sum is more conveniently written as

☐

If we compute these sums for each machine and each micropart then we will have a total of *mp* values which we might store in a third table MACHSUM(m,p). Regarding these tables as matrices this application leads to the general definition of matrix product:

Given *A* and *B* where *A* is $m \times n$ and *B* is $n \times p$, the product matrix *C* has dimension $m \times p$. Its *i,j* element is defined as

☐

for ☐ and ☐. The product of two sparse matrices may no longer be sparse, for instance,

☐

Consider an algorithm which computes the product of two sparse matrices represented as an ordered list instead of an array. To compute the elements of *C* row-wise so we can store them in their proper place without moving previously computed elements, we must do the following: fix a row of *A* and find all elements in column *j* of *B* for $j = 1,2,...,p$. Normally, to find all the elements in column *j* of *B* we would have to scan all of *B*. To avoid this, we can first compute the transpose of *B* which will put all column elements consecutively. Once the elements in row *i* of *A* and column *j* of *B* have been located, we just do a merge operation similar to the polynomial addition of section 2.2. An alternative approach is explored in the exercises.

Before we write a matrix multiplication procedure, it will be useful to define a sub-procedure:

```
procedure STORESUM (C,q,row,col,sum)

//if sum is nonzero then along with its row and column position

it is stored into the q-th entry of the matrix//

if sum ≠ 0 then [(C(q,1),C(q,2),C(q,3) □

(row,col,sum)

q □ q + 1; sum □ 0]

end STORESUM
```

The algorithm MMULT which multiplies the matrices *A* and *B* to obtain the product matrix *C* uses the strategy outlined above. It makes use of variables *i,j,q,r*,col and row__begin. The variable *r* is the row of *A* that is currently being multiplied with the columns of *B*. *row*__begin is the position in *A* of the first element of row *r*. col is the column of *B* that is currently being multiplied with row *r* of *A*. *q* is the position in *C* for the next element generated. *i* and *j* are used to successively examine elements of row *r* and column col of *A* and *B* respectively. In addition to all this, line 6 of the algorithm introduces a dummy term into each of *A* and □. This enables us to handle end conditions (i.e., computations involving the last row of *A* or last column of *B*) in an elegant way.

We leave the correctness proof of this algorithm as an exercise. Let us examine its complexity. In addition to the space needed for *A, B, C* and some simple variables, space is also needed for the transpose matrix □. Algorithm FAST__TRANSPOSE also needs some additional space. The exercises explore a strategy for MMULT which does not explicitly compute □ and the only additional space needed is the same as that required by FAST__TRANSPOSE. Turning our attention to the computing time of MMULT, we see that lines 1-6 require only $O(p + t_2)$ time. The **while** loop of lines 7-34 is executed at most *m* times (once for each row of *A*). In each iteration of the **while** loop of lines 9-29 either the value of *i* or *j* or of both increases by 1 or *i* and *col* are reset. The maximum total increment in *j* over the whole loop is t_2. If d_r is the number of terms in row *r* of *A* then the value of *i* can increase at most d_r times before *i* moves to the next row of *A*.

□

When this happens, *i* is reset to row__begin in line 13. At the same time col is advanced to the next column. Hence, this resetting can take place at most *p* times (there are only *p* columns in *B*). The total maximum increments in *i* is therefore pd_r. The maximum number of iterations of the while loop of lines

9-29 is therefore $p + pd_r + t_2$. The time for this loop while multiplying with row r of A is $O(pd_r + t_2)$. Lines 30-33 take only $O(d_r)$ time. Hence, the time for the outer while loop, lines 7-34, for the iteration with row r of A is $O(pd_r + t_2)$. The overall time for this loop is then $O(\Sigma_r(pd_r + t_2)) = O(pt_1 + mt_2)$.

Once again, we may compare this computing time with the time to multiply matrices when arrays are used. The classical multiplication algorithm is:

```
for i ← 1 to m do

for j ← 1 to p do

sum ← 0

for k ← 1 to n do

sum ← sum + A(i,k) * B(k,j)

end

C(i,j) ← sum

end

end
```

The time for this is $O(mnp)$. Since $t_1 \leq nm$ and $t_2 \leq np$, the time for MMULT is at most $O(mnp)$. However, its constant factor is greater than that for matrix multiplication using arrays. In the worst case when $t_1 = nm$ or $t_2 = np$, MMULT will be slower by a constant factor. However, when t_1 and t_2 are sufficiently smaller than their maximum values i.e., A and B are sparse, MMULT will outperform the above multiplication algorithm for arrays.

The above analysis for MMULT is nontrivial. It introduces some new concepts in algorithm analysis and you should make sure you understand the analysis.

As in the case of polynomials, this representation for sparse matrices permits one to perform operations such as addition, transpose and multiplication efficiently. There are, however, other considerations which make this representation undesirable in certain applications. Since the number of terms in a sparse matrix is variable, we would like to represent all our sparse matrices in one array rather than using a separate array for each matrix. This would enable us to make efficient utilization of space. However,

when this is done, we run into difficulties in allocating space from this array to any individual matrix. These difficulties also arise with the polynomial representation of the previous section and will become apparent when we study a similar representation for multiple stacks and queues (section 3.4).

2.4 REPRESENTATION OF ARRAYS

Even though multidimensional arrays are provided as a standard data object in most high level languages, it is interesting to see how they are represented in memory. Recall that memory may be regarded as one dimensional with words numbered from 1 to *m*. So, we are concerned with representing *n* dimensional arrays in a one dimensional memory. While many representations might seem plausible, we must select one in which the location in memory of an arbitrary array element, say $A(i_1,i_2, ...,i_n)$, can be determined efficiently. This is necessary since programs using arrays may, in general, use array elements in a random order. In addition to being able to retrieve array elements easily, it is also necessary to be able to determine the amount of memory space to be reserved for a particular array. Assuming that each array element requires only one word of memory, the number of words needed is the number of elements in the array. If an array is declared $A(l_1 :u_1,l_2:u_2, ...,l_n:u_n)$, then it is easy to see that the number of elements is

One of the common ways to represent an array is in row major order. If we have the declaration

A(4:5, 2:4, 1:2, 3:4)

then we have a total of 2 ☐ 3 ☐ 2 ☐ 2 = 24 elements. Then using row major order these elements will be stored as

A(4,2,1,3), *A*(4,2,1,4), *A*(4,2,2,3), *A*(4,2,2,4)

and continuing

A(4,3,1,3), *A*(4,3,1,4), *A*(4,3,2,3), *A*(4,3,2,4)

for 3 more sets of four until we get

A(5,4,1,3), *A*(5,4,1,4), *A*(5,4,2,3), *A*(5,4,2,4).

We see that the subscript at the right moves the fastest. In fact, if we view the subscripts as numbers, we see that they are, in some sense, increasing:

4213,4214, ...,5423,5424.

Another synonym for row major order is lexicographic order.

From the compiler's point of view, the problem is how to translate from the name $A(i_1,i_2, ...,i_n)$ to the correct location in memory. Suppose $A(4,2,1,3)$ is stored at location 100. Then $A(4,2,1,4)$ will be at 101 and $A(5,4,2,4)$ at location 123. These two addresses are easy to guess. In general, we can derive a formula for the address of any element. This formula makes use of only the starting address of the array plus the declared dimensions.

To simplify the discussion we shall assume that the lower bounds on each dimension l_i are 1. The general case when l_i can be any integer is discussed in the exercises. Before obtaining a formula for the case of an n-dimensional array, let us look at the row major representation of 1, 2 and 3 dimensional arrays. To begin with, if A is declared $A(1:u_1)$, then assuming one word per element, it may be represented in sequential memory as in figure 2.3. If $\square$ is the address of $A(1)$, then the address of an arbitrary element $A(i)$ is just $\square + (i - 1)$.

```
array element:   A(1), A(2), A(3), ..., A(i),       ..., A(u₁)

address:         □,  □ + 1,  □ + 2,  ...,  □ + i - 1, ...,  □ + u₁ - 1

                 total number of elements = u₁
```

Figure 2.3: Sequential representation of A(1:u$_1$)

The two dimensional array $A(1:u_1,1:u_2)$ may be interpreted as u_1 rows: row $_1$,row $_2$, ...,row $_{u1}$, each row consisting of u_2 elements. In a row major representation, these rows would be represented in memory as in figure 2.4.

$\square$

$\square$

Figure 2.4: Sequential representation of A(u$_1$,u$_2$)

Again, if $\square$ is the address of $A(1 ,1)$, then the address of $A(i,1)$ is $\square + (i - 1)u_2$, as there are $i - 1$ rows each of size u_2 preceding the first element in the i-th row. Knowing the address of $A(i,1)$, we can say that the address of $A(i,j)$ is then simply $\square + (i - 1)u_2 + (j - 1)$.

Figure 2.5 shows the representation of the 3 dimensional array $A(1:u_1,1:u_2,1:u_3)$. This array is interpreted as u_1 2 dimensional arrays of dimension u_2 x u_3 . To locate $A(i,j,k)$, we first obtain ☐ + $(i - 1)u_2 u_3$ as the address for $A(i,1,1)$ since there are i- 1 2 dimensional arrays of size u_2 x u_3 preceding this element. From this and the formula for addressing a 2 dimensional array, we obtain ☐ + $(i - 1)u_2 u_3 + (j - 1)u_3 + (k - 1)$ as the address of $A(i,j,k)$.

Generalizing on the preceding discussion, the addressing formula for any element $A(i_1,i_2, ...,i_n)$ in an n-dimensional array declared as $A(u_1,u_2, ...,u_n)$ may be easily obtained. If ☐ is the address for $A(1,1, ...,1)$ then ☐ + $(i_1, - 1)u_2u_3 ... u_n$ is the address for $A(i_1,1, ...,1)$. The address for $A(i_1,i_2,1, ...,1)$ is then ☐ + $(i_1, - 1)u_2u_3 ... u_n + (i_2 - 1)u_3u_4 ... u_n$.

Repeating in this way the address for $A(i_1,i_2, ...,i_n)$ is

☐

(a) 3-dimensional array $A(u_1,u_2,u_3)$ regarded as u_1 2-dimensional arrays.

☐

(b) Sequential row major representation of a 3-dimensional array. Each 2-dimensional array is represented as in Figure 2.4.

Figure 2.5: Sequential representation of $A(u_1,u_2,u_3)$

☐

Note that a_j may be computed from ☐ using only one multiplication as $a_j = u_{j+1} a_{j+1}$. Thus, a compiler will initially take the declared bounds $u_1, ...,u_n$ and use them to compute the constants $a_1, ...,a_{n-1}$ using $n - 2$ multiplications. The address of $A(i_1, ...,i_n)$ can then be found using the formula, requiring $n - 1$ more multiplications and n additions.

An alternative scheme for array representation, column major order, is considered in exercise 21.

To review, in this chapter we have used arrays to represent ordered lists of polynomials and sparse matrices. In all cases we have been able to move the values around, accessing arbitrary elements in a fixed amount of time, and this has given us efficient algorithms. However several problems have been raised. By using a sequential mapping which associates a_i of $(a_1, ...,a_n)$ with the i-th element of the

array, we are forced to move data around whenever an insert or delete operation is used. Secondly, once we adopt one ordering of the data we sacrifice the ability to have a second ordering simultaneously.

EXERCISES

1. Write a SPARKS procedure which multiplies two polynomials represented using scheme 2 in Section 2.2. What is the computing time of your procedure?

2. Write a SPARKS procedure which evaluates a polynomial at a value x_0 using scheme 2 as above. Try to minimize the number of operations.

3. If $A = (a_1, ...,a_n)$ and $B = (b_1, ...,b_m)$ are ordered lists, then $A < B$ if $a_i = b_i$ for ▢ and $a_j < b_j$ or if $a_i = b_i$ for ▢ and $n < m$. Write a procedure which returns -1,0, + 1 depending upon whether $A < B, A = B$ or $A > B$. Assume you can compare atoms a_i and b_j.

4. Assume that n lists, $n > 1$, are being represented sequentially in the one dimensional array SPACE (1: m). Let FRONT(i) be one less than the position of the first element in the ith list and let REAR(i) point to the last element of the ith list, $1 \le i \le n$. Further assume that REAR(i) $\le$ FRONT($i + 1$), $1 \le i \le n$ with FRONT($n + 1$) = m. The functions to be performed on these lists are insertion and deletion.

a) Obtain suitable initial and boundary conditions for FRONT(i) and REAR(i)

b) Write a procedure INSERT(i,j,item) to insert item after the (j - 1)st element in list i. This procedure should fail to make an insertion only if there are already m elements in SPACE.

5. Using the assumptions of (4) above write a procedure DELETE(i,j,item) which sets item to the j-th element of the i-th list and removes it. The i-th list should be maintained as sequentially stored.

6. How much space is actually needed to hold the Fibonacci polynomials $F_0, F_1, .., F_{100}$?

7. In procedure MAIN why is the second dimension of $F = 203$?

8. The polynomials $A(x) = x^{2n} + x^{2n-2} + ... + x^2 + x^0$ and $B(x) = x^{2n+1} + x^{2n-1} + ... + x^3 + x$ cause PADD to work very hard. For these polynomials determine the exact number of times each statement will be executed.

9. Analyze carefully the computing time and storage requirements of algorithm FAST--TRANSPOSE. What can you say about the existence of an even faster algorithm?

10. Using the idea in FAST--TRANSPOSE of *m* row pointers, rewrite algorithm MMUL to multiply two sparse matrices *A* and *B* represented as in section 2.3 without transposing *B*. What is the computing time of your algorithm?

11. When all the elements either above or below the main diagonal of a square matrix are zero, then the matrix is said to be triangular. Figure 2.6 shows a lower and upper triangular matrix.

Figure 2.6

In a lower triangular matrix, *A*, with *n* rows, the maximum number of nonzero terms in row *i* is *i*. Hence, the total number of nonzero terms is $\sum^n_{i=1} i = n(n + 1)/2$. For large *n* it would be worthwhile to save the space taken by the zero entries in the upper triangle. Obtain an addressing formula for elements a_{ij} in the lower triangle if this lower triangle is stored by rows in an array $B(1:n(n + 1)/2)$ with $A(1,1)$ being stored in $B(1)$.

What is the relationship between *i* and *j* for elements in the zero part of A?

12. Let A and B be two lower triangular matrices, each with *n* rows. The total number of elements in the lower triangles is $n(n + 1)$. Devise a scheme to represent both the triangles in an array $C(1:n,1:n + 1)$. [Hint: represent the triangle of *A* as the lower triangle of *C* and the transpose of *B* as the upper triangle of *C*.] Write algorithms to determine the values of $A(i,j), B(i,j)$ $1 \le i, j \le n$ *from the array* C.

13. Another kind of sparse matrix that arises often in numerical analysis is the tridiagonal matrix. In this square matrix, all elements other than those on the major diagonal and on the diagonals immediately above and below this one are zero

Figure 2.7: Tridiagonal matrix A

If the elements in the band formed by these three diagonals are represented rowwise in an array, *B*, with $A(1,1)$ being stored at $B(1)$, obtain an algorithm to determine the value of $A(i,j)$, $1 \le i, j \le n$ *from the array* B.

14. Define a square band matrix $A_{n,a}$ to be a *n* x *n* matrix in which all the nonzero terms lie in a band centered around the main diagonal. The band includes *a* - 1 diagonals below and above the main diagonal and also the main diagonal.

☐

a) How many elements are there in the band of $A_{n,a}$?

b) What is the relationship between i and j for elements a_{ij} in the band of $A_{n,a}$?

c) Assume that the band of An,a is stored sequentially in an array B by diagonals starting with the lowermost diagonal. Thus $A4,3$ above would have the following representation:

B(1)	B(2)	B(3)	B(4)	B(5)	B(6)	B(7)	B(8)	B(9)	B(10)	B(11)	B(12)	B(13)	B(14)
9	7	8	3	6	6	0	2	8	7	4	9	8	4
a_{31}	a_{42}	a_{21}	a_{32}	a_{43}	a_{11}	a_{22}	a_{33}	a_{44}	a_{12}	a_{23}	a_{34}	a_{13}	a_{24}

Obtain an addressing formula for the location of an element aij in the lower band of An,a.

e.g. LOC(a31) = 1, LOC(a42) = 2 in the example above.

15. A generalized band matrix $A_{n,a,b}$ is a n x n matrix A in which all the nonzero terms lie in a band made up of a - 1 diagonals below the main diagonal, the main diagonal and b - 1 diagonals above the main diagonal (see the figure on the next page)

a) How many elements are there in the band of $A_{n,a,b}$?

b) What is the relationship between i and j for elements a_{ij} in the band of $A_{n,a,b}$?

c) Obtain a sequential representation of the band of $A_{n,a,b}$ in the one dimensional array B. For this representation write an algorithm VALUE (n,a,b,i,j,B) which determines the value of element a_{ij} in the matrix $A_{n,a,b}$. The band of $A_{n,a,b}$ is represented in the array B.

☐

16. How much time does it take to locate an arbitrary element $A(i,j)$ in the representation of section 2.3 and to change its value?

17. A variation of the scheme discussed in section 2.3 for sparse matrix representation involves

representing only the non zero terms in a one dimensional array V_A in the order described. In addition, a strip of n x m bits, $B_A(n,m)$, is also kept. $B_A(i,j) = 0$ if $A(i,j) = 0$ and $B_A(i,j) = 1$ if $A(i,j) \neq 0$. The figure below illustrates the representation for the sparse matrix of figure 2.1.

(i) On a computer with w bits per word, how much storage is needed to represent a sparse matrix Anxm with t nonzero terms?

(ii) Write an algorithm to add two sparse matrices A and C represented as above to obtain $D = A + C$. How much time does your algorithm take ?

(iii) Discuss the merits of this representation versus the representation of section 2.3. Consider space and time requirements for such operations as random access, add, multiply, and transpose. Note that the random access time can be improved somewhat by keeping another array $R_A(i)$ such that $R_A(i) =$ number of nonzero terms in rows 1 through i - 1.

18. A complex-valued matrix X is represented by a pair of matrices (A,B) where A and B contain real values. Write a program which computes the product of two complex valued matrices (A,B) and (C,D), where

$(A,B) * (C,D) = (A + iB) * (C + iD) = (AC - BD) + i(AD + BC)$

Determine the number of additions and multiplications if the matrices are all $n \times n$.

19. How many values can be held by an array with dimensions $A(0:n)$, $B(-1:n,1:m)$, $C(-n:0,2)$?

20. Obtain an addressing formula for the element $A(i_1,i_2, ...,i_n)$ in an array declared as $A(l_1:u_1,l_2:u_2, ...,l_n:u_n)$. Assume a row major representation of the array with one word per element and $\square$ the address of A $(l_1,l_2, ...,l_n)$.

21. Do exercise 20 assuming a column major representation. In this representation, a 2 dimensional array is stored sequentially by column rather than by rows.

22. An m X n matrix is said to have a *saddle point* if some entry $A(i,j)$ is the smallest value in row i and the largest value in column j. Write a SPARKS program which determines the location of a saddle point if one exists. What is the computing time of your method?

23. Given an array $A(1:n)$ produce the array $Z(1:n)$ such that $Z(1) = A(n)$, $Z(2) = A(n$ - $1)$, ..., $Z(n$ - $1) = A(2)$, $Z(n) = A(1)$. Use a minimal amount of storage.

24. One possible set of axioms for an ordered list comes from the six operations of section 2.2.

structure ORDERED__LlST(atoms)

declare *MTLST( )* □ *list*

LEN(list) □ *integer*

RET(list,integer) □ *atom*

STO(list,integer,atom) □ *list*

INS(list,integer,atom) □ *list*

DEL(list,integer) □ *list;*

for all *L* ∈ *list, i.j* ∈ *integer a,b* ∈ *atom* **let**

LEN(MTLST) :: = 0; *LEN(STO(L,i,a))* :: = 1 + *LEN(L)*

RET(MTLST,j) :: = *error*

RET(STO(L,i,a),j) :: =

if *i = j* **then** *a* **else** *RET(L,j)*

INS(MTLST,j,b) :: = *STO(MTLST,j,b)*

INS(STO(L,i,a),j,b) :: =

if *i* ≥ *j* **then** *STO(INS(L,j,b), i + 1,a)*

else *STO(INS(L,j,b),i,a)*

DEL(MTLST,j) :: = *MTLST*

DEL(STO(L,i,a),j) :: =

if *i = j* **then** *DEL(L,j)*

else if *i* > *j* **then** *STO(DEL(L,j),i - 1,a)*

else *STO(DEL(L,j),i,a)*

end

end *ORDERED__LIST*

Use these axioms to describe the list $A = (a, b, c, d, e)$ and show what happens when DEL $(A,2)$ is executed.

25. There are a number of problems, known collectively as "random walk" problems which have been of long standing interest to the mathematical community. All but the most simple of these are extremely difficult to solve and for the most part they remain largely unsolved. One such problem may be stated as follows:

A (drunken) cockroach is placed on a given square in the middle of a tile floor in a rectangular room of size n x m tiles. The bug wanders (possibly in search of an aspirin) randomly from tile to tile throughout the room. Assuming that he may move from his present tile to any of the eight tiles surrounding him (unless he is against a wall) with equal probability, how long will it take him to touch every tile on the floor at least once?

Hard as this problem may be to solve by pure probability theory techniques, the answer is quite easy to solve using the computer. The technique for doing so is called "simulation" and is of wide-scale use in industry to predict traffic-flow, inventory control and so forth. The problem may be simulated using the following method:

An array KØUNT dimensioned N X M is used to represent the number of times our cockroach has reached each tile on the floor. All the cells of this array are initialized to zero. The position of the bug on the floor is represented by the coordinates (IBUG,JBUG) and is initialized by a data card. The 8 possible moves of the bug are represented by the tiles located at (IBUG + IMØVE(K), JBUG + JMØVE(K)) where $1 \le K \le 8$ and:

```
IMØVE(1) = -1            JMØVE(1) = 1

IMØVE(2) = 0             JMØVE(2) = 1

IMØVE(3) = 1             JMØVE(3) = 1

IMØVE(4) = 1             JMØVE(4) = 0
```

```
IMØVE(5) = 1              JMØVE(5) = -1

IMØVE(6) = 0             JMØVE(6) = -1

IMØVE(7) = -1             JMØVE(7) = -1

IMØVE(8) = -1             JMØVE(8) = 0
```

A *random* walk to one of the 8 given squares is simulated by generating a random value for *K* lying between 1 and 8. Of course the bug cannot move outside the room, so that coordinates which lead up a wall must be ignored and a new random combination formed. Each time a square is entered, the count for that square is incremented so that a non-zero entry shows the number of times the bug has landed on that square so far. When every square has been entered at least once, the experiment is complete.

Write a program to perform the specified simulation experiment. Your program *MUST*:

1) Handle values of *N* and *M*

$2 < N \leq 40$

$2 \leq M \leq 20$

2) Perform the experiment for

a) *N* = 15, *M* = 15 starting point: (20,10)

b) *N* = 39, *M* = 19 starting point: (1,1)

3) Have an iteration limit, that is, a maximum number of squares the bug may enter during the experiment. This assures that your program does not get "hung" in an "infinite" loop. A maximum of 50,000 is appropriate for this lab.

4) For each experiment print: a) the total number of legal moves which the cockroach makes; b) the final KØUNT array. This will show the "density" of the walk, that is the number of times each tile on the floor was touched during the experiment.

(Have an aspirin) This exercise was contributed by Olson.

26. Chess provides the setting for many fascinating diversions which are quite independent of the game itself. Many of these are based on the strange "L-shaped" move of the knight. A classical example is the problem of the knight's tour, which has captured the attention of mathematicians and puzzle enthusiasts

since the beginning of the eighteenth century. Briefly stated, the problem is to move the knight, beginning from any given square on the chessboard, in such a manner that it travels successively to all 64 squares, touching each square once and only once. It is convenient to represent a solution by placing the numbers 1,2, ...,64 in the squares of the chessboard indicating the order in which the squares are reached. Note that it is not required that the knight be able to reach the initial position by one more move; if this is possible the knight's tour is called re-entrant.

One of the more ingenious methods for solving the problem of the knight's tour is that given by J. C. Warnsdorff in 1823. His rule is that the knight must always be moved to one of the squares from which there are the fewest exits to squares not already traversed.

The goal of this exercise is to write a computer program to implement Warnsdorff's rule. The ensuing discussion will be much easier to follow, however, if the student will first try to construct a particular solution to the problem by hand before reading any further.

The most important decisions to be made in solving a problem of this type are those concerning how the data is to be represented in the computer. Perhaps the most natural way to represent the chessboard is by an 8 x 8 array BØARD as shown in the figure below. The eight possible moves of a knight on square (5,3) are also shown in the figure.

BØARD

```
    1   2   3   4   5   6   7   8

    ----------------------

1


2


3       8       1


4   7               2


5           K


6   6               3


7       5       4


8
```

In general a knight at (I,J) may move to one of the squares $(I - 2, J + 1)$, $(I -1, J + 2)$, $(I + 1, J + 2)$, $(I + 2, J + 1)$, $(I + 2, J - 1)$, $(I + 1, J - 2)$, $(I -1, J - 2)$, $(I - 2, J - 1)$. Notice, however that if (I,J) is located near one of the edges of the board, some of these possibilities could move the knight off the board, and of course this is not permitted. The eight possible knight moves may conveniently be represented by two arrays KTM∅V1 and KTM∅V2 as shown below.

KTM∅**V1**	**KTM**∅V2
-2	1
-1	2
1	2
2	1
2	-1
1	-2
-1	-2
-2	-1

Then a knight at (I,J) may move to $(I + \text{KTM∅V1}(K), J + \text{KTM∅V2}(K))$, where K is some value between 1 and 8, provided that the new square lies on the chessboard.

Below is a description of an algorithm for solving the knight's tour problem using Warnsdorff's rule. The data representation discussed in the previous section is assumed.

a. [Initialize chessboard] For $1 \leq I,J \leq 8$ set B∅ARD$(I,J) = 0$.

b. [Set starting position] Read and print I,J and then set B∅ARD(I,J) to 1.

c. [Loop] For $2 \leq M \leq 64$ do steps d through g.

d. [Form set of possible next squares] Test each of the eight squares one knight's move away from (I,J) and form a list of the possibilities for the next square (NEXTI(L), NEXTJ(L)). Let NP∅S be the number of possibilities. (That is, after performing this step we will have NEXTI$(L) = I + \text{KTM∅V1}(K)$ and NEXTJ$(L) = J + \text{KTM∅V2}(K)$, for certain values of K between 1 and 8. Some of the squares $(I + \text{KTM∅V1}(K), J + \text{KTM∅V2}(K))$ may be impossible for the next move either because they lie off the

chessboard or because they have been previously occupied by the knight--i.e., they contain a nonzero number. In every case we will have $0 \leq NP\emptyset S \leq 8$.)

e. [Test special cases] If $NP\emptyset S = 0$ the knight's tour has come to a premature end; report failure and then go to step h. If $NP\emptyset S = 1$ there is only one possibility for the next move; set MIN = 1 and go right to step g.

f. [Find next square with minimum number of exits] For $1 \leq L \leq NP\emptyset S$ set EXITS(L) to the number of exits from square (NEXTI(L),NEXTJ(L)). That is, for each of the values of L examine each of the next squares (NEXTI(L) + KTM$\emptyset$V1(K), NEXTJ(L) + KTM$\emptyset$V2(K)) to see if it is an exit from NEXTI(L), NEXTJ(L)), and count the number of such exits in EXITS(L). (Recall that a square is an exit if it lies on the chessboard and has not been previously occupied by the knight.) Finally, set MIN to the location of the minimum value of EXITS. (There may be more than one occurrences of the minimum value of EXITS. If this happens, it is convenient to let MIN denote the first such occurrence, although it is important to realize that by so doing we are not actually guaranteed of finding a solution. Nevertheless, the chances of finding a complete knight's tour in this way are remarkably good, and that is sufficient for the purposes of this exercise.)

g. [Move knight] Set I = NEXTI(MIN), J = NEXTJ(MIN) and B$\emptyset$ARD(I,J) = M. (Thus, (I,J) denotes the new position of the knight, and B$\emptyset$ARD(I,J) records the move in proper sequence.)

h. [Print] Print out B$\emptyset$ARD showing the solution to the knight's tour, and then terminate the algorithm.

The problem is to write a program which corresponds to this algorithm. This exercise was contributed by Legenhausen and Rebman.

CHAPTER 3: STACKS AND QUEUES

3.1 FUNDAMENTALS

Two of the more common data objects found in computer algorithms are stacks and queues. They arise so often that we will discuss them separately before moving on to more complex objects. Both these data objects are special cases of the more general data object, an ordered list which we considered in the previous chapter. Recall that $A = (a_1, a_2, ...,a_n)$, is an ordered list of ☐ elements. The a_i are referred to as atoms which are taken from some set. The null or empty list has $n = 0$ elements.

A *stack* is an ordered list in which all insertions and deletions are made at one end, called the *top*. A *queue* is an ordered list in which all insertions take place at one end, the *rear*, while all deletions take place at the other end, the *front*. Given a stack $S = (a_1, ...a_n)$ then we say that a_1 is the *bottommost* element and element a_i is on *top* of element a_{i-1}, $1 < i \leq n$. When viewed as a queue with a_n as the rear element one says that a_{i+1} is behind a_i, $1 \leq i < n$.

☐

Figure 3.1

The restrictions on a stack imply that if the elements A,B,C,D,E are added to the stack, in that order, then the first element to be removed/deleted must be E. Equivalently we say that the last"element to be inserted into the stack will be the first to be removed. For this reason stacks are sometimes referred to as *Last In First Out* (LIFO) lists. The restrictions on a queue require that the first element which is inserted into the queue will be the first one to be removed. Thus A is the first letter to be removed, and queues are known as *First In First Out* (FIFO) lists. Note that the data object queue as defined here need not necessarily correspond to the mathematical concept of queue in which the insert/delete ru les may be different.

One natural example of stacks which arises in computer programming is the processing of subroutine calls and their returns. Suppose we have a main procedure and three subroutines as below:

```
| proc MAIN |    | proc A1 |    | proc A2 |    | proc A3 |

|      ___  |    |    ___  |    |    ___  |    |    ___  |

|      ___  |    |    ___  |    |    ___  |    |    ___  |
```
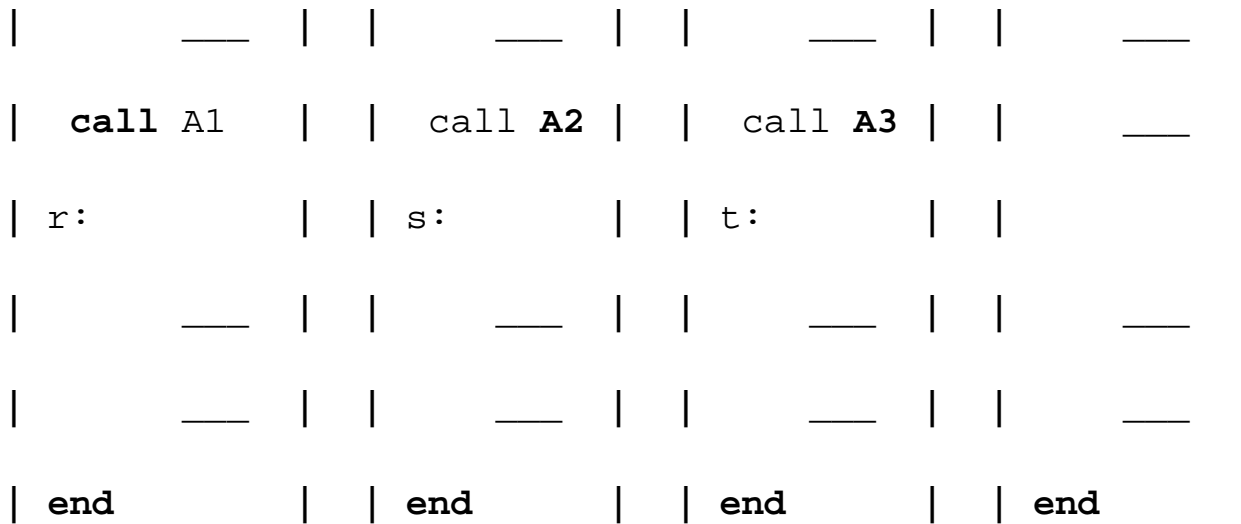
```
|        ___  | |        ___  | |        ___  | |        ___  |

|  call A1    | |  call A2 | |  call A3 | |        ___  |

| r:          | | s:        | | t:       | |          |

|        ___  | |        ___  | |        ___  | |        ___  |

|        ___  | |        ___  | |        ___  | |        ___  |

| end         | | end        | | end      | | end      |
```

Figure 3.2. Sequence of subroutine calls

The MAIN program calls subroutine $A1$. On completion of $A1$ execution of MAIN will resume at location r. The address r is passed to $A1$ which saves it in some location for later processing. $A1$ then invokes $A2$ which in turn calls $A3$. In each case the calling procedure passes the return address to the called procedure. If we examine the memory while $A3$ is computing there will be an implicit stack which looks like

(q,r,s,t).

The first entry, q, is the address in the operating system where MAIN returns control. This list operates as a stack since the returns will be made in the reverse order of the calls. Thus t is removed before s, s before r and r before q. Equivalently, this means that $A3$ must finish processing before $A2$, $A2$ before $A1$, and $A1$ before MAIN. This list of return addresses need not be maintained in consecutive locations. For each subroutine there is usually a single location associated with the machine code which is used to retain the return address. This can be severely limiting in the case of recursive calls and re-entrant routines, since every time we call a subroutine the new return address wipes out the old one. For example, if we inserted a call to $A1$ within subroutine $A3$ expecting the return to be at location u, then at execution time the stack would become (q,u,s,t) and the return address r would be lost. When recursion is allowed, it is no longer adequate to reserve one location for the return address of each subroutine. Since returns are made in the reverse order of calls, an elegant and natural solution to this subroutine return problem is afforded through the explicit use of a stack of return addresses. Whenever a return is made, it is to the top address in the stack. Implementing recursion using a stack is discussed in Section 4.10.

Associated with the object stack there are several operations that are necessary:

CREATE (S) which creates S as an empty stack;

ADD (i,S) which inserts the element i onto the stack S and returns the new stack;

DELETE (*S*) which removes the top element of stack *S* and returns the new stack;

TOP (*S*) which returns the top element of stack *S*;

ISEMTS (*S*) which returns true if *S* is empty else false;

These five functions constitute a working definition of a stack. However we choose to represent a stack, it must be possible to build these operations. But before we do this let us describe formally the structure STACK.

structure *STACK (item)*

1 **declare** *CREATE ( )* $\Box$ *stack*

2 $\qquad$ *ADD (item, stack)* $\Box$ *stack*

3 $\qquad$ *DELETE (stack)* $\Box$ *stack*

4 $\qquad$ *TOP (stack)* $\Box$ *item*

5 $\qquad$ *ISEMTS (stack)* $\Box$ *boolean;*

6 **for all** *S* $\in$ *stack, i* $\in$ *item* **let**

7 $\qquad$ *ISEMTS (CREATE)* $\qquad$ :: = **true**

8 $\qquad$ *ISEMTS (ADD (i,S))* $\qquad$ :: = **false**

9 $\qquad$ *DELETE (CREATE)* $\qquad$ :: = *error*

10 $\qquad$ *DELETE (ADD (i,S))* $\qquad$ :: = *S*

11 $\qquad$ *TOP(CREATE)* $\qquad$ :: = *error*

12 $\qquad$ *TOP(ADD(i,S))* $\qquad$ :: = *i*

13 **end**

end *STACK*

The five functions with their domains and ranges are declared in lines 1 through 5. Lines 6 through 13 are the set of axioms which describe how the functions are related. Lines 10 and 12 are the essential ones which define the last-in-first-out behavior. The above definitions describe an infinite stack for no upper bound on the number of elements is specified. This will be dealt with when we represent this structure in a computer.

The simplest way to represent a stack is by using a one-dimensional array, say STACK(1:*n*), where *n* is the maximum number of allowable entries. The first or bottom element in the stack will be stored at STACK(1), the second at STACK(2) and the i-th at STACK(*i*). Associated with the array will be a variable, top, which points to the top element in the stack. With this decision made the following implementations result:

CREATE ( ) :: = **declare** *STACK(1:n); top* $\square$ 0

ISEMTS(STACK) :: = **if** *top* = 0 **then true**

else false

TOP(STACK) :: = **if** *top* = 0 **then** *error*

else *STACK(top)*

The implementations of these three operations using an array are so short that we needn't make them separate procedures but can just use them directly whenever we need to. The ADD and DELETE operations are only a bit more complex.

procedure *ADD (item, STACK, n, top)*

//insert item into the STACK of maximum size n; top is the number

of elements curently in *STACK//*

if *top* $\geq$ *n* **then call** *STACK_FULL*

top $\square$ *top* + 1

STACK (top) $\square$ *item*

end *ADD*

procedure *DELETE* (*item*, *STACK*, *top*)

//removes the top element of *STACK* and stores it in item

unless *STACK* is empty//

if *top* ☐ 0 **then call** *STACK_EMPTY*

item ☐ *STACK* (*top*)

top ☐ *top* - 1

end *DELETE*

These two procedures are so simple that they perhaps need no more explanation. Procedure DELETE actually combines the functions TOP and DELETE. STACK_FULL and STACK_EMPTY are procedures which we leave unspecified since they will depend upon the particular application. Often a stack full condition will signal that more storage needs to be allocated and the program re-run. Stack empty is often a meaningful condition. In Section 3.3 we will see a very important computer application of stacks where stack empty signals the end of processing.

The correctness of the stack implementation above may be established by showing that in this implementation, the stack axioms of lines 7-12 of the stack structure definition are true. Let us show this for the first three rules. The remainder of the axioms can be shown to hold similarly .

(i) line 7: *ISEMTS*(*CREATE*):: = **true**

Since CREATE results in top being initialized to zero, it follows from the implementation of ISEMTS that *ISEMTS(CREATE)*:: = true.

(ii) line 8: *ISEMTS(ADD(i,S))*:: = **false**

The value of top is changed only in procedures CREATE, ADD and DELETE. CREATE initializes *top* to zero while ADD increments it by 1 so long as *top* is less than *n* (this is necessary because we can implement only a finite stack). DELETE decreases top by 1 but never allows its value to become less than zero. Hence, ADD(*i,S*) either results in an error condition (STACK_FULL), or leaves the value of *top* > 0. This then implies that *ISEMTS(ADD(i,S))*:: = **false**.

(iii) line 9: *DELETE*(*CREATE*):: = *error*

This follows from the observation that CREATE sets *top* = 0 and the procedure DELETE signals the error condition STACK_EMPTY when *top* = 0.

Queues, like stacks, also arise quite naturally in the computer solution of many problems. Perhaps the most common occurrence of a queue in computer applications is for the scheduling of jobs. In batch processing the jobs are "queued-up" as they are read-in and executed, one after another in the order they were received. This ignores the possible existence of priorities, in which case there will be one queue for each priority.

As mentioned earlier, when we talk of queues we talk about two distinct ends: the front and the rear. Additions to the queue take place at the rear. Deletions are made from the front. So, if a job is submitted for execution, it joins at the rear of the job queue. The job at the front of the queue is the next one to be executed. A minimal set of useful operations on a queue includes the following:

CREATEQ(*Q*) which creates *Q* as an empty queue;

ADDQ(*i*,*Q*) which adds the element *i* to the rear of a queue and returns the new queue;

DELETEQ(*Q*) which removes the front element from the queue *Q* and returns the resulting queue;

FRONT(*Q*) which returns the front element of *Q*;

ISEMTQ(*Q*) which returns true if *Q* is empty else false.

A complete specification of this data structure is

structure *QUEUE* (*item*)

1   **declare** *CREATEQ( )* □ *queue*

2   *ADDQ(item,queue)* □ *queue*

3   *DELETEQ(queue)* □ *queue*

4   *FRONT(queue)* □ *item*

5   *ISEMTQ(queue)* □ *boolean;*

6   **for all** *Q* ∈ *queue, i* ∈ *item* **let**

```
7        ISEMTQ(CREATEQ)    :: = true

8        ISEMTQ(ADDQ(i,Q)) :: = false

9        DELETEQ(CREATEQ)   :: = error

10       DELETEQ(ADDQ(i,Q)):: =

11         if ISEMTQ(Q) then CREATEQ

12            else ADDQ(i,DELETEQ(Q))

13       FRONT(CREATEQ)      :: = error

14       FRONT(ADDQ(i,Q))   :: =

15         if ISEMTQ(Q) then i else FRONT(Q)

16    end

17  end QUEUE
```

The axiom of lines 10-12 shows that deletions are made from the front of the queue.

The representation of a finite queue in sequential locations is somewhat more difficult than a stack. In addition to a one dimensional array $Q(1:n)$, we need two variables, *front* and *rear*. The conventions we shall adopt for these two variables are that *front* is always 1 less than the actual front of the queue and *rear* always points to the last element in the queue. Thus, *front = rear* if and only if there are no elements in the queue. The initial condition then is *front = rear* = 0. With these conventions, let us try an example by inserting and deleting jobs, J_i, from a job queue

```
           Q(1)   (2)   (3)   (4)    (5)  (6)  (7)  ... Remarks

front  rear

  0      0            queue       empty                      Initial

  0      1    Jl                                       Job 1 joins
Q
```

```
  0     2    J1     J2                                Job 2 joins
Q

  0     3    J1     J2     J3                         Job 3 joins
Q

  1     3           J2     J3                         Job 1
leaves Q

  1     4           J2     J3     J4                  Job 4 joins
Q

  2     4                  J3     J4                  Job 2
leaves Q
```

With this scheme, the following implementation of the CREATEQ, ISEMTQ, and FRONT operations results for a queue with capacity *n*:

CREATEQ(Q) :: = **declare** *Q(l:n); front* ☐ *rear* ☐ 0

ISEMTQ(Q) :: = **if** *front = rear* **then true**

else false

FRONT(Q)   :: = **if** *ISEMTQ(Q)* **then error**

else *Q(front + 1)*

The following algorithms for ADDQ and DELETEQ result:

procedure *ADDQ(item, Q, n, rear)*

//insert item into the queue represented in *Q(l:n)*//

 if *rear = n* **then call** *QUEUE_FULL*

 rear ☐ *rear + 1*

 Q(rear) ☐ *item*

end *ADDQ*

procedure *DELETEQ(item, Q, front, rear)*

//delete an element from a queue//

  if *front* = *rear* **then call** *QUEUE_EMPTY*

 front ☐ *front* + 1

 item ☐ *Q(front)*

end *DELETEQ*

The correctness of this implementation may be established in a manner akin to that used for stacks. With this set up, notice that unless the front regularly catches up with the rear and both pointers are reset to zero, then the QUEUE_FULL signal does not necessarily imply that there are *n* elements in the queue. That is, the queue will gradually move to the right. One obvious thing to do when QUEUE_FULL is signaled is to move the entire queue to the left so that the first element is again at $Q(1)$ and front = 0. This is time consuming, especially when there are many elements in the queue at the time of the QUEUE_FULL signal.

Let us look at an example which shows what could happen, in the worst case, if each time the queue becomes full we choose to move the entire queue left so that it starts at $Q(1)$. To begin, assume there are *n* elements $J_1, ...,J_n$ in the queue and we next receive alternate requests to delete and add elements. Each time a new element is added, the entire queue of *n* - 1 elements is moved left.

☐

Figure 3.3

A more efficient queue representation is obtained by regarding the array $Q(1:n)$ as circular. It now becomes more convenient to declare the array as $Q(0:n$ - 1$)$. When rear = *n* - 1, the next element is entered at $Q(0)$ in case that spot is free. Using the same conventions as before, front will always point one position counterclockwise from the first element in the queue. Again, *front* = *rear* if and only if the queue is empty. Initially we have *front* = *rear* = 1. Figure 3.4 illustrates some of the possible configurations for a circular queue containing the four elements *J*1-*J*4 with *n* > 4. The assumption of circularity changes the ADD and DELETE algorithms slightly. In order to add an element, it will be necessary to move *rear* one position clockwise, i.e.,

if *rear* = *n* - 1 **then** *rear* ☐ 0

else *rear* ☐ *rear* + 1.

☐

Figure 3.4: Circular queue of n elements and four jobs J1, J2, J3, J4

Using the modulo operator which computes remainders, this is just *rear* ☐ *(rear + 1)mod* n. *Similarly, it will be necessary to move* front *one position clockwise each time a deletion is made. Again, using the modulo operation, this can be accomplished by* front ☐ *(front + l)mod* n. *An examination of the algorithms indicates that addition and deletion can now be carried out in a fixed amount of time or* O*(1).*

```
procedure ADDQ(item, Q, n, front, rear)

//insert item in the circular queue stored in Q(0:n - 1);

 rear points to the last item and front is one position

  counterclockwise from the first item in Q//

rear ☐ (rear + 1)mod n       //advance rear clockwise//

if front = rear then call QUEUE-FULL

Q(rear) ☐ item               //insert new item//

end ADDQ

procedure DELETEQ(item, Q, n, front, rear)

//removes the front element of the queue Q(0:n - 1)//

if front = rear then call QUEUE-EMPTY

front ☐ (front + 1)mod n       //advance front clockwise//

item ☐ Q(front)          //set item to front of queue//

end DELETEQ
```

One surprising point in the two algorithms is that the test for queue full in ADDQ and the test for queue empty in DELETEQ are the same. In the case of ADDQ, however, when *front = rear* there is actually one space free, i.e. *Q*(*rear*), since the first element in the queue is not at *Q*(*front*) but is one position clockwise from this point. However, if we insert an item here, then we will not be able to distinguish between the cases full and empty, since this insertion would leave *front = rear.* To avoid this, we signal queue-full thus permitting a maximum, of *n* - 1 rather than *n* elements to be in the queue at any time. One way to use all *n* positions would be to use another variable, *tag*, to distinguish between the two situations, i.e. *tag* = 0 if and only if the queue is empty. This would however slow down the two algorithms. Since the ADDQ and DELETEQ algorithms will be used many times in any problem involving queues, the loss of one queue position will be more than made up for by the reduction in computing time.

The procedures QUEUE_FULL and QUEUE_EMPTY have been used without explanation, but they are similar to STACK_FULL and STACK_EMPTY. Their function will depend on the particular application.

3.2 A MAZING PROBLEM

The rat-in-a-maze experiment is a classical one from experimental psychology. A rat (or mouse) is placed through the door of a large box without a top. Walls are set up so that movements in most directions are obstructed. The rat is carefully observed by several scientists as it makes its way through the maze until it eventually reaches the other exit. There is only one way out, but at the end is a nice hunk of cheese. The idea is to run the experiment repeatedly until the rat will zip through the maze without taking a single false path. The trials yield his learning curve.

We can write a computer program for getting through a maze and it will probably not be any smarter than the rat on its first try through. It may take many false paths before finding the right one. But the computer can remember the correct path far better than the rat. On its second try it should be able to go right to the end with no false paths taken, so there is no sense re-running the program. Why don't you sit down and try to write this program yourself before you read on and look at our solution. Keep track of how many times you have to go back and correct something. This may give you an idea of your own learning curve as we re-run the experiment throughout the book.

Let us represent the maze by a two dimensional array, MAZE(1:*m,* 1:*n*), where a value of 1 implies a blocked path, while a 0 means one can walk right on through. We assume that the rat starts at MAZE (1,1) and the exit is at MAZE(*m,n*).

Figure 3.5

With the maze represented as a two dimensional array, the location of the rat in the maze can at any time be described by the row, *i,* and column, *j* of its position. Now let us consider the possible moves the rat can make at some point (*i,j*) in the maze. Figure 3.6 shows the possible moves from any point (*i,j*). The position (*i,j*) is marked by an *X.* If all the surrounding squares have a 0 then the rat can choose any of these eight squares as its next position. We call these eight directions by the names of the points on a compass north, northeast, east, southeast, south, southwest, west, and northwest, or N, NE, E, SE, S, SW, W, NW.

Figure 3.6

We must be careful here because not every position has eight neighbors. If *(i,j)* is on a border where either $i = 1$ or *m,* or $j = 1$ or *n*, then less than eight and possibly only three neighbors exist. To avoid checking for these border conditions we can surround the maze by a border of ones. The array will therefore be declared as MAZE(0:m + 1,0:n + 1).

Another device which will simplify the problem is to predefine the possible directions to move in a table, MOVE(1:8.1:2), which has the values

```
MOVE    1    2

        --   --

(1)    -1    0   north

(2)    -1    1   northeast

(3)     0    1   east

(4)     1    1   southeast

(5)     1    0   south

(6)     1   -1   southwest

(7)     0   -1   west

(8)    -1   -1   northwest
```

By equating the compass names with the numbers 1,2, ...,8 we make it easy to move in any direction. If

we are at position (i,j) in the maze and we want to find the position (g,h) which is southwest of i,j, then we set

$$g \quad i + \text{MOVE(6,1)}; \ h \quad j + MOVE(6,2)$$

For example, if we are at position (3,4), then position (3 + 1 = 4, 4 +(-1) = 3) is southwest.

As we move through the maze we may have the chance to go in several directions. Not knowing which one to choose, we pick one but save our current position and the direction of the last move in a list. This way if we have taken a false path we can return and try another direction. With each new location we will examine the possibilities, starting from the north and looking clockwise. Finally, in order to prevent us from going down the same path twice we use another array MARK(0:m + 1,0:n + 1) which is initially zero. MARK(i,j) is set to 1 once we arrive at that position. We assume MAZE(m,n) = 0 as otherwise there is no path to the exit. We are now ready to write a first pass at an algorithm.

```
set list to the maze entrance coordinates and direction north;

while list is not empty do

(i,j, mov) □ coordinates and direction from front of list

while there are more moves do

(g,h) □ coordinates of next move

if (g,h) = (m,n) then success

if MAZE (g,h) = 0          //the move is legal//

and MARK (g,h) = 0         //we haven't been here before//

then [MARK (g,h) □ 1

add (i,j, mov) to front of list

(i,j,mov) □ (g,h, null)]

end

end
```

```
print no path has been found
```

This is not a SPARKS program and yet it describes the essential processing without too much detail. The use of indentation for delineating important blocks of code plus the use of SPARKS key words make the looping and conditional tests transparent.

What remains to be pinned down? Using the three arrays MAZE, MARK and MOVE we need only specify how to represent the list of new triples. Since the algorithm calls for removing first the most recently entered triple, this list should be a stack. We can use the sequential representation we saw before. All we need to know now is a reasonable bound on the size of this stack. Since each position in the maze is visited at most once, at most *mn* elements can be placed into the stack. Thus *mn* locations is a safe but somewhat conservative bound. In the following maze

the only path has at most $\lceil m/2 \rceil (n + 1)$ positions. Thus *mn* is not too crude a bound. We are now ready to give a precise maze algorithm.

procedure *PATH* (*MAZE*, *MARK*, *m*, *n*, *MOVE*, *STACK*)

```
//A binary matrix MAZE (0:m + 1, 0:n + 1) holds the maze.

MARK (0:m + 1, 0:n + 1) is zero in spot (i,j) if MAZE (i,j) has not

yet been reached. MOVE (8,2) is a table used to change coordinates

(i,j) to one of 8 possible directions. STACK (mn,3) holds the

current path// MARK (1,1) ⬜ 1

(STACK(1,1),STACK(1,2),STACK(1,3)) ⬜ (1,1,2);top ⬜ 1
```

while *top* $\neq$ 0 **do**

```
(i,j,mov) ⬜ (STACK(top,1),STACK(top,2), STACK(top,3) + 1)

top ⬜ top - 1
```

while *mov* ⬜ 8 **do**

```
g ☐ i + MOVE (mov,1); h ☐ j + MOVE(mov,2)

if g = m and h = n

then [for p ☐ 1 to top do        //goal//

print (STACK(p,1),STACK(p,2)

end

print(i,j); print(m,n);return]

if MAZE(g,h) = 0 and MARK(g,h) = 0

then[MARK(g,h) ☐ 1

top ☐ top + 1

(STACK(top,1),STACK(top,2),STACK(top,3)) ☐

(i,j,mov)      //save (i,j) as part of current path//

mov ☐ 0; i ☐ g; j ☐ h]

mov ☐ mov + 1              //point to next direction//

end

end

print ('no path has been found')

end PATH
```

Now, what can we say about the computing time for this algorithm? It is interesting that even though the problem is easy to grasp, it is difficult to make any but the most trivial statement about the computing time. The reason for this is because the number of iterations of the main while loop is entirely dependent upon the given maze. What we can say is that each new position (*i,j*) that is visited gets marked, so paths

are never taken twice. There are at most eight iterations of the inner while loop for each marked position. Each iteration of the inner **while** loop takes a fixed amount of time, $O(1)$, and if the number of zeros in MAZE is z then at most z positions can get marked. Since z is bounded above by mn, the computing time is bounded by $\boxed{}$. (In actual experiments, however, the rat may be inspired by the watching psychologist and the invigorating odor from the cheese at the exit. It might reach its goal by examining far fewer paths than those examined by algorithm PATH. This may happen despite the fact that the rat has no pencil and only a very limited mental stack. It is difficult to incorporate the effect of the cheese odor and the cheering of the psychologists into a computer algorithm.) The array MARK can be eliminated altogether and MAZE(i,j) changed to 1 instead of setting MARK(i,j) to 1, but this will destroy the original maze.

3.3 EVALUATION OF EXPRESSIONS

When pioneering computer scientists conceived the idea of higher level programming languages, they were faced with many technical hurdles. One of the biggest was the question of how to generate machine language instructions which would properly evaluate any arithmetic expression. A complex assignment statement such as

$X \boxed{} A/B ** C + D* E - A * C$

(3.1)

might have several meanings; and even if it were uniquely defined, say by a full use of parentheses, it still seemed a formidable task to generate a correct and reasonable instruction sequence. Fortunately the solution we have today is both elegant and simple. Moreover, it is so simple that this aspect of compiler writing is really one of the more minor issues.

An expression is made up of operands, operators and delimiters. The expression above has five operands: A,B,C,D, and E. Though these are all one letter variables, operands can be any legal variable name or constant in our programming language. In any expression the values that variables take must be consistent with the operations performed on them. These operations are described by the operators. In most programming languages there are several kinds of operators which correspond to the different kinds of data a variable can hold. First, there are the basic arithmetic operators: plus, minus, times, divide, and exponentiation (+,-,*,/,**). Other arithmetic operators include unary plus, unary minus and **mod, ceil**, and **floor**. The latter three may sometimes be library subroutines rather than predefined operators. A second class are the relational operators: $\boxed{}$. These are usually defined to work for arithmetic operands, but they can just as easily work for character string data. ('CAT' is less than 'DOG' since it precedes 'DOG' in alphabetical order.) The result of an expression which contains relational operators is one of the two constants: **true** or **false**. Such all expression is called Boolean, named after the mathematician George Boole, the father of symbolic logic.

The first problem with understanding the meaning of an expression is to decide in what order the operations are carried out. This means that every language must uniquely define such an order. For instance, if $A = 4$, $B = C = 2$, $D = E = 3$, then in eq. 3.1 we might want X to be assigned the value

```
4/(2 ** 2) + (3 * 3) - (4 * 2)
```

```
= (4/4) + 9 - 8
```

```
= 2.
```

However, the true intention of the programmer might have been to assign X the value

```
(4/2) ** (2 + 3) * (3 - 4) * 2
```

```
= (4/2) ** 5* -1 * 2
```

```
= (2**5)* -2
```

```
= 32* -2
```

```
= -64.
```

Of course, he could specify the latter order of evaluation by using parentheses:

$X \ \boxed{\ } \ ((((A/B) ** (C + D)) * (E - A)) * C).$

To fix the order of evaluation, we assign to each operator a priority. Then within any pair of parentheses we understand that operators with the highest priority will be evaluated first. A set of sample priorities from PL/I is given in Figure 3.7.

$\boxed{\ }$

Figure 3.7. Priority of arithmetic, Boolean and relational operators

Notice that all of the relational operators have the same priority. Similarly, exponentiation, unary minus, unary plus and Boolean negation all have top priority. When we have an expression where two adjacent operators have the same priority, we need a rule to tell us which one to perform first. For example, do we want the value of $- A ** B$ to be understood as $(-A) ** B$ or $-(A ** B)$? Convince yourself that there will be a difference by trying $A = -1$ and $B = 2$. From algebra we normally consider $A ** B ** C$ as $A ** (B ** C)$ and so we rule that operators in priority 6 are evaluated right-to-left. However, for expressions such as $A * B/C$ we generally execute left-to-right or $(A * B)/C$. So we rule that for all other priorities,

evaluation of operators of the same priority will proceed left to right. Remember that by using parentheses we can override these rules, and such expressions are always evaluated with the innermost parenthesized expression first.

Now that we have specified priorities and rules for breaking ties we know how $X \square A/B$ ** $C + D * E - A * C$ will be evaluated, namely as

$X \square ((A/(B** C)) + (D* E)) - (A * C)$.

How can a compiler accept such an expression and produce correct code? The answer is given by reworking the expression into a form we call postfix notation. If e is an expression with operators and operands, the conventional way of writing e is called *infix*, because the operators come *in*-between the operands. (Unary operators precede their operand.) The *postfix* form of an expression calls for each operator to appear *after* its operands. For example,

infix: $A * B/C$ has postfix: $AB * C/$.

If we study the postfix form of $A * B/C$ we see that the multiplication comes immediately after its two operands A and B. Now imagine that $A * B$ is computed and stored in T. Then we have the division operator, $/$, coming immediately after its two arguments T and C.

Let us look at our previous example

infix: $A/B ** C + D * E - A * C$

postfix: $ABC ** /DE * + AC * -$

and trace out the meaning of the postfix.

Every time we compute a value let us store it in the temporary location T_i, $i \square 1$. Reading left to right, the first operation is exponentiation:

```
Operation          Postfix

---------          -------

T₁ □  B **C        AT₁/DE * + AC * -

T₂ □  A/T₁         T₂DE * + AC * -
```

T_3 ☐ D * E $\qquad$ $T_2 T_3$ + AC * -

T_4 ☐ T_2 + T_3 $\quad$ T_4AC * -

T_5 ☐ A * C $\qquad$ $T_4 T_5$ -

T_6 ☐ T_4, - T_5 $\quad$ T_6

So T_6 will contain the result. Notice that if we had parenthesized the expression, this would change the postfix only if the order of normal evaluation were altered. Thus, $A / (B ** C) + (D * E) - A * C$ will have the same postfix form as the previous expression without parentheses. But $(A / B) ** (C + D) * (E - A) * C$ will have the postfix form $AB / CD + ** EA - * C *$.

Before attempting an algorithm to translate expressions from infix to postfix notation, let us make some observations regarding the virtues of postfix notation that enable easy evaluation of expressions. To begin with, the need for parentheses is elimlnated. Secondly, the priority of the operators is no longer relevant. The expression may be evaluated by making a left to right scan, stacking operands, and evaluating operators using as operands the correct number from the stack and finally placing the result onto the stack. This evaluation process is much simpler than attempting a direct evaluation from infix notation.

procedure *EVAL (E)*

//evaluate the postfix expression *E*. It is assumed that the

last character in *E* is an '∞'. A procedure *NEXT-TOKEN* is

used to extract from *E* the next token. A token is either a

operand, operator, or '∞'. A one dimensional array *STACK(l:n)* is

used as a stack//

top ☐ 0 // initialize *STACK*//

loop

x ☐ *NEXT-TOKEN (E)*

case

: *x* = '⊕' : **return**//answer is at top of stack//

: *x* is an operand: **call** *ADD(x,STACK,n,top)*

:else: *remove the correct number of operands*

for operator x from STACK, perform

the operation and store the result, if

any, onto the stack

end

forever

end *EVAL*

To see how to devise an algorithm for translating from infix to postfix, note that the order of the operands in both forms is the same. In fact, it is simple to describe an algorithm for producing postfix from infix:

1) fully parenthesize the expression;

2) move all operators so that they replace their corresponding right parentheses;

3) delete all parentheses.

For example, *A/B* ** *C* + *D* * *E* - *A* * *C* when fully parenthesized yields

The arrows point from an operator to its corresponding right parenthesis. Performing steps 2 and 3 gives

ABC ** / *DE* * + *AC* * -.

The problem with this as an algorithm is that it requires two passes: the first one reads the expression and parenthesizes it while the second actually moves the operators. As we have already observed, the order of the operands is the same in infix and postfix. So as we scan an expression for the first time, we

can form the postfix by immediately passing any operands to the output. Then it is just a matter of handling the operators. The solution is to store them in a stack until just the right moment and then to unstack and pass them to the output.

For example, since we want $A + B * C$ to yield $ABC * +$ our algorithm should perform the following sequence of stacking (these stacks will grow to the right):

```
Next Token   Stack   Output

----------   -----   ------

   none      empty    none

    A        empty     A

    +          +       A

    B          +       AB
```

At this point the algorithm must determine if * gets placed on top of the stack or if the + gets taken off. Since * has greater priority we should stack * producing

```
    *         + *       AB

    C         + *       ABC
```

Now the input expression is exhausted, so we output all remaining operators in the stack to get

ABC * +

For another example, $A * (B + C) * D$ has the postfix form $ABC + * D *$, and so the algorithm should behave as

```
Next Token   Stack   Output

----------   -----   ------

none         empty   none

A            empty   A
```

```
*               *        A

(               *(       A

B               *(       AB

+               *(+      AB

C               *(+      ABC
```

At this point we want to unstack down to the corresponding left parenthesis, and then delete the left and right parentheses; this gives us:

```
)               *        ABC +

*               *        ABC + *

D               *        ABC + * D

done            empty    ABC + * D *
```

These examples should motivate the following hierarchy scheme for binary arithmetic operators and delimiters. The general case involving all the operators of figure 3.7 is left as an exercise.

```
Symbol      In-Stack Priority   In-Coming Priority

------      ----------------    ------------------

)           -                   -

**          3                   4

*,/         2                   2

binary +,-  1                   1

(           0                   4
```

Figure 3.8 Priorities of Operators for Producing Postfix

The rule will be that *operators are taken out of the stack as long as their in-stack priority, isp, is greater*

than or equal to the in-coming priority, icp of the new operator. ISP(X) and ICP(X) are functions which reflect the table of figure 3.8.

procedure *POSTFIX* (*E*)

//convert the infix expression *E* to postfix. Assume the last

character of *E* is a '∞', which will also be the last character of

the postfix. Procedure *NEXT-TOKEN* returns either the next

operator, operand or delimiter--whichever comes next.

STACK (1:*n*) is used as a stack and the character '-∞' with

ISP('-∞') = -1 is used at the bottom of the stack. ISP and ICP

are functions.//

STACK(1) ☐ '-∞'; *top* ☐ 1   //initialize stack//

loop

x ☐ *NEXT-TOKEN*(*E*)

case

:*x* = '∞': **while** *top* > 1 **do** //empty the stack//

print(*STACK*(*top*));*top* ☐ *top* - 1

end

print ('∞')

return

:*x* is an operand: **print** (*x*)

:*x* = ')': **while** *STACK*(*top*) ≠ '(' **do**// unstack until '('//

```
print (STACK(top));top ☐ top - 1

end

top ☐ top - 1        //delete')'//

:else while ISP(STACK(top)) ☐ ICP(x) do

print (STACK(top)); top ☐ top - 1

end

call ADD(x,STACK,n,top)         //insert x in STACK//

end

forever

end POSTFIX
```

As for the computing time, the algorithm makes only one pass across the input. If the expression has *n* symbols, then the number of operations is proportional to some constant times *n*. The stack cannot get any deeper than the number of operators plus 1, but it may achieve that bound as it does for $A + B * C ** D$.

3.4 MULTIPLE STACKS AND QUEUES

Up to now we have been concerned only with the representation of a single stack or a single queue in the memory of a computer. For these two cases we have seen efficient sequential data representations. What happens when a data representation is needed for several stacks and queues? Let us once again limit ourselves, to sequential mappings of these data objects into an array $V(1:m)$. If we have only 2 stacks to represent. then the solution is simple. We can use $V(1)$ for the bottom most element in stack 1 and $V(m)$ for the corresponding element in stack 2. Stack 1 can grow towards $V(m)$ and stack 2 towards $V(1)$. It is therefore possible to utilize efficiently all the available space. Can we do the same when more than 2 stacks are to be represented? The answer is no, because a one dimensional array has only two fixed points $V(1)$ and $V(m)$ and each stack requires a fixed point for its bottommost element. When more than two stacks, say *n*, are to be represented sequentially, we can initially divide out the available memory $V(1:m)$ into *n* segments and allocate one of these segments to each of the *n* stacks. This initial division of $V(1:m)$ into segments may be done in proportion to expected sizes of the various stacks if the sizes are

known. In the absence of such information, $V(1:m)$ may be divided into equal segments. For each stack i we shall use $B(i)$ to represent a position one less than the position in V for the bottommost element of that stack. $T(i)$, $1 \leq i \leq n$ will point to the topmost element of stack i. We shall use the boundary condition $B(i) = T(i)$ iff the i'th stack is empty. If we grow the i'th stack in lower memory indexes than the $i + 1$'st, then with roughly equal initial segments we have

$$B(i) = T(i) = \lfloor m/n \rfloor (i - 1), \ 1 \leq i \leq n$$

(3.2)

as the initial values of $B(i)$ and $T(i)$, (see figure 3.9). Stack i, $1 \leq i \leq n$ can grow from $B(i) + 1$ up to $B(i + 1)$ before it catches up with the $i + 1$'st stack. It is convenient both for the discussion and the algorithms to define $B(n + 1) = m$. Using this scheme the add and delete algorithms become:

```
procedure ADD(i,X)

//add element X to the i'th stack, 1 ≤ i ≤ n//

if T(i) = B(i + 1) then call STACK-FULL (i)

T(i) ☐ T(i) + 1

V(T(i)) ☐ X          //add X to the i'th stack//

end ADD

procedure DELETE(i,X)

//delete topmost element of stack i//

if T(i) = B(i) then call STACK-EMPTY(i)

X ☐ V(T(i))

T(i) ☐ T(i) - 1

end DELETE
```

The algorithms to add and delete appear to be a simple as in the case of only 1 or 2 stacks. This really is not the case since the STACK_FULL condition in algorithm ADD does not imply that all m locations of

V are in use. In fact, there may be a lot of unused space between stacks j and $j + 1$ for $1 \le j \le n$ and $j \ne i$ (figure 3.10). The procedure STACK_FULL (i) should therefore determine whether there is any free space in V and shift stacks around so as to make some of this free space available to the i'th stack.

Several strategies are possible for the design of algorithm STACK_FULL. We shall discuss one strategy in the text and look at some others in the exercises. The primary objective of algorithm STACK_FULL is to permit the adding of elements to stacks so long as there is some free space in V. One way to guarantee this is to design STACK_FULL along the following lines:

a) determine the least j, $i < j \le n$ such that there is free space between stacks j and $j + 1$, i.e., $T(j) < B(j + 1)$. If there is such a j, then move stacks $i + 1$, $i + 2$, ...,j one position to the right (treating $V(1)$ as leftmost and $V(m)$ as rightmost), thereby creating a space between stacks i and $i + 1$.

b) if there is no j as in a), then look to the left of stack i. Find the largest j such that $1 \le j < i$ and there is space between stacks j and $j + 1$, i.e., $T(j) < B(j + 1)$. If there is such a j, then move stacks $j + 1$, $j + 2$, ...,i one space left creating a free space between stacks i and $i + 1$.

c) if there is no j satisfying either the conditions of a) or b), then all m spaces of V are utilized and there is no free space.

Figure 3.9 Initial configuration for n stacks in V(1:m). All stacks are empty and memory is divided into roughly equal segments.

Figure 3.10 Configuration when stack i meets with stack i + 1 but there is still free space elsewhere in V.

The writing of algorithm STACK_FULL using the above strategy is left as an exercise. It should be clear that the worst case performance of this representation for the n stacks together with the above strategy for STACK_FULL would be rather poor. In fact, in the worst case $O(m)$ time may be needed for each insertion (see exercises). In the next chapter we shall see that if we do not limit ourselves to sequential mappings of data objects into arrays, then we can obtain a data representation for m stacks that has a much better worst case performance than the representation described here. Sequential representations for n queues and other generalizations are discussed in the exercises.

EXERCISES

1. Consider a railroad switching network as below

Railroad cars numbered 1,2,3 ...,*n* are at the right. Each car is brought into the stack and removed at any time. For instance, if *n* = 3, we could move 1 in, move 2 in, move 3 in and then take the cars out producing the new order 3,2,1. For *n* = 3 and 4 what are the possible permutations of the cars that can be obtained? Are any permutations not possible?

2. Using a Boolean variable to distinguish between a circular queue being empty or full, write insert and delete procedures.

3. Complete the correctness proof for the stack implementation of section 3.1.

4. Use the queue axioms to prove that the circular queue representation of section 3.1 is correct.

5. A double ended queue (deque) is a linear list in which additions and deletions may be made at either end. Obtain a data representation mapping a deque into a one dimensional array. Write algorithms to add and delete elements from either end of the deque.

6. [Mystery function] Let *f* be an operation whose argument and result is a queue and which is defined by the axioms:

```
f(CREATEQ)    :: = CREATEQ

f(ADDQ(i,q)) :: = if ISEMTQ(q) then ADDQ(i,q)

else ADDQ(FRONT(q), f(DELETEQ(ADDQ(i,q))))
```

what does f do?

7. A linear list is being maintained circularly in an array *C*(0: *n* - 1) with *F* and *R* set up as for circular queues.

a) Obtain a formula in terms of *F*, *R* and *n* for the number of elements in the list.

b) Write an algorithm to delete the *k*'th element in the list.

c) Write an algorithm to insert an element *Y* immediately after the *k*'th element .

What is the time complexity of your algorithms for b) and c)?

8. Let $L = (a_1, a_2, ..., a_n)$ be a linear list represented in the array $V(1:n)$ using the mapping: the i'th element of L is stored in $V(i)$. Write an algorithm to make an inplace reversal of the order of elements in V. I.e., the algorithm should transform V such that $V(i)$ contains the $n - i + 1$'st element of L. The only additional space available to your algorithm is that for simple variables. The input to the algorithm is V and n. How much time does your algorithm take to accomplish the reversal?

9. a) Find a path through the maze of figure 3.5.

b) Trace out the action of procedure PATH on the maze of figure 3.5. Compare this to your own attempt in a).

10. What is the maximum path length from start to finish in any maze of dimensions n X m?

11. Write the postfix form of the following expressions:

a) $A ** B ** C$

b) $-A + B - C + D$

c) $A ** -B + C$

d) $(A + B) * D + E/(F + A * D) + C$

12. Obtain isp and icp priorities for all the operators of figure 3.7 together with the delimiters '(',and')'. These priorities should be such that algorithm POSTFIX correctly generates the postfix form for all expressions made up of operands and these operators and delimiters.

13. Use the isp and icp priorities obtained in exercise 12 to answer the following:

a) In algorithm POSTFIX what is the maximum number of elements that can be on the stack at any time if the input expression E has n operators and delimiters?

b) What is the answer to a) if E contains no operators of priority 6, has n operators and the depth of nesting of parentheses is at most 6?

14. Another expression form that is easy to evaluate and is parenthesis free is known as *prefix*. In this way of writing expressions, the operators precede their operands. For example:

```
        infix                        prefix
```

_____

```
     A * B/C                       /* ABC

A/B ** C + D * E - A * C  - +/A ** BC * DE * AC

A * (B + C)/D - G                -/* A + BCDG
```

Notice that the order of operands is not changed in going from infix to prefix .

a) What is the prefix form of the expressions in exercise 11.

b) Write an algorithm to evaluate a prefix expression, E (Hint: Scan E right to left and assume that the leftmost token of E is '∞'.)

c) Write an algorithm to transform an infix expression E into its prefix equivalent. Assume that the input expression E begins with a '∞' and that the prefix expression should begin with a '∞'.

What is the time complexity of your algorithms for b) and c)? How much space is needed by each of these algorithms?

15. Write an algorithm to transform from prefix to postfix. Carefully state any assumptions you make regarding the input. How much time and space does your algorithm take?

16. Do exercise 15 but this time for a transformation from postfix to prefix.

17. Write an algorithm to generate fully parenthesized infix expressions from their postfix form. What is the complexity (time and space) of your algorithm?

18. Do exercise 17 starting from prefix form.

19. Two stacks are to be represented in an array $V(1:m)$ as described in section 3.4. Write algorithms ADD($i.X$) and DELETE(i) to add X and delete an element from stack i, $1 \le i \le 2$. Your algorithms should be able to add elements to the stacks so long as there are fewer than m elements in both stacks together.

20. Obtain a data representation mapping a stack S and a queue Q into a single array $V(1:n)$. Write algorithms to add and delete elements from these two data objects. What can you say about the suitability of your data representation?

21. Write a SPARKS algorithm implementing the strategy for STACK_FULL(i) outlined in section 3.4.

22. For the ADD and DELETE algorithms of section 3.4 and the STACK_FULL(i) algorithm of exercise 21 produce a sequence of adds and deletes that will require $O(m)$ time for each add. Use $n = 2$ and start from a configuration representing a full utilization of $V(1:m)$.

23. It has been empirically observed that most programs that get close to using all available space eventually run out of space. In the light of this observation, it seems futile to move stacks around providing space for other stacks to grow in if there is only a limited amount of space that is free. Rewrite the algorithm of exercise 21 so that the algorithm terminates if there are fewer than C free spaces. C is an empirically determined constant and is provided to the algorithm.

24. Another strategy for the STACK_FULL(i) condition of section 3.4 is to redistribute all the free space in proportion to the rate of growth of individual stacks since the last call to STACK_FULL. This would require the use of another array $LT(1:n)$ where $LT(j)$ is the value of $T(j)$ at the last call to STACK_FULL. Then the amount by which each stack has grown since the last call is $T(j) - LT(j)$. The figure for stack i is actually $T(i) - LT(i) + 1$, since we are now attempting to add another element to i.

Write algorithm STACK_FULL (i) to redistribute all the stacks so that the free space between stacks j and $j + 1$ is in proportion to the growth of stack j since the last call to STACK_FULL. STACK_FULL (i) should assign at least 1 free location to stack i.

25. Design a data representation sequentially mapping n queues into all array $V(1:m)$. Represent each queue as a circular queue within V. Write algorithms ADDQ, DELETEQ and QUEUE-FULL for this representation.

26. Design a data representation, sequentially mapping n data objects into an array $V(1:m)$. n_1 of these data objects are stacks and the remaining $n_2 = n - n_1$ are queues. Write algorithms to add and delete elements from these objects. Use the same SPACE_FULL algorithm for both types of data objects. This algorithm should provide space for the i-th data object if there is some space not currently being used. Note that a circular queue with space for r elements can hold only $r - 1$.

27. [Landweber]

People have spent so much time playing card games of solitaire that the gambling casinos are now capitalizing on this human weakness. A form of solitaire is described below. Your assignment is to write a computer program to play the game thus freeing hours of time for people to return to more useful endeavors.

To begin the game, 28 cards are dealt into 7 piles. The leftmost pile has 1 card, the next two cards, and so forth up to 7 cards in the rightmost pile. Only the uppermost card of each of the 7 piles is turned face up. The cards are dealt left to right, one card to each pile, dealing to one less pile each time, and turning the first card in each round face up.

On the top-most face up card of each pile you may build in descending sequences red on black or black on red. For example, on the 9 of spades you may place either the 8 of diamonds or the 8 of hearts. All face up cards on a pile are moved as a unit and may be placed on another pile according to the bottommost face up card. For example, the 7 of clubs on the 8 of hearts may be moved as a unit onto the 9 of clubs or the 9 of spades.

Whenever a face down card is uncovered, it is turned face up. If one pile is removed completely, a face-up King may be moved from a pile (together with all cards above it) or the top of the waste pile (see below)) into the vacated space. There are four output piles, one for each suit, and the object of the game is to get as many cards as possible into the output piles. Each time an Ace appears at the top of a pile or the top of the stack it is moved into the appropriate output pile. Cards are added to the output piles in sequence, the suit for each pile being determined by the Ace on the bottom.

From the rest of the deck, called the stock, cards are turned up one by one and placed face up on a waste pile. You may always play cards off the top of the waste pile, but only one at a time. Begin by moving a card from the stock to the top of the waste pile. If there is ever more than one possible play to be made, the following order must be observed:

i) Move a card from the top of a playing pile or from the top of the waste pile to an output pile. If the waste pile becomes empty, move a card from the stock to the waste pile.

ii) Move a card from the top of the waste pile to the leftmost playing pile to which it can be moved. If the waste pile becomes empty move a card from the stock to the waste pile.

iii) Find the leftmost playing pile which can be moved and place it on top of the leftmost playing pile to which it can be moved.

iv) Try i), ii) and iii) in sequence, restarting with i) whenever a move is made.

v) If no move is made via (i)-(iv) move a card from the stock to the waste pile and retry (i).

Only the topmost card of the playing piles or the waste pile may be played to an output pile. Once played on an output pile, a card may not be withdrawn to help elsewhere. The game is over when either

i) all the cards have been played to the output or

ii) the stock pile has been exhausted and no more cards can be moved

When played for money, the player pays the house $52 at the beginning and wins $5 for every card played to the output piles.

Write your program so that it will play several games, and determine your net winnings. Use a random number generator to shuffle the deck.

Output a complete record of two games in easily understood form. Include as output the number of games played and the net winning (+ or -).

CHAPTER 4: LINKED LISTS

4.1 SINGLY LINKED LISTS

In the previous chapters, we studied the representation of simple data structures using an array and a sequential mapping. These representations had the property that successive nodes of the data object were stored a fixed distance apart. Thus, (i) if the element a_{ij} of a table was stored at location L_{ij}, then $a_{i,j+1}$ was at the location $L_{ij} + c$ for some constant c; (ii) if the ith node in a queue was at location L_i, then the $i + 1$-st node was at location $L_i + c$ mod n for the circular representation; (iii) if the topmost node of a stack was at location L_T, then the node beneath it was at location $L_T - c$, etc. These sequential storage schemes proved adequate given the functions one wished to perform (access to an arbitrary node in a table, insertion or deletion of nodes within a stack or queue).

However when a sequential mapping is used for ordered lists, operations such as insertion and deletion of arbitrary elements become expensive. For example, consider the following list of all of the three letter English words ending in AT:

(BAT, CAT, EAT, FAT, HAT, JAT, LAT, MAT, OAT, PAT, RAT, SAT, TAT, VAT, WAT)

To make this list complete we naturally want to add the word GAT, which means gun or revolver. If we are using an array to keep this list, then the insertion of GAT will require us to move elements already in the list either one location higher or lower. We must either move HAT, JAT, LAT, ..., WAT or else move BAT, CAT, EAT and FAT. If we have to do many such insertions into the middle, then neither alternative is attractive because of the amount of data movement. On the other hand, suppose we decide to remove the word LAT which refers to the Latvian monetary unit. Then again, we have to move many elements so as to maintain the sequential representation of the list.

When our problem called for several ordered lists of varying sizes, sequential representation again proved to be inadequate. By storing each list in a different array of maximum size, storage may be wasted. By maintaining the lists in a single array a potentially large amount of data movement is needed. This was explicitly observed when we represented several stacks, queues, polynomials and matrices. All these data objects are examples of ordered lists. Polynomials are ordered by exponent while matrices are ordered by rows and columns. In this chapter we shall present an alternative representation for ordered lists which will reduce the time needed for arbitrary insertion and deletion.

An elegant solution to this problem of data movement in *sequential* representations is achieved by using *linked* representations. Unlike a sequential representation where successive items of a list are located a fixed distance apart, in a linked representation these items may be placed anywhere in memory. Another way of saying this is that in a sequential representation the order of elements is the same as in the

ordered list, while in a linked representation these two sequences need not be the same. To access elements in the list in the correct order, with each element we store the address or location of the next element in that list. Thus, associated with each data item in a linked representation is a pointer to the next item. This pointer is often referred to as a link. In general, a *node* is a collection of data, DATA1, ..., DATA*n* and links LINK1, ...,LINK*m*. Each item in a node is called a *field*. A field contains either a data item or a link.

Figure 4.1 shows how some of the nodes of the list we considered before may be represented in memory by using pointers. The elements of the list are stored in a one dimensional array called DATA. But the elements no longer occur in sequential order, BAT before CAT before EAT, etc. Instead we relax this restriction and allow them to appear anywhere in the array and in any order. In order to remind us of the real order, a second array, LINK, is added. The values in this array are pointers to elements in the DATA array. Since the list starts at DATA(8) = BAT, let us set a variable $F = 8$. LINK(8) has the value 3, which means it points to DATA(3) which contains CAT. The third element of the list is pointed at by LINK(3) which is EAT. By continuing in this way we can list all the words in the proper order.

Figure 4.1 Non-Sequential List Representation

We recognize that we have come to the end when LINK has a value of zero.

Some of the values of DATA and LINK are undefined such as DATA(2), LINK(2), DATA(5), LINK(5), etc. We shall ignore this for the moment.

It is customary to draw linked lists as an ordered sequence of nodes with links being represented by arrows as in figure 4.2. Notice that we do not explicitly put in the values of the pointers but simply draw arrows to indicate they are there. This is so that we reinforce in our own mind the facts that (i) the nodes do not actually reside in sequential locations, and that (ii) the locations of nodes may change on different runs. Therefore, when we write a program which works with lists, we almost never look for a specific address except when we test for zero.

Figure 4.2 Usual Way to Draw a Linked List

Let us now see why it is easier to make arbitrary insertions and deletions using a linked list rather than a sequential list. To insert the data item GAT between FAT and HAT the following steps are adequate:

(i) get a node which is currently unused; let its address be *X*;

(ii) set the DATA field of this node to GAT;

(iii) set the LINK field of *X* to point to the node after FAT which contains HAT;

(iv) set the LINK field of the node containing FAT to *X*.

Figure 4.3a shows how the arrays DATA and LINK will be changed after we insert GAT. Figure 4.3b shows how we can draw the insertion using our arrow notation. The new arrows are dashed. The important thing to notice is that when we insert GAT we do not have to move any other elements which are already in the list. We have overcome the need to move data at the expense of the storage needed for the second field, LINK. But we will see that this is not too severe a penalty.

Figure 4.3a Insert GAT Into DATA(5)

Figure 4.3b Insert Node GAT Into List

Now suppose we want to delete GAT from the list. All we need to do is find the element which immediately precedes GAT, which is FAT, and set LINK(9) to the position of HAT which is 1. Again, there is no need to move the data around. Even though the LINK field of GAT still contains a pointer to HAT, GAT is no longer in the list (see figure 4.4).

Figure 4.4 Delete GAT from List

From our brief discussion of linked lists we see that the following capabilities are needed to make linked representations possible:

(i) A means for dividing memory into nodes each having at least one link field;

(ii) A mechanism to determine which nodes are in use and which are free;

(iii) A mechanism to transfer nodes from the reserved pool to the free pool and vice-versa.

Though DATA and LINK look like conventional one dimensional arrays, it is not necessary to implement linked lists using them. For the time being let us assume that all free nodes are kept in a "black box" called the storage pool and that there exist subalgorithms:

(i) GETNODE(*X*) which provides in *X* a pointer to a free node but if no node is free, it prints an error message and stops;

(ii) RET(*X*) which returns node *X* to the storage pool.

In section 4.3 we shall see how to implement these primitives and also how the storage pool is maintained.

Example 4.1: Assume that each node has two fields DATA and LINK. The following algorithm creates a linked list with two nodes whose DATA fields are set to be the values 'MAT' and 'PAT' respectively. *T* is a pointer to the first node in this list.

procedure *CREATE2(T)*

call *GETNODE(I)*   //get an available node//

T ☐ *I; DATA(I)* ☐ *'MAT'*   //store information into the node//

call *GETNODE(I)*   //get a second available node//

LINK(T) ☐ *I*   // attach first node to the second//

LINK(I) ☐ *0: DATA(I)* ☐ *'PAT'*

end *CREATE2*

The resulting list structure is

☐

Example 4.2: Let *T* be a pointer to a linked list as in Example 4.1. *T*= 0 if the list has no nodes. Let *X* be a pointer to some arbitrary node in the list *T*. The following algorithm inserts a node with DATA field 'OAT' following the node pointed at by *X*.

procedure *INSERT(T,X)*

call *GETNODE(I)*

DATA(I) ☐ *'OAT'*

if *T* = 0 **then** [*T* ☐ I; LINK(I) ☐ 0] //insert into empty list//

else [*LINK(I)* ☐ *LINK(X)*   //insert after *X*//

LINK(X) ☐ *I*]

end *INSERT*

The resulting list structure for the two cases $T = 0$ and $T \neq 0$ is

☐

Example 4.3: Let *X* be a pointer to some node in a linked list *T* as in example 4.2. Let *Y* be the node preceding *X*. $Y = 0$ if *X* is the first node in *T* (i.e., if $X = T$). The following algorithm deletes node *X* from *T*.

procedure *DELETE(X, Y, T)*

if *Y* = 0 **then** *T* ☐ LINK(T)   *//remove the first node//*

else *LINK(Y)* ☐ LINK(X)   *//remove an interior*

node//

call *RET(X)*   //return node to storage pool//

end *DELETE*

4.2 LINKED STACKS AND QUEUES

We have already seen how to represent stacks and queues sequentially. Such a representation proved efficient if we had only one stack or one queue. However, when several stacks and queues co-exist, there was no efficient way to represent them sequentially. In this section we present a good solution to this problem using linked lists. Figure 4.5 shows a linked stack and a linked queue.

☐

Figure 4.5

Notice that the direction of links for both the stack and queue are such as to facilitate easy insertion and deletion of nodes. In the case of figure 4.5(a), one can easily add a node at the top or delete one from the top. In figure 4.5(b), one can easily add a node at the rear and both addition and deletion can be performed at the front, though for a queue we normally would not wish to add nodes at the front. If we wish to represent n stacks and m queues simultaneously, then the following set of algorithms and initial conditions will serve our purpose:

```
T(i) = Top of ith stack     1 ≤ i ≤ n

F(i) = Front of ith queue  1 ≤ i ≤ m

R(i) = Rear of ith queue    1 ≤ i ≤ m
```

Initial conditions:

```
T(i) = 0                          1 ≤ i ≤ n

F(i) = 0                          1 ≤ i ≤ m
```

Boundary conditions:

```
T(i) = 0      iff stack i empty

F(i) = 0      iff queue i empty
```

procedure *ADDS(i, Y)*

```
//add element Y onto stack i//
```

call *GETNODE(X)*

DATA(X) ☐ *Y*   //store data value Y into new node//

LINK(X) ☐ *T(i)*   //attach new node to top of i-th stack//

T(i) ☐ *X*   //reset stack pointer//

end *ADDS*

procedure *DELETES(i, Y)*

//delete top node from stack *i* and set *Y* to be the *DATA* field of

this node//

if *T(i)* = 0 **then call** *STACK__EMPTY*

X ☐ *T(i)*   //set *X* to top node of stack *i*//

Y ☐ *DATA(X)*   //*Y* gets new data//

T(i) ☐ *LINK(X)*   //remove node from top of stack *i*//

call *RET(X)*   //return node to storage pool//

end *DELETES*

procedure *ADDQ(i,Y)*

//add *Y* to the *i*th queue//

call *GETNODE(X)*

DATA(X) ☐ *Y: LINK(X)* ☐ 0

if *F(i)* = 0 **then** [*F(i)* ☐ *R(i)* ☐ *X*]   //the queue was empty//

else [*LlNK(R(i))* ☐ *X;R(i)* ☐ *X*]   //the queue was

not empty//

end *ADDQ*

procedure *DELETEQ(i, Y)*

//delete the first node in the *i*th queue, set *Y* to its *DATA* field//

if *F(i)* = 0 **then** call *QUEUE__EMPTY*

else [X ☐ $F(i)$; $F(i)$ ☐ $LINK(X)$

//set X to front node//

Y ☐ $DATA(X)$; call $RET(X)$]   //remove data

and return node//

end $DELETEQ$

The solution presented above to the *n*-stack, *m*-queue problem is seen to be both computationaliy and conceptually simple. There is no need to shift stacks or queues around to make space. Computation can proceed so long as there are free nodes. Though additional space is needed for the link fields, the cost is no more than a factor of 2. Sometimes the DATA field does not use the whole word and it is possible to pack the LINK and DATA fields into the same word. In such a case the storage requirements for sequential and linked representations would be the same. For the use of linked lists to make sense, the overhead incurred by the storage for the links must be overriden by: (i) the virtue of being able to represent complex lists all within the same array, and (ii) the computing time for manipulating the lists is less than for sequential representation.

Now all that remains is to explain how we might implement the GETNODE and RET procedures.

4.3 THE STORAGE POOL

The storage pool contains all nodes that are not currently being used. So far we have assumed the existence of a RET and a GETNODE procedure which return and remove nodes to and from the pool. In this section we will talk about the implementation of these procedures.

The first problem to be solved when using linked allocation is exactly how a node is to be constructed The number ,and size of the data fields will depend upon the kind of problem one has. The number of pointers will depend upon the structural properties of the data and the operations to be performed. The amount of space to be allocated for each field depends partly on the problem and partly on the addressing characteristics of the machine. The packing and retrieving of information in a single consecutive slice of memory is discussed in section 4.12. For now we will assume that for each field there is a function which can be used to either retrieve the data from that field or store data into the field of a given node.

The next major consideration is whether or not any nodes will ever be returned. In general, we assume we want to construct an arbitrary number of items each of arbitrary size. In that case, whenever some structure is no longer needed, we will "erase" it, returning whatever nodes we can to the available pool. However, some problems are not so general. Instead the problem may call for reading in some data,

examining the data and printing some results without ever changing the initial information. In this case a linked structure may be desirable so as to prevent the wasting of space. Since there will never be any returning of nodes there is no need for a RET procedure and we might just as well allocate storage in consecutive order. Thus, if the storage pool has *n* nodes with fields DATA and LINK, then GETNODE could be implemented as follows:

procedure *GETNODE(I)*

//*I* is set as a pointer to the next available node//

if *AV* > *n* **then call** *NO__MORE__NODES*

I ☐ *AV*

AV ☐ *AV* + 1

end *GETNODE*

The variable *AV* must initially be set to one and we will assume it is a global variable. In section 4.7 we will see a problem where this type of a routine for GETNODE is used.

Now let us handle the more general case. The main idea is to initially link together all of the available nodes in a single list we call *AV*. This list will be singly linked where we choose any one of the possible link fields as the field through which the available nodes are linked. This must be done at the beginning of the program, using a procedure such as:

procedure *INIT(n)*

//initialize the storage pool, through the *LlNK* field, to contain nodes

with addresses 1,2,3, ...,*n* and set *AV* to point to the first node

in this list//

for *i* ☐ 1 **to** *n* - 1 **do**

LINK(i) ☐ *i* + 1

end

$LlNK(n) \square 0$

$AV \square 1$

end *INIT*

This procedure gives us the following list:

Figure 4.6 Initial Available Space List

Once INIT has been executed, the program can begin to use nodes. Every time a new node is needed, a call to the GETNODE procedure is made. GETNODE examines the list *AV* and returns the first node on the list. This is accomplished by the following:

procedure *GETNODE(X)*

//X is set to point to a free node if there is one on AV//

if *AV* = 0 **then call** *NO__MORE__NODES*

$X \square AV$

$AV \square LINK(AV)$

end *GETNODE*

Because *AV* must be used by several procedures we will assume it is a global variable. Whenever the programmer knows he can return a node he uses procedure RET which will insert the new node at the front of list *AV*. This makes RET efficient and implies that the list *AV* is used as a stack since the last node inserted into *AV* is the first node removed (LIFO).

procedure *RET(X)*

//X points to a node which is to be returned to the available space

list//

```
LINK(X) ☐ AV
```

```
AV ☐ X
```

end *RET*

If we look at the available space pool sometime in the middle of processing, adjacent nodes may no longer have consecutive addresses. Moreover, it is impossible to predict what the order of addresses will be. Suppose we have a variable ptr which is a pointer to a node which is part of a list called SAMPLE.

