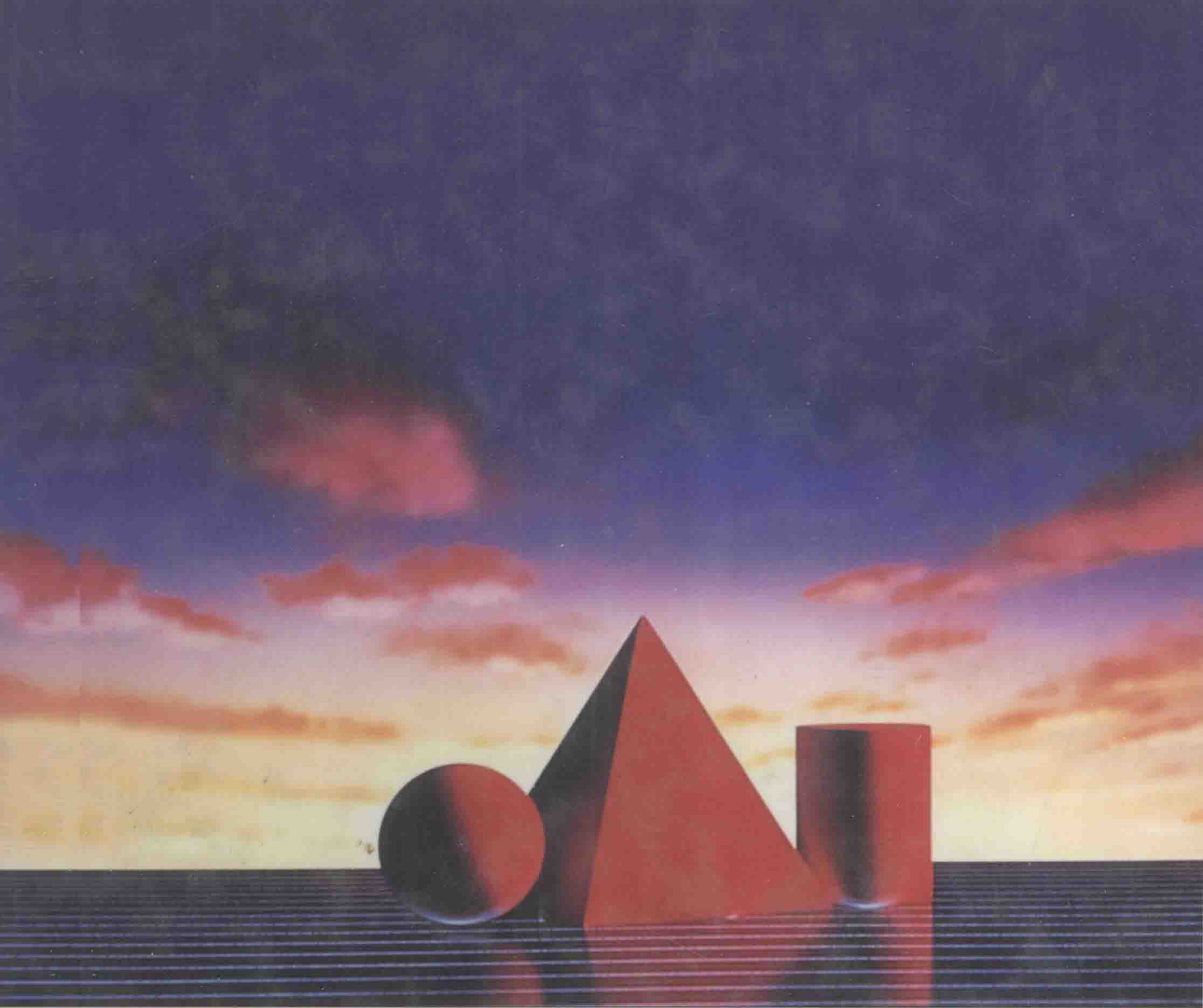

ELEMENTARY DATA STRUCTURES WITH PASCAL

ANGELA B. SHIFLET

Elementary Data Structures with PASCAL

ANGELA B. SHIFLET
WOFFORD COLLEGE

West Publishing Company
ST. PAUL · NEW YORK · LOS ANGELES · SAN FRANCISCO

Production Credits
Copyedit: Sheryl Rose
Text design: Paula Schlosser
Composition: G&S Typesetters, Inc.
Text art: G&S Typesetters, Inc.
Cover photo: Michel Tcherevkoff/The Image Bank

