Fundamentals of DATA STRUCTURES IN PASCAL

ELLIS HOROWITZ • SARTAJ SAHNI

COMPUTER SCIENCE PRESS

Fundamentals of DATA STRUCTURES IN PASCAL

ELLIS HOROWITZ

University of Southern California

SARTAJ SAHNI

University of Minnesota

COMPUTER SCIENCE PRESS

Computer Science Press, Inc.
11 Taft Court
Rockville, MD 20850

2 3 4 5 6 Printing Year 89 88 87 86 85 84

Fundamentals of Data Structures in Pascal is another version of *Fundamentals of Data Structures* by Horowitz and Sahni. In the earlier work, all of the algorithms are written in the SPARKS language while in this work all the algorithms are written in the Pascal language.

Fundamentals of Data Structures in Pascal is the result of the combined efforts of the authors. Their names have been listed in alphabetical order with no implication that one is senior and the other junior.

Library of Congress Cataloging in Publication Data

Horowitz, Ellis.
Fundamentals of data structures in Pascal.

Bibliography,
Includes index.
1. Data structures (Computer science)
2. Pascal (Computer program language)
I. Sahni, Sartaj. II. Title.
QA76.9.D35H67 1983 001.64'2 83-10136
ISBN 0-914894-94-3

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 middle 1960'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 specification is 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. But today the programming language Pascal has become pervasive as the language of instruction in Computer Science departments. With that fact in mind we decided to alter our book *Fundamentals of Data Structures*, rewriting all of the algorithms into Pascal.

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 Pascal 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 finished algorithm 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 ability to 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 our original book *Fundamentals of Data Structures* and we would like to thank them here again. We wish to thank Arvind [sic], T. Gonzalez, L. Landweber, J. Mistra, 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
June 1983

CHAPTER 5 TREES

CHAPTER 6 GRAPHS

CHAPTER 7 INTERNAL SORTING

CHAPTER 8 EXTERNAL SORTING

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 algo-

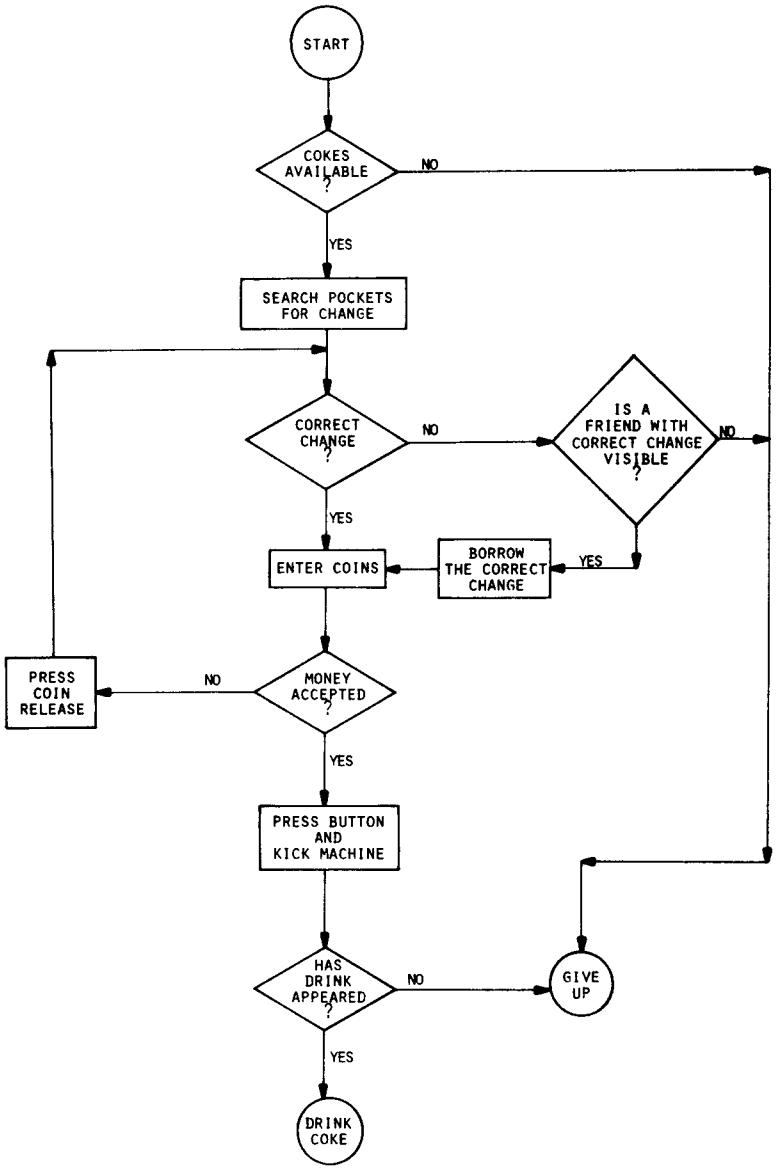

Figure 1.1 Flowchart for obtaining a Coca-Cola.

rithm, 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.

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 though 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 $(a_1, b_1)(a_2, b_2), \ldots, (a_n, b_n)$, 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). Some of the standard data types in Pascal are: integer, real, boolean, char, and array. 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.

Data object is a term referring to a set of elements, say D. For example the data object *integers* refers to $D = \{0, \pm 1, \pm 2, \ldots\}$. The data object *alphabetic character strings of length less than thirty one* implies $D = \{'', 'A', 'B', \ldots, 'Z', 'AA', \ldots\}$. 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 let's examine a modest example. Suppose we want to define the data structure natural number (abbreviated natno) where natno $= \{0,1,2,3,\ldots\}$ with the three operations being a test for zero, addition and equality. The following notation can be used:

```
      structure NATNO
1       declare ZERO() → natno
2               ISZERO(natno) → boolean
3               SUCC(natno) → natno
4               ADD(natno, natno) → natno
5               EQ(natno, natno) → boolean
6       for all x, y ∈ 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
11              EQ(SUCC(x), SUCC(y)) ::= EQ(x, y)
12      end
13    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, 1 = 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 $\mathscr{D}$, a designated domain $d \in \mathscr{D}$, a set of functions $\mathscr{F}$ and a set of axioms $\mathscr{A}$. The triple $(\mathscr{D}, \mathscr{F}, \mathscr{A})$ denotes the data structure d and it will usually be abbreviated by writing d.

In the previous example

d = natno, $\mathcal{S}$ = {natno, boolean}
$\mathcal{F}$ = {ZERO,ISZERO,SUCC,ADD}
$\mathcal{A}$ = {lines 7 thru 10 of the structure NATNO}

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 ($\mathcal{S}, \mathcal{F}, \mathcal{A}$) 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 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.