☐

What are the permissable operations that can be performed on a variable which is a pointer? One legal operation is to test for zero, assuming that is the representation of the empty list. (**if** *ptr* = 0 **then** ... is a correct use of *ptr*). An illegal operation would be to ask if *ptr* = 1 or to add one to *ptr* (*ptr* ☐ *ptr* + 1). These are illegal because we have no way of knowing what data was stored in what node. Therefore, we do not know what is stored either at node one or at node *ptr* + 1. In short, the only legal questions we can ask about a pointer variable is:

1) Is *ptr* = 0 (or is *ptr* ≠ 0)?

2) Is *ptr* equal to the value of another variable of type pointer, e.g., is *ptr* = SAMPLE?

The only legal operations we can perform on pointer variables is:

1) Set *ptr* to zero;

2) Set *ptr* to point to a node.

Any other form of arithmetic on pointers is incorrect. Thus, to move down the list SAMPLE and print its values we *cannot* write:

```
ptr ☐ SAMPLE
```

while *ptr* ≠ 5 **do**

print (DATA (*ptr*))

```
ptr ☐ ptr + 1
```

end

This may be confusing because when we begin a program the first values that are read in are stored sequentially. This is because the nodes we take off of the available space list are in the beginning at consecutive positions 1,2,3,4, ...,*max*. However, as soon as nodes are returned to the free list, subsequent items may no longer reside at consecutive addresses. A good program returns unused nodes to available space as soon as they are no longer needed. This free list is maintained as a stack and hence the most recently returned node will be the first to be newly allocated. (In some special cases it may make sense to add numbers to pointer variables, e.g., when *ptr + i* is the location of a field of a node starting at *ptr* or when nodes are being allocated sequentially, see sections 4.6 and 4.12).

4.4 POLYNOMIAL ADDITION

Let us tackle a reasonable size problem using linked lists. This problem, the manipulation of symbolic polynomials, has become a classical example of the use of list processing. As in chapter 2, we wish to be able to represent any number of different polynomials as long as their combined size does not exceed our block of memory. In general, we want to represent the polynomial

$$A(x) = a_m x^{e_m} + ... + a_1 x^{e_1}$$

where the a_i are non-zero coefficients with exponents e_i such that $e_m > e_{m-1} > ... > e_2 > e_1 >= 0$. Each term will be represented by a node. A node will be of fixed size having 3 fields which represent the coefficient and exponent of a term plus a pointer to the next term

```
     _____

     |       |       |        |

     |COEF   | EXP   | LINK   |

     |_____|_____|_____|
```
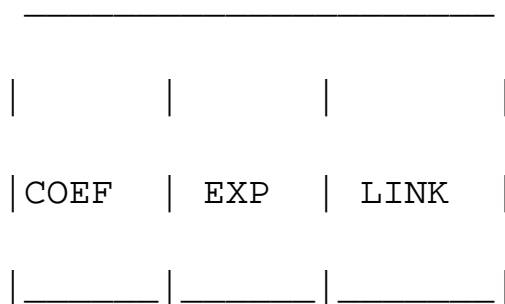
For instance, the polynomial $A = 3x^{14} + 2x^8 + 1$ would be stored as

while $B = 8x^{14} - 3x^{10} + 10x^6$ would look like

In order to add two polynomials together we examine their terms starting at the nodes pointed to by *A* and *B*. Two pointers *p* and *q* are used to move along the terms of *A* and *B*. If the exponents of two terms are equal, then the coefficients are added and a new term created for the result. If the exponent of the current term in *A* is less than the exponent of the current term of *B*, then a duplicate of the term of *B* is created and attached to *C*. The pointer *q* is advanced to the next term. Similar action is taken on *A* if EXP(*p*) > EXP(*q*). Figure 4.7 illustrates this addition process on the polynomials *A* and *B* above.

Figure 4.7 Generating the First Three Terms of C = A + B

Each time a new node is generated its COEF and EXP fields are set and it is appended to the end of the list *C*. In order to avoid having to search for the last node in *C* each time a new node is added, we keep a pointer *d* which points to the current last node in *C*. The complete addition algorithm is specified by the procedure PADD. PADD makes use of a subroutine ATTACH which creates a new node and appends it to the end of *C*. To make things work out neatly, *C* is initially given a single node with no values which is deleted at the end of the algorithm. Though this is somewhat inelegant, it avoids more computation. As long as its purpose is clearly documented, such a tactic is permissible.

```
procedure ATTACH(C,E,d)

//create a new term with COEF = C and EXP = E and attach it

to the node pointed at by d//

call GETNODE(I)

EXP(I) □ E

COEF(I) □ C

LINK(d) □ I        //attach this node to the end of this list//

d □ I       //move pointer d to the new last node//

end ATTACH
```

This is our first really complete example of the use of list processing, so it should be carefully studied. The basic algorithm is straightforward, using a merging process which streams along the two polynomials either copying terms or adding them to the result. Thus, the main while loop of lines 3-15 has 3 cases depending upon whether the next pair of exponents are =, <, or >. Notice that there are 5 places where a new term is created, justifying our use of the subroutine ATTACH.

Finally, some comments about the computing time of this algorithm. In order to carry out a computing time analysis it is first necessary to determine which operations contribute to the cost. For this algorithm there are several cost measures:

(i) coefficient additions;

(ii) coefficient comparisons;

(iii) additions/deletions to available space;

procedure *PADD(A,B,C)*

//polynomials A and B represented as singly linked lists are

summed to form the new list named C//

1 *p* ☐ *A; q* ☐ *B*   *//p,q pointers to next term of A, B//*

2   **call** *GETNODE(C); d* ☐ *C*   *//initial node for C, returned*

later//

3   **while** *p* ≠ 0 **and** *q* ≠ *0 do*   *//while there are more terms in*

A and *B//*

4   **case**

5   : *EXP(p) = EXP(q):*   *//equal exponents//*

6   *x* ☐ *COEF(p) + COEF(q)*

7   **if** *x* ≠ 0 **then call** *ATTACH(x, EXP(p),d)*

```
8           p ☐ LINK(p); q ☐ LINK(q)     //advance to next

terms//

9        : EXP(p) < EXP(q):

10             call ATTACH(COEF(q),EXP(q),d)

11          q ☐ LINK(q)      //advance to next term//

12        : else: call ATTACH(COEF(p),EXP(p),d)

13             p ☐ LINK(p)      //advance to next term of A//

14     end

15   end

16   while p ≠ 0 do     //copy remaining terms of A//

17     call ATTACH(COEF(p),EXP(p),d)

18     p ☐ LINK(p)

19   end

20   while q ≠ 0 do     //copy remaining terms of B//

21     call ATTACH(COEF(q),EXP(q),d)

22     q ☐ LINK(q)

23   end

24   LINK(d) ☐ 0; t ☐ C; C ☐ LINK(C)     //delete extra initial

node//

25   call RET(t)
```

26 **end** *PADD*

(iv) creation of new nodes for *C*.

Let us assume that each of these four operations, if done once, takes a single unit of time. The total time taken by algorithm PADD is then determined by the number of times these operations are performed. This number clearly depends on how many terms are present in the polynomials *A* and *B*. Assume that *A* and *B* have *m* and *n* terms respectively.

$$A(x) = a_m x^{em} + ... + a_1 x^{e1}, B(x) = b_n x^{fn} + ... + b_1 x^{f1}$$

where

$$a_i, b_i \neq 0 \text{ and } e_m > ... > e_1 \geq 0, f_n > ... > f_1 \geq 0.$$

Then clearly the number of coefficient additions can vary as

$$0 \leq \text{coefficient additions} \leq \min \{m, n\}.$$

The lower bound is achieved when none of the exponents are equal, while the upper bound is achieved when the exponents of one polynomial are a subset of the exponents of the other.

As for exponent comparisons, one comparison is made on each iteration of the **while** loop of lines 3-15. On each iteration either *p* or *q* or both move to the next term in their respective polynomials. Since the total number of terms is *m* + *n*, the number of iterations and hence the number of exponent comparisons is bounded by *m* + *n*. One can easily construct a case when *m* + *n* - 1 comparisons will be necessary: e.g. *m* = *n* and

$$e_m > f_n > e_{m-1} > f_{n-1} > ... > f_2 > e_{n-m+2} > ... > e_1 > f_1.$$

The maximum number of terms in *C* is *m* + *n*, and so no more than *m* + *n* new nodes are created (this excludes the additional node which is attached to the front of *C* and later returned). In summary then, the maximum number of executions of any of the statements in PADD is bounded above by *m* + *n*. Therefore, the computing time is $O(m + n)$. This means that if the algorithm is implemented and run on a computer, the time taken will be $c_1 m + c_2 n + c_3$ where c_1, c_2, c_3 are constants. Since any algorithm that adds two polynomials must look at each nonzero term at least once, every algorithm must have a time requirement of $c'_1 m + c'_2 n + c'3$. Hence, algorithm PADD is optimal to within a constant factor.

The use of linked lists is well suited to polynomial operations. We can easily imagine writing a collection of procedures for input, output addition, subtraction and multiplication of polynomials using

linked lists as the means of representation. A hypothetical user wishing to read in polynomials $A(x)$, $B(x)$ and $C(x)$ and then compute $D(x) = A(x) * B(x) + C(x)$ would write in his main program:

call *READ(A)*

call *READ(B)*

call *READ(C)*

T ☐ *PMUL(A, B)*

D ☐ *PADD(T, C)*

call *PRINT(D)*

Now our user may wish to continue computing more polynomials. At this point it would be useful to reclaim the nodes which are being used to represent T(x). This polynomial was created only as a partial result towards the answer D(x). By returning the nodes of T(x), they may be used to hold other polynomials.

procedure *ERASE(T)*

//return all the nodes of *T* to the available space list avoiding repeated

calls to procedure *RET*//

if *T* = 0 **then return**

p ☐ *T*

while *LINK (p)* ≠ 0 **do**   //find the end of *T*//

p ☐ *LINK (p)*

end

LlNK (p) ☐ *AV*   // *p* points to the last node of *T*//

```
AV  □  T        //available list now includes T//
```

end *ERASE*

Study this algorithm carefully. It cleverly avoids using the RET procedure to return the nodes of *T* one node at a time, but makes use of the fact that the nodes of *T* are already linked. The time required to erase $T(x)$ is still proportional to the number of nodes in *T*. This erasing of entire polynomials can be carried out even more efficiently by modifying the list structure so that the LINK field of the last node points back to the first node as in figure 4.8. A list in which the last node points back to the first will be termed a *circular list*. A *chain* is a singly linked list in which the last node has a zero link field.

□

Figure 4.8 Circular List Representation of A = 3x^{14} + 2x^8 + 1

Circular lists may be erased in a fixed amount of time independent of the number of nodes in the list. The algorithm below does this.

procedure *CERASE(T)*

```
//return the circular list T to the available pool//
```

if *T* = 0 **then return;**

```
X  □  LINK (T)
```

```
LINK(T)  □  AV
```

```
AV  □  X
```

end *CERASE*

Figure 4.9 is a schematic showing the link changes involved in erasing a circular list.

□

Figure 4.9 Dashes Indicate Changes Involved in Erasing a Circular List

A direct changeover to the structure of figure 4.8 however, causes some problems during addition, etc., as the zero polynomial has to be handled as a special case. To avoid such special cases one may

introduce a head node into each polynomial; i.e., each polynomial, zero or non-zero, will contain one additional node. The EXP and COEF fields of this node will not be relevant. Thus the zero polynomial will have the representation:

☐

while $A = 3x^{14} + 2x^8 + 1$ will have the representation

☐

For this circular list with head node representation the test for $T = 0$ may be removed from CERASE. The only changes to be made to algorithm PADD are:

(i) at line 1 define p, q by p ☐ $LINK$ (A); q ☐ $LINK(B)$

(ii) at line 3: **while** $p \neq A$ **and** $q \neq B$ **do**

(iii) at line 16: **while** $p \neq A$ **do**

(iv) at line 20: **while** $q \neq B$ **do**

(v) at line 24: replace this line by $LINK(d)$ ☐ C

(vi) delete line 25

Thus the algorithm stays essentially the same. Zero polynomials are now handled in the same way as nonzero polynomials.

A further simplification in the addition algorithm is possible if the EXP field of the head node is set to -1. Now when all nodes of A have been examined $p = A$ and $EXP(p) = -1$. Since $-1 \leq EXP(q)$ the remaining terms of B can be copied by further executions of the case statement. The same is true if all nodes of B are examined before those of A This implies that there is no need for additional code to copy the remaining terms as in PADD. The final algorithm takes the following simple form.

```
procedure CPADD (A, B, C)

//polynomials A and B are represented as circular lists with head

nodes so that EXP(A) = EXP(B) = -1. C is returned as their sum,

represented as a circular list//
```

```
p ← LINK (A); q ← LINK(B)

call GETNODE(C); EXP(C) ← -1          //set up head node//

d ← C                                 //last node in C//

loop

case

: EXP(p) = EXP(q): if EXP(p)=-1then [LINK(d)←C; return]

x ← COEF(p) + COEF(q)

if x ≠ 0 then call ATTACH(x,EXP(p),d

p ← LINK(p); q ← LINK(q)

: EXP(p) < EXP(q): call ATTACH(COEF(q), EXP(q),d)

q ← LINK(q)

: else : call ATTACH(COEF(p), EXP(p),d)

p ← LINK(p)

end

forever

end CPADD
```

Let us review what we have done so far. We have introduced the notion of a singly linked list. Each element on the list is a node of fixed size containing 2 or more fields one of which is a link field. To represent many lists all within the same block of storage we created a special list called the available space list or *AV*. This list contains all nodes which are cullently not in use. Since all insertions and deletions for *AV* are made at the front, what we really have is a linked list being used as a stack.

There is nothing sacred about the use of either singly linked lists or about the use of nodes with 2 fields. Our polynomial example used three fields: COEF, EXP and LINK. Also, it was convenient to use circular linked lists for the purpose of fast erasing. As we continue, we will see more problems which call for variations in node structure and representation because of the operations we want to perform.

4.5 MORE ON LINKED LISTS

It is often necessary and desirable to build a variety of routines for manipulating singly linked lists. Some that we have already seen are: 1) INIT which originally links together the *AV* list; 2) GETNODE and 3) RET which get and return nodes to *AV*. Another useful operation is one which inverts a chain. This routine is especially interesting because it can be done "in place" if we make use of 3 pointers.

procedure *INVERT(X)*

//a chain pointed at by *X* is inverted so that if $X = (a_1, \ldots, a_m)$

then after execution $X = (a_m, \ldots, a_1)$//

$p \leftarrow X; q \leftarrow 0$

while $p \neq 0$ **do**

$r \leftarrow q; q \leftarrow p$ //*r* follows *q*; *q* follows *p*//

$p \leftarrow LINK(p)$ //*p* moves to next node//

$LINK(q) \leftarrow r$ //link *q* to previous node//

end

$X \leftarrow q$

end *INVERT*

The reader should try this algorithm out on at least 3 examples: the empty list, and lists of length 1 and 2 to convince himself that he understands the mechanism. For a list of $m \geq 1$ nodes, the **while** loop is executed *m* times and so the computing time is linear or *O(m)*.

Another useful subroutine is one which concatenates two chains *X* and *Y*.

procedure *CONCATENATE(X, Y, Z)*

//X = (a_1, ...,a_m), Y = (b_1, ...,b_n), m,n $\geq$ 0, produces a new chain

Z = (a_1, ...,a_m,b_1 , ...,b_n)//

Z ← *X*

if *X* = 0 **then** [*Z* ← *Y*; **return**]

if *Y* = 0 **then return**

p ← *X*

while *LINK(p)* ≠ 0 **do**   *//find last node of X//*

p ← *LINK(p)*

end

LINK(p) ← *Y*   *//link last node of X to Y//*

end *CONCATENATE*

This algorithm is also linear in the length of the first list. From an aesthetic point of view it is nicer to write this procedure using the case statement in SPARKS. This would look like:

procedure *CONCATENATE(X, Y, Z)*

case

: *X = 0* :*Z* ← *Y*

: *Y = 0* : *Z* ← *X*

: **else** : *p* ← *X*; *Z* ← *X*

while *LINK(p)* ≠ 0 **do**

p □ *LINK (p)*

end

LINK(p) □ *Y*

end

end *CONCATENATE*

Now let us take another look at circular lists like the one below:

□

$A = (x_1, x_2, x_3)$. Suppose we want to insert a new node at the front of this list. We have to change the LINK field of the node containing x_3. This requires that we move down the entire length of *A* until we find the last node. It is more convenient if the name of a circular list points to the last node rather than the first, for example:

□

Now we can write procedures which insert a node at the front or at the rear of a circular list and take a fixed amount of time.

procedure *INSERT__FRONT(A, X)*

//insert the node pointed at by X to the front of the circular list

A, where A points to the last node//

if A = 0 **then** [A □ X

LINK (X) □ *A]*

else [*LINK(X)* □ *LINK (A)*

LINK(A) □ *X]*

end *INSERT--FRONT*

To insert *X* at the rear, one only needs to add the additional statement *A* ▢ *X* to the **else** clause of *INSERT_-_FRONT*.

As a last example of a simple procedure for circular lists, we write a function which determines the length of such a list.

procedure *LENGTH(A)*

```
//find the length of the circular list A//
```

i ▢ 0

if A ≠ 0 **then** [*ptr* ▢ A

repeat

i ▢ *i* + 1; *ptr* ▢ LINK(*ptr*)

until *ptr* = A]

return (*i*)

end *LENGTH*

4.6 EQUIVALENCE RELATIONS

Let us put together some of these ideas on linked and sequential representations to solve a problem which arises in the translation of computer languages, the processing of equivalence relations. In FORTRAN one is allowed to share the same storage among several program variables through the use of the EQUIVALENCE statement. For example, the following pair of Fortran statements

DIMENSION *A*(3), *B*(2,2), *C*(6)

EQUIVALENCE (*A*(2), *B*(1,2), *C*(4)), (*A*(1),*D*), (*D*,*E*,*F*), (*G*,*H*)

would result in the following storage assignment for these variables:

☐

As a result of the equivalencing of *A* (2), *B*(1,2) and *C*(4), these were assigned the same storage word 4. This in turn equivalenced *A*(1), *B*(2,1) and *C*(3), *B*(1,1) and *C*(2), and *A*(3), *B*(2,2) and *C*(5). Because of the previous equivalence group, the equivalence pair (*A*(1), *D*) also resulted in *D* sharing the same space as *B*(2,1) and *C*(3). Hence, an equivalence of the form (*D*, *C*(5)) would conflict with previous equivalences, since *C*(5) and *C*(3) cannot be assigned the same storage. Even though the sum of individual storage requirements for these variables is 18, the memory map shows that only 7 words are actually required because of the overlapping specified by the equivalence groups in the EQUIVALENCE statement. The functions to be performed during the processing of equivalence statements, then, are:

(i) determine whether there are any conflicts;

(ii) determine the total amount of storage required;

(iii) determine the relative address of all variables (i.e., the address of *A*(1), *B*(1,1), *C*(1), *D*, *E*, *F*, *G* and *H* in the above example).

In the text we shall solve a simplified version of this problem. The extension to the general Fortran equivalencing problem is fairly straight-forward and appears as an exercise. We shall restrict ourselves to the case in which only simple variables are being equivalenced and no arrays are allowed. For ease in processing, we shall assume that all equivalence groups are pairs of numbers (*i*,*j*), where if EQUIVALENCE(*A*,*F*) appears then *i*,*j* are the integers representing *A* and *F*. These can be thought of as the addresses of *A* and *F* in a symhol table. Furthermore, it is assumed that if there are *n* variables, then they ale represented by the numbers 1 to *n*.

The FORTRAN statement EQUIVALENCE specifies a relationship among addresses of variables. This relation has several properties which it shares with other relations such as the conventional mathematical equals. Suppose we denote an arbitrary relation by the symbol ☐ and suppose that:

(i) For any variable *x*, *x* ☐ *x*, *e.g.* *x* is to be assigned the same location as itself. Thus ☐ is *reflexive.*

(ii) For any two variables *x* and *y*, if *x* ☐ *y* then *y* ☐ x, e.g. *assigning* y *the same location as* x *is the same as assigning* x *the same location as* y. *Thus, the relation* ☐ *is* symmetric.

(iii) For any three variables *x*, *y* and *z*, if *x* ☐ y *and* y ☐ *z* then *x* ☐ z, *e.g. if* x *and* y *are to be assigned the same location and* y *and* z *are also to be assigned the same location, then so also are* x *and* z. *The relation* ☐ *is* transitive.

Definition: A relation, ▢, over a set *S*, is said to be an *equivalence relation* over *S* iff it is symmetric, reflexive and transitive over *S*.

Examples of equivalence relations are numerous. For example, the "equal to" (=) relationship is an equivalence relation since: (i) $x = x$, (ii) $x = y$ implies $y = x$, and (iii) $x = y$ and $y = z$ implies $x = z$. One effect of an equivalence relation is to partition the set *S* into equivalence classes such that two members *x* and *y* of *S* are in the same equivalence class iff x ▢ y. For example, if we have 12 variables numbered 1 through 12 and the following equivalences were defined via the EQUIVALENCE statement:

```
1 ▢ 5,  4 ▢ 2,  7 ▢ 11,  9 ▢ 10,  8 ▢ 5,  7 ▢ 9,  4 ▢ 6,  3 ▢ 12 and
12 ▢ 1
```

then, as a result of the reflexivity, symmetry and transitivity of the relation =, we get the following partitioning of the 12 variables into 3 equivalence classes:

{1, 3, 5, 8, 12}; {2, 4, 6}; {7, 9, 10, 11}.

So, only three words of storage are needed for the 12 variables. In order to solve the FORTRAN equivalence problem over simple variables, all one has to do is determine the equivalence classes. The number of equivalence classes ii the number of words of storage to be allocated and the members of the same equivalence class are allocated the same word of storage.

The algorithm to determine equivalence classes works in essentially two phases. In the first phase the equivalence pairs (i,j) are read in and stored somewhere. In phase two we begin at one and find all pairs of the form $(1,j)$. The values 1 and *j* are in the same class. By transitivity, all pairs of the form (j,k) imply *k* is in the same class. We continue in this way until the entire equivalence class containing one has been found, marked and printed. Then we continue on.

The first design for this algorithm might go this way:

procedure *EQUIVALENCE (m,n)*

initialize

for *k* ▢ 1 **to** m **do**

read *the next pair (i,j)*

process this pair

end

initialize for output

repeat

output a new equivalence class

until *done*

end *EQUIVALENCE*

The inputs *m* and *n* represent the number of related pairs and the number of objects respectively. Now we need to determine which data structure should be used to hold these pairs. To determine this we examine the operations that are required. The pair (*i*,*j*) is essentially two random integers in the range 1 to *n*. Easy random access would dictate an array, say PAIRS (1: *n*, 1: *m*). The *i*-th row would contain the elements *j* which are paired directly to *i* in the input. However, this would potentially be very wasteful of space since very few of the array elements would be used. It might also require considerable time to insert a new pair, (*i*,*k*), into row *i* since we would have to scan the row for the next free location or use more storage.

These considerations lead us to consider a linked list to represent each row. Each node on the list requires only a DATA and LINK field. However, we still need random access to the *i*-th row so a one dimensional array, SEQ(1:*n*) can be used as the headnodes of the *n* lists. Looking at the second phase of the algorithm we need a mechanism which tells us whether or not object *i* has already been printed. An array of bits, BIT(1:*n*) can be used for this. Now we have the next refinement of the algorithm.

procedure *EQUIVALENCE* (*m,n*)

declare *SEQ(1:n), DATA(1:2m), LINK(1:2m), BIT(1:n)*

initialize BIT, SEQ to zero

for *k* ☐ 1 **to** *m* **do**

read *the next pair (i,j)*

put j on the SEQ(i) list

put i on the SEQ(j) list

end

index □ 1

repeat

if *BIT(index)* = 0

then [*BIT(index)* □ 1

output this new equivalence class]

index □ *index* + 1

until *index* > *n*

end

Let us simulate the algorithm as we have it so far, on the previous data set. After the for **loop** is completed the lists will look like this.

□

For each relation *i* □ *j*, two nodes are used. SEQ(*i*) points to a list of nodes which contains every number which is directly equivalenced to *i* by an input relation.

In phase two we can scan the SEQ array and start with the first *i*, $1 \le i \le n$ such that BIT(*i*) = 0. Each element in the list SEQ(*i*) is printed. In order to process the remaining lists which, by transitivity, belong in the same class as *i* a stack of their nodes is created. This is accomplished by changing the LINK fields so they point in the reverse direction. The complete algorithm is now given.

procedure EQUIVALENCE (*m*,*n*)

//Input: *m*, the number of equivalence pairs

n, the number of variables

Output: variables 1, ...,*n* printed in equivalence classes//

declare *SEQ*(1:*n*), *BIT*(1:*n*)

```
DATA(1: 2m), LINK(1: 2m);

for i ☐ 1 to n do SEQ(i) ☐BIT(i) ☐ 0 end

av ☐1

for k ☐ 1 to m do        //phase 1: process all input//

read next equivalence pair (i, j)

DATA(av) ☐ j; LlNK(av) ☐ SEQ(i)      //add j to list i//

SEQ(i) ☐ av; av ☐ av + 1

DATA(av) ☐i; LINK(av) ☐ SEQ(j)      //add i to list j//

SEQ(j) ☐ av; av ☐ av + 1

end

index ☐ 1

repeat      //phase 2: output all classes//

if BIT (index) = 0

then [print ('A new class', index)

BIT(index) ☐ 1        //mark class as output//

ptr ☐ SEQ(index); top ☐ 0        //initialize stack//

loop        //find the entire class//

while ptr ≠ 0 do        //process a list//

j ☐ DATA (ptr)
```

```
if BIT(j) = 0

then [print (j); BIT(j) ☐ 1

t ☐ LINK (ptr); LINK (ptr) ☐ top

top ☐ ptr; ptr ☐ t]

else ptr ☐ LINK (ptr)

end

if top = 0 then exit      //stack empty//

ptr ☐ SEQ(DATA(top))

top ☐ LINK (top)

forever]

index ☐ index + 1

until index > n

end EQUIVALENCE
```

Analysis of Algorithm EQUIVALENCE

The initialization of SEQ and BIT takes $O(n)$ time. The processing of each input pair in phase 1 takes a constant amount of time. Hence, the total time for this phase is $O(m)$. In phase 2 each node is put onto the linked stack at most once. Since there are only $2m$ nodes and the **repeat** loop is executed n times, the time for this phase is $O(m + n)$. Hence, the overall computing time is $O(m + n)$. Any algorithm which processes equivalence relations must look at all the m equivalence pairs and also at all the n variables at least once. Thus, there can be no algorithm with a computing time less than $O(m + n)$. This means that the algorithm EQUIVALENCE is optimal to within a constant factor. Unfortunately,the space required by the algorithm is also $O(m + n)$. In chapter 5 we shall see an alternative solution to this problem which requires only $O(n)$ space.

4.7 SPARSE MATRICES

In Chapter 2, we saw that when matrices were sparse (i.e. many of the entries were zero), then much space and computing time could be saved if only the nonzero terms were retained explicitly. In the case where these nonzero terms did not form any "nice" pattern such as a triangle or a band, we devised a sequential scheme in which each nonzero term was represented by a node with three fields: row, column and value. These nodes were sequentially organized. However, as matrix operations such as addition, subtraction and multiplication are performed, the number of nonzero terms in matrices will vary, matrices representing partial computations (as in the case of polynomials) will be created and will have to be destroyed later on to make space for further matrices. Thus, sequential schemes for representing sparse matrices suffer from the same inadequacies as similar schemes for polynomials. In this section we shall study a very general linked list scheme for sparse matrix representation. As we have already seen, linked schemes facilitate efficient representation of varying size structures and here, too, our scheme will overcome the aforementioned shortcomings of the sequential representation studied in Chapter 2.

In the data representation we shall use, each column of a sparse matrix will be represented by a circularly linked list with a head node. In addition, each row will also be a circularly linked list with a head node. Each node in the structure other than a head node will represent a nonzero term in the matrix A and will be made up of five fields:

ROW, COL, DOWN, RIGHT and VALUE. The DOWN field will be used to link to the next nonzero element in the same column, while the RIGHT field will be used to link to the next nonzero element in the same ROW. Thus, if $a_{ij} \neq 0$, then there will be a node with VALUE field a_{ij}, ROW field i and COL field j. This node will be linked into the circular linked list for row i and also into the circular linked list for column j. It will, therefore, be a member of two lists at the same time .

In order to avoid having nodes of two different sizes in the system, we shall assume head nodes to be configured exactly as nodes being used to represent the nonzero terms of the sparse matrix. The ROW and COL fields of head nodes will be set to zero (i.e. we assume that the rows and columns of our matrices have indices >0). Figure 4.12 shows the structure obtained for the 6 x 7 sparse matrix, A, of figure 4.11.

Figure 4.10 Node Structure for Sparse Matrix Representation

Figure 4.11 6 x 7 Sparse Matrix A

For each nonzero term of A, we have one five field node which is in exactly one column list and one row

list. The head nodes are marked Hl-H7. As can be seen from the figure, the VALUE field of the head nodes for each column list is used to link to the next head node while the DOWN field links to the first nonzero term in that column (or to itself in case there is no nonzero term in that column). This leaves the RIGHT field unutilized. The head nodes for the row lists have the same ROW and COL values as the head nodes for the column lists. The only other field utilized by the row head nodes is the RIGHT field which is not used in the column head nodes. Hence it is possible to use the same node as the head node for row i as for column i. It is for this reason that the row and column head nodes have the same labels. The head nodes themselves, linked through the VALUE field, form a circularly linked list with a head node pointed to by A. This head node for the list of row and column head nodes contains the dimensions of the matrix. Thus, ROW(A) is the number of rows in the matrix A while COL(A) is the number of columns. As in the case of polynomials, all references to this matrix are made through the variable A. If we wish to represent an n x m sparse matrix with r nonzero terms, then the number of nodes needed is r + max $\{n,m\}$ + 1. While each node may require 2 to 3 words of memory (see section 4.12), the total storage needed will be less than nm for sufficiently small r.

Figure 4.12 Linked Representation of the Sparse Matrix A

Having arrived at this representation for sparse matrices, let us see how to manipulate it to perform efficiently some of the common operations on matrices. We shall present algorithms to read in a sparse matrix and set up its linked list representation and to erase a sparse matrix (i.e. to return all the nodes to the available space list). The algorithms will make use of the utility algorithm GETNODE(X) to get nodes from the available space list.

To begin with let us look at how to go about writing algorithm MREAD(A) to read in and create the sparse matrix A. We shall assume that the input consists of n, the number of rows of A, m the number of columns of A, and r the number of nonzero terms followed by r triples of the form (i,j,a_{ij}). These triples consist of the row, column and value of the nonzero terms of A. It is also assumed that the triples are ordered by rows and that within each row, the triples are ordered by columns. For example, the input for the 6 X 7 sparse matrix of figure 4.11, which has 7 nonzero terms, would take the form: 6,7,7; 1,3,11;1,6,13;2,1,12;2,7,14;3,2,-4;3,6,-8;6,2,-9. We shall not concern ourselves here with the actual format of this input on the input media (cards, disk, etc.) but shall just assume we have some mechanism to get the next triple (see the exercises for one possible input format). The algorithm MREAD will also make use of an auxiliary array HDNODE, which will be assumed to be at least as large as the largest dimensioned matrix to be input. HDNODE(i) will be a pointer to the head node for column i, and hence also for row i. This will permit us to efficiently access columns at random while setting up the input matrix. Algorithm MREAD proceeds by first setting up all the head nodes and then setting up each row list, simultaneously building the column lists. The VALUE field of headnode i is initially used to keep track of the last node in column i. Eventually, in line 27, the headnodes are linked together through this field.

Analysis of Algorithm MREAD

Since GETNODE works in a constant amount of time, all the head nodes may be set up in O(max $\{n, m\}$) time, where n is the number of rows and m the number of columns in the matrix being input. Each nonzero term can be set up in a constant amount of time because of the use of the variable LAST and a random access scheme for the bottommost node in each column list. Hence, the **for** loop of lines 9-20 can be carried out in $O(r)$ time. The rest of the algorithm takes O(max $\{n,m\}$) time. The total time is therefore O(max $\{n,m\} + r) = O(n + m + r)$. Note that this is asymptotically better than the input time of $O(nm)$ for an n x m matrix using a two-dimensional array, but slightly worse than the sequential sparse method of section 2.3.

Before closing this section, let us take a look at an algorithm to return all nodes of a sparse matrix to the available space list.

procedure *MERASE(A)*

//return all nodes of A to available space list. Assume that the available

space list is a singly linked list linked through the field RIGHT

with AV pointing to the first node in this list.//

RIGHT(A) ☐ *AV; AV* ☐ *A; NEXT* ☐ *VALUE(A)*

while *NEXT* ≠ *A* **do**   *//erase circular lists by rows//*

T ☐ *RIGHT(NEXT)*

RIGHT(NEXT) ☐ *AV*

AV ☐ *T*

NEXT ☐ *VALUE(NEXT)*

end

end *MERASE*

Analysis of Algorithm MERASE

Since each node is in exactly one row list, it is sufficient to just return all the row lists of the matrix A. Each row list is circularly linked through the field RIGHT. Thus, nodes need not be returned one by one as a circular list can be erased in a constant amount of time. The computing time for the algorithm is readily seen to be $O(n + m)$. Note that even if the available space list had been linked through the field DOWN, then erasing could still have been carried out in $O(n + m)$ time. The subject of manipulating these matrix structures is studied further in the exercises. The representation studied here is rather general. For most applications this generality is not needed. A simpler representation resulting in simpler algorithms is discussed in the exercises.

procedure *MREAD(A)*

//read in a matrix *A* and set up its internal representation as

discussed previously. The triples representing nonzero terms

are assumed ordered by rows and within rows by columns.

An auxiliary array HDNODE is used.//

1   **read** (*n,m,r*)   //*m, n* are assumed positive. *r* is the number

of nonzero elements//

2   *p* ☐ **max** {*m,n*}

3   **for** *i* ☐ 1 **to** *p* **do**   //get *p* headnodes for rows and

columns//

4   **call** GETNODE(X); HDNODE(i) ☐ X

5   ROW(X) ☐ COL(X) ☐ 0

6   RIGHT(X) ☐ VALUE(X) ☐ X   //these fields point to

themselves//

```
7    end

8    current_row ← 1; LAST ← HDNODE(1)

9    for i ← 1 to r do

10     read (rrow, ccol, val)     //get next triple//

11     if rrow > current__row     //current row is done; close it and

begin another//

12        then [RIGHT(LAST) ← HDNODE(current__row)

13             current__row ← rrow; LAST ← HDNODE(rrow)]

//LAST points to rightmost node//

14     call GETNODE(X)

15     ROW(X) ← rrow; COL(X) ← ccol; VALUE(X) ← val

//store triple into new node//

16     RIGHT(LAST) ← X; LAST ← X      //link into row list//

17     DOWN(VALUE(HDNODE(ccol))) ← X;

//link into column list//

18     VALUE(HDNODE(ccol)) ← X

19

20     end

21     //close last row//
```

if $r \neq 0$ **then** *RIGHT(LAST)* □ *HDNODE(current__row)*

22   **for** i □ 1 **to** m **do**   //close all column lists//

23   *DOWN(VALUE(HDNODE(i)))* *HDNODE(i)*

24   **end**

25   //set up list of headnodes linked through *VALUE* field//

26   **call** *GETNODE(A): ROW(A)* □ n; COL(A) □ m

//set up headnode of matrix//

27   **for** i □ 1 **to** p - 1 **do** *VALUE(HDNODE(i))* □ HDNODE(i + 1)

end

28   **if** p = 0 **then** *VALUE(A)* □ A

29   **else** [*VALUE(HDNODE(p))* □ A

VALUE(A) □ HDNODE(1)]

30 **end** *MREAD*

4.8 DOUBLY LINKED LISTS AND DYNAMIC STORAGE MANAGEMENT

So far we have been working chiefly with singly linked linear lists. For some problems these would be too restrictive. One difficulty with these lists is that if we are pointing to a specific node, say *P*, then we can easily move only in the direction of the links. The only way to find the node which precedes *P* is to start back at the beginning of the list. The same problem arises when one wishes to delete an arbitrary node from a singly linked list. As can be seen from example 4.3, in order to easily delete an arbitrary node one must know the preceding node. If we have a problem where moving in either direction is often necessary, then it is useful to have doubly linked lists. Each node now has two link fields, one linking in the forward direction and one in the backward direction.

A node in a doubly linked list has at least 3 fields, say DATA, LLINK (left link) and RLINK (right link). A doubly linked list may or may not be circular. A sample doubly linked circular list with 3 nodes is given in figure 4.13. Besides these three nodes a special node has been

Figure 4.13 Doubly Linked Circular List with Head Node

added called a head node. As was true in the earlier sections, head nodes are again convenient for the algorithms. The DATA field of the head node will not usually contain information. Now suppose that P points to any node in a doubly linked list. Then it is the case that

$P = \text{RLINK}(\text{LLINK}(P)) = \text{LLINK}(\text{RLINK}(P))$.

This formula reflects the essential virtue of this structure, namely, that one can go back and forth with equal ease. An empty list is not really empty since it will always have its head node and it will look like

Now to work with these lists we must be able to insert and delete nodes. Algorithm DDLETE deletes node X from list L.

```
procedure DDLETE(X, L)

if X = L then call NO__MORE__NODES

//L is a list with at least one node//

RLINK(LLINK(X)) ⬚ RLINK(X)

LLINK(RLINK(X)) ⬚ LLINK(X)

call RET(X)

end DDLETE
```

X now points to a node which is no longer part of list L. Let us see how the method works on a doubly linked list with only a single node.

Even though the RLINK and LLINK fields of node *X* still point to the head node, this node has effectively been removed as there is no way to access *X* through *L*.

Insertion is only slightly more complex.

procedure *DINSERT (P, X)*

//insert node P to the right of node X//

LLINK(P) ☐ *X*　　　　*//set LLINK and RLINK fields of node P//*

RLINK(P) ☐ *RLINK(X)*

LLINK(RLINK(X)) ☐ *P*

RLINK(X) ☐ *P*

end *DINSERT*

In the next section we will see an important problem from operating systems which is nicely solved by the use of doubly linked lists.

Dynamic Storage Management

In a multiprocessing computer environment, several programs reside in memory at the same time. Different programs have different memory requirements. Thus, one program may require 60K of memory, another 100K, and yet another program may require 300K. Whenever the operating system needs to request memory, it must be able to allocate a block of contiguous storage of the right size. When the execution of a program is complete, it releases or frees the memory block allocated to it and this freed block may now be allocated to another program. In a dynamic environment the request sizes that will be made are not known ahead of time. Moreover, blocks of memory will, in general, be freed in some order different from that in which they were allocated. At the start of the computer system no jobs are in memory and so the whole memory, say of size *M* words, is available for allocation to programs. Now, jobs are submitted to the computer and requests are made for variable size blocks of memory. Assume we start off with 100,000 words of memory and five programs *P*1, *P*2, *P*3, *P*4 and *P*5 make requests of size 10,000, 15,000, 6,000, 8,000 and 20,000 respectively. Figure 4.14 indicates the status of memory after storage for *P*5 has been allocated. The unshaded area indicates the memory that is currently not in use. Assume that programs *P*4 and *P*2 complete execution, freeing the memory used by them. Figure 4.15 show the status of memory after the blocks for *P*2 and *P*4 are freed. We now have three blocks of contiguous memory that are in use and another three that are free. In order to make

further allocations, it is necessary to keep track of those blocks that are not in use. This problem is similar to the one encountered in the previous sections where we had to maintain a list of all free nodes. The difference between the situation then and the one we have now is that the free space consists of variable size blocks or nodes and that a request for a block of memory may now require allocation of only a portion of a node rather than the whole node. One of the functions of an operating system is to maintain a list of all blocks of storage currently not in use and then to allocate storage from this unused pool as required. One can once again adopt the chain structure used earlier to maintain the available space list. Now, in addition to linking all the free blocks together, it is necessary to retain information regarding the size of each block in this list of free nodes. Thus, each node on the free list has two fields in its first word, i.e., SIZE and LINK. Figure 4.16 shows the free list corresponding to figure 4.15. The use of a head node simplifies later algorithms.

Figure 4.14 Memory After Allocation to P1-P5

Figure 4.15 Status of Memory After Completion of P2 and P4

If we now receive a request for a block of memory of size N, then it is necessary to search down the list of free blocks finding the first block of size $\geq N$ and allocating N words out of this block. Such an allocation strategy is called *first fit*. The algorithm below makes storage allocations using the first fit strategy. An alternate strategy, *best fit*, calls for finding a free block whose size is as close to N as possible, but not less than N. This strategy is examined in the exercises.

Figure 4.16 Free List with Head Node Corresponding to figure 4.15

```
procedure FF(n, p)

//AV points to the available space list which is searched for a node

of size at least n. p is set to the address of a block of size n

that may be allocated. If there is no block of that size then p = 0.

It is assumed that the free list has a head node with SIZE field = 0//

p ☐ LINK(AV); q ☐ AV
```

```
while p ≠ 0 do

  if SIZE(p) ≥ n then [SIZE(p) □ SIZE(p) - n

  if SIZE(p) = 0 then LINK(q) □ LINK(p)

  else p □ p + SIZE(p)

  return]

  q □ p; p □ LINK (p)

end

//no block is large enough//

end FF
```

This algorithm is simple enough to understand. In case only a portion of a free block is to be allocated, the allocation is made from the bottom of the block. This avoids changing any links in the free list unless an entire block is allocated. There are, however, two major problems with *FF*. First, experiments have shown that after some processing time many small nodes are left in the available space list, these nodes being smaller than any requests that would be made. Thus, a request for 9900 words allocated from a block of size 10,000 would leave behind a block of size 100, which may be smaller than any requests that will be made to the system. Retaining these small nodes on the available space list tends to slow down the allocation process as the time needed to make an allocation is proportional to the number of nodes on the available space list. To get around this, we choose some suitable constant $\in$ such that if the allocation of a portion of a node leaves behind a node of size $< \in$, then the entire node is allocated. I.e., we allocate more storage than requested in this case. The second problem arises from the fact that the search for a large enough node always begins at the front of the list. As a result of this, all the small nodes tend to collect at the front so that it is necessary to examine several nodes before an allocation for larger blocks can be made. In order to distribute small nodes evenly along the list, one can begin searching for a new node from a different point in the list each time a request is made. To implement this. the available space list is maintained as a circular list with a head node of size zero. *AV* now points to the last node from which an allocation was made. We shall see what the new allocation algorithm looks like after we discuss what has to be done to free a block of storage.

The second operation is the freeing of blocks or returning nodes to *AV*. Not only must we return the node but we also want to recognize if its neighbors are also free so that they can be coalesced into a single block. Looking back at figure 4.15, we see that if *P*3 is the next program to terminate, then rather

than just adding this node onto the free list to get the free list of figure 4.17, it would be better to combine the adjacent free blocks corresponding to *P*2 and *P*4, obtaining the free list of figure 4.18. This combining of adjacent free blocks to get bigger free blocks is necessary. The block allocation algorithm splits big blocks while making allocations. As a result, available block sizes get smaller and smaller. Unless recombination takes place at some point, we will no longer be able to meet large requests for memory.

Figure 4.17 Available Space List When Adjacent Free Blocks Are Not Coalesced.

Figure 4.18 Available Space List When Adjacent Free Blocks Are Coalesced.

With the structure we have for the available space list, it is not easy to determine whether blocks adjacent to the block (*n, p*) (*n* = size of block and *p* = starting location) being returned are free. The only way to do this, at present, is to examine all the nodes in *AV* to determine whether:

(i) the left adjacent block is free, i.e., the block ending at *p* - 1;

(ii) the right adjacent block is free, i.e., the block beginning at *p* + *n.*

In order to determine (i) and (ii) above without searching the available space list, we adopt the node structure of figure 4.19 for allocated and free nodes:

Figure 4.19

The first and last words of each block are reserved for allocation information. The first word of each free block has four fields: LLINK, RLINK, TAG and SIZE. Only the TAG and SIZE field are important for a block in use. The last word in each free block has two fields: TAG and UPLINK. Only the TAG field is important for a block in use. Now by just examining the tag at *p* - 1 and *p* + *n* one can determine whether the adjacent blocks are free. The UPLINK field of a free block points to the start of the block. The available space list will now be a doubly linked circular list, linked through the fields LLINK and RLINK. It will have a head node with SIZE = 0. A doubly linked list is needed, as the return block algorithm will delete nodes at random from *AV*. The need for UPLINK will become clear when we study the freeing algorithm. Since the first and last nodes of each block have TAG fields, this system of allocation and freeing is called the *Boundary Tag method*. It should be noted that the TAG fields in allocated and free blocks occupy the same bit position in the first and last words respectively. This is not obvious from figure 4.19 where the LLINK field precedes the TAG field in a free node. The labeling of

fields in this figure has been done so as to obtain clean diagrams for the available space list. The algorithms we shall obtain for the boundary tag method will assume that memory is numbered 1 to m and that $TAG(0) = TAG(m + 1) = 1$. This last requirement will enable us to free the block beginning at 1 and the one ending at m without having to test for these blocks as special cases. Such a test would otherwise have been necessary as the first of these blocks has no left adjacent block while the second has no right adjacent block. While the TAG information is all that is needed in an allocated block, it is customary to also retain the size in the block. Hence, figure 4.19 also includes a SIZE field in an allocated block.

Before presenting the allocate and free algorithms let us study the initial condition of the system when all of memory is free. Assuming memory begins at location 1 and ends at m, the *AV* list initially looks like:

While these algorithms may appear complex, they are a direct consequence of the doubly linked list structure of the available space list and also of the node structure in use. Notice that the use of a head node eliminates the test for an empty list in both algorithms and hence simplifies them. The use of circular linking makes it easy to start the search for a large enough node at any point in the available space list. The UPLINK field in a free block is needed only when returning a block whose left adjacent block is free (see lines 18 and 24 of algorithm FREE). The readability of algorithm FREE has been greatly enhanced by the use of the **case** statement. In lines 20 and 27 *AV* is changed so that it always points to the start of a free block rather than into

procedure *ALLOCATE* (*n,p*)

//Use next fit to allocate a block of memory of size at least

n, n > 0. The available space list is maintained as described

above and it is assumed that no blocks of size < ∈ are to

be retained. *p* is set to be the address of the first word in

the block allocated. *AV* points to a node on the available list.//

1   *p* ☐ *RLINK(AV)*   //begin search at *p*//

2   **repeat**

3   **if** *SIZE* (*p*) ≥ *n* **then**   //block is big enough//

4   [*diff* □ *SlZE*(*p*) - *n*

5   **if** *diff* < ∈ **then**   //allocate whole block//

6   [*RLINK* (*LLINK*(*p*)) □ *RLINK*(*p*)   //delete node

from *AV*//

7   *LLINK*(*RLlNK*(*p*)) □ *LLINK*(*p*)

8   *TAG*(*p*) □ *TAG*(*p* + *SlZE*(*p*) - 1) □ 1 //set

tags//

9   *AV* □ *LLINK*(*p*)   //set starting point of next

search//

10   **return**]

11   **else**   //allocate lower *n* words//

12   [*SIZE*(*p*) □ *diff*

13   *UPLINK* (*p* + *diff* - 1) □ *p*

14   *TAG*(*p* + *diff* - 1) □ 0   //set upper portion as

unused//

15   *AV* □ *p*   //position for next search//

16   *p* □ *p* + *diff*   //set *p* to point to start of allocated

block//

17   *SlZE*(*p*) □ *n*

```
18              TAG(p) ← TAG(p + n - 1) ← 1

//set tags for allocated block//

19          return]]

20      p ← RLlNK(p)        //examine next node on list//

21    until p = RLINK(AV)

22    //no block large enough//

23    p ← 0;

24 end ALLOCATE

procedure FREE(p)

//return a block beginning at p and of size SlZE(p)//

1     n ← SlZE(p)

2    case

3       :TAG(p - 1) = 1 and TAG(p + n) = 1:

//both adjacent blocks in use//

4         TAG(p) ← TAG(p + n - 1) ← 0     //set up a free

block//

5         UPLINK (p + n - 1) ← p

6         LLINK(p) ← AV; RLINK (p) ← RLINK (AV)

//insert at right of AV//

7         LLINK (RLlNK(p)) ← p; RLINK(AV) ← p
```

```
8        :TAG(p + n) = 1 and TAG(p - 1) = 0:    //only left block

free//

9        q ← UPLINK (p - 1 )   //start of left block//

10       SIZE(q) ← SlZE(q) + n

11       UPLINK(p + n - 1) ← q; TAG(p + n - 1) ← 0

12       :TAG(p + n) = 0 and TAG(p - 1) = 1:

//only right adjacent block free//

13       RLINK(LLINK(p + n)) ← p          //replace block

beginning//

14       LLINK(RLINK(p + n)) ← p          //at p + n by one//

15       LLINK(p) ← LLINK(p + n)          //beginning at p//

16       RLINK(p) ← RLINK(p + n)

17       SlZE(p) ← n + SlZE(p + n)

18       UPLINK(p + SIZE(p) - 1) ← p

19       TAG(p) ← 0

20       AV ← p

21     :else: //both adjacent blocks free//

22             //delete right free block from AV list//

  RLINK (LLINK(p + n)) ← RLINK (p + n)
```

```
23              LLINK(RLINK(p + n))LLINK(p + n)

24              q  UPLINK(p - 1)    //start of left free

block//

25              SIZE(q)  SIZE(q) + n + SIZE(p + n)

26              UPLINK(q + SIZE(q) - 1)  q

27              AV  LLINK(p + n)

28      end

29  end FREE
```

the middle of a free block. One may readily verify that the algorithms work for special cases such as when the available space list contains only the head node.

The best way to understand the algorithms is to simulate an example. Let us start with a memory of size 5000 from which the following allocations are made: $r_1 = 300$, $r_2 = 600$, $r_3 = 900$, $r_4 = 700$, $r_5 = 1500$ and $r_6 = 1000$. At this point the memory configuration is as in figure 4.20. This figure also depicts the different blocks of storage and the available space list. Note that when a portion of a free block is allocated, the allocation is made from the bottom of the block so as to avoid unnecessary link changes in the *AV* list. First block r_1 is freed. Since TAG(5001) = TAG(4700) = 1, no coalescing takes place and the block is inserted into the *AV* list (figure 4.21a). Next, block r_4 is returned. Since both its left adjacent block (r_5) and its right adjacent block (r_3) are in use at this time (TAG(2500) = TAG(3201) =1), this block is just inserted into the free list to get the configuration of figure 4.21b. Block r_3 is next returned. Its left adjacent block is free, TAG(3200) = 0; but its right adjacent block is not, TAG(4101) = 1. So, this block is just attached to the end of its adjacent free block without changing any link fields (figure 4.21c). Block r_5 next becomes free. TAG(1000) = 1 and TAG(2501) = 0 and so this block is coalesced with its right adjacent block which is free and inserted into the spot previously occupied by this adjacent free block (figure 4.21(d)). r_2 is freed next. Both its upper and lower adjacent blocks are free. The upper block is deleted from the free space list and combined with r_2. This bigger block is now just appended to the end of the free block made up of r_3, r_4 and r_5 (figure 4.21(e)).

Figure 4.20

Figure 4.21 Freeing of Blocks in Boundary Tag System

Figure 4.21 Freeing of Blocks in Boundary Tag System (contd.)

Figure 4.21 Freeing of Blocks in Boundary Tag System (contd.)

Figure 4.21 Freeing of Blocks in Boundary Tag System (contd.)

As for the computational complexity of the two algorithms, one may readily verify that the time required to free a block of storage is independent of the number of free blocks in *AV*. Freeing a block takes a constant amount of time. In order to accomplish this we had to pay a price in terms of storage. The first and last words of each block in use are reserved for TAG information. Though additional space is needed to maintain *AV* as a doubly linked list, this is of no consequence as all the storage in *AV* is free in any case. The allocation of a block of storage still requires a search of the *AV* list. In the worst case all free blocks may be examined.

An alternative scheme for storage allocation, the Buddy System, is investigated in the exercises.

4.9 GENERALIZED LISTS

In Chapter 3, a linear list was defined to be a finite sequence of $n \geq 0$ elements, $\square_1,...,\square_n$, which we write as $A = (\square_1, ...,\square_n)$. The elements of a linear list are restricted to be atoms and thus the only structural property a linear list has is the one of position, i.e. $\square_i$ precedes $\square_{i+1}$, $1 \leq i < n$. It is sometimes useful to relax this restriction on the elements of a list, permitting them to have a structure of their own. This leads to the notion of a generalized list in which the elements $\square_i$, $1 \leq i \leq n$ may be either atoms or lists.

Definition. A *generalized list*, A, is a finite sequence of $n > 0$ elements, $\square_1$, ..., $\square_n$ where the $\square_i$ are either atoms or lists. The elements $\square_i$, $1 \le i \le n$ which are not atoms are said to be the *sublists* of A.

The list A itself is written as $A = (\square_1, ..., \square_n)$. A is the *name* of the list $(\square_1, ..., \square_n)$ and n its *length*. By convention, all list names will be represented by capital letters. Lower case letters will be used to represent atoms. If $n \ge 1$, then $\square_1$ is the *head* of A while $(\square_2, ..., \square_2)$ is the *tail* of A.

The above definition is our first example of a recursive definition so one should study it carefully. The definition is recursive because within our description of what a list is, we use the notion of a list. This may appear to be circular, but it is not. It is a compact way of describing a potentially large and varied structure. We will see more such definitions later on. Some examples of generalized lists are:

```
(i)    D = ( )            the null or empty list, its length is zero.

(ii)   A = (a, (b,c))   a list of length two; its first element is

                          the atom 'a' and its second element is the

                          linear list (b,c).

(iii) B = (A,A,( ))     a list of length three whose first two

                          elements are the lists A, the third element

                          the null list.

(iv)   C = (a, C)       a recursive list of length two. C corresponds

                          to the infinite list C = (a,(a,(a, ...).
```

Example one is the empty list and is easily seen to agree with the definition. For list A, we have

head $(A) = $ 'a', tail $(A) = ((b,c))$.

The tail (A) also has a head and tail which are (b,c) and ( ) respectively. Looking at list B we see that

head $(B)) = A$, tail $(B)) = (A, ( ))$

Continuing we have

head $(\text{tail}(B)) = A$, tail $(\text{tail}(B)) = ((\ ))$

both of which are lists.

Two important consequences of our definition for a list are: (i) lists may be shared by other lists as in example iii, where list A makes up two of the sublists of list B; and (ii) lists may be recursive as in example iv. The implications of these two consequences for the data structures needed to represent lists will become evident as we go along.

First, let us restrict ourselves to the situation where the lists being represented are neither shared nor recursive. To see where this notion of a list may be useful, consider how to represent polynomials in several variables. Suppose we need to devise a data representation for them and consider one typical example, the polynomial $P(x,y,z) =$

$$x^{10} y^3 z^2 + 2x^8 y^3 z^2 + 3x^8 y^2 z^2 + x^4 y^4 z + 6x^3 y^4 z + 2yz$$

One can easily think of a sequential representation for P, say using

Figure 4.22 Representation of P(x,y,z) using three fields per node

nodes with four fields: COEF, EXPX, EXPY, and EXPZ. But this would mean that polynomials in a different number of variables would need a different number of fields, adding another conceptual inelegance to other difficulties we have already seen with sequential representation of polynomials. If we used linear lists, we might conceive of a node of the form

These nodes would have to vary in size depending on the number of variables, causing difficulties in storage management. The idea of using a general list structure with fixed size nodes arises naturally if we consider re-writing $P(x,y,z)$ as

$$((x^{10} + 2x^8)y^3 + 3x^8y^2)z^2 + ((x^4 + 6x^3)y^4 + 2y)z$$

Every polynomial can be written in this fashion, factoring out a main variable z, followed by a second variable y, etc. Looking carefully now at $P(x,y,z)$ we see that there are two terms in the variable z, $Cz^2 + Dz$, where C and D are polynomials themselves but in the variables x and y. Looking closer at $C(x,y)$, we see that it is of the form $Ey^3 + Fy^2$, where E and F are polynomials in x. Continuing in this way we see that every polynomial consists of a variable plus coefficient exponent pairs. Each coefficient is itself a polynomial (in one less variable) if we regard a single numerical coefficient as a polynomial in zero

variables.

We can represent $P(x,y,z)$ as a list in the following way using nodes with three fields each, COEF, EXP, LINK, as in section 4.4. Note that each level has a head node indicating the variable that has been factored out. This variable is stored in the COEF field.

The list structure of figure 4.22 uses only fixed size nodes. There is, however, one difficulty which needs to be resolved: how do we distinguish between a coefficient and a pointer to another list? Ultimately, both of these values will be represented as numbers so we cannot readily distinguish between them. The solution is to add another field to each node. This field called TAG, will be zero if the COEF field truly contains a numerical coefficient and a one otherwise. Using the structure of figure 4.22 the polynomial $P = 3x^2y$ would be represented as

Notice that the TAG field is set to one in the nodes containing x and y. This is because the character codes for variable names would not really be stored in the COEF field. The names might be too large. Instead a pointer to another list is kept which represents the name in a symbol table. This setting of TAG = 1 is also convenient because it allows us to distinguish between a variable name and a constant.

It is a little surprising that every generalized list can be represented using the node structure:

```
 _____
|            |       |        |
|            |       |        |
|TAG = 0/1   | DATA  |  LINK  |
|            |       |        |
|_____|_____|_____|
```
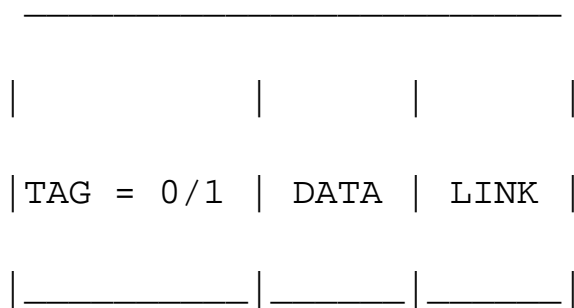
The reader should convince himself that this node structure is adequate for the representation of any list A. The LINK field may be used as a pointer to the tail of the list while the DATA field can hold an atom in case head (A) is an atom or be a pointer to the list representation of head (A) in case it is a list. Using this node structure, the example lists i-iv have the representation shown in figure 4.23.

Figure 4.23 Representation of Lists i-iv

Recursive Algorithms for Lists

Now that we have seen a particular example where generalized lists are useful, let us return to their definition again. Whenever a data object is defined recursively, it is often easy to describe algorithms which work on these objects recursively. If our programming language does not allow recursion, that should not matter because we can always translate a recursive program into a nonrecursive version. To see how recursion is useful, let us write an algorithm which produces an exact copy of a nonrecursive list *L* in which no sublists are shared. We will assume that each node has three fields, TAG, DATA and LINK, and also that there exists the procedure GETNODE(*X*) which assigns to *X* the address of a new node.

procedure *COPY (L)*

//L points to a nonrecursive list with no common sublists. COPY

returns a pointer to a new list which is a duplicate of L//

ptr □ 0

if *L* ≠ 0 *then*

[**if** *TAG(L)* = 0 **then** *q* □ *DATA(L)*　　//save an atom//

else *q* □ *COPY(DATA(L))*　　//recursion//

r □ *COPY(LINK(L))*　　//copy tail//

call *GETNODE(ptr)*　　//get a node//

DATA(ptr) □ *q; LINK(ptr)* □ *r*　　//combine head and tail//

TAG(ptr) □ *TAG(L)*]

return *(ptr)*

end *COPY*

The above procedure reflects exactly the definition of a list. We immediately see that COPY works correctly for an empty list. A simple proof using induction will verify the correctness of the entire procedure. Once we have established that the program is correct we may wish to remove the recursion for efficiency. This can be done using some straightforward rules.

(i) At the beginning of the procedure, code is inserted which declares a stack and initializes it to be empty. In the most general case, the stack will be used to hold the values of parameters, local variables, function value, and return address for each recursive call.

(ii) The label L1 is attached to the first executable statement.

Now, each recursive call is replaced by a set of instructions which do the following:

(iii) Store the values of all parameters and local variables in the stack. The pointer to the top of the stack can be treated as global.

(iv) Create the ith new label, Li, and store i in the stack. The value i of this label will be used to compute the return address. This label is placed in the program as described in rule (vii).

(v) Evaluate the arguments of this call (they may be expressions) and assign these values to the appropriate formal parameters.

(vi) Insert an unconditional branch to the beginning of the procedure.

(vii) Attach the label created in (iv) to the statement immediately following the unconditional branch. If this procedure is a function, attach the label to a statement which retrieves the function value from the top of the stack. Then make use of this value in whatever way the recursive program describes.

These steps are sufficient to remove all recursive calls in a procedure. We must now alter any **return** statements in the following way. In place of each **return** do:

(viii) If the stack is empty then execute a normal return.

(ix) therwise take the current values of all output parameters (explicitly or implicitly understood to be of type **out** or **inout**) and assign these values to the corresponding variables which are in the top of the stack.

(x) Now insert code which removes the index of the return address from the stack if one has been placed there. Assign this address to some unused variable.

(xi) Remove from the stack the values of all local variables and parameters and assign them to their corresponding variables.

(xii) If this is a function, insert instructions to evaluate the expression immediately following **return** and store the result in the top of the stack.

(xiii) Use the index of the label of the return address to execute a branch to that label.

By following these rules carefully one can take any recursive program and produce a program which works in exactly the same way, yet which uses only iteration to control the flow of the program. On many compilers this resultant program will be much more efficient than its recursive version. On other compilers the times may be fairly close. Once the transformation to iterative form has been accomplished, one can often simplify the program even further thereby producing even more gains in efficiency. These rules have been used to produce the iterative version of COPY which appears on the next page.

It is hard to believe that the nonrecursive version is any more intelligible than the recursive one. But it does show explicitly how to implement such an algorithm in, say, FORTRAN. The non-recursive version does have some virtues, namely, it is more efficient. The overhead of parameter passing on most compilers is heavy. Moreover, there are optimizations that can be made on the latter version, but not on the former. Thus,

```
procedure COPY(L)

//nonrecursive version//

i ← 0  //initialize stack index//

1: ptr ← 0

if L ≠ 0 then

[if TAG(L) = 0

then q ← DATA(L)

else [STACK(i + 1) ← q     //stack local variables//

STACK(i + 2) ← r

STACK(i + 3) ← ptr

STACK(i + 4) ← L     //stack parameter//

STACK(i + 5) ← 2     //stack return address//
```

```
i ← i + 5;

L ← DATA(L); go to 1        //set parameter and

begin//

2:                  q ← STACK(i)      //remove function value//

i ← i - 1]

STACK(i + 1) ← q; STACK(i + 2) ← r  //stack

variables and//

STACK(i + 3) ← ptr; STACK(i + 4) ← L

//parameter for second//

STACK(i + 5) ← 3; i ← i +5      //recursive call//

L ← LINK(L); go to 1

3:          r ← STACK(i); i ← i ← 1   //remove function value//

call GETNODE(ptr)

DATA(ptr) ← q; LlNK(ptr) ← r

TAG(ptr) ← TAG(L)]

if i ≠ 0 then [addr ← STACK(i); L ← STACK(i - 1)

//execute a return//

t ← STACK(i - 2); r ← STACK(i - 3)

q ← STACK(i - 4);
```

```
STACK(i - 4) ▢ ptr; ptr ▢ t   //store function
value//

i ▢ i - 4; go to addr]     //branch to 2 or 3//

return (ptr)

end COPY
```

both of these forms have their place. We will often use the recursive version for descriptive purposes.

Now let us consider the computing time of this algorithm. The null

▢

Figure 4.24 Linked Representation for A

list takes a constant amount of time. For the list

A = ((a,b),((c,d),e))

which has the representation of figure 4.24 *L* takes on the values given in figure 4.25.

Levels of recursion	Value of L	Continuing Levels	L	Continuing Levels	L
1	q	2	r	3	u
2	s	3	u	4	v
3	t	4	w	5	o
4	o	5	x	4	v
3	t	6	o	3	u
2	s	5	x	2	r

1	q	4	w	3	o
				2	r
				1	q

Figure 4.25 Values of Parameter in Execution of COPY(A)

The sequence of values should be read down the columns. *q, r, s, t, u, v, w, x* are the addresses of the eight nodes of the list. From this particular example one should be able to see that nodes with TAG = 0 will be visited twice, while nodes with TAG= 1 will be visited three times. Thus, if a list has a total of *m* nodes, no more than 3*m* executions of any statement will occur. Hence, the algorithm is $O(m)$ or linear which is the best we could hope to achieve. Another factor of interest is the maximum depth of recursion or, equivalently, how many locations one will need for the stack. Again, by carefully following the algorithm on the previous example one sees that the maximum depth is a combination of the lengths and depths of all sublists. However, a simple upper bound to use is *m*, the total number of nodes. Though this bound will be extremely large in many cases, it is achievable, for instance, if

L= (((((*a*))))).

Another procedure which is often useful is one which determines whether two lists are identical. This means they must have the same structure and the same data in corresponding fields. Again, using the recursive definition of a list we can write a short recursive program which accomplishes this task.

procedure *EQUAL(S,T)*

//S and T are nonrecursive lists, each node having three fields: *TAG,*

DATA and *LINK.* The procedure returns the value **true** if the

lists are identical else **false**//

ans ⬚ **false**

case

:S = 0 **and** *T* = 0: *ans* ⬚ **true**

:S ≠ *0 and* T ≠ *0:*

```
if TAG(S) = TAG(T)

then [if TAG(S) = 0

then ans ☐ DATA (S) = DATA(T)

else ans ☐ EQUAL(DATA(S),DATA(T))

if ans then

ans ☐ EQUAL(LINK(S),LINK(T))]

end

return (ans)

end EQUAL
```

Procedure EQUAL is a function which returns either the value **true** or **false**. Its computing time is clearly no more than linear when no sublists are shared since it looks at each node of *S* and *T* no more than three times each. For unequal lists the procedure terminates as soon as it discovers that the lists are not identical.

Another handy operation on nonrecursive lists is the function which computes the depth of a list. The depth of the empty list is defined to be zero and in general

☐

Procedure DEPTH is a very close transformation of the definition which is itself recursive.

By now you have seen several programs of this type and you should be feeling more comfortable both reading and writing recursive algorithms. To convince yourself that you understand the way these work, try exercises 37, 38, and 39.

```
procedure DEPTH(S)

//S is a nonrecursive list having nodes with fields TAG, DATA an

LINK. The procedure returns the depth of the list//
```

```
max ☐ 0

if S = 0 then return (max)        //null list has zero depth//

ptr ☐ S

while ptr ≠ 0 do

if TAG(ptr) = 0 then ans ☐ 0

else ans ☐ DEPTH(DATA(ptr))      //recursion//

if ans > max     then max ☐ ans     //find a new maximum//

ptr ☐ LINK(ptr)

end

return (max + 1)

end DEPTH
```

Reference Counts, Shared and Recursive Lists

In this section we shall consider some of the problems that arise when lists are allowed to be shared by other lists and when recursive lists are permitted. Sharing of sublists can in some situations result in great savings in storage used, as identical sublists occupy the same space. In order to facilitate ease in specifying shared sublists, we extend the definition of a list to allow for naming of sublists. A sublist appearing within a list definition may be named through the use of a list name preceding it. For example, in the list $A = (a,(b,c))$, the sublist (b,c) could be assigned the name Z by writing $A = (a,Z(b,c))$. In fact, to be consistent we would then write $A(a,Z(b,c))$ which would define the list A as above.

Lists that are shared by other lists, such as list A of figure 4.23, create problems when one wishes to add or delete a node at the front. If the first node of A is deleted, it is necessary to change the pointers from the list B to point to the second node. In case a new node is added then pointers from B have to be changed to point to the new first node. However, one normally does not know all the points from which a particular list is being referenced. (Even if we did have this information, addition and deletion of nodes could require a large amount of time.) This problem is easily solved through the use of head nodes. In case one expects to perform many add/deletes from the front of lists, then the use of a head node with each list or named sublist will eliminate the need to retain a list of all pointers to any specific list. If each

list is to have a head node, then lists i-iv are represented as in figure 4.26. The TAG field of head nodes is not important and is assumed to be zero. Even in situations where one does not wish to dynamically add or delete nodes from lists. ai in the case of multivariate polynomials, head nodoes prove usefull in determining when the nodes of a particular structure may be returned to the storage pool. For example, let T and U be program variables pointing to the two polynomials $(3x^4 + 5x^3 + 7x)y^3$ and $(3x^4 + 5x^3 + 7x)y^6 + (6x)y$ of figure 4.27. If PERASE(X) is all algorithm to erase the polynomial X, then a call to PERASE(T) should not return the nodes corresponding to the coefficient $3x^4 + 5x^3 + 7x$ since this sublist is also part of U.

Thus, whenever lists are being shared by other lists, we need a mechanism to help determine whether or not the list nodes may be physically returned to the available space list. This mechanism is generally provided through the use of a reference count maintained in the head node of each list. Since the DATA field of the head nodes is free, the reference count is maintained in this field. This reference count of a list is the number of pointers (either program variables or pointers from other lists) to that list. If the lists i-iv of figure 4.26 are accessible via the program variables X, Y, Z and W, then the reference counts for the lists are:

(i) REF$(X) = 1$ accessible only via X

(ii) REF$(Y) = 3$ pointed to by Y and two points from Z

(iii) REF$(Z) = 1$ accessible only via Z

(iv) REF$(W) = 2$ two pointers to list C

Now a call to LERASE(T) (list erase) should result only in a

Figure 4.26 Structure with Head Nodes for Lists i-iv

Figure 4.27 T = $(3x^4 + 5x^3 + 7x)y^3$ U = $(3x^4 + 5x^3 + 7x)y^6 + 6xy$

decrementing by 1 of the reference counter of T. Only if the reference count becomes zero are the nodes of T to be physically returned to the available space list. The same is to be done to the sublists of T.

Assuming the structure TAG, REF, DATA, LINK, an algorithm to erase a list X could proceed by examining the top level nodes of a list whose reference count has become zero. Any sublists encountered are erased and finally, the top level nodes are linked into the available space list. Since the DATA field

of head nodes will usually be unused, the REF and DATA fields would be one and the same.

procedure *LERASE(X)*

//recursive algorithm to erase a list assuming a *REF* field in each

head node which has the number of pointers to this list and a

TAG field such that *TAG(X)* = 0 if *DATA(X)* is actually an atom

and *TAG(X)* = 1 if *DATA(X)* is a link to a sublist. The storage

pool is assumed linked through the field *LINK* with *AV* pointing

to the first node in the pool//

REF(X) ☐ *REF(X)* – 1   //decrement reference count//

if *REF(X)* ≠ 0 **then return**

Y ☐ *X* // *Y* will traverse top level of *X*//

while *LINK(Y)* ≠ 0 **do**

Y ☐ LINK(Y)

if TAG(Y) = 1 **then call** LERASE (DATA (Y))// recursion//

end

LINK(Y) ☐ *AV*   //attach top level nodes to avail list//

AV ☐ *X*

end *LERASE*

A call to LERASE(*Y*) will now only have the effect of decreasing the reference count of *Y* to 2. Such a call followed by a call to LERASE(*Z*) will result in:

(i) reference count of Z becomes zero;

(ii) next node is processed and REF(Y) reduces to 1;

(iii) REF(Y) becomes zero and the five nodes of list A(a,(b,c)) are returned to the available space list;

(iv) the top level nodes of Z are linked into the available space list

The use of head nodes with reference counts solves the problem of determining when nodes are to be physically freed in the case of shared sublists. However, for recursive lists, the reference count never becomes zero. LERASE(*W*) just results in REF(*W*) becoming one. The reference count does not become zero even though this list is no longer accessible either through program variables or through other structures. The same is true in the case of indirect recursion (figure 4.28). After calls to LERASE(*R*) and LERASE(*S*), REF(*R*) = 1 and REF(*S*)= 2 but the structure consisting of *R* and *S* is no longer being used and so it should have been returned to available space.

Figure 4.28 Indirect Recursion of Lists A and B Pointed to by Program Variables R and S.

Unfortunately, there is no simple way to supplement the list structure of figure 4.28 so as to be able to determine when recursive lists may be physically erased. It is no longer possible to return all free nodes to the available space list when they become free. So when recursive lists are being used, it is possible to run out of available space even though not all nodes are in use. When this happens, it is possible to collect unused nodes (i.e., garbage nodes) through a process known as garbage collection. This will be described in the next section.

4.10 GARBAGE COLLECTION AND COMPACTION

As remarked at the close of the last section, garbage collection is the process of collecting all unused nodes and returning them to available space. This process is carried out in essentially two phases. In the first phase, known as the marking phase, all nodes in use are marked. In the second phase all unmarked nodes are returned to the available space list. This second phase is trivial when all nodes are of a fixed size. In this case, the second phase requires only the examination of each node to see whether or not it has been marked. If there are a total of n nodes, then the second phase of garbage collection can be carried out in $O(n)$ steps. In this situation it is only the first or marking phase that is of any interest in designing an algorithm. When variable size nodes are in use, it is desirable to compact memory so that all free nodes form a contiguous block of memory. In this case the second phase is referred to as memory compaction. Compaction of disk space to reduce average retrieval time is desirable even for fixed size nodes. In this section we shall study two marking algorithms and one compaction algorithm.

Marking

In order to be able to carry out the marking, we need a mark bit in each node. It will be assumed that this mark bit can be changed at any time by the marking algorithm. Marking algorithms mark all directly accessible nodes (i.e., nodes accessible through program variables referred to as pointer variables) and also all indirectly accessible nodes (i.e., nodes accessible through link fields of nodes in accessible lists). It is assumed that a certain set of variables has been specified as pointer variables and that these variables at all times are either zero (i.e., point to nothing) or are valid pointers to lists. It is also assumed that the link fields of nodes always contain valid link information.

Knowing which variables are pointer variables, it is easy to mark all directly accessible nodes. The indirectly accessible nodes are marked by systematically examining all nodes reachable from these directly accessible nodes. Before examining the marking algorithms let us review the node structure in use. Each node regardless of its usage will have a one bit mark field, MARK, as well as a one bit tag field, TAG. The tag bit of a node will be zero if it contains atomic information. The tag bit is one otherwise. A node with a tag of one has two link fields DLINK and RLINK. Atomic information can be stored only in a node with tag 0. Such nodes are called atomic nodes. All other nodes are list nodes. This node structure is slightly different from the one used in the previous section where a node with tag 0 contained atomic information as well as a RLINK. It is usually the case that the DLINK field is too small for the atomic information and an entire node is required. With this new node structure, the list (a, (b)) is represented as:

Both of the marking algorithms we shall discuss will require that all nodes be initially unmarked (i.e. MARK(i) = 0 for all nodes i) . In addition they will require MARK(0) = 1 and TAG(0) = 0. This will enable us to handle end conditions (such as end of list or empty list) easily. Instead of writing the code for this in both algorithms we shall instead write a driver algorithm to do this. Having initialized all the mark bits as well as TAG(0), this driver will then repeatedly call a marking algorithm to mark all nodes accessible from each of the pointer variables being used. The driver algorithm is fairly simple and we shall just state it without further explanation. In line 7 the algorithm invokes MARK1. In case the second marking algorithm is to be used this can be changed to call MARK2. Both marking algorithms are written so as to work on collections of lists (not necessarily generalized lists as defined here).

procedure *DRIVER*

```
//driver for marking algorithm. n is the number of nodes in the

system//

1     for i ⬜ 1 to n do  //unmark all nodes//
```

```
2        MARK (i) ☐ 0

3    end

4    MARK (0) ☐ 1; TAG (0) ☐ 0          //boundary conditions//

5    for each pointer variable X with MARK (X) = 0 do

6        MARK (X) ☐ 1

7        if TAG(X) = 1 then call MARK1(X)          //X is a list node//

8    end

9   end DRIVER
```

The first marking algorithm MARK1(*X*) will start from the list node *X* and mark all nodes that can be reached from *X* via a sequence of RLINK's and DLINK's; examining all such paths will result in the examination of all reachable nodes. While examining any node of type list we will have a choice as to whether to move to the DLINK or to the RLINK. MARK1 will move to the DLINK but will at the same time place the RLINK on a stack in case the RLINK is a list node not yet marked. The use of this stack will enable us to return at a later point to the RLINK and examine all paths from there. This strategy is similar to the one used in the previous section for LERASE.

Analysis of MARK1

In line 5 of MARK1 we check to see if *Q*= RLINK(*P*) can lead to other unmarked accessible nodes. If so, *Q* is stacked. The examination

```
procedure MARK1 (X)

//X is a list node. Mark all nodes accessible from X. It is assumed

that MARK(0) = 1 and TAG(0) = 0. ADD and DELETE

perform the standard stack operations//

1    P ☐ X; initialize stack
```

```
2    loop           //follow all paths from P, P is a marked list node//

3      loop         //move downwards stacking RLINK's if needed//

4        Q ☐ RLINK (P)

5        if TAG(Q) = 1 and MARK(Q) = 0 then call ADD(Q)

//stack Q//

6        MARK(Q) ☐ 1      //Q may be atomic//

7        P ☐ DLINK (P)

//any unmarked nodes accessible from P?//

8        if MARK(P) = 1 or TAG(P) = 0 then exit

9        MARK(P) ☐ 1

10     forever

11     MARK(P) ☐ 1              //P may be an unmarked atomic node//

12     if stack empty then return      //all accessible nodes
marked//

13     call DELETE(P)              //unstack P//

14   forever

15 end MARK1
```

of nodes continues with the node at DLINK(*P*). When we have moved downwards as far as is possible, line 8, we exit from the loop of lines 3-10. At this point we try out one of the alternative moves from the stack, line 13. One may readily verify that MARK1 does indeed mark all previously unmarked nodes which are accessible from *X*.

In analyzing the computing time of this algorithm we observe that on each iteration (except for the last) of the loop of lines 3-10, at least one previously unmarked node gets marked (line 9). Thus, if the outer

loop, lines 2-14, is iterated *r* times and the total number of iterations of the inner loop, lines 3-10, is *p* then at least $q = p - r$ previously unmarked nodes get marked by the algorithm. Let *m* be the number of new nodes marked. Then $m \geq q = p - r$. Also, the number of iterations of the loop of lines 2-14 is one plus the number of nodes that get stacked. The only nodes that can be stacked are those previously unmarked (line 5). Once a node is stacked it gets marked (line 6). Hence $r \leq m + 1$. *From this and the knowledge that* $m \geq p - r$, *we conclude that* $p \leq 2m + 1$. *The computing time of the algorithm is* $O(p + r)$. *Substituting for* p *and* r *we obtain* $O(m)$ *as the computing time. The time is linear in the number of new nodes marked! Since any algorithm to mark nodes must spend at least one unit of time on each new node marked, it follows that there is no algorithm with a time less than* $O(m)$. *Hence MARK1 is optimal to within a constant factor (recall that* $2m = O(m)$ *and* $10m = O(m)$*).*

Having observed that MARK1 is optimal to within a constant factor you may be tempted to sit back in your arm chair and relish a moment of smugness. There is, unfortunately, a serious flaw with MARK1. This flaw is sufficiently serious as to make the algorithm of little use in many garbage collection applications. Garbage collectors are invoked only when we have run out of space. This means that at the time MARK1 is to operate, we do not have an unlimited amount of space available in which to maintain the stack. In some applications each node might have a free field which can be used to maintain a linked stack. In fact, if variable size nodes are in use and storage compaction is to be carried out then such a field will be available (see the compaction algorithm COMPACT). When fixed size nodes are in use, compaction can be efficiently carried out without this additional field and so we will not be able to maintain a linked stack (see exercises for another special case permitting the growth of a linked stack). Realizing this deficiency in MARK1, let us proceed to another marking algorithm MARK2. MARK2 will not require any additional space in which to maintain a stack. Its computing is also $O(m)$ but the constant factor here is larger than that for MARK1.

Unlike MARK1(*X*) which does not alter any of the links in the list *X*, the algorithm MARK2(*X*) will modify some of these links. However, by the time it finishes its task the list structure will be restored to its original form. Starting from a list node *X*, MARK2 traces all possible paths made up of DLINK's and RLINK's. Whenever a choice is to be made the DLINK direction is explored first. Instead of maintaining a stack of alternative choices (as was done by MARK1) we now maintain the path taken from *X* to the node *P* that is currently being examined. This path is maintained by changing some of the links along the path from *X* to *P*.

Consider the example list of figure 4.29(a). Initially, all nodes except node *A* are unmarked and only node *E* is atomic. From node *A* we can either move down to node *B* or right to node *I*. MARK2 will always move down when faced with such an alternative. We shall use *P* to point to the node currently being examined and *T* to point to the node preceding *P* in the path from *X* to *P*. The path *T* to *X* will be maintained as a chain comprised of the nodes on this *T*- *X* path. If we advance from node *P* to node *Q* then either $Q = RLINK(P)$ *or* $Q = DLINK(P)$ and *Q* will become the node currently being examined . The node preceding *Q* on the *X* - *Q* path is *P* and so the path list must be updated to represent the path from *P* to *X*. This is simply done by adding node *P* to the *T* - *X* path already constructed. Nodes will be linked onto this path either through their DLINK or RLINK field. Only list nodes will be placed onto this path chain. When node *P* is being added to the path chain, *P* is linked to *T* via its DLINK field if *Q*

= DLINK(P). When Q = RLINK(P), P is linked to T via its RLINK field. In order to be able to determine whether a node on the T-X path list is linked through its DLINK or RLINK field we make use of the tag field. Notice that since the T-X path list will contain only list nodes, the tag on all these nodes will be one. When the DLINK field is used for linking, this tag will be changed to zero. Thus, for nodes on the T-X path we have:

☐

☐

Figure 4.29 Example List for MARK2

☐

☐

A   B   C   D   F   G   F   D   C   B   H   B   A   I   J   I   A

(e) Path taken by P

Figure 4.29 Example List for MARK2 (contd.)

☐

The tag will be reset to 1 when the node gets off the T-X path list.

Figure 4.29(b) shows the T-X path list when node P is being examined. Nodes A, B and C have a tag of zero indicating that linking on these nodes is via the DLINK field. This also implies that in the original list structure, B = DLINK(A), C = DLINK(B) and D = P = DLINK(C). Thus, the link information destroyed while creating the T-X path list is present in the path list. Nodes B, C, and D have already been marked by the algorithm. In exploring P we first attempt to move down to Q = DLINK(P) = E. E is an atomic node so it gets marked and we then attempt to move right from P. Now, Q = RLINK(P) = F. This is an unmarked list node. So, we add P to the path list and proceed to explore Q. Since P is linked to Q by its RLINK field, the linking of P onto the T-X path is made throught its RLINK field. Figure 4.29(c) shows the list structure at the time node G is being examined. Node G is a deadend. We cannot move further either down or right. At this time we move backwards on the X-T path resetting links and tags until we reach a node whose RLINK has not yet been examined. The marking continues from this node. Because nodes are removed from the T-X path list in the reverse order in which they were added to it, this list behaves as a stack. The remaining details of MARK2 are spelled out in the formal SPARKS algorithm. The same driver as for MARK1 is assumed.

```
procedure MARK 2(X)
```

```
//same function as MARK1//

//it is assumed that MARK(0) = 1 and TAG(0) = 0//

1    P ☐ X; T ☐ 0      //initialize T - X path list//

2    repeat

3      Q ☐ DLINK(P)         //go down list//

4      case

5        :MARK(Q) = 0 and TAG(Q) = 1:       //Q is an unexamined

list node//

6          MARK(Q) ☐ 1; TAG(P) ☐ 0

7          DLINK(P) ☐ T; T ☐ P       //add Pto T - X path list//

8          P ☐ Q       //move down to explore Q//

9      :else:      //Q is an atom or already examined//

10         MARK(Q) ☐ 1

11    L1:  Q ☐ RLINK(P)     //move right of P//

12            case

13        :MARK(Q) = 0 and TAG(Q) = 1:      //explore Q

further//

14            MARK(Q) ☐ 1

15            RLINK (P) ☐ T; T ☐ P
```

```
16              P □ Q

17        :else: MARK(Q) □ 1      //Q is not to be explored;

back up on path list//

18        while T ≠ 0 do     //while path list not empty//

19            Q □ T

20            if TAG(Q) = 0     //link to P through DLINK//

21            then [T < DLINK(Q); DLINK (Q) □ P

22                  TAG(Q) □ 1; P □ Q; go to L1]

//P is node to right of Q//

23                  T □ RLINK(Q); RLINK(Q) □ P

24                  P □ Q

25            end

26        end

27     end

28     until T= 0

29 end MARK 2
```

Analysis of MARK2

The correctness of MARK2 may be proved by first showing that the T - X path list is always maintained correctly and that except when $T = 0$ it is the case that the node T was previously linked to P. Once this has been shown, it is a simple matter to show that the backup procedure of lines 17-25 restructures the list X correctly. That all paths from X are examined follows from the observation that at every list node when a DLINK path is used, that node gets added to the path list (lines 6-8) and that backing up along

the *T - X* path stops when a node whose RLINK hasn't yet been examined is encountered (lines 20-22). The details of the proof are left as an exercise. One can also show that at the time the algorithm terminates *P = X* and so we need not use two distinct variables *P* and *X*.

Figure 4.29(e) shows the path taken by *P* on the list of 4.29(a). It should be clear that a list node previously unmarked gets visited at most three times. Except for node *X*, each time a node already marked is reached at least one previously unmarked node is also examined (i.e. the one that led to this marked node). Hence the computing time of MARK2 is $O(m)$ where m is the number of newly marked nodes. The constant factor associated with *m* is, however, larger than that for MARK1 but MARK2 does not require the stack space needed by MARK1. A faster marking algorithm can be obtained by judiciously combining the strategies of MARK1 and MARK2 (see the exercises).

When the node structure of section 4.9 is in use, an additional one bit field in each node is needed to implement the strategy of MARK2. This field is used to distinguish between the case when a DLINK is used to link into the path list and when a RLINK is used. The already existing tag field cannot be used as some of the nodes on the *T- X* path list will originally have a tag of 0 while others will have a tag of 1 and so it will not be possible to correctly reset tag values when nodes are removed from the *T - X* list.

Storage Compaction

When all requests for storage are of a fixed size, it is enough to just link all unmarked (i.e., free) nodes together into an available space list. However, when storage requests may be for blocks of varying sizes, it is desirable to compact storage so that all the free space forms one contiguous block. Consider the memory configuration of figure 4.30. Nodes in use have a MARK bit = 1 while free nodes have their MARK bit = 0. The nodes are labeled 1 through 8, with n_i, $1 \le i \le 8$ being the size of the ith node.

Figure 4.30 Memory Configuration After Marking. Free Nodes Have MARK bit = O

The free nodes could be linked together to obtain the available space list of figure 4.31. While the total amount of memory available is $n_1 + n_3 + n_5 + n_8$, a request for this much memory cannot be met since the memory is fragmented into 4 nonadjacent nodes. Further, with more and more use of these nodes, the size of free nodes will get smaller and smaller.

Figure 4.31 Available Space List Corresponding to Figure 4.30

Ultimately, it will be impossible to meet requests for all but the smallest of nodes. In order to overcome this, it is necessary to reallocate the storage of the nodes in use so that the used part of memory (and

hence also the free portion) forms a contiguous block at one end as in figure 4.32. This reallocation of storage resulting in a partitioning of memory into two contiguous blocks (one used, the other free) is referred to as storage compaction. Since there will, in general, be links from one node to another, storage compaction must update these links to point to the relocated address of the respective node. If node n_i starts at location l_i before compaction and at l'_i after compaction, then all link references to l_i must also be changed to l'_i in order not to disrupt the linked list structures existing in the system. Figure 4.33(a) shows a possible link configuration at the time the garbage collection process is invoked. Links are shown only for those nodes that were marked during the marking phase. It is assumed that there are only two links per node. Figure 4.33(b) shows the configuration following compaction. Note that the list structure is unchanged even though the actual addresses represented by the links have been changed. With storage compaction we may identify three tasks: (i) determine new addresses for nodes in use; (ii) update all links in nodes in use; and (iii) relocate nodes to new addresses. Our storage compaction algorithm, COMPACT, is fairly straightforward, implementing each of these three tasks in a separate scan of memory. The algorithm assumes that each node, free or in use, has a SIZE field giving the length of the node and an additional field, NEW_ADDR, which may be used to store the relocated address of the node. Further, it is assumed that each node has two link fields LINK1 and LINK2. The extension of the algorithm to the most general situation in which nodes have a variable number of links is simple and requires only a modification of phase II.

Figure 4.32 Memory Configuration After Reallocating Storage to Nodes in Use

Figure 4.33

In analyzing this algorithm we see that if the number of nodes in memory is n, then phases I and II each require n iterations of their respective

```
procedure COMPACT(MEMORY, MARK,SlZE, M, NEW__ADDR)

//compact storage following the marking phase of garbage collection.

Nodes in use have MARK bit=1. SlZE(i)= number of words

in that node. NEW__ADDR is a free field in each node. Memory

is addressed 1 to M and is an array MEMORY.//
```

```
//phase I: Scan memory from left assigning new addresses to nodes

in use. AV = next available word//

AV □ 1;i □ 1          //variable i will scan all nodes left to
right//

while i ≤ M do

if MARK(i) = 1 then [//relocate node//

NEW__ADDR(i) □ AV

AV □ AV + SIZE(i)]

i □ i + SIZE(i)      //next node//

end

//phase II: update all links. Assume the existence of a fictitious

node with address zero and NEW__ADDR(0) = 0//

i □ 1

while i ≤ M do

if MARK(i) = 1 then [//update all links to reflect new

addresses//

LINK1(i) □ NEW_ADDR(LINK1(i))

LlNK2(i) □ NEW_ADDR(LINK2(i))]

i □ i + SIZE(i)

end
```

```
//phase III: relocate nodes//

i ← 1

while i ≤ M do

if MARK (i) = 1 then [//relocate to NEW_ADDR(i)//

k ← NEW_ADDR(i); l ← k

for j ← i to i + SIZE(i) - 1 do

MEMORY(k) ← MEMORY(j)

k ← k+ 1

end

i ← i + SIZE(l)]

else i ← i + SIZE(i)

end

end COMPACT
```

while loops. Since each iteration of these loops takes a fixed amount of time, the time for these two phases is $O(n)$. Phase III, however, will in general be more expensive. Though the **while** loop of this phase is also executed only n times, the time per iteration depends on the size of the node being relocated. If s is the amount of memory in use, then the time for this phase is $O(n + s)$. The overall computing time is, therefore, $O(n + s)$. The value of *AV* at the end of phase I marks the beginning of the free space. At the termination of the algorithm the space MEMORY(*AV*) to MEMORY(*M*) is free space. Finally, the physical relocation of nodes in phase III can be carried out using a long shift in case your computer has this facility.

In conclusion, we remark that both marking and storage compaction are slow processes. The time for the former is O (number of nodes) while the time for the latter is O (number of nodes + Σ size of nodes relocated)). In the case of generalized lists, garbage collection is necessitated by the absence of any other efficient means to free storage when needed. Garbage collection has found use in some programming

languages where it is desirable to free the user from the task of returning storage. In both situations, a disciplined use of pointer variables and link fields is required. Clever coding tricks involving illegal use of link fields could result in chaos during marking and compaction. While compaction has been presented here primarily for use with generalized list systems using nodes of variable size, compaction can also be used in other environments such as the dynamic storage allocation environment of section 4.8. Even though coalescing of adjacent free blocks takes place in algorithm FREE of section 4.8, it is still possible to have several small nonadjacent blocks of memory free. The total size of these blocks may be large enough to meet a request and it may then be desirable to compact storage. The compaction algorithm in this case is simpler than the one described here. Since all addresses used within a block will be relative to the starting address rather than the absolute address, no updating of links within a block is required. Phases I and III can, therefore, be combined into one phase and phase II eliminated altogether. Since compaction is very slow one would like to minimize the number of times it is carried out. With the introduction of compaction, several alternative schemes for dynamic storage management become viable. The exercises explore some of these alternatives.