97 96 95 94 93 92 91 90 8 7 6 5 4 3 2 1 0

Library of Congress Cataloging-in-Publication Data

Shiflet, Angela B.
Elementary data structures with Pascal / Angela B. Shiflet.
p. cm.
ISBN 0-314-66780-6
1. Pascal (Computer program language) 2. Data structures (Computer science) I. Title.
QA76.73.P2S524 1990
005.13'3—dc20 89-29442
CIP

Elementary Data Structures with PASCAL

Dedicated to
my husband,
George,
and my parents,
Isabell and Carroll Buzzett

PREFACE

Elementary Data Structures with Pascal introduces the CS2 or elementary data structures student to the subject in a clear, visual, top-down manner. The text first presents each data structure as an abstract data type (ADT), or a set of data objects and fundamental operations on this set. The use of boldface type emphasizes changes in the structures and the values of the variables. English descriptions and accompanying diagrams, give the student a visual concept of the data structure.

Applications are then developed using pseudocode and ADT operations, often with diagrams to clarify the words of the algorithms. By first considering data structures on a high level without concern for the implementation details, the student obtains a powerful tool which simplifies the process of handling data and extends naturally the concept of structured programming. One of the major goals of data abstraction is to encapsulate the structure so that details of implementation are hidden from the user. With such information hidden, the programmer can consider applications on a higher plane with major operations as opposed to on a lower level where there is the very real danger of becoming lost in a sea of details.

After studying and using the data structure on the abstract data type level, the text examines how the structure is represented and how the operations can be written in Standard Pascal. Appendix B covers important extensions in Turbo Pascal and UCSD Pascal. Often, there are several ways to implement a particular ADT. Advantages and disadvantages of these implementations and the circumstances which make one more desirable than another are also discussed.

For the convenience of the instructor, a disk is available with Standard Pascal implementations of the abstract data types. Use of these files saves time in designing, coding, typing, and testing the Pascal implementations of the abstract data types. This disk, which is described in greater detail under *Supplementary Materials,* may be copied freely for the students.

The text presents many applications in examples, exercises, and programming projects. Each section has a number of exercises with answers to selected problems with italicized numbers in Appendix D. Among other features described below, the *Instructor's Manual* contains answers to all of the remaining exercises. Pascal code in the text, the *Instructor's Manual*, and on the disk of ADT implementations have been computer tested.

A chapter-by-chapter description of the organization of the text and a more detailed description of learning features follows.

Organization

Chapter 1

Sections 1.1 through 1.3 examine the process of developing quality software. These sections provide a good review and enhancement of the programming principles covered in the prerequisite course of Pascal programming. Early discussion of analysis of algorithms (Section 1.4) provides a measure of the efficiency of algorithms to be used throughout the text.

Chapter 2

Recursion is covered early (Section 2.1) in the text. Thus, this powerful technique is employed in the development of a number of procedures and functions. Section 2.2 compares recursion and iteration and discusses how to convert a recursive algorithm to an iterative one. The study of program verification techniques is important to the computer science major. The instructor may decide, however, to postpone its study in Section 2.3 to later in the course. The last section of the chapter introduces the topic of abstract data type, the fundamental approach to data structures employed in the text.