4.11 STRINGS--A CASE STUDY

Suppose we have two character strings $S = 'x_1 \ldots x_m'$ and $T = 'y_1 \ldots y_n'$. The characters x_i, y_j come from a set usually referred to as the *character set* of the programming language. The value n is the length of the character string T, an integer which is greater than or equal to zero. If $n = 0$ then T is called the empty or *null string.* In this section we will discuss several alternate ways of implementing strings using the techniques of this chapter.

We begin by defining the data structure STRING using the axiomatic notation. For a set of operations we choose to model this structure after the string operations of *PL/I*. These include:

(i) NULL produces an instance of the null string;

(ii) ISNULL returns true if the string is null else false;

(iii) IN takes a string and a character and inserts it at the end of the string;

(iv) LEN returns the length of a string;

(v) CONCAT places a second string at the end of the first string;

(vi) SUBSTR returns any length of consecutive characters;

(vii) INDEX determines if one string is contained within another.

We now formally describe the string data structure.

structure *STRING*

declare NULL( ) ☐ *string;* ISNULL (*string*) ☐ *boolean*

IN (*string, char*) ☐ *string;* LEN (*string*) ☐ *integer*

CONCAT (string, string) ☐ string

SUBSTR (*string, integer, integer*) ☐ *string*

LINK_DEST 3184*string, string*) ☐ *integer;*

for all S,T ∈ *string*, i,j ∈ *integer*, c,d ∈ *char* **let**

ISNULL (NULL) :: = true; ISNULL (IN(S,c)) :: = **false**

LEN (NULL) :: = 0; LEN (IN(S,c)) :: = 1 + LEN(S)

CONCAT (S,NULL):: = S

CONCAT (S,IN(T,c)) :: = IN (CONCAT (S,T),c)

SUBSTR (NULL,i,j):: = NULL

SUBSTR (IN(S,c),i,j) :: =

if j = 0 **or** *i* + j - 1 > LEN(IN(S,c)) **then** NULL

else if *i* + j - 1 = LEN(IN(S,c))

then IN(SUBSTR (S,*i*,j - 1),c)

else SUBSTR(S,i,j)

INDEX (S,NULL) :: = LEN (S) + 1

INDEX (NULL,IN(T,d)) :: = 0

```
INDEX (IN(S,c),IN(T,d)) :: =
```

if $c = d$ **and** INDEX(S,T) = LEN(S) – LEN(T) + 1

then INDEX(S,T) **else** INDEX(S,IN(T,d))

end

end *STRING*

As an example of how these axioms work, let *S = `abcd'*. This will be represented as

IN(IN(IN(IN(NULL,*a*),*b*),*c*),*d*).

Now suppose we follow the axioms as they apply to SUBSTR(*S*,2,1). By the SUBSTR axioms we get

```
SUBSTR(S,2,1) = SUBSTR(IN(IN(IN(NULL,a),b),c),2,1)
```

```
= SUBSTR(IN(IN(NULL,a),b),2,1)
```

```
= IN(SUBSTR(IN(NULL,a),2,0),b)
```

```
= IN(NULL,b)
```

```
= 'b'
```

Suppose we try another example, SUBSTR(*S*,3,2)

```
= IN(SUBSTR(IN(IN(IN(NULL,a),b),c),3,1),d)
```

```
= IN(IN(SUBSTR(IN(IN(NULL,a),b),3,0),c),d)
```

```
= IN(IN(NULL,c),d)
```

```
= 'cd'
```

For the readers amusement try to simulate the steps taken for INDEX(*S,T*) where *T= 'bc'* = IN(IN(NULL,*b*),*c*).

4.11.1 DATA REPRESENTATIONS FOR STRINGS

In deciding on a data representation for a given data object one must take into consideration the cost of performing different operations using that representation. In addition, a hidden cost resulting from the necessary storage management operations must also be taken into account. For strings, all three types of representation: sequential, linked list with fixed size nodes, and linked list with variable size nodes, are possible candidates. Let us look at the first two of these schemes and evaluate them with respect to storage management as well as efficiency of operation. We shall assume that the available memory is an array C of size n and that each element of C is large enough to hold exactly one character.

On most computers this will mean that each element of C uses only a fraction of a word and hence several such elements will be packed into one word. On a 60 bit per word machine requiring 6 bits per character, each word will be able to hold 10 characters. Thus, $C(1)$ through $C(10)$ will occupy one word, $C(11)$ thru $C(20)$ another and so on.

Sequential. In this representation successive characters of a string will be placed in consecutive character positions in the vector C. The string $S = 'x_1, ..., x_n'$ could then be represented as in figure 4.34 with S a pointer to the first character. In order to facilitate easy length determination, the length of string S could be kept in another variable, SL. Thus, we would have $SL = n$. SUBSTRING SUBSTR $(S, j, k - j + 1)$ could be done now by copying over the characters $X_j, ..., X_k$ from locations $C(S + j - 1)$ through $C(S + k - 1)$ into a free space. The length of the string created would be $k - j + 1$ and the time required $O(k - j + 1)$ plus the time needed to locate a free space big enough to hold the string. CONCAT (S, T) could similarly be carried out; the length of the resulting string would be $SL + TL$. For storage management two possibilities exist. The boundary tag scheme of section 4.8 could be used in conjunction with the storage compaction strategies of section 4.10. The storage overhead would be enormous for small strings. Alternatively, we could use garbage collection and compaction whenever more free space was needed . This would eliminate the need to return free spaces to an available space list and hence simplify the storage allocation process (see exercises).

Figure 4.34 Sequential Representation of S = 'x$_1$ ... x$_n$'

While a sequential representation of strings might be adequate for the functions discussed above, such a representation is not adequate when insertions and deletions into and from the middle of a string are carried out. An insertion of '$y_1, ..., y_n$' after the i-th character of S will, in general require copying over the characters $x_1, ..., x_i$ followed by $y_1, ..., y_m$ and then $x_{i+1}, ..., x_n$ into a new free area (see figure 4.35). The time required for this is $O(n + m)$. Deletion of a substring may be carried out by either replacing the deleted characters by a special symbol Φ or by compacting the space originally occupied by this substring (figure 4.36). The former entails storage waste while the latter in the worst case takes time proportional to LENGTH(S). The replacement of a substring of S by another string T is efficient only if the length of the substring being replaced is equal to LENGTH(T). If this is not the case, then some form of string movement will be required.

Figure 4.35 Insertion Into a Sequential String

Figure 4.36 Deletion of a Substring

Linked List--**Fixed Size Nodes.** An alternative to sequential string representation is a linked list representation. Available memory is divided into nodes of a fixed size. Each node has two fields: DATA and LINK. The size of a node is the number of characters that can be stored in the DATA field. Figure 4.37 shows the division of memory into nodes of size 4 with a link field that is two characters long. On a computer with 6 bits/character this would permit link values in the range $[0,2^{12}- 1]$. In the purest form of a linked list representation of strings, each node would be of size one. Normally, this would represent extreme wastage of space.

Figure 4.37 Partitioning Available Memory Into Nodes of Size 4.

With a link field of size two characters, this would mean that only 1/3 of available memory would be available to store string information while the remaining 2/3 will be used only for link information. With a node size of 8, 80% of available memory could be used for string information. When using nodes of size > 1, it is possible that a string may need a fractional number of nodes. With a node size of 4, a string of length 13 will need only 3-1/4 nodes. Since fractional nodes cannot be allocated, 4 full nodes may be used with the last three characters of the last node set to Φ (figure 4.38(a)). An in place insertion might require one node to be split into two as in figure 4.38(b). Deletion of a substring can be carried out by replacing all characters in this substring by Φ and freeing nodes in which the DATA field consists only of Φ's. In place replacement can be carried similarly. Storage management is very similar to that of section 4.3. Storage compaction may be carried out when there are no free nodes. Strings containing many occurrences of Φ could be compacted freeing several nodes. String representation with variable sized nodes is similar.

Figure 4.38

When the node size is l things work out very smoothly. Insertion, deletion and concatenation are particularly easy. The length may be determined easily by retaining a head node with this information. Let us look more closely at the operations of insertion and concatenation when the node size is one and no head node is maintained. First, let us write a procedure which takes two character strings and inserts

the second after the *i*th character of the first.

procedure SINSERT(S,T,*i*)

//insert string T after the i-th character of S destroying the original//

//strings S,T and creating a new string S//

1   **case**   //degenerate cases//

2   :*i* < 0 **or** *i* > LENGTH(S): **print** ('string length error'); **stop**

3   : T = 0: **return**

4   : S = 0: S ☐ T; **return**

5   **end**

//at this point LENGTH(S) > 0, LENGTH(T) > 0, 0 ≤ i ≤

LENGTH(S)//

6   ptr ☐ S; j ☐ 1

7   **while** j < i **do**   //find i-th character of S//

8   *ptr* ☐ LINK (ptr); j ☐ j + 1   //ptr points to i-th node//

9   **end**

10   **if** i = 0 **then** [save ☐ S; ptr ☐ T; S ☐ T]   //save i + 1 character//

11   **else** [save ☐ LINK (ptr)

12   LINK (ptr) ☐ T]   //attach list T to list S//

13   **while** LINK (ptr) ≠ 0 **do**   //find end of T//

```
14      ptr ☐ LINK (ptr)

15      end

16    LINK (ptr) ☐ save        //point end of T to//

//i + 1-st character of S//

17   end SINSERT
```

Examine how the algorithm works on the strings below.

☐

T is to be inserted after the fifth character of *S* to give the result 'THIS NOW IS.' The fifth character in S and the last character of *T* are blanks. After the **if** statement is executed in SINSERT the following holds.

☐

The list *S* has been split into two parts named *S* and *save*. The node pointed at by *ptr* has been attached to *T*. This variable will now move across the list *T* until it points to the last node. The LINK field will be set to point to the same node that *save* is pointing to, creating a single string.

☐

Figure 4.39 Circular Representation of S and T

The computing time of SINSERT is proportional to i + LENGTH(*T*). We can produce a more efficient algorithm by altering the data structure only slightly. If we use singly linked circular lists, then *S* and *T* will be represented as in Figure 4.39 above. The new version of SINSERT is obtained by replacing lines 6-16 by:

```
ptr S; j ☐ 0

while j < i do

ptr ☐ LINK (ptr); j ☐ j + 1   //find i-th character of S//

end
```

```
save □ LINK (ptr)      //save i + 1-st character of S//

else LINK (ptr) □ LINK (T)

LINK (T) □ save        //attach end of T to S//

if ptr = S and i ≠ 0 then S □ T
```

By using circular lists we avoided the need to find the end of list *T*. The computing time for this version is *O*(*i*) and is independent of the length of *T*.

4.11.2 PATTERN MATCHING IN STRINGS

Now let us develop an algorithm for a more sophisticated application of strings. Given two strings *S* and PAT we regard the value of PAT as a pattern to be searched for in *S*. If it occurs, then we want to know the node in *S* where PAT begins. The following procedure can be used.

This algorithm is a straightforward consequence of the data representation. Unfortunately, it is not very efficient. Suppose

S = '*aaa ... a*'; PAT = '*aaa ... ab*'

where LENGTH(S) = *m*, LENGTH(PAT)= *n* and *m* is much larger than *n*. Then the first *n* -1 letters of PAT will match with the *a*'s in string *S* but the *n*-th letter of PAT will not. The pointer *p* will be moved to the second occurrence of '*a*' in *S* and the *n* - 1 *a*'s of PAT will match with *S* again. Proceeding in this way we see there will be *m* - *n* + 1 times that *S* and PAT have *n* - 1 *a*'s in common. Therefore,

```
procedure FIND (S,PAT,i)

//find in string S the first occurrence of the string PAT and return

i as a pointer to the node in S where PAT begins. Otherwise return

i as zero//

i □ 0

if PAT = 0 or S = 0 then return
```

```
p ⬚ S

repeat

save ⬚ p; q ⬚ PAT        //save the starting position//

while p ≠ 0 and q ≠ 0 and DATA(p) = DATA(q) do

p ⬚ LINK (p); q ⬚ LINK (q)        //characters match,

move to next pair//

end

if q = 0 then [i ⬚ save; return]    //a match is found//

p ⬚ LINK (save)            //start at next character in S//

until p = 0      //loop until no more elements in S//

end FIND
```

algorithm FIND will require at least $(m - n + 1)(n - 1) = O(mn)$ operations. This makes the cost of FIND proportional to the product of the lengths of the two lists or quadratic rather than linear.

There are several improvements that can be made. One is to avoid the situation where LENGTH(PAT) is greater than the remaining length of *S* but the algorithm is still searching for a match. Another improvement would be to check that the first and last characters of PAT are matched in *S* before checking the remaining characters. Of course, these improve- will speed up processing on the average. Procedure NFIND incorporates these improvements.

If we apply NFIND to the strings *S* = 'aa ... a' and PAT = 'a ... ab', then the computing time for these inputs is $O(m)$ where m = LENGTH(S) which is far better than FIND which required $O(mn)$. However, the worst case computing time for NFIND is still $O(mn)$.

NFIND is a reasonably complex program employing linked lists and it should be understood before one reads on. The use of pointers like *p*, *q*, *j*, *r* is very typical for programs using this data representation.

It would be far better if we could devise an algorithm which works in time $O(\text{LENGTH}(S) + \text{LENGTH}(\text{PAT}))$, which is linear for this problem since we must certainly look at all of PAT and potentially all of

S.

```
procedure NFIND (S,PAT,i)

//string S is searched for PAT and i is set to the first node in S

where PAT occurs else zero. Initially the last and first characters

of PAT are checked for in S//

i ← 0

if PAT = 0 or S = 0 then return

p ← q ← PAT;t ← 0

while LINK (q) ≠ 0 do          //q points to last node of PAT//

q ← LINK(q); t ← t + 1

end       //t + 1 is the length of PAT//

j ← r ← save ← S

for k ← 1 to t while j ≠ 0 do      //find t + 1-st node of S//

j ← LINK (j)

end

while j ≠ 0 do       //while S has more chars to inspect//

p ← PAT; r ← save

if DATA(q) = DATA(j)      //check last characters//

then [while DATA(p) = DATA(r) and p ≠ q do

p ← LINK(p); r ← LINK(r)     //check pattern//
```

end

if *p* = *q* **then** [*i* □ *save*; **return**]]   //success//

save □ *LINK (save); j* □ *LINK(j)*

end

end NFIND

□

Figure 4.40 Action of NFIND on S and PAT

Another desirable feature of any pattern finding algorithm is to avoid rescanning the string *S*. If *S* is so large that it cannot conveniently be stored in memory, then rescanning adds complications to the buffering operations. Such an algorithm has recently been developed by Knuth, Morris and Pratt. Using their example suppose

```
PAT = 'a b c a b c a c a b'
```

Let $S = s_1 s_2 ... s_m$ and assume that we are currently determining whether or not there is a match beginning at s_i. If $s_i \neq a$ then clearly, we may proceed by comparing s_{i+1} and *a*. Similarly if $s_i = a$ and $s_{i+1} \neq b$ then we may proceed by comparing s_{i+1} and *a*. If $s_i s_{i+1} = ab$ and $s_{i+2} \neq c$ then we have the situation:

```
S = ' - a b ? ? ? . . . . ?'

PAT =   ' a b c a b c a c a b'
```

The ? implies we do not know what the character in *S* is. The first ? in *S* represents s_{i+2} and $s_{i+2} \neq c$. At this point we know that we may continue the search for a match by comparing the first character in PAT with s_{i+2}. There is no need to compare this character of PAT with s_{i+1} as we already know that s_{i+1} is the same as the second character of PAT, *b* and so $s_{i+1} \neq a$. Let us try this again assuming a match of the first four characters in PAT followed by a non-match i.e. $s_{i+4} \neq b$. We now have the situation:

```
S = ' - a b c a ? ? . . . ?'

PAT =   ' a b c a b c a c a b'
```

We observe that the search for a match can proceed by comparing s_{i+4} and the second character in PAT, b. This is the first place a partial match can occur by sliding the pattern PAT towards the right. Thus, by knowing the characters in the pattern and the position in the pattern where a mismatch occurs with a character in S we can determine where in the pattern to continue the search for a match without moving backwards in S. To formalize this, we define a failure function for a pattern. **Definition**: If $P = p_1 p_2 \ldots p_n$ is a pattern, then its *failure function*, f, is defined as:

☐

For the example pattern above, PAT = abcabcacab, we have

```
j   1 2 3 4 5 6 7 8 9 10

PAT   a b c a b c a c a b

f   0 0 0 1 2 3 4 0 1 2
```

From the definition of the failure function we arrive at the following rule for pattern matching: *If a partial match is found such that* $s_{i-j+1} \ldots s_{i-1} = p_1 p_2 \ldots p_{j-1}$ *and* $s_i \neq p_j$ *then matching may be resumed by comparing* s_i `and` $Pf(j-1)+1$ *If* $j \neq 1$. *If* $j = 1$, *then we may continue by comparing* s_{i+1} *and* p_1.

In order to use the above rule when the pattern is represented as a linked list as per our earlier discussion, we include in every node representing the pattern, an additional field called NEXT. If LOC (j) is the address of the node representing p_j, $1 \leq j \leq n$, then we define

☐

Figure 4.41 shows the pattern PAT with the NEXT field included.

With this definition for NEXT, the pattern matching rule translates to the following algorithm:

procedure *PMATCH (S,PAT)*

```
//determine if PAT is a substring of S//

1      i ☐ S          //i will move through S//

2      j ☐ PAT            //j will move through PAT//
```

```
3         while i ≠ 0 and j ≠ 0 do

4              if DATA (i) = DATA (j)

5                   then [//match//

6                        i ☐ LINK (i)

7                        j ☐ LINK (j)]

8              else [if j = PAT then i ☐ LINK(i)

9                              else j ☐ NEXT(j]

10        end

11        if j = 0 then print ('a match')

12             else print ('no match')

13   end PMATCH
```

The correctness of PMATCH follows from the definitions of the failure function and of NEXT. To determine the computing time, we observe that the **then** clauses of lines 5-7 and 8-9 can be executed for a total of at most m = LENGTH(S) times as in each iteration i moves right on S but i never moves left in the algorithm. As a result j can move right on PAT at most m times (line 7). Since each execution of the **else** clause in line 9 moves j left on PAT, it follows that this clause can be executed at most m times as otherwise j must fall off the left end of PAT. As a result, the maximum number of iterations of the **while** loop of lines 3-10 is m and the computing time of PMATCH is $O(m)$. The performance of the algorithm may be improved by starting with a better failure function (see exercise 57).

☐

Figure 4.41 Augmented Pattern PAT with NEXT Field

The preceding discussion shows that NFIND can be improved to an $O(m)$ algorithm provided we are either given the failure function or NEXT. We now look into the problem of determining f. Once f is known, it is easy to get NEXT. From exercise 56 we know that the failure function for any pattern $p_1 p_2 ... p_n$ is given by:

(note that $f^1(j) = f(j)$ and $f^m(j) = f(f^{m-1}(j))$)

This directly yields the following algorithm to compute f.

procedure *FAIL (PAT,f)*

//compute the failure function for PAT = $p_1 p_2 \ldots p_n$//

1   *f(1)* ☐ 0

2   **for** *j* ☐ 2 **to** *n* **do**   *//compute f(j)//*

3   *i* ☐ *f(j - 1)*

4   **while** $p_j \neq p_{i+1}$ **and** *i* > 0 **do**

5   *i* ☐ *f(i)*

6   **end**

7   **if** $p_j = p_{i+1}$ **then** *f(j)* ☐ *i + 1*

8   **else** *f(j)* ☐ 0

9   **end**

10 **end** *FAIL*

In analyzing the computing time of this algorithm we note that in each iteration of the **while** loop of lines 4-6 the value of *i* decreases (by the definition of *f*). The variable *i* is reset at the beginning of each iteration of the **for** loop. However, it is either reset to 0 (when *j* = 2 or when the previous iteration went through line 8) or it is reset to a value 1 greater than its terminal value on the previous iteration (i.e. when the previous iteration went through line 7). Since only *n* executions of line 3 are made, the value of *i* therefore has a total increment of at most *n*. Hence it cannot be decremented more than *n* times. Consequently the **while** loop of lines 4-6 is iterated at most *n* times over the whole algorithm and the computing time of FAIL is *O(n)*.

Even if we are not given the failure function for a pattern, pattern matching can be carried out in time $O(n + m)$. This is an improvement over NFIND.

4.12 IMPLEMENTING NODE STRUCTURES

Throughout this chapter we have dealt with nodes without really seeing how they may be set up in any high level programming language. Since almost all such languages provide array variables, the easiest way to establish n nodes is to define one dimensional arrays, each of size n, for each of the fields in a node. The node structure to represent univariate polynomials (section 4.4) consisted of three fields: EXP, COEF and LINK. Assuming integer coefficients, 200 polynomial nodes can be set up in FORTRAN through the statement:

INTEGER EXP(200), COEF(200), LINK(200)

The nodes then have indices 1 through 200. The values of various fields in any node i, $1 \le i \le 200$ can be determined or changed through the use of standard assignment statements. With this structure for nodes, the following FORTRAN subroutine would perform the function of algorithm INIT of section 4.3.

```
      SUBROUTINE INIT(N)

C     LINK N NODES TO FORM THE INITIAL AVAILABLE

C     SPACE LIST AND SET AV TO POINT TO FIRST NODE.

C     ARRAYS EXP, COEF AND LINK AND VARIABLE AV

C     ARE IN BLANK COMMON.

      INTEGER EXP(200),COEF(200),LINK(200),AV

      COMMON EXP,COEF,LINK,AV

      M = N - 1

      DO 10 I = 1, M

10    LINK(I) = I + 1

      LINK(N) = 0
```

```
      AV= 1

      RETURN

      END
```

Using global one dimensional arrays for each field, the function INIT can be realized in PASCAL by the procedure:

```
PROCEDURE INIT(N: INTEGER);

{This procedure initializes the available space list. LINK is a global

array of type integer. AV is a global variable of type integer.}

VAR I: INTEGER; {Local variable I}

BEGIN FOR I: = 1 TO N - 1 DO LINK(I): = I + 1;

LINK(N): = 0;

AV: = 1

END;
```
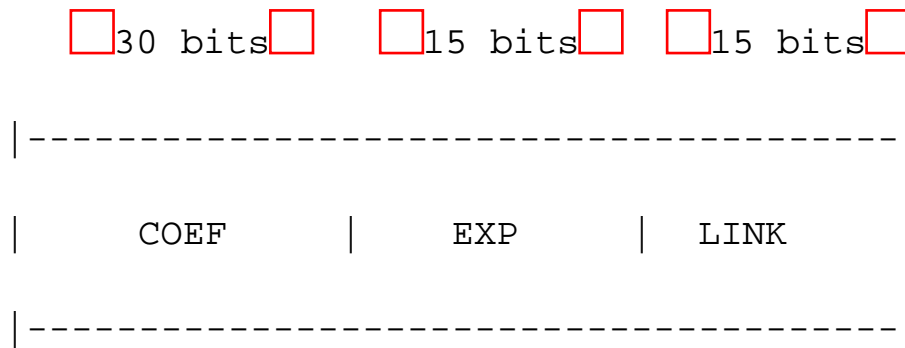
Usually one does not need a full memory word for each field. So space can be saved by packing several fields into one word. If we have a 60 bit word, as in a CDC computer, choosing field sizes of 15 bits for the EXP field, 15 bits for the LINK field and 30 bits for the COEF field would permit the use of one word per node. This node structure would be adequate to handle LINK and EXP values in the range [0,32767] and coefficient values in excess of the range $[-5 \times 10^8, + 5 \times 10^8]$. On a 32 bit per word machine roughly the same ranges could be obtained through the use of two words per node. One word of each node could be used for the COEF field and half of the second word would be used for each of the remaining two fields. In the case of 60 bit words, this packing of fields would reduce storage requirements by 2/3, while in the 32 bit case the reduction would be by a factor of 1/3 (this, of course, assumes that the field sizes discussed above are enough to handle the problem being solved). If we are going to pack fields into words, then we must have a way to extract and set field values in any node. In a language like FORTRAN this can be done through the use of the SHIFT function together with masking operations using the logical operations of AND, OR and NOT. The SHIFT function has two arguments, X and N, where X is the word whose bits are to be shifted and N is the amount of the shift. If $N > 0$, then the result of the shift is X shifted circularly left by N bits. The value of X is left unchanged. If $N < 0$, then the result is X shifted right with sign extension by N bits. X is left unchanged. Figure 4.42 illustrates the

result of a left circular and right shift on the bits of a 5 bit word. Figure 4.43 lists FORTRAN subroutines to extract and set the three fields of a 1 word node. Figure 4.44 does this for the two fields packed into 1 word for the case of a 32 bit word.

```
    30 bits       15 bits     15 bits

      |--------------------------------------|

      |     COEF      |    EXP    |  LINK     |

      |--------------------------------------|
```

            node structure using one 60 bit word per node

```
  INTEGER FUNCTION COEF(J)                SUBROUTINE SCOEF(J,I)

C EXTRACT COEF FIELD OF J BY          C SET THE COEF FIELD OF J TO I
BY

C USING A RIGHT SHIFT BY 30 BITS.     C FIRST ZEROING OUT OLD

  COEF = SHIFT(J,-30)                 C CONTENTS AND THEN USE AN OR

  RETURN                              C TO PUT IN NEW VALUE. ASSUME
I IS

  END                                 C IN RIGHT RANGE.

                                         J = (J.AND.7777777777B).OR.

                                            SHIFT(I.AND.7777777777B,30)

                                         RETURN

                                         END

  INTEGER FUNCTION EXP(J)                SUBROUTINE SEXP(J,I )

C EXTRACT EXP FIELD BY SHIFTING       C SET EXP FIELD OF J TO I

C RIGHT AND ZEROING OUT EXTRA
```

```
   BITS                                J = (J.
AND.7777777770000077777B)

   EXP = SHIFT(J, - 15).AND.77777B         .OR.SHIFT(I,15)

   RETURN                              RETURN

   END                                END

   FUNCTION LINK(J)                    SUBROUTINE SLINK(J,I)

C EXTRACT LINK FIELD OF J            C SET LINK FIELD OF J TO I

   LINK = J.AND.77777B                J = (J.
AND.7777777777777700000B)

   RETURN                                 .OR. I

   END                                RETURN

   END
```
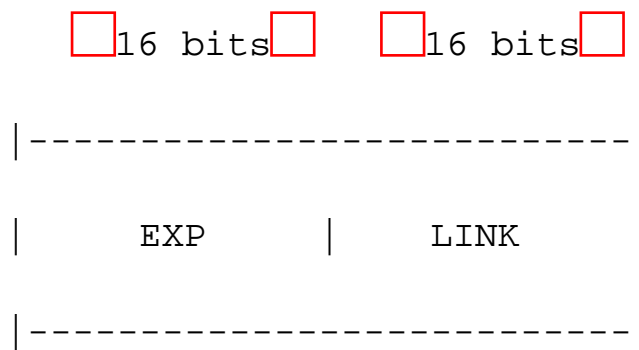
Figure 4.43 Setting and Extracting Fields in a 60 Bit Word (EXP, LINK nonnegative integers)

```
        □16 bits□    □16 bits□

        |--------------------------|

        |    EXP    |   LINK    |

        |--------------------------|
```

```
   INTEGER FUNCTION EXP(J)             SUBROUTINE SEXP(J,I)

C EXTRACT EXP FIELD OF J             C SET EXP FIELD OF J TO I

   EXP = SHIFT(J, - 16).AND.177777B    J = (J.AND.177777B).OR.SHIFT
(I,16)
```

```
  RETURN   RETURN

  END   END
```

(NOTE: the AND operation is needed

        to mask out sign bit extension)

```
  INTEGER FUNCTION LINK(J)              SUBROUTINE SLINK(J,I)

C EXTRACT LINK FIELD OF J            C SET LINK FIELD OF J TO I

  LINK = J.AND. 177777B                J = (J.AND.37777600000B).OR.I

  RETURN   RETURN

  END   END
```

These subroutines and functions assume that negative numbers are
stored using '1s complement arithmetic. A "B' following a constant
means that the constant is an octal constant.

Figure 4.44 Setting and Extracting Fields in a 32 Bit Word

```
VAR LINK:   ARRAY [1 . . 200] OF 0 . . 32767;

     EXP:   ARRAY [1 . . 200] OF 0 . . 32767;

    COEF:   ARRAY [1 . . 200] OF - 1073741823 . . 1073741823;
```

Figure 4.45 PASCAL Array Declaration to Get Field Sizes

Figure 4.42 Left and Right Shifts

In PASCAL, the task of packing fields into words is simplified or eliminated as the language definition itself permits specifying a range of values for each variable. Thus, if the range of variable I is defined to be $[0,2^{15} - 1]$, then only 15 bits are allocated to I. The PASCAL declaration, figure 4.45 for arrays EXP, COEF and LINK would result in the same storage requirements for 200 nodes as the packing and

unpacking routines of figure 4.44. Alternatively, one could use record variables as in figure 4.46.

```
TYPE POLY = PACKED RECORD COEF: -1073741823 . . 1073741823

                              EXP: 0 . . 32767

                              LINK: 0 . . 32767

        END;

VAR NODE: ARRAY [1 . . 200] OF POLY

usage ... NODE [subscript]. COEF; NODE [subscript]. EXP; NODE
[subscript]. LINK
```

Figure 4.46 Declaring NODE(200) to be a Record Variable With Fields EXP, LINK and COEF of the Same Size as in Figure 4.45

REFERENCES

A general reference for the topics discussed here is *The Art of Computer Programming: Fundamental Algorithms* by D. Knuth, volume I 2nd edition, Addison-Wesley, Reading, 1973.

For a discussion of early list processing systems see "Symmetric list processor" by J. Weizenbaum, *CACM*, vol. 6, no. 9, Sept. 1963, pp. 524-544.

"A comparison of list processing computer languages," by Bobrow and B. Raphael, *CACM*, vol. 7, no. 4, April 1964, pp. 231-240.

"An introduction to IPL-V" by A. Newell and F. M. Tonge, *CACM*, vol. 3, no. 4, April 1960, pp. 205-211.

"Recursive functions of symbolic expressions and their computation by machine: I," by J. McCarthy, *CACM*, vol. 3, no. 4, April 1960, pp. 184-195.

For a survey of symbol manipulation systems see *Proceedings of the Second Symposium on Symbolic and Algebraic Manipulation*, ed. S. R. Petrick, March, 1971, available from ACM.

For further articles on dynamic storage allocation see "An estimate of the store size necessary for dynamic storage allocation" by J. Robson, *JACM*, vol. 18, no. 3, July 1971, pp. 416-423.

"Multiword list items" by W. T. Comfort, *CACM*, vol. 7, no. 6, June 1964, pp. 357-362.

"A weighted buddy method for dynamic storage allocation" by K. Shen and J. Peterson, *CACM*, vol. 17, no. 10, October 1974, pp. 558-562.

"Statistical properties of the buddy system," by P. Purdom and S. Stigler *JACM*, vol. 14, no. 4, October 1970, pp. 683-697.

"A class of dynamic memory allocation algorithms," by D. Hirschberg, *CACM*, vol. 16, no. 10, October 1973, pp. 615-618.

The reference count technique for lists was first given in "A method for overlapping and erasure of lists," by G. Collins, *CACM*, vol. 3, no. 12, December 1960, pp. 655-657.

The algorithm MARK2 is adapted from: "An efficient machine-independent procedure for garbage collection in various list structures," by H. Schorr and W. Waite, *CACM*, vol. 10, no. 8, Aug. 1967, p. 501-506.

More list copying and marking algorithms may be found in "Copying list structures using bounded workspace," by G. Lindstrom, *CACM*, vol. 17, no. 4, April 1974, p. 198-202, "A nonrecursive list moving algorithm," by E. Reingold, *CACM*, vol. 16, no. 5, May 1973, p. 305-307, "Bounded workspace garbage collection in an address-order preserving list processing environment," by D. Fisher, *Information Processing Letters*, vol. 3, no. 1, July 1974, p. 29-32.

For string processing systems see the early works "COMIT" by V. Yngve, *CACM*. vol. 6, no. 3, March 1963, pp. 83-84. SNOBOL, A String Manipulation Language by D. Farber, R. Griswold and I. Polonsky, *JACM*, vol. 11, no. 1, 1964, pp. 21-30. "String processing techniques" by S. Madnick, *CACM*, vol. 10, no. 7, July 1967, pp. 420-427. "Fast Pattern Matching in Strings" by D. Knuth, J. Morris and V. Pratt Stanford Technical Report #74-440, August 1974.

EXERCISES

For exercises 1-6 assume that each node has two fields: DATA and LINK. Also assume the existence of subalgorithms GETNODE(*X*)and RET(*X*)to get and return nodes from/to the storage pool.

1. Write an algorithm LENGTH(*P*) to count the number of nodes in a singly linked list *P*, where *P* points to the first node in the list. The last node has link field 0.

2. Let *P* be a pointer to the first node in a singly linked list and *X* an arbitrary node in this list. Write an algorithm to delete this node from the list. If *X* = *P*, then *P* should be reset to point to the new first node

in the list.

3. Let $X = (x_1, x_2, ..., x_n)$ *and* $Y = (y_1, y_2, ..., y_m)$ be two linked lists. Write an algorithm to merge the two lists together to obtain the linked list $Z = (x_1, y_1, x_2, y_2, ..., x_m, y_m, x_{m+1}, ... x_n)$ if $m \leq n$ and $Z = (x_1, y_1, x_2, y_2, ..., x_n, y_n, y_{n+1}, ..., y_m)$ if $m > n$. No additional nodes may be used.

4. Do exercise 1 for the case of circularly linked lists.

5. Do exercise 2 for the case of circularly linked lists.

6. Do exercise 3 for the case of circularly linked lists.

7. Devise a representation for a list where insertions and deletions can be made at either end. Such a structure is called a deque. Write a procedure for inserting and deleting at either end.

8. Consider the hypothetical data object $X2$. $X2$ is a linear list with the restriction that while additions to the list may be made at either end, deletions can be made from one end only.

Design a linked list representation for $X2$. Write addition and deletion algorithms for $X2$. Specify initial and boundary conditions for your representation.

9. Give an algorithm for a singly linked circular list which reverses the direction of the links.

10. Let P be a pointer to a circularly linked list. Show how this list may be used as a queue. I.e., write algorithms to add and delete elements. Specify the value for P when the queue is empty.

11. It is possible to traverse a singly linked list in both directions (i.e., left to right and a restricted right to left traversal) by reversing the links during the left to right traversal. A possible configuration for a list P under this scheme would be:

P points to the node currently being examined and L to the node on its left. Note that all nodes to the left of P have their links reversed.

i) Write an algorithm to move P, n nodes to the right from a given position (L,P).

ii) Write an algorithm to move P, n nodes left from any given position (L,P).

12. Consider the operation XOR (exclusive OR, also written as $\oplus$) defined as below (for i,j binary):

☐

This differs from the usual OR of logic in that

☐

The definition can be extended to the case where i and j are binary strings (i.e., take the XOR of corresponding bits of i and j). So, for example, if i = 10110 and j = 01100, then i XOR j = i $\oplus$ j = 11010.

Note that a $\oplus$ (a $\oplus$ b) = (a $\oplus$ a) = b

and that (a $\oplus$ b) $\oplus$ = a $\oplus$ (b $\oplus$ b) = a

This gives us a space saving device for storing the right and left links of a doubly linked list. The nodes will now have only two fields: INFO and LINK. If L_1 is to the left of node X and R_1 to its right, then LINK(X) = L1 $\oplus$ R1. For the leftmost node L_1 = 0 and for the rightmost node R_1 = 0.

Let (L,R) be a doubly linked list so represented. L points to the left most node and R to the right most node in the list.

i) Write an algorithm to traverse the doubly linked list (*L,R*) from left to right listing out the contents of the INFO field of each node.

ii) Write an algorithm to traverse the list right to left listing out the contents of the INFO field of each node.

13. Write an algorithm PREAD(*X*) to read in *n* pairs of coefficients and exponents, (c_i,e_i) $1 \le i \le n$ of a univariate polynomial, *X*, and to convert the polynomial into the circular linked list structure of section 4.4. Assume $e_1 > e_{i+1}$, $1 \le i < n$, and that $c_i \ne 0$, $1 \le i \le n$. Your algorithm should leave *X* pointing to the head node. Show that this operation can be performed in time *O(n)*.

14. Let *A* and *B* be pointers to the head nodes of two polynomials represented as in exercise 13. Write an algorithm to compute the product polynomial *C = A * B*. Your algorithm should leave *A* and *B* unaltered and create *C* as a new list. Show that if *n* and *m* are the number of terms in *A* and *B* respectively, then this multiplication can be carried out in time $O(nm^2)$ or $O(mn^2)$. If *A, B* are dense show that the multiplication takes *O(mn)*.

15. Let *A* be a pointer to the head node of a univariate polynomial as in section 4.4. Write an algorithm, PEVAL(*A,x*) to evaluate the polynomial *A* at the point *x*, where *x* is some real number.

16. Extend the equivalence algorithm of section 4.6 to handle the case when the variables being

equivalenced may be subscripted. The input now consists of 4-tuples $(i, ioff, j, joff)$ where i and j as before represent the variables. Ioff and joff give the positions in i and j being equivalenced relative to the start of i or j. For example, if i represents the array A dimensioned $A(-6{:}10)$ and j, the array B dimensioned $B(1{:}20)$, the equivalence EQUIVALENCE$(A(5),B(6))$ will be represented by 4-tuple $(i,12, j,6)$. Your algorithm will begin by reading in these 4-tuples and setting them up in lists as in section 4.6. Each node on a list will have three fields: IOFF, JVAR, JOFF. Thus, for the equivalence 4-tuple above, the node ☐ will be put into the list for i and the node ☐ onto the list for j. Now process these lists outputting equivalence classes. With each class, output the relative position of each member of the class and check for conflicting equivalences .

In exercises 17-21 the sparse matrices are represented as in section 4.7

17. Let A and B be two sparse matrices represeted as in section 4.7. Write an algorithm. MADD(A,B,C) to create the matrix $C = A + B$. Your algorithm should leave the matrices A and B unchanged and set up C as a new matrix in accordance with this data representation. Show that if A and B are n x m matrices with r_A and r_B nonzero terms, then this addition can be carried out in $O(n + m + r_A + r_B)$ time.

18. Let A and B be two sparse matrices. Write an algorithm MMUL(A,B,C) to set up the structure for $C = A * B$. Show that if A is a n x m matrix with r_A nonzero terms and if B is a m x p matrix with r_B nonzero terms, then C can be computed in time $O(pr_A + nr_B)$. Can you think of a way to compute C in $O(min\{pr_A,nr_B\})$?

19. Write an algorithm to write out the terms of a sparse matrix A as triples (i,j,a_{ij}). The terms are to be output by rows and within rows by columns. Show that this operation can be performed in time $O(n + r_A)$ if there are r_A nonzero terms in A and A is a n x m matrix.

20. Write an algorithm MTRP(A,B) to compute the matrix $B = A^T$, the transpose of the sparse matrix A. What is the computing time of your algorithm?

21. Design an algorithm to copy a sparse matrix. What is the computing time of your method?

22. A simpler and more efficient representation for sparse matrices can be obtained when one is restricted to the operations of addition, subtraction and multiplication. In this representation nodes have the same fields as in the representation of section 4.7. Each nonzero term is represented by a node. These nodes are linked together to form two circular lists. The first list, the rowlist, is made up by linking nodes by rows and within rows by columns. This is done via the RIGHT field. The second list, the column list, is made up by linking nodes via the DOWN field. In this list, nodes are linked by columns and within columns by rows. These two lists share a common head node. In addition, a node is added to contain the dimensions of the matrix. The matrix A of figure 4.11 has the representation shown on page 206.

Using the same assumptions as for algorithm MREAD of section 4.7 write an algorithm to read in a matrix *A* and set up its internal representation as above. How much time does your algorithm take? How much additional space is needed?

23. For the representation of exercise 22 write algorithms to

(i) erase a matrix

(ii) add two matrices

(iii) multiply two matrices

(iv) print out a matrix

For each of the above obtain computing times. How do these times compare with the corresponding times for the representation of section 4.7?

* solid links represent row list

dashed links represent column list

Representation of matrix A of figure 4.11 using the scheme of exercise 22.

24. Compare the sparse representations of exercise 22 and section 4.7 with respect to some other operations. For example how much time is needed to output the entries in an arbitrary row or column?

25. (a) Write an algorithm *BF(n,p)* similar to algorithm *FF* of section 4.8 to allocate a block of size *n* using a *best fit* strategy. Each block in the chain of available blocks has a SIZE field giving the number of words in that block. The chain has a head node, *AV* (see Figure 4.16). The best fit strategy examines each block in this chain. The allocation is made from the smallest block of size $\geq n$. *P* is set to the starting address of the space allocated.

(b) Which of the algorithms BF and FF take less time?

(c) Give an example of a sequence of requests for memory and memory freeing that can be met by BF but not by FF.

(d) Do (c) for a sequence that can be met by FF but not by BF.

26. Which of the two algorithms ALLOCATE and FREE of section 4.8 require the condition $TAG(0) = TAG(m + 1) = 1$ in order to work right? Why?

27. The boundary tag method for dynamic storage management maintained the available space list as a doubly linked list. Is the XOR scheme for representing doubly linked lists (see exercise 12) suitable for this application? Why?

28. Design a storage management scheme for the case when all requests for memory are of the same size, say k. Is it necessary to coalesce adjacent blocks that are free? Write algorithms to free and allocate storage in this scheme.

29. Consider the dynamic storage management problem in which requests for memory are of varying sizes as in section 4.8. Assume that blocks of storage are freed according to the LAFF discipline (Last Allocated First Freed).

i) Design a structure to represent the free space.

ii) Write an algorithm to allocate a block of storage of size n.

iii) Write an algorithm to free a block of storage of size n beginning at p.

30. In the case of static storage allocation all the requests are known in advance. If there are n requests r_1, r_2, ...,r_n and $\Sigma r_i \leq M$ where M is the total amount of memory available, then all requests can be met. So, assume $\Sigma r_i > M$.

i) Which of these n requests should be satisfied if we wish to maximize the number of satisfied requests?

ii) Under the maximization criteria of (i), how small can the ratio

```
    storage allocated

    ----------------- get?

          M
```

iii) Would this be a good criteria to use if jobs are charged a flat rate, say $3 per job, independent of the size of the request?

iv) The pricing policy of (iii) is unrealistic when there can be much variation in request size. A more realistic policy is to charge say x cents per unit of request. Is the criteria of (i) a good one for this pricing policy? What would be a good maximization criteria for storage allocation now?

Write an algorithm to determine which requests are to be satisfied now. How much time does your algorithm take as a function of n, the number of requests? [If your algorithm takes a polynomial amount of time, and works correctly, take it to your instructor immediately. You have made a major discovery.]

31. [Buddy System] The text examined the boundary tag method for dynamic storage management. The next 6 exercises will examine an alternative approach in which only blocks of size a power of 2 will be allocated. Thus if a request for a block of size *n* is made, then a block of size $2^{\lceil \log_2 n \rceil}$ is allocated. As a result of this, all free blocks are also of size a power of 2. If the total memory size is 2^m addressed from 0 to 2^m - 1, then the possible sizes for free blocks are 2^k, $0 \le k \le m$. Free blocks of the same size will be maintained in the same available space list. Thus, this system will have $m + 1$ available space lists. Each list is a doubly linked circular list and has a head node AVAIL(*i*), $0 \le i \le m$. Every free node has the following structure:

Initially all of memory is free and consists of one block beginning at 0 and of size 2^m. Write an algorithm to initialize all the available space lists.

32. [Buddy System Allocation] Using the available space list structure of exercise 31 write an algorithm to meet a request of size *n* if possible. Note that a request of size *n* is to be met by allocating a block of size 2^k k = [$\log_2 n$]. To do this examine the available space lists AVAIL(i), $k \le i \le m$ finding the smallest *i* for which AVAIL(*i*) is not empty. Remove one block from this list. Let *P* be the starting address of this block. If $i > k$, then the block is too big and is broken into two blocks of size 2^{i-1} beginning at *P* and $P + 2^{i-1}$ respectively. The block beginning at $P + 2^{i-1}$ is inserted into the corresponding available space list. If $i - 1 > k$, then the block is to be further split and so on. Finally, a block of size 2^k beginning at *P* is allocated. A block in use has the form:

i) Write an algorithm using the strategy outlined above to allocate a block of storage to meet a request for *n* units of memory.

ii) For a memory of size $2^m = 16$ draw the binary tree representing the splitting of blocks taking place in satisfying 16 consecutive requests for memory of size 1. (Note that the use of the TAG in the allocated block does not really create a problem in allocations of size 1 since memory would be allocated in units where 1 unit may be a few thousand words.) Label each node in this tree with its starting address and present KVAL, i.e. power of 2 representing its size.

33. [Locating Buddies] Two nodes in the tree of exercise 32 are said to be buddies if they are sibling nodes. Prove that two nodes starting at *x* and *y* respectively are buddies iff:

i) the KVALS for x and y are the same: and

ii) $x = y \oplus 2^k$ where $\oplus$ is the exclusive OR (XOR) operation defined in exercise 12. The $\oplus$ is taken pair wise bit wise on the binary representation of y and 2^k.

34. [Freeing and Coalescing Blocks] When a block with KVAL k becomes free it is to be returned to the available space list. Free blocks are combined into bigger free blocks iff they are buddies. This combining follows the reverse process adopted during allocation. If a block beginning at P and of size k becomes free, it is to be combined with its buddy $P \oplus 2^k$ if the buddy is free. The new free block beginning at $L = \min \{P, P \oplus 2^k\}$ and of size $k + 1$ is to be combined with its buddy $L \oplus 2^{k+1}$ if free and so on. Write an algorithm to free a block beginning at P and having KVAL k combining buddies that are free.

35. i) Does the freeing algorithm of exercise 34 always combine adjacent free blocks? If not, give a sequence of allocations and freeings of storage showing this to be the case.

storage requested

ii) How small can the ratio------------------------be for the Buddy System?

storage allocated

Storage requested $= \Sigma n_i$ where n_i is actual amount requested. Give an example approaching this ratio.

iii) How much time does the allocation algorithm take in the worst case to make an allocation if the total memory size is 2^m?

iv) How much time does the freeing algorithm take in the worst case to free a block of storage?

36. [Buddy system when memory size is not a power of 2]

i) How are the available space lists to be initialized if the total storage available is not a power of 2?

ii) What changes are to be made to the block freeing algorithm to take care of this case? Do any changes have to be made to the allocation algorithm?

37. Write a nonrecursive version of algorithm LERASE(X) of section 4.9.

38. Write a nonrecursive version of algorithm EQUALS(S,T) of section 4.9.

39. Write a nonrecursive version of algorithm DEPTH(*S*) of section 4.9.

40. Write a procedure which takes an arbitrary nonrecursive list *L* with no shared sublists and inverts it and all of its sublists. For example, if *L* = (*a*, (*b*,*c*)), then inverse (*L*) = ((*c*,*b*),*a*).

41. Devise a procedure that produces the list representation of an arbitrary list given its linear form as a string of atoms, commas, blanks and parentheses. For example, for the input *L* = (*a*,(*b*,*c*)) your procedure should produce the structure:

42. One way to represent generalized lists is through the use of two field nodes and a symbol table which contains all atoms and list names together with pointers to these lists. Let the two fields of each node be named ALINK and BLINK. Then BLINK either points to the next node on the same level, if there is one, or is a zero. The ALINK field either points to a node at a lower level or, in the case of an atom or list name, to the appropriate entry in the symbol table. For example, the list *B*(*A*,(*D*,*E*),( ),*B*) would have the representation:

(The list names D and E were already in the table at the time the list B was input. A was not in the table and so assumed to be an atom.)

The symbol table retains a type bit for each entry. Type = 1 if the entry is a list name and type = 0 for atoms. The NIL atom may either be in the table or ALINKS can be set to 0 to represent the NIL atom. Write an algorithm to read in a list in parenthesis notation and to set up its linked representation as above with X set to point to the first node in the list. Note that no head nodes are in use. The following subalgorithms may be used by LREAD:

i) GET(*A*,*P*) ... searches the symbol table for the name *A*. *P* is set to 0 if *A* is not found in the table, otherwise, *P* is set to the position of *A* in the table.

ii) PUT(*A*,*T*,*P*) ... enters *A* into the table. *P* is the position at which *A* was entered. If *A* is already in the table, then the type and address fields of the old entry are changed. *T* = 0 to enter an atom or *T* > 0 to enter a list with address *T*. (*Note*: this permits recursive definition of lists using indirect recursion).

iii) NEXT TOKEN ... gets next token in input list. (A token may be a list name, atom,'(',')'or ','. A '⊥'is returned if there are no more tokens.)

iv) GETNODE(*X*) ... gets a node for use.

You may assume that the input list is syntactically correct. In case a sublist is labeled as in the list C(D,E

(F,G)) the structure should be set up as in the case C(D,(F,G)) and E should be entered into the symbol table as a list with the appropriate storing address.

43. Rewrite algorithm MARK1 of section 4.10 using the conventions of section 4.9 for the tag field.

44. Rewrite algorithm MARK1 of section 4.10 for the case when each list and sublist has a head node. Assume that the DLINK field of each head node is free and so may be used to maintain a linked stack without using any additional space. Show that the computing time is still $O(m)$.

45. When the DLINK field of a node is used to retain atomic information as in section 4.9, implementing the marking strategy of MARK2 requires an additional bit in each node. In this exercise we shall explore a marking strategy which does not require this additional bit. Its worst case computing time will however be $O(mn)$ where m is the number of nodes marked and n the total number of nodes in the system. Write a marking algorithm using the node structure and conventions of section 4.9. Each node has the fields: MARK, TAG, DLINK and RLINK. Your marking algorithm will use variable P to point to the node currently being examined and NEXT to point to the next node to be examined. If L is the address of the as yet unexplored list node with least address and P the address of the node currently being examined then the value of NEXT will be min $\{L, P + 1\}$. Show that the computing time of your algorithm is $O(mn)$.

46. Prove that MARK2(X) marks all unmarked nodes accessible from X.

47. Write a composite marking algorithm using MARK1, MARK2 and a fixed amount M of stack space. Stack nodes as in MARK1 until the stack is full. When the stack becomes full, revert to the strategy of MARK2. On completion of MARK2, pick up a node from the stack and explore it using the composite algorithm. In case the stack never overflows, the composite algorithm will be as fast as MARK1. When $M = 0$, the algorithm essentially becomes MARK2. The computing time will in general be somewhere in between that of MARK1 and MARK2.

48. Write a storage compaction algorithm to be used following the marking phase of garbage collection. Assume that all nodes are of a fixed size and can be addressed NODE(i), $1 \le i \le m$. Show that this can be done in two phases, where in the first phase a left to right scan for free nodes and a right to left scan for nodes in use is carried out. During this phase, used nodes from the right end of memory are moved to free positions at the left end. The relocated address of such nodes is noted in one of the fields of the old address. At the end of this phase all nodes in use occupy a contiguous chunk of memory at the left end. In the second phase links to relocated nodes are updated.

49. Write a compaction algorithm to be used in conjunction with the boundary tag method for storage management. Show that this can be done in one left to right scan of memory blocks. Assume that memory blocks are independent and do not reference each other. Further assume that all memory references within a block are made relative to the start of the block. Assume the start of each in-use block is in a table external to the space being allocated.

50. Design a dynamic storage management system in which blocks are to be returned to the available space list only during compaction. At all other times, a TAG is set to indicate the block has become free. Assume that initially all of memory is free and available as one block. Let memory be addressed 1 through *M*. For your design, write the following algorithms:

(i) ALLOCATE(*n,p*) ... allocates a block of size *n*; *p* is set to its starting address. Assume that size *n* includes any space needed for control fields in your design.

(ii) FREE(*n,p*) ... free a block of size *n* beginning at *p*.

(iii) COMPACT ... compact memory and reinitialize the available space list. You may assume that all address references within a block are relative to the start of the block and so no link fields within blocks need be changed. Also, there are no interblock references.

51. Write an algorithm to make an in-place replacement of a substring of *X* by the string *Y*. Assume that strings are represented using fixed size nodes and that each node in addition to a link field has space for 4 characters. Use Φ to fill up any unused space in a node.

52. Using the definition of STRING given in section 4.11, simulate the axioms as they would apply to (i) CONCAT(*S,T*) (ii) SUBSTR(S,2,3), (iii) INDEX(*S,T*) where *S* = 'abcde' and *T* = 'cde'.

53. If $X = (x_1, ...,x_m)$ and $Y = (y_1, ...,y_n)$ are strings where x_i and y_i are letters of the alphabet, then *X* is less than *Y* if $x_i = y_i$ for $1 \le i \le j$ and $x_j < y_j$ or if $x_i = y_i$ for $1 \le i \le m$ and $m < n$. Write an algorithm which takes two strings *X,Y* and returns either -1, 0, +1 if $X < Y$, $X = Y$ or $X > Y$ respectively.

54. Let *X* and *Y* be strings represented as singly linked lists. Write a procedure which finds the first character of *X* which does not occur in the string *Y*.

55. Show that the computing time for procedure NFIND is still $O(mn)$. Find a string and a pattern for which this is true.

56. (a) Compute the failure function for the following patterns:

(i) a a a a b

(ii) a b a b a a

(iii) a b a a b a a b b

(b) For each of the above patterns obtain the linked list representations including the field NEXT as in

figure 4.41.

(c) let $p_1 p_2 \ldots p_n$ be a pattern of length n. Let f be its failure function. Define $f^1(j) = f(j)$ and $f^m(j) = f(f^{m-1}(j))$, $1 \leq j \leq n$ and $m > 1$. Show using the definition of f that:

☐

57. The definition of the failure function may be strengthened to

☐

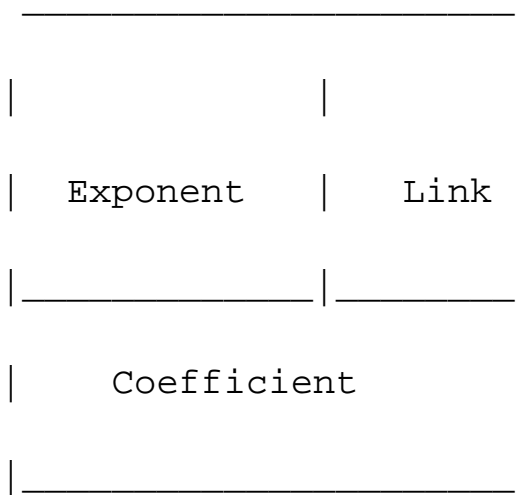(a) Obtain the new failure function for the pattern PAT of the text.

(b) Show that if this definition for f is used in the definition of NEXT then algorithm PMATCH still works correctly.

(c) Modify algorithm FAIL to compute f under this definition. Show that the computing time is still $O(n)$.

(d) Are there any patterns for which the observed computing time of PMATCH is more with the new definition of f than with the old one? Are there any for which it is less? Give examples.

58. [Programming Project]

Design a linked allocation system to represent and manipulate univariate polynomials with integer coefficients (use circular linked lists with head nodes). Each term of the polynomial will be represented as a node. Thus, a node in this system will have three fields as below:

```
     _____
    |                |       |
    |   Exponent     |  Link |
    |_____|_____|
    |                        |
    |     Coefficient        |
    |_____|
```

For purposes of this project you may assume that each field requires one word and that a total of 200 nodes are available.

The available space list is maintained as a chain with AVAIL pointing to the first node. To begin with, write computer subprograms or procedures to perform the following basic list handling tasks:

i) INIT(N) ... initialize the storage pool. I.e., set it up as a singly linked list linked through the field LINK with AVAIL pointing to the first node in this list.

ii) GETNODE(X) ... provides the node X from the storage pool for use. It is easiest to provide the first node in AVAIL if there is a free node.

iii) RET(X) ... return node X to the storage pool.

iv) LENGTH(X) ... determines the number of nodes in the list X where X may be either a circular list or a chain.

The external (i.e., for input and output) representation of a univariate polynomial will be assumed to be a sequence of integers of the form: $ne_1 \, c_1 \, e_2 \, c_2 \, e_3 \, c_3 \, ... \, e_n \, c_n$, where the e_i represents the exponents and the c_i the coefficients. n gives the number of terms in the polynomial. The exponents are in decreasing order, i.e., $e_{1 \,>} \, e_2 > \, ... \, > e_n$.

Write and test (using the routines i-iv above) the following routines:

v) PREAD (X) ... read in an input polynomial and convert it to a circular list representation using a head node. X is set to point to the head node of this polynomial.

vi) PWRITE(X) ... convert the polynomial X from its linked list representation to external representation and print it out.

Note: Both PREAD and PWRITE may need a buffer to hold the input while setting up the list or to assemble the output. Assuming that at most 10 terms may appear on an input or output line, we can use the two arrays $E(10)$ and $C(10)$ for this.

vii) PADD($X, \, Y, \, Z$) ... $Z = X + Y$

viii) PSUB($X, \, Y, \, Z$) ... $Z = X - Y$

ix) PMUL($X, \, Y, \, Z$) ... $Z = X * Y$

x) PEVAL($X, \, A, \, V$) ... A is a real constant and the polynomial X is evaluated at the point A. V is set to this value.

Note: Routines vi-x should leave the input polynomials unaltered after completion of their respective tasks.

xi) PERASE(*X*) ... return the circular list *X* to the storage pool.

Use the routine LENGTH to check that all unused nodes are returned to the storage pool.

E.g., *LO* = LENGTH(AVAIL)

CALL PMUL(*X, Y, Z*)

LN = LENGTH(*Z*) + LENGTH(AVAIL)

should result in *LN= LO*.

59. [Programming Project] In this project, we shall implement a complete linked list system to perform arithmetic on sparse matrices using the representation of section 4.7. First, design a convenient node structure assuming VALUE is an integer for the computer on which the program is to be run. Then decide how many bits will be used for each field of the node. In case the programming language you intend to use is FORTRAN, then write function subprograms to extract various fields that may be packed into a word. You will also need subroutine subprograms to set certain fields. If your language permits use of structure variables, then these routines would not be needed. To begin with, write the basic list processing routines:

a) INIT(*N*) ... initialize the available space list to consist of *N* nodes. Depending on your design each node may be 2 or 3 words long.

b) GETNODE(*X*) ... provide node *X* for use.

c) RET(*X*) ... return node *X* to the available space list.

Test these routines to see if they work. Now write and test these routines for matrix operations:

d) MREAD(*A*) ... read matrix *A* and set up according to the representation of section 4.7. The input has the following format:

```
line 1:        n m r   n = # or rows

                       m = # or columns

                       r = # of nonzero terms
```

```
        line 2

                                triples of (row, columns, value) on each line

        line r + 1
```

These triples are in increasing order by rows. Within rows, the triples are in increasing order of columns. The data is to be read in one card at a time and converted to internal representation. The variable A is set to point to the head node of the circular list of head nodes (as in the text).

e) MWRITE(*A*) ... print out the terms of *A*. To do this, you will have to design a suitable output format. In any case, the output should be ordered by rows and within rows by columns.

f) MERASE(*A*) . . . return all nodes of the sparse matrix *A* to the available space list.

g) MADD(*A,B,C*) ... create the sparse matrix *C* = *A* + *B*. *A* and *B* are to be left unaltered.

h) MSUB(*A,B,C*) ... *C* = *A* - *B*

i) MMUL(*A,B,C*) ... create the sparse matrix *C* = *A* * *B*. *A* and *B* are to be left unaltered.

j) MTRP(*A,B*) ... create the sparse matrix $B = A^T$. *A* is to be left unaltered.

60. [Programming Project] Do the project of exercise 59 using the matrix representation of exercise 22.

61. (Landweber)

This problem is to simulate an airport landing and takeoff pattern. The airport has 3 runways, runway l, runway 2 and runway 3. There are 4 landing holding patterns, two for each of the first two runways. Arriving planes will enter one of the holding pattern queues, where the queues are to be as close in size as possible. When a plane enters a holding queue, it is assigned an integer ID number and an integer giving the number of time units the plane can remain in the queue before it must land (because of low fuel level). There is also a queue for takeoffs for each of the three runways. Planes arriving in a takeoff queue are also assigned an integer ID. The takeoff queues should be kept the same size.

At each time 0-3 planes may arrive at the landing queues and 0-3 planes may arrive at the takeoff queues. Each runway can handle one takeoff or landing at each time slot. Runway 3 is to be used for takeoffs except when a plane is low on fuel. At each time unit, planes in either landing queue whose air time has reached zero must be given priority over other landings and takeoffs. If only one plane is in this category, runway 3 is to be used. If more than one, then the other runways are also used (at each time at most 3 planes can be serviced in this way).

Use successive even (odd) integers for ID's of planes arriving at takeoff (landing) queues. At each time unit assume that arriving planes are entered into queues before takeoffs or landings occur. Try to design your algorithm so that neither landing nor takeoff queues grow excessively. However, arriving planes must be placed at the ends of queues. Queues cannot be reordered.

The output should clearly indicate what occurs at each time unit. Periodically print (a) the contents of each queue; (b) the average takeoff waiting time; (c) the average landing waiting time; (d) the average flying time remaining on landing; and (e) the number of planes landing with no fuel reserve. (b)-(c) are for planes that have taken off or landed respectively. The output should be self explanatory and easy to understand (and uncluttered).

The input can be on cards (terminal, file) or it can be generated by a random number generator. For each time unit the input is of the form:

CHAPTER 5: TREES

5.1 BASIC TERMINOLOGY

In this chapter we shall study a very important data object, trees. Intuitively, a tree structure means that the data is organized so that items of information are related by branches. One very common place where such a structure arises is in the investigation of genealogies. There are two types of genealogical charts which are used to present such data: the *pedigree* and the *lineal* chart. Figure 5.1 gives an example of each.

The pedigree chart shows someone's ancestors, in this case those of Dusty, whose two parents are Honey Bear and Brandy. Brandy's parents are Nuggett and Coyote, who are Dusty's grandparents on her father's side. The chart continues one more generation farther back to the great-grandparents. By the nature of things, we know that the pedigree chart is normally two-way branching, though this does not allow for inbreeding. When that occurs we no longer have a tree structure unless we insist that each occurrence of breeding is separately listed. Inbreeding may occur frequently when describing family histories of flowers or animals.

The lineal chart of figure 5.1(b), though it has nothing to do with people, is still a genealogy. It describes, in somewhat abbreviated form, the ancestry of the modern European languages. Thus, this is a chart of descendants rather than ancestors and each item can produce several others. Latin, for instance, is the forebear of Spanish, French, Italian and Rumanian. Proto Indo-European is a prehistoric language presumed to have existed in the fifth millenium B.C. This tree does not have the regular structure of the pedigree chart, but it is a tree structure nevertheless .

With these two examples as motivation let us define formally what we mean by a tree.

Figure 5.1 Two Types of Geneological Charts

Definition: A *tree* is a finite set of one or more nodes such that: (i) there is a specially designated node called the *root*; (ii) the remaining nodes are partitioned into $n \geq 0$ disjoint sets $T_1, ...,T_n$ where each of these sets is a tree. $T_1, ...,T_n$ are called the *subtrees* of the root.

Again we have an instance of a recursive definition (compare this with the definition of a generalized list in section 4.9). If we return to Figure 5.1 we see that the roots of the trees are Dusty and Proto Indo-European. Tree (a) has two subtrees whose roots are Honey Bear and Brandy while tree (b) has 3 subtrees with roots Italic, Hellenic, and Germanic. The condition that $T_1, ...,T_n$ be disjoint sets prohibits

subtrees from ever connecting together (no cross breeding). It follows that every item in a tree is the root of some subtree of the whole. For instance, West Germanic is the root of a subtree of Germanic which itself has three subtrees with roots: Low German, High German and Yiddish. Yiddish is a root of a tree with no subtrees.

There are many terms which are often used when referring to trees. A *node* stands for the item of information plus the branches to other items. Consider the tree in figure 5.2. This tree has 13 nodes, each item of data being a single letter for convenience. The root is *A* and we will normally draw trees with the root at the top. The number of subtrees of a node is called its *degree*. The degree of *A* is 3, of *C* is 1 and of *F* is zero. Nodes that have degree zero are called *leaf* or *terminal nodes*. {*K,L,F,G,M,I,J*} is the set of leaf nodes. Alternatively, the other nodes are referred to as *nonterminals*. The roots of the subtrees of a node, *X*, are the *children* of *X. X* is the *parent* of its children. Thus, the children of *D* are *H, I, J*; the parent of *D* is *A*. Children of the same parent are said to be *siblings*. *H, I* and *J* are siblings. We can extend this terminology if we need to so that we can ask for the grandparent of *M* which is *D,* etc. The *degree of a tree* is the maximum degree of the nodes in the tree. The tree of figure 5.2 has degree 3. The *ancestors* of a node are all the nodes along the path from the root to that node. The ancestors of *M* are *A, D* and *H*.

Figure 5.2 A Sample Tree

The *level* of a node is defined by initially letting the root be at level one. If a node is at level *l*, then its children are at level *l* + 1. Figure 5.2 shows the levels of all nodes in that tree. The *height or depth* of a tree is defined to be the maximum level of any node in the tree.
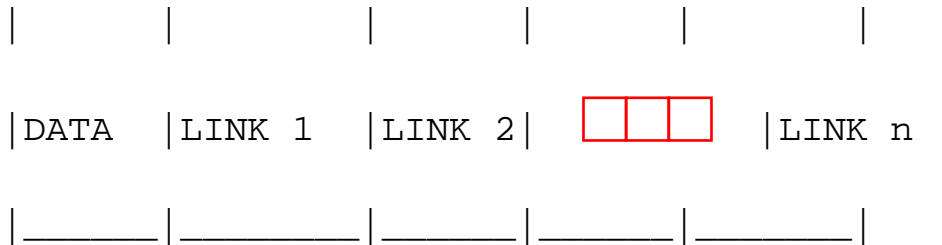
A *forest* is a set of $n \geq 0$ disjoint trees. The notion of a forest is very close to that of a tree because if we remove the root of a tree we get a forest. For example, in figure 5.2 if we remove *A* we get a forest with three trees.

There are other ways to draw a tree. One useful way is as a list. The tree of figure 5.2 could be written as the list

(*A*(*B*(*E*(*K,L*),*F*),*C*(*G*),*D*(*H*(*M*),*I,J*)))

The information in the root node comes first followed by a list of the subtrees of that node.

Now, how do we represent a tree in memory? If we wish to use linked lists, then a node must have a varying number of fields depending upon the number of branches.

———————————————————————————————

```
  |        |          |        |            |          |
  |DATA    |LINK 1    |LINK 2|   ▢▢      |LINK n |
  |_____|_____|_____|_____|_____|
```

However it is often simpler to write algorithms for a data representation where the node size is fixed. Using data and pointer fields we can represent a tree using the fixed node size list structure we devised in Chapter 4. The list representation for the tree of Figure 5.2 is on page 222. We can now make use of many of the general procedures that we originally wrote for handling lists. Thus, the data object tree is a special instance of the data object list and we can specialize the list representation scheme to them. In a later section we will see that another data object which can be used to represent a tree is the data object binary tree.

▢

List representation for the tree of figure 5.2

5.2 BINARY TREES

A binary tree is an important type of tree structure which occurs very often. It is characterized by the fact that any node can have at most two branches, i.e., there is no node with degree greater than two. For binary trees we distinguish between the subtree on the left and on the right, whereas for trees the order of the subtrees was irrelevant. Also a binary tree may have zero nodes. Thus a binary tree is really a different object than a tree.

Definition: A *binary tree* is a finite set of nodes which is either empty or consists of a root and two disjoint binary trees called the *left subtree* and the *right subtree*.

Using the notation introduced in chapter one we can define the data structure binary tree as follows:

structure *BTREE*

declare *CREATE( )* ▢ *btree*

ISMTBT(btree) ▢ *boolean*

MAKEBT(btree,item,btree) ▢ *btree*

LCHILD(*btree*) ☐ *btree*

DATA(*btree*) ☐ *item*

RCHILD(*btree*) ☐ *btree*

for all *p,r* ∈ *btree, d* ∈ *item* **let**

ISMTBT(*CREATE*) :: = **true**

ISMTBT(*MAKEBT*(*p,d,r*)) :: = **false**

LCHILD(*MAKEBT*(*p,d,r*)):: = *p*; *LCHILD*(*CREATE*):: = *error*

DATA(*MAKEBT*(*p,d,r*)) :: = *d*; *DATA*(*CREATE*) :: = *error*

RCHILD(*MAKEBT*(*p,d,r*)) :: = *r*; *RCHILD*(*CREATE*) :: = *error*

end

end *BTREE*

This set of axioms defines only a minimal set of operations on binary trees. Other operations can usually be built in terms of these. See exercise 35 for an example.

The distinctions between a binary tree and a tree should be analyzed. First of all there is no tree having zero nodes, but there is an empty binary tree. The two binary trees below

are different. The first one has an empty right subtree while the second has an empty left subtree. If the above are regarded as trees, then they are the same despite the fact that they are drawn slightly differently.

Figure 5.3 Two Sample Binary Trees

Figure 5.3 shows two sample binary trees. These two trees are special kinds of binary trees. The first is a *skewed* tree, skewed to the left and there is a corresponding one which skews to the right. Tree 5.3b is

called a *complete* binary tree. This kind of binary tree will be defined formally later on. Notice that all terminal nodes are on adjacent levels. The terms that we introduced for trees such as: degree, level, height, leaf, parent, and child all apply to binary trees in the natural way. Before examining data representations for binary trees, let us first make some relevant observations regarding such trees. First, what is the maximum number of nodes in a binary tree of depth *k*?

Lemma 5.1

(i) The maximum number of nodes on level *i* of a binary tree is 2^{i-1}, $i \geq 1$ and

(ii) The maximum number of nodes in a binary tree of depth *k* is $2^k - 1$, $k \geq 1$.

Proof:

(i) The proof is by induction on *i*.

Induction Base: The root is the only node on level *i* = 1. Hence the maximum number of nodes on level *i* = 1 is $2^{\square} = 2^{i-1}$.

Induction Hypothesis: For all *j*, $1 \leq j < i$, the maximum number of nodes on level *j* is 2^{j-1}.

Induction Step: The maximum number of nodes on level *i* - 1 is 2^{i-2}, by the induction hypothesis. Since each node in a binary tree has maximum degree 2, the maximum number of nodes on level *i* is 2 times the maximum number on level *i* - 1 or 2^{i-1}.

(ii) The maximum number of nodes in a binary tree of depth *k* is $\square$ (maximum number of nodes on level *i*)

$\square$

Next, let us examine the relationship between the number of terminal nodes and the number of nodes of degree 2 in a binary tree.

Lemma 5.2: For any nonempty binary tree, *T*, if n_o is the number of ter minal nodes and n_2 the number of nodes of degree 2, then $n_0 = n_2 + 1$.

Proof: Let n_1 be the number of nodes of degree 1 and *n* the total number of nodes. Since all nodes in *T* are of degree ≤ 2 we have:

$n = n_0 + n_1 + n_2$

(5.1)

If we count the number of branches in a binary tree, we see that every node except for the root has a branch leading into it. If B is the number of branches, then $n = B + 1$. All branches emanate either from a node of degree one or from a node of degree 2. Thus, $B = n_1 + 2n_2$. Hence, we obtain

$n = 1 + n_1 + 2n_2$

(5.2)

Subtracting (5.2) from (5.1) and rearranging terms we get

$n_0 = n_2 + 1$

In Figure 5.3(a) $n_0 = 1$ and $n_2 = 0$ while in Figure 5.3(b) $n_0 = 5$ and $n_2 = 4$.

As we continue our discussion of binary trees, we shall derive some other interesting properties.

5.3 BINARY TREE REPRESENTATIONS

A *full* binary tree of depth k is a binary tree of depth k having $2^k - 1$ nodes. By Lemma 5.1, this is the maximum number of nodes such a binary tree can have. Figure 5.4 shows a full binary tree of depth 4. A very elegant sequential representation for such binary trees results rom sequentially numbering the nodes, starting with nodes on level 1, then those on level 2 and so on. Nodes on any level are numbered from left to right (see figure 5.4). This numbering scheme gives us the definition of a complete binary tree. A binary tree with n nodes and of depth k is *complete* iff its nodes correspond to the nodes which re numbered one to n in the full binary tree of depth k. The nodes may now be stored in a one dimensional array, TREE, with the node umbered i being stored in TREE(i). Lemma 5.3 enables us to easily determine the locations of the parent, left child and right child of any node i in the binary tree.

Figure 5.4 Full Binary Tree of Depth 4 with Sequential Node Numbers

Lemma 5.3: If a complete binary tree with n nodes (i.e., depth= $[\log_2 n] + 1$) is represented sequentially as above then for any node with index i, $1 \le i \le n$ we have:
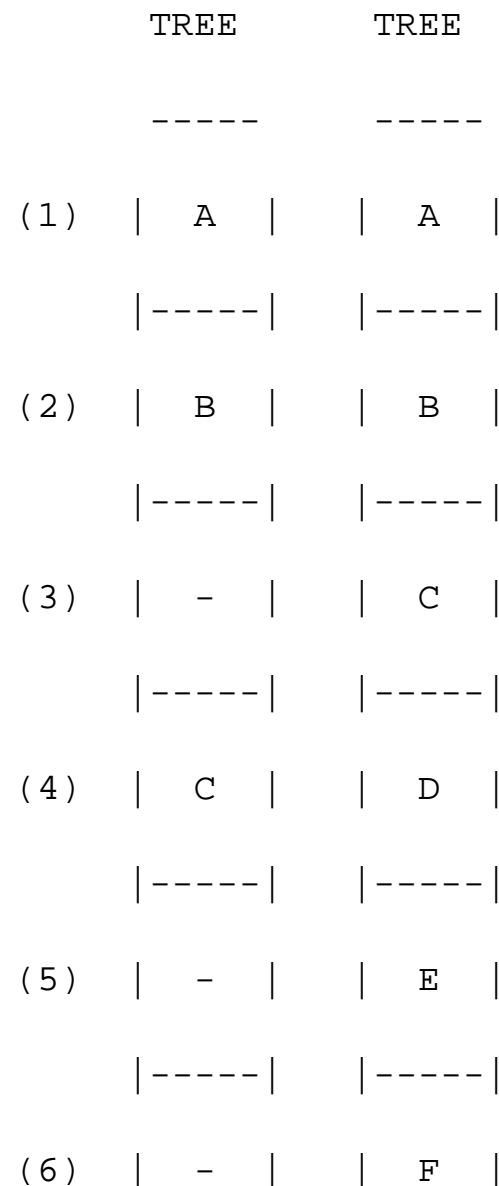
(i) PARENT(i) is at $[i/2]$ if $i \ne 1$. When $i = 1$, i is the root and has no parent.

(ii) LCHILD(i) is at $2i$ if $2i \leq n$. If $2i > n$, then i has no left child.

(iii) RCHILD(i) is at $2i + 1$ if $2i + 1 \leq n$. If $2i + 1 > n$, then i has no right child.

Proof: We prove (ii). (iii) is an immediate consequence of (ii) and the numberin g of nodes on the same level from left to right. (i) follows from (ii) and (iii). We prove (ii) by induction on i. For $i = 1$, clearly the left child is at 2 unless $2 > n$ in which case 1 has no left child. Now assume that for all j, $1 \leq j \leq i$, LCHILD(j) is at $2j$. Then, the two nodes immediately preceding LCHILD $(i + 1)$ in the representation are the right child of i and the left child of i. The left child of i is at $2i$. Hence, the left child of $i + 1$ is at $2i + 2 = 2(i + 1)$ unless $2(i + 1) > n$ in which case $i + 1$ has no left child.

This representation can clearly be used for all binary trees though in most cases there will be a lot of unutilized space. For complete binary trees the representation is ideal as no space is wasted. For the skewed tree of figure 5.3(a), however, less than half the array is utilized. In the worst case a skewed tree of depth k will require $2^k - 1$ spaces. Of these only k will be occupied.
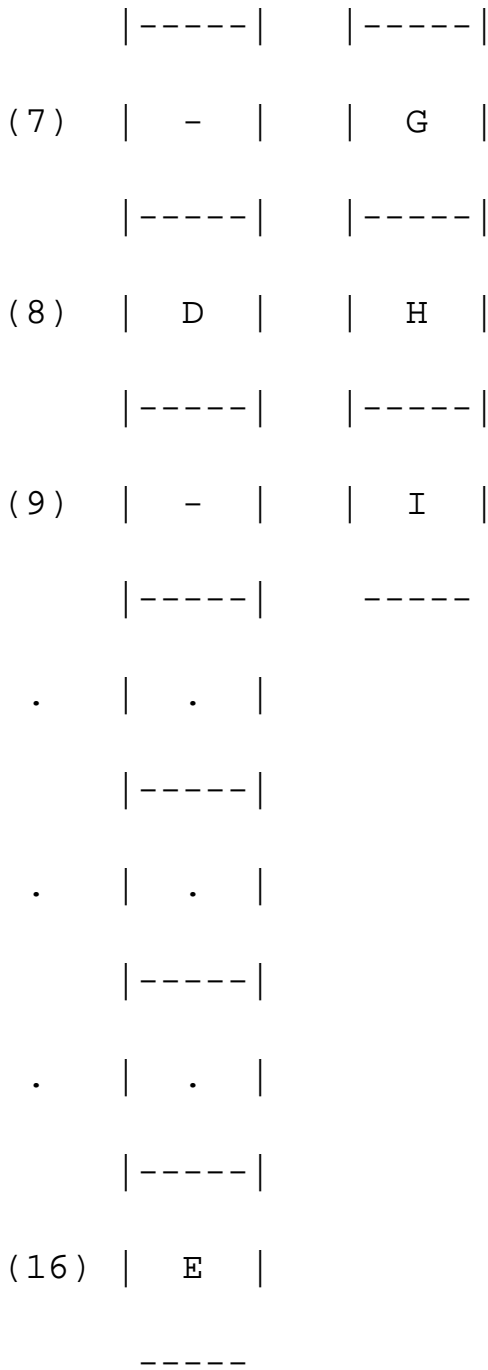
```
          TREE        TREE

          -----       -----

  (1)   |   A   |   |   A   |

        |-----|     |-----|

  (2)   |   B   |   |   B   |

        |-----|     |-----|

  (3)   |   -   |   |   C   |

        |-----|     |-----|

  (4)   |   C   |   |   D   |

        |-----|     |-----|

  (5)   |   -   |   |   E   |

        |-----|     |-----|

  (6)   |   -   |   |   F   |
```

```
        |-----|     |-----|

(7)    |  -  |     |  G  |

        |-----|     |-----|

(8)    |  D  |     |  H  |

        |-----|     |-----|

(9)    |  -  |     |  I  |

        |-----|      -----

  .    |  .  |

        |-----|

  .    |  .  |

        |-----|

  .    |  .  |

        |-----|

(16)   |  E  |

         -----
```

Figure 5.5 Array Representation of the Binary Trees of Figure 5.3

While the above representation appears to be good for complete binary trees it is wasteful for many other binary trees. In addition, the representation suffers from the general inadequacies of sequential representations. Insertion or deletion of nodes from the middle of a tree requires the movement of potentially many nodes to reflect the change in level number of these nodes. These problems can be easily overcome through the use of a linked representation. Each node will have three fields LCHILD, DATA and RCHILD as below:

While this node structure will make it difficult to determine the parent of a node, we shall see that for most applications, it is adequate. In case it is necessary to be able to determine the parent of random nodes. then a fourth field PARENT may be included. The representation of the binary trees of figure 5.3 using this node structure is given in figure 5.6.

Figure 5.6 Linked Representation for the Binary Trees of Figure 5.3.

5.4 BINARY TREE TRAVERSAL

There are many operations that we often want to perform on trees. One notion that arises frequently is the idea of traversing a tree or visiting each node in the tree exactly once. A full traversal produces a linear order for the information in a tree. This linear order may be familiar and useful. When traversing a binary tree we want to treat each node and its subtrees in the same fashion. If we let *L, D, R* stand for moving left, printing the data, and moving right when at a node then there are six possible combinations of traversal: *LDR, LRD, DLR, DRL, RDL,* and *RLD*. If we adopt the convention that we traverse left before right then only three traversals remain: *LDR, LRD* and *DLR*. To these we assign the names inorder, postorder and preorder because there is a natural correspondence between these traversals and producing the infix, postfix and prefix forms of an expression. Consider the binary tree of figure 5.7. This tree contains an arithmetic expression with binary operators: add(+), multiply(*), divide(/), exponentiation(**) and variables *A, B, C, D*, and *E*. We will not worry for now how this binary tree was formed, but assume that it is available. We will define three types of traversals and show the results for this tree.

Figure 5.7 Binary Tree with Arithmetic Expression

Inorder Traversal: informally this calls for moving down the tree towards the left until you can go no farther. Then you "visit" the node, move one node to the right and continue again. If you cannot move to the right, go back one more node. A precise way of describing this traversal is to write it as a recursive procedure.

```
procedure INORDER(T)

//T is a binary tree where each node has three fields L-

CHILD,DATA,RCHILD//
```

if $T \neq 0$ **then** [**call** *INORDER(LCHILD(T))*

print(*DATA(T))*

call(*INORDER(RCHILD(T))*]

end *INORDER*

Recursion is an elegant device for describing this traversal. Let us trace how INORDER works on the tree of figure 5.7.

```
Call of     value

INORDER   in root      Action

-------   -------   ----------

MAIN         +

 1           *

 2           /

 3           A

 4           0      print('A')

 4           0      print('/')

 3          **

 4           B

 5           0      print('B')

 5           0      print('**')

 4           C

 5           0      print('C')
```

```
5          0      print('*')

 2         D

 3         0      print('D')

 3         0      print('+')

 1         E

 2         0      print('E')

 2         0
```

The elements get printed in the order

*A/B ** C * D + E*

which is the *in*fix form of the expression.

A second form of traversal is *preorder*:

procedure *PREORDER (T)*

//T is a binary tree where each node has three fields L-

CHILD,DATA,RCHILD//

if *T* ≠ 0 **then** [**print** *(DATA(T))*

call *PREORDER(LCHILD(T))*

call *PREORDER(RCHILD(T))*]]

end *PREORDER*

In words we would say "visit a node, traverse left and continue again. When you cannot continue, move right and begin again or move back until you can move right and resume." The nodes of figure 5.7 would be printed in *pre*order as

+ * / A ** B C D E

which we recognize as the *pre*fix form of the expression.

At this point it should be easy to guess the next traversal method which is called *postorder*:

procedure *POSTORDER (T)*

//T is a binary tree where each node has three fields L-

CHILD,DATA,RCHILD//

if *T* ≠ 0 **then** [**call** *POSTORDER(LCHILD(T))*

call *POSTORDER(RCHILD(T))*

print *(DATA(T))]*

end *POSTORDER*

The output produced by POSTORDER is

*A B C ** / D * E +*

which is the postfix form of our expression.

Though we have written these three algorithms using recursion, it is very easy to produce an equivalent nonrecursive procedure. Let us take inorder as an example. To simulate the recursion we need a stack which will hold pairs of values (*pointer, returnad*) where *pointer* points to a node in the tree and *returnad* to the place where the algorithm should resume after an end is encountered. We replace every recursive call by a mechanism which places the new pair (*pointer, returnad*) onto the stack and goes to the beginning; and where there is a return or end we insert code which deletes the top pair from the stack if possible and either ends or branches to *returnad* (see section 4.10 for the exact details).

Though this procedure seems highly unstructured its virtue is that it is semiautomatically produced from the recursive version using a fixed set of rules. Our faith in the correctness of this program can be justified

procedure *INORDER*1*(T)*

//a nonrecursive version using a stack of size n//

```
i ← 0        //initialize the stack//

L1: if T ≠ 0 then [i ← i + 2; if i > n then call STACK_FULL

STACK (i - 1) ← T; STACK(i) ← 'L2'

T ← LCHILD(T); go to L1;      //traverse left

subtree//

L2: print (DATA(T))

i ← i + 2; if i > n then call STACK--FULL

STACK (i - 1) ← T; STACK(i) ← 'L3'

T ← RCHILD(T); go to L1]     //traverse right

subtree//

L3: if i ≠ 0 then [//stack not empty, simulate a return//

T ← STACK (i - 1); X ← STACK (i)

i ← i - 2; go to X]

end INORDER 1
```

if we first prove the correctness of the original version and then prove that the transformation rules result in a correct and equivalent program. Also we can simplify this program after we make some observations about its behavior. For every pair $(T_1,L3)$ in the stack when we come to label L3 this pair will be removed. All such consecutive pairs will be removed until either i gets set to zero or we reach a pair $(T_2,L2)$. Therefore, the presence of L_3 pairs is useful in no way and we can delete that part of the algorithm. This means we can eliminate the two lines of code following **print** (DATA(T)). We next observe that this leaves us with only one return address, L2, so we need not place that on the stack either. Our new version now looks like:

procedure *INORDER 2(T)*

```
//a simplified, nonrecursive version using a stack of size n//

i ← 0          //initialize stack//

L1: if T ≠ 0 then [i ← i + 1; if i > n then call STACK__FULL

STACK (i) ← T; T ← LCHILD(T); go to L1

L2: print (DATA(T))

T ← RCHILD(T); go to L1]

if i ≠ 0 then [//stack not empty//

T ← STACK (i): i ← i - 1; go to L2]

end INORDER2
```

This program is considerably simpler than the previous version, but it may still offend some people because of the seemingly undisciplined use of **go to**'s. A SPARKS version without this statement would be:

```
procedure INORDER3(T)

//a nonrecursive, no go to version using a stack of size n//

i ← 0 //initialize stack//

loop

while T ≠ 0 do          //move down LCHILD fields//

i ← i + 1; if i > n then call STACK--FULL

STACK (i) ← T; T ← LCHILD(T)

end
```

```
if i = 0 then return

T ☐ STACK (i); i ☐ i - 1

print(DATA(T)); T ☐ RCHILD(T)

forever

end INORDER3
```

In going from the recursive version to the iterative version INORDER3 one undesirable side effect has crept in. While in the recursive version *T* was left pointing to the root of the tree, this is not the case when INORDER3 terminates. This may readily be corrected by introducing another variable *P* which is used in place of *T* during the traversal.

What are the computing time and storage requirements of INORDER3 ? Let *n* be the number of nodes in *T*. If we consider the action of the above algorithm, we note that every node of the tree is placed on the stack once. Thus, the statements STACK(*i*) ☐ *T* and *T* ☐ STACK(*i*) are executed *n* times. Moreover, *T* will equal zero once for every zero link in the tree which is exactly

$$2n_0 + n_1 = n_0 + n_1 + n_2 + 1 = n + 1.$$

So every step will be executed no more than some small constant times *n* or *O(n)*. With some further modifications we can lower the constant (see exercises). The space required for the stack is equal to the depth of *T*. This is at most *n*.

Before we leave the topic of tree traversal, we shall consider one final question. Is it possible to traverse binary trees without the use of extra space for a stack? One simple solution is to add a PARENT field to each node. Then we can trace our way back up to any root and down again. Another solution which requires two bits per node is given in section 5.6. If the allocation of this extra space is too costly then we can use the method of algorithm MARK2 of section 4.10. No extra storage is required since during processing the LCHILD and RCHILD fields are used to maintain the paths back to the root. The stack of addresses is stored in the leaf nodes. The exercises examine this algorithm more closely.

5.5 MORE ON BINARY TREES

Using the definition of a binary tree and the recursive version of the traversals, we can easily write other routines for working with binary trees. For instance, if we want to produce an exact copy of a given binary tree we can modify the postorder traversal algorithm only slightly to get:

procedure *COPY(T)*

//for a binary tree *T, COPY* returns a pointer to an exact copy of

T; new nodes are gotten using the usual mechanism//

Q ▢ 0

if *T* ≠ 0 **then** [*R* ▢ *COPY(LCHILD(T))*   //copy left subtree//

S ▢ *COPY(RCHILD(T))*   //copy right subtree//

call *GETNODE(Q)*

LCHILD(Q) ▢ *R; RCHILD(Q)* ▢ *S*

//store in fields of *Q*//

DATA(Q) ▢ *DATA(T)*]

return(*Q*)   //copy is a function//

end *COPY*

Another problem that is especially easy to solve using recursion is determining the equivalence of two binary trees. Binary trees are equivalent if they have the same topology and the information in corresponding nodes is identical. By the same topology we mean that every branch in one tree corresponds to a branch in the second in the same order. Algorithm EQUAL traverses the binary trees in preorder, though any order could be used.

We have seen that binary trees arise naturally with genealogical information and further that there is a natural relationship between the tree traversals and various forms of expressions. There are many other

procedure *EQUAL(S,T)*

//This procedure has value **false** if the binary trees *S* and *T* are not

equivalent. Otherwise, its value is **true**//

```
ans ☐ false

case

: S = 0 and T = 0: ans ☐ true

: S ≠ 0 and T ≠ 0:

if DATA(S) = DATA(T)

then [ans ☐ EQUAL(LCHILD(S),LCHILD(T))

if ans then ans ☐ EQUAL(RCHILD(S),RCHILD(T))]

end

return (ans)

end EQUAL
```

instances when binary trees are important, and we will look briefly at two of them now. The first problem has to do with processing a list of alphabetic data, say a list of variable names such as

X1, I, J, Z, FST, X2, K.

We will grow a binary tree as we process these names in such a way that for each new name we do the following: compare it to the root and if the new name alphabetically precedes the name at the root then move left or else move right; continue making comparisons until we fall off an end of the tree; then create a new node and attach it to the tree in that position. The sequence of binary trees obtained for the above data is given in figure 5.8. Given the tree in figure 5.8(g) consider the order of the identifiers if they were printed out using an inorder traversal

FST, I, J, K, X1, X2, Z

So by growing the tree in this way we turn inorder into a sorting method. In Chapter 9 we shall prove that this method works in general.

As a second example of the usefulness of binary trees, consider the set of formulas one can construct by taking variables $x_1, x_2, x_3, ...$ and the operators ☐ (and), ☐ (or) and ☐ (not). These variables can only hold one of two possible values, true or false. The set of expressions which can be formed using these

variables and operators is defined by the rules: (i) a variable is an expression, (ii) if *x,y* are expressions then $x \square$ y, x $\square$ y, $\neg$ x are expressions. Parentheses can be used to alter the normal order of evaluation which is **not** before **and** before **or**. This comprises the formulas in the *propositional calculus* (other operations such as implicaion can be expressed using $\square$, $\square$, $\neg$). The expression

$\square$

Figure 5.8 Tree of Identifiers

$x_1 \square (x_2 \square \neg x_3)$

is a formula (read "x_1 or x_2 and not X_3"). If x_1 and X_3 are false and x_2 is true, then the value of this expression is

false $\square$ (true $\square$ $\neg$ false)

= false $\square$ true

= true

The *satisfiability problem* for formulas of the propositional calculus asks if there is an assignment of values to the variables which causes the value of the expression to be true. This problem is of great historical interest in computer science. It was originally used by Newell, Shaw and Simon in the late 1950's to show the viability of heuristic programming (the Logic Theorist).

Again, let us assume that our formula is already in a binary tree, say

$(x_1 \square \neg x_2) \square (\neg x_1 \square x_3) \square \neg x_3$
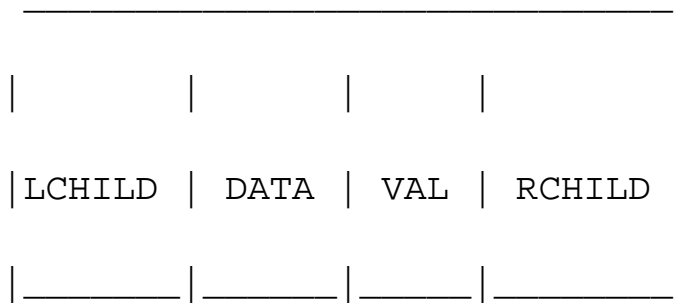
in the tree

$\square$

Figure 5.9 Propositional Formula in a Binary Tree

The inorder of this tree is $x_1 \square \neg x_2 \square \neg x_1 \square x_3 \square \neg x_3$, the infix form of the expression. The most obvious algorithm to determine satisfiability is to let (x_1,x_2,x_3) take on all possible combinations of truth and falsity and to check the formula for each combination. For *n* variables there are 2^n possible combinations of true $= t$ and false $= f$, e.g. for $n = 3$ (*t,t,t*), (*t,t,f*), (*t,f, t*), (*t,f,f*), (*f,t,t*), (*f,t,f*), (*f,f,t*), (*f,f,f*).

The algorithm will take at least $O(g2^n)$ or exponential time where g is the time to substitute values for x_1, x_2, x_3 and evaluate the expression.

To evaluate an expression one method would traverse the tree in postorder, evaluating subtrees until the entire expression is reduced to a single value. This corresponds to the postfix evaluation of an arithmetic expression that we saw in section 3.3. Viewing this from the perspective of the tree representation, for every node we reach, the values of its arguments (or children) have already been computed. So when we reach the v node on level two, the values of $x_1 \square \neg x_2$ and $\neg x_1 \square x_3$ will already be available to us and we can apply the rule for *or*. Notice that a node containing $\neg$ has only a single right branch since *not* is a unary operator.

For the purposes of this algorithm we assume each node has four fields:

```
 _____
|         |        |       |            |
|         |        |       |            |
|LCHILD   | DATA   | VAL   | RCHILD     |
|         |        |       |            |
|_____|_____|_____|_____|
```

where LCHILD, DATA, RCHILD are as before and VAL is large enough to hold the symbols true or false. Also we assume that DATA(*T*) instead of containing the variable 'x_i', is a pointer to a table DEF which has one entry for each variable. Then DEF(DATA(*T*)) will contain the current value of the variable. With these preparations and assuming an expression with *n* variables pointed at by *T* we can now write a first pass at our algorithm for satisfiability:

```
for all 2ⁿ possible combinations do

generate the next combination;

store it in DEF(1) to DEF(n);

call POSTORDER and evaluate T;

if VAL(T) = true then [print DEF(1) to DEF(n)

stop]

end
```

```
print ('no satisfiable combination')
```

Now let us concentrate on this modified version of postorder. Changing the original recursive version seems the simplest thing to do.

```
procedure POSTORDER_EVAL(T)

//a binary tree T containing a propositional formula is evaluated with

the result stored in the VAL field of each node//

if T ≠ 0 then[ call POSTORDER_EVAL(LCHILD(T))

call POSTORDER_EVAL(RCHILD(T))

case

:DATA(T) = '¬ : VAL(T) ☐ not VAL(RCHILD(T))

:DATA(T) = '☐ : VAL(T) ☐ VAL(LCHILD(T) or

VAL(RCHILD(T))

:DATA(T) = '☐ : VAL(T) ☐ VAL(LCHILD(T) and

VAL(RCHILD(T))

:else:       VAL(T) ☐ DEF(DATA(T))

end]

end POSTORDER_EVAL
```

5.6 THREADED BINARY TREES

If we look carefully at the linked representation of any binary tree, we notice that there are more null links than actual pointers. As we saw before, there are $n + 1$ null links and $2n$ total links. A clever way to make use of these null links has been devised by A. J. Perlis and C. Thornton. Their idea is to replace the

null links by pointers, called threads, to other nodes in the tree. If the RCHILD(*P*) is normally equal to zero, we will replace it *by a pointer to the node which would be printed after P when traversing the tree in inorder*. A null LCHILD link at node *P* is replaced *by a pointer to the node which immediately precedes node P in inorder*. Figure 5.10 shows the binary tree of figure 5.3(b) with its new threads drawn in as dotted lines.

The tree *T* has 9 nodes and 10 null links which have been replaced by threads. If we traverse *T* in inorder the nodes will be visited in the order *H D I B E A F C G*. For example node *E* has a predecessor thread which points to B and a successor thread which points to *A*.

In the memory representation we must be able to distinguish between threads and normal pointers. This is done by adding two extra one bit fields LBIT and RBIT.

```
LBIT(P) =1        if LCHILD(P) is a normal pointer

LBIT(P) = 0       if LCHILD(P) is a thread

RBIT(P) = 1       if RCHILD(P) is a normal pointer

RBIT(P) = 0       if RCHILD(P) is a thread
```

Figure 5.10 Threaded Tree Corresponding to Figure 5.3(b)

In figure 5.10 we see that two threads have been left dangling in LCHILD(*H*) and RCHILD(*G*). In order that we leave no loose threads we will assume a head node for all threaded binary trees. Then the complete memory representation for the tree of figure 5.10 is shown in figure 5.11. The tree *T* is the left subtree of the head node. We assume that an empty binary tree is represented by its head node as

This assumption will permit easy algorithm design. Now that we have made use of the old null links we will see that the algorithm for inorder traversal is simplified. First, we observe that for any node *X* in a binary tree, if RBIT(*X*) = 0, then the inorder successor of *X* is RCHILD(*X*) by definition of threads. If RBIT(*X*) = 1, then the inorder successor of *X* is obtained by following a path of left child links from the right child of *X* until a node with LBIT = 0 is reached. The algorithm INSUC finds the inorder successor of any node *X* in a threaded binary tree.

Figure 5.11 Memory Representation of Threaded Tree

```
procedure INSUC(X)

//find the inorder succcesor of X in a threaded binary tree//

S □ RCHILD(X)           //if RBIT(X) = 0 we are done//

if RBIT(X) = 1 then [while LBIT(S) = 1 do       //follow left//

S □ LCHILD(S)          //until a thread//

end]

return (S)

end INSUC
```

The interesting thing to note about procedure INSUC is that it is now possible to find the inorder successor of an arbitrary node in a threaded binary tree without using any information regarding inorder predecessors and also without using an additional stack. If we wish to list in inorder all the nodes in a threaded binary tree, then we can make repeated calls to the procedure INSUC. Since vhe tree is the left subtree of the head node and because of the choice of RBIT= 1 for the head node, the inorder sequence of nodes for tree *T* is obtained by the procedure TINORDER.

```
procedure TINORDER (T)

//traverse the threaded binary tree, T, in inorder//

HEAD □ T

loop

T □ INSUC(T)

if T = HEAD then return

print(DATA(T))

forever
```

end *TINORDER*

The computing time is still $O(n)$ for a binary tree with n nodes. The constant here will be somewhat smaller than for procedure INORDER3.

We have seen how to use the threads of a threaded binary tree for inorder traversal. These threads also simplify the algorithms for preorder and postorder traversal. Before closing this section let us see how to make insertions into a threaded tree. This will give us a procedure for growing threaded trees. We shall study only the case of inserting a node *T* as the right child of a node *S*. The case of insertion of a left child is given as an exercise. If *S* has an empty right subtree, then the insertion is simple and diagrammed in figure 5.12(a). If the right subtree of *S* is non-empty, then this right subtree is made the right subtree of *T* after insertion. When this is done, *T* becomes the inorder predecessor of a node which has a LBIT = 0 and consequently there is a thread which has to be updated to point to *T*. The node containing this thread was previously the inorder successor of *S*. Figure 5.12(b) illustrates the insertion for this case. In both cases *S* is the inorder predecessor of *T*. The details are spelled out in algorithm INSERT_RIGHT.

□

Figure 5.12 Insertion of T as a Right Child of S in a Threaded Binary Tree

procedure *INSERT_RIGHT (S,T)*

//insert node *T* as the right child of *S* in a threaded binary tree//

RCHILD(T) □ *RCHILD(S); RBIT(T)* □ *RBIT(S)*

LCHILD(T) □ *S; LBIT(T)* □ 0   //LCHILD(T) is a thread//

RCHILD(S) □ *T; RBIT(S)* □ 1   //attach node *T* to *S*//

if *RBIT (T)* = 1 **then** [*W* □ *INSUC(T)*//*S* had a right child//

LCHILD(W) □ *T*]

end *INSERT_RIGHT*

5.7 BINARY TREE REPRESENTATION OF TREES

We have seen several representations for and uses of binary trees. In this section we will see that every tree can be represented as a binary tree. This is important because the methods for representing a tree as suggested in section 5.1 had some undesirable features. One form of representation used variable size nodes. While the handling of nodes of variable size is not impossible, in section 4.8 we saw that it was considerably more difficult than the handling of fixed size nodes (section 4.3). An alternative would be to use fixed size nodes each node having k child fields if k is the maximum degree of any node. As Lemma 5.4 shows, this would be very wasteful in space.

Lemma 5.4: If T is a k-ary tree (i.e., a tree of degree k) with n nodes, each having a fixed size as in figure 5.13, then $n(k-1) + 1$ of the nk link fields are zero, $n \geq 1$.

Proof: Since each nonzero link points to a node and exactly one link points to each node other than the root, the number of nonzero links in an n node tree is exactly $n - 1$. The total number of link fields in a k-ary tree with n nodes is nk. Hence, the number of null links is $nk - (n - 1) = n(k - 1) + 1$. •

Lemma 5.4 implies that for a 3-ary tree more than 2/3 of the link fields are zero! The proportion of zero links approaches 1 as the degree of the tree increases. The importance of using binary trees to represent trees is that for binary trees only about 1/2 of the link fields are zero.


Figure 5.13 Possible Node Structure for a k-ary Tree

In arriving at the binary tree representation of a tree we shall implicitly make use of the fact that the order of the children of a node is not important. Suppose we have the tree of figure 5.14.

Figure 5.14 A Sample Tree

Then, we observe that the reason we needed nodes with many link fields is that the prior representation was based on the parent-child relationship and a node can have any number of children. To obtain a binary tree representation, we need a relationship, between the nodes, that can be characterized by at most two quantities. One such relationship is the leftmost-child-next-right-sibling relationship. Every node has at most one leftmost child and at most one next right sibling. In the tree of figure 5.14, the leftmost child of B is E and the next right sibling of B is C. Strictly speaking, since the order of children in a tree is not important, any of the children of a node could be its leftmost child and any of its siblings could be its next right sibling. For the sake of definiteness, we choose the nodes based upon how the tree is drawn. The binary tree corresponding to the tree of figure 5.14 is thus obtained by connecting together all siblings of a node and deleting all links from a node to its children except for the link to its leftmost child. The node structure corresponds to that of

```
  _____

 |          DATA          |

 |------------------------|

 |  CHILD   |  SIBLING  |

 |_____|_____|
```
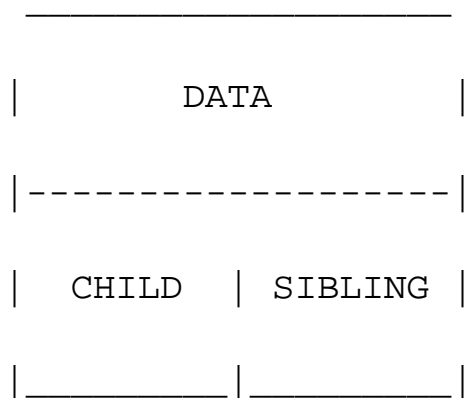
Using the transformation described above, we obtain the following representation for the tree of figure 5.14.

This does not look like a binary tree, but if we tilt it roughly 45 clockwise we get

Figure 5.15 Associated Binary Tree for Tree of Figure 5.14

Let us try this transformation on some simple trees just to make sure we've got it.

One thing to notice is that the RCHILD of the root node of every resulting binary tree will be empty. This is because the root of the tree we are transforming has no siblings. On the other hand, if we have a forest then these can all be transformed into a single binary tree by first obtaining the binary tree representation of each of the trees in the forest and then linking all the binary trees together through the SIBLING field of the root nodes. For instance, the forest with three trees

yields the binary tree

We can define this transformation in a formal way as follows:

If $T_1, ..., T_n$ is a forest of trees, then the binary tree corresponding to this forest, denoted by $B(T_1, ..., T_n)$:

(i) is empty if $n = 0$;

(ii) has root equal to root (T_1); has left subtree equal to $B(T_{11}, T_{12}, ..., T_{1_m})$ where $T_{11}, ..., T_{1_m}$ are the subtrees of root(T_1); and has right subtree $B(T_2, ..., T_n)$.

Preorder and inorder traversals of the corresponding binary tree T of a forest F have a natural correspondence with traversals on F. Preorder traversal of T is equivalent to visiting the nodes of F in *tree preorder* which is defined by:

(i) if F is empty then return;

(ii) visit the root of the first tree of F;

(iii) traverse the subtrees of the first tree in tree preorder;

(iv) traverse the remaining trees of F in tree preorder.

Inorder traversal of T is equivalent to visiting the nodes of F in *tree inorder* as defined by:

(i) if F is empty then return;

(ii) traverse the subtrees of the first tree in tree inorder;

(iii) visit the root of the first tree;

(iv) traverse the remaining trees in tree inorder.

The above definitions for forest traversal will be referred to as preorder and inorder. The proofs that preorder and inorder on the corresponding binary tree are the same as preorder and inorder on the forest are left as exercises. There is no natural analog for postorder traversal of the corresponding binary tree of a forest. Nevertheless, we can define the *postorder traversal of a forest* as:

(i) if F is empty then return;

(ii) traverse the subtrees of the first tree of F in tree postorder;

(iii) traverse the remaining trees of F in tree postorder;

(iv) visit the root of the first tree of F.

This traversal is used later on in section 5.8.3 for describing the minimax procedure.

5.8 APPLICATIONS OF TREES

5.8.1 Set Representation

In this section we study the use of trees in the representation of sets. We shall assume that the elements of the sets are the numbers 1, 2, 3, ..., n. These numbers might, in practice, be indices into a symbol table where the actual names of the elements are stored. We shall assume that the sets being represented are pairwise disjoint; i.e., if S_i and S_j, $i \neq j$, are two sets then there is no element which is in both S_i and S_j. For example, if we have 10 elements numbered 1 through 10, they may be partitioned into three disjoint sets $S_1 = \{1, 7, 8, 9\}$; $S_2 = \{2, 5, 10\}$ and $S_3 = \{3, 4, 6\}$. The operations we wish to perform on these sets are:

(i) Disjoint set union ... if S_i and S_j are two disjoint sets, then their union $S_i \cup S_j = \{$all elements x such that x is in S_i or $S_j\}$. Thus, $S_1 \cup S_2 = \{1, 7, 8, 9, 2, 5, 10\}$. Since we have assumed that all sets are disjoint, following the union of S_i and S_j we can assume that the sets S_i and S_j no longer exist independently, i.e., they are replaced by $S_i \cup S_j$ in the collection of sets.

(ii) Find(i) ... find the set containing element i. Thus, 4 is in set S_3 and 9 is in set S_1.

The sets will be represented by trees. One possible representation for the sets S_1, S_2 and S_3 is:

Note that the nodes are linked on the parent relationship, i.e. each node other than the root is linked to its parent. The advantage of this will become apparent when we present the UNION and FIND algorithms. First, to take the union of S_1 and S_2 we simply make one of the trees a subtree of the other. $S_1 \cup S_2$ could then have one of the following representations

In order to find the union of two sets, all that has to be done is to set the parent field of one of the roots to the other root. This can be accomplished easily if, with each set name, we keep a pointer to the root of the tree representing that set. If, in addition, each root has a pointer to the set name, then to determine which set an element is currently in, we follow parent links to the root of its tree and use the pointer to the set name. The data representation for S_1, S_2 and S_3 may then take the form:

In presenting the UNION and FIND algorithms we shall ignore the actual set names and just identify sets by the roots of the trees representing them. This will simplify the discussion. The transition to set names is easy. If we determine that element i is in a tree with root j and j has a pointer to entry k in the set name table, then the set name is just NAME(k). If we wish to union sets S_i and S_j, then we wish to union the trees with roots POINTER(S_i) and POINTER(S_j). As we shall see, in many applications the set name is just the element at the root. The operation of FIND(i) now becomes: determine the root of the tree containing element i. UNION(i,j) requires two trees with roots i and j to be joined. We shall assume that the nodes in the trees are numbered 1 through n so that the node index corresponds to the element index. Thus, element 6 is represented by the node with index 6. Consequently, each node needs only one field: the PARENT field to link to its parent. Root nodes have a PARENT field of zero. Based on the above discussion, our first attempt at arriving at UNION, FIND algorithms would result in the algorithms U and F below.

procedure *U(i,j)*

//replace the disjoint sets with roots *i* and *j*, *i* $\neq$ *j* with their union//

PARENT(i) $\Box$ *j*

end *U*

procedure *F(i)*

//find the root *j* of the tree containing element *i*//

j $\Box$ *i*

while *PARENT(j)* > 0 **do**   //*PARENT (j)* = 0 if this node is a

root//

j $\Box$ *PARENT(j)*

end

return(*j*)

end *F*

While these two algorithms are very easy to state, their performance characteristics are not very good. For instance, if we start off with p elements each in a set of its own, i.e., $S_i = \{i\}$, $1 \leq i \leq p$, then the initial configuration consists of a forest with p nodes and $PARENT(i) = 0$, $1 \leq i \leq p$. Now let us process the following sequence of UNION-FIND operations.

$U(1,2)$, $F(1)$, $U(2,3)$, $F(1)$, $U(3,4)$

$F(1)$, $U(4,5)$, ...,$F(1)$, $U(n - 1,n)$

This sequence results in the degenerate tree:

Since the time taken for a union is constant, all the $n - 1$ unions can be processed in time $O(n)$. However, each FIND requires following a chain of PARENT links from one to the root. The time required to process a FIND for an element at level i of a tree is O(i). Hence, the total time needed to process the $n - 2$ finds is . It is easy to see that this example represents the worst case behavior of the UNION-FIND algorithms. We can do much better if care is taken to avoid the creation of degenerate trees. In order to accomplish this we shall make use of a *Weighting Rule* for UNION (i,j). *If the number of nodes in tree i is less than the number in tree j, then make j the parent of i, otherwise make i the parent of j.* Using this rule on the sequence of set unions given before we obtain the trees on page 252. Remember that the arguments of UNION must both be roots. The time required to process all the n finds is only $O(n)$ since in this case the maximum level of any node is 2. This, however, is not the worst case. In Lemma 5.5 we show that using the weighting rule, the maximum level for any node is $\lfloor \log n \rfloor + 1$. First, let us see how easy it is to implement the weighting rule. We need to know how many nodes there are in any tree. To do this easily, we maintain a count field in the root of every tree. If i is a root node, then COUNT $(i) =$ number of nodes in that tree. The count can be maintained in the PARENT field as a negative number. This is equivalent to using a one bit field to distinguish a count from a pointer. No confusion is created as for all other nodes the PARENT is positive.

Trees obtained using the weighting rule

```
procedure UNION(i,j)

//union sets with roots i and j, i ≠ j, using the weighting rule.
PARENT

(i) = - COUNT(i) and PARENT(j) = - COUNT(j)//
```

```
x ☐ PARENT (i) + PARENT(j)

if PARENT(i) > PARENT(j)

then [PARENT (i) ☐ j    //i has fewer nodes//

PARENT (j) ☐ x]

else [PARENT (j) ☐ i    //j has fewer nodes//

PARENT (i) ☐ x]

end UNION
```

The time required to perform a union has increased somewhat but is still bounded by a constant, i.e. it is $O(1)$. The FIND algorithm remains unchanged. The maximum time to perform a find is determined by Lemma 5.5.

Lemma 5.5: Let T be a tree with n nodes created as a result of algorithm UNION. No node in T has level greater $\lfloor \log_2 n \rfloor + 1$.

Proof: The lemma is clearly true for $n = 1$. Assume it is true for all trees with i nodes, $i \leq n - 1$. We shall show that it is also true for $i = n$. Let T be a tree with n nodes created by the UNION algorithm. Consider the last union operation performed, UNION(k,j). Let m be the number of nodes in tree j and $n - m$ the number in k. Without loss of generality we may assume $1 \leq m \leq n/2$. Then the maximum level of any node in T is either the same as that in k or is one more than that in j. If the former is the case, then the maximum level in T is $\leq \lfloor \log_2 (n - m) \rfloor + 1 \leq \lfloor \log_2 n \rfloor + 1$. If the latter is the case then the maximum level in T is $\leq \lfloor \log_2 m \rfloor + 2 \leq \lfloor \log_2 n/2 \rfloor + 2. \leq \lfloor \log_2 n \rfloor + 1$.

Example 5.1 shows that the bound of Lemma 5.5 is achievable for some sequence of unions.

Example 5.1: Consider the behavior of algorithm UNION on the following sequence of unions starting from the initial configuration

PARENT(i) = -COUNT(i) = -1, $1 \leq i \leq n = 2^3$

UNION(1,2), UNION(3,4), UNION(5,6), UNION(7,8),

UNION(1,3), UNION(5,7), UNION(1,5).

The following trees are obtained:

☐

☐

As is evident, the maximum level in any tree is $\lfloor \log_2 m \rfloor + 1$ if the tree has m nodes.

As a result of Lemma 5.5, the maximum time to process a find is at most $O(\log n)$ if there are n elements in a tree. If an intermixed sequence of $n - 1$ UNION and m FIND operations is to be processed, then the worst case time becomes $O(n + m \log n)$. Surprisingly, further improvement is possible. This time the modification will be made in the FIND algorithm using the *Collapsing Rule: If j is a node on the path from i to its root and PARENT(j) $\neq$ root(i) then set PARENT(j) ☐ root(i)*. The new algorithm then becomes:

This modification roughly doubles the time for an individual find. However, it reduces the worst case time over a sequence of finds.

Example 5.2: : Consider the tree created by algorithm UNION on the sequence of unions of example 5.1. Now process the following 8 finds:

FIND(8), FIND(8), ... FIND(8)

procedure *FIND(i)*

//find the root of the tree containing element *i*. Use the collapsing

rule to collapse all nodes from *i* to the root *j*//

j ☐ i

while *PARENT(j)* > 0 **do**   //find root//

j ☐ *PARENT(j)*

end

k ☐ i

```
while k ≠ j do      //collapse nodes from i to root j//

t    PARENT(k)

PARENT(k)    j

k    t

end

return(j)

end FIND
```

Using the old version F of algorithm FIND, FIND(8) requires going up 3 parent link fields for a total of 24 moves to process all 8 finds. In algorithm FIND, the first FIND(8) requires going up 3 links and then resetting 3 links. Each of the remaining 7 finds requires going up only 1 link field. The total cost is now only 13 moves.

The worst case behavior of the UNION-FIND algorithms while processing a sequence of unions and finds is stated in Lemma 5.6. Before stating this lemma, let us introduce a very slowly growing function $\square(m,n)$ which is related to a functional inverse of Ackermann's function $A(p,q)$. We have the following definition for $\square(m,n)$:

$$\square(m,n) = \min\{z \geq 1 \mid A(z, 4\lceil m/n \rceil) > \log_2 n\}$$

The definition of Ackermann's function used here is:

The function $A(p,q)$ is a very rapidly growing function. One may prove the following three facts:

If we assume $m \neq 0$ then (b) and (c) together with the definition of $\square(m,n)$ imply that $\square(m,n) \leq 3$ for $\log_2 n < A(3,4)$. But from (a), $A(3,4)$ is a very large number indeed! In Lemma 5.6 n will be the number of UNIONs performed. For all practical purposes we may assume $\log_2 n < A(3,4)$ and hence $\square(m,n) \leq 3$.

Lemma 5.6: [Tarjan] Let $T(m,n)$ be the maximum time required to process any intermixed sequence of

$m \geq n$ FINDs and n - 1 UNIONs. Then $k_1 m \square (m,n) \leq T(m,n) \leq k_2 m \square (m,n)$ for some positive constants k_1 and k_2.

Even though the function $\square (m,n)$ is a very slowly growing function, the complexity of UNION-FIND is not linear in m, the number of FINDs. As far as the space requirements are concerned, the space needed is one node for each element.

Let us look at an application of algorithms UNION and FIND to processing the equivalence pairs of section 4.6. The equivalence classes to be generated may be regarded as sets. These sets are disjoint as no variable can be in two equivalence classes. To begin with all n variables are in an equivalence class of their own; thus $PARENT(i) = -1, 1 \leq i \leq n$. If an equivalence pair, $i \square j$, is to be processed, we must first determine the sets containing i and j. If these are different, then the two sets are to be replaced by their union. If the two sets are the same, then nothing is to be done as the relation $i \square j$ is redundant; i and j are already in the same equivalence class. To process each equivalence pair we need to perform at most two finds and one union. Thus, if we have n variables and $m \geq n$ equivalence pairs, the total processing time is at most $O(m \square (2m,m))$. While for very large n this is slightly worse than the algorithm of section 4.6, it has the advantage of needing less space and also of being "on line."

In Chapter 6 we shall see another application of the UNION-FIND algorithms.

Example 5.3: We shall use the UNION-FIND algorithms to process the set of equivalence pairs of scction 4.6. Initially, there are 12 trees, one for each variable. $PARENT(i) = -1, 1 \leq i \leq 12$.

$\square$

Each tree represents an equivalence class. It is possible to determine if two elements are currently in the same equivalence class at each stage of the processing by simply making two finds.

5.8.2 Decision Trees

Another very useful application of trees is in decision making. Consider the well-known *eight coins* problem. Given coins *a,b,c,d,e,f,g,h,* we are told that one is a counterfeit and has a different weight than the others. We want to determine which coin it is, making use of an equal arm balance. We want to do so using a minimum number of comparisons and at the same time determine whether the false coin is heavier or lighter than the rest. The tree below represents a set of decisions by which we can get the answer to our problem. This is why it is called a decision tree. The use of capital *H* or *L* means that the counterfeit coin is *h*eavier or *l*ighter. Let us trace through one possible sequence. If $a + b + c < d + e + f$, then we know that the false coin is present among the six and is neither *g* nor *h*. If on our next measurement we find that $a + d < b + e$, then by interchanging *d* and *b* we have no change in the inequality. This tells us two things: (i) that *c* or *f* is not the culprit, and (ii) that *b* or *d* is also not the

culprit. If $a + d$ was equal to $b + e$, then c or f would be the counterfeit coin. Knowing at this point that either a or e is the counterfeit, we compare a with a good coin, say b. If $a = b$, then e is heavy, otherwise a must be light.

Figure 5.16 Eight Coins Decision Tree

By looking at this tree we see that all possibilities are covered, since there are 8 coins which can be heavy or light and there are 16 terminal nodes. Every path requires exactly 3 comparisons. Though viewing this problem as a decison tree is very useful it does not immediately give us an algorithm. To solve the 8 coins problem with a program, we must write a series of tests which mirror the structure of the tree. If we try to do this in SPARKS using the **if-then-else** statement, we see that the program looks like a dish of spaghetti. It is impossible to discern the flow of the program. Actually this type of processing is much more clearly expressed using the **case** statement.

We will make use of a procedure to do the last comparison.

```
procedure COMP(x,y,z)

//x is compared against the standard coin z//

if x > z then print (x 'heavy')

else print (y 'light')

end COMP
```

The procedure EIGHTCOINS appears on the next page. The program is now transparent and clearly mirrors the decision tree of figure 5.16.

5.8.3 Game Trees

Another interesting application of trees is in the playing of games such as tic-tac-toe, chess, nim, kalah, checkers, go, etc. As an example, let us consider the game of nim. This game is played by two players A and B. The game itself is described by a *board* which initially contains a pile of n toothpicks. The players A and B make moves alternately

```
procedure EIGHTCOINS

//eight weights are input; the different one is discovered using only
```

```
3 comparisons//

read (a, b, c, d, e, f, g, h)

case

: a + b + c = d + e + f : if g > h then call COMP(g,h,a)

else call COMP(h,g,a)

: a + b + c > d + e + f : case

: a + d = b + e: call COMP(c,f,a)

: a + d > b + e: call COMP(a,e,b)

: a + d < b + e: call COMP(b,d,a)

end

: a + b + c < d + e + f : case

: a + d = b + e: call COMP(f,c,a)

: a + d > b + e: call COMP(d,b,a)

: a + d < b + e: call COMP(e,a,b)

end

end

end EIGHTCOINS
```

with *A* making the first move. A *legal move* consists of removing either 1, 2 or 3 of the toothpicks from the pile. However, a player cannot remove more toothpicks than there are on the pile. The player who removes the last toothpick loses the game and the other player wins. The *board configuration* at any time is completely specified by the number of toothpicks remaining in the pile. At any time the game status is determined by the board configuration together with the player whose turn it is to make the next move. A *terminal board cofiguration* is one which represents either a *win, lose* or *draw* situation. All

other configurations are *nonterminal*. In nim there is only one terminal configuration: there are no toothpicks in the pile. This configuration is a win for player *A* if *B* made the last move, otherwise it is a win for *B*. The game of nim cannot end in a draw.

A sequence $C_1, ...,C_m$ of board configurations is said to be *valid* if:

(i) C_1 is the starting configuration of the game;

(ii) C_i, $0 < i < m$, are nonterminal configurations;

(iii) C_{i+1} is obtained from C_i by a legal move made by player *A* if *i* is odd and by player *B* if *i* is even. It is assumed that there are only finitely many legal moves.

A valid sequence $C_1, ...,C_m$ of board configurations with C_m a terminal configuration is an *instance* of the game. The *length* of the sequence $C_1,C_2, ...,C_m$ is *m*. A *finite game* is one in which there are no valid sequences of infinite length. All possible instances of a finite game may be represented by a *game tree*. The tree of figure 5.17 is the game free for nim with *n* = 6. Each node of the tree represents a board configuration. The root node represents the starting configuration C_1. Transitions from one level to the next are made via a move of *A* or *B*. Transitions from an odd level represent moves made by A. All other transitions are the result of moves made by *B*. Square nodes have been used in figure 5.17 to represent board configurations when it was *A*'s turn to move. Circular nodes have been used for other configurations. The edges from level 1 nodes to level 2 nodes and from level 2 nodes to level 3 nodes have been labeled with the move made by *A* and *B* respectively (for example, an edge labeled 1 means 1 toothpick is to be removed). It is easy to figure out the labels for the remaining edges of the tree. Terminal configurations are represented by leaf nodes. Leaf nodes have been labeled by the name of the player who wins when that configuration is reached. By the nature of the game of nim, player A can win only at leaf nodes on odd levels while *B* can win only at leaf nodes on even levels. The degree of any node in a game tree is at most equal to the number of distinct legal moves. In nim there are at most 3 legal moves from any configuration. By definition, the number of legal moves from any configuration is finite. The *depth* of a game tree is the length of a longest instance of the game. The depth of the nim tree of figure 5.17 is 7. Hence, from start to finish this game involves at most 6 moves. It is not difficult to see how similar game trees may be constructed for other finite games such as chess, tic-tac-toe, kalah, etc. (Strictly speaking, chess is not a finite game as it is possible to repeat board configurations in the game. We can view chess as a finite game by disallowing this possibility. We could, for instance, define the repetition of a board configuration as resulting in a draw.)

Now that we have seen what a game tree is, the next question is "of what use are they?" Game trees are useful in determining the next move a player should make. Starting at the initial configuration represented by the root of figure 5.17 player *A* is faced with the choice of making any one of three possible moves. Which one should he make? Assuming that player *A* wants to win the game, he should make the move that maximizes his chances of winning. For the simple tree of figure 5.17 this move is

not too difficult to determine. We can use an evaluation function $E(X)$ which assigns a numeric value to the board configuration X. This function is a measure of the value or worth of configuration X to player A. So, $E(X)$ is high for a configuration from which A has a good chance of winning and low for a configuration from which A has a good chance of losing. $E(X)$ has its maximum value for configurations that are either winning terminal configurations for A or configurations from which A is guaranteed to win regardless of B's countermoves. $E(X)$ has its minimum value for configurations from which B is guaranteed to win.

□

Figure 5.17 Complete Game Tree for Nim with n = 6

For a game such as nim with $n = 6$, whose game tree has very few nodes, it is sufficient to define $E(X)$ only for terminal configurations. We could define $E(X)$ as:

□

Using this evaluation function we wish to determine which of the configurations b, c, d player A should move the game into. Clearly, the choice is the one whose value is max $\{V(b), V(c), V(d)\}$ where $V(x)$ is the value of configuration x. For leaf nodes x, $V(x)$ is taken to be $E(x)$. For all other nodes x let $d \geq 1$ be *the degree of* x *and*

□

let $c_1, c_2, ..., c_d$ be the configurations represented by the children of x. Then $V(x)$ is defined by:

□

(5.3)

The justification for (5.3) is fairly simple. If x is a square node, then it is at an odd level and it will be A's turn to move from here if the game ever reaches this node. Since A wants to win he will move to that child node with maximum value. In case x is a circular node it must be on an even level and if the game ever reaches this node, then it will be B's turn to move. Since B is out to win the game for himself, he will (barring mistakes) make a move that will minimize A's chances of winning. In this case the next configuration will be □ $\{V(c_i)\}$. Equation (5.3) defines the *minimax* procedure to determine the value of a configuration x. This is illustrated on the hypothetical game of figure 5.18. P_{11} represents an arbitrary board configuration from which A has to make a move. The values of the leaf nodes are obtained by evaluating the function $E(x)$. The value of P_{11} is obtained by starting at the nodes on level 4 and computing their values using eq. (5.3). Since level 4 is a level with circular nodes all unknown values on this level may be obtained by taking the minimum of the children values. Next, values on levels 3, 2 and

1 may be computed in that order. The resulting value for P_{11} is 3. This means that starting from P_{11} the best A can hope to do is reach a configuration of value 3. Even though some nodes have value greater than 3, these nodes will not be reached, as B's countermoves will prevent the game from reaching any such configuration (assuming B's countermoves are optimal for B with respect to A's evaluation function). For example, if A made a move to P_{21}, hoping to win the game at P_{31}, A would indeed be surprised by B's countermove to P_{32} resulting in a loss to A. Given A's evaluation function and the game tree of figure 5.18, the best move for A to make is to configuration P_{22}. Having made this move, the game may still not reach configuration P_{52} as B would, in general, be using a different evaluation function, which might give different values to various board configurations. In any case, the *minimax* procedure can be used to determine the best move a player can make given his evaluation function. Using the minimax procedure on the game tree for nim (figure 5.17) we see that the value of the root node is $V(a) = 1$. Since $E(X)$ for this game was defined to be 1 iff A was guaranteed to win, this means that if A makes the optimal move from node a then no matter what B's countermoves A will win. The optimal move is to node b. One may readily verify that from b A can win the game independent of B's countermove!

Figure 5.18 Portion of Game Tree for a Hypothetical Game. The value of terminal nodes is obtained from the evaluation function E(x) for player A.

For games such as nim with $n = 6$, the game trees are sufficiently small that it is possible to generate the whole tree. Thus, it is a relatively simple matter to determine whether or not the game has a winning strategy. Moreover, for such games it is possible to make a decision on the next move by looking ahead all the way to terminal configurations. Games of this type are not very interesting since assuming no errors are made by either player, the outcome of the game is predetermined and both players should use similar evaluation functions, i.e., $E_A(X) = 1$ for X a winning configuration and $E_A(X) = -1$ for X a losing configuration for A; $E_B(X) = -E_A(X)$.

Of greater interest are games such as chess where the game tree is too large to be generated in its entirety. It is estimated that the game tree for chess has $>10^{100}$ nodes. Even using a computer which is capable of generating 10^{11} nodes a second, the complete generation of the game tree for chess would require more than 10^{80} years. In games with large game trees the decision as to which move to make next can be made only by looking at the game tree for the next few levels. The evaluation function $E(X)$ is used to get the values of the leaf nodes of the subtree generated and then eq. (5.3) can be used to get the values of the remaining nodes and hence to determine the next move. In a game such as chess it may be possible to generate only the next few levels (say 6) of the tree. In such situations both the quality of the resulting game and its outcome will depend upon the quality of the evaluating functions being used by the two players as well as of the algorithm being used to determine $V(X)$ by minimax for the current game configuration. The efficiency of this algorithm will limit the number of nodes of the search tree that can be generated and so will have an effect on the quality of the game.

Let us assume that player *A* is a computer and attempt to write an algorithm that *A* can use to compute *V*(*X*). It is clear that the procedure to compute *V*(*X*) can also be used to determine the next move that *A* should make. A fairly simple recursive procedure to evaluate *V*(*X*) using minimax can be obtained if we recast the definition of minimax into the following form:

□

(5.4)

where $e(X) = E(X)$ if *X* is a position from which A is to move and $e(X) = - E(X)$ otherwise.

Starting at a configuration *X* from which *A* is to move, one can easily prove that eq. (5.4) computes $V'(X) = V(X)$ as given by eq. (5.3). In fact, values for all nodes on levels from which *A* is to move are the same as given by eq. (5.3) while values on other levels are the negative of those given by eq. (5.3).

The recursive procedure to evaluate $V'(X)$ based on eq. (5.4) is then $VE(X,l)$. This algorithm evaluates $V'(X)$ by generating only *l* levels of the game tree beginning with *X* as root. One may readily verify that this algorithm traverses the desired subtree of the game tree in postorder.

```
procedure VE(X,l)

//compute V'(X) by looking at most l moves ahead. e(X) is the

evaluation function for player A. For convenience, it is assumed

that starting from any board configuration X the legal moves of

the game permit a transition only to the configurations C₁,C₂, ...,C_d

if X is not a terminal configuration.//

if X is terminal or l = 0 then return e(X)

ans □ - VE(C₁,l - 1          //traverse the first subtree//

for i □ 2 to d do          //traverse the remaining subtrees//

ans □ max {ans, - VE(C_i, l - 1)}
```

end

return (*ans*)

end *VE*

An Initial call to algorithm VE with $X = P_{11}$ and $l = 4$ for the hypothetical game of figure 5.18 would result in the generation of the complete game tree. The values of various configurations would be determined in the order: $P_{31}, P_{32}, P_{21}, P_{5l}, P_{52}, P_{53}, P_{41}, P_{54}, P_{55}, P_{56}, P_{42}, P_{33}, ...,P_{37},P_{24}, P_{11}$. It is possible to introduce, with relative ease, some heuristics into algorithm *VE* that will in general result in the generation of only a portion of the possible configurations while still computing $V'(X)$ accurately.

Consider the game tree of figure 5.18. After $V(P_{41})$ has been computed, it is known that $V(P_{33})$ is at least $V(P_{41})= 3$. Next, when $V(P_{55})$ is determined to be 2, then we know that $V(P_{42})$ is at most 2. Since P_{33} is a max position, $V(P_{42})$ cannot affect $V(P_{33})$. Regardless of the values of the remaining children of P_{42}, the value of P_{33} is not determined by $V(P_{42})$ as $V(P_{42})$ cannot be more than $V(P_{41})$. This observation may be stated more formally as the following rule: The *alpha* value of a max position is defined to be the minimum possible value for that position. *If the value of a min position is determined to be less than or equal to the alpha value of its parent, then we may stop generation of the remaining children of this min position.* Termination of node generation under this rule is known as *alpha cutoff.* Once $V(P_{41})$ in figure 5.18 is determined, the alpha value of P_{33} becomes 3. $V(P_{55}) \leq$ alpha value of P_{33} implies that P_{56} need not be generated.

A corresponding rule may be defined for min positions. The *beta* value of a min position is the maximum possible value for that position. *If the value of a max position is determined to be greater than or equal to the beta value of its parent node, then we may stop generation of the remaining children of this max position.* Termination of node generation under this rule is called *beta cutoff.* In figure 5.18, once $V(P_{35})$ is determined, the beta value of P_{23} is known to be at most $-\infty$. Generation of P_{57}, P_{58}, P_{59} gives $V(P_{43}) = 0$. Thus, $V(P_{43})$ is greater than or equal to the beta value of P_{23} and we may terminate the generation of the remaining children of P_{36}. The two rules stated above may be combined together to get what is known as *alpha-beta pruning.* When alpha-beta pruning is used on figure 5.18, the subtree with root P_{36} is not generated at all! This is because when the value of P_{23} is being determined the alpha value of P_{11} is 3. $V(P_{35})$ is less than the alpha value of P_{11} and so an alpha cutoff takes place. It should be emphasized that the alpha or beta value of a node is a dynamic quantity. Its value at any time during the game tree generation depends upon which nodes have so far been generated and evaluated.

In actually introducing alpha-beta pruning into algorithm *VE* it is necessary to restate this rule in terms of the values defined by eq. (5.4). Under eq. (5.4) all positions are max positions since the values of the min positions of eq. (5.3) have been multiplied by -1. The alpha-beta pruning rule now reduces to the following rule: let the *B* value of a position be the minimum value that that position can have.

For any position X, let B be the B - value of its parent and let D = -B. Then, if the value of X is determined the greater than or equal to D, we may terminate generation of the remaining children of X. Incorporating this rule into algorithm VE is fairly straightforward and results in algorithm VEB. This algorithm has the additional parameter *D* which is the negative of the *B* value of the parent of *X*.

```
procedure VEB(X,l,D)

//determine V'(X) as in eq. (5.4) using the B-rule and looking

only l moves ahead. Remaining assumptions and notation are

the same as for algorithm VE.//

if X is terminal or l = 0 then return e(x)

ans ☐ -∞.            //current lower bound on V'(x)//

for i ☐ l to d do

ans ☐ max {ans, - VEB(Cᵢ,l-1,-ans)}

if ans ≥ D then return (ans)   //use B-rule//

end

return (ans)

end VEB
```

If *Y* is a position from which *A* is to move, then the initial call VEB(*Y*,*l*, ∞) *correctly computes* V' *(Y) with an* l *move look ahead. Further pruning of the game tree may be achieved by realizing that the* B - *value of a node* X *places a lower bound on the value grandchildren of* X *must have in order to affect* X's *value. Consider the subtree of figure 5.19(a). If* V' (GC(X)) ≤ B *then* V' (C(X)) ≥ -B. *Following the evaluation of* C(X), *the* B-*value of* X *is max{B,-V' (C(X))} =* B *as* V'(C(X)) ≥ -B. *Hence unless* V' (GC (X)) > B, *it cannot affect* V' (X) *and so* B *is a lower bound on the value* GC(X) *should have. Incorporating this lowerbound into algorithm VEB yields algorithm AB. The additional parameter LB is a lowerbound on the value* X *should have.*

```
procedure AB(X,l,LB,D)
```

```
//same as algorithm VEB. LB is a lowerbound on V' (X)//

if X is terminal or l = 0 then return e(X)

ans [ ] LB              //current lowerbound on V' (X)//

for i [ ] 1 to d do

ans [ ] max{ans,-AB(C_i,l-1,-D,-ans)}

if ans ≥ D then return (ans)

end

return (ans)

end AB.
```

One may easily verify that the initial call $AB(Y,l, -\infty,\infty)$ gives the same result as the call $VE(Y,l)$.

Figure 5.19(b) shows a hypothetical game tree in which the use of algorithm *AB* results in greater pruning than achieved by algorithm VEB. Let us first trace the action of VEB on the tree of figure 5.19 (b). We assume the initial call to be $VEB(P_1,l,\infty)$ where l is the depth of the tree. After examining the left subtree of P_1, the B value of P_1 is set to 10 and nodes P_3, P_4, P_5 and P_6 are generated. Following this, $V'(P_6)$ is determined to be 9 and then the B-value of P_5 becomes -9. Using this, we continue to evaluate the node P_7. In the case of *AB* however, since the B-value of P_1 is 10, the lowerbound for P_4 is 10 and so the effective B-value of P_4 becomes 10. As a result the node P_7 is not generated since no matter what its value $V'(P_5) \geq$ -9 and this will not enable $V'(P_4)$ to reach its lower bound.

[ ]

Figure 5.19 Game trees showing lower bounding

5.9 COUNTING BINARY TREES

As a conclusion to our chapter on trees, we determine the number of distinct binary trees having n nodes. We know that if $n = 0$ or $n = 1$ there is one such tree. If $n = 2$, then there are two distinct binary trees

and if $n = 3$, there are five

How many distinct binary trees are there with n nodes?

Before solving this problem let us look at some other counting problems that are equivalent to this one.

In section 5.4 we introduced the notion of preorder, inorder and postorder traversals. Suppose we are given the preorder sequence

A B C D E F G H I

and the inorder sequence

B C A E D G H F I

of the same binary tree. Does such a pair of sequences uniquely define a binary tree? Asked another way, can the above pair of sequences come from more than one binary tree. We can construct the binary tree which has these sequences by noticing that the first letter in preorder, *A*, must be the root and by the definition of inorder all nodes preceding *A* must occur in the left subtree and the remaining nodes occur in the right subtree.

This gives us

as our first approximation to the correct tree. Moving right in the preorder sequence we find *B* as the next root and from the inorder we see *B* has an empty left subtree and *C* is in its right subtree. This gives

as the next approximation. Continuing in this way we arrive at the binary tree

By formalizing this argument, see the exercises, we can verify that every binary tree has a unique pair of preorder-inorder scquences.

Let the nodes of an *n* node binary tree be numbered 1 to *n*. The *inorder permutation* defined by such a binary tree is the order in which its nodes are visited during an inorder traversal of the tree. A *preorder permutation* is similarly defined.

As an example, consider the binary tree above with the following numbering of nodes:

Its preorder permutation is 1,2, ...,9 and its inorder permutation is 2,3,1,5,4,7,8,6,9.

If the nodes of a binary tree are numbered such that its preorder permutation is 1,2, ...,*n*, then from our earlier discussion it follows that distinct binary trees define distinct inorder permutations. The number of distinct binary trees is thus equal to the number of distinct inorder permutations obtainable from binary trees having the preorder permutation 1,2, ...,*n*.

Using this concept of an inorder permutation, it is possible to show that the number of distinct permutations obtainable by passing the numbers 1 to *n* through a stack and deleting in all possible ways is equal to the number of distinct binary trees with *n* nodes (see the exercises). If we start with the numbers 1,2,3 then the possible permutations obtainable by a stack are

1,2,3; 1,3,2; 2,1,3; 3,2,1; 3, 2, 1;

It is not possible to obtain 3,1,2. Each of these five permutations corresponds to one of the five distinct binary trees with 3 nodes

Another problem which surprisingly has connection with the previous two is the following: we have a product of *n* matrices

$$M_1 * M_2 * M_3 * ... * M_n$$

that we wish to compute. We can perform these operations in any order because multiplication of matrices is associative. We ask the question: how many different ways can we perform these multiplications? For example, if *n* = 3, there are two possibilities

$(M_1 * M_2) * M_3$ and $M_1 * (M_2 * M_3)$

and if *n* = 4, there are five ways

$((M_1 * M_2) * M_3) * M_4$, $(M_1 * (M_2 * M_3)) * M_4$,

$M_1 * ((M_2 * M_3) * M_4)$

$(M_1 * (M_2 * (M_3 * M_4))), ((M_1 * M_2) * (M_3 * M_4))$

Let b_n be the number of different ways to compute the product of n matrices. Then $b_2 = 1$, $b_3 = 2$, $b_4 = 5$. Let M_{ij}, $i \leq j$, be the product $M_i * M_{i+1} * ... * M_j$. The product we wish to compute is M_{1_n}. M_{1_n} may be computed by computing any one of the products $M_{1_i} * M_{i+1,_n}$, $1 \leq i < n$. The number of ways to obtain M_{1_i} and $M_{i+1,_n}$ is b_i and b_{n-i} respectively. Therefore, letting $b_1 = 1$ we have:

□

If we can determine an expression for b_n only in terms of n, then we have a solution to our problem. Now instead let b_n be the number of distinct binary trees with n nodes. Again an expression for b_n in terms of n is what we want. Then we see that b_n is the sum of all possible binary trees formed in the following way, a root and two subtrees with b_i and b_{n-i-1} nodes,

□

for $0 \leq i \leq n - 1$. This says that

□

(5.5)

This formula and the previous one are essentially the same.

So, *the number of binary trees with n nodes, the number of permutations of 1 to n obtainable with a stack, and the number of ways to multiply n + 1 factors are all equal to the same number!*

To obtain this number we must solve the recurrence of eq. (5.5). To begin we let

□

(5.6)

which is the generating function for the number of binary trees. Next, observe that by the recurrence relation we get the identity

$x\,B^2(x) = B(x) - 1$

Using the formula to solve quadratics and the fact (eq. (5.5)) that $B(0) = b_0 = 1$ we get:

☐

It is not clear at this point that we have made any progress but by using the binomial theorem to expand $(1 - 4x)^{1/2}$ we get

☐

(5.7)

Comparing eqs. (5.6) and (5.7)we see that b_n which is the coefficient of x^n in $B(x)$ is:

☐

Some simplification yields the more compact form

☐

which is approximately

$b_n = O(4^n/n^{3/2})$

REFERENCES

For other representations of trees see

The Art of Computer Programming: Fundamental Algorithms, by D. Knuth, second edition, Addison-Wesley, Reading, 1973.

For the use of trees in generating optimal compiled code see

"The generation of optimal code for arithmetic expressions" by R. Sethi and J. Ullman, *JACM*, vol. 17, no. 4, October 1970, pp. 715-728.

"The generation of optimal code for a stack machine" by J. L. Bruno and T. Lassagne, *JACM*, vol. 22, no. 3, July 1975, pp. 382-396.

Algorithm INORDER4 of the exercises is adapted from

"An improved algorithm for traversing binary trees without auxiliary stack," by J. Robson, *Information Processing Letters*, vol. 2, no. 1, March 1973, p. 12-14.

Further tree traversal algorithms may be found in:

"Scanning list structures without stacks and tag bits," by G. Lindstrom, *Information Processing Letters*, vol. 2, no. 2, June 1973, p. 47-51.

"Simple algorithms for traversing a tree without an auxiliary stack," by B. Dwyer, *Information Processing Letters*, vol. 2, no. 5, Dec. 1973, p. 143-145.

The use of threads in conneetion with binary trees is given in

"Symbol manipulation by threaded lists," by A. Perlis and C. Thornton, *CACM*, vol. 3, no. 4, April 1960, pp. 195-204.

For a further analysis of the set representation problem see

The Design and Analysis of Computer A1gorithms, A. Aho, J. Hopcroft and J. Ullman, Addison-Wesley, Reading, 1974.

The computing time analysis of the UNION-FIND algorithms may be found in:

"Efficiency of a good but not linear set union algorithm" by R. Tarjan, *JACM*, vol. 22, no. 2, April 1975, pp. 215-225.

Our discussion of alpha-beta cutoffs is from

"An analysis of alpha beta cutoffs" by D. Knuth, Stanford Technical Report 74-441, Stanford University, 1975.

For more on game playing see

Problem Solving Methods in Artificial Intelligence by N. Nilsson, McGraw-Hill, New York, 1971.

Artificial Intelligence: The Heuristic Programming Approach by J. Slagle, McGraw-Hill, New York, 1971.

EXERCISES

1. For the binary tree below list the terminal nodes, the nonterminal nodes and the level of each node.

2. Draw the internal memory representation of the above binary tree using (a) sequential, (b) linked, and (c) threaded linked representations.

3. Write a procedure which reads in a tree represented as a list as in section 5.1 and creates its internal representation using nodes with 3 fields, TAG, DATA, LINK.

4. Write a procedure which reverses the above process and takes a pointer to a tree and prints out its list representation.

5. Write a nonrecursive version of procedure PREORDER.

6. Write a nonrecursive version of procedure POSTORDER without using **go to**'s**.**

7. Rework INORDER3 so it is as fast as possible. (Hint: minimize the stacking and the testing within the loop.)

8. Write a nonrecursive version of procedure POSTORDER using only a fixed amount of additional space. (See exercise 36 for details)

9. Do exercise 8 for the case of PREORDER.

10. Given a tree of names constructed as described in section 5.5 prove that an inorder traversal will always print the names in alphabetical order.

Exercises 11-13 assume a linked representation for a binary tree.

11. Write an algorithm to list the DATA fields of the nodes of a binary tree T by level. Within levels nodes are to be listed left to right.

12. Give an algorithm to count the number of leaf nodes in a binary tree T. What is its computing time?

13. Write an algorithm SWAPTREE(T) which takes a binary tree and swaps the left and right children of every node. For example, if T is the binary tree

☐

14. Devise an external representation for formulas in the propositional calculus. Write a procedure which reads such a formula and creates a binary tree representation of it. How efficient is your procedure?

15. Procedure POSTORDER-EVAL must be able to distinguish between the symbols ☐ , ☐ , ¬ and a pointer in the DATA field of a node. How should this be done?

16. What is the computing time for POSTORDER-EVAL? First determine the logical parameters.

17. Write an algorithm which inserts a new node *T* as the left child of node *S* in a threaded binary tree. The left pointer of *S* becomes the left pointer of *T*.

18. Write a procedure which traverses a threaded binary tree in postorder. What is the time and space requirements of your method?

19. Define the inverse transformation of the one which creates the associated binary tree from a forest. Are these transformations unique?

20. Prove that preorder traversal on trees and preorder traversal on the associated binary tree gives the same result.

21. Prove that inorder traversal for trees and inorder traversal on the associated binary tree give the same result.

22. Using the result of example 5. 3, draw the trees after processing the instruction UNION(12,10).

23. Consider the hypothetical game tree:

☐

(a) Using the minimax technique (eq. (5.3)) obtain the value of the root node .

(b) What move should player *A* make?

(c) List the nodes of this game tree in the order in which their value is computed by algorithm *VE*.

(d) Using eq. (5.4) compute *V'* (*X*) for every node *X* in the tree.

(e) Which nodes of this tree are not evaluated during the computation of the value of the root node using

algorithm AB with $X = $ root, $l = {}^\infty$, $LB = -{}^\infty$ and $D = {}^\infty$?

24. Show that $V'(X)$ computed by eq. (5.4) is the same as $V(X)$ computed by eq. (5.3) for all nodes on levels from which A is to move. For all other nodes show that $V(X)$ computed by eq. (5.3) is the negative of $V'(X)$ computed by eq. (5.4).

25. Show that algorithm AB when initially called with $LB = -{}^\infty$ and $D = {}^\infty$ yields the same results as VE does for the same X and l.

26. Prove that every binary tree is uniquely defined by its preorder and inorder sequences .

27. Do the inorder and postorder sequences of a binary tree uniquely define the binary tree? Prove your answer.

28. Answer exercise 27 for preorder and postorder.

29. Write an algorithm to construct the binary tree with a given preorder and inorder sequence.

30. Do exercise 29 for inorder and postorder.

31. Prove that the number of distinct permutations of 1,2, ...,n obtainable by a stack is equal to the number of distinct binary trees with n nodes. (Hint: Use the concept of an inorder permutation of a tree with preorder permutation 1,2, ...,n).

32. Using Stirling's formula derive the more accurate value of the number of binary trees with n nodes,

□

33. Consider threading a binary tree using preorder threads rather than inorder threads as in the text. Is it possible to traverse a binary tree in preorder without a stack using these threads?

34. Write an algorithm for traversing an inorder threaded binalry tree in preorder.

35. The operation PREORD(btree) □ queue returns a queue whose elements are the data items of btree in preorder. Using the operation APPENDQ(queue, queue) □ queue which concatenates two queues, PREORD can be axiomatized by

```
PREORD(CREATE)  ::  = MTQ
```

```
PREORD(MAKBT(p,d,r))  ::  =
```

APPENDQ(APPENDQ(ADDQ(MTQ,d),(PREORD(p)),PREORD(r))

Devise similar axioms for INORDER AND POSTORDER.

36. The algorithm on page 281 performs an inorder traversal without using threads, a stack or a PARENT field. Verify that the algorithm is correct by running it on a variety of binary trees which cause every statement to execute at least once. Before attempting to study this algorithm be sure you understand MARK2 of section 4.10.

37. Extend the equivalence algorithm discussed in section 5.8.1 so it handles the equivalencing of arrays (see section 4.6). Analyze the computing time of your solution.

38. [Wilczynski] Following the conventions of LISP assume nodes with two fields ☐. If $A = ((a(bc)))$ then HEAD(A) = ($a(bc)$), TAIL(A) = NIL, HEAD(HEAD(A)) = a, TAIL(HEAD(A)) = ((bc)). CONS(A, B) gets a new node T, stores A in its HEAD, B in its TAIL and returns T. B must always be a list. If $L = a$, $M = (bc)$ then CONS(L,M) = (abc), CONS(M,M) = ((bc)bc). Three other useful functions are: ATOM(X) which is true if X is an atom else false, NULL(X) which is true if X is NIL else false, EQUAL(X,Y) which is true if X and Y are the same atoms or equivalent lists else false.

a) Give a sequence of HEAD, TAIL operations for extracting a from the lists: ((cat)), ((a)), ($mart$), (((cb))a).

b) Write recursive procedures for: COPY, REVERSE, APPEND.

c) Implement this "LISP" subsystem. Store atoms in an array, write procedures MAKELIST and LISTPRINT for input and output of lists.

```
line procedure INORDER4(T)

//inorder traversal of binary tree T using a fixed amount of

additional storage//

1        if T = 0 then return      //empty binary tree//

2        top ☐ last_right ☐ 0; p ☐ q ☐ T      //initialize//

3        loop

4           loop          //move down as far as possible//
```

```
5          case

6               :LCHILD(p) = 0 and RCHILD(p) = 0:

//can't move down//

7                              print (DATA(p)); exit

8               :LCHILD(p) = 0:       //move to RCHILD(p)//

9                         print(DATA(p))        //visit p//

10                          r ☐ RCHILD(p); RCHILD(p) ☐ q;

q ☐ p;p ☐ r

11              :else:    //move to LCHILD(p)//

12                      r ☐ LCHILD(p); LCHILD(p) ☐ q; q ☐ p;

p ☐ r

13          end

14       forever

//p is a leaf node, move upwards to a node whose right

subtree hasn't yet been examined//

15       av ☐ p            //leaf node to be used in stack//

16       loop         //move up from p//

17          case

18          :p = T: return       //can't move up from root//

19          :LCHILD(q) = 0:          //q is linked via RCHILD//
```

```
20              r □ RCHILD(q); RCHILD(q) □ p; p □ q; q □ r

21          :RCHILD(q) = 0:       //q is linked via LCHILD//

22              r □ LCHILD(q); LCHILD(q) □ p; p □ q; q □ r;
```

print(*DATA*(*p*))

```
23          :else:       //check if p is RCHILD of q//

24              if q = last_right then    [//p is RCHILD of q//

25                  r □ top; last--right □ LCHILD(r)    //update
```

last--right//

```
26                  top □ RCHILD(r);         //unstack//

27                  LCHILD(r) □ RCHILD(r) □ 0          //reset leaf
```

node links//

```
28                  r □ RCHILD(q); RCHILD(q) □ p; p □ q; q □ r]

29                                  else [//p is LCHILD of q//

30                  print (DATA(q))      //visit q//

31                  LCHILD(av) □ last_right; RCHILD(av) □ top;
```

top □ *av*

```
32                  last_right □ q

33                  r □ LCHILD(q); LCHILD(q) □ p
```

//restore link to *p*//

34   r_1 ☐ `RCHILD(`q`);` `RCHILD(`q`)` ☐ `r;` `p` ☐ r_1`; exit`

`//move right//`]

35   **end**

36   **forever**

37   **forever**

38   **end** *INORDER*4

Go to [Chapter 6](#)   Back to [Table of Contents](#)

CHAPTER 6: GRAPHS

6.1 TERMINOLOGY AND REPRESENTATIONS

6.1.1 Introduction

The first recorded evidence of the use of graphs dates back to 1736 when Euler used them to solve the now classical Koenigsberg bridge problem. In the town of Koenigsberg (in Eastern Prussia) the river Pregal flows around the island Kneiphof and then divides into two. There are, therefore, four land areas bordering this river (figure 6.1). These land areas are interconnected by means of seven bridges *a-g. The land areas themselves are labeled* A-D. The Koenigsberg bridge problem is to determine whether starting at some land area it is possible to walk across all the bridges exactly once returning to the starting land area. One possible walk would be to start from land area *B*; walk across bridge *a* to island *A*; take bridge *e* to area *D*; bridge *g* to *C*; bridge *d* to *A*; bridge *b* to *B* and bridge *f* to *D*. This walk does not go across all bridges exactly once, nor does it return to the starting land area *B*. Euler answered the Koenigsberg bridge problem in the negative: The people of Koenigsberg will not be able to walk across each bridge exactly once and return to the starting point. He solved the problem by representing the land areas as vertices and the bridges as edges in a graph (actually a multigraph) as in figure 6.1(b). His solution is elegant and applies to all graphs. Defining the *degree* of a vertex to be the number of edges incident to it, Euler showed that there is a walk starting at any vertex, going through each edge exactly once and terminating at the start vertex iff the degree of each, vertex is even. A walk which does this is called *Eulerian*. There is no Eulerian walk for the Koenigsberg bridge problem as all four vertices are of odd degree.

Since this first application of graphs, they have been used in a wide variety of applications. Some of these applications are: analysis of electrical circuits, finding shortest routes, analysis of project planning, identification of chemical compounds, statistical mechanics, genetics, cybernetics, linguistics, social sciences, etc. Indeed, it might well be said that of all mathematical structures, graphs are the most widely used.

☐

Figure 6.1 Section of the river Pregal in Koenigsberg and Euler's graph.

6.1.2 Definitions and Terminology

A graph, *G*, consists of two sets *V* and *E*. *V* is a finite non-empty set of *vertices*. *E* is a set of pairs of vertices, these pairs are called *edges*. *V*(*G*) and *E*(*G*) will represent the sets of vertices and edges of graph *G*. We will also write *G* = (*V,E*) to represent a graph. In an *undirected graph* the pair of vertices

representing any edge is unordered . Thus, the pairs (v_1, v_2) and (v_2, v_1) represent the same edge. In a *directed graph* each edge is represented by a directed pair (v_1, v_2). v_1 is the *tail* and v_2 the *head* of the edge. Therefore $<v_2, v_1>$ and $<v_1, v_2>$ represent two different edges. Figure 6.2 shows three graphs G_1, G_2 and G_3.

☐

Figure 6.2 Three sample graphs.

The graphs G_1 and G_2 are undirected. G_3 is a directed graph.

$V(G_1) = \{1,2,3,4\}; E(G_1) = \{(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)\}$

$V(G_2) = \{1,2,3,4,5,6,7\}; E(G_2) = \{(1,2),(1,3),(2,4),(2,5),(3,6),(3,7)\}$

$V(G_3) = \{1,2,3\}; E(G_3) = \{<1,2>, <2,1>, <2,3>\}$.

Note that the edges of a directed graph are drawn with an arrow from the tail to the head. The graph G_2 is also a tree while the graphs G_1 and G_3 are not. Trees can be defined as a special case of graphs, but we need more terminology for that. If (v_1,v_2) or $<v_1,v_2>$ is an edge in $E(G)$, then we require $v_1 \neq v_2$. In addition, since $E(G)$ is a set, a graph may not have multiple occurrences of the same edge. When this restriction is removed from a graph, the resulting data object is referred to as a multigraph. The data object of figure 6.3 is a multigraph which is not a graph.

The number of distinct unordered pairs (v_i,v_j) with $v_i \neq v_j$ in a graph with n vertices is $n(n-1)/2$. This is the maximum number of edges in any n vertex undirected graph. An n vertex undirected graph with exactly $n(n-1)/2$ edges is said to be *complete*. G_1 is the complete graph on 4 vertices while G_2 and G_3 are not complete graphs. In the case of a directed graph on n vertices the maximum number of edges is $n(n-1)$.

If (v_1,v_2) is an edge in $E(G)$, then we shall say the vertices v_1 and v_2 are *adjacent* and that the edge (v_1, v_2) is *incident* on vertices v_1 and v_2. The vertices adjacent to vertex 2 in G_2 are 4, 5 and 1. The edges incident on vertex 3 in G_2 are $(1,3)$, $(3,6)$ and $(3,7)$. If $<v_1,v_2>$ is a directed edge, then vertex v_1 will be said to be *adjacent to v_2* while v_2 is *adjacent from v_1*. The edge $<v_1,v_2>$ is incident to v_1 and v_2. In G_3 the edges incident to vertex 2 are $<1,2>$, $<2,1>$ and $<2,3>$.

☐

Figure 6.3 Example of a multigraph that is not a graph.

A *subgraph* of G is a graph G' such that $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$. Figure 6.4 shows some of the subgraphs of G_1 and G_3.

A *path* from vertex v_p to vertex v_q in graph G is a sequence of vertices $v_p, v_{i1}, v_{i2}, ..., v_{in}, v_q$ such that $(v_p, v_{i1}), (v_{i1}, v_{i2}), ..., (v_{in}, v_q)$ are edges in $E(G)$. If G' is directed then the path consists of $<v_p, v_{i1}>, <v_i, v_{i2}>, ..., <v_{in}, v_q>$, edges in $E(G')$. The *length* of a path is the number of edges on it. A *simple path* is a path in which all vertices except possibly the first and last are distinct. A path such as (1,2) (2,4) (4,3) we write as 1,2,4,3. Paths 1,2,4,3 and 1,2,4,2 are both of length 3 in G_1. The first is a simple path while the second is not. 1,2,3 is a simple directed path in G_3. 1,2,3,2 is not a path in G_3 as the edge $<3,2>$ is not in $E(G_3)$. A *cycle* is a simple path in which the first and last vertices are the same. 1,2,3,1 is a cycle in G_1. 1,2,1 is a cycle in G_3. For the case of directed graphs we normally add on the prefix "directed" to the terms cycle and path. In an undirected graph, G, two vertices v_1 and v_2 are said to be *connected* if there is a path in G from v_1 to v_2 (since G is undirected, this means there must also be a path from v_2 to v_1). An undirected graph is said to be connected if for every pair of distinct vertices v_i, v_i in $V(G)$ there is a path from v_i to v_j in G. Graphs G_1 and G_2 are connected while G_4 of figure 6.5 is not. A *connected component* or simply a component of an undirected graph is a *maximal* connected subgraph. G_4 has two components H_1 and H_2 (see figure 6.5). A *tree* is a connected acyclic (i.e., has no cycles) graph . A directed graph G is said to be *strongly connected* if for every pair of distinct vertices v_i, v_j in $V(G)$ there is a directed path from v_i to v_j and also from v_j to v_i. The graph G_3 is not strongly connected as there is no path from v_3 to v_2. A *strongly connected component* is a maximal subgraph that is strongly connected. G_3 has two strongly connected components.

(a) Some of the subgraphs of G$_1$

(b) Some of the subgraphs of G$_3$

Figure 6.4 (a) Subgraphs of G$_1$ and (b) Subgraphs of G$_3$

Figure 6.5 A graph with two connected components.

☐

Figure 6.6 Strongly connected components of G$_3$.

The degree of a vertex is the number of edges incident to that vertex. The degree of vertex 1 in G_1 is 3. In case G is a directed graph, we define the *in-degree* of a vertex v to be the number of edges for which v is the head. The *out-degree* is defined to be the number of edges for which v is the tail. Vertex 2 of G_3 has in-degree 1, out-degree 2 and degree 3. If d_i is the degree of vertex i in a graph G with n vertices and e edges, then it is easy to see that $e = (1/2)\,\boxed{}$.

In the remainder of this chapter we shall refer to a directed graph as a *digraph*. An undirected graph will sometimes be referred to simply as a graph.

6.1.3 Graph Representations

While several representations for graphs are possible, we shall study only the three most commonly used: adjacency matrices, adjacency lists and adjacency multilists. Once again, the choice of a particular representation will depend upon the application one has in mind and the functions one expects to perform on the graph.

Adjacency Matrix

Let G = (V,E) be a graph with n vertices, $n \ge 1$. The adjacency matrix of G is a 2-dimensional n $\times$ n array, say A, with the property that $A(i,j) = 1$ iff the edge (v_i,v_j) ($<v_i,v_j>$ for a directed graph) is in $E(G)$. $A(i,j) = 0$ if there is no such edge in G. The adjacency matrices for the graphs G_1, G_3 and G_4 are shown in figure 6.7. The adjacency matrix for an undirected graph is symmetric as the edge (v_i,v_j) is in $E(G)$ iff the edge (v_j,v_i) is also in $E(G)$. The adjacency matrix for a directed graph need not be symmetric (as is the case for G_3). The space needed to represent a graph using its adjacency matrix is n_2 bits. About half this space can be saved in the case of undirected graphs by storing only the upper or lower triangle of the matrix.

From the adjacency matrix, one may readily determine if there is an edge connecting any two vertices i and j. For an undirected graph the degree of any vertex i is its row sum $\boxed{}$ A(i,j). For a directed graph the row sum is the out-degree while the column sum is the in-degree. Suppose we want to answer a nontrivial question about graphs such as: How many edges are there in G or is G connected. Using adjacency matrices all algorithms will require at least $O(n^2)$ time as $n^2 - n$ entries of the matrix (diagonal entries are zero) have to be examined. When graphs are sparse, i.e., most of the terms in the adjacency matrix are zero, one would expect that the former questions would be answerable in significantly less time, say O(e + n) where e is the number of edges in G and e $\ll$ n^2/2. Such a speed up can be made possible through the use of linked lists in which only the edges that are in G are represented. This leads

to the next representation for graphs.

Figure 6.7 Adjacency matrices for (i) G$_1$, (ii) G$_3$ and (iii) G$_4$.

Adjacency Lists

In this representation the *n* rows of the adjacency matrix are represented as *n* linked lists. There is one list for each vertex in *G*. The nodes in list *i* represent the vertices that are adjacent from vertex *i*. Each node has at least two fields: VERTEX and LINK. The VERTEX fields contain the indices of the vertices adjacent to vertex *i*. The adjacency lists for G_1, G_3 and G_4 are shown in figure 6.8. Each list has a headnode. The headnodes are sequential providing easy random access to the adjacency list for any particular vertex. In the case of an undirected graph with *n* vertices and *e* edges, this representation requires *n* head nodes and 2*e* list nodes. Each list node has 2 fields. In terms of the number of bits of storage needed, this count should be multiplied by log *n* for the head nodes and log *n* + log *e* for the list nodes as it takes $O(\log m)$ bits to represent a number of value *m*. Often one can sequentially pack the nodes on the adjacency lists and eliminate the link fields.

(i) Adjacency list for G$_1$

(ii) Adjacency lists for G$_3$

(iii) Adjacency list for G$_4$

Figure 6.8 Adjacency Lists

The degree of any vertex in an undirected graph may be determined by just counting the number of nodes in its adjacency list. The total number of edges in *G* may, therefore, be determined in time $O(n + e)$. In the case of a digraph the number of list nodes is only *e*. The out-degree of any vertex may be determined by counting the number of nodes on its adjacency list. The total number of edges in *G* can, therefore, be determined in $O(n + e)$. Determining the in-degree of a vertex is a little more complex. In case there is a need to repeatedly access all vertices adjacent to another vertex then it may be worth the effort to keep another set of lists in addition to the adjacency lists. This set of lists, called *inverse adjacency lists*, will contain one list for each vertex. Each list will contain a node for each vertex

adjacent to the vertex it represents (figure 6.9). Alternatively, one could adopt a

Figure 6.9 Inverse adjacency lists for G₃.

simplified version of the list structure used for sparse matrix representation in section 4.6. Each node would now have four fields and would represent one edge. The node structure would be

Figure 6.10 shows the resulting structure for the graph G_3 . The headnodes are stored sequentially.

The nodes in the adjacency lists of figure 6.8 were ordered by the indices of the vertices they represented. It is not necessary that lists be ordered in this way and, in general, the vertices may appear in any order. Thus, the adjacency lists of figure 6.11 would be just as valid a representation of G_1.

Adjacency Multilists

In the adjacency list representation of an undirected graph each edge (v_i,v_j) is represented by two entries, one on the list for v_i and the

Figure 6.10 Orthogonal List Representation for G₃.

Figure 6.11 Alternate Form Adjacency List for G₁.

other on the list for v_j. As we shall see, in some situations it is necessary to be able to determine the second entry for a particular edge and mark that edge as already having been examined. This can be accomplished easily if the adjacency lists are actually maintained as multilists (i.e., lists in which nodes may be shared among several lists). For each edge there will be exactly one node, but this node will be in two lists, i.e., the adjacency lists for each of the two nodes it is incident to. The node structure now becomes where

M is a one bit mark field that may be used to indicate whether or not the edge has been examined. The

storage requirements are the same as for normal adjacency lists except for the addition of the mark bit M. Figure 6.12 shows the adjacency multilists for G_1. We shall study multilists in greater detail in Chapter 10.

Sometimes the edges of a graph have weights assigned to them. These weights may represent the distance from one vertex to another or the

cost of going from one vertex to an adjacent vertex. In this case the adjacency matrix entries $A(i,j)$ would keep this information, too. In the case of adjacency lists and multilists this weight information may be kept in the list nodes by including an additional field. A graph with weighted edges is called a *network*.

☐

```
The lists are:   vertex 1:  N1 ☐ N2 ☐ N3

                 vertex 2:  N1 ☐ N4 ☐ N5

                 vertex 3:  N2 ☐ N4 ☐ N6

                 vertex 4:  N3 ☐ N5 ☐ N6
```

Figure 6.12 Adjacency Multilists for G$_1$.

6.2 TRAVERSALS, CONNECTED COMPONENTS AND SPANNING TREES

Given the root node of a binary tree, one of the most common things one wishes to do is to traverse the tree and visit the nodes in some order. In the chapter on trees, we defined three ways (preorder, inorder, and postorder) for doing this. An analogous problem arises in the case of graphs. Given an undirected graph $G = (V,E)$ and a vertex v in $V(G)$ we are interested in visiting all vertices in G that are reachable from v (i.e., all vertices connected to v). We shall look at two ways of doing this: Depth First Search and Breadth First Search.

Depth First Search

Depth first search of an undirected graph proceeds as follows. The start vertex v is visited. Next an unvisited vertex w adjacent to v is selected and a depth first search from w initiated. When a vertex u is reached such that all its adjacent vertices have been visited, we back up to the last vertex visited which

has an unvisited vertex *w* adjacent to it and initiate a depth first seaerch from *w*. The search terminates when no unvisited vertex can be reached from any of the visited one. This procedure is best described recursively as in

procedure *DFS(v)*

//Given an undirected graph G = (V,E) with n vertices and an array

VlSlTED(n) initially set to zero, this algorithm visits all vertices

reachable from *v. G* and *VISITED* are global.//

VISITED (v) □ 1

for *each vertex w adjacent to v* **do**

if *VISlTED(w)* = 0 **then call** *DFS(w)*

end

end *DFS*

In case *G* is represented by its adjacency lists then the vertices *w* adjacent to *v* can be determined by following a chain of links. Since the algorithm DFS would examine each node in the adjacency lists at most once and there are 2*e* list nodes, the time to complete the search is *O(e)*. If *G* is represented by its adjacency matrix, then the time to determine all vertices adjacent to *v* is *O(n)*. Since at most *n* vertices are visited, the total time is $O(n^2)$.

The graph *G* of figure 6.13(a) is represented by its adjacency lists as in figure 6.13(b). If a depth first search is initiated from vertex v_1, then the vertices of *G* are visited in the order: v_1, v_2, v_4, v_8, v_5, v_6, v_3, v_7. One may easily verify that DFS (v_1) visits all vertices connected to v_1. So, all the vertices visited, together with all edges in *G* incident to these vertices form a connected component of *G*.

Breadth First Search

Starting at vertex *v* and marking it as visited, breadth first search differs from depth first search in that all unvisited vertices adjacent to *v* are visited next. Then unvisited vertices adjacent to these vertices are visited and so on. A breadth first search beginning at vertex v_1 of the graph in figure 6.13(a) would first visit v_1 and then v_2 and v_3. Next vertices v_4, v_5, v_6 and v_7 will be visited and finally v_8. Algorithm BFS gives the details.

```
procedure BFS(v)

//A breadth first search of G is carried out beginning at vertex v.

All vertices visited are marked as VISlTED(i)= 1. The graph G

and array VISITED are global and VISITED is initialized to zero.//

VISITED (v) ☐ 1

initialize Q to be empty          //Q is a queue//

loop

for all vertices w adjacent to v do

if VISITED(w) = 0            //add w to queue//

then [call ADDQ(w,Q); VISITED(w)☐1]

//mark w as VISITED//

end

if Q is empty then return

call DELETEQ(v,Q)

forever

end BFS
```

☐

Figure 6.13 Graph G and Its Adjacency Lists.

Each vertex visited gets into the queue exactly once, so the **loop forever** is iverated at most n times. If an adjacency matrix is used, then the **for** loop takes $O(n)$ time for each vertex visited. The total time is, therefore, $O(n^2)$. In case adjacency lists are used the **for** loop has a total cost of $d_1 + \cdots + d_n = O(e)$ where

d_i = degree (v_i). Again, all vertices visited, together with all edges incident to them form a connected component of *G*.

We now look at two simple applications of graph traversal: (i) finding the components of a graph, and (ii) finding a spanning tree of a connected graph.

Connected Components

If *G* is an undirected graph, then one can determine whether or not it is connected by simply making a call to either DFS or BFS and then determining if there is any unvisited vertex. The time to do this is $O(n^2)$ if adjacency matrices are used and O(*e*) if adjacency lists are used. A more interesting problem is to determine all the connected components of a graph. These may be obtained by making repeated calls to either DFS(*v*) or BFS(*v*), with *v* a vertex not yet visited. This leads to algorithm COMP which determines all the connected components of *G*. The algorithm uses DFS. BFS may be used instead if desired.The computing time is not affected.

```
procedure COMP(G,n)

//determine the connected components of G. G has n ≥ 1 vertices.

VISITED is now a local array.//

for i ⬜ 1 to n do

VISITED(i) ⬜ 0      //initialize all vertices as unvisited//

end

for i ⬜ 1 to n do

if VISITED(i) = 0 then [call DFS(i);        //find a component//

output all newly visited vertices together

with all edges incident to them]

end

end COMP
```

If G is represented by its adjacency lists, then the total time taken by DFS is $O(e)$. The output can be completed in time $O(e)$ if DFS keeps a list of all newly visited vertices. Since the **for** loops take $O(n)$ time, the total time to generate all the connected components is $O(n + e)$.

By the definition of a connected component, there is a path between every pair of vertices in the component and there is no path in G from vertex v to w if v and w are in two different components. Hence, if A is the adjacency matrix of an undirected graph (i.e., A is symmetric) then its transitive closure A^+ may be determined in $O(n^2)$ time by first determining the connected components. $A^+(i,j) = 1$ iff there is a path from vertex i to j. For every pair of distinct vertices in the same component $A + (i,j) = 1$. On the diagonal $A + (i,i) = 1$ iff the component containing i has at least 2 vertices. We shall take a closer look at transitive closure in section 6.3.

Spanning Trees and Minimum Cost Spanning Trees

When the graph G is connected, a depth first or breadth first search starting at any vertex, visits all the vertices in G. In this case the edges of G are partitioned into two sets T (for tree edges) and B (for back edges), where T is the set of edges used or traversed during the search and B the set of remaining edges. The set T may be determined by inserting the statement $T \quad T \cup \{(v,w)\}$ in the **then** clauses of DFS and BFS. The edges in T form a tree which includes all the vertices of G. Any tree consisting solely of edges in G and including all vertices in G is called a *spanning tree*. Figure 6.14 shows a graph and some of its spanning trees. When either DFS or BFS are used the edges of T form a spanning tree. The spanning tree resulting from a call to DFS is known as a *depth first spanning tree*. When BFS is used, the resulting spanning tree is called a *breadth first spanning tree.*

Figure 6.14 A Complete Graph and Three of Its Spanning Trees.

Figure 6.15 shows the spanning trees resulting from a depth first and breadth first search starting at vertex v_1 in the graph of figure 6.13. If any of the edges (v,w) in B (the set of back edges) is introduced into the spanning tree T, then a cycle is formed. This cycle consists of the edge (v,w) and all the edges on the path from w to v in T. If the edge (8,7) is introduced into the DFS spanning tree of figure 6.15(a), then the resulting cycle is 8,7,3,6,8.

Spanning trees find application in obtaining an independent set of circuit equations for an electrical network. First, a spanning tree for the network is obtained. Then the edges in B (i.e., edges not in the spanning tree) are introduced one at a time. The introduction of each such edge results in a cycle. Kirchoff's second law is used on this cycle to obtain a circuit equation. The cycles obtained in this way are independent (i.e., none of these cycles can be obtained by taking a linear combination of the remaining cycles) as each contains an edge from B which is not contained in any other cycle. Hence, the circuit equations so obtained are also independent. In fact, it may be shown that the cycles obtained by

introducing the edges of *B* one at a time into the resulting spanning tree form a cycle basis and so all other cycles in the graph can be constructed by taking a linear combination of the cycles in the basis (see Harary in the references for further details).

☐

(a) DFS (1) Spanning Tree

☐

(b) BFS (1) Spanning Tree

Figure 6.15 DFS and BFS Spanning Trees for Graph of Figure 6.13.

It is not difficult to imagine other applications for spanning trees. One that is of interest arises from the property that a spanning tree is a minimal subgraph *G'* of *G* such that $V(G') = V(G)$ and *G'* is connected (by a minimal subgraph, we mean one with the fewest number of edges). Any connected graph with *n* vertices must have at least *n* - 1 edges and all connected graphs with *n* - 1 edges are trees. If the nodes of *G* represent cities and the edges represent possible communication links connecting 2 cities, then the minimum number of links needed to connect the *n* cities is *n* - 1. The spanning trees of *G* will represent all feasible choices. In any practical situation, however, the edges will have weights assigned to them. These weights might represent the cost of construction, the length of the link, etc. Given such a weighted graph one would then wish to select for construction a set of communication links that would connect all the cities and have minimum total cost or be of minimum total length. In either case the links selected will have to form a tree (assuming all weights are positive). In case this is not so, then the selection of links contains a cycle. Removal of any one of the links on this cycle will result in a link selection of less cost connecting all cities. We are, therefore, interested in finding a spanning tree of *G* with minimum cost. The cost of a spanning tree is the sum of the costs of the edges in that tree.

One approach to determining a minimum cost spanning tree of a graph has been given by Kruskal. In this approach a minimum cost spanning tree, *T*, is built edge by edge. Edges are considered for inclusion in *T* in nondecreasing order of their costs. An edge is included in *T* if it does not form a cycle with the edges already in *T*. Since *G* is connected and has $n > 0$ vertices, exactly *n* - 1 edges will be selected for inclusion in *T*. As an example, consider the graph of figure 6.16(a). The edges of this graph are considered for inclusion in the minimum cost spanning tree in the order (2,3), (2,4), (4,3), (2,6), (4,6), (1,2), (4,5), (1,5) and (5,6). This corresponds to the cost sequence 5, 6, 10, 11, 14, 16, 18, 19 and 33. The first two edges (2,3) and (2,4) are included in *T*. The next edge to be considered is (4,3). This edge, however, connects two vertices already connected in *T* and so it is rejected. The edge (2,6) is selected while (4,6) is rejected as the vertices 4 and 6 are already connected in *T* and the inclusion of (4,6) would result in a cycle. Finally, edges (1,2) and (4,5) are included. At this point, *T* has *n* - 1 edges and is a tree spanning *n* vertices. The spanning tree obtained (figure 6.16(b)) has cost 56. It is somewhat surprising that this straightforward approach should always result in a minimum spanning tree. We shall soon

prove that this is indeed the case. First, let us look into the details of the algorithm. For clarity, the Kruskal algorithm is written out more formally in figure 6.18. Initially, E is the set of all edges in G. The only functions we wish to perform on this set are: (i) determining an edge with minimum cost (line 3), and (ii) deleting that edge (line 4). Both these functions can be performed efficiently if the edges in E are maintained as a sorted sequential list. In Chapter 7 we shall see how to sort these edges into nondecreasing order in time $O(e \log e)$, where e is the number of edges in E. Actually, it is not essential to sort all the edges so long as the next edge for line 3 can be determined easily. It will be seen that the heap of Heapsort (section 7.6) is ideal for this and permits the next edge to be determined and deleted in $O(\log e)$ time. The construction of the heap itself takes $O(e)$ time.

Figure 6.16 Graph and A Spanning Tree of Minimum Cost.

Figure 6.17 Stages in Kruskal's Algorithm Leading to a Minimum Cost Spanning Tree.

```
1   T

2   while T contains less than n - 1 edges and E not empty do

3       choose an edge (v,w) from E of lowest cost;

4       delete (v,w) from E;

5       if (v,w) does not create a cycle in T

6           then add (v,w) to T

7           else discard (v,w)

8   end

9   if T contains fewer than n - 1 edges then print ('no spanning
tree')
```

Figure 6.18 Early Form of Minimum Spanning Tree Algorithm-Kruskal.

In order to be able to perform steps 5 and 6 efficiently, the vertices in G should be grouped together in such a way that one may easily determine if the vertices v and w are already connected by the earlier

selection of edges. In case they are, then the edge (v,w) is to be discarded. If they are not, then (v,w) is to be added to T. One possible grouping is to place all vertices in the same connected component of T into a set (all connected components of T will also be trees). Then, two vertices v,w are connected in T iff they are in the same set. For example, when the edge (4,3) is to be considered, the sets would be {1}, {2,3,4}, {5}, {6}. Vertices 4 and 3 are already in the same set and so the edge (4,3) is rejected. The next edge to be considered is (2,6). Since vertices 2 and 6 are in different sets, the edge is accepted. This edge connects the two components {2,3,4} and {6} together and so these two sets should be unioned to obtain the set representing the new component. Using the set representation of section 5.8 and the FIND and UNION algorithms of that section we can obtain an efficient implementation of lines 5 and 6. The computing time is, therefore, determined by the time for lines 3 and 4 which in the worst case is $O(e \log e)$. We leave the writing of the resulting algorithm as an exercise. Theorem 6.1 proves that the algorithm resulting from figure 6.18 does yield a minimum spanning tree of G. First, we shall obtain a result that will be useful in the proof of this theorem.

Definition: A *spanning forest* of a graph G = (V,E) is a collection of vertex disjoint trees $T_i = (V_i, E_i)$, $1 \le i \le k$ such that ☐ and $E_i \subseteq E(G)$, $1 \le i \le k$.

Lemma 6.1: Let $T_i = (V_i, E_i)$, $1 \le i \le k$, $k > 1$, *be a spanning forest for the connected undirected graph* G= (V,E). *Let* w *be a weighting function for* E(G) *and let* e = (u, v) *be an edge of minimum weight such that if* u $\in V_i$ *then* $v \notin V_i$. *Let* i = 1 *and* ☐. *There is a spanning tree for* G *which includes* E' $\cup$ {e} *and has minimum weight among all spanning trees for* G *that include* E'.

Proof: If the lemma is false, then there must be a spanning tree $T= (V,E'')$ for G such that E'' includes E' but not e and T has a weight less than the weight of the minimum spanning tree for G including $E' \cup$ {e}. *Since* T *is a spanning tree, it has a path from* u *to* v. *Consequently, the addition of* e *to* E'' *creates a unique cycle (exercise 22). Since* u $\in V_1$ *and* v $\notin V_1$, it follows that there is another edge $e' = (u',v')$ on this cycle such that $u' \in V_1$ *and* v' $\notin V_1$ (v' may be v). By assumption, $w(e) \le w(e')$. Deletion of the edge e' from $E'' \cup$ {e} *breaks this cycle and leaves behind a spanning tree* T' *that include* E' $\cup$ {e}. *But, since* w(e) $\le w(e')$, it follows that the weight of T' is no more than the weight of T. This contradicts the assumption on T. Hence, there is no such T and the lemma is proved.

Theorem 6.1: The algorithm described in figure 6.18 generates a minimum spanning tree.

Proof: The proof follows from Lemma 6.1 and the fact that the algorithm begins with a spanning forest with no edges and then examines the edges of G in nondecreasing order of weight.

6.3 SHORTEST PATHS AND TRANSITIVE CLOSURE

Graphs may be used to represent the highway structure of a state or country with vertices representing cities and edges representing sections of highway. The edges may then be assigned weights which might be either the distance between the two cities connected by the edge or the average time to drive along that section of highway. A motorist wishing to drive from city A to city B would be interested in answers to the following questions:

(i) Is there a path from A to B?

(ii) If there is more than one path from A to B, which is the shortest path?

The problems defined by (i) and (ii) above are special cases of the path problems we shall be studying in this section. The length of a path is now defined to be the sum of the weights of the edges on that path rather than the number of edges. The starting vertex of the path will be referred to as the *source* and the last vertex the *destination*. The graphs will be digraphs to allow for one way streets. Unless otherwise stated, we shall assume that all weights are positive.

Single Source All Destinations

In this problem we are given a directed graph $G = (V,E)$, a weighting function $w(e)$ for the edges of G and a source vertex v_o. The problem is to determine the shortest paths from v_o to all the remaining vertices of G. It is assumed that all the weights are positive. As an example, consider the directed graph of figure 6.19(a). The numbers on the edges are the weights. If v_o is the source vertex, then the shortest path from v_o to v_1 is $v_o\, v_2\, v_3\, v_1$. The length of this path is $10 + 15 + 20 = 45$. Even though there are three edges on this path, it is shorter than the path $v_o\, v_1$ which is of length 50. There is no path from v_o to v_5. Figure 6.19(b) lists the shortest paths from v_o to v_1, v_2, v_3 and v_4. The paths have been listed in nondecreasing order of path length. If we attempt to devise an algorithm which generates the shortest paths in this order, then we can make several observations. Let S denote the set of vertices (including v_o) to which the shortest paths have already been found. For w not in S, let DIST(w) be the length of the shortest path starting from v_o going through only those vertices which are in S and ending at w. We observe that:

Figure 6.19 Graph and Shortest Paths from v_0 to All Destinations.

(i) If the next shortest path is to vertex u, then the path begins at v_o, ends at u and goes through only those vertices which are in S. To prove this we must show that all of the intermediate vertices on the shortest path to u must be in S. Assume there is a vertex w on this path that is not in S. Then, the v_o to u path also contains a path from v_o to w which is of length less than the v_o to u path. By assumption the shortest paths are being generated in nondecreasing order of path length, and so the shorter path v_o to w

must already have been generated. Hence, there can be no intermediate vertex which is not in *S*.

(ii) The destination of the next path generated must be that vertex *u* which has the minimum distance, DIST(*u*), among all vertices not in *S*. This follows from the definition of DIST and observation (i). In case there are several vertices not in *S* with the same DIST, then any of these may be selected.

(iii) Having selected a vertex *u* as in (ii) and generated the shortest v_o to *u* path, vertex *u* becomes a member of *S*. At this point the length of the shortest paths starting at v_o, going through vertices only in *S* and ending at a vertex *w* not in *S* may decrease. I.e., the value of DIST(*w*) may change. If it does change, then it must be due to a shorter path starting at v_o going to *u* and then to *w*. The intermediate vertices on the v_o to *u* path and the *u* to *w* path must all be in *S*. Further, the v_o to *u* path must be the shortest such path, otherwise DIST(*w*) is not defined properly. Also, the *u* to *w* path can be chosen so as to not contain any intermediate vertices. Therefore, we may conclude that if DIST(*w*) is to change (i.e., decrease), then it is because of a path from v_o to *u* to *w* where the path from v_o to *u* is the shortest such path and the path from *u* to *w* is the edge <*u,w*>. *The length of this path is DIST(*u*) + length (<u,w>).*

The algorithm SHORTEST_PATH as first given by Dijkstra, makes use of these observations to determine the cost of the shortest paths from v_o to all other vertices in *G*. The actual generation of the paths is a minor extension of the algorithm and is left as an exercise. It is assumed that the *n* vertices of *G* are numbered *1* through *n*. The set *S* is maintained as a bit array with *S(i)* = 0 if vertex *i* is not in *S* and *S(i)* = 1 if it is. It is assumed that the graph itself is represented by its cost adjacency matrix with COST(*i, j*) being the weight of the edge <i,j>. *COST(i,j)* will be set to some large number, $+\infty$, in case the edge <i,j> is not in *E(G)*. For *i= j*, COST(*i,j*) may be set to any non-negative number without affecting the outcome of the algorithm.