Chapter 3

In this chapter the text continues the study of abstract data types by examining various composite types that are built into Pascal—array, record, file, and set. After defining each as an ADT, the text presents its built-in Standard Pascal implementation along with applications. The discussion of arrays is augmented by a study of the sequential and binary search methods in Section 3.2. Depending on the level of the class, the instructor can treat this chapter as reference material or cover the topics in detail.

Chapter 4

Chapter 4 covers the ADT string along with various array implementations and applications in text editing. After the discussion in Chapter 7, a linked list implementation is considered and compared to the array one (Section 8.2). The instructor again has the option of treating the material of this chapter in a cursory or detailed fashion.

Chapter 5

Section 5.1 gives a conceptual view of a stack, defines stack as an abstract data type, and presents various short applications. The next section discusses an implementation of the ADT stack with arrays, postponing one with linked lists until Chapter 8. Additional applications in Section 5.3 include postfix notation and simulation of recursion using a stack and iteration.

Chapter 6

The format of this chapter parallels that of Chapter 5. The first section covers the abstract data type queue with several short applications, and Section 6.2 presents an array implementation. The last section develops a simulation of a waiting line at a post office with top-down design and pseudocode development of the procedures.

Chapter 7

After studying pointers in Section 7.1, the text devotes the rest of this chapter to the abstract data type linked list. Manipulations of linked lists pictorially (Section 7.2) precede the formal ADT definition (Section 7.3). Section 7.4 covers implementations with dynamic and static memory allocations. Circular, doubly, and multiply linked lists are discussed in the optional, last section along with an application to sparse matrices.

Chapter 8

In Section 8.1 the text discusses various applications of linked lists: memory management with the development of user-defined *New* and *Dispose* operations; arithmetic on very large, nonnegative integers; and a generalized list structure similar to that employed by the language LISP. In Sections 8.2 through 8.4 the text reconsiders the abstract data types of string, stack, and queue, previously implemented statically with arrays, now implemented dynamically with linked lists. For each ADT the advantages and disadvantages of these two implementation techniques are discussed. Faced with a choice in the coding of the abstract data types, in Section 8.5 the text reexamines how the programmer should develop Pascal programs with ADT objects and operations hidden, to be used like they are predefined. The student sees that by using ADT objects and operations in the development of applications, he or she can substitute different implementations of the ADT without changing significantly the code of the application program.

Chapter 9

This chapter examines the ADT table along with five implementations and applications, such as to relational data bases. Section 9.2 compares and contrasts four implementations of the ADT table involving a static memory allocation with ordered and unordered arrays and a dynamic memory allocation with ordered and unordered linked lists. The hash table implementation is presented in the last section.

Chapter 10

The first section of this chapter introduces tree terminology and traversals, while the next defines the abstract data type binary tree. Two binary tree implementations involving pointers are presented in Sections 10.3 and 10.4. With the first of these implementations, many of the algorithms are developed recursively. The instructor may choose to omit the second method, using threaded trees, which illustrates a nonrecursive alternative. Section 10.5 explores binary search trees, while optional Section 10.6 introduces AVL trees. The last of this chapter examines several applications of binary trees: expression trees, decision trees, game trees, and Huffman codes.

Chapter 11

In this chapter the text presents several algorithms for sorting: insertion sort, selection sort, quicksort, heapsort, mergesort, and sorting with a permutation array. This examination of sorting considers the complexities of the various methods and the situations for which they are best suited. The instructor may, however, choose to omit one or more of these techniques, such as the selection sort of Section 11.2. Because this chapter is substantially self-contained, the instructor also has the option of covering the material earlier in the course.

Chapter 12

The abstract data type graph, discussed in Section 12.1, is implemented with adjacency matrices (Section 12.2) and with adjacency lists (Section 12.3). Section 12.4 presents algorithms with accompanying diagrams for depth-first and breadth-first traversals of graphs. In the last section the text covers applications involving mazes, minimal spanning trees, and shortest paths in networks.