Analysis of Algorithm SHORTEST PATH

From our earlier discussion, it is easy to see that the algorithm works. The time taken by the algorithm on a graph with *n* vertices is $O(n^2)$. To see this note that the **for** loop of line 1 takes $O(n)$ time. The **while** loop is executed *n* - 2 times. Each execution of this loop requires $O(n)$ time at line 6 to select the next vertex and again at lines 8-10 to update DIST. So the total time for the **while** loop is $O(n^2)$. In case a list *T* of vertices currently not in *S* is maintained, then the number of nodes on this list would at any time be *n* - num. This would speed up lines 6 and 8-10, but the asymptotic time would remain $O(n^2)$. This and other variations of the algorithm are explored in the exercises.

```
procedure SHORTEST-PATH (v,COST,DIST,n)

//DIST(j), 1 ≤ j ≤ n is set to the length of the shortest path

from vertex v to vertex j in a digraph G with n vertices. DIST(v)
```

is set to zero. G is represented by its cost adjacency matrix,

$COST(n,n)//$

declare S (1: n)

```
1      for i ⬜ 1 to n do        //initialize set S to empty//

2          S(i) ⬜ 0; DIST(i) ⬜ COST(v,i)

3      end

4      S(v) ⬜ 1; DIST(v) ⬜ 0; num ⬜ 2      //put vertex v in set
S//

5      while num < n do      //determine n - 1 paths from vertex v//

6          choose u: DIST(u) = min {DIST(w)}
S(w)=0

7          S(u) ⬜ 1; num ⬜ num + 1     //put vertex u in set S//

8          for all w with S(w) = 0 do   //update distances//

9              DIST(w) ⬜ min {DIST(w),DIST(u) + COST(u,w)}

10         end

11      end

12 end SHORTEST-PATH
```

Any shortest path algorithm must examine each edge in the graph at least once since any of the edges could be in a shortest path. Hence, the minimum possible time for such an algorithm would be $O(e)$. Since cost adjacency matrices were used to represent the graph, it takes $O(n^2)$ time just to determine which edges are in G and so any shortest path algorithm using this representation must take $O(n^2)$. For this representation then, algorithm SHORTEST PATH is optimal to within a constant factor. Even if a

change to adjacency lists is made, only the overall time for the **for** loop of lines 8-10 can be brought down to $O(e)$ (since the DIST can change only for those vertices adjacent from u). The total time for line 6 remains $O(n^2)$.

Example 6.1: Consider the 8 vertex digraph of figure 6.20(a) with cost adjacency matrix as in 6.20(b). The values of DIST and the vertices selected at each iteration of the **while** loop of line 5 for finding all the shortest paths from Boston are shown in table 6.21. Note that the algorithm terminates when only seven of the eight vertices are in S. By the definition of DIST, the distance of the last vertex, in this case

Figure 6.20(a)

Figure 6.20(b) Cost Adjacency Matrix for Figure 6.20(a). All Entries not Shown are $+\infty$.

Iteration	S	Selected	Vertex LA (1)	SF (2)	D (3)	C (4)	B (5)	NY (6)	M (7)	NO (8)
Initial		--	$+\infty$	$+\infty$	$+\infty$	1500	0	250	$+\infty$	$+\infty$
1	5	6	$+\infty$	$+\infty$	$+\infty$	1250	0	250	1150	1650
2	5,6	7	$+\infty$	$+\infty$	$+\infty$	1250	0	250	1150	1650
3	5,6,7	4	$+\infty$	$+\infty$	2450	1250	0	250	1150	1650
4	5,6,7,4	8	3350	$+\infty$	2450	1250	0	250	1150	1650
5	5,6,7,4,8	3	3350	3250	2450	1250	0	250	1150	1650

```
6        5,6,7,4,8,3  2          3350   3250   2450   1250   0   250
1150   1650

         5,6,7,4,8,3,2
```

Table 6.21 Action of SHORTEST_PATH

Los Angeles is correct as the shortest path from Boston to Los Angeles can go through only the remaining six vertices.

All Pairs Shortest Paths

The *all pairs shortest path problem* calls for finding the shortest paths between all pairs of vertices v_i, v_j, $i \neq j$. One possible solution to this is to apply the algorithm SHORTEST__ PATH n times, once with each vertex in $V(G)$ as the source. The total time taken would be $O(n^3)$. For the all pairs problem, we can obtain a conceptually simpler algorithm which will work even when some edges in G have negative weights so long as G has no cycles with negative length. The computing time of this algorithm will still be $O(n^3)$ though the constant factor will be smaller.

The graph G is represented by its cost adjacency matrix with $COST(i.i) = 0$ and $COST(i,j) = +\infty$ in case edge $<i,j>$, $i \neq j$ is not in G. Define $A^k(i,j)$ to be the cost of the shortest path from i to j going through no intermediate vertex of index greater than k. Then, $A^n(i,j)$ will be the cost of the shortest i to j path in G since G contains no vertex with index greater than n. $A^o(i,j)$ is just $COST(i,j)$ since the only i to j paths allowed can have no intermediate vertices on them. The basic idea in the all pairs algorithm is to successively generate the matrices $A^0, A^1, A^2, ..., A^n$. If we have already generated A^{k-1}, then we may generate A^k by realizing that for any pair of vertices i,j either (i) the shortest path from i to j going through no vertex with index greater than k does not go through the vertex with index k and so its cost is $A^{k-1}(i,j)$; or (ii) the shortest such path does go through vertex k. Such a path consists of a path from i to k and another one from k to j. These paths must be the shortest paths from i to k and from k to j going through no vertex with index greater than $k - 1$, and so their costs are $A^{k-1}(i,k)$ and $A^{k-1}(k,j)$. Note that this is true only if G has no cycle with negative length containing vertex k. If this is not true, then the shortest i to j path going through no vertices of index greater than k may make several cycles from k to k and thus have a length substantially less than $A^{k-1}(i,k) + A^{k-1}(k,j)$ (see example 6.2). Thus, we obtain the following formulas for $A^k(i,j)$:

$$A^k(i,j) = \min \{A^{k-1}(i,j), A^{k-1}(i,k) + A^{k-1}(k,j)\} \quad k \geq 1$$

and

$A^o(i,j) = \text{COST}(i,j).$

Example 6.2: Figure 6.22 shows a digraph together with its matrix A^0. For this graph $A^2(1,3) \neq$ min $\{A^1(1,3), A^1(1,2) + A^1(2,3)\} = 2$. Instead we see that $A^2(1,3) = -\infty$ as the length of the path

1,2,1,2,1,2, ...,1,2,3

can be made arbitrarily small. This is so because of the presence of the cycle 1 2 1 which has a length of - 1.

☐

Figure 6.22 Graph with Negative Cycle

The algorithm ALL__*COSTS* computes $A^n(i,j)$. The computation is done in place so the superscript on A is dropped. The reason this computation can be carried out in place is that $A^k(i,k) = A^{k-1}(i,k)$ *and* $A_k(k,j) = A_{k-1}(k,j)$ and so the in place computation does not alter the outcome.

```
procedure ALL_COSTS(COST,A,n)

//COST(n,n) is the cost adjacency matrix of a graph with n

vertices; A(i,j) is the cost of the shortest path between vertices

vᵢ,vⱼ. COST(i,i) = 0, 1 ≤ i ≤ n//

1  for i ☐ 1 to n do

2    for j ☐ 1 to n do

3      A (i,j) ☐ COST (i,j)   //copy COST into A//

4    end

5  end

6  for k ☐ 1 to n do      //for a path with highest vertex index k//

7    for i ☐ 1 to n do     //for all possible pairs of vertices//
```

```
8        for j ☐ 1 to n do

9            A (i,j) ☐ min {A (i,j),A(i,k) + A(k,j)}

10       end

11      end

12   end

13 end ALL_COSTS
```

This algorithm is especially easy to analyze because the looping is independent of the data in the matrix *A*.

The total time for procedure ALL_COSTS is $O(n^3)$. An exercise examines the extensions needed to actually obtain the (i,j) paths with these lengths. Some speed up can be obtained by noticing that the innermost **for** loop need be executed only when $A (i,k)$ and $A (k,j)$ are not equal to ∞.

Example 6.3: : Using the graph of figure 6.23(a) we obtain the cost matrix of figure 6.23(b). The initial *A* matrix, A^o plus its value after 3 iterations A^1, A^2, A^3 is given in figure 6.24.

☐

Figure 6.23 Directed Graph and Its Cost Matrix.

```
A⁰·· 1   2   3        A¹ 1   2   3

----------------------------

1    0   4   11        1    0    4   11

2    6   0   2         2    6    0    2

3    3   ∞   0         3    3    7    0

A² 1   2   3         A³ 1   2   3

----------------------------
```

```
1   0   4   6          1    0   4   6

2   6   0   2          2    5   0   2

3   3   7   0          3    3   7   0
```

Figure 6.24 Matrices A^k Produced by ALL_COSTS for the Digraph of Figure 6.23.

Transitive Closure

A problem related to the all pairs shortest path problem is that of determining for every pair of vertices i, j in G the existence of a path from i to j. Two cases are of interest, one when all path lengths (i.e., the number of edges on the path) are required to be positive and the other when path lengths are to be nonnegative. If A is the adjacency matrix of G, then the matrix A^+ having the property $A^+ (i,j) = 1$ if there is a path of length > 0 from i to j and 0 otherwise is called the *transitive closure* matrix of G. The matrix A^* with the property $A^* (i,j) = 1$ if there is a path of length ≥ 0 from i to j and 0 otherwise is the *reflexive transitive closure* matrix of G.

□

□

□

□

Figure 6.25 Graph G and Its Adjacency Matrix A, A$^+$ and A*

Figure 6.25 shows A^+ and A^* for a digraph. Clearly, the only difference between A^* and A^+ is in the terms on the diagonal. $A^+(i,i) = 1$ iff there a cycle of length > 1 containing vertex i while $A^*(i,i)$ is always one as there is a path of length 0 from i to i. If we use algorithm ALL_COSTS with COST$(i,j) = 1$ if $<i, j>$ is an edge in G and COST$(i,j) = + \infty$ if $<i,j>$ is not in G, then we can easily obtain A^+ from the final matrix A by letting $A^+ (i,j) = 1$ iff $A (i,j) < + \infty$. A^* can be obtained from A^+ by setting all diagonal elements equal 1. The total time is $O(n^3)$. Some simplification is achieved by slightly modifying the algorithm. In this modification the computation of line 9 of ALL_COSTS becomes $A(i,j)$ □ $A (i,j)$ **or** $(A (i,k)$ **and** $A(k,))$ and COST(i,j) is just the adjacency matrix of G. With this modification, A need only be a bit matrix and then the final matrix A will be A^+.

6.4 ACTIVITY NETWORKS, TOPOLOGICAL SORT

AND CRITICAL PATHS

Topological Sort

All but the simplest of projects can be subdivided into several subprojects called activities. The successful completion of these activities will result in the completion of the entire project. A student working towards a degree in Computer Science will have to complete several courses successfully. The project in this case is to complete the major, and the activities are the individual courses that have to be taken. Figure 6.26 lists the courses needed for a computer science major at a hypothetical university. Some of these courses may be taken independently of others while other courses have prerequisites and can be taken only if all their prerequisites have already been taken. The data structures course cannot be started until certain programming and math courses have been completed. Thus, prerequisites define precedence relations between the courses. The relationships defined may be more clearly represented using a directed graph in which the vertices represent courses and the directed edges represent prerequisites. This graph has an edge $\langle i,j \rangle$ iff course i is a prerequisite for course j.

Definition: A directed graph G in which the vertices represent tasks or activities and the edges represent precedence relations between tasks is an *activity on vertex network* or AOV-network.

Definition: Vertex i in an AOV network G is a *predecessor* of vertex j iff there is a directed path from vertex i to vertex j. i is an *immediate predecessor* of j iff $\langle i,j \rangle$ is an edge in G. If i is a predecessor of j, then j is a *successor* of i. If i is an immediate predecessor of j, then j is an *immediate successor* of i.

Figure 6.26(b) is the AOV-network corresponding to the courses of figure 6.26(a). C3, C4 and C10 are the immediate predecessors of C7. C2, C3 and C4 are the immediate successors of C1. C12 is a succesor of C4 but not an immediate successor. The precedence relation defined by the set of edges on the set of vertices is readily seen to be transitive. (Recall that a relation ☐ is transitive iff it is the case that for all triples i,j,k, i ☐ j and j ☐ $k \Rightarrow i\ k$.) In order for an AOV-network to represent a feasible project, the precedence relation should also be irreflexive.

```
Course Number        Course Name                    Prerequisites

    Cl    Introduction to Programming       None

    C2    umerical Analysis                 Cl, C14

    C3    ata Structures                    Cl, C14

    C4    ssembly Language                  Cl, C13
```

```
C5     utomata Theory                      C15

C6     rtificial Intelligence              C3

C7     omputer Graphics                    C3, C4, C10

C8     achine Arithmetic                   C4

C9     nalysis of Algorithms               C3

C10    Higher Level Languages              C3, C4

Cll    Compiler Writing                    C10

C12    Operating Systems                   Cl l

C13    Analytic Geometry and Calculus I    None

C14    Analytic Geometry and Calculus II   C13

C15    Linear Algebra                      C14
```

(a) Courses Needed for a Computer Science Degree at Some Hypothetical University

(b) AOV-Network Representing Courses as Vertices and Prerequisites as Edges

Figure 6.26 An Activity on Vertex Network

Definition: A relation ☐ is *irreflexive* on a set *S* if for no element *x* in *S* it is the case that *x*☐*x*. *A precedence relation which is both transitive and irreflexive is a* partial order.

If the precedence relation is not irreflexive, then there is an activity which is a predecessor of itself and so must be completed before it can be started. This is clearly impossible. When there are no inconsistencies of this type, the project is feasible. Given an AOV network one of our concerns would be to determine whether or not the precedence relation defined by its edges is irreflexive. This is identical to determining whether or not the network contains any directed cycles. A directed graph with no directed cycles is an *acyclic* graph. Our algorithm to test an AOV-network for feasibility will also generate a linear ordering, v_{i1}, v_{i2}, ...,v_{in}, of the vertices (activities). This linear ordering will have the property that if *i* is a predecessor of *j* in the network then *i* precedes *j* in the linear ordering. A linear

ordering with this property is called a *topological order*. For the network of figure 6.26(b) two of the possible topological orders are: Cl, C13, C4, C8, C14, C15, C5, C2, C3, C10, C7, C11, C12, C6, C9 and C13, C14, C15, C5, Cl, C4, C8, C2, C3, C10, C7, C6, C9, Cll, C12. If a student were taking just one course per term, then he would have to take them in topological order. If the AOV-network represented the different tasks involved in assembling an automobile, then these tasks would be carried out in topological order on an assembly line. The algorithm to sort the tasks into topological order is straightforward and proceeds by listing out a vertex in the network that has no predecessor. Then, this vertex together with all edges leading out from it are deleted from the network. These two steps are repeated until either all vertices have been listed or all remaining vertices in the network have predecessors and so none can be performed. In this case there is a cycle in the network and the project is infeasible. Figure 6.27 is a crude form of the algorithm.

```
input the AOV-network. Let n be the number of vertices.

1  for i ☐ 1 to n do     //output the vertices//

2     if every vertex has a predecessor

then [the network has a cycle and is infeasible. stop]

3     pick a vertex v which has no predecessors

4     output v

5     delete v and all edges leading out of v from the network

6  end
```

Figure 6.27 Design of a Topological Sorting Algorithm

Trying this out on the network of figure 6.28 we see that the first vertex to be picked in line 2 is v_1, as it is the only one with no predecessors. v_1 and the edges $<v_1,v_2>$, $<v_1,v_3>$ and $<v_1,v_4>$ are deleted. In the resulting network (figure 6.28(b)), v_2, v_3 and v_4 have no predecessor.

Figure 6.28 Action of Algorithm of Figure 6.27 on an AOV Network

Any of these can be the next vertex in the topological order. Assume that v_4 is picked. Deletion of $v4$ and the edges (v_4, v_6) and (v_4,v_5) results in the network of figure 6.28(c). Either v_2 or v_3 may next be

picked. Figure 6.28 shows the progress of the algorithm on the network. In order to obtain a complete algorithm that can be easily translated into a computer program, it is necessary to specify the data representation for the AOV-network. The choice of a data representation, as always, depends on the functions one wishes to perform. In this problem, the functions are: (i) decide whether a vertex has any predecessors (line 2), and (ii) delete a vertex together with all its incident edges. (i) is efficiently done if for each vertex a count of the number of its immediate predecessors is kept. (ii) is easily implemented if the network is represented by its adjacency lists. Then the deletion of all edges leading out of a vertex *v* can be carried out by decreasing the predecessor count of all vertices on its adjacency list. Whenever the count of a vertex drops to zero, that vertex can be placed onto a list of vertices with a zero count. Then the selection in line 3 just requires removal of a vertex from this list. Filling in these details into the algorithm of figure 6.27, we obtain the SPARKS program TOPOLOGICAL__ORDER.

The algorithm assumes that the network is represented by adjacency lists. The headnodes of these lists contain two fields: COUNT and LINK. The COUNT field contains the in-degree of that vertex and LINK is a pointer to the first node on the adjacency list. Each list node has 2 fields: VERTEX and LINK. COUNT fields can be easily set up at the time of input. When edge $<i,j>$ is input, the count of vertex *j* is incremented by 1. The list of vertices with zero count is maintained as a stack. A queue could have been used but a stack is slightly simpler. The stack is linked through the COUNT field of the headnodes since this field is of no use after the COUNT has become zero. Figure 6.29(a) shows the input to the algorithm in the case of the network of figure 6.28(a).

☐

Figure 6.29 Input for Algorithm TOPOLOGICAL__ORDER

```
procedure TOPOLOGICAL__ORDER (COUNT, VERTEX, LINK, n)

//the n vertices of an AOV-network are listed in topological order.

The network is represented as a set of adjacency lists with

COUNT (i) = the in-degree of vertex i//

1   top ☐ 0      //initialize stack//

2   for i ☐ 1 to n do      //create a linked stack of vertices with

no predecessors//

3       if COUNT(i) = 0 then [COUNT(i) ☐ top; top ☐ i]
```

```
4      end

5      for i □ 1 to n do      //print the vertices in topological order//

6        if top = 0 then [print ('network has a cycle'); stop]

7        j □ top; top □ COUNT (top); print (j)  //unstack a vertex//

8        ptr □ LINK (j)

9      while ptr ≠ 0 do

//decrease the count of successor vertices of j//

10       k □ VERTEX(ptr)              //k is a successor of j//

11       COUNT(k) □ COUNT(k) - 1                    //decrease count//

12       if COUNT(k) = 0   //add vertex k to stack//

13          then [COUNT(k) □ top; top □ k]

14       ptr □ LINK(ptr)

15      end

16   end

17  end TOPOLOGICAL_ORDER
```

As a result of a judicious choice of data structures the algorithm is very efficient. For a network with n vertices and e edges, the loop of lines 2-4 takes $O(n)$ time; Lines 6-8 take $O(n)$ time over the entire algorithm; the **while** loop takes time $O(d_i)$ for each vertex i, where di is the out-degree of vertex i. Since this loop is encountered once for each vertex output, the total time for this part of the algorithm is □. Hence, the asymptotic computing time of the algorithm is $O(e + n)$. It is linear in the size of the problem!

Critical Paths

An activity network closely related to the AOV-network is the activity on edge or AOE network. The tasks to be performed on a project are represented by directed edges. Vertices in the network represent events. Events signal the completion of certain activities. Activities represented by edges leaving a vertex cannot be started until the event at that vertex has occurred. An event occurs only when all activities entering it have been completed. Figure 6.30(a) is an AOE network for a hypothetical project with 11 tasks or activities a_1, ...,a_{11}. There are 9 events v_1, v_2, ..., v_9. The events v_1 and v_9 may be interpreted as "start project" and "finish project" respectively. Figure 6.30(b) gives interpretations for some of the 9 events. The number associated with each activity is the time needed to perform that activity. Thus, activity a_1 requires 6 days while a_{11} requires 4 days. Usually, these times are only estimates. Activities a_1, a_2 and a_3 may be carried out concurrently after the start of the project. a_4, a_5 and a_6 cannot be started until events v_2, v_3 and v_4, respectively, occur. a_7 and a_8 can be carried out concurrently after the occurrence of event v_5 (i.e., after a_4 and a_5 have been completed). In case additional ordering constraints are to be put on the activities, dummy activities whose time is zero may be introduced. Thus, if we desire that activities a_7 and a_8 not start until both events v_5 and v_6 have occurred, a dummy activity a_{12} represented by an edge $<v_6, v_5>$ may be introduced. Activity networks of the AOE type have proved very useful in the performance evaluation of several types of projects. This evaluation includes determining such facts about the project as: what is the least amount of time in which the project may be completed (assuming there are no cycles in the network); which activities should be speeded up in order to reduce completion time; etc. Several sophisticated techniques such as PERT (performance evaluation and review technique), CPM (critical path method), RAMPS (resource allocation and multi-project scheduling) have been developed to evaluate network models of projects. CPM was originally developed in connection with maintenance and construction projects. Figure 6.31 shows a network used by the Perinia Corporation of Boston in 1961 to model the construction of a floor in a multistory building. PERT was originally designed for use in the development of the Polaris Missile system.

Figure 6.30

(a) AOE Network. Activity Graph of a Hypothetical Project.

```
event              interpretation


-----              --------------


 v₁     start of project


 v₂     completion of activity a₁
```

v_5 completion of activities a_4 and a_5

v_8 completion of activities a_8 and a_9

v_9 completion of project

(b) Interpretation for Some of the Events in the Activity Graph of Figure 6.30(a).

Since the activities in an AOE network can be carried out in parallel the minimum time to complete the project is the length of the longest path from the start vertex to the finish vertex (the length of a path is the sum of the times of activities on this path). A path of longest length is a *critical path*. The path v_1, v_2, v_5, v_7, v_9 is a critical path in the network of figure 6.30(a). The length of this critical path is 18. A network may have more than one critical path (the path v_1, v_2, v_5, v_8, v_9 is also critical). The *earliest time* an event v_i can occur is the length of the longest path from the start vertex v_1 to the vertex v_i. The earliest time event v_5 can occur is 7. The earlist time an event can occur determines the *earliest start time* for all activities represented by edges leaving that vertex. Denote this time by $e(i)$ for activity a_i. For example $e(7) = e(8) = 7$. For every activity a_i we may also define the *latest time, $l(i)$*, an activity may start without increasing the project duration (i.e., length of the longest path from start to finish). In figure 6.30(a) we have $e(6) = 5$ and $l(6) = 8$, $e(8) = 7$ and $l(8) = 7$. All activities for which $e(i) = l(i)$ are called *critical activities*. The difference $l(i) - e(i)$ is a measure of the criticality of an activity. It gives the time by which an activity may be delayed or slowed without increasing the total time needed to finish the project. If activity a_6 is slowed down to take 2 extra days, this will not affect the project finish time. Clearly, all activities on a critical path are critical and speeding noncritical activities will not reduce the project duration.

The purpose of critical path analysis is to identify critical activities so that resources may be concentrated on these activities in an attempt to reduce project finish time. Speeding a critical activity will not result in a reduced project length unless that activity is on all critical paths. In figure 6.30(a) the activity a_{11} is critical but speeding it up so that it takes only three days instead of four does not reduce the finish time to seventeen days. This is so because there is another critical path v_1, v_2, v_5, v_7, v_9 that does not contain this activity. The activities a_1 and a_4 are on all critical paths. Speeding a_1 by two days reduces the critical path length to sixteen days. Critical path methods have proved very valuable in evaluating project performance and identifying bottlenecks. In fact, it is estimated that without such analysis the Polaris missile project would have taken seven instead of the five years it actually took.

Figure 6.31 AOE network for the construction of a typical floor in a multistory building

[Redrawn from Engineering News-Record (McGraw-Hill Book Company, Inc., January 26, 1961).]

Critical path analysis can also be carried out with AOV-networks. The length of a path would now be the sum of the activity times of the vertices on that path. For each activity or vertex, we could analogously define the quantities $e(i)$ and $l(i)$. Since the activity times are only estimates, it is necessary to re-evaluate the project during several stages of its completion as more accurate estimates of activity times become available. These changes in activity times could make previously non-critical activities critical and vice versa. Before ending our discussion on activity networks, let us design an algorithm to evaluate $e(i)$ and $l(i)$ for all activities in an AOE-network. Once these quantities are known, then the critical activities may be easily identified. Deleting all noncritical activities from the AOE network, all critical paths may be found by just generating all paths from the start to finish vertex (all such paths will include only critical activities and so must be critical, and since no noncritical activity can be on a critical path, the network with noncritical activities removed contains all critical paths present in the original network).

In obtaining the $e(i)$ and $l(i)$ for the activities of an AOE network, it is easier to first obtain the earliest event occurrence time, $ee(j)$, and latest event occurrence time, $le(j)$, for all events, j, in the network. Then if activity a_i is represented by edge $<k,l>$, we can compute $e(i)$ and $l(i)$ from the formulas:

$e(i) = ee(k)$

and

$l(i) = le(l)$ - duration of activity a_i

(6.1)

The times $ee(j)$ and $le(j)$ are computed in two stages: a forward stage and a backward stage. During the forward stage we start with $ee(1) = 0$ and compute the remaining early start times, using the formula

☐

(6.2)

where $P(j)$ is the set of all vertices adjacent to vertex j. In case this computation is carried out in topological order, the early start times of all predecessors of j would have been computed prior to the computation of $ee(j)$. The algorithm to do this is obtained easily from algorithm TOPOLOGICAL_ORDER by inserting the step

$ee(k)$ ☐ $max \{ee(k), ee(j) + \text{DUR} (ptr)\}$

between lines 11 and 12. It is assumed that the array *ee* is initialized to zero and that DUR is another field in the adjacency list nodes which contains the activity duration. This modification results in the evaluation of equation (6.2) in parallel with the generation of a topological order. *ee*(*j*) is updated each time the *ee*(*i*) of one of its predecessors is known (i.e., when *i* is ready for output). The step **print** (*j*) of line 7 may be omitted. To illustrate the working of the modified TOPOLOGICAL__ORDER algorithm let us try it out on the network of figure 6.30(a). The adjacency lists for the network are shown in figure 6.32(a). The order of nodes on these lists determines the order in which vertices will be considered by the algorithm. At the outset the early start time for all vertices is 0, and the start vertex is the only one in the stack. When the adjacency list for this vertex is processed, the early start time of all vertices adjacent from v_1 is updated. Since vertices 2, 3 and 4 are now in the stack, all their predecessors have been processed and equation (6.2) evaluated for these three vertices. *ee*(6) is the next one determined. When vertex v_6 is being processed, *ee*(8) is updated to 11. This, however, is not the true value for *ee*(8) since equation (6.2) has not been evaluated over all predecessors of v_8 (v_5 has not yet been considered). This does not matter since v_8 cannot get stacked until all its predecessors have been processed. *ee*(5) is next updated to 5 and finally to 7. At this point *ee*(5) has been determined as all the predecessors of v_5 have been examined. The values of *ee*(7) and *ee*(8) are next obtained. *ee*(9) is ultimately determined to be 18, the length of a critical path. One may readily verify that when a vertex is put into the stack its early time has been correctly computed. The insertion of the new statement does not change the asymptotic computing time; it remains $O(e + n)$.

☐

(a) Adjacency Lists for Figure 6.30(a)

ee	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	stack
initial	0	0	0	0	0	0	0	0	0	1
output v_1	0	6	4	5	0	0	0	0	0	4
										3
										2
output v_4	0	6	4	5	0	7	0	0	0	6
										3
										2

output v_6	0	6	4	5	0	7	0	11	0	3
										2
output v_3	0	6	4	5	5	7	0	11	0	2
output v_2	0	6	4	5	7	7	0	11	0	5
output v_5	0	6	4	5	7	7	16	14	0	8
										7
output v_8	0	6	4	5	7	7	16	14	16	7
output v_7	0	6	4	5	7	7	16	14	18	9
output v_9										

(b) Computation of ee

Figure 6.32 Action of Modified Topological Order

In the backward stage the values of $le(i)$ are computed using a procedure analogous to that used in the forward stage. We start with $le(n) = ee(n)$ and use the equation

$\boxed{}$

(6.3)

where $S(j)$ is the set of vertices adjacent from vertex j. The initial values for $le(i)$ may be set to $ee(n)$. Basically, equation (6.3) says that if $<j,i>$ is an activity and the latest start time for event i is $le(i)$, then event j must occur no later than $le(i)$ - duration of $<j,i>$. Before $le(j)$ can be computed for some event j, the latest event time for all successor events (i.e., events adjacent from j) must be computed. These times can be obtained in a manner identical to the computation of the early times by using inverse adjacency lists and inserting the step $le(k) \boxed{}$ min $\{le(k), le(j) - DUR(ptr)\}$ at the same place as before in algorithm TOPOLOGICAL_ORDER. The COUNT field of a headnode will initially be the out-degree of the vertex. Figure 6.33 describes the process on the network of figure 6.30(a). In case the forward stage has already been carried out and a topological ordering of the vertices obtained, then the values of $le(i)$ can be computed directly, using equation (6.3), by performing the computations in the reverse topological order. The topological order generated in figure 6.32(b) is $v_1, v_4, v_6, v_3, v_2, v_5, v_8, v_7, v_9,$. We may

compute the values of $le(i)$ in the order 9,7,8,5,2,3,6,4,1 as all successors of an event precede that event in this order. In practice, one would usually compute both ee and le. The procedure would then be to compute ee first using algorithm TOPOLOGICAL_ORDER modified as discussed for the forward stage and to then compute le directly from equation (6.3) in reverse topological order.

Using the values of ee (figure 6.32) and of le (figure 6.33) and equation 6.1) we may compute the early and late times $e(i)$ and $l(i)$ and the degree of criticality of each task. Figure 6.34 gives the values. The critical activities are $a_1, a_4, a_7, a_8, a_{10}$ and a_{11}. Deleting all noncritical activities from the network we get the directed graph of figure 6.35. All paths from v_1 to v_9 in this graph are critical paths and there are no critical paths in the orginal network that are not paths in the graph of figure 6.35 .

As a final remark on activity networks we note that the algorithm TOPOLOGICAL-ORDER detects only directed cycles in the network. There may be other flaws, such as vertices not reachable from the start vertex (figure 6.36). When a critical path analysis is carried out on such networks, there will be several vertices with $ee(i) = 0$. Since all activity times are assumed > 0 only the start vertex can have $ee(i) = 0$. Hence, critical path analysis can also be used to detect this kind of fault in project planning.

☐

(a) Inverted Adjacency Lists for AOE-Network of Figure 6.30(a)

le	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	stack
initial	18	18	18	18	18	18	18	18	18	9
output v_9	18	18	18	18	18	18	16	14	18	8
										7
output v_8	18	18	18	18	7	10	16	14	18	6
										7
output v_6	18	18	18	18	7	10	16	14	18	4
										7
output v_4	3	18	18	8	7	10	16	14	18	7
output v_7	3	18	18	8	7	10	16	14	18	5

```
output v₅   3     6     6     8     7     10    16    14    18    3

                                                              2

output v₃   2     6     6     8     7     10    16    14    18    2

output v₂   0     6     6     8     7     10    16    14    18    1
```

(b) Computation of TOPOLOGICAL-ORDER Modified to Compute Latest Event Times.

$le(9) = ee(9) = 18$

$le(7) = \min \{le(9) - 2\} = 16$

$le(8) = \min \{le(9) - 4\} = 14$

$le(5) = \min \{le(7) - 9, \ le(8) - 7\} = 7$

$le(2) = \min \{le(5) - 1\} = 6$

$le(3) = \min \{le(5) - 1\} = 6$

$le(6) = \min \{le(8) - 4\} = 10$

$le(4) = \min \{le(6) - 2\} = 8$

$le(1) = \min \{le(2) - 6, \ le(3) - 4, \ le(4) - 5\} = 0$

c) Computation of le Directly from Equation (6.3) Using a Reverse Topological Order.

Figure 6.33

```
activity   e     l     l - e

   a₁      0     0     0

   a₂      0     2     2

   a₃      0     3     3
```

a_4	6	6	0
a_5	4	6	2
a_6	5	8	3
a_7	7	7	0
a_8	7	7	0
a_9	7	10	3
a_{10}	16	16	0
a_{11}	14	14	0

Figure 6.34 Early, Late and Criticality Values

Figure 6.35 Graph Obtained After Deleting All Noncritical Activities.

Figure 6.36 AOE-Network with Some Non-Reachable Activities.

6.5 ENUMERATING ALL PATHS

In section 6.3 we looked at the problem of finding a shortest path between two vertices. In this section we will be concerned with listing all possible simple paths between two vertices in order of nondecreasing path length. Such a listing could be of use, for example, in a situation where we are interested in obtaining the shortest path that satisfies some complex set of constraints. One possible solution to such a problem would be generate in nondecreasing order of path length all paths between the two vertices. Each path generated could be tested against the other constraints and the first path satisfying all these constraints would be the path we were looking for.

Let $G = (V,E)$ be a digraph with n vertices. Let v_1 be the source vertex and v_n the destination vertex. We shall assume that every edge has a positive cost. Let $p_1 = r(0), r(1), ..., r(k)$ be a shortest path from v_1 to

v_n. I.e., p_1 starts at $v_{r(0)} = v_1$, goes to $v_{r(1)}$ and then to $v_r(2)$, ...,$v_n = v_{r(k)}$. If P is the set of all simple v_1 to v_n paths in G, then it is easy to see that every path in $P - \{p_1\}$ differs from p_1 in exactly one of the following k ways:

(1): It contains the edges $(r(1), r(2))$, ..., $(r(k - 1), r(k))$ but not $(r(0), r(1))$

(2): It contains the edges $(r(2), r(3))$, ..., $(r(k - 1), r(k))$ but not $(r(1), r(2))$

:

:

(k): It does not contain the edge $(r(k - 1), r(k))$.

More compactly, for every path p in $P - \{p_1\}$ there is exactly one j, $1 \leq j \leq k$ such that p contains the edges

$(r(j), r(j + 1))$, ...,$(r(k - 1), r(k))$ but not $(r(j - 1), r(j))$.

The set of paths $P - \{p_1\}$ may be partitioned into k disjoint sets $P^{(1)}$, ..., $P^{(k)}$ with set $P^{(j)}$ containing all paths in $P - \{P_1\}$ satisfying condition j above, $1 \leq j \leq k$.

Let $p^{(j)}$ be a shortest path in $P^{(j)}$ and let q be the shortest path from v_1 to $v_{r(j)}$ in the digraph obtained by deleting from G the edges $(r(j - 1),r(j)),(r(j),r(j + 1))$, ...,$(r(k - 1),r(k))$. Then one readily obtains $p^{(j)} = q$, $r(j)$, $r(j + 1)$, ...,$r(k) = n$. Let $p^{(l)}$ have the minimum length among $p^{(1)}$, ...,$p(k)$. Then, $p^{(l)}$ also has the least length amongst all paths in $P - \{p_1\}$ and hence must be a second shortest path. The set $P^{(l)} - \{p^{(l)}\}$ may now be partitioned into disjoint subsets using a criterion identical to that used to partition $P - \{p_1\}$. If $p^{(l)}$ has k' edges, then this partitioning results in k' disjoint subsets. We next determine the shortest paths in each of these k' subsets. Consider the set Q which is the union of these k' shortest paths and the paths $p^{(1)}$, ...,$p^{(l - 1)}$, ...,$p^{(k)}$. The path with minimum length in Q is a third shortest path p_3. The corresponding set may be further partitioned. In this way we may successively generate the v_1 to v_n paths in nondecreasing order of path length.

At this point, an example would be instructive. Figure 6.38 shows the generation of the v_1 to v_6 paths of the graph of figure 6.37.

A very informal version of the algorithm appears as the procedure M_SHORTEST.

```
procedure M_SHORTEST (M)
```

//Given a digraph G with positive edge weights this procedure outputs

the M shortest paths from v_1 to v_n. Q contains tuples of the

form (p,C) where p is a shortest path in G satisfying constraints

C. These constraints either require certain edges to be included

or excluded from the path//

1   Q ☐ $\{(shortest\ v_1\ to\ v_n\ path,\ \Phi)\}$

2   **for** i ☐ 1 **to** M **do**   //generate M shortest paths//

3   *let (p,C) be the tuple in Q such that path p is of minimal length.*

//p is the i'th shortest path//

4   **print** *path p;* delete path p from Q

5   *determine the shortest paths in G under the constraints C and*

the additional constraints imposed by the partitioning described

in the text

6   *add these shortest paths together with their constraints to Q*

7   **end**

8 **end** *M_SHORTEST*

Since the number of v_1 to v_n paths in a digraph with n vertices ranges from 0 to $(n - 1)!$, a worst case analysis of the computing time of M_SHORTEST is not very meaningful. Instead, we shall determine the time taken to generate the first m shortest paths. Line 5 of the **for** loop may require determining $n - 1$ shortest paths in a n vertex graph. Using a modified version of algorithm SHORTEST_PATH this would

take $O(n^3)$ for each iteration of the **for** loop. The total contribution of lkne 5 is therefore $O(mn^3)$. In the next chapter, when we study heap sort, we shall see that it is possible to maintain Q as a heap with the result that the total contribution of lines 3 and 6 is less than $O(mn^3)$. The total time to generate the m shortest paths is, therefore, $O(mn^3)$. The space needed is determined by the number of tuples in Q. Since at most n - 1 shortest paths are generated at each iteration of line 5, at most $O(mn)$ tuples get onto Q. Each path has at most $O(n)$ edges and the additional constraints take less space than this. Hence, the total space requirements are $O(mn^2)$

Figure 6.37 A Graph

```
Shortest
```

Path	Cost	Included Edges	Excluded Edges	New Path
------	----	-------------	-------------	--------
$v_1v_3v_5v_6$	8	none	none	
		none	(v_5v_6)	$v_1v_3v_4v_6 = 9$
		(v_5v_6)	(v_3v_5)	$v_1v_2v_5v_6 = 12$
		$(v_3v_5)(v_5v_6)$	(v_1v_3)	$v_1v_2v_3v_5v_6 = 14$
$v_1v_3v_4v_6$	9	none	(v_5v_6)	
		none	$(v_4v_6)(v_5v_6)$	∞
		(v_4v_6)	$(v_3v_4)(v_5v_6)$	$v_1v_2v_4v_6 = 13$
		$(v_3v_4)(v_4v_6)$	$(v_1v_3)(v_5v_6)$	$v_1v_2v_3v_4v_6 = 15$
$v_1v_2v_5v_6$	12	(v_5v_6)	(v_3v_5)	
		(v_5v_6)	$(v_2v_5)(v_3v_5)$	$v_1v_3v_4v_5v_6 = 16$
		$(v_2v_5)(v_5v_6)$	$(v_1v_2)(v_3v_5)$	∞

$v_1v_2v_4v_6$ 13 (v_4v_6) $(v_3v_4)(v_5v_6)$

            (v_4v_6) $(v_2v_4)(v_3v_4)(v_5v_6)$ ∞

            $(v_2v_4)(v_4v_6)$ " $(v_1v_2)(v_3v_4)(v_5v_6)$ ∞

$v_1v_2v_3v_5v_6$ 14 $(v_3v_5)(v_5v_6)$ (v_1v_3)

            $(v_3v_5)(v_5v_6)$ $(v_2v_3)(v_1v_3)$ ∞

            $(v_2v_3)(v_3v_5)(v_5v_6)$ $(v_1v_2)(v_5v_6)$

$v_1v_2v_3v_4v_6$ 15 $(v_3v_4)(v_4v_6)$ $(v_1v_3)(v_5v_6)$

            $(v_3v_4)(v_4v_6)$ $(v_2v_3)(v_1v_3)(v_5v_6)$ ∞

            $(v_2v_3)(v_3v_5)(v_5v_6)$ $(v_1v_2)(v_1v_3)(v_5v_6)$ ∞

$v_1v_3v_4v_5v_6$ 16 (v_5v_6) $(v_2v_5)(v_3v_5)$

            (v_5v_6) $(v_4v_5)(v_2v_5)(v_3v_5)$ ∞

            $(v_4v_5)(v_5v_6)$ $(v_3v_4)(v_2v_5)(v_3v_5)$ $v_1v_2v_4v_5v_6 = 20$

            $(v_3v_4)(v_4v_5)(v_5v_6)$ $(v_1v_3)0(v_2v_5)(v_3v_5)$ $v_1v_2v_3v_4v_5v_6 = 22$

Figure 6.38 Action of M_SHORTEST

REFERENCES

Eulers original paper on the Koenigsberg bridge problem makes interesting reading. This paper has been reprinted in:

"Leonhard Euler and the Koenigsberg Bridges," *Scientific American*, vol. 189, no. 1, July 1953, pp. 66-70.

Good general references for this chapter are:

Graph Theory by F. Harary, Addison-Wesley, Reading, Massachusetts, 1972.

The theory of graphs and its applications by C. Berge, John Wiley, 1962.

Further algorithms on graphs may be found in:

The design and analysis of computer algorithms by A. Aho, J. Hopcroft and J. Ullman, Addison-Wesley, Reading, Massachusetts, 1974.

Graph theory with applications to engineering and computer science by N. Deo, Prentice-Hall, Englewood Cliffs, New Jersey, 1974.

Combinatorial Optimization by E. Lawler, Holt, Reinhart and Winston, 1976.

"Depth-first search and linear graph algorithms" by R. Tarjan, *SIAM Journal on Computing*, vol. 1, no. 2, 1972, pp. 146-159.

Flows in Networks by L. Ford and D. Fulkerson, Princeton University Press, 1962.

Integer Programming and Network Flows by T. C. Hu, Addison-Wesley, Reading, Massachusetts, 1970.

For more on activity networks and critical path analysis see:

Project Management with CPM and PERT by Moder and C. Phillips, Van Nostran Reinhold Co., 1970.

EXERCISES

1. Does the multigraph below have an Eulerian walk? If so, find one.

2. For the digraph at the top of page 329 obtain:

i) the in degree and out degree of each vertex;

ii) its adjacency matrix;

iii) its adjacency list representation;

iv) its adjacency multilist representation;

v) its strongly connected components

3. Devise a suitable representation for graphs so they can be stored on punched cards. Write an algorithm which reads in such a graph and creates its adjacency matrix. Write another algorithm which creates the adjacency lists from those input cards.

☐

4. Draw the complete undirected graphs on one, two, three, four and five vertices. Prove that the number of edges in a *n* vertex complete graph is $n(n-1)/2$.

5. Is the directed graph below strongly connected? List all the simple paths.

☐

6. Show how the graph above would look if represented by its adjacency matrix, adjacency lists, adjacency multilist.

7. For an undirected graph *G* with *n* vertices and *e* edges show that ☐ where d_i = degree of vertex *i*.

8. (a) Let *G* be a connected undirected graph on *n* vertices. Show that *G* must have at least *n* - 1 edges and that all connected undirected graphs with *n* - 1 edges are trees.

(b) What is the minimum number of edges in a strongly connected digraph on *n* vertices? What shape do such digraphs have?

9. For an undirected graph *G* with *n* vertices prove that the following are equivalent:

(a) *G* is a tree;

(b) *G* is connected, but if any edge is removed the resulting graph is not connected;

(c) For any distinct vertices $u \in V(G)$ *and* $v \in V(G)$ there is exactly one simple path from *u* to *v*;

(d) *G* contains no cycles and has *n* - 1 edges;

(e) *G* is connected and has *n* - 1 edges.

10. A *bipartite graph G = (V,E)* is an undirected graph whose vertices can be partitioned into two

disjoint sets V_1 and $V_2 = V - V_1$ with the properties (i) no two vertices in V_1 are adjacent in G and (ii) no two vertices in V_2 are adjacent in G. The graph G_4 of figure 6.5 is bipartite. A possible partitioning of V is: $V_1 = \{1,4.5,7\}$ and $V_2 = \{2,3,6,8\}$. Write an algorithm to determine whether a graph G is bipartite. In case G is bipartite your algorithm should obtain a partitioning of the vertices into two disjoint sets V_1 and V_2 satisfying properties (i) and (ii) above. Show that if G is represented by its adjacency lists, then this algorithm can be made to work in time $O(n + e)$ where $n = |V|$ and $e = |E|$.

11. Show that every tree is a bipartite graph.

12. Prove that a graph G is bipartite iff it contains no cycles of odd length.

13. The following algorithm was obtained by Stephen Bernard to find an Eulerian circuit in an undirected graph in case there was such a circuit.

procedure *EULER (v)*

1   *path* $\square$ *{Φ}*

2   **for all** *vertices w adjacent to v* **and** *edge (v,w) not yet used* **do**

3   *mark edge (v,w) as used*

4   *path* $\square$ *{(v,w)}* ⌣ *EULER(w)* ⌣ *path*

5   **end**

6   **return** *(path)*

7 **end** *EULER*

a) Show that if G is represented by its adjacency multilists and *path* by a linked list, then algorithm EULER works in time $O(n + e)$.

b) Prove by induction on the number of edges in G that the above algorithm does obtain an Euler circuit for all graphs G having such a circuit. The initial call to Euler can be made with any vertex v.

c) At termination, what has to be done to determine whether or not G has an Euler circuit?

14. Apply depth first and breadth first search to the complete graph on four vertices. List the vertices in

the order they would be visited.

15. Show how to modify algorithm DFS as it is used in COMP to produce a list of all newly visited vertices.

16. Prove that when algorithm DFS is applied to a connected graph the edges of *T* form a tree.

17. Prove that when algorithm BFS is applied to a connected graph the edges of *T* form a tree.

18. Show that $A^+ = A^* \times A$ where matrix multiplication of the two matrices is defined as $\Box$. *V* is the logical *or* operation and ^ is the logical *and* operation.

19. Obtain the matrices A^+ and A^* for the digraph of exercise 5.

20. Another way to represent a graph is by its incidence matrix, INC. There is one row for each vertex and one column for each edge. Then INC $(i,j) = 1$ if edge *j* is incident to vertex *i*. The incidence matrix for the graph of figure 6.13(a) is:

$\Box$

The edges of figure 6.13(a) have been numbered from left to right, top to bottom. Rewrite algorithm DFS so it works on a graph represented by its incidence matrix.

21. If ADJ is the adjacency matrix of a graph $G = (V,E)$ and INC is the incidence matrix, under what conditions will ADJ = INC x INC^T - I where INC^T is the transpose of matrix INC. Matrix multiplication is defined as in exercise 18. I is the identity matrix.

22. Show that if *T* is a spanning tree for the undirected graph *G*, then the addition of an edge *e*, *e* $\notin$ *E(T)* and *e* $\in$ *E(G)*, to *T* creates a unique cycle .

23. By considering the complete graph with *n* vertices, show that the number of spanning trees is at least $2^{n-1} - 1$

24. The *radius* of a tree is the maximum distance from the root to a leaf. Given a connected, undirected graph write an algorithm for finding a spanning tree of minimum radius. (Hint: use breadth first search.) Prove that your algorithm is correct.

25. The *diameter* of a tree is the maximum distance between any two vertices. Given a connected, undirected graph write all algorithm for finding a spanning tree of minimum diameter. Prove the correctness of your algorithm.

26. Write out Kruskal's minimum spanning tree algorithm (figure 6.18) as a complete SPARKS program. You may use as subroutines the algorithms UNION and FIND of Chapter 5. Use algorithm SORT to sort the edges into nondecreasing order by weight.

27. Using the idea of algorithm SHORTEST_PATH, give an algorithm for finding a minimum spanning tree whose worst case time is $O(n^2)$.

28. Use algorithm SHORTEST_PATH to obtain in nondecreasing order the lengths of the shortest paths from vertex 1 to all remaining vertices in the digraph:

29. Rewrite algorithm SHORTEST_PATH under the following assumptions:

(i) G is represented by its adjacency lists. The head nodes are HEAD(1), ...HEAD(n) and each list node has three fields: VERTEX, COST, and LINK. COST is the length of the corresponding edge and n the number of vertices in G.

(ii) Instead of representing S, the set of vertices to which the shortest paths have already been found, the set $T = V(G) - S$ is represented using a linked list.

What can you say about the computing time of your new algorithm relative to that of SHORTEST_PATH?

30. Modify algorithm SHORTEST_PATH so that it obtains the shortest paths in addition to the lengths of these paths. What is the computing time of your algorithm?

31. Using the directed graph below, explain why SHORTEST_PATH will not work properly. What is the shortest path between vertices v_1, and v_7?

32. Modify algorithm ALL_COSTS so that it obtains a shortest path for all pairs of vertices i,j. What is the computing time of your new algorithm?

33. By considering the complete graph with n vertices show that the maximum number of paths between two vertices is $(n - 1)!$

34. Use algorithm ALL_COSTS to obtain the lengths of the shortest paths between all pairs of vertices in the graph of exercise 31. Does ALL_COSTS give the right answers? Why?

35. Does the following set of precedence relations (<) define a partial order on the elements 1 thru 5? Why?

$1 < 2; < 2 <4; 2 < 3, 3 < 4; 3 < 5: 5;< 1$

36. a) For the AOE network below obtain the early, $e(\ )$, and late $l(\ )$, start times for each activity. Use the forward-backward approach.

b) What is the earliest time the project can finish?

c) Which activities are critical?

d) Is there any single activity whose speed up would result in a reduction of the project length?

☐

37. Define a critical AOE-network to be an AOE-network in which all activities are critical. Let G be the undirected graph obtained by removing the directions and weights from the edges of the network.

a) Show that the project length can be decreased by speeding exactly one activity if there is an edge in G which lies on every path from the start vertex to the finish vertex. Such an edge is called a bridge. Deletion of a bridge from a connected graph disconnects the graph into two connected components.

b) Write an $O(n + e)$ algorithm using adjacency lists to determine whether the connected graph G has a bridge. In case G has a bridge, your algorithm should output one such bridge.

38. Write a set of computer programs for manipulating graphs. Such a collection should allow input and input of arbitrary graphs. determining connected components and spanning trees. The capability of attaching weights to the edges should also be provided.

39. Make sure you can define the following terms.

```
adjacent(v)          connected(v_p, v_q)

adjacent-to(v)       connected(G)

adjacent-from(v)     connected component

degree(v)            strongly connected

in-degree(v)         tree
```

out-degree(*v*)   network

path   spanning tree

simple path

cycle

subgraph

Go to [Chapter 7](#)   Back to [Table of Contents](#)

CHAPTER 7: INTERNAL SORTING

7.1 SEARCHING

A file is a collection of records, each record having one or more fields The fields used to distinguish among the records are known as *keys*. Since the same file may be used for several different applications, the key fields for record identification will depend on the particular application. For instance, we may regard a telephone directory as a file, each record having three fields: name, address, and phone number. The key is usually the person's name. However, one may wish to locate the record corresponding to a given number, in which case the phone number field would be the key. In yet another application one may desire the phone number at a particular address, so this field too could be the key. Once we have a collection of records there are at least two ways in which to store them: sequentially or non-sequentially. For the time being let us assume we have a sequential file F and we wish to retrieve a record with a certain key value K. If F has n records with K_i the key value for record R_i, then one may carry out the retrieval by examining the key values $K_n, K_{n-1}, ..., K_1$ in that order, until the correct record is located. Such a search is known as sequential search since the records are examined sequentially.

procedure *SEQSRCH(F,n,i,K)*

//Search a file F with key values $K_1, ...,K_n$ for a record R_i such

that $K_i = K$. If there is no such record, i is set to 0//

$K_0 \leftarrow K; \quad i \leftarrow n$

while $K_i \neq K$ **do**

$i \leftarrow i - 1$

end

end *SEQSRCH*

Note that the introduction of the dummy record R_0 with key $K_0 = K$ simplifies the search by eliminating the need for an end of file test ($i < 1$) in the **while** loop. While this might appear to be a minor improvement, it actually reduces the running time by 50% for large n (see table 7.1). If no record in the file has key value K, then $i = 0$, and the above algorithm requires $n + 1$ comparisons. The number of key

comparisons made in case of a successful search, depends on the position of the key in the file. If all keys are distinct and key K_i is being searched for, then $n - i + 1$ key comparisons are made. The average number of comparisons for a successful search is, therefore, $\sum_{1 \le i \le n} (n - i + 1) / n = (n + 1)/2$. For large n this many comparisons is very inefficient. However, we all know that it is possible to do much better when looking up phone numbers. What enables us to make an efficient search? The fact that the entries in the file (i.e., the telephone directory) are in lexicographic order (on the name key) is what enables one to look up a number while examining only a very few entries in the file. So, if the file is ordered one should be able to search for specific records quickly.

One of the better known methods for searching an ordered sequential file is called binary search. In this method, the search begins by examining the record in the middle of the file rather than the one at one of the ends as in sequential search. Let us assume that the file being searched is ordered by nondecreasing values of the key (i.e., in alphabetical order for strings) Then, based on the results of the comparison with the middle key, K_m, one can draw one of the following conclusions:

(i) if $K < K_m$ then if the record being searched for is in the file, it must be in the lower numbered half of the file;

(ii) if $K = K_m$ then the middle record is the one being searched for;

(iii) if $K > K_m$ then if the record being searched for is in the file, it must be in the higher numbered half of the file.

Consequently, after each comparison either the search terminates successfully or the size of the file remaining to be searched is about one half of the original size (note that in the case of sequential search, after each comparison the size of the file remaining to be searched decreases by only 1). So after k key comparisons the file remaining to be examined is of size at most $\lceil n/2^k \rceil$ (n is the number of records). Hence, in the worst case, this method requires $O(\log n)$ key comparisons to search a file. Algorithm BINSRCH implements the scheme just outlined.

```
procedure BINSRCH(F,n,i,K)

//Search an ordered sequential file F with records R_1, ...,R_n and

the keys K_1 ≤ K_2 ≤ [   ] ≤ K_n for a record R_i such that K_i = K;

i = 0 if there is no such record else K_i = K. Throughout the

algorithm, l is the smallest index such that K_l may be K and u
```

the largest index such that K_u may be K//

$l \leftarrow 1; u \leftarrow n$

while $l \leq u$ **do**

$m \leftarrow \lfloor (l + u)/2 \rfloor$ //compute index of middle record//

case

:$K > K_m$: $l \leftarrow m + 1$ //look in upper half//

:$K = K_m$: $i \leftarrow m;$ **return**

:$K < K_m$: $u \leftarrow m - 1$ //look in lower half//

end

end

$i \leftarrow 0$ //no record with key K//

end *BINSRCH*

In the binary search method described above, it is always the key in the middle of the subfile currently being examined that is used for comparison. This splitting process can be described by drawing a binary decision tree in which the value of a node is the index of the key being tested. Suppose there are 31 records, then the first key tested is K_{16} since $\lfloor (1 + 31)/2 \rfloor = 16$. If K is less than K_{16}, then K_8 is tested next $\lfloor (1 + 15)/2 \rfloor = 8$; or if K is greater than K_{16}, then K_{24} is tested. The binary tree describing this process is

A path from the root to any node in the tree represents a sequence of comparisons made by BINSRCH to either find K or determine that it is not present. From the depth of this tree one can easily see that the algorithm makes no more than $O(\log_2 n)$ comparisons.

It is possible to consider other criteria than equal splitting for dividing the remainig file. An alternate

method is Fibonacci search, which splits the subfile according to the Fibonacci sequence,

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

which is defined as $F_0 = 0$, $F_1 = 1$ and

$$F_i = F_{i-1} + F_{i-2}, i \geq 2.$$

An advantage of Fibonacci search is that it involves only addition and subtraction rather than the division in BINSRCH. So its average performance is better than that of binary search on computers for which division takes sufficiently more time than addition/subtraction.

Suppose we begin by assuming that the number of records is one less than some Fibonacci number, $n = F_k - 1$. Then, the first comparison of key K is made with K_{F_k-1} with the following outcomes:

(i) $K < K_{F_k-1}$ in which case the subfile from 1 to $F_{k-1} - 1$ is searched and this file has one less than a Fibonacci number of records;

(ii) $K = K_{F_k-1}$ in which case the search terminates successfully;

(iii) $K > K_{F_k-1}$ in which case the subfile from $F_{k-1} + 1$ to $F_k - 1$ is searched and the size of this file is $F_k - 1 - (F_{k-1} + 1) + 1 = F_k - F_{k-1} = F_{k-2} - 1$.

Again it is helpful to think of this process as a binary decision tree; the resulting tree for $n = 33$ is given on page 340.

This tree is an example of a Fibonacci tree. Such a tree has $n = F_k - 1$ nodes, and its left and right subtrees are also Fibonacci trees with $F_{k-1} - 1$ and $F_{k-2} - 1$ nodes respectively. The values in the nodes show how the file will be broken up during the searching process. Note how the values of the children differ from the parent by the same amount. Moreover, this difference is a Fibonacci number. If we look at a grandparent, parent and two children where the parent is a left child, then if the difference from grandparent to present is Fj, the next difference is F_{j-1}. If instead the parent is a right child then the next difference is F_{j-2}.

The following algorithm implements this Fibonacci splitting idea.

procedure *FlBSRCH(G,n,i,K)*

//search a sequential file *G* with keys ordered in nondecreasing order,

for a record R_i with key $K_i = K$. Assume that $F_k + m = n + 1$,

m $\geq$ 0 and $F_{k+1} > n + 1$. n is the number of records in G. F_k

and F_{k+1} are consecutive Fibonacci numbers. If K is not present,

i is set to zero.//

i □ F_{k-1}; p □ F_{k-2}; q □ F_{k-3}

if $K > K_i$ **then** [//set i so that size of the right subfile is F_{k-2}//

i □ $i + m$]

while $i \neq 0$ **do**

case

:$K < K_i$: **if** $q = 0$ **then** i □ 0

else [i □ $i;$ – $q;$ t □ $p;$ p □ $q;$ q □ t –q]

:$K = K_i$: **return**

:$K > K_i$: **if** $p = 1$ **then** i □ 0

else [i □ $i + q;$ p □ $p - q;q$ □ $q - p$]

end

end

end *FIBSRCH*

Getting back to our example of the telephone directory, we notice that neither of the two ordered search methods suggested above corresponds to the one actually employed by humans in searching the directory. If we are looking for a name beginning with *W*, we start the search towards the end of the directory rather than at the middle. A search method based on this interpolation search would then begin

by comparing key K_i with ☐ are the values of the smallest and largest keys in the file). The behavior of such an algorithm will clearly depend on the distribution of the keys in the file.

The four search procedures sequential, binary, Fibonacci and interpolation search were programmed in Fortran and their performance evaluated on a Cyber 74 computer. The results of this evaluation are presented in table 7.1. Since the input for the last three algorithms is a sorted file, algorithm SEQSRCH was modified to take advantage of this. This was achieved by changing the conditional in the while statement from $K_i \neq K$ to $K_i > K$ and introducing an additional test after the end of the **while** to determine whether $K_i = K$. The inferior sequential method referred to in the table differs from the SEQSRCH just described in that the line K_o ☐ K is not included and a test for $i < 1$ is made in the **while** loop. As the table indicates, inferior sequential search takes almost twice as much time as normal sequential search. For average behavior, binary search is better than sequential search for $n > 15$, while for worst case behavior binary search is better than a sequential search for $n > 4$.

☐

Fibonacci search with n = 33

As can be seen, Fibonacci search always performed worse than binary search. The performance of interpolation search is highly dependent on key value distribution. Its performance is best on uniform distributions. Its average behavior on uniform distributions was better than that for binary search for $n \geq 500$.

We have seen that as far as the searching problem is concerned, something is to be gained by maintaining the file in an ordered manner if the file is to be searched repeatedly. Let us now look at another example where the use of ordered files greatly reduces the computational effort. The problem we are now concerned with is that of comparing two files of records containing data which is essentially the same data but has been obtained from two different sources. We are concerned with verifying that the two files contain the same data. Such a problem could arise, for instance, in the case of the U.S. Internal Revenue Service which might receive millions of forms from various employers stating how much they paid their employees and then another set of forms from individual employees stating how much they received. So we have two files of records, and we wish to verify that there is no discrepancy between the information on the files. Since the forms arrive at the IRS in essentially a random order, we may assume a random arrangement of the records in the files. The key here would probably be the social security numbers. Let the records in the two files, ☐ and ☐ be (☐) and (☐). The corresponding keys are ☐ and ☐. Let us make the following assumptions about the required verification: (i) if corresponding to a key ☐ in the employer file there is no record in the employee file a message is to be sent to the employee; (ii) if the reverse is true, then a message is to be sent to the employer; and (iii) if there is a discrepancy between two records with the same key a message to that effect is to be output.

If one proceeded to carry out the verification directly, one would probably end up with an algorithm

similar to VERIFY1.

☐

(a) Worst case times

☐

(b) Average times obtained by searching for each key once and averaging

Table 7.1 Worst case and average times for different search methods. All times are in milliseconds. (Table prepared by Randall Istre)

☐

One may readily verify that the worst case asymptotic computing time of the above algorithm is $O(mn)$. On the other hand if we first ordered the two files and then made the comparison it would be possible to carry out the verification task in time $O(t_{sort}(n) + t_{sort}(m) + n + m)$ where $t_{sort}(n)$ is the time needed to sort a file of n records. As we shall see, it is possible to sort n records in $O(n \log n)$ time, so the computing time becomes $O(\max \{n \log n, m \log m\})$. The algorithm VERIFY2 achieves this.

We have seen two important uses of sorting: (i) as an aid in searching, and (ii) as a means for matching entries in files. Sorting also finds application in the solution of many other more complex problems, e.g. from operations research and job scheduling. In fact, it is estimated that over 25% of all computing time is spent on sorting with some installations spending more than 50% of their computing time sorting files. Consequently, the problem of sorting has great relevance in the study of computing. Unfortunately, no one method is the "best" for all initial orderings of the file being sorted. We shall therefore study several methods, indicating when one is superior to the others.

First let us formally state the problem we are about to consider.

☐

We are given a file of records $(R_1, R_2, ..., R_n)$. Each record, Ri, has key value K_i. In addition we assume an ordering relation $(<)$ on the key so that for any two key values x and y either $x = y$ or $x < y$ or $y < x$. The ordering relation $(<)$ is assumed to be transitive, i.e., for any three values x, y and z, $x < y$ and $y < z$ implies $x < z$. The sorting problem then is that of finding a permutation, Σ, such that $K\Sigma_{(1)} \leq K\Sigma_{(i+1)}$, $1 \leq i \leq n - 1$. The desired ordering is then $(R\Sigma_{(1)}, R\Sigma_{(2)}, ..., R\Sigma_{(n)})$.

Note that in the case when the file has several key values that are identical, the permutation, Σ, defined

above is not unique. We shall distinguish one permutation, Σ_s, from all the others that also order the file. Let Σ_s be the permutation with the following properties:

(i) $K\Sigma_{s(i)} \leq K\Sigma_{s(i+1)}, 1 \leq i \leq n - 1$.

(ii) If $i < j$ and $K_i = K_j$ in the input file, then R_i precedes R_j in the sorted file.

A sorting method generating the permutation Σ_s will be said to be *stable.*

To begin with we characterize sorting methods into two broad categories: (i) internal methods, i.e., methods to be used when the file to be sorted is small enough so that the entire sort can be carried out in main memory; and (ii) external methods, i.e., methods to be used on larger files. In this chapter we shall study the following internal sorting methods:

a) Insertion sort

b) Quick sort

c) Merge sort

d) Heap sort

e) Radix sort

External sorting methods will be studied in Chapter 8.

7.2 INSERTION SORT

The basic step in this method is to insert a record R into a sequence of ordered records, $R_1, R_2, ..., R_i$, ($K_1 \leq K_2, ..., \leq K_i$) in such a way that the resulting sequence of size $i + 1$ is also ordered. The algorithm below accomplishes this insertion. It assumes the existence of an artificial record R_o with key $K_o = -\infty$ (i.e., all keys are $\geq K_o$).

procedure *INSERT (R,i)*

//Insert record R with key K into the ordered sequence R_o, ...,R_i

in such a way that the resulting sequence is also ordered on key

K. We assume that R_o is a record (maybe a dummy) such that

$K \geq K_o//$

$j \quad \boxed{} \quad i$

while $K < K_j$ **do**

//move R_j one space up as R is to be inserted left of R_j//

$R_{j+1} \quad \boxed{} \quad R_j; \quad j \quad \boxed{} \quad j - 1$

end

$R_{j+1} \quad \boxed{} \quad R$

end *INSERT*

Again, note that the use of R_o enables us to simplify the **while** loop, avoiding a test for end of file, i.e., $j < 1$.

Insertion sort is carried out by beginning with the ordered sequence R_o, R_1, and then successively inserting the records $R_2, R_3, \ldots R_n$ into the sequence. since each insertion leaves the resultant sequence ordered, the file with *n* records can be ordered making *n* - 1 insertions. The details are given in algorithm INSORT on the next page.

Analysis of INSERTION SORT

In the worst case algorithm INSERT(R,i) makes $i + 1$ comparisons before making the insertion. Hence the computing time for the insertion is $O(i)$. INSORT invokes procedure INSERT for $i = 1, 2, \ldots, n - 1$

procedure *INSORT(R,n)*

//sort the records $R_1, \ldots, R_n$ in nondecreasing value of the key K.

Assume *n > 1//*

K_0 ☐ $-\infty$ //Create a dummy record R_0 such that $K_0 < K_i$,

$1 \leq i \leq n$//

for j ☐ 2 **to** n **do**

T ☐ R_j

call *INSERT(T, j - 1)*   //insert records R_2 to R_n//

end

end *INSORT*

resulting in an overall worst case time of ☐.

One may also obtain an estimate of the computing time of this method based upon the relative disorder in the input file. We shall say that the record R_i is *left out of order* (LOO) iff ☐. Clearly, the insertion step has to be carried out only for those records that are LOO. If k is the number of records LOO, then the computing time is $O((k + 1)n)$. The worst case time is still $O(n^2)$. One can also show that $O(n^2)$ is the average time.

Example 7.1: Assume $n = 5$ and the input sequence is (5,4,3,2,1) [note the records have only one field which also happens to be the key]. Then, after each insertion we have the following:

```
-∞ , 5, 4, 3, 2, 1                        [initial sequence]

-∞ , 4, 5, 3, 2, 1                                i = 2

-∞ , 3, 4, 5, 2, 1                                i = 3

-∞ , 2, 3, 4, 5, 1                                i = 4

-∞ , 1, 2, 3, 4, 5                                i = 5
```

Note that this is an example of the worst case behavior.

Example 7.2: $n = 5$ and the input sequence is (2, 3, 4, 5, 1). After each execution of INSERT we have:

```
- ∞ , 2, 3, 4, 5, 1                              [initial]

- ∞ , 2, 3, 4, 5, 1                                     i = 2

- ∞ , 2, 3, 4, 5, 1                                     i = 3

- ∞ , 2. 3, 4, 5, 1                                     i = 4

- ∞ , 1, 2, 3, 4, 5                                     i = 5
```

In this example only R_5 is LOO and the time for each $i = 2,3$ and 4 is $O(1)$ while for $i = 5$ it is $O(n)$.

It should be fairly obvious that this method is stable. The fact that the computing time is $O(kn)$ makes this method very desirable in sorting sequences where only a very few records are LOO (i.e., $k << n$). The simplicity of this scheme makes it about the fastest sorting method for $n \leq 20$ - 25 elements, depending upon the implementation and machine properties. For variations on this method see exercises 3 and 4.

7.3 QUICKSORT

We now turn our attention to a sorting scheme with a very good average behavior. The quicksort scheme developed by C. A. R. Hoare has the best average behavior among all the sorting methods we shall be studying. In Insertion Sort the key K_i currently controlling the insertion is placed into the right spot with respect to the sorted subfile $(R_1, ...,R_{i - 1})$. Quicksort differs from insertion sort in that the key K_i controlling the process is placed at the right spot with respect to the whole file. Thus, if key K_i is placed in position $s(i)$, then $K_j \leq_K s(i)$ for $j < s(i)$ and $_K j \geq K_{s(i)}$ for j > s(i). Hence after this positioning has been made, the original file is partitioned into two subfiles one consisting of records $R_1, ...,R_{s(i)-1}$ and the other of records $R_{s(i)+ 1}, ...,R_n$. Since in the sorted sequence all records in the first subfile may appear to the left of $s(i)$ and all in the second subfile to the right of $s(i)$, these two subfiles may be sorted independently. The method is best stated recursively as below and where INTERCHANGE (x,y) performs $t \square x; x \square y; y \square t$.

procedure *QSORT(m,n)*

```
//sort records Rm, ...,Rn into nondecreasing order on key K. Key

Km is arbitrarily chosen as the control key. Pointers i and j are
```

used to partition the subfile so that at any time $K_l \leq K$, $l < i$

and $K_l \geq K$, $l > j$. It is assumed that $K_m \leq K_{n+1}$//

if $m < n$

then [$i \boxed{} m$; $j \boxed{} n + 1$; $K \boxed{} K_m$

loop

repeat $i \boxed{} i + 1$ **until** $K_i \geq K$;

repeat $j \boxed{} j - 1$ **until** $K_j \leq K$;

if $i < j$

then call *INTERCHANGE* $(R(i),R(j))$

else exit

forever

call *INTERCHANGE* $(R(m),R(j))$

call *QSORT* $(m,j - 1)$

call *QSORT* $(j + 1, n)$]

end *QSORT*

Example 7.3: The input file has 10 records with keys (26, 5, 37, 1, 61, 11, 59, 15, 48, 19). The table below gives the status of the file at each call of QSORT. Square brackets are used to demarcate subfiles yet to be sorted.

R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}	m	n
[26	5	37	1	61	11	59	15	48	19]	1	10
[11	5	19	1	15]	26	[59	61	48	37]	1	5

```
[ 1   5]   11   [19   15]   26   [59   61    48    37]    1    2

  1   5    11   [19   15]   26   [59   61    48    37]    4    5

  1   5    11    15   19    26   [59   61    48    37]    7   10

  1   5    11    15   19    26   [48   37]   59   [61]    7    8

  1   5    11    15   19    26    37   48    59   [61]   10   10

  1   5    11    15   19    26    37   48    59    61
```

Analysis of Quicksort

The worst case behavior of this algorithm is examined in exercise 5 and shown to be $O(n^2)$. However, if we are lucky then each time a record is correctly positioned, the subfile to its left will be of the same size as that to its right. This would leave us with the sorting of two subfiles each of size roughly $n/2$. The time required to position a record in a file of size n is $O(n)$. If $T(n)$ is the time taken to sort a file of n records, then when the file splits roughly into two equal parts each time a record is positioned correctly we have

```
T(n) ≤ cn + 2T(n/2) , for some constant c

    ≤ cn + 2(cn/2 + 2T(n/4))

    ≤ 2cn + 4T(n/4)

    :

    ≤ cn log₂n + nT(1) = O(n log₂ n)
```

In our presentation of QSORT, the record whose position was being fixed with respect to the subfile currently being sorted was always chosen to be the first record in that subfile. Exercise 6 examines a better choice for this control record. Lemma 7.1 shows that the average computing time for Quicksort is $O(n \log_2 n)$. Moreover, experimental results show that as far as average computing time is concerned, it is the best of the internal sorting methods we shall be studying.

Unlike Insertion Sort where the only additional space needed was for one record, Quicksort needs stack space to implement the recursion. In case the files split evenly as in the above analysis, the maximum recursion depth would be log n requiring a stack space of $O(\log n)$. The worst case occurs when the file

is split into a left subfile of size $n - 1$ and a right subfile of size 0 at each level of recursion. In this case, the depth of recursion becomes n requiring stack space of $O(n)$. The worst case stack space can be reduced by a factor of 4 by realizing that right subfiles of size less than 2 need not be stacked. An asymptotic reduction in stack space can be achieved by *sorting smaller subfiles first*. In this case the additional stack space is at most $O(\log n)$.

Lemma 7.1: Let $T_{avg}(n)$ be the expected time for QSORT to sort a file with n records. Then there exists a constant k such that $T_{avg}(n) \leq k\, n \log_e n$ for $n \geq 2$.

Proof: In the call to QSORT $(1,n)$, K_1 gets placed at position j. This leaves us with the problem of sorting two subfiles of size $j - 1$ and $n - j$. The expected time for this is $T_{avg}(j - 1) + T_{avg}(n - j)$. The remainder of the algorithm clearly takes at most cn time for some constant c. Since j may take on any of the values 1 to n with equal probability we have:

☐

(7.1)

We may assume $T_{avg}(0) \leq b$ and $T_{avg}(1) \leq b$ for some constant b. We shall now show $T_{avg}(n) \leq kn \log_e n$ for $n \geq 2$ and $k = 2(b + c)$. The proof is by induction on n.

Induction Base: For $n = 2$ we have from eq. (7.1):

$T_{avg}(2) \leq 2c + 2b \leq kn \log_e 2$

Induction Hypothesis: Assume $T_{avg}(n) \leq kn \log_e n$ for $1 \leq n < m$

Induction Step: From eq. (7.1) and the induction hypothesis we have:

☐

(7.2)

Since $j \log_e j$ is an increasing function of j eq. (7.2) yields:

☐

7.4 HOW FAST CAN WE SORT?

Both of the sorting methods we have seen have a worst case behavior of $O(n^2)$ It is natural at this point to ask the question: "What is the best computing time for sorting that we can hope for?" The theorem we shall prove shows that if we restrict our question to algorithms for which the only operations permitted on keys are comparisons and interchanges then $O(n \log_2 n)$ is the best possible time.

The method we use is to consider a tree which describes the sorting process by having a vertex represent a key comparison and the branches indicate the result. Such a tree is called a *decision tree*. A path through a decision tree represents a possible sequence of computations that an algorithm could produce.

As an example of such a tree, let us look at the tree obtained for Insertion Sort working on a file with three records in it. The input sequence is R_1, R_2 and R_3 so the root of the tree is labeled (1,2,3). Depending on the outcome of the comparison between K_1 and K_2, this sequence may or may not change. If $K_2 < K_1$, then the sequence becomes (2,1,3) otherwise it stays 1,2,3. The full tree resulting from these comparisons is shown below. The leaf nodes are numbered I-VI and are the only points at which the algorithm may terminate. Hence only six permutations of the input sequence are obtainable from this algorithm. Since all six of these are different and 3! = 6, it follows that this algorithm has enough leaves to constitute a valid sorting algorithm for three records. The maximum depth of this tree is 3. The table below gives six different orderings of key values 7, 9, 10 which show that all six permutations are possble. The tree is not a full binary tree of depth 3 and so it has fewer than $2^3 = 8$ leaves. The possible output permutations are:

```
                    SAMPLE  INPUT  KEY  VALUES  WHICH

LEAF    PERMUTATION        GIVE THE  PERMUTATION

  I          1 2 3                   (7,9,10)

 II          1 3 2                   (7,10,9)

III          3 1 2                   (9,10,7)

 IV          2 1 3                   (9,7,10)

  V          2 3 1                   (10,7,9)

 VI          3 2 1                   (10,9,7)
```

The decision tree is

Theorem 7.1: Any decision tree that sorts n distinct elements has a height of at least $\log_2(n!) + 1$.

Proof: When sorting n elements there are $n!$ different possible results. Thus, any decision tree must have $n!$ leaves. But a decision tree is also a binary tree which can have at most 2^{k-1} leaves if its height is k. Therefore, the height must be at least $\log_2 n! + 1$.

Corollary: Any algorithm which sorts by comparisons only must have a worst case computing time of $O(n \log_2 n)$.

Proof: We must show that for every decision tree with $n!$ leaves there is a path of length $c\, n\log_2 n$, c a constant. By the theorem, there is a path of length $\log_2 n!$. Now

$$n! = n(n - 1)(n - 2) \ldots (3)(2)(1)$$

$$\geq (n/2)^{n/2},$$

so $\log_2 n! \geq (n/2) \log_2 (n/2) = O(n \log_2 n)$.

7.5 2-WAY MERGE SORT

Before looking at the merge sort algorithm to sort n records let us see how one may merge two files $(X_l, ...,X_m)$ and $(X_{m+1}, ...,X_n)$ that are already sorted to get a third file $(Z_l, ...,Z_n)$ that is also sorted. Since this merging scheme is very simple, we directly present the algorithm.

```
procedure MERGE(X,l,m,n,Z)

//(X_l, ...,X_m) and (X_{m+1}, ...,X_n) are two sorted files with keys

x_l ≤ ... ≤ x_m and x_{m+1} ≤ ... ≤ x_n. They are merged to obtain the

sorted file (Z_1, ..., Z_n) such that z_l ≤ ... ≤ z_n//

i □ k □ l; j □ m + 1      //i, j and k are position in the three
files//

while i ≤ m and j ≤ n do
```

if $x_i \leq x_j$ *then* $[Z_k \quad\square\quad X_i; \ i \quad\square\quad i + 1]$

else $[Z_k \quad\square\quad X_j; \ j \quad\square\quad j + 1]$

$k \quad\square\quad k + 1$

end

if $i > m$ **then** $(Z_k, \ldots, Z_n) \quad\square\quad (X_j, \ldots, X_n)$

else $(Z_k, \ldots, Z_n) \quad\square\quad (X_i \ldots, X_m)$

end *MERGE*

Analysis of Algorithm MERGE

At each iteration of the **while** loop k increases by 1. The total increment in k is $n - l + 1$. Hence the **while** loop is iterated at most $n - l + 1$ times. The **if** statement moves at most one record per iteration. The total time is therefore $O(n - l + 1)$. If records are of length M then this time is really $O(M(n - l + 1))$. When M is greater than 1 we could use linked lists $(X_1, \ldots, X_m)$ and $(X_{m+1}, \ldots, X_n)$ and obtain a new sorted linked list containing these $n - l + 1$ records. Now, we won't need the additional space for $n - l + 1$ records as needed above for Z. Instead only space for $n - l + 1$ links is needed. The merge time becomes independent of M and is $O(n - l + 1)$. Note that $n - l + 1$ is the number of records being merged.

Two-way merge sort begins by interpreting the input as n sorted files each of length 1. These are merged pairwise to obtain $n/2$ files of size 2 (if n is odd, then one file is of size 1). These $n/2$ files are then merged pairwise and so on until we are left with only one file. The example below illustrates the process.

Example 7.4.1: The input file is (26, 5, 77, 1, 61, 11, 59, 15, 15, 48, 19). The tree below illustrates the subfiles being merged at each pass:

As is apparent from the example above, merge sort consists of several passes over the records being sorted. In the first pass files of size 1 are merged. In the second, the size of the files being merged is 2. On the ith pass the files being merged are of size 2^{i-1}. Consequently, a total of $\lceil \log_2 n \rceil$ passes are made over the data. Since two files can be merged in linear time (algorithm MERGE), each pass of merge sort takes $O(n)$ time. As there are $\lceil \log_2 n \rceil$ passes, the total computing time is $O(n \log n)$.

In formally writing the algorithm for 2-way merge, it is convenient to first present an algorithm to perform one merge pass of the merge sort.

procedure *MPASS(X,Y,n,l)*

//This algorithm performs one pass of merge sort. It merges adjacent

pairs of subfiles of length *l* from file *X* to file *Y*. *n* is the number

of records in *X*//

i ☐ 1

while *i* ≤ *n* - 2*l* + 1 **do**

call *MERGE* (*X*, *i*, *i* + *l* - 1, *i* + 2*l* - 1, *Y*)

i ☐ *i* + 2*l*

end

//merge remaining file of length <2*l*//

if *i* + *l* - 1 < *n* **then call** *MERGE* (*X*, *i*, *i* + *l* - 1, *n*, *Y*)

else (Y_i, ...Y_n) ☐ (X_i, ...,X_n)

end *MPASS*

The merge sort algorithm then takes the form:

procedure *MSORT(X,n)*

//Sort the file $X = (X_1, ...,X_n)$ into nondecreasing order of the keys

$x_1, ...,x_n$//

declare *X(n)*, *Y(n)*   //Y is an auxilliary array//

```
//l is the size of subfiles currently being merged//

l □ 1

while l < n do

call MPASS(X,Y,n,l)

l □ 2*l

call MPASS(Y,X,n,l)            //interchange role of X and Y//

l □ 2*l

end

end MSORT
```

It is easy to verify that the above algorithm results in a stable sorting procedure. Exercise 10 discusses a variation of the two-way merge sort discussed above. In this variation the prevailing order within the input file is taken into account to obtain initially sorted subfiles of length ≥ 1.

Recursive Formulation of Merge Sort

Merge sort may also be arrived at recursively. In the recursive formulation we divide the file to be sorted into two roughly equal parts called the left and the right subfiles. These subfiles are sorted using the algorithm recursively and then the two subfiles are merged together to obtain the sorted file. First, let us see how this would work on our earlier example.

Example 7.4.2: The input file (26, 5, 77, 1, 61, 11, 59, 15, 49, 19) is to be sorted using the recursive formulation of 2-way merge sort. If the subfile from l to u is currently to be sorted then its two subfiles are indexed from l to $\lfloor (l + u)/2 \rfloor$ and from $\lfloor (l + u)/2 \rfloor + 1$ to u. The subfile partitioning that takes place is described by the following binary tree. Note that the subfiles being merged are different from those being merged in algorithm MSORT.

From the preceding example, we may draw the following conclusion. If algorithm MERGE is used to merge sorted subfiles from one array into another, then it is necessary to copy subfiles. For example to merge [5, 26] and [77] we would have to copy [77] into the same array as [5, 26]. To avoid this

unnecessary copying of subfiles using sequential allocation, we look to a linked list representation for subfiles. This method of representation will permit the recursive version of merge sort to work efficiently.

Each record is assumed to have two fields LINK and KEY. LINK (i) and KEY(i) are the link and key value fields in record i, $1 \le i \le n$. We assume that initially LINK(i) = 0, $1 \le i \le n$. Thus each record is initially in a chain containing only itself. Let Q and R be pointers to two chains of records. The records on each chain are assumed linked in nondecreasing order of the key field. Let RMERGE(Q,R,P) be an algorithm to merge the two chains Q and R to obtain P which is also linked in nondecreasing order of key values. Then the recursive version of merge sort is given by algorithm RMSORT. To sort the records X_1, ...,X_n this algorithm is invoked as **call** RMSORT(X,1,n,P). P is returned as the start of a chain ordered as described earlier. In case the file is to be physically rearranged into this order then one of the schemes discussed in section 7.8 may be used.

procedure *RMSORT(X,l,u,P)*

//The file X = (X₁, ...,Xᵤ) is to be sorted on the field KEY. LINK

is a link field in each record and is initially set to 0. The sorted

file is a chain beginning at P//

if *l* $\ge$ *u* **then** *P* □ *l*

else [mid □ $\lfloor$*(l + u)*/2$\rfloor$

call *RMSORT(X,l,*mid*,Q)*

call *RMSORT(X,*mid + 1*,u,R)*

call *RMERGE(Q,R,P)*]

end *RMSORT*

The algorithm RMERGE below uses a dummy record with index d. It is assumed that d is provided externally and that d is not one of the valid indexes of records i.e. d is not one of the numbers 1 through n.

procedure *RMERGE(X,Y,Z)*

```
//The linked files X and Y are merged to obtain Z. KEY(i) denotes

the key field and LINK(i) the link field of record i. In X, Y and

Z the records are linked in order of nondecreasing KEY values.

A dummy record with index d is made use of. d is not a valid

index in X or Y//

i ← X; j ← Y; z ← d

while i ≠ 0 and j ≠ 0 do

if KEY(i) ≤ KEY(j) then [LINK(z) ← i

z ← i; i ← LINK (i)]

else [LINK(z) ← j

z ← j; j ← LINK(j)]

end

//move remainder//

if i = 0 then LINK(z) ← j

else LINK(z) ← i

Z ← LINK(d)

end RMERGE
```

One may readily verify that this linked version of 2-way merge sort results in a stable sorting procedure and that the computing time is $O(n \log n)$.

7.6 HEAP SORT

While the Merge Sort scheme discussed in the previous section has a computing time of $O(n \log n)$ both in the worst case and as average behavior, it requires additional storage proportional to the number of records in the file being sorted. The sorting method we are about to study will require only a fixed amount of additional storage and at the same time will have as its worst case and average computing time $O(n \log n)$. In this method we shall interpret the file to be sorted $R = (R_1, ...,R_n)$ as a binary tree. (Recall that in the sequential representation of a binary tree discussed in Chapter 5 the parent of the node at location i is at $\lfloor i/2 \rfloor$, the left child at $2i$ and the right child at $2i + 1$. If $2i$ or $2i + 1$ is greater than n (the number of nodes), then the corresponding children do not exist.) Thus, initially the file is interpreted as being structured as below:

□

Heap sort may be regarded as a two stage method. First the tree representing the file is converted into a heap. A *heap* is defined to be a complete binary tree with the property that the value of each node is at least as large as the value of its children nodes (if they exist) (i.e., $K_{\lfloor j/2 \rfloor} \geq K_j$ for $1 \leq \lfloor j/2 \rfloor < j \leq n$). This implies that the root of the heap has the largest key in the tree. In the second stage the output sequence is generated in decreasing order by successively outputting the root and restructuring the remaining tree into a heap.

Essential to any algorithm for Heap Sort is a subalgorithm that takes a binary tree T whose left and right subtrees satisfy the heap property but whose root may not and adjusts T so that the entire binary tree satisfies the heap property. Algorithm ADJUST does this.

```
procedure ADJUST (i,n)

//Adjust the binary tree with root i to satisfy the heap property.

The left and right subtrees of i, i.e., with roots 2i and 2i+ 1,
already

satisfy the heap property. The nodes of the trees contain records,

R, with keys K. No node has index greater than n//

R □ Rᵢ; K □ Kᵢ; j □ 2i

while j ≤ n do

if j < n and Kⱼ < Kⱼ₊₁ then j □ j + 1    //find max of left
```

and right child//

//compare max. child with K. If K is max. then done//

if $K \geq K_j$ **then exit**

$R_{\lfloor j/2 \rfloor} \Box R_j$; j $\Box$ 2j   //move R_j up the tree//

end

$R_{\lfloor j/2 \rfloor} \Box R$

end *ADJUST*

Analysis of Algorithm Adjust

If the depth of the tree with root i is k, then the **while** loop is executed at most *k* times. Hence the computing time of the algorithm is *O*(*k*).

The heap sort algorithm may now be stated.

procedure *HSORT* (*R,n*)

//The file $R = (R_1, ...,R_n)$ is sorted into nondecreasing order of

the key *K*//

for i $\Box$ $\lfloor n/2 \rfloor$ **to** 1 **by** -1 **do**   //convert *R* into a heap//

call *ADJUST* (*i,n*)

end

for i $\Box$ $n - 1$ **to** 1 **by** -1 **do**   //sort *R*//

T $\Box$ R_{i+1}; R_{i+1} $\Box$ R_1; R_1 $\Box$ T //interchange R_1 and R_{i+1}//

call *ADJUST* (1,*i*)   //recreate heap//

end

end *HSORT*

Example 7.5: The input file is (26, 5, 77, 1, 61, 11, 59, 15, 48, 19). Interpreting this as a binary tree we have the following transformations:

□

The figures on pages 360-361 illustrate the heap after restructuring and the sorted part of the file.

Analysis of Algorithm HSORT

Suppose $2^{k-1} \leq n < 2^k$ so that the tree has k levels and the number of nodes on level i is 2^{i-1}. In the first **for** loop ADJUST is called once for each node that has a child. Hence the time required for this loop is the sum, over each level, of the number of nodes on a level times the maximum distance the node can move. This is no more than

□

In the next **for** loop $n - 1$ applications of ADJUST are made with maximum depth $k = \lceil \log_2 (n + 1) \rceil$. Hence the computing time for this loop is O($n \log n$). Consequently, the total computing time is $O(n \log n)$. Note that apart from pointer variables, the only additional space needed is space for one record to carry out the exchange in the second **for** loop. Note also that instead of making the exchange, one could perform the sequence $R \ \square \ R_{i+1}, R_{i+1} \ \square \ R_1$ and then proceed to adjust the tree.

7.7 SORTING ON SEVERAL KEYS

Let us now look at the problem of sorting records on several keys, $K^1, K^2, ..., K^r$ (K^1 is the most significant key and K^r the least). A file of records $R_1, ..., R_n$ will be said to be sorted with respect to the keys $K^1, K^2, ..., K^r$ iff for every pair of records $\square$. The *r-tuple* ($x_1, ..., x_r$) is less than or equal to the *r-tuple* ($y_1, ..., y_r$) iff either $x_i = y_i$, $1 \leq i \leq j$ and $x_{j+1} < y_{j+1}$ for some $j \leq r$ *or* $x_i = y_i$, $1 \leq i \leq r$.

□

□

For example, the problem of sorting a deck of cards may be regarded as a sort on two keys, the suit and

face values, with the following ordering relations:

and face values: $2 < 3 < 4 \ldots < 10 < J < Q < K < A.$

There appear to be two popular ways to accomplish the sort. The first is to sort on the most significant key K^1 obtaining several "piles" of records each having the same value for K^1. Then each of these piles is independently sorted on the key K^2 into "subpiles" such that all the records in the same subpile have the same values for K^1 and K^2. The subpiles are then sorted on K^3, etc., and the piles put together. In the example above this would mean first sorting the 52 cards into four piles one for each of the suit values. Then sort each pile on the face value. Now place the piles on top of each other to obtain the ordering: 2 ☐ , ...,A ☐ , ..., ...,2 ♠ , ...,A ♠ .

A sort proceeding in this fashion will be referred to as a most significant digit first (MSD) sort. The second way, quite naturally, is to sort on the least significant digit first (LSD). This would mean sorting the cards first into 13 piles corresponding to their face values (key K^2). Then, place the 3's on top of the 2's, ..., the kings on top of the queens, the aces on top of the kings; turn the deck upside down and sort on the suit (K^1) using some stable sorting method obtaining four piles each of which is ordered on K^2; combine the piles to obtain the required ordering on the cards.

Comparing the two procedures outlined above (MSD and LSD) one notices that LSD is simpler as the piles and subpiles obtained do not have to be sorted independently (provided the sorting scheme used for sorting on key K^i, $1 \leq i < r <$ is stable). This in turn implies less overhead.

LSD and MSD only specify the order in which the different keys are to be sorted on and not the sorting method to be used within each key. The technique generally used to sort cards is a MSD sort in which the sorting on suit is done by a bin sort (i.e., four "bins" are set up, one for each suit value and the cards are placed into their corresponding "bins"). Next, the cards in each bin are sorted using an algorithm similar to Insertion Sort. However, there is another way to do this. First use a bin sort on the face value. To do this we need thirteen bins one for each distinct face value. Then collect all the cards together as described above and perform bin sort on the suits using four bins. Note that a bin solt requires only $O(n)$ time if the spread in key values is $O(n)$.

LSD or MSD sorting can also be used to sort records on only one logical key by interpreting this key as being composed of several keys. For example, if the keys are numeric, then each decimal digit may be regarded as a key. So if all the keys are in the range $0 \leq K \leq 999$, then we can use either the LSD or MSD sorts for three keys (K^1,K^2,K^3), where K^1 is the digit in the hundredths place, K^2 the digit in the tens place, and K^3 that in the units place. Since all the keys lie in the range $0 \leq K^i \leq 9$, the sort within the keys can be carried out using a bin sort with ten bins. This, in fact, is essentially the process used to sort records punched on cards using a card sorter. In this case using the LSD process would be more

convenient as it eliminates maintaining several independent subpiles. If the key is interpreted as above, the resulting sort is called a radix 10 sort. If the key decomposition is carried out using the binary representation of the keys, then one obtains a radix 2 sort. In general, one could choose any radix r obtaining a radix r sort. The number of bins required is r.

Let us look in greater detail at the implementation of an LSD radix r sort. We assume that the records $R_1, ..., R_n$ have keys that are d-tuples $(x_1, x_2, ..., x_d)$ and $0 \leq x_i < r$. Thus, we shall need r bins. The records are assumed to have a LINK field. The records in each bin will be linked together into a linear linked list with $F(i)$, $0 \leq i < r$, a pointer to the first record in bin i and $E(i)$ a pointer to the last record in bin i. These lists will essentially be operated as queues. Algorithm LRSORT formally presents the LSD radix r method.

Analysis of LRSORT

The algorithm makes d passes over the data, each pass taking $O(n + r)$ time. Hence the total computing time is $O(d(n + r))$. In the sorting of numeric data, the value of d will depend on the choice of the radix r and also on the largest key. Different choices of r will yield different computing times (see Table 7.2).

```
procedure LRSORT(R,n,d)

//records R = (R₁, ...,Rₙ) are sorted on the keys K¹, ...,Kᵈ. The

range of each key is 0 ≤ Kⁱ < r. Sorting within a key is done using

bin sort. All records are assumed to be initially linked together

such that LINK(i) points to Rᵢ₊₁, 1 ≤ i ≤ n and LINK(n) = 0.//

declare E(0:r - 1), F(0:r - 1)     //queue pointers//

p ⬜ 1        //pointer to start of list of records//

for i ⬜ d to 1 by - 1 do       //sort on key Kⁱ//

for j ⬜ 0 to r - 1 do      //initialize bins to be empty queues//

F(j) ⬜ 0

end
```

```
while p ≠ 0 do          //put records into queues//

k ← □   //k is the i-th key of p-th record//

if F(k) = 0 then F(k) ← p          //attach record p into bin k//

else LINK(E(k)) ← p

E(k) ← p

p ← LINK(p)          //get next record//

end

j ← 0; while F(j) = 0 do j ← j + 1 end   //find first nonempty

queue//

p ← F(j); t ← E(j)

for k ← j + 1 to r - 1 do          //concatenate remaining queues//

if F(k) ≠ 0 then [LINK(t) ← F(k); t ← E(k)]

end

LINK(t) ← 0

end

end LRSORT
```

Example 7.6: We shall illustrate the operation of algorithm LRSORT while sorting a file of 10 numbers in the range [0,999]. Each decimal digit in the key will be regarded as a subkey. So, the value of d is 3 and that of r is 10. The input file is linked and has the form given on page 365 labeled R_1, ...,R_{10}. The figures on pages 365-367 illustrates the $r = 10$ case and the list after the queues have been collected from the 10 bins at the end of each phase. By using essentially the method above but by varying the radix, one can obtain (see exercises 13 and 14) linear time algorithms to sort n record files when the keys are in the

range $0 \le K_i < n^k$ for some constant k.

☐

☐

☐

7.8 PRACTICAL CONSIDERATIONS FOR INTERNAL SORTING

Apart from radix sort, all the sorting methods we have looked at require excessive data movement; i.e., as the result of a comparison, records may be physically moved. This tends to slow down the sorting process when records are large. In sorting files in which the records are large it is necessary to modify the sorting methods so as to minimize data movement. Methods such as Insertion Sort and Merge Sort can be easily modified to work with a linked file rather than a sequential file. In this case each record will require an additional link field. Instead of physically moving the record, its link field will be changed to reflect the change in position of that record in the file (see exercises 4 and 8). At the end of the sorting process, the records are linked together in the required order. In many applications (e.g., when we just want to sort files and then output them record by record on some external media in the sorted order) this is sufficient. However, in some applications it is necessary to physically rearrange the records *in place* so that they are in the required order. Even in such cases considerable savings can be achieved by first performing a linked list sort and then physically rearranging the records according to the order specified in the list. This rearranging can be accomplished in linear time using some additional space.

If the file, F, has been sorted so that at the end of the sort P is a pointer to the first record in a linked list of records then each record in this list will have a key which is greater than or equal to the key of the previous record (if there is a previous record), see figure 7.1. To physically rearrange these records into the order specified by the list, we begin by interchanging records R_1 and R_P. Now, the record in the position R_1 has the smallest key. If $P \neq 1$ then there is some record in the list with link field = 1. If we could change this link field to indicate the new position of the record previously at position 1 then we would be left with records $R_2, ...,R_n$ linked together in nondecreasing order. Repeating the above process will, after $n - 1$ iterations, result in the desired rearrangement. The snag, however, is that in a singly linked list we do not know the predecessor of a node. To overcome this difficulty, our first rearrangement algorithm LIST1, begins by converting the singly linked list P into a doubly linked list and then proceeds to move records into their correct places.

☐

Figure 7.1 List Sort

procedure *LIST*1(*R,n,P*)

//*P* is a pointer to a list of *n* sorted records linked together by the

field *LINK*. A second link field, *LINKB*, is assumed to be present

in each record. The records are rearranged so that the resulting

records R_1, ..., R_n are consecutive and sorted//

u ← 0; *s* ← *P*

while *s* ≠ 0 **do**   //convert *P* into a doubly linked list using *LINKB*//

LINKB(*s*) ← *u*   //*u* follows *s*//

u ← *s*; *s* ← *LINK*(*s*)

end

for *i* ← 1 to *n* - 1 **do**   //move R_P to position *i* while//

if *P* ≠ *i*   //maintaining the list//

then [**if** *LINK* (*i*) ≠ 0 then *LINKB*(*LINK* (*i*)) ← *P*

LINK(*LINKB*(*i*)) ← *P*; *A* ← R_P

R_P ← R_i; R_i ← *A*]

P ← *LINK*(*i*)   //examine the next record//

end

end *LIST*1

Example 7.7: After a list sort on the input file (35, 18, 12, 42, 26, 14) has been made the file is linked as below (only three fields of each record are shown):

```
          R₁   R₂   R₃   R₄   R₅   R₆

Key       35   18   12   42   26   14

LINK       4    5    6    0    1    2   P = 3

LINK B
```

Following the links starting at R_P we obtain the logical sequence of records R_3, R_6, R_2, R_5, R_1, and R_4 corresponding to the key sequence 12, 14, 18, 26, 35, and 42. Filling in the backward links, we have

```
R1   R2   R3   R4   R5   R6

35   18   12   42   26   14

 4    5    6    0    1    2   P = 3

 5    6    0    1    2    3
```

The configuration at the end of each execution of the **for** loop is:

```
12   18   35   42   26   14

 6    5    4    0    3    2   P = 6

 0    6    5    3    2    3

          i = 1
```

The logical sequence of the remaining list (*LS*) is: R_6, R_2, R_5, R_3, R_4. The remaining execution yields

```
         12      14   35   42   26   18

          6       6    4    0    3    5   P = 6

          0       3    5    3    6    6
```

```
LS:   R6.   R5,   R3,   R4    i = 2

              12    14   18   42   26   35

               6    6    5    0    6    4   P = 5

               0    3    6    6    6    5

LS:  R5,  R6,  R4   i = 3

              12   14   18   26   42   35

               6    6    5    6    0    5   P = 6

               0    3    6    6    6    5

LS:  R6,  R5,        i = 4

              12   14   18   26   35   42

               6    6    5    6    6    0   P = 6

               0    3    6    6    5    6

LS: R6,              i = 5
```

Analysis of Algorithm LIST1

If there are n records in the file then the time required to convert the chain P into a doubly linked list is O (n). The **for** loop is iterated n-1 times. In each iteration at most two records are interchanged. This requires 3 records to move. If each record is m words long, then the cost per intgrchange is $3m$. The total time is therefore $O(nm)$. The worst case of $3(n - 1)$ record moves is achievable. For example consider the input key sequence $R_1, R_2, ... R_n$ with $R_2 < R_3 < ... < R_n$ and $R_1 > R_n$. For $n = 4$ and keys 4, 1, 2, 3 the file after each iteration has the following form: $i = 1$: 1,4,2,3; $i = /2$: 1,2,4,3; $i = 3$: 1,2,3,4. A total of 9 record moves is made.

Several modifications to algorithm LIST1 are possible. One that is of interest was given by M. D. MacLaren. This results in a rearrangement algorithm in which no additional link fields are necessary. In this algorithm, after the record R_P is exchanged with R_i the link field of the new R_i is set to P to indicate that the original record was moved. This, together with the observation that P must always be $\geq i$,

permits a correct reordering of the records. The computing time remains $O(nm)$.

```
procedure LIST2 (R,n,P)

//Same function as LIST1 except that a second link field LINKB

is not reguired//

for i ⬚ 1 to n - 1 do

//find correct record to place into i'th position. The index of this

record must be ≥ i as records in positions 1,2, ...,i -1 are already

correctly positioned//

while P < i do

P ⬚ LINK (P)

end

Q ⬚ LINK (P)          //R_Q is next record with largest key//

if P ≠ i then [//interchange R_i and R_p moving R_p to its correct

spot as R_p has i'th smallest key. Also set link from

old position of R_i to new one//   T ⬚ R_i R_i ⬚ R_p; R_p ⬚ T

LINK (i) ⬚ P]

P ⬚ Q

end

end LIST2
```

Example 7.8: The data is the same as in Example 7.7. After the list sort we have:

```
        R₁   R₂   R₃   R₄   R₅   R₆
```

Key   35   18   12   42   26   14

LINK   4   5   6   0   1   2   P = 3

After each iteration of the **for** loop, we have:

12   18   35   42   26   14

 3   5   4   0   3   2   P = 6

         i = 1

12   14   35   42   26   18

 3   6   4   0   1   5   P = 2

         i = 2

Now $P < 3$ and so it is advanced to $\text{LINK}(P) = 6$.

12   14   18   42   26   35

 3   6   6   0   1   4 P = 5

         i = 3

12   14   18   26   42   35

 3   6   6   5   0   4   P = 1

         i = 4

Again $P < 5$ and following links from R_p we find R_6 to be the record with fifth smallest key.

12   14   18   26   35   42

 3   6   6   5   6   0   P = 4

```
        i = 5
```

Analysis of Algorithm LIST2

The sequence of record moves for LIST2 is identical to that for LIST1. Hence, in the worst case $3(n - 1)$ record moves for a total cost of $O(n\ m)$ are made. No node is examined more than once in the **while** loop. So the total time for the [**while** loop is $O(n)$. While the asymptotic computing time for both LIST1 and LIST2 is the same and the same number of record moves is made in either case, we would expect LIST2 to be slightly faster than LIST1 because each time two records are interchanged LIST1 does more work than LIST2 does. LIST1 is inferior to LIST2 on both space and time considerations.

The list sort technique discussed above does not appear to be well suited for use with sort methods such as Quick Sort and Heap Sort. The sequential representation of the heap is essential to Heap Sort. In such cases as well as in the methods suited to List Sort, one can maintain an auxiliary table with one entry per record. The entries in this table serve as an indirect reference to the records. Let this table be T(l), T(2), ...,T(n). At the start of the sort $T(i) = i$, $1 \le i \le n$. If the sorting algorithm requires an interchange of R_i and R_j, then only the table entries need be interchanged, i.e., T(i) and T(j). At the end of the sort, the record with the smallest key is $R_{T(1)}$ and that with the largest $R_{T(n)}$. In general, following a table sort $R_{T(i)}$ is the record with the i'th smallest key. The required permutation on the records is therefore $R_{T(1)}$, $R_{T(2)}$, ...,$R_{T(n)}$ (see Figure 7.2). This table is adequate even in situations such as binary search, where a sequentially ordered file is needed. In other situations, it may be necessary to physically rearrange the records according to the permutation specified by T. The algorithm to rearrange records corresponding to the permutation T(1), T(2), ...,T(n) is a rather interesting application of a theorem from mathematics: viz, every permutation is made up of disjoint cycles. The cycle for any element i is made up of i, T(i), $T^2(i)$, ...,$T^k(i)$, (where $T^j(i) = T(T^{j-1}(i))$ and $T^o(i) = i$) such that $T^k(i) = i$. Thus, the permutation T of figure 7.2 has two cycles, the first involving R_1 and R_5 and the second involving R_4, R_3 and R_2. Algorithm TABLE utilizes this cyclic decomposition of a permutation. First, the cycle containing R_1 is followed and all records moved to their correct positions. The cycle containing R_2 is the next one examined unless this cycle has already been examined. The cycles for R_3, R_4, ...,R_{n-1} are followed in that order, achieving a reordering of all the records. While processing a trivial cycle for R_i (i.e. $T(i) = i$), no rearrangement involving record R_i is required since the condition $T(i) = i$ means that the record with the i'th smallest key is R_i. In processing a nontrivial cycle for record R_i (i.e. $T(i) \ne i$), R_i is moved to a temporary position P, then the record at T(i) is moved to (i); next the record at T(T(i)) is moved to T(i); and so on until the end of the cycle $T^k(i)$ is reached and the record at P is moved to $T^{k-1}(i)$.

Figure 7.2 Table Sort

```
procedure TABLE(R,n,T)
```

//The records R_1, ...,R_n are rearranged to correspond to the sequence

$R_{T(1)}$, ...,$R_{T(n)}$, $n \geq 1$//

for i ⬚ 1 **to** n - 1 **do**

if $T(i) \neq$ i **then**   //there is a nontrivial cycle starting at i//

[P ⬚ R_i; j ⬚ i //move R_i to a temporary spot P and follow//

repeat   //cycle i, $T(i)$, $T(T(i))$, ... until the correct spot//

k ⬚ $T(j)$

Rj ⬚ R_k

$T(j)$ ⬚ j

j ⬚ k

until $T(j)$ = i

R_j ⬚ P //j is position for record P//

$T(j)$ ⬚ j]

end

end TABLE

Example 7.9: : Following a table sort on the file *F* we have the following values for *T* (only the key values for the 8 records of *F* are shown):

R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8
F 35	14	12	42	26	50	31	18

```
T    3    2    8    5    7    1    4    6
```

There are two nontrivial cycles in the permutation specified by T. The first is R_1, R_3, R_8, R_6 and R_1. The second is R_4, R_5, R_7, R_4. During the first iteration ($i = 1$) of the for loop of algorithm TABLE, the cycle $R_1, R_{T(1)}, R_{T^2(1)}, R_{T^3(1)}, R_1$ is followed. Record R_1 is moved to a temporary spot P. $R_{T(1)}$ (i.e. R_3) is moved to the position R_1; $R_{T^2(1)}$ (i.e. R_8) is moved to R_3; R_6 to R_8 and finally P to R_6. Thus, at the end of the first iteration we have:

```
F   12   14   18   42   26   35   31   50

T    1    2    3    5    6    7    4    8
```

For $i = 2,3$, $T(i) = i$, indicating that these records are already in their correct positions. When $i = 4$, the next nontrivial cycle is discovered and the records on this cycle R_4, R_5, R_7, R_4 are moved to their correct positions. Following this we have:

```
F   12   14   18   26   31   35   42   50

T    1    2    3    5    7    6    4    8
```

For the remaining values of i ($i = 5, 6$ and 7), $T(i) = i$, and no more nontrivial cycles are found.

Analysis of Algorithm TABLE

If each record uses m words of storage then the additional space needed is m words for P plus a few more for variables such as i, j and k. To obtain an estimate of the computing time we observe that the **for** loop is executed $n-1$ times. If for some value of i, $T(i) \neq i$ then there is a nontrivial cycle including $k > 1$ distinct records $R_i, R_{T(i)}, ...,R_{T^{k-1}(i)}$. Rearranging these records requires $k + 1$ record moves. Following this, the records involved in this cycle are not moved again at any time in the algorithm since $T(j) = j$ for all such records Rj. Hence no record can be in two different nontrivial cycles. Let k_l be the number of records on a nontrivial cycle starting at R_l when $i = l$ in the algorithm. Let $k_l = 0$ for a trivial cycle. Then, the total number of record moves is $\boxed{}$. Since the records on nontrivial cycles must be different, $\Sigma kl \leq n$. The total record moves is thus maximum when $\Sigma k_l = n$ and there are $\lfloor n/2 \rfloor$ cycles. When n is even, each cycle contains 2 records. Otherwise one contains three and the others two. In either case the number of record moves is $\lfloor 3n/2 \rfloor$. One record move costs $O(m)$ time. The total computing time is therefore $O(nm)$.

In comparing the algorithms LIST2 and TABLE for rearranging records we see that in the worst case

LIST2 makes $3(n - 1)$ record moves while TABLE makes only $\lfloor 3n/2 \rfloor$ record moves. For larger values of *m* it would therefore be worthwhile to make one pass over the sorted list of records creating a table *T* corresponding to a table sort. This would take $O(n)$ time. Then algorithm TABLE could be used to rearrange the records in the order specified by *T*.

n ⬜	10	20	50	100	250	500	1000
Quicksort [with median of 3] (File: [N,1,2,3, ..., N - 2, N - 1])	.499	1.26	4.05	12.9	68.7	257.	1018.
Quicksort [without median of 3] (File: [1,2,3, ..., N - 1, N])	.580	1.92	9.92	38.2	226.	856.	3472.
Insertion Sort [with K(0) = $-\infty$] (File: [N,N - 1, ...,2,1])	.384	1.38	8.20	31.7	203.	788.	--
Insertion Sort [without K(0) = $-\infty$] (File: [N,N - 1, ...,2,1])	.382	1.48	9.00	35.6	214.	861.	--
Heap Sort	.583	1.52	4.96	11.9	36.2	80.5	177.
Merge Sort	.726	1.66	4.76	11.3	35.3	73.8	151

(a) Worst case times m milliseconds

n	10	20	50	100	250	500	1000
Radix Sort (L.S.D.) (R $\geq$ 100,000; optimal D & R)	1.82 R = 18; D = 4	3.05 R = 18; D = 4	5.68 R = 47; D = 3	9.04 R = 47; D = 3	20.1 R = 317; D = 2	32.5 R = 317; D = 2	58.0 R = 317; D = 2
Radix Sort (L.S.D.) (R = 10, D = 5)	1.95	3.23	6.97	13.2	32.1	66.4	129.
Quicksort [with median of 3]	.448	1.06	3.17	7.00	20.0	43.1	94.9

Quicksort

[without

median

of 3]   .372   .918   2.89   6.45   20.0
43.6   94.0

Insertion

Sort   .232   .813   4.28   16.6   97.1
385.   --

Insertion

Sort

(without

K(0) = $-\infty$)   .243   .885   4.72   18.5   111.
437.   --

Heap Sort   .512   1.37   4.52   10.9   33.2   76.1
166.

Merge Sort   .642   1.56   4.55   10.5   29.6   68.2
144.

(b) Average times in milliseconds

Table 7.2 Computing times for sorting methods. (Table prepared by Randall Istre)

Of the several sorting methods we have studied there is no one method that is best. Some methods are good for small n, others for large n. Insertion Sort is good when the file is already partially ordered. Because of the low overhead of the method it is also the best sorting method for "small" n. Merge Sort has the best worst case behavior but requires more storage than Heap Sort (though also an $O(n \log n)$ method it has slightly more overhead than Merge Sort). Quick Sort has the best average behavior but its worst case behavior is $O(n^2)$. The behavior of Radix Sort depends on the size of the keys and the choice of r.

These sorting methods have been programmed in FORTRAN and experiments conducted to determine the behavior of these methods. The results are summarized in Table 7.2. Table 7.2(a) gives the worst case sorting times for the various methods while table 7.2(b) gives the average times. Since the worst case and average times for radix sort are almost the same, only the average times have been reported. Table 7.2(b) contains two rows for Radix Sort. The first gives the times when an optimal radix r is used for the sort. The second row gives the times when the radix is fixed at 10. Both tables contain two rows for Insertion Sort. Comparing the two rows gives us an indication of the time saved by using a dummy key, $K(0)$, in the algorithm as opposed to explicitly checking for the left most record (i.e. $R(1)$). In a separate study it was determined that for average times, Quicksort is better than Insertion Sort only when $n \geq 23$ and that for worst case times Merge Sort is better than Insertion Sort only when $n \geq 25$. The exact cut off points will vary with different implementations. In practice, therefore, it would be worthwhile to couple Insertion Sort with other methods so that subfiles of size less than about 20 are sorted using Insertion Sort.

REFERENCES

A comprehensive discussion of sorting and searching may be found in:

The *Art of Computer Programming: Sorting and Searching*, by D. Knuth, vol. 3, Addison-Wesley, Reading, Massachusetts, 1973.

Two other useful references on sorting are:

Sorting and Sort Systems by H. Lorin, Addison-Wesley, Reading, Massachusetts, 1975.

Internal sorting methods illustrated with PL/1 *Programs* by R. Rich, Prentice-Hall, Englewood Cliffs, 1972.

For an in depth study of quicksort and stable merge sort see:

"Quicksort" by R. Sedgewick, STAN-CS-75-492, May 1975, Computer Science Department, Stanford University.

"Stable Sorting and Merging With Optimal Space and Time Bounds" by L. Pardo, STAN-CS-74-470, December 1974, Computer Science Department, Stanford University.

EXERCISES

1. Work through algorithms BINSRCH and FIBSRCH on an ordered file with keys (1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16) and determine the number of key comparisons made while

searching for the keys 2, 10 and 15. For FIBSRCH we need the three Fibonnaci numbers $F_5 = 5$, $F_6 = 8$, $F_7 = 13$.

2. [Count sort] About the simplest known sorting method arises from the observation that the position of a record in a sorted file depends on the number of records with smaller keys. Associated with each record there is a COUNT field used to determine the number of records which must precede this one in the sorted file. Write an algorithm to determine the COUNT of each record in an unordered file. Show that if the file has n records then all the COUNTs can be determined by making at most $n(n - 1)/2$ key comparisons.

3. The insertion of algorithm INSERT was carried out by (a) searching for the spot at which the insertion is to be made and (b) making the insertion. If as a result of the search it was decided that the insertion had to be made between R_i and R_{i+1}, then records R_{i+1}, ...,R_n were moved one space to locations R_{i+2}, ..., R_{n+1}. This was carried out in parallel with the search of (a). (a) can be sped up using the idea of BINSRCH or FIBSRCH. Write an INSERT algorithm incorporating one of these two searches.

4. Phase (b) (see exercise 3) can be sped up by maintaining the sorted file as a linked list. In this case the insertion can be made without any accompanying movement of the other records. However, now (a) must be carried out sequentially as before. Such an insertion scheme is known as list insertion. Write an algorithm for list insertion. Note that the insertion algorithms of exercises 3 and 4 can be used for a sort without making any changes in INSORT.

5. a) Show that algorithm QSORT takes $O(n^2)$ time when the input file is already in sorted order.

b) Why is $K_m \leq K_{n+1}$ required in QSORT?

6. (a) The quicksort algorithm QSORT presented in section 7.3 always fixes the position of the first record in the subfile currently being sorted. A better choice for this record is to choose the record with key value which is the median of the keys of the first, middle and last record in the subfile. Thus, using this median of three rule we correctly fix the position of the record R_i with $K_i = $ median $\{K_m, K_{(m+n)/2}, K_n\}$ i.e. K_i is the second largest key e.g. median $\{10,5,7\} = 7 = $ median $\{10,7,7\}$. Write a nonrecursive version of QSORT incorporating this median of three rule to determine the record whose position is to be fixed. Also, adopt the suggestion of section 7.8 and use Insertion Sort to sort subfiles of size less than 21. Show that this algorithm takes $O(n \log n)$ time on an already sorted file.

(b) Show that if smaller subfiles are sorted first then the recursion in algorithm QSORT can be simulated by a stack of depth $O(\log n)$.

7. Quicksort is an unstable sorting method. Give an example of an input file in which the order of records with equal keys is not preserved.

8. a) Write a nonrecursive merge sort algorithm using linked lists to represent sorte subfiles. Show that if *n* records each of size *m* are being sorted then the time required is only $O(n \log n)$ as no records are physically moved.

b) Use the rules of section 4.9 to automatically remove the recursion from the recursive version of merge sort.

c) Take the two algorithms written above and run them on random data for $n = 100, 200, ...,1000$ and compare their running times.

9. (i) Prove that algorithm MSORT is stable.

(ii) Heap sort is unstable. Give an example of an input file in which the order of records with equal keys is nto preserved.

10. In the 2-way merge sort scheme discussed in section 7.5 the sort was started with *n* sorted files each of size 1. Another approach would be to first make one pass over the data determining sequences of records that are in order and then using these as the initially sorted files. In this case, a left to right pass over the data of example 7.4 would result in the following partitioning of the data file into sorted subfiles. This would be followed by pairwise merging of the files until only one file remains.

□

Rewrite the 2-way merge sort algorithm to take into account the existing order in the records. How much time does this algorithm take on an initially sorted file? Note that the original algorithm took $O(n \log n)$ on such an input file. What is the worst case computing time of the new algorithm? How much additional space is needed? Use linked lists.

11. Does algorithm LRSORT result in a stable sort when used to sort numbers as in Example 7.6?

12. Write a sort algorithm to sort records $R_1, ...,R_n$ lexically on keys $(K^1, ...,K^r)$ for the case when the range of each key is much lager than *n*. In this case the bin sort scheme used in LRSORT to sort within each key becomes inefficienct (why?). What scheme would you use to sort within a key if we desired an algorithm with

a) good worst case behavior

b) good average behavior

c) *n* is small, say < 15.

13. If we have n records with integer keys in the range $[0, n^2)$, then they may be sorted in $O(n \log n)$ time using heap or merge sort. Radix sort on a single key, i.e., $d = 1$ and $r = n^2$ takes $O(n^2)$ time. Show how to interpret the keys as 2 subkeys so that radix sort will take only $O(n)$ time to sort n records. (Hint: each key, K_i, may be written as ☐ with ☐ and ☐ integers in the range $[0, n)$.)

14. Generalize the method of the previous exercise to the case of integer keys in the range $[0, n^p)$ obtaining $O(pn)$ sorting method.

15. Write the status of the following file F at the end of each phase of the following algorithms;

a) INSORT

b) QSORT

c) MSORT

d) HSORT

e) LSORT - radix 10

$F = (12, 2, 16, 30, 8, 28, 4, 10, 20, 6, 18)$

16. Write a table sort version of quicksort. Now during the sort, records are not physically moved. Instead, $T(i)$ is the index of the record that would have been in position i if records were physically moved around as in algorithm QSORT. To begin with $T(i) = i$, $1 \le i \le n$. At the end of the sort $T(i)$ is the index of the record that should be in the i'th position in the sorted file. So now algorithm TABLE of section 7.8 may be sued to rearrange the records into the sorted order specified by T. Note that this reduces the amount of data movement taking place when compared to QSORT for the case of large records.

17. Write an algorithm similar to algorithm TABLE to rearrange the records of a file if with each record we have a COUNT of the number of records preceding it in the sorted file (see Exercise 2).

18. Under what conditions would a MSD Radix sort be more efficient than an LSD Radix sort?

19. Assume you are given a list of five-letter English words and are faced with the problem of lisitng out these words in sequences such that the words in each sequence are anagrams, i.e., if x and y are in the same sequence, then word x is a permutation of word y. You are required to list out the fewest such sequences. With this restriction show that no word can appear in more than one sequence. How would you go about solving this problem?

20. Assume you are working in the census department of a small town where the number of records, about 3,000, is small enough to fit into the internal memory of a computer. All the people currently living in this town were born in the United States. There is one record for each person in this town. Each record contains

i) the state in which the person was born;

ii) county of birth;

iii) name of person.

How would you produce a list of all persons living in this town? The list is to be ordered by state. Within each state the persons are to be listed by their counties. The counties being arranged in aphabetical order. Within each county the names are also listed in alphabetical order. Justify any assumptions you may make.

Go to Chapter 8   Back to Table of Contents

CHAPTER 8: EXTERNAL SORTING

In this chapter we consider techniques to sort large files. The files are assumed to be so large that the whole file cannot be contained in the internal memory of a computer, making an internal sort impossible. Before discussing methods available for external sorting it is necessary first to study the characteristics of the external storage devices which can be used to accomplish the sort. External storage devices may broadly be categorized as either sequential access (e.g., tapes) or direct access (e.g., drums and disks). Section 8.1 presents a brief study of the properties of these devices. In sections 8.2 and 8.3 we study sorting methods which make the best use of these external devices.

8.1 STORAGE DEVICES

8.1.1 Magnetic Tapes

Magnetic tape devices for computer input/output are similar in principle to audio tape recorders. The data is recorded on magnetic tape approximately 1/2\" wide. The tape is wound around a spool. A new reel of tape is normally 2400 ft. long (with use, the length of tape in a reel tends to decrease because of frequent cutting off of lengths of the tape). Tracks run across the length of the tape, with a tape having typically 7 to 9 tracks across its width. Depending on the direction of magnetization, a spot on the track can represent either a 0 or a 1 (i.e., a bit of information). At any point along the length of the tape, the combination of bits on the tracks represents a character (e.g., *A-Z*, 0-9, +, :, ;, etc.). The number of bits that can be written per inch of track is referred to as the tape density. Examples of standard track densities are 800 and 1600 bpi (bits per inch). Since there are enough tracks across the width of the tape to represent a character, this density also gives the number of characters per inch of tape. Figure 8.1 illustrates this. With the conventions of the figure, the code for the first character on the tape is 10010111 while that for the third character is 00011100. If the tape is written using a density of 800 bpi then the length marked *x* in the figure is 3/800 inches.

Figure 8.1 Segment of a Magnetic Tape

Reading from a magnetic tape or writing onto one is done from a tape drive, as shown in figure 8.2. A tape drive consists of two spindles. On one of the spindles is mounted the source reel and on the other the take up reel. During forward reading or forward writing, the tape is pulled from the source reel across the read/write heads and onto the take up reel. Some tape drives also permit backward reading and writing of tapes; i.e., reading and writing can take place when tape is being moved from the take up to the source reel.

If characters are packed onto a tape at a density of 800 bpi, then a 2400 ft. tape would hold a little over 23 x 10^6 characters. A density of 1600 bpi would double this figure. However, it is necessary to block data on a tape since each read/write instruction transmits a whole block of information into/from memory. Since normally we would neither have enough space in memory for one full tape load nor would we wish to read the whole tape at once, the information on a tape will be grouped into several blocks. These blocks may be of a variable or fixed size. In between blocks of data is an interblock gap normally about 3/4 inches long. The interblock gap is long enough to permit the tape to accelerate from rest to the correct read/write speed before the beginning of the next block reaches the read/write heads. Figure 8.3 shows a segment of tape with blocked data.

Figure 8.2 A Tape Drive

Figure 8.3 Blocked Data on a Tape

In order to read a block from a tape one specifies the length of the block and also the address, *A*, in memory where the block is to be transmitted. The block of data is packed into the words $A, A + 1, A + 2, ...$ Similarly, in order to write a block of data onto tape one specifies the starting address in memory and the number of consecutive words to be written. These input and output areas in memory will be referred to as buffers. Usually the block size will correspond to the size of the input/output buffers set up in memory. We would like these blocks to be as large as possible for the following reasons:

(i) Between any pair of blocks there is an interblock gap of 3/4\". With a track density of 800 bpi, this space is long enough to write 600 characters. Using a block length of 1 character/block on a 2400 ft. tape would result in roughly 38,336 blocks or a total of 38,336 characters on the entire tape. Tape utilization is $1/601 < 0.17\%$. With 600 characters per block, half the tape would be made up of interblock gaps. In this case, the tape would have only about 11.5×10^6 characters of information on it, representing a 50% utilization of tape. Thus, the longer the blocks the more characters we can write onto the tape.

(ii) If the tape starts from rest when the input/output command is issued, then the time required to write a block of *n* characters onto the tape is $t_a + nt_w$ where t_a is the delay time and t_w the time to transmit one character from memory to tape. The delay time is the time needed to cross the interblock gap. If the tape starts from rest then t_a includes the time to accelerate to the correct tape speed. In this case t_a is larger than when the tape is already moving at the correct speed when a read/write command is issued. Assuming a tape speed of 150 inches per second during read/write and 800 bpi the time to read or write a character is 8.3×10^{-6}sec. The transmission rate is therefore 12×10^4 characters/second. The delay time t_a may typically be about 10 milliseconds. If the entire tape consisted of just one long block, then it could be read in thus effecting an average transmission rate of almost 12×10^4 charac/sec. If, on the other hand, each block were one character long, then the tape would have at most 38,336 characters or blocks. This would be the worst case and the read time would be about 6 min 24 sec or an average of 100 charac/sec. Note that if the read of the next block is initiated soon enough after the read of the present block, then the delay time would be reduced to 5 milliseconds, corresponding to the time needed to get across the interblock gap of 3/4\" at a tape speed of 150 inches per second. In this case the time to read 38,336 one character blocks would be 3 min 12 sec, corresponding to an average of about 200 charac/sec.

While large blocks are desirable from the standpoint of efficient tape usage as well as reduced input/output time, the amount of internal memory available for use as input/output buffers puts a limitation on block size.

Computer tape is the foremost example of a sequential access device. If the read head is positioned at the front of the tape and one wishes to read the information in a block 2000 ft. down the tape then it is necessary to forward space the tape the correct number of blocks. If now we wish to read the first block, the tape would have to be rewound 2000 ft. to the front before the first block could be read. Typical rewind times over 2400 ft. of tape could

be around one minute.

Unless otherwise stated we will make the following assumptions about our tape drives:

(i) Tapes can be written and read in the forward direction only.

(ii) The input/output channel of the computer is such as to permit the following three tasks to be carried out in parallel: writing onto one tape, reading from another and CPU operation.

(iii) If blocks 1, ...,i have been written on a tape, then the tape can be moved backwards block by block using a backspace command or moved to the first block via a rewind command. Overwriting block i- 1 with another block of the same size destroys the leading portion of block i. While this latter assumption is not true of all tape drives, it is characteristic of most of them.

8.1.2 Disk Storage

As an example of direct access external storage, we consider disks. As in the case of tape storage, we have here two distinct components: (1) the disk module (or simply disk or disk pack) on which information is stored (this corresponds to a reel of tape in the case of tape storage) and (2) the disk drive (corresponding to the tape drive) which performs the function of reading and writing information onto disks. Like tapes, disks can be removed from or mounted onto a disk drive. A disk pack consists of several platters that are similar to phonograph records. The number of platters per pack varies and typically is about 6. Figure 8.4 shows a disk pack with 6 platters. Each platter has two surfaces on which information can be recorded. The outer surfaces of the top and bottom platters are not used. This gives the disk of figure 8.4 a total of 10 surfaces on which information may be recorded. A disk drive consists of a spindle on which a disk may be mounted and a set of read/write heads. There is one read/write head for each surface. During a read/write the heads are held stationary over the position of the platter where the read/write is to be performed, while the disk itself rotates at high speeds (speeds of 2000-3000 rpm are fairly common). Thus, this device will read/write in concentric circles on each surface. The area that can be read from or written onto by a single stationary head is referred to as a *track*. Tracks are thus concentric circles, and each time the disk completes a revolution an entire track passes a read/write head. There may be from 100 to 1000 tracks on each surface of a platter. The collection of tracks simultaneously under a read/write head on the surfaces of all the platters is called a *cylinder*. Tracks are divided into sectors. A *sector* is the smallest addressable segment of a track. Information is recorded along the tracks of a surface in blocks. In order to use a disk, one must specify the track or cylinder number, the sector number which is the start of the block and also the surface. The read/write head assembly is first positioned to the right cylinder. Before read/write can commence, one has to wait for the right sector to come beneath the read/write head. Once this has happened, data transmission can take place. Hence, there are three factors contributing to input/output time for disks:

(i) *Seek time:* time taken to position the read/write heads to the correct cylinder. This will depend on the number of cylinders across which the heads have to move.

(ii) *Latency time:* time until the right sector of the track is under the read/write head.

(iii) *Transmission time:* time to transmit the block of data to/from the disk.

Figure 8.4 A Disk Drive with Disk Pack Mounted (Schematic).

Maximum seek times on a disk are around 1/10 sec. A typical revolution speed for disks is 2400 rpm. Hence the latency time is at most 1/40 sec (the time for one revolution of the disk). Transmission rates are typically between 10^5 characters/second and 5×10^5 characters/second. The number of characters that can be written onto a disk depends on the number of surfaces and tracks per surface. This figure ranges from about 10^7 characters for small disks to about 5×10^8 characters for a large disk.

8.2 SORTING WITH DISKS

The most popular method for sorting on external storage devices is merge sort. This method consists of essentially two distinct phases. First, segments of the input file are sorted using a good internal sort method. These sorted segments, known as *runs*, are written out onto external storage as they are generated. Second, the runs generated in phase one are merged together following the merge tree pattern of Example 7.4, until only one run is left. Because the merge algorithm of section 7.5 requires only the leading records of the two runs being merged to be present in memory at one time, it is possible to merge large runs together. It is more difficult to adapt the other methods considered in Chapter 7 to external sorting. Let us look at an example to illustrate the basic external merge sort process and analyze the various contributions to the overall computing time. A file containing 4500 records, $A_1, \ldots, A_{4500}$, is to be sorted using a computer with an internal memory capable of sorting at most 750 records. The input file is maintained on disk and has a block length of 250 records. We have available another disk which may be used as a scratch pad. The input disk is not to be written on. One way to accomplish the sort using the general procedure outlined above is to:

(i) Internally sort three blocks at a time (i.e., 750 records) to obtain six runs R_1-R_6. A method such as heapsort or quicksort could be used. These six runs are written out onto the scratch disk (figure 8.5).

Figure 8.5 Blocked Runs Obtained After Internal Sorting

(ii) Set aside three blocks of internal memory, each capable of holding 250 records. Two of these blocks will be used as input buffers and the third as an output buffer. Merge runs R_1 and R_2. This is carried out by first reading one block of each of these runs into input buffers. Blocks of runs are merged from the input buffers into the output buffer. When the output buffer gets full, it is written out onto disk. If an input buffer gets empty, it is refilled with another block from the same run. After runs R_1 and R_2 have been merged, R_3 and R_4 and finally R_5 and R_6 are merged. The result of this pass is 3 runs, each containing 1500 sorted records of 6 blocks. Two of these runs are now merged using the input/output buffers set up as above to obtain a run of size 3000. Finally, this run is merged with the remaining run of size 1500 to obtain the desired sorted file (figure 8.6).

Let us now analyze the method described above to see how much time is required to sort these 4500 records. The analysis will use the following notation:

t_s = maximum seek time

t_l = maximum latency time

t_{rw} = time to read or write one block of 250 records

$t_{IO} = t_s + t_l + t_{rw}$

t_{IS} = time to internally sort 750 records

$n\ t_m$ = time to merge n records from input buffers to the output buffer

We shall assume that each time a block is read from or written onto the disk, the maximum seek and latency times are experienced. While this is not true in general, it will simplify the analysis. The computing time for the various operations are:

Figure 8.6 Merging the 6 runs

```
                  Operation                              Time

                 ----------                              ----

1) read 18 blocks of input, 18t_IO, internally sort, 6t_IS

   write 18 blocks, 18t_IO                          36 t_IO + 6 t_IS

2) merge runs 1-6 in pairs                          36 t_IO + 4500 t_m

3) merge 2 runs of 1500 records each, 12 blocks     24 t_IO + 3000 t_m

4) merge one run of 3000 records with one run of 1500

   records                                          36 t_IO + 4500 t_m

                                         --------------------------

                 Total Time      132 t_IO + 12000 t_m + 6 t_IS
```

Note that the contribution of seek time can be reduced by writing blocks on the same cylinder or on adjacent cylinders. A close look at the final computing time indicates that it depends chiefly on the number of passes made over the data. In addition to the initial input pass made over the data for the internal sort, the merging of the runs requires 2-2/3 passes over the data (one pass to merge 6 runs of length 750 records, two thirds of a pass to merge two runs of length 1500 and one pass to merge one run of length 3000 and one of length 1500). Since one full pass covers 18 blocks, the input and output time is 2 X (2-2/3 + 1) X 18 t_{IO} = 132 t_{IO}. The leading factor of 2 appears because each record that is read is also written out again. The merge time is 2-2/3 X 4500 t_m = 12,000 t_m. Because of this close relationship between the overall computing time and the number of passes made over the data, future analysis will be concerned mainly with counting the number of passes being made. Another point to note regarding

the above sort is that no attempt was made to use the computer's ability to carry out input/output and CPU operation in parallel and thus overlap some of the time. In the ideal situation we would overlap almost all the input/output time with CPU processing so that the real time would be approximately $132\ t_{IO}\ \square\ 12000\ t_m + 6\ t_{IS}$.

If we had two disks we could write on one while reading from the other and merging buffer loads already in memory all at the same time. In this case a proper choice of buffer lengths and buffer handling schemes would result in a time of almost $66\ t_{IO}$. This parallelism is an important consideration when the sorting is being carried out in a non-multi-programming environment. In this situation unless input/output and CPU processing is going on in parallel, the CPU is idle during input/output. In a multi-programming environment, however, the need for the sorting program to carry out input/output and CPU processing in parallel may not be so critical since the CPU can be busy working on another program (if there are other programs in the system at that time), while the sort program waits for the completion of its input/output. Indeed, in many multi-programming environments it may not even be possible to achieve parallel input, output and internal computing because of the structure of the operating system.

The remainder of this section will concern itself with: (1) reduction of the number of passes being made over the data and (2) efficient utilization of program buffers so that input, output and CPU processing is overlapped as much as possible. We shall assume that runs have already been created from the input file using some internal sort scheme. Later, we investigate a method for obtaining runs that are on the average about 2 times as long as those obtainable by the methods discussed in Chapter 7.

8.2.1 k-Way Merging

The 2-way merge algorithm of Section 7.5 is almost identical to the merge procedure just described (figure 8.6). In general, if we started with m runs, then the merge tree corresponding to figure 8.6 would have $\lceil \log_2 m \rceil + 1$ levels for a total of $\lceil \log_2 m \rceil$ passes over the data file. The number of passes over the data can be reduced by using a higher order merge, i.e., k-way merge for $k \geq 2$. In this case we would simultaneously merge k runs together. Figure 8.7 illustrates a 4-way merge on 16 runs. The number of passes over the data is now 2, versus 4 passes in the case of a 2-way merge. In general, a k-way merge on m runs requires at most $\lceil \log_k m \rceil$ passes over the data. Thus, the input/output time may be reduced by using a higher order merge. The use of a higher order merge, however, has some other effects on the sort. To begin with, k-runs of size $S_1, S_2, S_3, ...,S_k$ can no longer be merged internally in $\square$ time. In a k-way merge, as in a 2-way merge, the next record to be output is the one with the smallest key. The smallest has now to be found from k possibilities and it could be the leading record in any of the k-runs. The most direct way to merge k-runs would be to make $k - 1$ comparisons to determine the next record to output. The computing time for this would be $\square$. Since $\log_k m$ passes are being made, the total number of key comparisons being made is $n(k - 1) \log_k m = n(k - 1) \log_2 m / \log_2 k$ where n is the number of records in the file. Hence, $(k - 1)/\log_2 k$ is the factor by which the number of key comparisons increases. As k increases, the reduction in input/output time will be overweighed by the resulting increase in CPU time needed to perform the k-way merge. For large k (say, $k \geq$ 6) we can achieve a significant reduction in the number of comparisons needed to find the next smallest element by using the idea of a selection tree. A *selection tree* is a binary tree where each node represents the smaller of its two children. Thus, the root node represents the smallest node in the tree. Figure 8.9 illustrates a selection tree for an 8-way merge of 8-runs.

Figure 8.7 A 4-way Merge on 16 Runs

Figure 8.8 A k-Way Merge

The construction of this selection tree may be compared to the playing of a tournament in which the winner is the record with the smaller key. Then, each nonleaf node in the tree represents the winner of a tournament and the root node represents the overall winner or the smallest key. A leaf node here represents the first record in the corresponding run. Since the records being sorted are generally large, each node will contain only a pointer to the record it represents. Thus, the root node contains a pointer to the first record in run 4. The selection tree may be represented using the sequential allocation scheme for binary trees discussed in section 5.3. The number above each node in figure 8.9 represents the address of the node in this sequential representation. The record pointed to by the root has the smallest key and so may be output. Now, the next record from run 4 enters the selection tree. It has a key value of 15. To restructure the tree, the tournament has to be replayed only along the path from node 11 to the root. Thus, the winner from nodes 10 and 11 is again node 11 (15 < 20). The winner from nodes 4 and 5 is node 4 (9 < 15). The winner from 2 and 3 is node 3 (8 < 9). The new tree is shown in figure 8.10. The tournament is played between sibling nodes and the result put in the parent node. Lemma 5.3 may be used to compute the address of sibling and parent nodes efficiently. After each comparison the next takes place one higher level in the tree. The number of levels in the tree is $\lceil \log_2 k \rceil + 1$. So, the time to restructure the tree is $O(\log_2 k)$. The tree has to be restructured each time a record is merged into the output file. Hence, the time required to merge all n records is $O(n \log_2 k)$. The time required to set up the selection tree the first time is $O(k)$. Hence, the total time needed per level of the merge tree of figure 8.8 is $O(n \log_2 k)$. Since the number of levels in this tree is $O(\log_k m)$, the asymptotic internal processing time becomes $O(n \log_2 k \log_k m) = O(n \log_2 m)$. The internal processing time is independent of k.

Figure 8.9 Selection tree for 8-way merge showing the first three keys in each of the 8 runs.

Note, however, that the internal processing time will be increased slightly because of the increased overhead associated with maintaining the tree. This overhead may be reduced somewhat if each node represents the loser of the tournament rather than the winner. After the record with smallest key is output, the selection tree of figure 8.9 is to be restructured. Since the record with the smallest key value is in run 4, this restructuring involves inserting the next record from this run into the tree. The next record has key value 15. Tournaments are played between sibling nodes along the path from node 11 to the root. Since these sibling nodes represent the losers of tournaments played earlier, we could simplify the restructuring process by placing in each nonleaf node a pointer to the record that loses the tournament rather than to the winner of the tournament. A tournament tree in which each nonleaf node retains a pointer to the loser is called a *tree of losers.* Figure 8.11 shows the tree of losers corresponding to the selection tree of figure 8.9. For convenience, each node contains the key value of a record rather than a pointer to the record represented. The leaf nodes represent the first record in each run. An additional node, node 0, has been added to represent the overall winner of the tournament. Following the output of the overall winner, the tree is restructured by playing tournaments along the path from node 11 to node 1. The records with which these tournaments are to be played are readily available from the parent nodes. We shall see more of loser trees when we study run generation in section 8.2.3.

Figure 8.10 Selection tree of Figure 8.9 after one record has been output and tree restructured. Nodes that were changed are marked by ☐.

Figure 8.11 Tree of Losers Corresponding to Figure 8.9

In going to a higher order merge, we save on the amount of input/output being carried out. There is no significant loss in internal processing speed. Even though the internal processing time is relatively insensitive to the order of the merge, the decrease in input/output time is not as much as indicated by the reduction to $\log_k m$ passes. This is so because the number of input buffers needed to carry out a k-way merge increases with k. Though $k + 1$ buffers are sufficient, we shall see in section 8.2.2 that the use of $2k + 2$ buffers is more desirable. Since the internal memory available is fixed and independent of k, the buffer size must be reduced as k increases. This in turn implies a reduction in the block size on disk. With the reduced block size each pass over the data results in a greater number of blocks being written or read. This represents a potential increase in input/output time from the increased contribution of seek and latency times involved in reading a block of data. Hence, beyond a certain k value the input/output time would actually increase despite the decrease in the number of passes being made. The optimal value for k clearly depends on disk parameters and the amount of internal memory available for buffers (exercise 3).

8.2.2 Buffer Handling for Parallel Operation

If k runs are being merged together by a k-way merge, then we clearly need at least k input buffers and one output buffer to carry out the merge. This, however, is not enough if input, output and internal merging are to be carried out in parallel. For instance, while the output buffer is being written out, internal merging has to be halted since there is no place to collect the merged records. This can be easily overcome through the use of two output buffers. While one is being written out, records are merged into the second. If buffer sizes are chosen correctly, then the time to output one buffer would be the same as the CPU time needed to fill the second buffer. With only k input buffers, internal merging will have to be held up whenever one of these input buffers becomes empty and another block from the corresponding run is being read in. This input delay can also be avoided if we have $2k$ input buffers. These $2k$ input buffers have to be cleverly used in order to avoid reaching a situation in which processing has to be held up because of lack of input records from any one run. Simply assigning two buffers per run does not solve the problem. To see this, consider the following example.

Example 8.1: Assume that a two way merge is being carried out using four input buffers, $IN(i)$, $1 \leq i \leq 4$, and two output buffers, $OU(1)$ and $OU(2)$. Each buffer is capable of holding two records. The first few records of run 1 have key value 1, 3, 5, 7, 8, 9. The first few records of run 2 have key value 2, 4, 6, 15, 20, 25. Buffers $IN(1)$ and $IN(3)$ are assigned to run 1. The remaining two input buffers are assigned to run 2. We start the merging by reading in one buffer load from each of the two runs. At this time the buffers have the configuration of figure 8.12(a). Now runs 1 and 2 are merged using records from $IN(1)$ and $IN(2)$. In parallel with this the next buffer load from run 1 is input. If we assume that buffer lengths have been chosen such that the times to input, output and generate an output buffer are all the same then when $OU(1)$ is full we have the situation of figure 8.12(b). Next, we simultaneously output $OU(1)$, input into $IN(4)$ from run 2 and merge into $OU(2)$. When $OU(2)$ is full we are in the situation of figure 8.12(c). Continuing in this way we reach the configuration of figure 8.12(e). We now begin to output $OU(2)$, input from run

1 into *IN*(3) and merge into *OU*(1). During the merge, all records from run 1 get exhausted before *OU*(1) gets full. The generation of merged output must now be delayed until the inputting of another buffer load from run 1 is completed!

Example 8.1 makes it clear that if $2k$ input buffers are to suffice then we cannot assign two buffers per run. Instead, the buffers must be floating in the sense that an individual buffer may be assigned to any run depending upon need. In the buffer assignment strategy we shall describe, for each run there will at any time be, at least one input buffer containing records from that run. The remaining buffers will be filled on a priority basis. I.e., the run for which the k-way merging algorithm will run out of records first is the one from which the next buffer will be filled. One may easily predict which run's records will be exhausted first by simply comparing the keys of the last record read from each of the k runs. The smallest such key determines this run. We shall assume that in the case of equal keys the merge process first merges the record from the run with least index. This means that if the key of the last record read from run i is equal to the key of the last record read from run j, and $i < j$, then the records read from i will be exhausted before those from j. So, it is possible that at any one time we might have more than two bufferloads from a given run and only one partially full buffer from another run. All bufferloads from the same run are queued together. Before formally presenting the algorithm for buffer utilization, we make the following assumptions about the parallel processing capabilities of the computer system available:

(i) We have two disk drives and the input/output channel is such that it is possible simultaneously to read from one disk and write onto the other.

(ii) While data transmission is taking place between an input/output device and a block of memory, the CPU cannot make references to that same block of memory. Thus, it is not possible to start filling the front of an output buffer while it is being written out. If this were possible, then by coordinating the transmission and merging rate only one output buffer would be needed. By the time the first record for the new output block was determined, the first record of the previous output block would have been written out.

(iii) To simplify the discussion we assume that input and output buffers are to be the same size.

☐

☐

Figure 8.12 Example showing that two fixed buffers per run are not enough for continued parallel operation

Keeping these assumptions in mind, we first formally state the algorithm obtained using the strategy outlined earlier and then illustrate its working through an example. Our algorithm merges k-runs, ☐, using a k-way merge. $2k$ input buffers and 2 output buffers are used. Each buffer is a contiguous block of memory. Input buffers are queued in k queues, one queue for each run. It is assumed that each input/output buffer is long enough to hold one block of records. Empty buffers are stacked with AV pointing to the top buffer in this stack. The stack is a linked list. The following variables are made use of:

```
IN (i) ... input buffers, 1 ≤ i ≤ 2k

OUT (i) ... output buffers, 0 ≤ i ≤ 1
```

```
FRONT (i) ... pointer to first buffer in queue for run i, 1 ≤ i ≤ k

  END (i) ... end of queue for i-th run, 1 ≤ i ≤ k

 LINK (i) ... link field for i-th input buffer

             in a queue or for buffer in stack 1 ≤ i ≤ 2k

 LAST (i) ... value of key of last record read

             from run i, 1 ≤ i ≤ k

      OU ... buffer currently used for output.
```

The algorithm also assumes that the end of each run has a sentinel record with a very large key, say $+\infty$. If block lengths and hence buffer lengths are chosen such that the time to merge one output buffer load equals the time to read a block then almost all input, output and computation will be carried out in parallel. It is also assumed that in the case of equal keys the *k*-way merge algorithm first outputs the record from the run with smallest index.

```
     procedure BUFFERING

1    for i <-- 1 to k do      //input a block from each run//

2      input first block of run i into IN(i)

3    end

4    while input not complete do end      //wait//

5    for i <-- 1 to k do      //initialize queues and free buffers//

6      FRONT(i) <-- END(i) <-- i

7      LAST(i) <-- last key in buffer IN(i)

8      LINK(k + i) <-- k + i + 1      //stack free buffer//

9    end

10   LINK(2k)  <-- 0; AV <-- k + 1; OU <-- 0

     //first queue exhausted is the one whose last key read is smallest//

11   find j such that LAST(j) =  min  {LAST(i)}
```

$$1 \leq i \leq k$$

```
12  l <-- AV; AV <-- LINK(AV)        //get next free buffer//
```

13 **if** LAST(j) $\neq$ + ∞ **then** [begin to read next block for run j into

buffer IN(l)]

```
14  repeat        //KWAYMERGE merges records from the k buffers
```

FRONT(i) into output buffer OU until it is full.

If an input buffer becomes empty before OU is filled, the

next buffer in the queue for this run is used and the empty

buffer is stacked or last key = + ∞ //

```
15    call KWAYMERGE
```

```
16    while input/output not complete do      //wait loop//
```

```
17    end
```

if LAST(j) $\neq$ + ∞ **then**

```
18    [LINK(END(j)) <-- l; END(j) <-- l; LAST(j) <-- last key read
```

//queue new block//

```
19    find j such that LAST(j) =  min {LAST(i)}
```

$$1 \leq i \leq k$$

```
20    l <-- AV; AV <-- LINK(AV)]      //get next free buffer//
```

```
21    last-key-merged <-- last key in OUT(OU)
```

22 **if** LAST(j) $\neq$ + ∞ **then** [begin to write OUT(OU) and read next block of

run j into IN(l)]

23   **else** [begin to write OUT(OU)]

```
24      OU <-- 1 - OU

25      until last-key-merged = +∞

26      while output incomplete do      //wait loop//

27      end

28   end BUFFERING
```

Notes: 1) For large k, determination of the queue that will exhaust first can be made in $\log_2 k$ comparisons by setting up a selection tree for LAST(i), $1 \leq i \leq k$, rather than making $k - 1$ comparisons each time a buffer load is to be read in. The change in computing time will not be significant, since this queue selection represents only a very small fraction of the total time taken by the algorithm.

2) For large k the algorithm KWAYMERGE uses a selection tree as discussed in section 8.2.1.

3) All input/output except for the initial k blocks that are read and the last block output is done concurrently with computing. Since after k runs have been merged we would probably begin to merge another set of k runs, the input for the next set can commence during the final merge stages of the present set of runs. I.e., when LAST(j) = $+^\infty$ we begin reading one by one the first blocks from each of the next set of k runs to be merged. In this case, over the entire sorting of a file, the only time that is not overlapped with the internal merging time is the time for the first k blocks of input and that for the last block of output.

4) The algorithm assumes that all blocks are of the same length. This may require inserting a few dummy records into the last block of each run following the sentinel record $+^\infty$.

Example 8.2: To illustrate the working of the above algorithm, let us trace through it while it performs a three-way merge on the following three runs:

Each run consists of four blocks of two records each; the last key in the fourth block of each of these three runs is $+^\infty$. We have six input buffers IN(i), $1 \leq i \leq 6$, and 2 output buffers OUT(0) and OUT(1). The diagram on page 404 shows the status of the input buffer queues, the run from which the next block is being read and the output buffer being ouput at the beginning of each iteration of the **repeat-until** of the buffering algorithm (lines 14-25).

image 404_a_a.gif not available.

image 404_a_b.gif not available.

From line 5 it is evident that during the k-way merge the test for "output buffer full?" should be carried out before the test "input buffer empty?" as the next input buffer for that run may not have been read in yet and so there would be no next buffer in that queue. In lines 3 and 4 all 6 input buffers are in use and the stack of free buffers is empty.

We end our discussion of buffer handling by proving that the algorithm BUFFERING works. This is stated formally in Theorem 8.1.

Theorem 8.1: The following is true for algorithm BUFFERING:

(i) There is always a buffer available in which to begin reading the next block; and

(ii) during the *k*-way merge the next block in the queue has been read in by the time it is needed.

Proof: (i) Each time we get to line 20 of the algorithm there are at most $k + 1$ buffer loads in memory, one of these being in an output buffer. For each queue there can be at most one buffer that is partially full. If no buffer is available for the next read, then the remaining *k* buffers must be full. This means that all the *k* partially full buffers are empty (as otherwise there will be more than $k + 1$ buffer loads in memory). From the way the merge is set up, only one buffer can be both unavailable and empty. This may happen only if the output buffer gets full exactly when one input buffer becomes empty. But $k > 1$ contradicts this. So, there is always at least one buffer available when line 20 is being executed.

(ii) Assume this is false. Let run R_i be the one whose queue becomes empty during the KWAYMERGE. We may assume that the last key merged was not the sentinel key $+^\infty$ since otherwise KWAYMERGE would terminate the search rather then get another buffer for R_i. This means that there are more blocks of records for run R_i on the input file and $LAST(i) \neq +^\infty$. Consequently, up to this time whenever a block was output another was simultaneously read in (see line 22). Input/output therefore proceeded at the same rate and the number of available blocks of data is always *k*. An additional block is being read in but it does not get queued until line 18. Since the queue for R_i has become empty first, the selection rule for the next run to read from ensures that there is at most one block of records for each of the remaining $k - 1$ runs. Furthermore, the output buffer cannot be full at this time as this condition is tested for before the input buffer empty condition. Thus there are fewer than *k* blocks of data in memory. This contradicts our earlier assertion that there must be exactly *k* such blocks of data.

8.2.3 Run Generation

Using conventional internal sorting methods such as those of Chapter 7 it is possible to generate runs that are only as large as the number of records that can be held in internal memory at one time. Using a tree of losers it is possible to do better than this. In fact, the algorithm we shall present will on the average generate runs that are twice as long as obtainable by conventional methods. This algorithm was devised by Walters, Painter and Zalk. In addition to being capable of generating longer runs, this algorithm will allow for parallel input, output and internal processing. For almost all the internal sort methods discussed in Chapter 7, this parallelism is not possible. Heap sort is an exception to this. In describing the run generation algorithm, we shall not dwell too much upon the input/output buffering needed. It will be assumed that input/output buffers have been appropriately set up for maximum overlapping of input, output and internal processing. Wherever in the run generation algorithm there is an input/output instruction, it will be assumed that the operation takes place through the input/output buffers. We shall assume that there is enough space to construct a tree of losers for *k* records, $R(i)$, $0 \leq i < k$. This will require a loser tree with *k* nodes numbered 0 to $k - 1$. Each node, *i*, in this tree will have one field $L(i)$. $L(i)$, $1 \leq i < k$ represents the loser of the tournament played at node *i*. Node 0 represents the overall winner of the tournament. This node will not be explicitly present in the algorithm. Each of the *k* record positions $R(i)$, has a run number field $RN(i)$, $0 \leq i < k$ associated with it. This field will enable us to determine whether or not $R(i)$ can be output as part of the run currently being generated. Whenever the tournament winner is output, a new record (if there is one) is input and the

tournament replayed as discussed in section 8.2.1. Algorithm RUNS is simply an implementation of the loser tree strategy discussed earlier. The variables used in this algorithm have the following significance:

```
  R(i), 0 ≤ i < k ... the k records in the tournament tree

KEY(i), 0 ≤ i < k ... key value of record R(i)

  L(i), 0 < i < k ... loser of the tournament played at node i

RN(i), 0 <= i < k ... the run number to which R(i) belongs

                RC ... run number of current run

                 Q ... overall tournament winner

                RQ ... run number for R(Q)

              RMAX ... number of runs that will be generated

          LAST__KEY ... key value of last record output
```

The loop of lines 5-25 repeatedly plays the tournament outputting records. The only interesting thing about this algorithm is the way in which the tree of losers is initialized. This is done in lines 1-4 by setting up a fictitious run numbered 0. Thus, we have $RN(i) = 0$ for each of the k records $R(i)$. Since all but one of the records must be a loser exactly once, the initialization of $L(i)$ i sets up a loser tree with R(0) the winner. With this initialization the loop of lines 5-26 correctly sets up the loser tree for run 1. The test of line 10 suppresses the output of these k fictitious records making up run 0. The variable LAST__KEY is made use of in line 13 to determine whether or not the new record input, R(Q), can be output as part of the current run. If KEY(Q) < LAST__KEY then $R(Q)$ cannot be output as part of the current run RC as a record with larger key value has already been output in this run. When the tree is being readjusted (lines 18-24), a record with lower run number wins over one with a higher run number. When run numbers are equal, the record with lower key value wins. This ensures that records come out of the tree in nondecreasing order of their run numbers. Within the same run, records come out of the tree in nondecreasing order of their key values . RMAX is used to terminate the algorithm. In line 11, when we run out of input, a record with run number RMAX + 1 is introduced. When this record is ready for output, the algorithm terminates in line 8. One may readily verify that when the input file is already sorted, only one run is generated. On the average, the run size is almost $2k$. The time required to generate all the runs for an n run file is $O(n \log k)$ as it takes $O(\log k)$ time to adjust the loser tree each time a record is output. The algorithm may be speeded slightly by explicitly initializing the loser tree using the first k records of the input file rather than k fictitious records as in lines 1-4. In this case the conditional of line 10 may be removed as there will no longer be a need to suppress output of certain records.

8.3 SORTING WITH TAPES

Sorting on tapes is carried out using the same basic steps as sorting on disks. Sections of the input file are internally sorted into runs which are written out onto tape. These runs are repeatedly merged together

procedure *RUNS*

//generate runs using a tree of losers//

```
1       for i ⬜ 1 to k - 1 do        //set up fictitious run 0 to initialize
tree//

2           RN(i) ⬜ 0; L(i) ⬜ i; KEY(i) ⬜ 0

3       end

4       Q ⬜ RQ ⬜ RC ⬜ RMAX ⬜ RN(0) ⬜ 0; LAST__KEY ⬜ ∞

5       loop      //output runs//

6          if RQ ≠ RC then [//end of run//

7                            if RC ≠ 0 then output end of run marker

8                            if RQ > RMAX then stop

9                                          else RC ⬜ RQ]

//output record R(Q) if not fictitious//

10         if RQ ≠ 0 then [output R(Q); LAST__KEY ⬜ KEY (Q)]

//input new record into tree//

11         if no more input then [RQ ⬜ RMAX + 1; RN (Q) ⬜ RQ]

12                            else [input a new record into R (Q)

13                                  if KEY (Q) < LAST__KEY

14                                    then [//new record belongs to next
run//

15                                        RQ ⬜ RQ + 1
```

```
16                                    RN (Q) □ RQ

RMAX □ RQ]

17                        else [RN (Q) □ RC]]

//adjust losers//

18       T □ ⌊(k + Q)/2⌋           //T is parent of Q//

19       while T ≠ 0 do

20         if RN (L(T)) < RQ or (RN(L(T)) = RQ and KEY(L(T)) <

KEY(Q))

21           then [TEMP □ Q; Q □ L(T); □ L(T) □ TEMP

//T is the winner//

22                RQ □ RN(Q)]

23       T □ ⌊T/2⌋      //move up tree//

24     end

25   forever

26 end RUNS
```

until there is only one run. The difference between disk and tape sorting lies essentially in the manner in which runs are maintained on the external storage media. In the case of disks, the seek and latency times are relatively insensitive to the specific location on the disk of the next block to be read with respect to the current position of the read/write heads. (The maximum seek and latency times were about 1/10 sec and 1/40 sec. respectively.) This is not true for tapes. Tapes are sequential access. If the read/write head is currently positioned at the front of the tape and the next block to be read is near the end of the tape (say ~ 2400 feet away), then the seek time, i.e., the time to advance the correct block to the read/write head, is almost 3 min 2 sec. (assuming a tape speed of 150 inches/second). This very high maximum seek time for tapes makes it essential that blocks on tape be arranged in such an order that they would be read in sequentially during the *k*-way merge of runs. Hence, while in the previous section we did not concern ourselves with the relative position of the blocks on disk, in this section, the distribution of blocks and runs will be our primary concern. We shall study two distribution schemes. Before doing this, let us look at an example to see the different factors involved in a tape sort.

Example 8.3: A file containing 4,500 records, R_1, ..., R_{4500} is to be sorted using a computer with an internal memory large enough to hold only 800 records. The available external storage consists of 4 tapes, $T1$, $T2$, $T3$ and $T4$. Each block on the input tape contains 250 records. The input tape is not to be destroyed during the sort. Initially, this tape is on one of the 4 available tape drives and so has to be dismounted and replaced by a work tape after all input records have been read. Let us look at the various steps involved in sorting this file. In order to simplify the discussion, we shall assume that the initial run generation is carried out using a conventional internal sort such as quicksort. In this case 3 blocks of input can be read in at a time, sorted and output as a run. We shall also assume that no parallelism in input, output and CPU processing is possible. In this case we shall use only one output buffer and two input buffers. This requires only 750 record spaces.

☐

☐

☐

t_{rw} = time to read or write on block of 250 records onto tape starting at present position of read/write head

t_{rew} = time to rewind tape over a length correponding to 1 block

nt_m = time to merge n records from input buffers to output buffer using a 2-way merge

☐ = delay cause by having to wait for T4 to be mounted in case we are ready to use T4 in step 4 before the completion of this tape mount.

The above computing time analysis assumes that no operations are carried out in parallel. The analysis could be carried out further as in the case of disks to show the dependence of the sort time on the number of passes made over the data.

8.3.1 Balanced Merge Sorts

Example 8.3 performed a 2-way merge of the runs. As in the case of disk sorting, the computing time here too depends essentially on the number of passes being made over the data. Use of a higher order merge results in a decrease in the number of passes being made without significantly changing the internal merge time. Hence, we would like to use as high an order merge as possible. In the case of disks the order of merge was limited essentially by the amount of internal memory available for use as input/output buffers. A k-way merge required the use of $2k + 2$ buffers. While this is still true of tapes, another probably more severe restriction on the merge order k is the number of tapes available. In order to avoid excessive tape seek times, it is necessary that runs being merged together be on different tapes. Thus, a k-way merge requires at least k-tapes for use as input tapes during the merge.

In addition, another tape is needed for the output generated during this merge. Hence, at least $k + 1$ tapes must be available for a k-way tape merge (recall that in the case of a disk, one disk was enough for a k-way merge though two were needed to overlap input and output). Using $k + 1$ tapes to perform a k-way merge requires an additional pass over the output tape to redistribute the runs onto k-tapes for the next level of merges (see the merge tree of

figure 8.8). This redistribution pass can be avoided through the use of 2*k* tapes. During the *k*-way merge, *k* of these tapes are used as input tapes and the remaining *k* as output tapes. At the next merge level, the role of input-output tapes is interchanged as in example 8.3, where in step 4, *T*1 and *T*2 are used as input tapes and *T*3 and *T*4 as output tapes while in step 6, *T*3 and *T*4 are the input tapes and *T*1 and *T*2 the output tapes (though there is no output for *T*2). These two approaches to a *k*-way merge are now examined. Algoritms *M*1 and *M*2 perform a *k*-way merge with the *k* + 1 tapes strategy while algorithm *M*3 performs a *k*-way merge using the 2*k* tapes strategy.

```
procedure M1

//Sort a file of records from a given input tape using a k-way

merge given tapes T₁, ...,Tₖ₊₁ are available for the sort.//

1    Create runs from the input file distributing them evenly over

tapes T₁, ...,Tₖ

2    Rewind T₁, ...,Tₖ and also the input tape

3    if there is only 1 run then return       //sorted file on T₁//

4    replace input tape by Tₖ₊₁

5    loop        //repeatedly merge onto Tₖ₊₁ and redistribute back

onto T₁, ...,Tₖ//

6       merge runs from T₁, ...,Tₖ onto Tₖ₊₁

7       rewind T₁, ....,Tₖ₊₁

8       if number of runs on Tₖ₊₁ = 1 then return //output on

Tₖ₊₁//

9       evenly distribute from Tₖ₊₁ onto T₁, ...,Tₖ

10       rewind T₁, ...,Tₖ₊₁

11    forever

12 end M1
```

Analysis of Algorithm M1

To simplify the analysis we assume that the number of runs generated, m, is a power of k. Line 1 involves one pass over the entire file. One pass includes both reading and writing. In lines 5-11 the number of passes is $\log_k m$ merge passes and $\log_k m - 1$ redistribution passes. So, the total number of passes being made over the file is $2\log_k m$. If the time to rewind the entire input tape is t_{rew}, then the non-overlapping rewind time is roughly $2\log_k m\, t_{rew}$.

A somewhat more clever way to tackle the problem is to rotate the output tape, i.e., tapes are used as output tapes in the cyclic order, $k + 1,1,2, ...,k$. When this is done, the redistribution from the output tape makes less than a full pass over the file. Algorithm $M2$ formally describes the process. For simplicity the analysis of $M2$ assumes that m is a power of k.

Analysis of Algorithm M2

The only difference between algorithms $M2$ and $M1$ is the redistributing

```
procedure M2

//same function as M1//

1    Create runs from the input tape, distributing them evenly over

tapes T₁, ...,Tₖ.

2    rewind T₁, ...,Tₖ and also the input tape

3    if there is only 1 run then return      //sorted file is on T₁//

4    replace input tape by Tₖ₊₁

5    i ▢ k + 1        //i is index of output tape//

6    loop

7      merge runs from the k tapes Tⱼ, 1 ≤ j ≤ k + 1 and j ≠ i onto

Tᵢ

8      rewind tapes T₁, ...,Tₖ₊₁

9      if number of runs on Tᵢ = 1 then return      //output on Tᵢ//

10       evenly distribute (k - 1)/k of the runs on Tᵢ onto tapes Tᵢ, 1 ≤
```

```
j ≤ k +1 and j ≠ i and j ≠ i mod(k + 1) + 1

11     rewind tapes T_j, 1 ≤ j ≤ k + 1 and j ≠ i

12      i □ i mod(k + 1) + 1

13   forever

14 end M2
```

time. Once again the redistributing is done $\log_k m - 1$ times. m is the number of runs generated in line 1. But, now each redistributing pass reads and writes only $(k - 1)/k$ of the file. Hence, the effective number of passes made over the data is $(2 - 1/k)\log_k m + 1/k$. For two-way merging on three tapes this means that algorithm $M2$ will make $3/2 \log_k m + 1/2$ passes while $M1$ will make $2 \log_k m$. If t_{rew} is the rewind time then the non-overlappable rewind time for $M2$ is at most $(1 + 1/k) (\log_k m) t_{rew} + (1 - 1/k) t_{rew}$ as line 11 rewinds only $1/k$ of the file. Instead of distributing runs as in line 10 we could write the first m/k runs onto one tape, begin rewind, write the next m/k runs onto the second tape, begin rewind, etc. In this case we can begin filling input buffers for the next merge level (line 7) while some of the tapes are still rewinding. This is so because the first few tapes written on would have completed rewinding before the distribution phase is complete (for k > 2).

In case a k-way merge is made using $2k$ tapes, no redistribution is needed and so the number of passes being made over the file is only $\log_k m + 1$. This means that if we have $2k + 1$ tapes available, then a $2k$-way merge will make $(2 - 1/(2k))\log_{2k} m + 1/(2k)$ passes while a k-way merge utilizing only $2k$ tapes will make $\log_k m + 1$ passes. Table 8.13 compares the number of passes being made in the two methods for some values of k.

```
k   2k-way              k-way

1   3/2 log₂ m + 1/2        --

2   7/8 log₂ m + 1/6    log₂ m + 1

3   1.124 log₃ m + 1/6  log₃ m + 1

4   1.25 log₄ m + 1/8   log₄ m + 1
```

Table 8.13 Number of passes using a 2k-way merge versus a k-way merge on 2k + 1 tapes

As is evident from the table, for $k > 2$ a k-way merge using only $2k$ tapes is better than a $2k$-way merge using $2k + 1$ tapes.

Algorithm $M3$ performs a k-way merge sort using $2k$ tapes.

```
procedure M3

//Sort a file of records from a given input tape using a k-way

merge on 2k tapes T₁, ...,T₂ₖ//
```

1   *Create runs from the input file distributing them evenly over tapes*

T_1, ...,T_k

2   *rewind T_1, ...,T_k; rewind the input tape; replace the input tape by*

tape T_{2_k}; i ☐ *0*

3   **while** *total number of runs on T_{ik+1}, ...,T_{ik+k} > 1* **do**

4   *j* ☐ *1 - i*

5   *perform a k-way merge from T_{ik+1}, ...,T_{ik+_k} evenly distributing*

output runs onto T_{jk+1}, ...,T_{jk+k}.

6   *rewind T_1, ...,T_{2_k}*

7   *i* ☐ *j* //switch input an output tapes//

8   **end**

//sorted file is on T_{ik+1}//

9   **end** *M3*

Analysis of M3

To simplify the analysis assume that *m* is a power of *k*. In addition to the initial run creation pass, the algorithm makes $\log_k m$ merge passes. Let t_{rew} be the time to rewind the entire input file. The time for line 2 is t_{rew} and if *m* is a power of *k* then the rewind of line 6 takes t_{rew}/k for each but the last iteration of the **while** loop. The last rewind takes time t_{rew}. The total rewind time is therefore bounded by $(2 + (\log_k m - 1)/k)t_{rew}$. Some of this may be overlapped using the strategy described in the analysis of algorithm *M2* (exercise 4).

One should note that *M*1, *M*2 and *M*3 use the buffering algorithm of section 8.2.2 during the *k*-way merge. This

merge itself, for large k, would be carried out using a selection tree as discussed earlier. In both cases the proper choice of buffer lengths and the merge order (restricted by the number of tape drives available) would result in an almost complete overlap of internal processing time with input/output time. At the end of each level of merge, processing will have to wait until the tapes rewind. Once again, this wait can be minimized if the run distribution strategy of exercise 4 is used.

8.3.2 Polyphase Merge

Balanced k-way merging is characterized by a balanced or even distribution of the runs to be merged onto k input tapes. One consequence of this is that $2k$ tapes are needed to avoid wasteful passes over the data during which runs are just redistributed onto tapes in preparation for the next merge pass. It is possible to avoid altogether these wasteful redistribution passes, when using fewer than $2k$ tapes and a k-way merge, by distributing the "right number" of runs onto different tapes. We shall see one way to do this for a k-way merge utilizing $k + 1$ tapes. To begin, let us see how $m = 21$ runs may be merged using a 2-way merge with 3 tapes $T1$, $T2$ and $T3$. Lengths of runs obtained during the merge will be represented in terms of the length of initial runs created by the internal sort. Thus, the internal sort creates 21 runs of length 1 (the length of initial runs is the unit of measure). The runs on a tape will be denoted by s^n where s is the run size and n the number of runs of this size. If there are six runs of length 2 on a tape we shall denote this by 2^6. The sort is carried out in seven phases. In the first phase the input file is sorted to obtain 21 runs. Thirteen of these are written onto $T1$ and eight onto $T2$. In phase 2, 8 runs from $T2$ are merged with 8 runs from $T1$ to get 8 runs of size 2 onto $T3$. In the next phase the 5 runs of length 1 from $T1$ are merged with 5 runs of length 2 from $T3$ to obtain 5 runs of length 3 on $T2$. Table 8.14 shows the 7 phases involved in the sort.

Phase	T1	T2	T3	Fraction of Total Records Read	
1	1^{13}	1^8	--	1	initial distribution
2	1^5	--	2^8	16/21	merge to T3
3	--	3^5	2^3	15/21	merge to T2
4	5^3	3^2	--	15/21	merge to T1
5	5^1	--	8^2	16/21	merge to T3
6	--	13^1	8^1	13/21	merge to T2
7	21^1	--	--	1	merge to T1

Table 8.14 Polyphase Merge on 3 Tapes

Counting the number of passes made during each phase we get $1 + 16/21 + 15/21 + 15/21 + 16/21 + 13/21 + 1 = 5$-

4/7 as the total number of passes needed to merge 21 runs. If algorithm $M2$ of section 8.3.1 had been used with $k = 2$, then $3/2 \lceil \log_2 21 \rceil + 1/2 = 8$ passes would have been made. Algorithm $M3$ using 4 tapes would have made $\lceil \log_2 21 \rceil = 5$ passes. What makes this process work? Examination of Table 8.14 shows that the trick is to distribute the runs initially in such a way that in all phases but the last only 1 tape becomes empty. In this case we can proceed to the next phase without redistribution of runs as we have 2 non-empty input tapes and one empty tape for output. To determine what the correct initial distribution is we work backwards from the last phase. Assume there are n phases. Then in phase n we wish to have exactly one run on $T1$ and no runs on $T2$ and $T3$. Hence, the sort will be complete at phase n. In order to achieve this, this run must be obtained by merging a run from $T2$ with a run from $T3$, and these must be the only runs on $T2$ and $T3$. Thus, in phase $n - 1$ we should have one run on each of $T2$ and $T3$. The run on $T2$ is obtained by merging a run from $T1$ with one from $T3$. Hence, in phase $n - 2$ we should have one run on $T1$ and 2 on $T3$.

Table 8.15 shows the distribution of runs needed in each phase so that merging can be carried out without any redistribution passes. Thus, if we had 987 runs, then a distribution of 377 runs onto $T1$ and 610 onto $T3$ at phase 1 would result in a 15 phase merge. At the end of the fifteenth phase the sorted file would be on $T1$. No redistribution passes would have been made. The number of runs needed for a n phase merge is readily seen to be $F_n + F_{n-1}$ where F_i is the i-th Fibonacci number (recall that $F_7 = 13$ and $F_6 = 8$ and that $F_{15} = 610$ and $F_{14} = 377$). For this reason this method of distributing runs is also known as Fibonacci merge. It can be shown that for three tapes this distribution of runs and resultant merge pattern requires only $1.04 \log_2 m + 0.99$ passes over the data. This compares very favorably with the $\log_2 m$ passes needed by algorithm $M3$ on 4 tapes using a balanced 2-way merge. The method can clearly be generalized to k-way merging on $k + 1$ tapes using generalized Fibonacci numbers. Table 8.16 gives the run distribution for 3-way merging on four tapes. In this case it can be shown that the number of passes over the data is about $0.703 \log_2 m + 0.96$.

Phase	T1	T2	T3
n	1	0	0
n – 1	0	1	1
n – 2	1	0	2
n – 3	3	2	0
n – 4	0	5	3
n – 5	5	0	8
n – 6	13	8	0
n – 7	0	21	13
n – 8	21	0	34
n – 9	55	34	0

```
n - 10    0   89    55

n - 11   89    0   144

n - 12  233  144     0

n - 13    0  377   233

n - 14  377    0   610
```

Table 8.15 Run Distribution for 3-Tape Polyphase Merge

```
Phase       T1     T2     T3     T4

  n          1      0      0      0

n - 1        0      1      1      1

n - 2        1      0      2      2

n - 3        3      2      0      4

n - 4        7      6      4      0

n - 5        0     13     11      7

n - 6       13      0     24     20

n - 7       37     24      0     44

n - 8       81     68     44      0

n - 9        0    149    125     81

n - 10     149      0    274    230

n - 11     423    274      0    504

n - 12     927    778    504      0

n - 13       0   1705   1431    927

n - 14    1705      0   3136   2632
```

Table 8.16 Polyphase Merge Pattern for 3-Way 4-Tape Merging

Example 8.4: The initial internal sort of a file of records creates 57 runs. 4 tapes are available to carry out the merging of these runs. The table below shows the status of the tapes using 3-way polyphase merge.

Phase	T1	T2	T3	T4	Fraction of Total Records Read	
1	1^{13}	--	1^{24}	1^{20}	1	initial distribution
2	--	3^{13}	1^{11}	1^{7}	39/57	merge onto T2
3	5^{7}	3^{6}	1^{4}	--	35/57	merge onto T1
4	5^{3}	3^{2}	--	9^{4}	36/57	merge onto T4
5	5^{1}	--	17^{2}	9^{2}	34/57	merge onto T3
6	--	31^{1}	17^{1}	9^{1}	31/57	merge onto T2
7	57^{1}	--	--	--	1	merge onto T1

The total number of passes over the data is $1 + 39/57 + 35/57 + 36/57 + 34/57 + 31/57 + 1 = 5\text{-}4/57$ compared to $[\log_2 57] = 6$ passes for 2-way balanced merging on 4 tapes.

Remarks on Polyphase Merging

Our discussion of polyphase merging has ignored altogether the rewind time involved. Before one can begin the merge for the next phase it is necessary to rewind the output tape. During this rewind the computer is essentially idle. It is possible to modify the polyphase merge scheme discussed here so that essentially all rewind time is overlapped with internal processing and read/write on other tapes. This modification requires at least 5 tapes and can be found in Knuth Vol. 3. Further, polyphase merging requires that the initial number of runs be a perfect Fibonacci number (or generalized Fibonacci number). In case this is not so, one can substitute dummy runs to obtain the required number of runs.

Several other ingenious merge schemes have been devised for tape sorts. Knuth, Vol. 3, contains an exhaustive study of these schemes.

8.3.3 Sorting with Fewer Than 3 Tapes

Both the balanced merge scheme of section 8.3.1 and the polyphase scheme of section 8.3.2 required at least 3 tapes

to carry out the sort. These methods are readily seen to require only $O(n \log n)$ time where n is the number of records. We state without proof the following results for sorting on 1 and 2 tapes.

Theorem 8.2: Any one tape algorithm that sorts n records must take time $\geq O(n^2)$.

Theorem 8.3: n records can be sorted on 2 tapes in $O(n \log n)$ time if it is possible to perform an inplace rewrite of a record without destroying adjacent records. I.e. if record R_2 in the sequence $R_1 R_2 R_3$ can be rewritten by an equal size recod R'_2 to obtain $R_1 R'_2 R_3$.

REFERENCES

See the readings for chapter 7.

The algorithm for theorem 8.3 may be found in:

"A linear time two tape merge" by R. Floyd and A. Smith, *Information Processing Letters,* vol. 2, no. 5, December 1973, pp. 123-126.

EXERCISES

1. Write an algorithm to construct a tree of losers for records ☐ with key values ☐. Let the tree nodes be ☐ with ☐ a pointer to the loser of a tournament and T_0 a pointer to the overall winner. Show that this construction can be carried out in time $O(k)$.

2. Write an algorithm, using a tree of losers, to carry out a k-way merge of k runs ☐. Show that if there are n records in the k runs together, then the computing time is $O(n \log_2 k)$.

3. a) n records are to be sorted on a computer with a memory capacity of S records ($S << n$). Assume that the entire S record capacity may be used for input/output buffers. The input is on disk and consists of m runs. Assume that each time a disk access in made the seek time $is \ t_s$ and the latency time is t_l. The transmission time is t_t per record transmitted. What is the total input time for phase II of external sorting if a k way merge is used with internal memory partitioned into I/O buffers so as to permit overlap of input, output and CPU processings as in algorithm BUFFERING?

b) Let the CPU time needed to merge all the runs together be t_{CPU} (we may assume it is independent of k and hence constant). Let $t_s = 80 \ ms, \ t_l = 20 \ ms, \ n = 200,000, \ m = 64, \ t_t = 10^{-3}$ sec/record, $S = 2000$. Obtain a rough plot of the total input time, t_{input}, versus k. Will there always be a value of k for which t_{CPU} ☐ t_{input}?

4. a) Modify algorithm $M3$ using the run distribution strategy described in the analysis of algorithm $M2$.

b) et t_{rw} be the time to read/write a block and t_{rew} the time to rewind over one block length. If the initial run creation

pass generates *m* runs for *m* a power of *k,* what is the time for a *k*-way merge using your algorithm. Compare this with the corresponding time for algorithm *M2.*

5. Obtain a table corresponding to Table 8.16 for the case of a 5-way polyphase merge on 6 tapes. Use this to obtain the correct initial distribution for 497 runs so that the sorted file will be on *T*1. How many passes are made over the data in achieving the sort? How many passes would have been made by a 5-way balanced merge sort on 6 tapes (algorithm *M2*)? How many passes would have been made by a 3-way balanced merge sort on 6 tapes (algorithm *M3*)?

6. In this exercise we shall investigate another run distribution strategy for sorting on tapes. Let us look at a 3-way merge on 4 tapes of 157 runs. These runs are initially distributed according to: 70 runs on *T*1, 56 on *T*2 and 31 on *T*3. The merging takes place according to the table below.

i.e., each phase consists of a 3-way merge followed by a two-way merge and in each phase almost all the initial runs are processed.

a) Show that the total number of passes made to sort 157 runs is 6-62/157.

b) Using an initial distribution from Table 8.16 show that Fibonacci merge on 4 tapes makes 6-59/193 passes when the number of runs is 193.

The distribution required for the process discussed above corresponds to the cascade numbers which are obtained as below for a *k*-way merge: Each phase (except the last) in a *k*-way cascade merge consists of a *k*-way merge followed by a *k* - 1 way merge followed by a *k* - 2 way merge . . . a 2-way merge. The last phase consists of only a *k*-way merge. The table below gives the cascade numbers corresponding to a *k*-way merge. Each row represents a starting configuration for a merge phase. If at the start of a phase we have the distribution $n_1, n_2, ..., n_k$ where $n_i > n_{i+1}$, $1 \leq i < k$, then at the start of the previous phase we need runs on the *k* input tapes respectively.

Initial Distribution for a k-way Cascade Merge

It can be shown that for a 4-way merge Cascade merge results in more passes over the data than a 4-way Fibonacci merge.

7. a) Generate the first 10 rows of the initial distribution table for a 5-way Cascade merge using 6 tapes (see exercise 6).

b) serve that 671 runs corresponds to a perfect 5-way Cascade distribution. How many passes are made in sorting the data using a 5-way Cascade merge?

c) How many passes are made by a 5-way Fibonacci merge starting with 497 runs and the distribution 120, 116,

108, 92, 61? ☐

The number of passes is almost the same even though Cascade merge started with 35% more runs! In general, for $\geq$ 6 tapes Cascade merge makes fewer passes over the data than does Fibonacci merge.

8. List the runs output by algorithm RUNS using the following input file and $k = 4$.

100,50, 18,60,2,70,30, 16,20, 19,99,55,20

9. For the example of Table 8.14 compute the total rewind time. Compare this with the rewind time needed by algorthm *M2*.

Go to Chapter 9   Back to Table of Contents

◀   ■   ▶

CHAPTER 9: SYMBOL TABLES

The notion of a symbol table arises frequently in computer science. When building loaders, assemblers, compilers, or any keyword driven translator a symbol table is used. In these contexts a symbol table is a *set of name-value pairs*. Associated with each name in the table is an attribute, a collection of attributes, or some directions about what further processing is needed. The operations that one generally wants to perform on symbol tables are (i) ask if a particular name is already present, (ii) retrieve the attributes of that name, (iii) insert a new name and its value, and (iv) delete a name and its value. In some applications, one wishes to perform only the first three of these operations. However, in its fullest generality we wish to study the representation of a structure which allows these four operations to be performed efficiently.

Is a symbol table necessarily a *set* of names or can the same name appear more than once? If we are translating a language with block structure (such as ALGOL or *PL/I*) then the variable *X* may be declared at several levels. Each new declaration of *.X* implies that a new variable (with the same name) must be used. Thus we may also need a mechanism to determine which of several occurrences of the same name has been the most recently introduced.

These considerations lead us to a specificaton of the structure *symbol table*. A set of axioms are given on the next page.

Imagine the following set of declarations for a language with block structure:

begin

integer *i,j;*

.

.

.

begin

real *x,i;*

.

.

The representation of the symbol table for these declarations would look like *S* =

INSERT(INSERT(INSERT(INSERT(CREATE,*i*,**integer**),

j,**integer**),*x*,**real**),*i*,**real**)

Notice the identifer *i* which is declared twice with different attributes. Now suppose we apply FIND (*S*, *i*). By the axioms EQUAL(*i*,*i*) is tested and has the value **true**. Thus the value **real** is returned as a result. If the function DELETE(*S*,*i*) is applied then the result is the symbol table

INSERT(INSERT(INSERT(CREATE,*i*,**integer**),*j*,**integer**),*x*,**real**)

If we wanted to change the specification of DELETE so that it removes all occurrences of *i* then one axiom needs to be redefined, namely

DELETE(INSERT(S,a,r),b) :: =

if *EQUAL(a,b)* **then** *DELETE(S,b)*

else *INSERT(DELETE(S,b),a,r)*

structure *SYMBOL__TABLE*

declare *CREATE( )* $\rightarrow$ *symtb*

INSERT(symtb,name, value) $\rightarrow$ *symtb*

DELETE(symtb,name) $\rightarrow$ *symtb*

FlND(symtb,name) $\rightarrow$ *value*

HAS(symtb,name) $\rightarrow$ *Boolean*

ISMTST(symtb) $\rightarrow$ *Boolean;*

for all *S* $\in$ *symtb, a,b* $\in$ *name, r* $\in$ *value* **let**

ISMTST(CREATE) :: = **true**

```
ISMTST(INSERT(S,a,r)) :: = false

HAS(CREATE,a) :: = false

HAS(INSERT(S,a,r),b) :: =

if EQUAL(a,b) then true else HAS(S,b)

DELETE(CREATE,a) :: = CREATE

DELETE(INSERT(S,a,r),b) :: =

if EQUAL(a,b) then S

else INSERT(DELETE(S,b),a,r)

FIND(CREATE,a) :: = error

FIND(INSERT(S,a,r),b) :: = if EQUAL(a,b) then r

else FIND(S,b)

end

end SYMBOL__TABLE
```

The remaining sections of this chapter are organized around different ways to implement symbol tables. Different strategies are preferable given different assumptions. The first case considered is where the identifiers are known in advance and no deletions or insertions are allowed. Symbol tables with this property are called *static*. One solution is to sort the names and store them sequentially. Using either the binary search or Fibonacci search method of section 7.1 allows us to find any name in $O(\log_2 n)$ operations. If each name is to be searched for with equal probability then this solution, using an essentially balanced binary search tree, is optimal. When different identifiers are searched for with differing probabilities and these probabilities are known in advance this solution may not be the best. An elegant solution to this problem is given in section 9.1.

In section 9.2 we consider the use of binary trees when the identifiers are not known in advance. Sequential allocation is no longer desirable because of the need to insert new elements. The solution which is examined is AVL or height balanced trees. Finally in section 9.3 we examine the most practical of the techniques for dynamic search and insertion, hashing.

9.1 STATIC TREE TABLES

Definition: A *binary search tree T* is a binary tree; either it is empty or each node in the tree contains an identifier and:

(i) all identifiers in the left subtree of *T* are less (numerically or alphabetically) than the identifier in the root node *T*;

(ii) all identifiers in the right subtree of *T* are greater than the identifier in the root node *T*;

(iii) the left and right subtrees of *T* are also binary search trees.

For a given set of identifiers several binary search trees are possible. Figure 9.1 shows two possible binary search trees for a subset of the reserved words of SPARKS.

To determine whether an identifier *X* is present in a binary search tree, *X* is compared with the root. If *X* is less than the identifier in the root, then the search continues in the left subtree; if *X* equals the identifier in the root, the search terminates successfully; otherwise the search continues in the right subtree. This is formalized in algorithm SEARCH.

☐

Figure 9.1 Two possible binary search trees

```
procedure SEARCH(T,X,i)

//search the binary search tree T for X. Each node has fields

LCHILD, IDENT, RCHILD. Return i = 0 if X is not in T.

Otherwise, return i such that IDENT(i) = X.//

1    i ☐ T

2    while i ≠ 0 do

3       case

4          :X < IDENT(i): i ☐ LCHILD(i)  //search left subtree//
```

```
5        :X = IDENT(i): return

6        :X > IDENT(i): i ☐ RCHILD(i)  //search right subtree//

7    end

8   end

9 end SEARCH
```

In our study of binary search in chapter 7, we saw that every sorted file corresponded to a binary search tree. For instance, a binary search on the file (**do, if, stop**) corresponds to using algorithm SEARCH on the binary search tree of figure 9.2. While this tree is a full binary tree, it need not be optimal over all binary search trees for this file when the identifiers are searched for with different probabilities. In order to determine an optimal binary search tree for a given static file, we must first decide on a cost measure for search trees. In searching for an identifier at level k using algorithm SEARCH, k iterations of the **while** loop of lines 2-8 are made. Since this loop determines the cost of the search, it is reasonable to use the level number of a node as its cost.

☐

Figure 9.2 Binary search tree corresponding to a binary search on the file (do,if,stop).

Consider the two search trees of figure 9.1 as possibly representing the symbol table of the SPARKS compiler. As names are encountered a match is searched for in the tree. The second binary tree requires at most three comparisons to decide whether there is a match. The first binary tree may require four comparisons, since any identifier which alphabetically comes after **if** but precedes **repeat** will test four nodes. Thus, as far as worst case search time is concerned, this makes the second binary tree more desirable than the first. To search for an identifier in the first tree takes one comparison for the **if**, two for each of **for** and **while**, three for **repeat** and four for **loop**. Assuming each is searched for with equal probability, the average number of comparisons for a successful search is 2.4. For the second binary search tree this amount is 2.2. Thus, the second tree has a better average behavior, too.

In evaluating binary search trees, it is useful to add a special "square" node at every place there is a null link. Doing this to the trees of figure 9.1 yields the trees of figure 9.3. Remember that every binary tree with n nodes has $n + 1$ null links and hence it will have $n + 1$ square nodes. We shall call these nodes *external* nodes because they are not part of the original tree. The remaining nodes will be called *internal* nodes. Each time a binary search tree is examined for an identifier which is not in the tree, the search terminates at an external node. Since all such searches represent unsuccessful searches, external nodes will also be referred to as *failure* nodes. A binary tree with external nodes added is an *extended binary tree*. Figure 9.3 shows the extended binary trees corresponding to the search trees of figure 9.1. We define the *external path length* of a binary tree to be the sum over all external nodes of the lengths of the

paths from the root to those nodes. Analogously, the *internal path length* is defined to be the sum over all internal nodes of the lengths of the paths from the root to those nodes. For the tree of figure 9.3(a) we obtain its internal path length, *I*, to be:

$$I = 0 + 1 + 1 + 2 + 3 = 7$$

Its external path length, *E*, is:

$$E = 2 + 2 + 4 + 4 + 3 + 2 = 17.$$

☐

Figure 9.3 Extended binary trees corresponding to search trees of figure 9.1.

Exercise 1 shows that the internal and external path lengths of a binary tree with *n* internal nodes are related by the formula $E = I + 2n$. Hence, binary trees with the maximum *E* also have maximum *I*. Over all binary trees with *n* internal nodes what are the maximum and minimum possible values for *I*? The worst case, clearly, is when the tree is skewed (i.e., when the tree has a depth of *n*). In this case,

☐

To obtain trees with minimal *I*, we must have as many internal nodes as close to the root as possible. We can have at most 2 nodes at distance 1, 4 at distance 2, and in general the smallest value for *I* is

$$0 + 2 \boxed{} 1 + 4 \boxed{} 2 + 8 \boxed{} 3 + ...$$

This can be more compactly written as

☐

One tree with minimal internal path length is the complete binary tree defined in section 5.2.

Before attempting to use these ideas of internal and external path length is to obtain optimal binary search trees, let us look at a related but simpler problem. We are given a set of $n + 1$ positive weights $q_1, ..., q_{n + 1}$. Exactly one of these weights is to be associated with each of the $n + 1$ external nodes in a binary tree with *n* internal nodes. The *weighted external path length* of such a binary tree is defined to be $\sum_{1 \le i \le n + 1} q_i k_i$ where k_i is the distance from the root node to the external node with weight q_i. The problem is to determine a binary tree with minimal weighted external path length. Note that here no information is contained within internal nodes.

For example, suppose $n = 3$ and we are given the four weights: $q_1 = 15$, $q_2 = 2$, $q_3 = 4$ and $q_4 = 5$. Two possible trees would be:

Their respective weighted external path lengths are:

$$2 \square 3 + 4 \square 3 + 5 \square 2 + 15 \square 1 = 43$$

and

$$2 \square 2 + 4 \square 2 + 5 \square 2 + 15 \square 2 = 52.$$

Binary trees with minimal weighted external path length find application in several areas. One application is to determine an optimal merge pattern for $n + 1$ runs using a 2-way merge. If we have four runs R_1-R_4 with q_i being the number of records in run R_i, $1 \le i \le 4$, then the skewed binary tree above defines the following merge pattern: merge R_2 and R_3; merge the result of this with R_4 and finally merge with R_1. Since two runs with n and m records each can be merged in time $O(n + m)$ (cf., section 7.5), the merge time following the pattern of the above skewed tree is proportional to $(q_2 + q_3) + \{(q_2 + q_3) + q_4\} + \{q_1 + q_2 + q_3 + q_4\}$. This is just the weighted external path length of the tree. In general, if the external node for run R_i is at a distance k_i from the root of the merge tree, then the cost of the merge will be proportional to $\Sigma q_i k_i$ which is the weighted external path length.

Another application of binary trees with minimal external path length is to obtain an optimal set of codes for messages M_1, ...,M_{n+1}. Each code is a binary string which will be used for transmission of the corresponding message. At the receiving end the code will be decoded using a decode tree. A decode tree is a binary tree in which external nodes represent messages. The binary bits in the code word for a message determine the branching needed at each level of the decode tree to reach the correct external node. For example, if we interpret a zero as a left branch and a one as a right branch, then the decode tree corresponds to codes 000, 001, 01 and 1 for messages M_1, M_2, M_3 and M_4 respectively. These codes are called Huffman codes. The cost of decoding a code word is proportional to the number of bits in the code. This number is equal to the distance of the corresponding external node from the root node. If q_i is the relative frequency with which message M_i will be transmitted, then the expected decode time is $\Sigma_{1 \le i \le n+1} q_i d_i$ where d_i is the distance of the external node for message M_i from the root node. The expected decode time is minimized by choosing code words resulting in a decode tree with minimal weighted external path length!

A very nice solution to the problem of finding a binary tree with minimum weighted external path length has been given by D. Huffman. We simply state his algorithm and leave the correctness proof as an exercise. The algorithm HUFFMAN makes use of a list *L* of extended binary trees. Each node in a tree has three fields: WEIGHT, LCHILD and RCHILD. Initially, all trees in *L* have only one node. For each tree this node is an external node, and its weight is one of the provided q_i's. During the course of the algorithm, for any tree in *L* with root node *T* and depth greater than 1, WEIGHT(*T*) is the sum of weights of all external nodes in *T*. Algorithm HUFFMAN uses the subalgorithms LEAST and INSERT. LEAST determines a tree in *L* with minimum WEIGHT and removes it from *L*. INSERT adds a new tree to the list *L*.

```
procedure HUFFMAN(L,n)

//L is a list of n single node binary trees as described above//

for i □ 1 to n - 1 do              //loop n - 1 times//

call GETNODE(T)                    //create a new binary tree//

LCHILD(T) □ LEAST(L)               //by combining the trees//

RCH1LD(T) □ LEAST(L)               //with the two smallest weights//

WEIGHT(T) □ WEIGHT(LCHILD(T))

+ WEIGHT(RCHILD(T))

call INSERT(L,T)

end

end HUFFMAN
```

We illustrate the way this algorithm works by an example. Suppose we have the weights $q_1 = 2$, $q_2 = 3$, $q_3 = 5$, $q_4 = 7$, $q_5 = 9$ and $q_6 = 13$. Then the sequence of trees we would get is: (the number in a circular node represents the sum of the weights of external nodes in that subtree).

The weighted external path length of this tree is

$2 \boxed{} 4 + 3 \boxed{} 4 + 5 \boxed{} 3 + 13 \boxed{} 2 + 7 \boxed{} 2 + 9 \boxed{} 2 = 93$

In comparison, the best complete binary tree has weighted path length 95.

Analysis of Algorithm HUFFMAN

The main loop is executed n - 1 times. If L is maintained as a heap (section 7.6) then each call to LEAST and INSERT requires only $O(\log n)$ time. Hence, the asymptotic computing time for the algorithm is $O(n \log n)$.

Let us now return to our original problem of representing a symbol table as a binary search tree. If the binary search tree contains the identifiers $a_1, a_2, ..., a_n$ with $a_1 < a_2 < ... < a_n$ and the probability of searching for each a_i is p_i, then the total cost of any binary search tree is $\sum_{1 \leq i \leq n} p_i$. level (a_i) when only successful searches are made. Since unsuccessful searches, i.e., searches for identifiers not in the table, will also be made, we should include the cost of these searches in our cost measure, too. Unsuccessful searches terminate with $i = 0$ in algorithm SEARCH. Every node with a null subtree defines a point at which such a termination can take place. Let us replace every null subtree by a failure node. The identifiers not in the binary search tree may be partitioned into $n + 1$ classes E_i, $0 \leq i \leq n$. E_o contains all identifiers X such that $X < a_1$. E_i contains all identifiers X such that $a_i < X < a_{i+1}$, $1 \leq i < n$ and E_n contains all identifiers X, $X > a_n$. It is easy to see that for all identifiers in a particular class E_i, the search terminates at the same failure node and it terminates at different failure nodes for identifiers in different classes. The failure nodes may be numbered 0 to n with i being the failure node for class E_i, $0 \leq i \leq n$. If q_i is the probability that the identifier being searched for is in E_i, then the cost of the failure nodes is $\sum_{0 \leq i \leq n} q_i$; (level (failure node i) - 1). The total cost of a binary search tree is therefore:

$\boxed{}$

(9.1)

An *optimal binary search tree* for the identifier set $a_1, \ldots, a_n$ is one which minimizes eq. (9.1) over all possible binary search trees for this identifier set. Note that since all searches must terminate either successfully or unsuccessfully we have $\sum_{1 \leq i \leq n} P_i + \sum_{0 \leq i \leq n} q_i = 1$.

Example 9.1: The possible binary search trees for the identifier set $(a_1, a_2, a_3) = ($**do**, **if**, **stop**$)$ are:

$\boxed{}$

With equal probabilities $p_i = q_j = 1/7$ for all i and j we have:

```
cost (tree a) = 15/7; cost (tree b) = 13/7
```

```
cost (tree c) = 15/7; cost (tree d) = 15/7
```

```
cost (tree e) = 15/7.
```

As expected, tree *b* is optimal. With $p_1 = .5$, $p_2 = .1$, $p_3 = .05$, $q_0 = .15$, $q_1 = .1$, $q_2 = .05$ and $q_3 = .05$ we have

```
cost (tree a) = 2.65; cost (tree b) = 1.9
```

```
cost (tree c) = 1.5;  cost (tree d) = 2.05
```

```
cost (tree e) = 1.6
```

Tree *c* is optimal with this assignment of *p's* and *q's*.

From among all the possible binary search trees how does one determine the optimal tree? One possibility would be to proceed as in example 9.1 and explicitly generate all possible binary search trees. The cost of each such tree could be computed and an optimal tree determined. The cost of each of the binary search trees can be determined in time $O(n)$ for an *n* node tree. If $N(n)$ is the total number of distinct binary search trees with *n* identifiers, then the complexity of the algorithm becomes $O(n\ N(n))$. From section 5.9 we know that $N(n)$ grows too rapidly with increasing *n* to make this brute force algorithm of any practical significance. Instead we can find a fairly efficient algorithm by making some observations regarding the properties of optimal binary search trees.

Let $a_1 < a_2 < ... < a_n$ be the *n* identifiers to be represented in a binary search tree. Let us denote by T_{ij} an optimal binary search tree for $a_{i+1}, ...,a_j$, $i < j$. We shall adopt the convention that T_{ii} is an empty tree for $0 \le i \le n$ and that T_{ij} is not defined for $i > j$. We shall denote by c_{ij} the cost of the search tree T_{ij}. By definition c_{ii} will be 0. r_{ij} will denote the root of T_{ij} and $w_{ij} = q_i + \Sigma^j_{k=i+1} (q_k + p_k)$ will denote the weight of T_{ij}. By definition we will have $r_{ii} = 0$ and $w_{ii} = q_i$, $0 \le i \le n$. An optimal binary search tree for $a_1, ...,a_n$ is therefore T_{on}, its cost is c_{on}, its weight w_{on} and its root r_{on}.

If T_{ij} is an optimal binary search tree for $a_{i+1}, ...,a_j$ and $r_{ij} = k$, $i < k \le j$, then T_{ij} has two subtrees *L* and *R*. *L* is the left subtree and contains the identifiers $a_{i+1}, ...,a_{k-1}$ and *R* is the right subtree and contains the identifiers $a_{k+1}, ...,a_j$ (figure 9.4). The cost c_{ij} of T_{ij} is

```
c_{i,j} = p_k + cost (L) + cost (R) + w_{i,k-1}+ w_{k,j}
```

```
weight (L) = weight (T_{i,k-1}) = w_{i,k-1}
```

```
weight (R) = weight (T_{k,j}) = w_{k,j}
```

(9.2)

Figure 9.4 An optimal binary search tree T_{ij}

From eq. (9.2) it is clear that if c_{ij} is to be minimal, then cost $(L) = c_{i,k-1}$ and cost $(R) = c_{k,j}$ as otherwise we could replace either L or R by a subtree of lower cost thus getting a binary search tree for $a_{i+1}, ..., a_j$ with lower cost than c_{ij}. This would violate the assumption that T_{ij} was optimal. Hence, eq. (9.2) becomes:

```
c_{i,j} = p_k + w_{i,k-1} + w_{k,j} + c_{i,k-1} + c_{k,j}
```

```
= wi,j + c_{i,k-1} + c_{k,j}
```

(9.3)

Since T_{ij} is optimal it follows from eq. (9.3) that $r_{ij} = k$ is such that

```
w_{ij} + c_{i,k-1} + c_{k,j} = min {w_{ij} + c_{i,l-1} + c_{l,j}}
```

```
i<l≤j
```

or

```
c_{i,k-1} + c_{k,j} = min {c_{i,l-1} + c_{l,j}}
```

```
i<l≤j
```

(9.4)

Equation (9.4) gives us a means of obtaining T_{on} and c_{on} starting from the knowledge that $T_{ii} = \Phi$ and $c_{ii} = 0$.

Example 9.2: Let $n = 4$ and $(a_1, a_2, a_3, a_4) = ($**do, if, read, while**$)$. Let $(p_1, p_2, p_3, p_4) = (3,3,1,1)$ and

$(q_0, q_1, q_2, q_3, q_4) = (2,3,1,1,1)$. The p's and q's have been multiplied by 16 for convenience. Initially, we have $w_{i,i} = q_i$, $c_{ii} = 0$ and $r_{ii} = 0$, $0 \leq i \leq 4$. Using eqs. (9.3) and (9.4) we get:

```
w01 = p1 + w00 + w11 = p1 + q1 + w00 = 8

c01 = w01 + min {c00 + c11} = 8

r01 = 1

w12 = p2 + w11 + w22 = p2 + q2 + w11 = 7

c12 = w12 + min {c11 + c22} = 7

r12 = 2

w23 = p3 + w22 + w33 = p3 + q3 + w22 = 3

c23 = w23 + min {c22 + c33} = 3

r23 = 3

w34 = p4 + w33 + w44 = p4 + q4 + w33 = 3

c34 = w34 + min {c33 + c44} = 3

r34= 4
```

Knowing $w_{i,i+1}$ and $c_{i,i+1}$, $0 \leq i \leq 4$ we can again use equations (9.3) and (9.4) to compute $w_{i,i+2}$, $c_{i,i+2}$, $r_{i,i+2}$, $0 \leq i \leq 3$. This process may be repeated until w_{04}, c_{04} and r_{04} are obtained. The table of figure 9.5 shows the results of this computation. From the table we see that $c_{04} = 32$ is the minimal cost of a binary search tree for a_1 to a_4. The root of tree T_{04} is a_2. Hence, the left subtree is T_{01} and the right subtree T_{24}. T_{01} has root a_1 and subtrees T_{00} and T_{11}. T_{24} has root a_3; its left subtree is therefore T_{22} and right subtree T_{34}. Thus, with the data in the table it is possible to reconstruct T_{04}. Figure 9.6 shows T_{04}.

☐

Figure 9.5 Computation of c_{04}, w_{04} and r_{04}. The computation is carried out row wise from row 0 to row 4.

☐

Figure 9.6 Optimal search tree for example 9.2

The above example illustrates how equation (9.4) may be used to determine the c's and r's and also how to reconstruct T_{on} knowing the r's. Let us examine the complexity of this procedure to evaluate the c's and r's. The evaluation procedure described in the above example requires us to compute c_{ij} for $(j - i) = 1,2, ...,n$ in that order. When $j - i = m$ there are $n - m + 1$ c_{ij}'s to compute. The computation of each of these c_{ij}'s requires us to find the minimum of m quantities (see equation (9.4)). Hence, each such c_{ij} can be computed in time $O(m)$. The total time for all c_{ij}'s with $j - i = m$ is therefore $O(nm - m^2)$. The total time to evaluate all the c_{ij}'s and r_{ij}'s is therefore

$$\Sigma 1 \leq m \leq n \ (nm - m^2) = O(n^3).$$

Actually we can do better than this using a result due to D. E. Knuth which states that the optimal l in equation (9.4) may be found by limiting the search to the range $r_{i,j}\text{-}1 \leq l \leq r_{i+1,j}$. In this case the computing time becomes $O(n^2)$ (exercise 4). Algorithm OBST uses this result to obtain in $O(n^2)$ time the values of w_{ij}, r_{ij} and c_{ij}, $0 \leq i \leq j \leq n$. The actual tree T_{on} may be constructed from the values of r_{ij} in $O(n)$ time. The algorithm for this is left as an exercise.

```
procedure OBST(p_i,q_i,n)

//Given n distinct identifiers a_1 < a_2 < ... < a_n and probabilities p_i,

1 ≤ i ≤ n and q_i, 0 ≤ i ≤ n this algorithm computes the cost c_ij

of optimal binary search trees T_ij for identifiers a_{i+1}, ...,a_j.

It also computes r_ij, the root of T_ij. w_ij is the weight of T_ij//

for i ☐ 0 to n - 1 do

(w_ii,r_ii,c_ii) ☐ (q_i,0,0)                //initialize//

(w_{i,i+1},r_{i,i+1},c_{i,i+1}) ☐ (q_i + q_{i+1} + p_{i+1}, i+1, q_i+ q_{i+1} + p_{i+1})

//optimal trees with one node//
```

end

(w_{nn}, r_{nn}, c_{nn}) ☐ $(q_n, 0, 0)$

for m ☐ 2 to n do   //find optimal trees with m nodes//

for i ☐ 0 **to** $n - m$ **do**

j ☐ $i + m$

w_{ij} ☐ $w_{i,j-1} + p_j + q_j$

k ☐ a value of l in the range $r_{i,j-1} \leq l \leq r_{i+1,j}$

that minimizes $\{c_{i,l-1} + c_{l,j}\}$ //solve (9.4) using

Knuth's result//

c_{ij} ☐ $w_{ij} + c_{i,k-1} + c_{k,j}$ //eq. (9.3)//

r_{ij} ☐ k

end

end

end *OBST*

9.2 DYNAMIC TREE TABLES

Dynamic tables may also be maintained as binary search trees. An identifier *X* may be inserted into a binary search tree *T* by using the search algorithm of the previous section to determine the failure node corresponding to *X*. This gives the position in *T* where the insertion is to be made. Figure 9.7 shows the binary search tree obtained by entering the months JANUARY to DECEMBER in that order into an initially empty binary search tree. Algorithm BST is a more formal rendition of the insertion process just described.

The maximum number of comparisons needed to search for any identifier in the tree of figure 9.7 is six for NOVEMBER. The average number of comparisons is (1 for JANUARY + 2 each for FEBRUARY and MARCH + 3 each for APRIL, JUNE and MAY + ... + 6 for NOVEMBER) / 12 = 42/ 12 = 3.5. If the months are entered in the order JULY, FEBRUARY, MAY, AUGUST, DECEMBER, MARCH, OCTOBER, APRIL, JANUARY, JUNE, SEPTEMBER and NOVEMBER then the tree of figure 9.8 is obtained. This tree is well balanced and does not have any paths to a node with a null link that are much longer than others. This is not true of the tree of figure 9.7 which has six nodes on the path from the root to NOVEMBER and only two nodes, JANUARY and FEBRUARY, on another path to a null link. Moreover, during the construction of the tree of figure 9.8 all intermediate trees obtained are also well balanced. The maximum number of identifier comparisons needed to find any identifier is now 4 and the average is 37/12 $\square$ 3.1. If instead the months are entered in lexicographic order, the tree degenerates to a chain as in figure 9.9. The maximum search time is now 12 identifier comparisons and the average is 6.5. Thus, in the worst case, binary search trees correspond to sequential searching in an ordered file. When the identifiers are entered in a random order, the tree tends to be balanced as in figure 9.8. If all permutations are equiprobable then it can be shown the average search and insertion time is $O(\log n)$ for an n node binary search tree.

```
procedure BST (X,T,j)

//search the binary search tree T for the node j such that IDENT(j)

= X. If X is not already in the table then it is entered at the

appropriate point. Each node has LCHILD, IDENT and RCHILD

fields//

p □ 0; j □ T          //p will trail j through the tree//

while j ≠ 0 do

case

:X < IDENT(j): p □ j; j □ LCHILD(j)      //search left

sub-tree//

:X = IDENT(j): return

:X > IDENT(j): p □ j; j □ RCHILD(j)      //search right
```

```
sub-tree//

end

end

//X is not in the tree and can be entered as a child of p//

call GETNODE(j); IDENT(j) ☐ X; LCHILD(j) ☐ 0;

RCHILD(j) ☐ 0

case

:T = 0: T ☐ j   //insert into empty tree//

:X < IDENT(p): LCHILD(p) ☐ j

:else: RCHILD(p) ☐ j

end

end BST
```


Figure 9.7 Binary search tree obtained for the months of the year.

Figure 9.8 A balanced tree for the months of the year.

From our earlier study of binary trees, we know that both the average and maximum search time will be minimized if the binary search tree is maintained as a complete binary tree at all times. However, since we are dealing with a dynamic situation, identifiers are being searched for while the table is being built and so it is difficult to achieve this ideal without making the time required to add new entries very high. This is so because in some cases it would be necessary to restructure the whole tree to accommodate the new entry and at the same time have a complete binary search tree. It is, however, possible to keep the trees balanced so as to ensure both an average and worst case retrieval time of $O(\log n)$ for a tree with n nodes. We shall study one method of growing balanced binary trees. These balanced trees will have

satisfactory search and insertion time properties.

Figure 9.9 Degenerate binary search tree

Height Balanced Binary Trees

Adelson-Velskii and Landis in 1962 introduced a binary tree structure that is balanced with respect to the heights of subtrees. As a result of the balanced nature of this type of tree, dynamic retrievals can be performed in $O(\log n)$ time if the tree has n nodes in it. At the same time a new identifier can be entered or deleted from such a tree in time $O(\log n)$. The resulting tree remains height balanced. The tree structure introduced by them is given the name AVL-tree. As with binary trees it is natural to define AVL trees recursively.

Definition: An empty tree is height balanced. If T is a nonempty binary tree with T_L and T_R as its left and right subtrees, then T is *height balanced* iff (i) T_L and T_R are height balanced and (ii) $|h_L - h_R| \leq 1$ where h_L and h_R are the heights of T_L and T_R respectively.

The definition of a height balanced binary tree requires that every subtree also be height balanced. The binary tree of figure 9.7 is not height balanced since the height of the left subtree of the tree with root 'APRIL' is 0 while that of the right subtree is 2. The tree of figure 9.8 is height balanced while that of figure 9.9 is not. To illustrate the processes involved in maintaining a height balanced binary search tree, let us try to construct such a tree for the months of the year. This time let us assume that the insertions are made in the order MARCH, MAY, NOVEMBER, AUGUST, APRIL, JANUARY, DECEMBER, JULY, FEBRUARY, JUNE, OCTOBER and SEPTEMBER. Figure 9.10 shows the tree as it grows and the restructuring involved in keeping the tree balanced. The numbers within each node represent the difference in heights between the left and right subtrees of that node. This number is referred to as the balance factor of the node.

Definition: The *balance factor*, $BF(T)$, of a node T in a binary tree is defined to be $h_L - h_R$ where h_L and h_R are the heights of the left and right subtrees of T. For any node T in an AVL tree $BF(T) = -1, 0$ or 1.

Figure 9.10 Balanced trees obtained for the months of the year.

Figure 9.10 (continued)

Figure 9.10 (continued)

Figure 9.10 (continued)

Figure 9.10 (continued)

Figure 9.10 (continued)

Inserting MARCH and MAY results in the binary search trees (i) and (ii) of figure 9.10. When NOVEMBER is inserted into the tree the height of the right subtree of MARCH becomes 2, while that of the left subtree is 0. The tree has become unbalanced. In order to rebalance the tree, a rotation is performed. MARCH is made the left child of MAY and MAY becomes the root. The introduction of AUGUST leaves the tree balanced. However, the next insertion, APRIL, causes the tree to become unbalanced again. To rebalance the tree, another rotation is performed. This time, it is a clockwise rotation. MARCH is made the right child of AUGUST and AUGUST becomes the root of the subtree (figure 9.10(v)). Note that both the previous rotationswere carried out with respect to the closest parent of the new node having a balance factor of ±2. The insertion of JANUARY results in an unbalanced tree. This time, however, the rotation involved is somewhat more complex than in the earlier situations. The common point, however, is that it is still carried out with respect to the nearest parent of JANUARY with balance factor ±2. MARCH becomes the new root. AUGUST together with its left subtree becomes the left subtree of MARCH. The left subtree of MARCH becomes the right subtree of AUGUST. MAY and its right subtree, which have identifiers greater than MARCH, become the right subtree of MARCH. If MARCH had a non-empty right subtree, this could have become the left subtree of MAY since all identifiers would have been less than MAY. Inserting DECEMBER and JULY necessitates no rebalancing. When FEBRUARYis inserted the tree again becomes unbalanced. The rebalancing process is very similar to that used when JANUARY was inserted. The nearest parent with balance factor ±2 is AUGUST. DECEMBER becomes the new root of that subtree. AUGUST with its left subtree becomes the left subtree. JANUARY and its right subtree becomes the right subtree of DECEMBER, while FEBRUARY becomes the left subtree of JANUARY. If DECEMBER had had a left subtree, it would have become the right subtree of AUGUST. The insertion of JUNE requires the same rebalancing as in figure 9.10 (vi). The rebalancing following the insertion of OCTOBER is identical to that following the insertion of NOVEMBER. Inserting SEPTEMBER leaves the tree balanced.

In the preceding example we saw that the addition of a node to a balanced binary search tree could

unbalance it. The rebalancing was carried out using essentially four different kinds of rotations *LL, RR, LR* and *RL* (figure 9.10 (v), (iii), (vi) and (ix) respectively). *LL* and *RR* are symmetric as are *LR* and *RL*. These rotations are characterized by the nearest ancestor, *A*, of the inserted node, *Y,* whose balance factor becomes ±2. The following characterization of rotation types is obtained:

LL: new node Y is inserted in the left subtree of the left subtree of A

LR: Y is inserted in the right subtree of the left subtree of A

RR: Y is inserted in the right subtree of the right subtree of A

RL: Y is inserted in the left subtree of the right subtree of A

Figures 9.11 and 9.12 show these rotations in terms of abstract binary trees. The root node in each of the trees of the figures represents the nearest ancestor whose balance factor has become ±2 as a result of the insertion. A moment's reflection will show that if a height balanced binary tree becomes unbalanced as a result of an insertion then these are the only four cases possible for rebalancing (if a moment's reflection doesn't convince you, then try exercise 6). In both the example of figure 9.10 and the rotations of figures 9.11 and 9.12, notice that the height of the subtree involved in the rotation is the same after rebalancing as it was before the insertion. This means that once the rebalancing has been carried out on the subtree in question, it is not necessary to examine the remaining tree. The only nodes whose balance factors can change are those in the subtree that is rotated.

In order to be able to carry out the rebalancing of figures 9.11 and 9.12 it is necessary to locate the node *A* around which the rotation is to be performed. As remarked earlier, this is the nearest ancestor of the newly inserted node whose balance factor becomes ±2. In order for a node's balance factor to become ±2, its balance factor must have been ±1 before the insertion. Furthermore, the nearest ancestor whose balance factor becomes ±2 is also the nearest ancestor with balance factor ±1 before the insertion. Therefore, before the insertion, the balance factors of all nodes on the path from *A* to the new insertion point must have been 0. With this information, the node *A* is readily determined to be the nearest ancestor of the new node having a balance factor ±1 before insertion. To complete the rotations the address of *F*, the parent of *A*, is also needed. The changes in the balance factors of the relevant nodes is shown in figures 9.11 and 9.12. Knowing *F* and *A*, all these changes can easily be carried out. What happens when the insertion of a node does not result in an unbalanced tree (figure 9.10 (i), (ii), (iv), (vii), (viii) and (xii))? While no restructuring of the tree is needed, the balance factors of several nodes change. Let *A* be the nearest ancestor of the new node with balance factor ±1 before insertion. If as a result of the insertion the tree did not get unbalanced even though some path length increased by 1, it must be that the new balance factor of *A* is 0. In case there is no ancestor *A* with balance factor ±1 (as in figure 9.10 (i), (ii), (iv), (vii) and (xii)), let *A* be the root. The balance factors of nodes from *A* to the parent of the new node will change to ±(see figure 9.10 (viii), *A* = JANUARY). Note that in both cases the procedure for determining *A* is the same as when rebalancing is needed. The remaining details of the insertion-rebalancing process are spelled out in algorithm AVL-INSERT.

Figure 9.11 Rebalancing rotations of type LL and RR

Figure 9.12 Rebalancing rotations of type LR and RL

Figure 9.12 (continued)

In order to really understand the insertion algorithm, the reader should try it out on the example of figure 9.10. Once you are convinced that it does keep the tree balanced, then the next question is how much time does it take to make an insertion? An analysis of the algorithm reveals that if h is the height of the tree before insertion, then the time to insert a new identifier is $O(h)$. This is the same as for unbalanced binary search trees, though the overhead is significantly greater now. In the case of binary search trees, however, if there were n nodes in the tree, then h could in the worst case be n (figure 9.9) and the worst case insertion time was $O(n)$. In the case of AVL trees, however, h can be at most $O(\log n)$ and so the worst case insertion time is $O(\log n)$. To see this, let N_h be the minimum number of nodes in a height balanced tree of height h. In the worst case, the height of one of the subtrees will be $h - 1$ and of the other $h - 2$. Both these subtrees are also height balanced. Hence, $N_h = N_{h-1} + N_{h-2} + 1$ and $N_0 = 0$, $N_1 = 1$ and $N_2 = 2$. Note the similarity between this recursive definition for N_h and that for the Fibonacci numbers $F_n = F_{n-1} + F_{n-2}$, $F_0 = 0$ and $F_1 = 1$. In fact, we can show (exercise 16) that $N_h = F_{h+2} - 1$ for $h \geq 0$. From Fibonacci number theory it is known that $\square$ where $\square$. Hence, $\square$. This means that if there are n nodes in the tree, then its height, h, is at most $\log (\square$. The worst case insertion time for a height balanced tree with n nodes is, therefore, $O(\log n)$.

```
procedure AVL-INSERT(X,Y,T,)

//the identifier X is inserted into the AVL tree with root T. Each

node is assumed to have an identifier field IDENT, left and right

child fields LCHILD and RCHILD and a two bit balance factor

field BF. BF(P) = height of LCHILD(P) - height of RCHILD(P).

Y is set such that IDENT(Y) = X.//
```

```
//special case: empty tree T = 0.//

if T = 0 then [call GETNODE(Y); IDENT(Y) ← X; T ← Y;

BF(T) ← 0;

LCHILD(T) ← 0; RCHILD(T) ← 0; return]

//Phase 1: Locate insertion point for X. A keeps track of most recent

node with balance factor ±1 and F is the parent of A. Q follows

P through the tree.//

F ← 0; A ← T; P ← T; Q ← 0

while P ≠ 0 do        //search T for insertion point for X//

if BF(P) ≠ 0 then [A ← P; F ← Q]

case

:X < IDENT(P): Q ← P; P ← LCHILD(P)      //take left

branch//

:X > IDENT(P): Q ← P; P ← RCHILD(P)      //take right

branch//

:else: Y ← P; return      //X is in T//

end

end

//Phase 2: Insert and rebalance. X is not in T and may be inserted

as the appropriate child of Q//
```

call *GETNODE(Y); IDENT(Y)* ☐ *X; LCHILD(Y)* ☐ *0;*

RCHILD(Y) ☐ *0; BF(Y)* ☐ *0*

if *X < IDENT(Q)* **then** *LCHILD(Q)* ☐ *Y*   //insert as left child//

else *RCHILD(Q)* ☐ *Y*   //insert as right child//

//adjust balance factors of nodes on path from *A* to *Q*. Note that

by the definition of *A*, all nodes on this path must have balance

factors of 0 and so will change to ±1. *d* = +1 implies *X* is inserted

in left subtree of *A*. *d* = -1 implies *X* is inserted in right subtree

of *A*//

if *X > IDENT(A)* **then** [*P* ☐ *RCHILD(A); B* ☐ *P; d* ☐ *-1*]

else [*P* ☐ *LCHILD(A); B* ☐ *P; d* ☐ *+1*]

while *P* ≠ *Y* **do**

if *X > IDENT(P)*

then [*BF(P)* ☐ *-1; P* ☐ *RCHILD(P)*]   //heigth of right

increases by 1//

else [*BF(P)* ☐ *+1; P* ☐ *LCHILD(P)*]   //height of left

increases by 1//

end

//Is tree unbalanced?//

if *BF(A)* = 0 **then** [*BF(A)* ☐ *d*; **return**   //tree still balanced//

if *BF(A)* + *d* = 0 **then** [*BF(A)* ☐ 0; **return**   //tree is balanced//

//tree unbalanced, determine rotation type//

if *d* = +1 **then**   //left imbalance//

case

:*BF(B)* = +1;   //rotation type *LL*//

LCHILD(A) ☐ *RCHILD(B)*; *RCHILD(B)* ☐ *A*;

BF(A) ☐ *BF(B)* ☐ 0

:**else**:   //type *LR*//

C ☐ *RCHILD(B)*

RCHILD(B) ☐ *LCHILD(C)*

LCHILD(A) ☐ *RCHILD(C)*

LCHILD(C) ☐ *B*

RCHILD(C) ☐ *A*

case

:*BF(C)* = +1: *BF(A)* ☐ -1; *BF(B)* ☐ 0   //*LR(b)*//

:*BF(C)* = -1: *BF(B)* ☐ +1; *BF(A)* ☐ 0   //*LR(c)*//

:**else**:*BF(B)* ☐ 0; *BF(A)* ☐ 0;   //*LR(a)*//

end

BF(C) ☐ 0; *B* ☐ *C*   //B is a new root//

end

else [// right imbalance; this is symmetric to left imbalance//

//and is left as an exercise//]

//subtree with root B has been rebalanced and is the new subtree//

//of *F*. The original subtree of *F* had root *A*

case

:*F* = 0 :*T* ☐ *B*

:*A* = *LCHILD(F)* : *LCHILD(F)* ☐ *B*

:*A* = *RCHILD(F)* : *RCHILD(F)* ☐ *B*

end

end *AVL__INSERT*

Exercises 9-13 show that it is possible to find and delete a node with identifier *X* and to find and delete the kth node from a height balanced tree in $O(\log n)$ time. Results of an empirical study of deletion in height balanced trees may be found in the paper by Karlton, et.al. (see the references). Their study indicates that a random insertion requires no rebalancing, a rebalancing rotation of type *LL* or *RR* and a rebalancing rotation of type *LR* and *RL* with probabilities .5349, .2327 and .2324 respectively. Table 9.12 compares the worst case times of certain operations on sorted sequential lists, sorted linked lists and AVL-trees. Other suitable balanced tree schemes are studied in Chapter 10.

Operation	Sequential List	Linked List	AVL-Tree
Search for X	$O(\log n)$	$O(n)$	$O(\log n)$
Search for kth item	$O(1)$	$O(k)$	$O(\log n)$

Delete X	O(n)	O(1)	O(log n)
		(doubly linked list and position of X known)	
Delete k^{th} item	O(n - k)	O(k)	O(log n)
Insert X	O(n)	O(1)	O(log n)
		(if position for insertion is known)	
Output in order	O(n)	O(n)	O(n)

Table 9.12 Comparison of various structures

9.3 HASH TABLES

In tree tables, the search for an identifier key is carried out via a sequence of comparisons. Hashing differs from this in that the address or location of an identifier, X, is obtained by computing some arithmetic function, f, of X. $f(X)$ gives the address of X in the table. This address will be referred to as the hash or home address of X. The memory available to maintain the symbol table is assumed to be sequential. This memory is referred to as the hash table, HT. The hash table is partitioned into b buckets, $HT(0)$, ...,$HT(b - 1)$. Each bucket is capable of holding s records. Thus, a bucket is said to consist of s slots, each slot being large enough to hold 1 record. Usually $s = 1$ and each bucket can hold exactly 1 record. A hashing function, $f(X)$, is used to perform an identifier transformation on X. $f(X)$ maps the set of possible identifiers onto the integers 0 through $b - 1$. If the identifiers were drawn from the set of all legal Fortran variable names then there would be $T = \sum_{0 \leq i \leq 5} 26 \times 36^i > 1.6 \times 10^9$ distinct possible values for X. Any reasonable program, however, would use far less than all of these identifiers. The ratio n / T is the *identifier density,* while $\boxed{} = n /(sb)$ is the *loading density* or *loading factor.* Since the number of identifiers, n, in use is usually several orders of magnitude less than the total number of possible identifiers, T, the number of buckets b, in the hash table is also much less then T. Therefore, the hash function f must map several different identifiers into the same bucket. Two identifiers I_1, I_2 are said to be *synonyms* with respect to f if $f(I_1) = f(I_2)$. Distinct synonyms are entered into the same bucket so long as all the s slots in that bucket have not been used. An *overflow* is said to occur when a new identifier I is mapped or hashed by f into a full bucket. A *collision* occurs when two nonidentical identifiers are

hashed into the same bucket. When the bucket size *s* is 1, collisions and overflows occur simultaneously.

As an example, let us consider the hash table *HT* with *b* = 26 buckets, each bucket having exactly two slots, i.e., *s* = 2. Assume that there are *n* = 10 distinct identifiers in the program and that each identifier begins with a letter. The loading factor, ☐, for this table is 10/52 = 0.19. The hash function *f* must map each of the possible identifiers into one of the numbers 1-26. If the internal binary representation for the letters *A-Z* corresponds to the numbers 1-26 respectively, then the function *f* defined by: *f(X)* = the first character of *X*; will hash all identifiers *X* into the hash table. The identifiers *GA, D, A, G, L, A2, A1, A3, A4* and *E* will be hashed into buckets 7, 4, 1, 7, 12, 1, 1, 1, 1 and 5 respectively by this function. The identifiers *A, A1, A2, A3* and *A*4 are synonyms. So also are *G* and *GA*. Figure 9.13 shows the identifiers *GA, D, A, G,* and *A2* entered into the hash table. Note that *GA* and *G* are in the same bucket and each bucket has two slots. Similarly, the synonyms *A* and *A2* are in the same bucket. The next identifier, *A1*, hashes into the bucket *HT*(1). This bucket is full and a search of the bucket indicates that A1 is not in the bucket. An overflow has now occurred. Where in the table should *A1* be entered so that it may be retrieved when needed? We will look into overflow handling strategies in section 9.3.2. In the case where no overflows occur, the time required to enter or search for identifiers using hashing depends only on the time required to compute the hash function *f* and the time to search one bucket. Since the bucket size *s* is usually small (for internal tables *s* is usually 1) the search for an identifier within a bucket is carried out using sequential search. The time, then, is independent of *n* the number of identifiers in use. For tree tables, this time was, on the average, log *n.* The hash function in the above example is not very well suited for the use we have in mind because of the very large number of collisions and resulting overflows that occur. This is so because it is not unusual to find programs in which many of the variables begin with the same letter. Ideally, we would like to choose a function *f* which is both easy to compute and results in very few collisions. Since the ratio *b/T* is usually very small, it is impossible to avoid collisions altogether.

☐

Figure 9.13 Has table with 26 buckets and two slots per bucket

In summary, hashing schemes perform an identifier transformation through the use of a hash function *f.* It is desirable to choose a function *f* which is easily computed and also minimizes the number of collisions. Since the size of the identifier space, *T,* is usually several orders of magnitude larger than the number of buckets *b,* and *s* is small, overflows necessarily occur. Hence a mechanism to handle overflows is also needed.

9.3.1 Hashing Functions

A hashing function, *f*, transforms an identifier *X* into a bucket address in the hash table. As mentioned earlier the desired properties of such a function are that it be easily computable and that it minimize the number of collisions. A function such as the one discussed earlier is not a very good choice for a hash function for symbol tables even though it is fairly easy to compute. The reason for this is that it depends

only on the first character in the identifier. Since many programs use several identifiers with the same first letter, we expect several collisions to occur. In general, then, we would like the function to depend upon all the characters in the identifiers. In addition, we would like the hash function to be such that it does not result in a biased use of the hash table for random inputs. I.e., if *X* is an identifier chosen at random from the identifier space, then we want the probability that $f(X) = i$ to be *1/b* for all buckets *i*. Then a random *X* has an equal chance of hashing into any of the *b* buckets. A hash function satisfying this property will be termed a *uniform hash function.*

Several kinds of uniform hash functions are in use. We shall describe four of these.

(i) Mid-Square

One hash function that has found much use in symbol table applications is the 'middle of square' function. This function, f_m, is computed by squaring the identifier and then using an appropriate number of bits from the middle of the square to obtain the bucket address; the identifier is assumed to fit into one computer word. Since the middle bits of the square will usually depend upon all of the characters in the identifier, it is expected that different identifiers would result in different hash addresses with high probability even when some of the characters are the same. Figure 9.14 shows the bit configurations resulting from squaring some sample identifiers. The number of bits to be used to obtain the bucket address depends on the table size. If *r* bits are used, the range of values is 2^r, so the size of hash tables is chosen to be a power of 2 when this kind of scheme is used.

IDENTIFIER	INTERNAL REPRESENTATION	
X	X	X^2
A	1	1
A1	134	20420
A2	135	20711
A3	136	21204
A4	137	21501
A9	144	23420
B	2	4

```
C                           3                              11

G                           7                              61

DMAX            4150130              21526443617100

DMAX1          415013034           5264473522151420

AMAX            1150130              1345423617100

AMAX1          115013034           3454246522151420
```

Figure 9.14 Internal representations of X and X^2 in octal notation. X is input right justified, zero filled, six bits or 2 octal digits per character.

(ii) Division

Another simple choice for a hash function is obtained by using the modulo (**mod**) operator. The identifier X is divided by some number M and the remainder is used as the hash address for X.

$$f_D(X) = X \textbf{ mod } M$$

This gives bucket addresses in the range 0 - (M - 1) and so the hash table is at least of size $b = M$. The choice of M is critical. If M is a power of 2, then $f_D(X)$ depends only on the least significant bits of X. For instance, if each character is represented by six bits and identifiers are stored right justified in a 60-bit word with leading bits filled with zeros (figure 9.15) then with $M = 2^i$, $i \le 6$ the identifiers $A1$, $B1$, $C1$, $X41$, DNTXY1, etc. all have the same bucket address. With $M = 2^i$, $i \le 12$ the identifiers AXY, BXY, WTXY, etc. have the same bucket address. Since programmers have a tendency to use many variables with the same suffix, the choice of M as a power of 2 would result in many collisions. This choice of M would have even more disastrous results if the identifier X is stored left justified zero filled. Then, all 1 character identifiers would map to the same bucket, 0, for $M = 2^i$, $i \le 54$; all 2 character identifiers would map to the bucket 0 for $M = 2^i$, $i \le 48$ etc. As a result of this observation, we see that when the division function f_D is used as a hash function, the table size should not be a power of 2 while when the "middle of square" function f_m is used the table size is a power of 2. If M is divisible by 2 then odd keys are mapped to odd buckets (as the remainder is odd) and even keys are mapped to even buckets. The use of the hash table is thus biased.

Figure 9.15 Identifier AI right and left justified and zero filled. (6 bits per character)

Let us try some other values for M and see what kind of identifiers get mapped to the same bucket. The goal being that we wish to avoid a choice of M that will lead to many collisions. This kind of an analysis is possible as we have some idea as to the relationships between different variable names programmers tend to use. For instance, the knowledge that a program tends to have variables with the same suffix led us to reject $M = 2^i$. For similar reasons even values of M prove undesirable. Let $X = x_1 x_2$ and $Y = x_2 x_1$ be two identifiers each consisting of the characters x_1 and x_2. If the internal binary representation of x_1 has value $C(x_1)$ and that for x_2 has value $C(x_2)$ then if each character is represented by 6 bits, the numeric value of X is $2^6 C(x_1) + C(x_2)$ while that for Y is $2^6 C(x_2) + C(x_1)$. If p is a prime number dividing M then $(f_D(X) - f_D(Y)) \bmod p = (2^6 C(x_1) \bmod p + C(x_2) \bmod p - 2^6 C(X_2) \bmod p - C(x_1) \bmod p) \bmod p$. If $p = 3$, then

```
(f_D(X) - f_D (Y))mod p

= (64 mod 3 C(x_1)mod 3 + C(x_2)mod 3

- 64 mod 3 C(x_2)mod 3 - C(x_1)mod 3) mod 3

= C(x_1)mod 3 + C(x_2)mod 3 - C(x_2)mod 3 - C(x_1)mod 3

= 0 mod 3.
```

I.e., permutations of the same set of characters are hashed at a distance a factor of 3 apart. Programs in which many variables are permutations of each other would again result in a biased use of the table and hence result in many collisions. This happens because 64 mod 3 = 1. The same behavior can be expected when 7 divides M as 64 mod 7 = 1. These difficulties can be avoided by choosing M a prime number. Then, the only factors of M are M and 1. Knuth has shown that when M divides $r^k \pm a$ where k and a are small numbers and r is the radix of the character set (in the above example $r = 64$), then X mod M tends to be a simple superposition of the characters in X. Thus, a good choice for M would be: *M a prime number such that M does not divide $r^k \pm a$ for small k and a.* In section 9.3.2 we shall see other reasons for choosing M a prime number. In practice it has been observed that it is sufficient to choose M such that it has no prime divisors less than 20.

(iii) Folding

In this method the identifier X is partitioned into several parts, all but the last being of the same length. These parts are then added together to obtain the hash address for X. There are two ways of carrying out this addition. In the first, all but the last part are shifted so that the least significant bit of each part lines up with the corresponding bit of the last part (figure 9.16(a)). The different parts are now added together to get $f(X)$. This method is known as *shift folding.* The other method of adding the parts is *folding at the boundaries.* In this method, the identifier is folded at the part boundaries and digits falling into the same

position are added together (figure 9.16(b)) to obtain *f*(*X*).

Figure 9.16 Two methods of folding

(iv) Digit Analysis

This method is particularly useful in the case of a static file where all the identifiers in the table are known in advance. Each identifier *X* is interpreted as a number using some radix *r*. The same radix is used for all the identifiers in the table. Using this radix, the digits of each identifier are examined. Digits having the most skewed distributions are deleted. Enough digits are deleted so that the number of digits left is small enough to give an address in the range of the hash table. The criterion used to find the digits to be used as addresses, based on the measure of uniformity in the distribution of values in each digit, is to keep those digits having no abnormally high peaks or valleys and those having small standard deviation. The same digits are used for all identifiers.

Experimental results presented in section 9.3.2 suggest the use of the division method with a divisor *M* that has no prime factors less than 20 for general purpose applications.

9.3.2 Overflow Handling

In order to be able to detect collisions and overflows, it is necessary to initialize the hash table, *HT,* to represent the situation when all slots are empty. Assuming that no record has identifier zero, then all slots may be initialized to zero.* When a new identifier gets hashed into a full bucket, it is necessary to find another bucket for this identifier.The simplest solution would probably be to find the closest unfilled bucket. Let us illustrate this on a 26-bucket table with one slot per bucket. Assume the identifiers to be *GA, D, A, G, L, A2, A1, A3, A4, Z, ZA, E.* For simplicity we choose the hash function *f* (*X*) = first character of *X*. Initially, all the entries in the table are zero. *f*(*GA*) = 7, this bucket is empty, so *GA* and any other information making up the record are entered into *HT*(7). *D* and *A* get entered into the buckets *HT*(4) and *HT*(1) respectively. The next identifier *G* has *f*(*G*) = 7. This slot is already used by *GA.* The next vacant slot is *HT*(8) and so *G* is entered there. *L* enters *HT*(12). *A2* collides with *A* at *HT* (1), the bucket overflows and *A2* is entered at the next vacant slot *HT*(2). *A1, A3* and *A4* are entered at *HT*(3), *HT*(5) and *HT*(6) respectively. *Z* is entered at *HT*(26), *ZA* at *HT*(9), (the hash table is used circularly), and *E* collides with *A3* at *HT*(5) and is eventually entered at *HT*(10). Figure 9.17 shows the resulting table. This method of resolving overflows is known as *linear probing* or *linear open addressing*.

*A clever way to avoid initializing the hash table has been discovered by T. Gonzalez (see exercise 22).

Figure 9.17 Hash table with linear probing. 26 buckets, 1 slot per bucket

In order to search the table for an identifier, X, it is necessary to first compute $f(X)$ and then examine keys at positions $HT(f(X))$, $HT(f(X) + 1)$, ...,$HT(f(X) + j)$ such that $HT(f(X) + j)$ either equals X (X is in the table) or 0 (X is not in the table) or we eventually return to $HT(f(X))$ (the table is full).

```
procedure LINSRCH (X,HT,b,j)

//search the hash table HT(0: b - 1) (each bucket has exactly 1

slot) using linear probing. If HT (j = 0 then the j-th bucket is empty

and X can be entered into the table. Otherwise HT (j) = X which

is already in the table. f is the hash function//

i ☐ f(X); j ☐ i              // compute hash address for X//

while HT(j) ≠ X and HT(j) ≠ 0 do

j ☐ (j + 1) mod b          //treat the table as circular//

if j = i then call TABLE_FULL                //no empty slots//

end

end LINSRCH
```

Our earlier example shows that when linear probing is used to resolve overflows, identifiers tend to cluster together, and moreover, adjacent clusters tend to coalesce, thus increasing the search time. To locate the identifier, ZA, in the table of figure 9.17, it is necessary to examine $HT(26)$, $HT(1)$, ...,$HT(9)$, a total of ten comparisons. This is far worse than the worst case behavior for tree tables. If each of the identifiers in the table of figure 9.12 was retrieved exactly once, then the number of buckets examined would be 1 for A, 2 for $A2$, 3 for $A1$, 1 for D, 5 for $A3$, 6 for $A4$, 1 for GA, 2 for G, 10 for ZA, 6 for E, 1 for L and 1 for Z for a total of 39 buckets examined. The average number examined is 3.25 buckets per identifier. An analysis of the method shows that the expected average number of identifier comparisons, P, to look up an identifier is approximately (2 - ☐) / (2 - 2☐) where ☐ is the loading density. This is the average over all possible sets of identifiers yielding the given loading density and using a uniform function f. In the above example ☐ = 12/26 = .47 and P = 1.5. Even though the average number of probes is small, the worst case can be quite large.

One of the problems with linear open addressing is that it tends to create clusters of identifiers. Moreover, these clusters tend to merge as more identifiers are entered, leading to bigger clusters. Some improvement in the growth of clusters and hence in the average number of probes needed for searching can be obtained by *quadratic probing*. Linear probing was characterized by searching the buckets $(f(X) + i)$ mod b; $0 \leq i \leq b - 1$ where b is the number of buckets in the table. In quadratic probing, a quadratic function of i is used as the increment. In particular, the search is carried out by examining buckets $f(X)$, $(f(X) + i^2)$ mod b and $(f(X) - i^2)$ mod b for $1 \leq i \leq (b - 1)/2$. When b is a prime number of the form $4j + 3$, for j an integer, the quadratic search described above examines every bucket in the table. The proof that when b is of the form $4j + 3$, quadratic probing examines all the buckets 0 to $b - 1$ relies on some results from number theory. We shall not go into the proof here. The interested reader should see Radke [1970] for a proof. Table 9.18 lists some primes of the form $4j + 3$. Another possibility is to use a series of hash functions $f_1, f_2, ..., f_m$. This method is known as *rehashing*. Buckets $f_i(X)$, $1 \leq i \leq m$ are examined in that order. An alternate method for handling bucket overflow, random probing, is discussed in exercise 19.

Prime	j	Prime	j
3	0	43	10
7	1	59	14
11	2	127	31
19	4	251	62
23	5	503	125
31	7	1019	254

Table 9.18 Some primes of the form 4j + 3

One of the reasons linear probing and its variations perform poorly is that searching for an identifier involves comparison of identifiers with different hash values. In the hash table of figure 9.17, for instance, searching for the identifier *ZA* involved comparisons with the buckets $HT(1)$ to $HT(8)$, even though none of the identifiers in these buckets had a collision with $HT(26)$ and so could not possibly be *ZA*. Many of the comparisons being made could be saved if we maintained lists of identifiers, one list per bucket, each list containing all the synonyms for that bucket. If this were done, a search would then involve computing the hash address $f(X)$ and examining only those identifiers in the list for $f(X)$. Since the sizes of these lists is not known in advance, the best way to maintain them is as linked chains. Additional space for a link is required in each slot. Each chain will have a head node. The head node,

however, will usually be much smaller than the other nodes since it has to retain only a link. Since the lists are to be accessed at random, the head nodes should be sequential. We assume they are numbered 1 to *M* if the hash function *f* has range 1 to *M*.

☐

Figure 9.19. Hash chains corresponding to Figure 9.17.

Using chaining to resolve collisions and the hash function used to obtain figure 9.17, the hash chains of figure 9.19 are obtained. When a new identifier, *X*, is being inserted into a chain, the insertion can be made at either end. This is so because the address of the last node in the chain is known as a result of the search that determined *X* was not in the list for *f*(*X*). In the example of figure 9.19 new identifiers were inserted at the front of the chains. The number of probes needed to search for any of the identifiers is now 1 for each of *A*4, *D*, *E*, *G*, *L*, and *ZA*; 2 for each of *A*3, *GA* and *Z*; 3 for *A*1: 4 for *A*2 and 5 for *A* for a total of 24. The average is now 2 which is considerably less than for linear probing. Additional storage, however, is needed for links.

procedure *CHNSRCH (X,HT,b,j)*

//search the hash table *HT* (0:*b* – 1) for *X*. Either *HT*(*i*) = 0 or it

is a pointer to the list of identifiers *X*: *f*(*X*) = *i*. List nodes have

field *IDENT* and *LINK*. Either *j* points to the node already containing

X or *j* = 0//

j ☐ *HT* (*f*(*X*))   //compute head node address//

//search the chain starting at *j*//

while *j* $\neq$ 0 **and** *IDENT* (*j*) $\neq$ *X* **do**

j ☐ *LINK* (*j*)

end

end *CHNSRCH*

The expected number of identifier comparison can be shown to be ☐ where ☐ is the loading density *n /*

b (b = number of head nodes). For □ = 0.5 this figure is 1.25 and for □ = 1 it is about 1.5. This scheme has the additional advantage that only the b head nodes must be sequential and reserved at the beginning. Each head node, however, will be at most 1/2 to 1 word long. The other nodes will be much bigger and need be allocated only as needed. This could represent an overall reduction in space required for certain loading densities despite the links. If each record in the table is five words long, $n = 100$ and □ = 0.5, then the hash table will be of size 200 x 5 = 1000 words. Only 500 of these are used as □ = 0.5. On the other hand, if chaining is used with one full word per link, then 200 words are needed for the head nodes ($b = 200$). Each head node is one word long. One hundred nodes of six words each are needed for the records. The total space needed is thus 800 words, or 20% less than when no chaining was being used. Of course, when □ is close to 1, chaining uses more space than linear probing.

However, when □ is close to 1, the average number of probes using linear probing or its variations becomes quite large and the additional space used for chaining can be justified by the reduction in the expected number of probes needed for retrieval. If one wishes to delete an entry from the table, then this can be done by just removing that node from its chain. The problem of deleting entries while using open addressing to resolve collisions is tackled in exercise 17.

□

Figure 9.20 Average number of bucket accesses per identifier retrieved (condensed from Lum, Yuen and Dodd, "Key-to-Address Transform Techniques: A Fundamental Performance Study on Large Existing Formatted Files," CACM, April 1971, Vol. 14, No. 4, pp. 228-239.)

The results of this section tend to imply that the performance of a hash table depends only on the method used to handle overflows and is independent of the hash function so long as a uniform hash function is being used. While this is true when the identifiers are selected at random from the identifier space, it is not true in practice. In practice, there is a tendency to make a biased use of identifiers. Many identifiers in use have a common suffix or prefix or are simple permutations of other identifiers. Hence, in practice we would expect different hash functions to result in different hash table performance. The table of figure 9.20 presents the results of an empirical study conducted by Lum, Yuen and Dodd. The values in each column give the average number of bucket accesses made in searching eight different tables with 33,575; 24,050; 4909; 3072; 2241; 930; 762 and 500 identifiers each. As expected, chaining outperforms linear open addressing as a method for overflow handling. In looking over the figures for the various hash functions, we see that division is generally superior to the other types of hash functions. For a general application, it is therefore recommended that the division method be used. The divisor should be a prime number, though it is sufficient to choose a divisor that has no prime factors less than 20. The table also gives the theoretical expected number of bucket accesses based on random keys.

9.3.3 THEORETICAL EVALUATION OF OVERFLOW TECHNIQUES

The experimental evaluation of hashing techniques indicates a very good performance over conventional techniques such as balanced trees. The worst case performance for hashing can, however, be very bad. In the worst case an insertion or a search in a hash table with n identifiers may take $O(n)$ time. In this section we present a probabilistic analysis for the expected performance of the chaining method and state without proof the results of similar analyses for the other overflow handling methods. First, we formalize what we mean by expected performance.

Let $HT(0: b$ - $1)$ be a hash table with b buckets, each bucket having one slot. Let f be a uniform hash function with range $[0, b$ - $1]$. If n identifiers $X_1, X_2, ..., X_n$ are entered into the hash table then there are b^n distinct hash sequences $f(X_1), f(X_2), ..., f(X_n)$. Assume that each of these is equally likely to occur. Let S_n denote the expected number of identifier comparisons needed to locate a randomly chosen X_i, $1 \leq i \leq n$. Then, S_n is the average number of comparisons needed to find the j'th key X_j; averaged over $1 \leq j \leq n$ with each j equally likely and averaged over all b^n hash sequences assuming each of these to also be equally likely. Let U_n be the expected number of identifier comparisons when a search is made for an identifier not in the hash table. This hash table contains n identifiers. The quantity U_n may be defined in a manner analogous to that used for S_n.

Theorem 9.1 Let $\square = n/b$ be the loading density of a hash table using a uniform hashing function f. Then:

(i) for linear open addressing

$\square$

(ii) for rehashing, random probing and quadratic probing

$\square$

(iii) for chaining

$U_n \square \square$

$S_n \square 1 + \square/2$

Proof Exact derivations of U_n and S_n are fairly involved and can be found in Knuth's book: The Art of Computer Programming: Sorting and Searching. Here, we present a derivation of the approximate formulas for chaining. First, we must make clear our count for U_n and S_n. In case the identifier X being searched for has $f(X) = i$ and chain i has k nodes on it (not including the head node) then k comparisons are needed if X is not on the chain. If X is j nodes away from the head node, $1 \leq j \leq k$ then j comparisons

are needed.

When the n identifiers distribute uniformly over the b possible chains, the expected number in each chain is $n/b=$ ☐. Since, $U_n =$ expected number of identifiers on a chain, we get $U_n =$ ☐.

When the i'th identifier, X_i, is being entered into the table, the expected number of identifiers on any chain is $(i - 1)/b$. Hence, the expected number of comparisons needed to search for X_i after all n identifiers have been entered is $1 + (i - 1)/b$ (this assumes that new entries will be made at the end of the chain). We therefore get:

☐

REFERENCES

The $O(n^2)$ optimum binary search tree algorithm is from:

"Optimum Binary Search Trees" by D. Knuth, *Acta Informatica,* vol. 1, Fasc 1, 1971, pp. 14-25.

For a discussion of heuristics that obtain in $O(n\log n)$ time nearly optimal binary search trees see:

"Nearly Optimal Binary Search Trees," by K. Melhorn, *Acta Informatica,* vol. 5, pp. 287-295, 1975.

"Binary Search Trees and File Organization," by J. Nievergelt, *ACM Computing Surveys*, vol. 6, no. 3, Sept. 1974, pp. 195-207.

For more on Huffman codes see: An Optimum Encoding with Minimum Longest Code and Total Number of Digits, by E. Schwartz, *Information and Control*, vol. 7, 1964, pp. 37-44.

Additional algorithms to manipulate AVL trees may be found in:

"Linear lists and priority queues as balanced binary trees" by C. Crane, STAN-CS-72-259, Computer Science Department, Stanford University, February 1972.

The art of computer programming: sorting and searching by D.Knuth, Addiso-Wesley, Reading, Massachusetts. 1973 (section 6.2.3).

Results of an empirical study on height balanced trees appear in:

"Performance of Height-Balanced Trees," by P. L. Karlton, S. H. Fuller, R. E. Scroggs and E. B. Koehler, *CACM,* vol. 19, no. 1, Jan. 1976, pp. 23-28.

Several interesting and enlightening works on hash tables exist. Some of these are:

"Scatter storage techniques" by R. Morris, *CACM,* vol. 11, no. 1, January 1968, pp. 38-44.

"Key to Address Transform Techniques: A Fundamental Performance Study on Large Existing Formatted Files" by V. Lum, P. Yuen and M. Dodd, *CACM*, vol. 14, no. 4, April 1971, pp. 228-239.

"The quadratic quotient method: a hash code eliminating secondary clustering" by J. Bell, *CACM*, vol. 13, no. 2, February 1970, pp. 107-109.

"Full table quadratic searching for scatter storage" by A. Day, *CACM*, vol. 13, no. 8, August 1970, pp. 481-482.

"Identifier search mechanisms: a survey and generalized model" by D. Severence, *ACM Computing Surveys*, vol. 6, no. 3, September 1974, pp. 175-194.

"A practitioners guide to addressing algorithms: a collection of reference tables and rules-of-thumb" by D. Severence and R. Duhne, Technical Report No. 240, Department of Operations Research, Cornell University, November 1974.

"Hash table methods" by W. Mauer and T. Lewis, *ACM Computing Surveys*, vol. 7, no. 1, March 1975, pp. 5-20.

"The quadratic hash method when the table size is not prime" by V. Batagelj, *CACM*, vol. 18, no. 4, April 1975, pp. 216-217.

The art of computer programming: sorting and searching by D. Knuth, Addison-Wesley, Reading, Massachusetts, 1973.

"Reducing the retrieval time of scatter storage techniques" by R. Brent, *CACM,* vol. 16, no. 2, February 1973, pp. 105-109.

"General performance analysis of key-to-address transformation methods using an abstract file concept" by V. Lum, *CACM*, vol. 16, no. 10, October 1973, pp. 603-612.

"The use of quadratic residue research," by C. E. Radke, *CACM*, vol. 13, no. 2, Feb. 1970, pp. 103-105.

A method to avoid hash table initialization may be found in:

"Algorithms for sets and related problems," by T. Gonzalez, University of Oklahoma, Nov. 1975.

EXERCISES

1. (a) Prove by induction that if T is a binary tree with n internal nodes, I its internal path length and E its external path length, then $E = I + 2n$, $n \geq 0$.

(b) Using the result of (a) show that the average number of comparisons s in a successful search is related to the average number of comparisons, u, in an unsuccessful search by the formula

$s = (1 + 1/n)u - 1$, $n \geq 1$.

2. (a) Show that algorithm HUFFMAN correctly generates a binary tree of minimal weighted external path length.

(b) When n runs are to be merged together using an m-way merge, Huffman's method generalizes to the following rule: "First add $(1 - n)$ mod $(m - 1)$ runs of length zero to the set of runs. Then repeatedly merge together the m shortest remaining runs until only one run is left." Show that this rule yields an optimal merge pattern for m-way merging.

3. Using algorithm OBST compute w_{ij}, r_{ij} and c_{ij}, $0 \leq i < j \leq 4$ for the identifier set $(a_1, a_2, a_3, a_4) = $ (**end, goto, print, stop**) with $p_1 = 1/20$, $p_2 = 1/5$, $p_3 = 1/10$, $p_4 = 1/20$, $q_o = 1/5$, $q_1 = 1/10$, $q_2 = 1/5$, $q_3 = 1/20$, $q_4 = 1/20$. Using the r_{ij}'s construct the optimal binary search tree.

4. (a) Show that the computing time of algorithm OBST is $O(n^2)$.

(b) Write an algorithm to construct the optimal binary search tree T_{on} given the roots r_{ij}, $0 \leq i \leq j \leq n$. *Show that this can be done in time* $O(n)$.

5. Since, often, only the approximate values of the p's and q's are known, it is perhaps just as meaningful to find a binary search tree that is nearly optimal i.e. its cost, eq. (9.1), is almost minimal for the given p's and q's. This exercise explores an $O(n\log n)$ algorithm that results in nearly optimal binary search trees. The search tree heuristic we shall study is:

Choose the root A_k such that $| w_{0,k-1} - w_{k,n}|$ is as small as possible. Repeat this procedure to find the left and right subtrees of A_k.

(a) Using this heuristic obtain the resulting binary search tree for the data of exercise 3. What is its cost?

(b) Write a SPARKS algorithm implementing the above heuristic. Your algorithm should have a time complexity of at most $O(n \log n)$.

An analysis of the performance of this heuristic may be found in the paper by Melhorn.

6. (a) Convince yourself that Figures 9.11 and 9.12 take care of all the possible situations that may arise when a height balanced binary tree becomes unbalanced as a result of an insertion. Alternately come up with an example that is not covered by any of the cases in Figures 9.11 and 9.12.

(b) Complete Figure 9.12 by drawing the tree configurations for the rotations *RL* (a), (b) and (c).

7. Complete algorithm AVL--INSERT by filling in the steps needed to rebalance the tree in case the imbalance was of type *RL*.

8. Obtain the height balanced trees corresponding to those of Figure 9.10 using algorithm AVL--INSERT, starting with an empty tree, on the following sequence of insertions:

DECEMBER, JANUARY, APRIL, MARCH, JULY, AUGUST, OCTOBER, FEBRUARY, NOVEMBER, MAY, JUNE.

Label the rotations according to type.

9. Assume that each node in an AVL tree *T* has the field LSIZE. For any node, *A*, LSIZE(*A*) is the number of nodes in its left subtree +1. Write an algorithm AVL--FINDK(*T,k*) to locate the kth smallest identifier in the subtree *T*. Show that this can be done in $O(\log n)$ time if there are *n* nodes in *T*.

10. Rewrite algorithm AVL--INSERT with the added assumption that each node has a LSIZE field as in exercise 9. Show that the insertion time remains $O(\log n)$.

11. Write an algorithm to list the nodes of an AVL-tree *T* in ascending order of IDENT fields. Show that this can be done in $O(n)$ time if *T* has *n* nodes.

12. Write an algorithm to delete the node with identifier *X* from an AVL-tree *T*. The resulting tree should be restructured if necessary. Show that the time required for this is $O(\log n)$ when there are *n* nodes in *T*.

13. Do exercise 12 for the case when each node has a LSIZE field and the kth smallest identifier is to be deleted.

14. Write an algorithm to merge the nodes of the two AVL-trees T_1 and T_2 together. What is the computing time of your algorithm?

15. Write an algorithm to split an AVL tree, *T,* into two AVL trees T_1 and T_2 such that all identifiers in

T_1 are $\leq X$ and all those in T_2 are $> X$.

16. Prove by induction that the minimum number of nodes in an AVL tree of height h is $N_h = F_{h+2} - 1$, $h \geq 0$.

17. Write an algorithm to delete identifier X from a hash table which uses hash function f and linear open addressing to resolve collisions. Show that simply setting the slot previously occupied by X to 0 does not solve the problem. How must algorithm LINSRCH be modified so that a correct search is made in the situation when deletions are permitted? Where can a new identifier be inserted?

18. (i) Show that if quadratic searching is carried out in the sequence $(f(x) + q^2)$, $f(x) + (q - 1)^2)$,...,$(f(x) + 1)$,$f(x)$,$(f(x) - 1)$, ...,$(f(x) - q^2)$ with $q = (b - 1)12$ then the address difference mod b between successive buckets being examined is

$b - 2, b - 4, b - 6, ...,5, 3, 1, 1, 3, 5, ...,b - 6, b - 4, b - 2$

(ii) Write an algorithm to search a hash table HT $(0,b - 1)$ of size b for the identifier X. Use f as the hash function and the quadratic probe scheme discussed in the text to resolve. In case X is not in the table, it is to be entered. Use the results of part (i) to reduce the computations.

19. [Morris 1968] In random probing, the search for an identifier, X, in a hash table with b buckets is carried out by examining buckets $f(x)$, $f(x) + S(i)$, $1 \leq i \leq b - 1$ where $S(i)$ is a pseudo random number. The random number generator must satisfy the property that every number from 1 to $b - 1$ must be generated exactly once. (i) Show that for a table of size 2^r, the following sequence of computations generates numbers with this property:

Initialize R to 1 each time the search routine is called.

On successive calls for a random number do the following:

$R \boxed{} R * 5$

$R \boxed{}$ *low order* r + *2 bits of* R

$S(i) \boxed{} R/4$

(ii) Write an algorithm, incorporating the above random number generator, to search and insert into a hash table using random probing and the middle of square hash function, f_m.

It can be shown that for this method, the expected value for the average number of comparisons needed

to search for X is - $(1/\boxed{})\log(1 - \boxed{})$ for large table sizes. $\boxed{}$ is the loading factor.

20. Write an algorithm to list all the identifiers in a hash table in lexicographic order. Assume the hash function f is $f(X) = $ first character of X and linear probing is used. How much time does your algorithm take?

21. Let the binary representation of identifier X be x_1x_2. Let $|x|$ denote the number of bits in x and let the first bit of x_1 be 1. Let $|x_1| = \lceil |x|/2 \rceil$ and $|x_2| = \lfloor |x|/2 \rfloor$. Consider the following hash function

$f(X) = $ middle k bits of $(x_1 \; XOR \; x_2)$

where *XOR* is exclusive or. Is this a uniform hash function if identifiers are drawn at random from the space of allowable FORTRAN identifiers? What can you say about the behavior of this hash function in a real symbol table usage?

22. [T. Gonzalez] Design a symbol table representation which allows one to search, insert and delete an identifier X in $O(1)$ time. Assume that $1 \leq X \leq m$ and that $m + n$ units of space are available where n is the number of insertions to be made (Hint: use two arrays $A(1:n)$ and $B(1:m)$ where $A(i)$ will be the ith identifier inserted into the table. If X is the ith identifier inserted then $B(X) = i$.). Write algorithms to search, insert and delete identifiers. Note that you cannot initialize either A or B to zero as this would take $O(n + m)$ time. Note that X is an integer.

23. [T. Gonzalez] Let $S = \{x_1, x_2, ..., x_n\}$ and $T = \{y_1, y_2, ..., y_r\}$ be two sets. Assume $1 \leq x_i \leq m$, $1 \leq i \leq n$ and $1 \leq y_i \, m$, $1 \leq i \leq r$. Using the idea of exercise 22 write an algorithm to determine if $S \subseteq T$. Your algorithm should work in $O(r + n)$ time. since $S \boxed{} T$ iff $S \subseteq T$ *and* $T \subseteq S$, this implies that one can determine in linear time if two sets are equivalent. How much space is needed by your algorithm?

24. [T. Gonzalez] Using the idea of exercise 22 write an $O(n + m)$ time algorithm to carry out the function of algorithm VERIFY2 of section 7.1. How much space does your algorithm need?

25. Complete table 9.12 by adding a column for hashing.

26. For a fixed $k, k \geq 1$ we define a height balanced tree $HB(k)$ as below:

Definition An empty binary tree is a $HB(k)$ tree. If T is a non-empty binary tree with T_L and T_R as its left and right subtrees, then T is $HB(k)$ iff (i) T_L and T_R are $HB(k)$ and (ii) $|h_L - h_R| \leq k$ where h_L and h_R are the heights of T_L and T_R respectively.

For the case of $HB(2)$ trees obtain the rebalancing transformations.

27. Write an insertion algorithm for *HB*(2) trees.

28. Using the notation of section 9.3.3 show that when linear open addressing is used:

☐

Using this equation and the approximate equality:

☐

☐

29. [Guttag] The following set of operations define a symbol table which handles a language with block structure. Give a set of axioms for these operations:

INIT--creates an empty table;

ENTERB--indicates a new block has been entered;

ADD--places an identifier and its attributes in the table;

LEAVEB--deletes all identifiers which are defined in the innermost block;

RETRIEVE--returns the attributes of the most recently defined identifier;

ISINB--returns true if the identifier is defined in the innermost block else false.

◀ ■ ▶

CHAPTER 10: FILES

10.1 FILES, QUERIES AND SEQUENTIAL ORGANIZATIONS

A file, as described in earlier chapters, is a collection of records where each record consists of one or more fields. For example, the records in an employee file could contain these fields:

Employee Number (E#)

Name

Occupation

Degree (Highest Degree Obtained)

Sex

Location

Marital Status (MS)

Salary

Sample data for such a file is provided in figure 10.1.

Record	E#	Name	Occupation	Degree	Sex	Location	MS	Salary
A	800	HAWKINS	programmer	B.S.	M	Los Angeles	S	10,000
B	510	WILLIAMS	analyst	B.S.	F	Los Angeles	M	15,000
C	950	FRAWLEY	analyst	M.S.	F	Minneapolis	S	12,000
D	750	AUSTIN	programmer	B.S.	F	Los Angeles	S	12,000

```
E    620  MESSER    programmer   B.S.   M   Minneapolis  M    9,000
```

Figure 10.1 Sample Data for Employee File

The primary objective of file organization is to provide a means for record retrieval and update. The update of a record could involve its deletion, changes in some of its fields or the insertion of an entirely new record. Certain fields in the record are designated as key fields. Records may be retrieved by specifying values for some or all of these keys. A combination of key values specified for retrieval will be termed a *query*. Let us assume that in the employee file of figure 10.1. the fields Employee Number, Occupation, Sex and Salary have been designated as key fields. Then, some of the valid queries to the file are:

Retrieve the records of all employees with

Q1: Sex = M

Q2: Salary > 9,000

Q3: Salary > average salary of all employees

Q4: (Sex = M *and* Occupation = Programmer) *or* (Employee Number > 700 and Sex = F)

One invalid query to the file would be to specify location = Los Angeles, as the location field was not designated as a key field. While it might appear that the functions one wishes to perform on a file are the same as those performed on a symbol table (chapter 9), several complications are introduced by the fact that the files we are considering in this chapter are too large to be held in internal memory. The tables of chapter 9 were small enough that all processing could be carried out without accessing external storage devices such as disks and tapes.

In this chapter we are concerned with obtaining data representations for files on external storage devices so that required functions (e.g. retrieval, update) may be carried out efficiently. The particular organization most suitable for any application will depend upon such factors as the kind of external storage device available, type of queries allowed, number of keys, mode of retrieval and mode of update. Let us elaborate on these factors.

Storage Device Types

We shall be concerned primarily with direct access storage devices, (DASD) as exemplified by disks. Some discussion of tape files will also be made.

Query Types

The examples Q1 - Q4 above typify the kinds of queries one may wish to make. The four query types are:

Q1: Simple Query: The value of a single key is specified.

Q2: Range Query: A range of values for a single key is specified

Q3: Functional Query: Some function of key values in the file is specified (e.g. average or median)

Q4: Boolean Query: A boolean combination of Q1 - Q3 using logical operators *and, or, not.*

Number of Keys

The chief distinction here will be between files having only one key and files with more than one key.

Mode of Retrieval

The mode of retrieval may be either real time or batched. In real time retrieval the response time for any query should be minimal (say a few seconds from the time the query is made). In a bank the accounts file has a mode of retrieval which is real time since requests to determine a balance must be satisfied quickly. Similarly, in an airline reservation system we must be able to determine the status of a flight (i.e. number of seats vacant) in a matter of a few seconds. In the batched mode of retrieval, the response time is not very significant. Requests for retrieval are batched together on a "transaction file" until either enough requests have been received or a suitable amount of time has passed. Then all queries on the transaction file are processed.

Mode of Update

The mode of update may, again, be either real time or batched. Real time update is needed, for example, in a reservation system. As soon as a seat on a flight is reserved, the file must be changed to indicate the new status. Batched update would be suitable in a bank account system where all deposits and withdrawals made on a particular day could be collected on a transaction file and the updates made at the end of the day. In the case of batched update one may consider two files: the Master File and the Transactions File. The Master File represents the file status after the previous update run. The transaction file holds all update requests that haven't yet been reflected in the master file. Thus, in the case of batched update, the master file is always 'out of date' to the extent that update requests have been batched on the transaction file. In the case of a bank file using real time retrieval and batched update, this would mean that only account balances at the end of the previous business day could be determined, since today's deposits and withdrawals haven't yet been incorporated into the master file.

The simplest situation is one in which there is only one key, the only queries allowed are of type Q1

(simple query), and all retrievals and updates are batched. For this situation tapes are an adequate storage medium. All the required functions can be carried out efficiently by maintaining the master file on a tape. The records in the file are ordered by the key field. Requests for retrieval and update are batched onto a transaction tape. When it is time to process the transactions, the transactions are sorted into order by the key field and an update process similar to algorithm VERIFY2 of section 7.1 is carried out, creating a new master file. All records in the old master file are examined, changed if necessary and then written out onto a new master file. The time required for the whole process is essentially $O(n + m \log m)$ where n and m are the number of records in the master and transaction files respectively (to be more accurate this has to be multiplied by the record length). This procedure is good only when the number of transactions that have been batched is reasonably large. If $m = 1$ and $n = 10^6$ then clearly it is very wasteful to process the entire master file. In the case of tapes, however, this is the best we can do since it is usually not possible to alter a record in the middle of a tape without destroying information in an adjacent record. The file organization described above for tape files will be referred to as: *sequentially ordered*.

In this organization, records are placed sequentially onto the storage media, (i.e., they occupy consecutive locations and in the case of a tape this would mean placing records adjacent to each other). In addition, the physical sequence of records is ordered on some key, called the *primary key*. For example, if the file of figure 10.1. were stored on a tape in the sequence *A, B, C, D, E* then we would have a sequential file. This file, however, is unordered. If the primary key is Employee Number then physical storage of the file in the sequence *B, E, D, A, C* would result in an ordered sequential file. For batched retrieval and update, ordered sequential files are preferred over unordered sequential files since they are easier to process (compare VERIFY2 with VERIFY1 of 7.1).

Sequential organization is also possible in the case of a DASD such as a disk. Even though disk storage is really two dimensional (cylinder x surface) it may be mapped down into a one dimensional memory using the technique of section 2.4. If the disk has c cylinders and s surfaces, one possibility would be to view disk memory sequentially as in figure 10.2. Using the notation t_{ij} to represent the j'th track of the i'th surface, the sequence is $t_{1,1}, t_{2,1}, t_{3,1}, ..., t_{s,1}, t_{1,2}, ..., t_{s,2}$ etc.

If each employee record in the file of figure 10.1 were one track long, then a possible sequential organization would store the records *A,B,C,D,E* onto tracks $t_{3,4}, t_{4,4}, t_{5,4}, t_{6,4}$ *and* $t_{7,4}$ respectively (assuming $c \geq 4$ and $s \geq 7$). Using the interpretation of figure 10.2 the physical sequence in which the records have been stored is *A,B,C,D,E*. If the primary key is Employee Number then the logical sequence for the records is *B,E,D,A,C* as $E\#(B) < E\#(E) < ... < E\#(C)$. This would thus correspond to an unordered sequential file. In case the records are stored in the physical sequence *B,E,D,A,C,* then the file is ordered on the primary key, the logical and physical record sequences are the same, and the organization is that of a sequentially ordered file. Batched retrieval and update can be carried out essentially in the same way as for a sequentially ordered tape file by setting up input and output buffers and reading in, perhaps, one track of the master and transaction files at a time (the transaction file should be sorted on the primary key before beginning the master file processing). If updates do not change the size of records and no insertions are involved then the updated track may be written back onto the old master file. The sequential interpretation of figure 10.2 is particularly efficient for batched update

Figure 10.2 Interpreting Disk Memory as Sequential Memory

and retrieval as the tracks are to be accessed in the order: all tracks on cylinder 1 followed by all tracks on cylinder 2 etc. As a result of this the read/write heads are moved one cylinder at a time and this movement is necessitated only once for every s tracks read (s = number of surfaces). The alternative sequential interpretation (figure 10.3) would require accessing tracks in the order: all tracks on surface 1, all tracks on surface 2, etc. In this case a head movement would be necessitated for each track being read (except for tracks 1 and c).

Figure 10.3 Alternative Sequential Interpretation of Disk Memory

Since in batched update and retrieval the entire master file is scanned (typical files might contain 10^5 or more records), enough transactions must be batched for this to be cost effective. In the case of disks it is possible to extend this sequential ordered organization for use even in situations where the number of transactions batched is not enough to justify scanning the entire master file. First, take the case of a retrieval. If the records are of a fixed size then it is possible to use binary search to obtain the record with the desired key value. For a file containing n records, this would mean at most $[\log_2 n]$ accesses would have to be made to retrieve the record. For a file with 10^5 records of length 300 characters this would mean a maximum of 17 accesses. On a disk with maximum seek time 1/10 sec, latency time 1/40 sec and a track density of 5000 characters/track this would mean a retrieval time of ☐ arbitrary record from the same file stored on a tape with density 1600 bpi and a tape speed of 150 in/sec would in the worst case require 125 sec as the entire file must be read to access the last record (this does not include the time to cross interblock gaps and to restart tape motion etc. The actual time needed will be more than 125 sec).

When records are of variable size, binary search can no longer be used as given the address of the first and last records in a file one can no longer calculate the address of the middle record. The retrieval time can be considerably reduced by maintaining an index to guide the search. An index is just a collection of key value and address pairs. In the case of the file of figure 10.1 stored in the physical sequence $B,E,D,A,$ C at addresses $t_{1,2}$, $t_{2,2}$, $t_{3,2}$, $t_{4,2}$ and $t_{5,2}$ the index could contain five entries, one for each record in the file. The entries would be the pairs $(510,t_{1,2})$ $(620,t_{2,2})$ $(750,t_{3,2})$ $(800,t_{4,2})$ $(900,t_{6,2})$. An index which contains one entry for every record in the file will be referred to as a *dense index*. If a dense index is maintained for the primary key then retrieval of a record with primary key = x could be carried out by first looking into the index and finding the pair (x,addr). The desired record would then be retrieved from the location addr. The total number of accesses needed to retrieve a record would now be one plus the number of accesses needed to locate the tuple (x,addr) in the index. In section 10.2 we shall look at efficient indexing techniques that allow index searches to be carried out using at most three accesses even for reasonably large files. This means that a retrieval from the file of 10^5 records discussed earlier could

be carried out making at most four accesses rather than the seventeen accesses needed by a binary search. Since all addresses are kept in the index, it is not necessary to have fixed size records.

Sequential file organizations on a disk suffer from the same deficiencies as sequential organizatkons in internal memory. Insertion and deletion of records require moving large amounts of data in order to create space for the new record or to utilize the space used by the record being deleted. In practice these difficulties can be overcome to some extent by marking deleted records as having been deleted and not physically deleting the record. If a record is to be inserted, it can be placed into some "overflow" area rather than in its correct place in the sequentially ordered file. The entire file will have to be periodically reorganized. Reorganization will take place when a sufficient number of overflow records have accumulated and when many "deletions" have been made. While this method of handling insertions and deletions results in a degradation of system performance as far as further retrievals and updates is concerned, it is preferable to copying over large sections of the file each time an update is made. In situations where the update rate is very high or when the file size is too large to allow for periodic file reorganization (which would require processing the whole file), other organizations are called for. These will be studied in a later section.

So far, we have discussed only file organization when the number of keys is one. What happens when there is more than one key? A sequential file, clearly, can be ordered on only one key, the primary key. If an employee file is maintained on a disk using sequential organization with Employee Numbers as the primary key then how does one retrieve all records with occupation = programmer? One obvious, and inefficient, way would be to process the entire file, outputting all records that satisfy this query. Another, possibly more efficient way would be to maintain a dense index on the occupation field, search this index and retrieve the necessary records. In fact one could maintain a dense index on every key. This and other possibilities will be studied in section 10.3.

Let us summarize the ideas we have discussed. File organization is concerned with representing data records on external storage media. The choice of a representation depends on the environment in which the file is to operate, e.g., real time, batched, simple query, one key, multiple keys, etc. When there is only one key, the records may be sorted on this key and stored sequentially either on tape or disk. This results in a sequentially ordered file. This organization is good for files operating in batched retrieval and update mode when the number of transactions batched is large enough to make processing the entire file cost effective. When the number of keys is more than one or when real time responses are needed, a sequential organization in itself is not adequate. In a general situation several indexes may have to be maintained. In these cases, file organization breaks down into two more or less distinct aspects: (i) the directory (i.e. collection of indexes) and (ii) the physical organization of the records themselves. This will be referred to as the physical file. We have already discussed one possible physical organization i.e. sequential (ordered and unordered). In this general framework, processing a query or update request would proceed in two steps. First, the indexes would be interrogated to determine the parts of the physical file that are to be searched. Second, these parts of the physical file will be searched. Depending upon the kinds of indexes maintained, this second stage may involve only the accessing of records satisfying the query or may involve retrieving nonrelevant records too.

10.2 INDEX TECHNIQUES

One of the important components of a file is the directory. A directory is a collection of indexes. The directory may contain one index for every key or may contain an index for only some of the keys. Some of the indexes may be dense (i.e., contain an entry for every record) while others may be nondense (i.e., contain an entry for only some of the records). In some cases all the indexes may be integrated into one large index. Whatever the situation, the index may be thought of as a collection of pairs of the form (key value, address). If the records A,B,C,D,E of figure 10.1 are stored on disk addresses $a_1,a_2, ...,a_5$ respectively then an index for the key Employee Number would have entries $(800,a_1)$; $(510,a_2)$; $(950,a_3)$; $(750,a_4)$ and $(620,a_5)$. This index would be dense since it contains an entry for each of the records in the file. We shall assume that all the key values in an index are distinct. This may appear to be restrictive since several records may have the same key value as in the case of the Occupation key in figure 10.1. Records A,D,E all have the value 'programmer' for the Occupation key. This difficulty can be overcome easily by keeping in the address field of each distinct key value a pointer to another address where we shall maintain a list of addresses of all records having this value. If at address b_1, we stored the list of addresses of all programmer records i.e. a_1, a_4 and a_5, and at b_2 we stored the address list for all analysts, i.e. a_2 and a_3 then we could achieve the effect of a dense index for Occupation by maintaining an index with entries ('programmer', b_1) and ('analyst', b_2). Another alternative would be to change the format of entries in an index to (key value, address 1. address 2, ...address n). In both cases, different entries will have distinct key values. The second alternative would require the use of variable size nodes. The use of variable size nodes calls for complex storage management schemes (see section 4.8), and so we would normally prefer the first alternative. An index, then, consists of pairs of the type (key value, address), the key values being distinct. The functions one wishes to perform on an index are: search for a key value; insert a new pair; delete a pair from the index; modify or update an existing entry. These functions are the same as those one had to perform on a dynamic table (section 9.2). An index differs from a table essentially in its size. While a table was small enough to fit into available internal memory, an index is too large for this and has to be maintained on external storage (say, a disk). As we shall see, the techniques for maintaining an index are rather different from those used in chapter 9 to maintain a table. The reason for this is the difference in the amount of time needed to access information from a disk and that needed to access information from internal memory. Accessing a word of information from internal memory takes typically about 10^{-8} seconds while accessing the same word from a disk could take about 10^{-1} seconds.

10.2.1 Cylinder-Surface Indexing

The simplest type of index organization is the cylinder-surface index. It is useful only for the primary key index of a sequentially ordered file. It assumes that the sequential interpretation of disk memory is that of figure 10.2 and that records are stored sequentially in increasing order of the primary key. The index consists of a cylinder index and several surface indexes. If the file requires c cylinders (1 through c) for storage then the cylinder index contains c entries. There is one entry corresponding to the largest key value in each cylinder. Figure 10.4 shows a sample file together with its cylinder index (figure 10.4(b)).

Associated with each of the c cylinders is a surface index. If the disk has s usable surfaces then each surface index has s entries. The i'th entry in the surface index for cylinder j, is the value of the largest key on the j'th track of the i'th surface. The total number of surface index entries is therefore $c \square s$. Figure 10.4(c) shows the surface index for cylinder 5 of the file of figure 10.4(a). A search for a record with a particular key value X is carried out by first reading into memory the cylinder index. Since the number of cylinders in a disk is only

☐

Figure 10.4(a) World War II aircraft

☐

Figure 10.4(a). World War II aircraft

☐

Abbreviations

```
Aus.   Australia       HF   heavy fighter

AB     attack bomber   LB   light bomber

Br     bomber          MB   medium bomber

C      cylinder        NB   night bomber

FB     fighter bomber  NF   night fighter

Fr     fighter         N    no

GB     Great Britain   S    surface

Ger.   Germany         y    yes

HB   heavy bomber
```

Figure 10.4(a) File of some World War II aircraft.

(Condensed from *Airplanes* by Enzo Angelucci, McGraw-Hill Book Co., New York, 1971.)

Cylinder/surface indexing is for a disk with four surfaces and assuming two records per track. The organization is that of a sequential file ordered on the field--Aircraft.

```
cylinder   highest key value

---------------------------------

    1        Bregeut 691

    2        Heinkel III H

    3        Junkers 188 E-l

    4        Messerschmitt Sturmvogel

    5        Spitfire Mk XVI

    6        Vengeance
```

Figure 10.4(b) Cylinder index for file of figure 10.4(a)

```
surface   highest key value

-------------------------

    1        Mitsubishi G 4M1

    2        Mosquito MkV1

    3        Nakajima B5N2

    4        Spitfire MkXVI
```

Figure 10.4(c) Surface index for cylinder 5.

a few hundred the cylinder index typically occupies only one track. The cylinder index is searched to determine which cylinder possibly contains the desired record. This search can be carried out using binary search in case each entry requires a fixed number of words. If this is not feasible, the cylinder index can consist of an array of pointers to the starting point of individual key values as in figure 10.5. In either case the search can be carried out in $O(\log c)$ time. Once the cylinder index has been searched and the appropriate cylinder determined, the surface index corresponding to that cylinder is retrieved from the

disk.

Figure 10.5 Using an Array of Pointers to Permit Binary Search with Variable Length Key Values

The number of surfaces on a disk is usually very small (say 10) so that the best way to search a surface index would be to use a sequential search. Having determined which surface and cylinder is to be accessed, this track is read in and searched for the record with key *X*. In case the track contains only one record, the search is trivial. In the example file, a search for the record corresponding to the Japanese torpedo bomber Nakajima B5N2 which was used at Pearl Harbor proceeds as follows: the cylinder index is accessed and searched. It is now determined that the desired record is either in cylinder 5 or it is not in the file. The surface index to be retrieved is that for cylinder 5. A search of this index results in the information that the correct surface number is 3. Track $t_{5,3}$ is now input and searched. The desired record is found on this track. The total number of disk accesses needed for retrieval is three (one to access the cylinder index, one for the surface index and one to get the track of records). When track sizes are very large it may not be feasible to read in the whole track. In this case the disk will usually be sector addressable and so an extra level of indexing will be needed: the sector index. In this case the number of accesses needed to retrieve a record will increase to four. When the file extends over several disks, a disk index is also maintained. This is still a great improvement over the seventeen accesses needed to make a binary search of the sequential file.

This method of maintaining a file and index is referred to as ISAM (Indexed Sequential Access Method). It is probably the most popular and simplest file organization in use for single key files. When the file contains more than one key, it is not possible to use this index organization for the remaining keys (though it can still be used for the key on which the records are sorted in case of a sequential file).

10.2.2 Hashed Indexes

The principles involved in maintaining hashed indexes are essentially the same as those discussed for hash tables in section 9.3. The same hash functions and overflow handling techniques are available. since the index is to be maintained on a disk and disk access times are generally several orders of magnitude larger than internal memory access times, much consideration must be given to hash table design and the choice of an overflow handling technique. Let us reconsider these two aspects of hash system design, giving special consideration to the fact that the hash table and overflow area will be on a disk.

Overflow Handling Techniques

The overflow handling techniques discussed in section 9.3.2 are:

(i) rehashing

(ii) open addressing (a) random

(b) quadratic

(c) linear

(iii) chaining.

The expected number of bucket accesses when $s = 1$ is roughly the same for methods (i), (iia) and (iib). Since the hash table is on a disk, and these overflow techniques tend to randomize the use of the hash table, we can expect each bucket access to take almost the maximum seek time. In the case of (iic), however, overflow buckets are adjacent to the home bucket and so their retrieval will require minimum seek time. While using chaining we can minimize the tendency to randomize use of the overflow area by designating certain tracks as overflow tracks for particular buckets. In this case successive buckets on a chain may be retrieved with little or no additional seek time. To the extent that this is possible, we may regard one bucket access to be equally expensive using methods (iic) and (iii). Bucket accesses using the other methods are more expensive, and since the average number of buckets retrieved isn't any better than (iii), we shall not discuss these further.

Hash Table

Let b, s, $\boxed{}$ and n be as defined in section 9.3. For a given $\boxed{}$ and n we have $\boxed{} = n/(bs)$, and so the product bs is determined. In the case of a hash table maintained in internal memory we chose the number of slots per bucket, s, to be 1. With this choice of s, the expected number of buckets accessed when open linear addressing is used is $(2 - \boxed{})/(2 - 2\boxed{})$, and $1 + \boxed{}/2$ when chaining is used to resolve overflows. Since individual bucket accesses from a disk are expensive, we wish to explore the possibility of reducing the number of buckets accessed by increasing s. This would of necessity decrease b, the number of buckets, as bs is fixed. We shall assume that when chaining is used, the home bucket can be retrieved with one access. The hash table for chaining is similar to that for linear open addressing. Each bucket in the hash table, i.e. each home bucket, has s slots. Each such bucket also has a link field. This differs from the organization of section 9.3.2 where $s = 0$ for home buckets using chaining. Remaining buckets on individual chains have only one slot each and require additional accesses. Thus, if the key value X is in the i'th node on a chain (the home bucket being the 1st node in the chain), the number of accesses needed to retrieve X is i. In the case of linear open addressing if X is i buckets away from the home bucket, $f(X)$, then the number of accesses to retrieve X is $1 + i$.

By way of example, consider the hash function $f(x) = $ first character of X and the values B, $B1$, $B2$, $B3$, A. Using a hash table with $b = 6$ and $s = 1$, the assignment of figure 10.6(a) is obtained if overflows are handled via linear open addressing. If each of the values is searched for once then the total number of bucket retrievals is 1(for A) + 1(for

Figure 10.6 Hash Tables With s = 1 and 2.

B) + 2(for $B1$) + 3(for $B2$) + 4(for $B3$) = 11. When the same table space is divided into 3 buckets each with two slots and $f'(X) = \lceil f(X)/2 \rceil$ the assignment of key values to buckets is as in figure 10.6(b). The number of bucket retrievals needed now is 1 (for each of B and $B1$) + 2(for each of $B2$ and $B3$) + 3(for A) = 9. Thus the average number of buckets retrieved is reduced from 11/5 per search to 9/5 per search. The buckets of figure 10.6(b) are twice as big as those of figure 10.6(a). However, unless the bucket size becomes very large, the time to retrieve a bucket from disk will be dominated largely by the seek and latency time. Thus, the time to retrieve a bucket in each of the two cases discussed above would be approximately the same. Since the total average search time is made up of two components--first, the time, t_r, to read in buckets from disk and second, the time, t_p, to search each bucket in internal memory-- we should choose b and s so as to minimize $a\,(t_r + t_p)$ where a = average number of buckets retrieved. Using a sequential search within buckets, the time t_p is proportional to s. Thus the average t_p for figure 10.6(b) is one and a half times that for figure 10.6(a). When s is small and the hash table is on a disk, we have $t_r \gg t_p$ and it is sufficient to minimize a □ t_r. Let us contrast this to the situation in section 9.3 where t_r and t_p are comparable. The table of figure 10.6(a) can be searched with an average of 11/5 key comparisons and accesses. The table of figure 10.6(b) requires 1 comparison to find B, 2 for $B1$, 3 for $B2$, 4 for $B3$ and 5 for A for a total of 15 comparisons. This implies an average of 3 comparisons per search. In this case the search time has actually increased with the increase in s. But, when the table is on disk, the average times are roughly $11t_r/5$ and $9t_r/5$, and the table with $s = 2$ gives better performance. In general, we can conclude that increasing s while maintaining b □ s fixed reduces a (see figure 10.7).

Figure 10.7 □ =.5

The table of figure 10.8 shows the results of some experiments conducted on existing hashed files maintained on disk. As is evident from this table, for a fixed □, the average number of bucket accesses decreases with increasing s. Moreover, for $s \geq 10$, the average number of accesses for open linear addressing is roughly the same as for chaining. This together with our earlier observation that unless care is taken while using chaining, successive accesses will access random parts of the disk, while in the case of open linear addressing consecutive disk segments would be accessed, leads us to the conclusion that with suitable □ and s linear addressing will outperform chaining (contrast this with the case of internal tables).

While both the data of figures 10.7 and 10.8 might suggest a choice for $b = 1$ (thus maximizing the value of s), such a choice for s clearly is not best. The optimal value of s will depend very much on the values of the latency time, seek time and transmission rates of the disk drive in use. We may rewrite the retrieval

time per bucket, t_r, as $t_s + t_l + s \boxed{} t_t$ where t_s, t_l and t_t are the seek time, latency time and transmission time per slot respectively. When the seek time is very large relative to t_t the optimal s would tend to be larger than when t_s is very small compared to t_t (as in the case of a drum). Another consideration is the size of the input buffer available. Since the bucket has to be read into some part of internal memory, we must set aside enough space to accomodate a bucket load of data.

$\boxed{}$

D = division

M = middle of square

C = chaining

L = linear open addressing

$\boxed{}$ = loading factor = (number of entries)/(number of slots in hash table)

Note For the same $\boxed{}$ C uses more space than L as overflows are handled in a separate area.

Figure 10.8 Observed average number of bucket accesses for different $\boxed{}$ and s. (Condensed from Lum, Yuen and Dodd: Key to Address Transform techniques: A fundamental performance study on large existing formatted files. CACM, Vol. 14, No. 4, Apr, 1974.)

Loading Order

As in the case of internal tables, the order in which key values are entered into the hash table is important. To the extent possible, an attempt should be made to enter these values in order of nonincreasing frequency of search. When this is done, new entries should be added to the end of the overflow chain rather than at the front.

10.2.3 Tree Indexing--B-Trees

The AVL tree of section 9.2 provided a means to search, insert and delete entries from a table of size *n* using at most $O(\log n)$ time. Since these same functions are to be carried out in an index, one could conceivably use AVL trees for this application too. The AVL tree would itself reside on a disk. If nodes are retrieved from the disk, one at a time, then a search of an index with *n* entries would require at most 1.4 log *n* disk accesses (the maximum depth of an AVL tree is 1.4 log *n*). For an index with a million entries, this would mean about 23 accesses in the worst case. This is a lot worse than the cylinder sector

index scheme of section 10.2.1. In fact, we can do much better than 23 accesses by using a balanced tree based upon an *m*-way search tree rather than one based on a binary search tree (AVL trees are balanced binary search trees).

Definition: An *m-way search tree, T*, is a tree in which all nodes are of degree $\le m$. If *T* is empty, (i.e., *T* = 0) then *T* is an *m*-way search tree. When *T* is not empty it has the following properties:

(i) *T* is a node of the type

$$n, A_o, (K_1, A_1), (K_2, A_2), ..., (K_n, A_n)$$

where the A_i, $0 \le i \le n$ are pointers to the subtrees of *T* and the K_i, $1 \le i \le n$ are key values; and $1 \le n < m$.

(ii) $K_i < K_{i+1}$, $1 \le i < n$

(iii) All key values in the subtree A_i are less than the key value K_{i+1}, $0 \le i < n$

(iv) All key values in the subtree A_n are greater than K_n.

(v) The subtrees A_i, $0 \le i \le n$ are also *m*-way search trees.

As an example of a 3-way search tree consider the tree of figure 10.9 for key values 10, 15, 20, 25, 30, 35, 40, 45 and 50. One may easily verify that it satisfies all the requirements of a 3-way search


```
          schematic

node      format

-----------------------

   a      2,b,(20,c),(40,d)

   b      2,0,(10,0),(15,0)

   c      2,0,(25,0),(30,e)

   d      2,0,(45,0),(50,0)
```

```
e      1,0,(35,0)
```

Figure 10.9 Example of a 3-way search tree.

tree. In order to search for any key value X in this tree, we first "look into" the root node $T = a$ and determine the value of i for which $K_i \le X < K_{i+1}$ (for convenience we use $K_0 = [-\infty]$ and $K_{n+1} = [+\Sigma^{\infty}]$ where $[-\infty]$ is smaller than all legal key values and $[+\infty]$ is larger than all legal key values). In case $X = K_i$ then the search is complete. If $X \ne K_i$ then by the definition of an m-way search tree X must be in subtree A_i if it is in the tree. When n (the number of keys in a node) is "large," the search for the appropriate value of i above may be carried out using binary search. For "small" n a sequential search is more appropriate. In the example if $X = 35$ then a search in the root node indicates that the appropriate subtree to be searched is the one with root $A_1 = c$. A search of this root node indicates that the next node to search is e. The key value 35 is found in this node and the search terminates. If this search tree resides on a disk then the search for $X = 35$ would require accessing the nodes a, c and e for a total of 3 disk accesses. This is the maximum number of accesses needed for a search in the tree of figure 10.9. The best binary search tree for this set of key values requires 4 disk accesses in the worst case. One such binary tree is shown in figure 10.10.

Algorithm MSEARCH searches an m-way search tree T for key value X using the scheme described above. In practice, when the search tree

Figure 10.10 Best AVL-tree for data of figure 10.9

represents an index, the tuples (K_i, A_i) in each of the nodes will really be 3-tuples (K_i, A_i, B_i) where B_i is the address in the file of the record with key K_i. This address would consist of the cylinder and surface numbers of the track to be accessed to retrieve this record (for some disks this address may also include the sector number). The A_i, $0 \le i \le n$ are the addresses of root nodes of subtrees. Since these nodes are also on a disk, the A_i are cylinder and surface numbers.

```
procedure MSEARCH(T,X)

//Search the m-way search tree T residing on disk for the key

value X. Individual node format is n,A₀, (K₁,A₁), ...,(Kₙ,Aₙ),

n < m. A triple (P,i,j) is returned. j = 1 implies X is found

at node P, key Kᵢ. Else j = 0 and P is the node into which
```

```
X can be inserted//

1      P ← T; K₀ ← [-∞]; Q ← 0           //Q is the parent of P//

2      while P ≠ 0 do

3          input node P from disk

4          Let P define n, A₀,(K₁,A₁), ..., (Kₙ,Aₙ)

5          K_{n+1} ← [+∞]

6          Let i be such that Kᵢ ≤ X < K_{i+1}

7          if X = Kᵢ then [//X has been found// return (P,i,1)]

8          Q ← P; P ← Aᵢ

9      end

//X not in T; return node into which insertion can take place//

10     return (Q,i,0)

11 end MSEARCH
```

Analyzing algorithm MSEARCH is fairly straightforward. The maximum number of disk accesses made is equal to the height of the tree T. since individual disk accesses are very expensive relative to the time needed to process a node (i.e. determine the next node to access, lines 4-8) we are concerned with minimizing the number of accesses needed to carry out a search. This is equivalent to minimizing the height of the search tree. In a tree of degree m and height $h \geq 1$ the maximum number of nodes is $\sum_{0 \leq i \leq h-1}$

$m^i = (m^h - 1)/(m - 1)$. Since each node has at most $m - 1$ keys, the maximum number of entries in an m-way tree index of height h would be $m^h - 1$. For a binary tree with $h = 3$ this figure is 7. For a 200-way tree with $h = 3$ we have $m^h - 1 = 8 \times 10^6 - 1$

Clearly, the potentials of high order search trees are much greater than those of low order search trees. To achieve a performance close to that of the best m-way search trees for a given number of entries n, it is necessary that the search tree be balanced. The particular variety of balanced m-way search trees we shall consider here is known as a B-tree. In defining a B-tree it is convenient to reintroduce the concept of

failure nodes as used for optimal binary search trees in section 9.1. A failure node represents a node which can be reached during a search only if the value *X* being searched for is not in the tree. Every subtree with root = 0 is a point that is reached during the search iff *X* is not in the tree. For convenience, these empty subtrees will be replaced by hypothetical nodes called failure nodes. These nodes will be drawn square and marked with an *F*. The actual tree structure does not contain any such nodes but only the value 0 where such a node occurs. Figure 10.11 shows the 3-way search tree of figure 10.9 with failure nodes. Failure nodes are the only nodes that have no children.

Definition: A *B*-tree, *T*, of order *m* is an *m*-way search tree that is either empty or is of height $\geq$ 1 and satisfies the following properties:

(i) the root node has at least 2 children

(ii) all nodes other than the root node and failure nodes have at least $\lceil m/2 \rceil$ children

(iii) all failure nodes are at the same level.

The 3-way search tree of figure 10.11 is not a *B*-tree since it has failure nodes at levels 3 and 4. This violates requirement (iii). One possible *B*-tree of order 3 for the data of figure 10.9 is shown in figure 10.12. Notice that all nonfailure nodes are either of degree 2 or 3. In fact, for a *B*-tree of order 3, requirements (i), (ii) and the definition

Figure 10.11 Search tree of figure 10.9 with failure nodes shown.

Figure 10.12 B-tree of order 3 for data of figure 10.9.

of a *m*-way search tree together imply that all nonfailure nodes must be of degree 2 or 3. For this reason, *B*-trees of order 3 are also known as 2-3 trees.

While the total number of nonfailure nodes in a *B*-tree of a given order may be greater than the number of such nodes in the best possible search tree of that order (the 2-3 tree of figure 10.12 has 7 nodes while the 3-way search tree of figure 10.9 has only 5), we shall see later that it is easier to insert and delete nodes into a *B*-tree retaining the *B*-tree properties than it is to maintain the best possible *m*-way search tree at all times. Thus, the reasons for using *B*-trees rather than optimal *m*-way search trees for indexes, are the same as those for using AVL-trees as opposed to optimal binary search trees when maintaining dynamic internal tables.

Number of Key Values in a B-Tree

If T is a B-tree of order m in which all failure nodes are at level $l + 1$ then we know that the maximum number of index entries in T is $m^l - 1$. What is the minimum number, N, of entries in T? From the definition of a B-tree we know that if $l > 1$, the root node jas at least 2 children. Hence, there are at least two nodes at level 2. Each of these nodes must have at least $\lceil m/2 \rceil$ children. Thus there are at least $2\lceil m/2 \rceil$ nodes at level 3. At level 4 there must be at least $2\lceil m/2 \rceil^2$ nodes, and continuing this argument, we see that there are at least $2\lceil m/2 \rceil^{l-2}$ nodes at level l when $l > 1$. All of these nodes are nonfailure nodes. If the key values in the tree are $K_1, K_2, ..., K_N$ and $K_i < K_{i+1}$, $1 \le i < N$ then the number of failure nodes is $N + 1$. This is so because failures occur for $K_i < X < K_{i+1}$, $0 \le i \le N$ and $K_0 = [-\infty]$, $K_{N+1} = [+\infty]$. This results in $N + 1$ different nodes that one could reach while searching for a key value X not in T. Therefore, we have,

```
N + 1 = number of failure nodes in T

      = number of nodes at level l + 1
```

$$\ge 2\lceil m/2 \rceil^{\,l-1}$$

and so, $\qquad\qquad N \ge 2\lceil m/2 \rceil^{\,l-1} - 1, \; l \ge 1$

This in turn implies that if there are N key value in a B-tree of order m then all nonfailure nodes are at levels less than or equal to l, $l \le \log_{\lceil m/2 \rceil} \{(N + 1)/2\} + 1$. The maximum number of accesses that have to be made for a search is l. Using a B-tree of order $m = 200$, an index with $N \le 2 \times 10^6 - 2$ will have $l \le \log_{100} \{(N+ 1)/2\} + 1$. Since l is integer, we obtain $l \le 3$. For $n \le 2 \times 10^8 - 2$ we get $l \le 4$. Thus, the use of a high order B-tree results in a tree index that can be searched making a very small number of disk accesses even when the number of entries is very large.

Choice of m

B-trees of high order are desirable since they result in a reduction in the number of disk accesses needed to search an index. If the index has N entries then a B-tree of order $m = N + 1$ would have only one level. This choice of m clearly is not reasonable, since by assumption the index is too large to fit in internal memory. Consequently, the single node representing the index cannot be read into memory and processed. In arriving at a reasonable choice for m, we must keep in mind that we are really interested in minimizing the total amount of time needed to search the B-tree for a value X. This time has two components. One, the time for reading in a node from the disk and two, the time needed to search this node for X. Let us assume that each node of a B-tree of order m is of a fixed size and is large enough to accomodate n, A_0 and $m - 1$ values for (K_i, A_i, B_i), $1 \le i < m$. If the K_i are at most $\square$ characters long and the A_i and B_i each $\square$ characters long, then the size of a node is approximately $m(\square + 2\square)$ characters. The time, t_i, required to read in a node is therefore:

$$t_i = t_s + t_l + m(\square + 2\square)\, t_c$$

$$= a + bm$$

where $\quad a = t_s + t_l$ = seek time + latency time

$b = (\square + 2\square)\, t_c$ and t_c = transmission time per character

If binary search is used to carry out the search of line 6 of algorithm MSEARCH then the internal processing time per node is $c\log_2 m + d$ for some constants c and d. The total processing time per node is thus,

$$\tau = a + bm + c \log_2 m + d$$

(10.1)

For an index with N entries, the number of levels, l, is bounded by:

$\square$

The maximum search time is therefore given by (10.2).

$\square$

(10.2)

We therefore desire a value of m that minimizes (10.2). Assuming that the disk drive available has a $t_s = 1/100$ sec and $t_l = 1/40$ sec we get $a = 0.035$ sec. Since d will typically be a few microseconds, we may ignore it in comparison with a. Hence, $a + d\square\, a = 0.035$ sec. Assuming each key value is at most 6 characters long and that each A_i and B_i is 3 characters long, $\square = 6$ and $\square = 3$. If the transmission rate $t_c = 5 \times 10^{-6}$ sec/character (corresponding to a track capacity of 5000 characters), $b = (\square + 2\square)t_c = 6 \times 10^{-5}$ sec. The right hand side of equation (10.2) evaluates to

$\square$

(10.3)

Plotting this function gives us the graph of figure 10.13. From this plot it is evident that there is a wide range of values of m for which nearly optimal performance is achieved. This corresponds to the almost

flat region $m \in [50,400]$. In case the lowest value of m in this region results in a node size greater than the allowable capacity of an input buffer, the value of m will be determined by the buffer size.

Figure 10.13 Plot of $(35 + .06m)/\log_2 m$

Insertion

In addition to searching an index for a particular key value X, we also wish to insert and delete entries. We shall now focus attention on the problem of inserting a new key value X into a B-tree. The insertion is to be carried out in such a way that the tree obtained following the insertion is also a B-tree. As we shall see, the algorithm for doing this is conceptually much simpler than the corresponding insertion algorithm for AVL-trees.

In attempting to discover the operations needed to carry out the insertion, let us insert $X = 38$ into the 2-3 tree of figure 10.12. First, a search of the tree is carried out to determine where the 38 is to be inserted. The failure node that is reached during this search for $X = 38$ is the fourth from the right. Its parent node is f. The node f contains only one key value and thus has space for another. The 38 may therefore be entered here, and the tree of figure 10.14(a) is obtained. In making this insertion, the tree nodes a, c and f were accessed from the disk during the search. In addition the new node f had to be written back onto the disk. The total number of accesses is therefore four. Next, let us insert $X = 55$. A search into the B-tree of figure 10.14(a) indicates that this key value should go into the node g. There is no space in this node, since it already contains $m - 1$ key values. Symbolically inserting the new key value at its appropriate position in a node that previously had $m - 1$ key values would yield a node, P, with the following format

```
m,A_0,(K_1,A_1)  ...,(K_m,A_m)
```

```
and         K_i < K_{i+1},  1 ≤ i < m.
```

This node may be split into two nodes P and P' with the following formats

```
node P: ⌈m/2⌉ – 1, A_0,(K_1,A_1),  ...,(K_⌈m/2⌉-1, A_⌈m/2⌉-1)
```

(10.4)

```
node P': m – ⌈m/2⌉,A_⌈m/2⌉, (K_⌈m/2⌉+1,A_⌈m/2⌉+1),  ...,(K_m,A_m)
```

The remaining value $K_{[m/2]}$ and the new node P' form a tuple $(K_{[m/2]},P')$, and an attempt is made to insert this tuple into the parent of P. In the current example the node g splits into two nodes g and h. The

tuple (50,*h*) is inserted into the parent of g i.e. node *c*. Since *c* has only one key value in it, this insertion can be carried out easily. This time, 3 accesses were made to determine that $X = 55$ should be inserted

☐

Figure 10.14 Insertion into a B-tree of order 3

into *g*. Since node *g* was changed, it had to be written out onto disk. The node *h* had also to be written out onto disk. Assuming that node *c* is still in internal memory, another disk access is needed to write out the new node *c*. The total number of accesses is therefore 6. The tree obtained is that of figure 10.14(b). Insertion of $X = 37$, 5, and 18 results in the trees of figure 10.14(c), (d) and (e). Into this final tree let us insert $X = 12$. The appropriate node for insertion is node *k*. In determining this, the nodes *a*, *b* and *k* were accessed from the

☐

Figure 10.14 continued

☐

Figure 10.14 Insertion into a B-tree of Order 3 (continued)

disk. We shall assume that there is enough internal memory available to hold these three nodes in memory. Insertion into node *k* requires that it be split into two nodes *k* and *l*. The tuple (15,*l*) is to be inserted into the parent node *b*. The new nodes *k* and *l* are written out onto the disk. Insertion into *b* results in this node splitting into two nodes *b* and *m*. These two nodes are written onto the disk. The tuple (15, *m*) is to be inserted into the parent node *a* which also splits. The new nodes *a* and *n* are written out, leaving us to insert the tuple (30,*n*) into the parent of *a*. Node *a* is the root node and thus has no parent. At this point, a new root node, *p*, with subtrees *a* and *n* is created. The height of the *B*-tree increases by 1. The total number of accesses needed for this insertion is nine.

One may readily verify that the insertion transformations described above preserve the index as a *B*-tree and take care of all possibilities. The resulting algorithm for insertion assumes all nodes on the path from root to insertion point are stacked. If memory is unavailable then only the addresses need be stacked and PARENT(P) can be used.

```
procedure INSERTB(T,X)

//Key value X is inserted into the B-tree, T, of order m. T

resides on a disk.//
```

```
1    A ← 0; K ← X   //(K,A) is tuple to be inserted//

2    (P,i,j,) ← MSEARCH(T,X)     //P is node for insertion//

3    if j ≠ 0 then return    //X already in T//

4    while P ≠ 0 do

5      insert (K,A) into appropriate position in P. Let the resulting

node have the form: n,A₀, (K₁,A₁), ..., (Kₙ,Aₙ).

6       if n ≤ m - 1 then [//resulting node is not too big//

output P onto disk; return]

//P has to be split//

7         Let P and P' be defined as in equation 10.4

8         output P and P' onto the disk

9         K ← K⌈m/2⌉; A ← P'; P ← PARENT(P)

10     end

//new root is to be created//

11     create a new node R with format 1,T,(K,A)

12     T ← R; output T onto disk

13   end INSERTB
```

Analysis of INSERTB

In evaluating algorithm INSERTB, we shall use as our performance measure the number of disk accesses that are made. If the *B*-tree *T* originally has *l* levels then the search of line 2 requires *l* accesses since *X* is not in *T*. Each time a node splits (line 7), two disk accesses are made (line 8). After the final splitting, an additional access is made either from line 6 or from line 12. If the number of nodes that split is *k* then the

total number of disk accesses is $l + 2k + 1$. This figure assumes that there is enough internal memory available to hold all nodes accessed during the call of MSEARCH in line 2. Since at most one node can split at each of the l levels, k is always $\leq l$. The maximum number of accesses needed is therefore $3l + 1$. Since in most practical situations l will be at most 4, this means about thirteen accesses are needed to make an insertion in the worst case.

In the case of algorithm INSERTB, this worst case figure of $3l + 1$ accesses doesn't tell the whole story. The average number of accesses is considerably less than this. Let us obtain an estimate of the average number of accesses needed to make an insertion. If we start with an empty tree and insert N values into it then the total number of nodes split is at most $p - 2$ where p is the number of nonfailure nodes in the final B-tree with N entries. This upper bound of $p - 2$ follows from the observation that each time a node splits, at least one additional node is created. When the root node splits, two additional nodes are created. The first node created results from no splitting, and if a B-tree has more than one node then the root must have split at least once. Figure 10.15 shows that $p - 2$ is the best possible upper bound on the number of nodes split in the creation of a p node B-tree when $p > 2$ (note that there is no B-tree with $p = 2$). A B-tree of order m with p nodes has at least

$$1 + (\lceil m/2 \rceil - 1)(p - 1)$$

key values as the root has at least one key value and remaining nodes have at least $\lceil m/2 \rceil - 1$ key values. The average number of splittings, s, may now be determined

```
s = (total number of splittings)/N
```

$$\leq (p - 2) / \{1 + (\lceil m/2 \rceil - 1)(p - 1)\}$$

$$< 1/(\lceil m/2 \rceil - 1).$$

For $m = 200$ this means that the average number of splittings is less than 1/99 per key inserted. The average number of disk accesses is therefore only $l + 2s + 1 < l + 101/99 \; \square \; l + 1$.

$\square$

Figure 10.15 B-trees of order 3

Deletion

Key values may be deleted from a B-tree using an algorithm that is almost as simple as the one for insertion. To illustrate the transformations involved, let us consider the B-tree of order 3 in figure 10.16 (a). First, we shall consider the problem of deleting values from leaf nodes (i.e. nodes with no children). Deletion of the key value $X = 58$ from node f is easy, since the removal of this key value leaves the node

with 1 key value which is the minimum every nonroot node must have. Once it has been determined that $X = 58$ is in node f, only one additional access is needed to rewrite the new node f onto disk. This deletion leaves us with the tree of figure 10.16(b). Deletion of the key value $X = 65$ from node g results in the number of keys left behind falling below the minimum requirement of $\lceil m/2 \rceil - 1$. An examination of g's

Figure 10.16 Deletion from a B-tree of order 3 (Failure nodes have been dispensed with)

Figure 10.16 Deletion from a B-tree of order 3 (continued)

nearest right sibling, h, indicates that it has $\geq \lceil m/2 \rceil$ keys, and the smallest key value in h, i.e., 75 is moved up to the parent node c. The value 70 is moved down to the child node g. As a result of this transformation, the number of key values in g and h is $\geq \lceil m/2 \rceil - 1$, the number in c is unchanged and the tree retains its B-tree properties. In the search for $X = 65$, the nodes a, c and g had to be accessed. If all three nodes are retained in memory then from c the address of h the nearest right sibling of g is readily determined. Node h is then accessed. Since three nodes are altered (g, c, and h), three more accesses are to be made to rewrite these nodes onto disk. Thus, in addition to the accesses made to search for $X = 65$, four more accesses need to be made to effect the deletion of $X = 65$. In case g did not have a nearest right sibling or if its nearest right sibling had only the minimum number of key values i.e. $\lceil m/2 \rceil - 1$ we could have done essentially the same thing on g's nearest left sibling. In deleting $X = 55$ from the tree of figure 10.16(c), we see that f's nearest right sibling, g, has only $\lceil m/2 \rceil - 1$ key values and that f does not have a nearest left sibling. Consequently, a different transformation is needed. If the parent node of f is of the form $n, A_0, (K_1, A_1), ... (K_n, A_n)$ and $A_1 = g$ then we can combine the remaining keys of f with the keys of g and the key K_i into one node, g. This node will have $(\lceil m/2 \rceil - 2) + (\lceil m/2 \rceil - 1) + 1 = 2\lceil m/2 \rceil - 2 \leq m - 1$ key values which will at most fill the node. Doing this with the nodes f, g and c results in the B-tree of figure 10.16(d). Three additional accesses are needed (one to determine that g was too small, one to rewrite each of g and c). Finally, let us delete $X = 40$ from node e of figure 10.16(d). Removal of this key value leaves e with $\lceil m/2 \rceil - 2$ key values. Its nearest left sibling d does not have any extra key values. The keys of d are combined with those remaining in e and the key value 30 from the parent node to obtain the full node d with values 10 and 30. This however leaves b with o When the key value X being deleted is not in a leaf node, a simple transformation reduces this deletion to the case of deletion from a leaf node. Suppose that $K_i = X$ in the node P and P is of the form $n, A_0, (K_1, A_1), ..., (K_n, A_n)$ with $1 \leq i \leq n$. Since P is not a leaf node, $A_i \neq 0$. We can determine Y, the smallest key value in the subtree A_i. This will be in a leaf node Q. Replace K_i in P by Y and write out the d new P. This leaves us with the problem of deleting Y from Q. Note that this retains the search properties of the tree. For example the deletion of $X = 50$ from the tree of figure 10.16(a) can be accomplished by replacing the 50 in node a by the 55 from node f and then proceeding to delete the 55 from node f (see figure 10.17).

The details are provided in algorithm DELETEB on page 515. To reduce the worst case performance of the algorithm, only either the nearest left or right sibling of a node is examined.

Figure 10.17 Deleting 50 from node a of figure 10.16(a)

Analysis of DELETEB

The search for X in line 1 together with the search for the leaf Q in lines 5-10 require l accesses if l is the number of levels in T. In case P is not a leaf node at line 3 then it is modified and written out in line 11. Thus, the maximum number of accesses made in lines 1-13 is $l + 1$. Starting from a leaf, each iteration of the **while** loop of lines 15-31 moves P one level up the tree. So, at most $l - 1$ iterations of this loop may be made. Assuming that all nodes from the root T to the leaf P are in internal memory, the only additional accesses needed are for sibling nodes and for rewriting nodes that have been changed. The worst case happens when $l - 1$ iterations are made and the last one results in termination at line 25 or the corresponding point in line 30. The maximum number of accesses for this loop is therefore $(l - 1)$

```
procedure DELETEB(T,X)

//delete X from the B-tree, T, of order m. It is assumed that

T reside on a disk//

1       (P,i,j) ⟵ MSEARCH (T,X)

2       if j ≠ 1 then return    //X not in T//

3       let P be of the form: ₙ,A₀,(K₁,A₁), ..., (Kₙ,Aₙ) and Kᵢ = X

4       if A₀ ≠ 0 then   [//deletion from non leaf, find key to move up//

5                               Q ⟵ Aᵢ        //move to right subtree//

6                               loop

7                                   let Q be of the form:

n',A'₀, (K'₁,A'₁) ..., (K'ₙ,A'ₙ')
```

8   **if** A'_0 = 0 **then exit**   //Q is a leaf

node//

9 Q ☐ A'_0

10   **forever**

11 $replace$ K_i in $node$ P by K'_1 and $write$
the

$altered$ $node$ P $onto$ $disk.$

12 P ☐ Q, i ☐ $1;$

13 let $n,A_0,(K_1,A_1),$ $...,(K_n,A_n)$ be as

$defined$ by the new $P]$

//delete K_i from node P, a leaf//

14     $from$ $P:$ $n,A_0,$ $(K_1,A_1),$ $...,(K_n,A_n)$ $delete$ (K_i,A_i) and $replace$ n

by n – 1

15   **while** $n < \lceil m/2 \rceil$ – 1 **and** $P \neq T$ **do**

16   **if** P has a $nearest$ $right$ $sibling,$ $Y,$ **then**

17   [let $Y:$ $n'',A''_0,$ $(K''_1,A''_1),$ $...,(K''_{n''},A''_{n''})$ and

18   let $Z:$ $n',A'_0,$ $(K'_1,A'_1),...,(K'_{n'},A'_{n'})$ be the $parent$ of P

and Y

19   let A'_j = Y and A'_{j-1} = P

20   **if** $n'' \geq \lceil m/2 \rceil$ **then**   [//redistribute key values//

```
21                    //update P//          (K_{n+1}, A_{n+1}) □ (K'_j, A''_0);

n □ n + 1

22                    //update Z//          K'_j □ K''_1

23                    //update Y//          (n'',A''_0,(K''_1,A''_1),...)□

(n'' - 1,A''_1,(K''_2,A''_2)...)

24                    output nodes P,Z and Y onto

disk

25                    return]

//combine P, K'_j and Y//

26            r □ 2⌈m/2⌉ - 2

27        output r,A_0,(K_1,A_1) ... (K_n,A_n),(K'_j,A''_0),(K''_1,A''_1),

... (K''_{n''},A''_{n''}) as new node P

28    //update//          (n,A_0, ...,) □ (n' - 1,A.'_0,...,(K'_{j-1},A'_{j-1}),

(K'_{j+1},A'_{j+1})...)

29            P □ Z]

else

30            [//P must have a left sibling//

this is symmetric to lines 17-29

and is left as an exercise]
```

```
31      end

32      if n ≠ 0 then [output P: (n,A₀, ...,(Kₙ,Aₙ)]

33              else [T □ A₀]

34   end DELETEB
```

for siblings, plus (l - 2) updates at line 27 and the corresponding point in line 30, plus 3 updates at line 24 or in line 30 for termination. This figure is 2l. The maximum number of accesses required for a deletion is therefore $3l + 1$.

The deletion time can be reduced at the expense of disk space and a slight increase in node size by including a delete bit, F_i, for each key value K_i in a node. Then we can set $F_i = 1$ if K_i has not been deleted and $F_i = 0$ if it has. No physical deletion takes place. In this case a delete requires a maximum of $l + 1$ accesses (l to locate the node containing X and 1 to write out this node with the appropriate delete bit set to 0). With this strategy, the number of nodes in the tree never decreases. However, the space used by deleted entries can be reused during further insertions (see exercises). As a result, this strategy would have little effect on search and insert times (the number of levels increases very slowly with increasing n when m is large). Insert times may even decrease slightly due to the ability to reuse deleted entry space. Such reuses would not require us to split any nodes.

For a variation of B-trees see exercises 10-14 through 10-20.

Variable Size Key Values

With a node format of the form $n, A_0, (K_1, A_1,), ..., (K_n, A_n)$, the first problem created by the use of variable size key values, K_i, is that a binary search can no longer be carried out since given the location of the first tuple (K_1, A_1) and n, we cannot easily determine Kn or even the location of $K_{(1+n)/2}$. When the range of key value size is small, it is best to allocate enough space for the largest size key value. When the range in sizes is large, storage may be wasted and another node format may become better, i.e., the format $n, A_0, □_1, □_2, ..., □_n, (K_1, A_1), ..., (K_n, A_n)$ where $□_i$ is the address of K_i in internal memory i.e. $K_i = $ memory ($□_i$). In this case a binary search of the node can still be made. The use of variable size nodes is not recommended since this would require a more complex storage management system. More importantly, the use of variable size nodes would result in degraded performance during insertion. For example, insertion of $X = 85$ intothe 2-3 tree of figure 10.16(e) would require us to request a larger node, j, to accomodate the new value being added to the node h. As a result, node c would also be changed since it contains a pointer to h. This pointer must be changed to j. Consequently, nodes of a fixed size should be used. The size should be such as to allow for at least m - 1 key values of the largest size. During

insertions, however, we can relax the requirement that each node have $\leq m$ - 1 key values. Instead a node will be allowed to hold as many values as can fit into it and will contain at least $\lceil m/2 \rceil$ - 1 values. The resulting performance will be at least as good as that of a B-tree of order m. Another possibility is to use some kind of key sampling scheme to reduce the key value size so as not to exceed some predetermined size d. Some possibilities are prefix and suffix truncation, removing vowels, etc. Whatever the scheme used, some provision will have to be made for handling synonyms (i.e., distinct key values that have the same sampled value). Key sampling is discussed further in section 10.2.4.

10.2.4 Trie Indexing

An index structure that is particularly useful when key values are of varying size is the trie. A *trie* is a tree of degree $m \geq 2$ in which the branching at any level is determined not by the entire key value but by only a portion of it. As an example consider the trie of figure 10.18. The trie contains two types of nodes. The first type we shall call a *branch node* and the second an *information node*. In the trie of figure 10.18 each branch node contains 27 link fields. All characters in the key values are assumed to be one of the 26 letters of the alphabet. A blank is used to terminate a key value. At level 1 all key values are partitioned into 27 disjoint classes depending on their first character. Thus, LINK(T,i) points to a subtrie containing all key values beginning with the i-th letter (T is the root of the trie). On the j th level the branching is determined by the j th character. When a subtrie contains only one key value, it is replaced by a node of type information. This node contains the key value, together with other relevant information such as the address of the record with this key value, etc. In the figure, branch nodes are represented by rectangles while ovals are used for information nodes.

Searching a trie for a key value X requires breaking up X into its constituent characters and following the branching patterns determined by these characters. The algorithm TRIE assumes that $P = 0$ is not a branch node and that KEY(P) is the key value represented in P if P is an information node.

Analysis of Algorithm TRIE

The search algorithm for tries is very straightforward and one may readily verify that the worst case search time is $O(l)$ where l is the number of levels in the trie (including both branch and information

procedure *TRIE (T,X)*

//Search a trie *T* for key value *X*. It is asssumed that branching

on the *i*-th level is determinded by the *i*-th character of the key

value.//

K ☐ *X* ||' '   //concatenate a blank to the end of *X*//

```
i □ 1; P □ T

while P is a branch node do

  C □ i-th character of K

  P □ LINK (P,C); □ i + 1

end

if P = 0 or KEY (P) ≠ X then return (0)        //X not in trie//

return (P)

end TRIE
```

nodes). In the case of an index, all these nodes will reside on disk and so at most l accesses will have to be made to effect the search. Given a set of key values to be represented in an index, the number of levels in the trie will clearly depend on the strategy or key sampling technique used to determine the branching at each level. This can be defined by a sampling function SAMPLE(X,i) which appropriately samples X for branching at the i-th level. In the example trie of figure 10.18 and in the search algorithm TRIE this function was

(a) SAMPLE(X,i) = i'th character of X.

Some other possible choices for this function are ($X = x_1 x_2 ... x_n$)

(b) SAMPLE(X,i) = xn - i+1

(c) SAMPLE(X,i) = xr(Xi) for r(X,i) a randomization function

(d) SAMPLE(X,i) = □

For each of these functions one may easily construct key value sets for which that particular function is best, i.e. it results in a trie with the fewest number of levels. The trie of figure 10.18 has 5 levels. Using the function (b) on the same key values yields the trie of figure 10.20, which has only 3 levels. An optimal sampling function for this data set will yield a trie that has only 2 levels (figure 10.21). Choosing the optimal sampling function for any particular set of values is very difficult. In a dynamic situation, with insertion and deletion, we wish to optimize average performance. In the absence of any further information on key values, probably the best choice would be (c). Even though all our examples of

sampling have involved single character sampling we need not restrict ourselves to this. The key value may be interpreted as consisting of digits using any radix we desire. Using a radix of 27^2

Figure 10.18 Trie created using characters of key value left to right, one at a time

Figure 10.19 Trie showing need for a terminal character (in this case a blank).

would result in 2 character sampling. Other radixes would give different samplings. The maximum number of levels in a trie can be kept low by adopting a different strategy for information nodes. These nodes can be designed to hold more than one key value. If the maximum number of levels allowed is *l* then all key values that are synonyms up to level *l* - 1 are entered into the same information node. If the sampling function is chosen correctly, there will be only a few synonyms in each information node. The information node will therefore be small and can be processed in internal memory. Figure 10.22 shows the use of this strategy on the trie of figure 10.18 with *l* = 3. In further discussion we shall, for simplicity, assume that the sampling function in use is (a) and that no restriction is placed on the number of levels in the trie.

Insertion

Insertion into a trie is straightforward. We shall indicate the procedure by means of two examples and leave the formal writing of the algorithm as an exercise. Let us consider the trie of figure 10.18 and insert into it the two entries: bobwhite and bluejay. First, we have *X* = bobwhite and we attempt to search for 'bobwhite' in *T*. This leads us to node Σ, *where we discover that LINK(Σ, 'O') = 0*. Hence *X* is not in *T* and may be inserted here (see figure 10.23). Next, *X* = bluejay and a search of *T* leads us to the information node P. A comparison indicates

Figure 10.20 Trie constructed for data of figure 10.18 sampling one character at a time, right to left

Figure 10.21 An optimal trie for data of figure 10.18 sampling on first level done by using 4th character of key values.

Figure 10.22 Trie obtained for data of figure 10.18 when number of levels is limited to 3. Key has been sampled left to right one character at a time.

☐

Figure 10.23 Section of trie of figure 10.18 showing changes resulting from inserting bobwhite and bluejay

that KEY(P) $\neq$ X. Both KEY(P) and X will form a subtrie of Σ. The two values KEY(P) and X are sampled until the sampling results in two different values. The happens when the 5th letter of KEY(P) and X are compared. The resulting trie after insertion is in figure 10.23.

Deletion

Once again, we shall not present the deletion algorithm formally but we will look at two examples to illustrate some of the ideas involved in deleting entries from a trie. From the trie of figure 10.23 let us first delete 'bobwhite.' To do this we just set LINK(Σ,'O') = 0. No other changes need be made. Next, let us delete 'bluejay.' This deletion leaves us with only one key value in the subtrie, ☐$_3$. This means that the node ☐$_3$ may be deleted and P moved up one level. The same can be done for nodes ☐$_1$ and ☐$_2$. Finally, the node Σ is reached. The subtrie with root Σ has more than one key value. Therefore P cannot be moved up any more levels and we set LINK(Σ,'L') = P. In order to facilitate deletions from tries, it is useful to add a COUNT field in each branch node. This count field will at all times give us the number of information nodes in the subtree for which it is the root. See exercise 26 for more on tries.

10.3 FILE ORGANIZATIONS

10.3.1 Sequential Organizations

The problems associated with sequential organizations were discussed earlier. The most popular sequential organization scheme is ISAM, in which a cylinder-surface index is maintained for the primary key. In order to efficiently retrieve records based on other keys, it is necessary to maintain additional indexes on the remaining keys (i.e. secondary keys). The structure of these indexes may correspond to any of the alternative index techniques discussed in the previous section. The use of secondary key indexes will be discussed in greater detail in connection with inverted files.

10.3.2 Random Organization

In this organization, records are stored at random locations on disk. This randomization could be achieved by any one of several techniques. Some of these techniques are direct addressing, directory lookup and hashing.

Direct Addressing

In direct addressing with equal size records, the available disk space is divided out into nodes large enough to hold a record. The numeric value of the primary key is used to determine the node into which a particular record is to be stored. No index on this key is needed. With primary key = Employee #, the record for Employee # = 259 will be stored in node 259. With this organization, searching and deleting a record given its primary key value requires only one disk access. Updating a record requires two (one to read and another to write back the modified record). When variable size records are being used an index can be set up with pointers to actual records on disk (see figure 10.24). The number of accesses needed using this scheme is one more than for the case when memory was divided into fixed size nodes. The storage management scheme becomes more complex (section 10.4). The space efficiency of direct accessing depends on the identifier density n/T (n = number of distinct primary key values in the file, T = total number of possible primary key values). In the case of internal tables, this density is usually very low and direct addressing was very space inefficient.

Figure 10.24 Direct Access Index for Data of Figure 10.1.

Directory LookUp

This is very similar to the scheme of figure 10.24 for the case of direct addressing with variable size records. Now, however, the index is not of direct access type but is a dense index maintained using a structure suitable for index operations. Retrieving a record involves searching the index for the record address and then accessing the record itself. The storage management scheme will depend on whether fixed size or variable size nodes are being used. Except when the identifier density is almost 1, this scheme makes a more efficient utilization of space than does direct addressing. However it requires more accesses for retrieval and update, since index searching will generally require more than 1 access. In both direct addressing and directory lookup, some provision must be made for collisions (i.e. when two or more records have the same primary key value). In many applications the possibility of collisions is ruled out since the primary key value uniquely identifies a record. When this is not the case, some of the schemes to be discussed in section 10.3.3 and section 10.3.4 may be used.

Hashing

The principles of hashed file organization are essentially the same as those for a hashed index. The available file space is divided into buckets and slots. Some space may have to be set aside for an overflow area in case chaining is being used to handle overflows. When variable size records are present, the number of slots per bucket will be only a rough indicator of the number of records a bucket can hold. The actual number will vary dynamically with the size of records in a particular bucket.

Random organization on the primary key using any of the above three techniques overcomes the difficulties of sequential organizations. Insertions and deletions become relatively trivial operations. At the same time, random organizations lose the advantages of a sequential ordered organization. Batch

processing of queries becomes inefficient as the records are not maintained in order of the primary key. In addition, handling of range queries becomes exceedingly inefficient except in the case of directory lookup with a suitable index structure. For example, consider these two queries: (i) retrieve the records of all employees with Employee Number > 800 and (ii) retrieve all records with $301 \leq$ employee number $\leq$ 800. To do (i) we will need to know the maximum employee number in the file or else examine every node. If the maximum employee number is known then (i) becomes similar to (ii), so lets look at (ii). With direct addressing and hashing, we would have to search for records with employee number = 301, 302, ... 800, a total of 500 independent searches. The number of records satisfying the query may be much smaller (even 0). In the case of ISAM we could in three accesses locate the record (if one exists) having the smallest employee number satisfying the query and retrieve the remaining records one at a time. The same is possible when a search tree type directory (index) is maintained.

10.3.3 Linked Organization

Linked organizations differ from sequential organizations essentially in that the logical sequence of records is generally different from the physical sequence. In a sequential organization, if the i'th record of the file is at location l_i then the $i + 1$'st record is in the next physical position $l_i + c$ where c may be the length of the i'th record or some constant that determines the inter-record spacing. In a linked organization the next logical record is obtained by following a link value from the present record. Linking records together in order of increasing primary key value facilitates easy insertion and deletion once the place at which the insertion or deletion to be made is known. Searching for a record with a given primary key value is difficult when no index is available, since the only search possible is a sequential search. To facilitate searching on the primary key as well as on secondary keys it is customary to maintain several indexes, one for each key. An employee number index, for instance, may contain entries corresponding to ranges of employee numbers. One possibility for the example of figure 10.1 would be to have an entry for each of the ranges 501-700, 701-900 and 901-1100. All records having E # in the same range will be linked together as in figure 10.25. Using an index in this way reduces the length of the lists and thus the search time. This idea is very easily generalized to allow for easy secondary key retrieval. We just set up indexes for each key and allow records to be in more than one list. This leads to the *multilist* structure for file representation. Figure 10.26 shows the indexes and lists corresponding to a multilist representation of the data of figure 10.1. It is assumed that the only fields designated as keys are: *E#*, Occupation, Sex and Salary. Each record in the file, in addition to all the relevant information fields, has 1 link field for each key field.

The logical order of records in any particular list may or may not

Figure 10.25

Figure 10.26 Multilist representation for figure 10.1

be important depending upon the application. In the example file, lists corresponding to *E#*, Occupation and Sex have been set up in order of increasing *E#*. The salary lists have been set up in order of increasing salary within each range (record *A* precedes *D* and *C* even though *E#*(*C*) and *E#*(*D*) are less than *E#*(*A*)).

Notice that in addition to key values and pointers to lists, each index entry also contains the length of the corresponding list. This information is useful when retrieval on boolean queries is required. In order to meet a query of the type, retrieve all records with Sex = female and Occupation = analyst, we search the Sex and Occupation indexes for female and analyst respectively. This gives us the pointers *B* and *B*. The length of the list of analysts is less than that of the list of females, so the analyst list starting at *B* is searched. The records in this list are retrieved and the Sex key examined to determine if the record truly satisfies the query. Retaining list lengths enables us to reduce search time by allowing us to search the smaller list. Multilist structures provide a seemingly satisfactory solution for simple and range queries. When boolean queries are involved, the search time may bear no relation to the number of records satisfying the query. The query *K1 = XX and K2 = XY* may lead to a *K1* list of length n and a *K2* list of length m. Then, $\min\{n,m\}$ records will be retrieved and tested against the query. It is quite possible that none or only a very small number of these $\min\{n,m\}$ records have both *K1 = XX* and *K2 = XY*. This situation can be remedied to some extent by the use of *compound keys*. A compound key is obtained by combining two or more keys together. We could combine the Sex and Occupation keys to get a new key Sex-Occupation. The values for this key would be: female analyst, female programmer, male analyst and male programmer. With this compound key replacing the two keys Sex and Occupation, we can satisfy queries of the type, all male programmers or all programmers, by retrieving only as many records as actually satisfy the query. The index size, however, grows rapidly with key compounding. If we have ten keys $K_1, ...,K_{10}$, the index for K_i having n_i entries, then the index for the compound key K_1-K_2- ... -K_{10} will have $\prod^{10}_{i=1} n_i$ entries while the original indexes had a total of $\sum^{10}_{i=1} n_i$ entries. Also, handling simple queries becomes more complex if the individual key indexes are no longer retained (see the exercises).

Inserting a new record into a multilist structure is easy so long as the individual lists do not have to be maintained in some order. In this case the record may be inserted at the front of the appropriate lists (see exercise 27). Deletion of a record is difficult since there are no back pointers. Deletion may be simplified at the expense of doubling the number of link fields and maintaining each list as a doubly linked list (see exercises 28 and 30). When space is at a premium, this expense may not be accpetable. An alternative is the coral ring structure described below.

Coral Rings

The coral ring structure is an adaptation of the doubly linked multilist structure discussed above. Each list is structured as a circular list with a headnode. The headnode for the list for key value $K_i = X$ will have an information field with value *X*. The field for key K_i is replaced by a link field. Thus, associated with each

record, Y, and key, K_i, in a coral ring there are two link fields: ALINK(Y,i) and BLINK(Y,i). The ALINK field is used to link together all records with the same value for key K_i. The ALINKs form a circular list with a headnode whose information field retains the value of K_i for the records in this ring. The BLINK field for some records is a back pointer and for others it is a pointer to the head node. To distinguish between these two cases another field FLAG(Y,i) is used. FLAG(Y,i) = 1 if BLINK(Y,i) is a back pointer and FLAG(Y,i) = 0 otherwise. In practice the FLAG and BLINK fields may be combined with BLINK(Y, i) > 0 when it is a back pointer and < 0 when it is a pointer to the head node. When the BLINK field of a record BLINK(Y,i) is used as a back pointer, it points to the nearest record, Z, preceding it in its circular list for K_i having BLINK(Z,i) also a back pointer. In any given circular list, all records with back pointers form another circular list in the reverse direction (see figure 10.27). The presence of these back pointers makes it possible to carry out a deletion without having to start at the front of each list containing the record being deleted in order to determine the preceding records in these lists (see exercise 32). Since these BLINK fields will usually be smaller than the original key fields they replace, an overall saving in space will ensue. This is, however, obtained at the expense of increased retrieval time (exercise 31). Indexes are maintained as for multilists. Index entries now point to head nodes. As in the case of multilists, an individual node may be a member of several rings on different keys.

Figure 10.27 Coral ring for analysts in a hypothetical file.

10.3.4 Inverted Files

Conceptually, inverted files are similar to multilists. The difference is that while in multilists records with the same key value are linked together with link information being kept in individual records, in the case of inverted files this link information is kept in the index itself. Figure 10.28 shows the indexes for the file of figure 10.1. A slightly different strategy has been used in the $E\#$ and salary indexes than was used in figure 10.26, though the same strategy could have been used here too. To simplify further discussion, we shall assume that the index for every key is dense and contains a value entry for each distinct value in the file. Since the index entries are variable length (the number of records with the same key value is variable), index maintenance becomes more complex than for multilists. However, several benefits accrue from this scheme. Boolean queries require only one access per record satisfying the query (plus some accesses to process the indexes). Queries of the type $K1 = XX$ or $K2 = XY$ can be processed by first accessing the indexes and obtaining the address lists for all records with $K1 = XX$ and $K2 = XY$. These two lists are then merged to obtain a list of all records satisfying the query. $K1 = XX$ *and* $K2 = XY$ can be handled similarly by intersecting the two lists. $K1 = $ .not. XX can be handled by maintaining a universal list, U, with the addresses of all records. Then, $K1 = $ .not. XX is just the difference between U and the list for $K1 = XX$. Any complex boolean query may be handled in this way. The retrieval works in two steps. In the first step, the indexes are processed to obtain a list of records satisfying the query and in the second, these records are retrieved using this list. The number of disk accesses needed is equal to the number of records being retrieved plus the number to process the indexes. Exercise 34 explores the time savings that can result using this structure rather than multilists.

Inverted files represent one extreme of file organization in which only the index structures are important. The records themselves may be stored in any way (sequentially ordered by primary key, random,

Figure 10.28 Indexes for fully inverted file

linked ordered by primary key etc). Inverted files may also result in space saving compared with other file structures when record retrieval does not require retrieval of key fields. In this case the key fields may be deleted from the records. In the case of multilist structures, this deletion of key fields is possible only with significant loss in system retrieval performance (why?). Insertion and deletion of records requires only the ability to insert and delete within indexes.

10.3.5 Cellular Partitions

In order to reduce file search times, the storage media may be divided into cells. A cell may be an entire disk pack or it may simply be a cylinder. Lists are localised to lie within a cell. Thus if we had a multilist organization in which the list for KEY1 = PROG included records on several different cylinders then we could break this list into several smaller lists where each PROG list included only those records in the same cylinder. The index entry for PROG will now contain several entries of the type (addr, length), where addr is a pointer to the start of a list of records with KEY1 = PROG and length is the number of records on this list. By doing this, all records in the same cell (i.e. cylinder) may be accessed without moving the read/write heads. In case a cell is a disk pack then using cellular partitions it is possible to search different cells in parallel (provided the system hardware permits simultaneous reading/writing from several disk drives).

It should be noted that in any real situation a judicious combination of the techniques of this section would be called for. I.e., the file may be inverted on certain keys, ringed on others, and a simple multilist on yet other keys.

10.4 STORAGE MANAGEMENT

The functions of a storage management system for an external storage device (such as a disk) are the same as those for a similar system for internal memory. The storage management system should be able to allocate and free a contiguous block of memory of a given size. In chapter 4 we studied several memory management schemes. For fixed size nodes we used a linked stack together with the routines RET and GETNODE of section 4.3. For variable size nodes the boundary tag method of section 4.8 was good. We also studied a general purpose management scheme involving garbage collection and compaction in section 4.10. When the storage to be managed is on an external device, the management scheme used should be such as to minimize the time needed to allocate and free storage. This time will to

a large extent depend on the number of external storage accesses needed to effect allocation or freeing. In this section we shall review and reevaluate the storage management schemes of chapter 4 with emphasis on the number of device accesses required.

Fixed Size Nodes

All nodes that are free may be linked together into an available space list as in section 4.3. Assuming that the value of AV is always in memory the allocate and free algorithms take the form:

procedure *GETNODE(I)*

if *AV* = 0 **then** [**print** "no more nodes;" **stop**]

I ☐ *AV; X* ☐ *GET(AV); AV* ☐ *LINK(X)*

end *GETNODE*

procedure *RET(I)*

LINK(I) ☐ *AV; WRITE(I); AV* ☐ *I*

end *RET*

GET and WRITE are primitives that read from and write onto an external storage device. Each algorithm requires one access to carry out its function. In case the total number of free nodes at all times is small, this access can be eliminated altogether by maintaining the available space list in the form of a stack, STK, which contains the addresses of all free nodes. This stack would not be maintained in place as in the case of section 4.3, where the free space itself made up the stack. Instead, the stack could reside permanently in memory and no accesses would be needed to free or allocate nodes.

Variable Size Nodes

In the boundary tag method, the use of boundary tags made it possible to free a block while coalescing adjacent blocks in a constant amount of time. In order to do this it is essential that the free space be maintained in place as we need to test TAG(p - 1) and TAG($p + n$)(cf. section 4.8). Such a scheme would therefore require several accesses (exercise 35) in order to free and allocate nodes. These accesses can again be eliminated by maintaining in memory a list of all blocks of storage that are free. Allocation can be made using first fit, and nodes can be freed in time $O(n)$ if n is the number of blocks in free space (exercise 36). Since the cost of a disk access is several orders of magnitude more than the cost of internal processing, this scheme will be quicker than the boundary tag method even when free space consists of several thousand blocks. When free space contains many nodes it may not be possible to keep the list of

free nodes in memory at all times. In this case the list will have to be read in from disk whenever needed and written back onto disk when modified. Still, the number of accesses will be fewer than when an in place free list is maintained (as is required by the boundary tag method).

Garbage Collection and Compaction

The principles for this technique stay essentially the same. The process was seen to be rather slow for the case of internal memory management. It is even slower for external storage because of the increased access times. In file systems that have a heavy activity rate (i.e. frequent real time update etc.), it is not possible to allocate a continuous chunk of time large enough to permit garbage collection and compaction. It is necessary, therefore, to devise these algorithms so that they can work with frequent interruptions, leaving the file in a usable state whenever interrupted.

REFERENCES

For additional material on file structures and data base management systems see:

File structures for on line systems by D. Lefkovitz, Hayden Book Co., New Jersey, 1969.

Data Management for on line systems by D. Lefkovitz, Hayden Book Co., New Jersey, 1974.

Computer data base organization by J. Martin, Prentice-Hall, Englewood Cliffs, 1975.

An introduction to data base management systems by C. Date, Addison-Wesley, Reading, Massachusetts, 1975.

Additional material on indexes can be found in:

The Art of Computer Programming: Sorting and Searching by D. Knuth, Addison-Wesley, Reading, Massachusetts, 1973.

"Binary search trees and file organization" by J. Nievergelt, *ACM Computing Surveys*, vol. 6, no. 3, September 1974, pp. 195-207.

EXERCISES

1. A file of employee records is being created. Each record has the following format:

```
E#   NAME   Occupation   Location
```

All four fields have been designated as keys. The file is to consist of the following 5 records:

```
A   10   JAMES    PROG   MPLS

B   27   SHERRY   ANAL   NY

C   39   JEAN     PROG   NY

D   50   RODNEY   KEYP   MPLS

E   75   SUSAN    ANAL   MPLS
```

Draw the file and index representations for the following organizations (assume that an entry in an index is a tuple (value, pointer 1, pointer 2, ...,pointer k) and these tuples are kept sequentially).

a) Multilist File

b) Fully Inverted File

c) Ring File

d) Sequential ordered by name, inverted on $E\#$ and location, ringed on occupation.

2. Write an algorithm to process a tape file in the batched mode. Assume the master file is ordered by increasing primary key value and that all such values are distinct. The transaction file contains transactions labeled: update, delete and insert. Each such transaction also contains the primary key value of the record to be updated, deleted or inserted. A new updated master file is to be created. What is the complexity of your algorithm?

3. Show that all B-trees of order 2 are full binary trees.

4. (a) Into the following 2-3 tree insert the key 62 using algorithm INSERTB.

☐

Assuming that the tree is kept on a disk and one node may be fetched at a time, how many disk accesses are needed to make this insertion? State any assumptions you make.

(b) From the following B-tree of order 3 delete the key 30 (use algorithm DELETEB). Under the same assumptions as kn (a), how many disk accesses are needed?

5. Complete line 30 of algorithm DELETEB.

6. Write insertion and deletion algorithms for *B*-trees assuming that with each key value is associated an additional field *F* such that *F* = 1 iff the corresponding key value has not been deleted. Deletions should be accomplished by simply setting the corresponding *F* = 0 and insertions should make use of deleted space whenever possible without restructuring the tree.

7. Write algorithms to search and delete keys from a *B*-tree by position. I.e., SEARCH(*k*) finds the *k*-th smallest key and DELETE(*k*) deletes the *k*-th smallest key in the tree. (*Hint:* In order to do this efficiently additional information must be kept in each node. With each pair (K_i, A_i) keep (number of key values in the subtree A_j + 1).) What is the worst case computing times of your algorithms?

8. Program the AVL tree insertion algorithm of section 9.2 and also the *B*-tree insertion algorithm with *m* = 3. Evaluate the relative performance of these two methods for maintaining dynamic internal tables when only searches and insertions are to be made. This evaluation is to be done by obtaining real computing times for various input sequences as well as by comparing storage requirements.

9. Modify algorithm INSERTB so that in case *n* = *m* in line 6 then we first check to see if either the nearest left sibling or the nearest right sibling of *P* has fewer than *m* - 1 key values. If so, then no additional nodes are created. Instead, a rotation is performed moving either the smallest or largest key in *P* to its parent. The corresponding key in the parent together with a subtree is moved to the sibling of *P* which has space for another key value.

10. [Bayer and McCreight] The idea of exercise 9 can be extended to obtain improved *B*-tree performance. In case the nearest sibling, *P*', of *P* already has *m* - 1 key values then we can split both *P* and *P*' to obtain three nodes *P*, *P*' and *P*\" with each node containing $\lfloor (2m - 2)/3 \rfloor$, $\lfloor (2m - 1)/3 \rfloor$ and $\lfloor 2m/3 \rfloor$ key values. Figure 10.29 below describes this splitting procedure when *P*' is *P*'s nearest right sibling.

Figure 10.29 Splitting P and nearest right sibling P'

Rewrite algorithm INSERTB so that node splittings occur only as described above.

11. A *B**-tree, *T*, of order *m* is a search tree that is either empty or is of height ≥ 1. When *T* is not empty then the extended tree *T* (i.e., *T* with failure nodes added) satisfies the following conditions:

(i) The root node has at least 2 and at most $2\lfloor (2m - 2)/3 \rfloor + 1$ children.

(ii) The remaining nonfailure nodes have at most m and at least $\lceil (2m - 1)/3 \rceil$ children each.

(iii) All failure nodes are on the same level.

For a B^*-tree of order m and containing N key values show that if $x = \lceil (2m - 1)/3 \rceil$ then

(a) The depth, d, of T satisfies:

$d \leq 1 + \log_x\{(N+1)/2\}$

(b) the number of nodes p in T satisfies:

$p \leq 1 + (N - 1)/(x-1)$

What is the average number of splittings if T is built up starting from an empty B^*-tree?

12. Using the splitting technique of exercise 10 write an algorithm to insert a new key value X into a B^*-tree, T, of order m. How many disk accesses are made in the worst case and on the average? Assume that T was initially of depth l and that T is maintained on a disk. Each access retrieves or writes one node.

13. Write an algorithm to delete the identifier X from the B^*-tree, T, of order m. What is the maximum number of accesses needed to delete X from a B^*-tree of depth l. Make the same assumptions as in exercise 12.

14. The basic idea of a B-tree may be modified differently to obtain a B'-tree. A B'-tree of order m differs from a B-tree of order m only in that in a B'-tree identifiers may be placed only in leaf nodes. If P is a nonleaf node in a B'-tree and is of degree j then the node format for P is: $j, L(1), L(2), ... L(j - 1)$ where $L(i), 1 \leq i < j$ is the value of the largest key in the i'th subtree of P. Figure 10.30 shows two B'-trees of order 3. Notice that in a B'-tree the key values in the leaf nodes will be increasing left to right. Only the leaf nodes contain such information as the address of records having that key value. If there are n key values in the tree then there are n leaf nodes. Write an algorithm to search for X in a B'-tree T of order 3. Show that the time for this is $O(\log n)$.

Figure 10.30 Two B'-trees of order 3

15. For a B'-tree of order 3 write an algorithm to insert X. Recall that all non leaf nodes in a B'-tree of order 3 are of degree 2 or 3. Show that the time for this is $O(\log n)$.

16. Write an algorithm to delete X from a B'-tree, T, of order 3. Since all key values are in leaf nodes, this

always corresponds to a deletion from a leaf. Show that if T has n leaf nodes then this requires only $O(\log n)$ time.

17. Let T and T' be two B'-trees of order 3. Let $T\backslash''$ be a B'-tree of order 3 containing all key values in T and $T\backslash''$. Show how to construct $T\backslash''$ from T and T' in $O(\log n)$ time.

18. Write computer programs to insert key values into AVL trees, B-trees of order 3, B^*-trees of order 3 and B'-trees of order 3. Evaluate the relative performance of these four representations of internal tables.

19. Do exercise 18 when the required operations are search for X, insert X and delete X.

20. Obtain SEARCH, INSERT and DELETE algorithms for B'-trees of order m. If the tree resides on disk, how many disk accesses are needed in the worst case for each of the three operations. Assume the tree has n leaf nodes.

21. Draw the trie obtained for the following data: Sample the keys left to right one character at a time. Using single character sampling obtain an optimal trie for the above data (an optimal trie is one with the fewest number of levels).

AMIOT, AVENGER, AVRO, HEINKEL, HELLDIVER, MACCHI, MARAUDER, MUSTANG, SPITFIRE, SYKHOI

22. Write an algorithm to insert a key value X into a trie in which the keys are sampled left to right, one character at a time.

23. Do exercise 22 with the added assumption that the trie is to have no more than 6 levels. Synonyms are to be packed into the same information node.

24. Write an algorithm to delete X from a trie T under the assumptions of exercise 22. Assume that each branch node has a count field equal to the number of information nodes in the subtrie for which it is the root.

25. Do exercise 24 for the trie of exercise 23.

26. In the trie of Figure 10.23 the nodes $\square_1$ and $\square_2$ each have only one child. Branch nodes with only one child may be eliminated from tries by maintaining a SKIP field with each node. The value of this field equals the number of characters to be skipped before obtaining the next character to be sampled. Thus, we can have SKIP ($\square_3$) = 2 and delete the nodes $\square_1$ and $\square_2$. Write algorithms to search, insert and delete from tries in which each branch node has a skip field.

In exercises 27-34 records have n keys. The i'th key value for record Z is KEY(Z,i). The link field for the

i'th key is LINK(Z,i). The number of accesses required by an algorithm should be given as a function of list lengths.

27. Write an algorithm to insert a record, Z, into a multilist file with n keys. Assume that the order of records in individual lists is irrelevant. Use the primitives SEARCH(X) and UPDATE(X,A) to search and update an index. UPDATE(X,A) changes the address pointer for X to A. How many disk accesses (in addition to those needed to update the index) are needed?

28. Write an algorithm to delete an arbitrary record Z from a multilist file (see exercise 27). How many disk accesses are needed?

29. Write an algorithm to insert record Z into a multilist file assuming that individual lists are ordered by the primary key KEY($Z,1$). How many accesses are needed for this (exclude index accesses)?

30. Assuming that each list in a multilist file is a doubly linked list, write an algorithm to delete an arbitrary record Z. The forward link for key i is ALINK(Z,i) while the corresponding backward link is BLINK(Z,i).

31. Write an algorithm to output all key values for record Z in a ring structure. How many accesses are needed for this? How many accesses are needed to do the same in a multilist file?

32. Write an algorithm to delete an arbitrary record Z from a ring file.

33. Describe briefly how to do the following:

(i) In a multilist organization: (a) output all records with KEY1 = PROG *and* KEY2 = *NY*. How many accesses are needed to carry this out? (b) Output all records with KEY1 = PROG or KEY2 = *NY*. How many accesses are needed for this. Assume that each access retrieves only one record.

(ii) If a ring organization is used instead, what complications are introduced into (a) and (b) above?

(iii) How could the following be carried out using an inverted organization:

a) output all records with KEY1 = PROG *and* KEY2 = *NY*

b) output all records with KEY1 = PROG *or* KEY2 = *NY*

c) output all records with KEY1 = PROG *and* KEY2 $\neq$ *NY*

How many accesses are needed in each case (exclude accesses to get indexes)?

34. A 10^5 record file is maintained as an inverted file on a disk with track capacity 5000 characters. This disk has 200 tracks on each of its 10 surfaces. Each record in the file is 50 characters long and has five key fields. Each key is binary (i.e., has only two distinct values) and so the index for each key can be maintained as a binary bit string of length 10^5 bits. If 1 character is 6 bits long, then each index takes about 4 tracks. How should the 5 indexes be stored on disk so as to minimize total seek time while processing the indexes in order to determine which records satisfy a given boolean query Q? This processing involves reading in 1 track of each index and testing the query against records represented by this track. Then the next set of index tracks is input and so on. How much time does it take to process all the indexes in order to determine which records are to be retrieved? Assume a seek time of 1/10 sec and a latency time of 1/40 sec. Also assume that only the input time is significant. If k records satisfy this query, how much more time is needed to retrieve these k records? Using other file structures it may be necessary to read in the whole file. What is the minimum time needed to read in the entire file of 10^5 records? How does this compare with the time needed to retrieve k records using an inverted file structure?

35. Determine how many disk accesses are needed to allocate and free storage using algorithms ALLOCATE and FREE of section 4.8 for the boundary tag method when the storage being managed is disk storage.

36. Write storage management algorithms for disk storage maintaining a list of free blocks in internal memory. Use first fit for allocation. Adjacent free blocks should be coalesced when a block is freed. Show that both these tasks can be accomplished in time $O(n)$ where n is the number of free blocks.

Go to [Appendix A](#)   Back to [Table of Contents](#)

APPENDIX A: SPARKS

This section is meant for people who do most of their programming in FORTRAN. FORTRAN has the distinction of being essentially the earliest higher level programming language, developed about 1957 by a group at IBM. Since then it and its derivatives have become established as the primary language for scientific and engineering computation. But, with our greater understanding of the process of creating programs has come a realization of the deficiencies of FORTRAN. Creating a program is properly thought of as taking a real world problem and translating it into a computer solution. Concepts in the real world such as a geneology tree or a queue of airplanes must be translated into computer concepts. A language is good if it enables one to describe these abstractions of the real world in a natural way. Perhaps because of its very early development, FORTRAN lacks many such features. In this appendix we explore the idea of writing a preprocessor for FORTRAN which inexpensively adds some of these missing features.

A preprocessor is a program which translates statements written in a language X into FORTRAN. In our case X is called SPARKS. Such a program is normally called a compiler so why give it the special name preprocessor? A preprocessor is distinguished from a compiler in the following way: the source and target language have many statements in common.

Such a translator has many advantages. Most importantly it preserves a close connection with FORTRAN. Despite FORTRAN's many negative attributes, it has several practical pluses: 1) it is almost always available and compilers are often good, 2) there is a language standard which allows a degree of portability not obtainable with other languages, 3) there are extensive subroutine libraries, and 4) there is a large labor force familiar with it. These reasons give FORTRAN a strong hold in the industrial marketplace. A structured FORTRAN translator preserves these virtues while it augments the language with improved syntactical constructs and other useful features.

Another consideration is that at many installations a nicely structured language is unavailable. In this event a translator provides a simple means for supplementing an existing FORTRAN capability. The translator to be described here can be obtained by writing to the address given at the end of this appendix.

In order to see the difference between FORTRAN and SPARKS consider writing a program which searches for X in the sorted array of integers A (N), $N \leq 100$. The output is the integer J which is either zero if X is not found or $A(J) = X$, $1 \leq J \leq N$. The method used here is the well known binary search algorithm. The FORTRAN version looks something like this:

```
SUBROUTINE BINS (A,N,X,J)

IMPLICIT INTEGER (A - Z)
```

```
DIMENSION A(100)

BOT= 1

TOP= N

J = 0

100  IF (BOT. GT. TOP) RETURN

MID = (BOT + TOP)/2

IF (X. GE. A (MID)) GO TO 101

TOP = MID - 1

GO TO 100

101   IF (X. EQ. A (MID)) GO TO 102

BOT = MID + 1

GO TO 100

102   J = MID

RETURN

END
```

This may not be the "best" way to write this program, but it is a reasonable attempt. Now we write this algorithm in SPARKS.

```
SUBROUTINE BINS (A,N,X,J)

IMPLICIT INTEGER (A - Z)

DIMENSION A(100)

BOT = 1; TOP = N; J = 0
```

```
WHILE BOT. LE. TOP DO

MID = (BOT + TOP)/2

CASE

: X. LT. A(MID): TOP = MID - 1

: X. GT. A(MID): BOT = MID + 1

:ELSE: J = MID; RETURN

ENDCASE

REPEAT

RETURN

END
```

The difference between these two algorithms may not be dramatic, but it is significant. The WHILE and CASE statements allow the algorithm to be described in a more natural way. The program can be read from top to bottom without your eyes constantly jumping up and down the page. When such improvements are consistently adopted in a large software project, the resulting code is bound to be easier to comprehend.

We begin by defining precisely the SPARKS language. A distinction is made between FORTRAN statements and SPARKS statements. The latter are recognized by certain keywords and/or delimiters. All other statements are regarded as FORTRAN and are passed directly to the FORTRAN compiler without alteration. Thus, SPARKS is compatible with FORTRAN and a FORTRAN program is a SPARKS program. SPARKS statements cause the translator to produce ANSI FORTRAN statements which accomplish the equivalent computation. Hence, the local compiler ultimately defines the semantics of all SPARKS statements.

The reserved words and special symbols are:

```
BY        CASE     CYCLE    DO     ELSE    ENDCASE

ENDIF     EOJ      EXIT     FOR    IF      LOOP

REPEAT    UNTIL    WHILE    TO     THEN    :
```

```
;                 //
```

Reserved words must always be surrounded by blanks. Reserved means they cannot be used by the programmer as variables.

We now define the SPARKS statements by giving their FORTRAN equivalents. In the following any reference to the term "statements" is meant to include both SPARKS and FORTRAN statements. There are six basic SPARKS statements, two which improve the testing of cases and four which improve the description of looping.

```
IF cond THEN              IF(.NOT. (cond)) GO TO 100

     S₁                            S₁

  ELSE                             GO TO 101

     S₂              100   S₂

ENDIF                     101   CONTINUE
```

S_1 and S_2 are arbitrary size groups of statements. Cond must be a legal FORTRAN conditional. The ELSE clause is optional but the ENDIF is required and it always terminates the innermost IF.

$S_1, S_2, ..., S_{n+1}$ are arbitrary size groups of statements. Cond1, cond2, ..., cond$n$ are legal FORTRAN conditionals. The symbol ELSE surrounded by colons designates that S_{n+1} will be automatically executed if all previous conditions are false. This part of the case statement is optional.

The four looping statements are:

```
WHILE cond DO         100   IF(.NOT. (cond)) GO TO 101

   S                            S

REPEAT                         GO TO 100

                      101   CONTINUE
```

S is an arbitrary group of statements and cond a legal FORTRAN conditional.

```
LOOP                      100    CONTINUE

  S                              S

UNTIL cond REPEAT                IF(.NOT. (cond)) GO TO 100
```

S and cond are the same as for the while statement immediately preceding.

```
LOOP              100          CONTINUE

  S                            S

REPEAT                         GO TO 100

                  101          CONTINUE
```

S is an arbitrary size group of statements.

```
FOR vble = expl TO exp2 BY exp3 DO

   S

REPEAT
```

This has a translation of:

```
    vble = expl

    GO TO 100

102  vble = vble + exp3

100  IF ((vble - (exp2))*(exp3).GT. 0) GO TO 101

      S

      GO TO 102

101  CONTINUE
```

The three expressions exp1, exp2, exp3 are allowed to be arbitrary FORTRAN arithmetic expressions of any type. Similarly vble may be of any type. However, the comparison test is made against integer zero. Since exp2 and exp3 are re-evaluated each time through the loop, care must be taken in its use.

EXIT is a SPARKS statement which causes a transfer of control to the first statement outside of the innermost LOOP-REPEAT statement which contains it. One example of its use is:

```
LOOP                    100     CONTINUE

 S₁                             S₁

 IF cond THEN EXIT              IF(.NOT. (cond)) GO TO 102

 ENDIF                                GO TO 101

 S₂                     102     CONTINUE

REPEAT                          S₂

                                GO TO 100

                        101     CONTINUE
```

A generalization of this statement allows EXIT to be used within any of the four SPARKS looping statements: WHILE, LOOP, LOOP-UNTIL and FOR. When executed, EXIT branches to the statement immediately following the innermost looping statement which contains it.

The statement CYCLE is also used within any SPARKS looping statement. Its execution causes a branch to the end of the innermost loop which contains it. A test may be made and if passed the next iteration is taken. An example of the use of EXIT and CYCLE follow.

```
LOOP                    100     CONTINUE

  S₁                            S₁

 CASE                           IF(.NOT. (cond1) GO TO 103

  : cond1 : EXIT                GO TO 102
```

```
 : cond2 : CYCLE              IF(.NOT. (cond2)) GO TO 104

ENDCASE            103      GO TO 101

S₂                          CONTINUE

REPEAT             104          S₂

                           GO TO 100

                   101     CONTINUE

                   102
```

EOJ or end of job must appear at the end of the entire SPARKS program. As a statement, it must appear somewhere in columns 7 through 72 and surrounded by blanks.

ENDIF is used to terminate the IF and ENDCASE to terminate the CASE statement. REPEAT terminates the looping statements WHILE, LOOP and FOR.

Labels follow the FORTRAN convention of being numeric and in columns one to five.

The use of doubleslash is as a delimiter for comments. Thus one can write

//This is a comment//

and all characters within the double slashes will be ignored. Comments are restricted to one line and FORTRAN comments are allowed.

The semi-colon can be used to include more than one statement on a single line For example, beginning in column one the statement

99999 A = B + C; C = D + E; X = A

would be legal in SPARKS. To include a semicolon in a hollerith field it should be followed by a second semicolon. This will be deleted in the resulting FORTRAN.

We are now ready to describe the operation of the translator. Two design approaches are feasible. The first is a table-driven method which scans a program and recognizes keywords. This approach is essentially the way a compiler works in that it requires a scanner, a symbol table (though limited), very limited parsing and the generation of object (FORTRAN) code. A second approach is to write a general

macro preprocessor and then to define each SPARKS statement as a new macro. Such a processor is usually small and allows the user to easily define new constructs. However, these processors tend to be slower than the approach of direct translation. Moreover, it is hard to build in the appropriate error detection and recovery facilities which are sorely needed if SPARKS is to be used seriously. Therefore, we have chosen the first approach. Figure A.1 contains a flow description of the translator.

☐

Figure A.1: Overview of SPARKS Translator

The main processing loop consists of determining the next statement and branching within a large CASE. This does whatever translation into FORTRAN is necessary. When EOJ is found the loop is broken and the program is concluded.

The SPARKS translator was first written in SPARKS. The original version was hand translated into FORTRAN to produce our first running system. Since that time it has been used by a variety of people and classes. Thus it is running far better than the original version. Nevertheless, the translator has not been proved correct and so it must be used with caution.

Extensions

Below is a list of possible extensions for SPARKS. Some are relatively easy to implement, while others require a great deal of effort.

E.1 Special cases of the CASE statement

```
    CASE SGN  : exp:              CASE: integer variable:

      : .EQ.0 : S₁                 : 1 : S₁

      : .LT.0 : S₂      and        : 2 : S₂

      : .GT.0 : S₃                 ☐

    ENDCASE                        : n : Sₙ

                                 ENDCASE
```

The first gets translated into the FORTRAN arithmetic IF statement. The second form is translated into a FORTRAN computed go to.

E.2 A simple form of the FOR statement would look like

```
LOOP exp TIMES

    S

REPEAT
```

where exp is an expression which evaluates to a non-negative integer. The statements meaning can be described by the SPARKS for statement:

```
FOR ITEMP = 1 TO exp DO

    S

REPEAT
```

An internal integer variable ITEMP must be created.

E.3 If F appears in column one then all subsequent cards are assumed to be pure FORTRAN. They are passed directly to the output until an F is encountered in column one.

E.4 Add the capability of profiling a program by determining the number of executions of each loop during a single execution and the value of conditional expressions.

HINT: For each subroutine declare a set of variables which can be inserted after encountering a WHILE, LOOP, REPEAT, FOR, THEN or ELSE statement. At the end of each subroutine a write statement prints the values of these counters.

E.5 Add the multiple replacement statement so that

$A = B = C = D + E$

is translated into

$C = D + E; B = C; A = B$

E.6 Add the vector replacement statement so that

$(A,B,C) = (X + Y, 10,2*E)$

produces A = X + Y: B = 10 ; C = 2*E

E.7 Add an array "fill" statement so that

NAME(*) ⬚ exp1,exp2,exp3

gets translated into

NAME(1) = exp1; NAME(2) = exp2; NAME(3) = exp3

E.8 Introduce appropriate syntax and reasonable conventions so that SPARKs programs can be recursive.

HINT: Mutually recursive programs are gathered together in a module, MODULE (X(A,B,C)(100)) whose name is X, whose parameters are A,B,C and whose stack size should be 100.

E.9 Add a character string capability to SPARKS.

E.10 Add an internal procedure capability to aid the programmer in doing top-down program refinement.

E.11 Attach sequence numbers to the resulting FORTRAN output which relates each statement back to the original SPARKS statement which generated it. This is particularly helpful for debugging.

E.12 Along with the indented SPARKS source print a number which represents the level of nesting of each statement.

E.13 Generalize the EXIT statement so that upon its execution it can be assigned a value, e.g.,

```
    LOOP

      S₁

      IF condl THEN EXIT : exp1: ENDIF

      S₂

      IF cond2 THEN EXIT : exp2: ENDIF

      S₃
```

```
     REPEAT
```

will assign either expl or exp2 as the value of the variable EXIT.

E.14 Supply a simplified read and write statement. For example, allow for hollerith strings to be included within quotes and translated to the nH x_1 ... x_n format.

All further questions about the definition of SPARKS should be addressed to:

Chairman, SPARKS Users Group

Computer Science, Powell Hall

University of Southern California

Los Angeles, California 9007

To receive a complete ANSI FORTRAN version of SPARKS send $20.00 (for postage and handling) to Dr. Ellis Horowitz at the above address.

APPENDIX B: ETHICAL CODE IN INFORMATION PROCESSING

ACM CODE OF PROFESSIONAL CONDUCT

PREAMBLE

Recognition of professional status by the public depends not only on skill and dedication but also on adherence to a recognized code of Professional Conduct. The following Code sets forth the general principles (Canons), professional ideals (Ethical Considerations), and mandatory rules (Disciplinary Rules) applicable to each ACM Member.

The verbs "shall" (imperative) and "should" (encouragement) are used purposefully in the Code. The Canons and Ethical Considerations are not, however, binding rules. Each Disciplinary Rule is binding on each individual Member of ACM. Failure to observe the Disciplinary Rules subjects the Member to admonition, suspension, or expulsion from the Association as provided by the Constitution and Bylaws. The term "member(s)" is used in the Code. The Disciplinary Rules of the Code apply, however, only to the classes of membership specified in Article 3, Section 4, of the Constitution of the ACM.

CANON 1

An ACM member shall act at all times with integrity.

Ethical Considerations

EC1.1. An ACM member shall properly qualify himself when expressing an opinion outside his areas of competence. A member is encouraged to express his opinion on subjects within his areas of competence.

EC1.2. An ACM member shall preface an partisan statements about information processing by indicating clearly on whose behalf they are made.

EC1.3. An ACM member shall act faithfully on behalf of his employers or clients.

Disciplinary Rules

DR1.1.1. An ACM member shall not intentionally misrepresent his qualifications or credentials to present or prospective employers or clients.

DR1.1.2. An ACM member shall not make deliberately false or deceptive statements as to the present or expected state of affairs in any aspect of the capability, delivery, or use of information processing systems.

DR1.2.1. An ACM member shall not intentionally conceal or misrepresent on whose behalf any partisan statements are made.

DR1.3.1. An ACM member acting or employed as a consultant shall, prior to accepting information from a prospective client, inform the client of all factors of which the member is aware which may affect the proper performance of the task.

DR1.3.2. An ACM member shall disclose any interest of which he is aware which does or may conflict with his duty to a present or prospective employer or client.

DR1.3.3. An ACM member shall not use any confidential information from any employer or client, past or present, without prior permission.

CANON 2

An ACM member should strive to increase his competence and the competence and prestige of the profession.

Ethical Considerations

EC2.1. An ACM member is encouraged to extend public knowledge, understanding, and appreciation of information processing, and to oppose any false or deceptive statements relating to information processing of which he is aware.

EC2.2 An ACM member shall not use his professional credentials to misrepresent his competence.

EC2.3. An ACM member shall undertake only those professional assignments and commitments for which he is qualified.

EC2.4. An ACM member shall strive to design and develop systems that adequately perform the intended functions and that satisfy his employer's or client's operational needs.

EC2.5. An ACM member should maintain and increase his competence through a program of continuing education encompassing the techniques, technical standards, and practices in his fields of professional activity.

EC2.6. An ACM member should provide opportunity and encouragement for professional development

and advancement of both professionals and those aspiring to become professionals.

Disciplinary Rules

DR2.2.1. An ACM member shall not use his professional credentials to misrepresent his competence.

DR2.3.1. An ACM member shall not undertake professional assignments without adequate preparation in the circumstances.

DR2.3.2. An ACM member shall not undertake professional assignments for which he knows or should know he is not competent or cannot become adequately competent without acquiring the assistance of a professional who is competent to perform the assignment.

DR2.4.1. An ACM member shall not represent that a product of his work will perform its function adequately and will meet the receiver's operational needs when he knows or should know that the product is deficient.

CANON 3

An ACM member shall accept responsibility for his work.

Ethical Considerations

EC3.1. An ACM member shall accept only those assignments for which there is reasonable expectancy of meeting requirements or specifications, and shall perform his assignments in a professional manner.

Disciplinary Rules

DR3.1.1. An ACM member shall not neglect any professional assignment which has been accepted.

DR3.1.2. An ACM member shall keep his employer or client properly informed on the progress of his assignments.

DR3.1.3. An ACM member shall not attempt to exonerate himself from, or to limit, his liability to his clients for his personal malpractice.

DR3.1.4. An ACM member shall indicate to his employer or client the consequences to be expected if his professional judgement is overruled.

CANON 4

An ACM member shall act with professional responsibility.

Ethical Considerations

EC4.1 An ACM member shall not use his membership in ACM improperly for professional advantage or to misrepresent the authority of his statements.

EC4.2. An ACM member shall conduct professional activities on a high plane.

EC4.3. An ACM member is encouraged to uphold and improve the professional standards of the Association through participation in their formulation, establishment, and enforcement.

Disciplinary Rules

DR4.1.1. An ACM member shall not speak on behalf of the Association or any of its subgroups without proper authority.

DR4.1.2. An ACM member shall not knowingly misrepresent the policies and views of the Association or any of its subgroups.

DR4.1.3. An ACM member shall preface partisan statements about information processing by indicating clearly on whose behalf they are made.

DR4.2.1. An ACM member shall not maliciously injure the professional reputation of any other person.

DR4.2.2. An ACM member shall not use the services of or his membership in the Association to gain unfair advantage.

DR4.2.3. An ACM member shall take care that credit for work is given to whom credit is properly due.

CANON 5

An ACM member should use his special knowledge and skills for the advancement of human welfare.

Ethical Considerations

EC5.1. An ACM member should consider the health, privacy, and general welfare of the public in the performance of his work.

EC5.2. An ACM member, whenever dealing with data concerning individuals, shall always consider the

principle of the individual's privacy and seek the following:

--To minimize the data collected.

--To limit authorized access to the data.

--To provide proper security for the data.

--To determine the required retention period of the data.

--To ensure proper disposal of the data.

Disciplinary rules

DR5.2.1. An ACM member shall express his professional opinion to his employers or clients regarding any adverse consequences to the public which might result from work proposed to him.

Go to [Appendix C](#)   Back to [Table of Contents](#)

◀ ■

APPENDIX C: ALGORITHM INDEX BY CHAPTER

CHAPTER 1--INTRODUCTION

Exchange sorting--SORT 1.3

Binary search--BINSRCH 1.3

FIBONACCI 1.4

Filling a magic square--MAGIC 1.4

CHAPTER 2**--ARRAYS**

Polynomial addition--PADD 2.2

Fibonacci polynomials--MAIN 2.2

Sparse matrix transpose--TRANSPOSE 2.3

Sparse matrix transpose--FAST-TRANSPOSE 2.3

Sparse matrix multiplication--MMULT 2.3

CHAPTER 3**--STACKS AND QUEUES**

Sequential stacks--ADD, DELETE 3.1

Sequential queues--ADDQ, DELETEQ 3.1

Circular sequential queues--ADDQ, DELETEQ 3.1

Path through a maze--PATH 3.2

Evaluation of a postfix expression--EVAL 3.3

Translating infix to postfix--POSTFIX 3.3

M-stacks sequential--ADD, DELETE 3.4

CHAPTER 4**--LINKED LISTS**

Create a list with 2 nodes--CREATE 2 4.1

Insert a node in a list--INSERT 4.1

Delete a node from a list--DELETE 4.1

Linked stacks--ADDS, DELETES 4.2

Linked queues--ADDQ, DELETEQ 4.2

Finding a new node--GETNODE 4.3

Initializing available space--INIT 4.3

Returning a node to available space--RET 4.3

Linked polynomial addition--PADD 4.4

Erasing a linked list--ERASE 4.4

Erasing a circular list--CERASE 4.4

Circular linked polynomial addition--CPADD 4.4

Inverting a linked list--INVERT 4.5

Concatenating two lists--CONCATENATE 4.5

Circular list insertion--INSERT-FRONT 4.5

Length of a circular list--LENGTH 4.5

Finding equivalence classes--EQUIVALENCE 4.6

Reading in a sparse matrix--MREAD 4.7

Erasing a linked sparse matrix--MERASE 4.7

Doubly linked list--DDELETE, DINSERT 4.8

First fit allocation--FF, ALLOCATE 4.8

Returning freed blocks--FREE

General lists--COPY, EQUAL, DEPTH 4.9

Erasing a list with reference counts--ERASE 4.9

Garbage collection--DRIVER, MARK 1, MARK 2 4.10

Compacting storage--COMPACT 4.10

Strings--SINSERT 4.11

Finding a pattern--FIND, NFIND, PMATCH, FAIL 4.11

Initializing available space in FORTRAN-INIT 4.12

Initializing available space in PASCAL-INIT 4.12

FORTRAN node routines--COEP, SCOEF, EXP, SEXP, LINK, SLINK 4.12

CHAPTER 5**--TREES**

Inorder traversal--INORDER,INORDER1,2,3 5.4

Preorder traversal--PREORDER 5.4

Postorder traversal--POSTORDER 5.4

Copy a binary tree--COPY 5.5

Equivalence of binary trees--EQUAL 5.5

Evaluating propositional expressions--POSTORDER-EVAL 5.5

The inorder successor--INSUC 5.6

Traversing a threaded binary tree--TINORDER 5.6

Insertion in a threaded tree--INSERT-RIGHT 5.6

Disjoint set union--U, F, UNION, FIND 5.8

Decision trees--EIGHTCOINS 5.8

Game playing--VE,AB(alpha-beta) 5.8

CHAPTER 6**--GRAPHS**

Depth first search--DFS 6.2

Breadth first search--BFS 6.2

Connected components--COMP 6.2

Minimum spanning tree--KRUSKAL 6.2

Shortest path, single source--SHORTEST-PATH 6.3

Shortest paths, ALL__COSTS 6.3

TOPOLOGICAL-ORDER 6.4

The first m shortest paths--M-SHORTEST 6.5

CHAPTER 7**--INTERNAL SORTING**

Sequential search--SEQSRCH 7.1

Binary search--BINSRCH 7.1

Fibonacci search--FIBSRCH 7.1

Determining equality of two lists--VERIFY1, VERIFY2 7.1

Insertion sorting--INSERT, INSORT 7.2

Quicksort--QSORT 7.3

Two-way merge sort--MERGE, MPASS, MSORT, RMSORT 7.5

Heapsort--ADJUST, HSORT 7.6

Radix sorting--LRSORT 7.7

Rearranging sorted records--LIST1, LIST2, TABLE 7.8

CHAPTER 8--EXTERNAL SORTING

K-way merging 8.2

K-way sorting--BUFFERING 8.2

Run generation--RUNS 8.2

Tape k-way merge--Ml, M2, M3 8.3

CHAPTER 9--SYMBOL TABLES

Binary search trees--SEARCH 9.1

Minimum external path length--HUFFMAN 9.1

Optimal binary search tree--OBST 9.1

Binary search tree insertion--BST, AVL-INSERT 9.2

Hashing--LINSRCH, CHNSRCH 9.3

CHAPTER 10--FILES

Searching an m-way tree--MSEARCH 10.2

Inserting into a B-tree--INSERTB 10.2

Deletion from a B-tree--DELETEB 10.2

Tree searching--TRIE 10.2

Back to [Table of Contents](#)