Learning Features

Chapter introductions One to several paragraphs at the beginning of each chapter give an overview of the material in the chapter.

Example operations/applications Examples help to clarify the material. The organized approach to examples, particularly with accompanying diagrams, aids understanding of the material. Applications illustrate the use of a data structure, demonstrate its importance, and provide interest.

Numerous diagrams with boldface type to emphasize changes Diagrams help students visualize the actions of operations and algorithms.

Implementations of an ADT presented after the formal definition Only after students become familiar with a data structure on the higher level of an ADT does the text consider implementation details.

Historical anecdotes Such anecdotes add interest to the text. Moreover, they often present material that a computer science major should know about the history of the discipline.

Numerous exercises at the end of each section Exercises are at the end of each section, not just at the end of the chapter. On the average there are 21 exercises per section, 95 per chapter. Exercises include short answer problems, diagrams of the execution of segments, design of procedures and functions using pseudocode and ADT operations, coding of procedures and functions, applications, and questions from the Advanced Placement Examination in Computer Science.

Answers to all exercises Answers in Appendix D to some exercises (those with italicized numbers) for each section allow students to check their work for immediate reinforcement. The *Instructor's Manual* contains answers to all the remaining exercises.

Programming projects On the average, there are 6 programming projects per chapter. These are major assignments to be implemented on the computer. By completing such a project, the student implements an application of the data structure and enhances his or her understanding of the material and abilities in software design.

Organized comparison of different implementations of ADTs These comparisons provide guidelines for the implementations which are best to employ in various situations.

Supplementary Materials

Disk of ADT implementations

A disk with implementations in Pascal of the abstract data types is available to adopters from West upon request. Each ADT implementation appears in two source files on the disk:

1. A text file in the format of a Turbo Pascal unit is ready for compilation in that version of the language. Minor modifications convert this file to one that can be used by other versions of Pascal which permit separate compilation.
2. Another text file contains the implementation as it would appear after a program statement. For appropriate versions of Pascal, this file can be used as an "include" file. If separately compiled units or include files are not available, this text file can be copied for use in a program.

A file, called "Read Me," and documentation in the *Instructor's Manual* describe how to employ these. Since the files are stored as text files, they can be easily edited with an editor or a word processor. The disk is available as a 3½″ Macintosh, 3½″ IBM, or 5¼″ IBM disk. A free site license is issued to any school which adopts the text.

Instructor's Manual

An Instructor's Manual, written by the author and John S. Hinkel, contains all solutions to text exercises that are not provided in the text appendix, answers to at least one project per chapter, and additional test problems with answers. The Pascal code was tested on a Macintosh computer using Turbo Pascal.

Acknowledgements

Many people provided valuable assistance in the completion of this book. John S. Hinkel has been extremely helpful in many ways. As coauthor of the *Instructor's Manual,* among other tasks, he wrote and implemented the programming exercises, projects, and units in Pascal. As problem checker, he carefully verified all examples and answers included in the text, testing operations on the computer where appropriate. As one of the reviewers of the first and second drafts of the manuscript, he provided insightful suggestions for improvements to the book. As a colleague, he gave me another viewpoint concerning issues relative to the text. I truly appreciate his help and friendship.

The administration at Wofford College, especially my chairman, Richard Robinson, has been generous in providing me encouragement and a reduced load to write the book.

Friends at Lawrence Livermore National Laboratory (LLNL), where I have worked for six summers, have given me many opportunities and taught me much which have enhanced this text. Special thanks go to Ted Einwohner, Bob Cralle, George Michael, and John Ranelletti.

Peter Marshall, Executive Editor at West Publishing Company, is a marvelous editor. He provided many good ideas, a clear direction, and valuable assistance. Tamborah Moore, Production Editor, kept the production moving smoothly and on time. My thanks also go to Sheryl Rose for her accurate copy editing; to Paula Schlosser for her nice design; and to Maralene Bates for her assistance in the review stage.

I am grateful to the following reviewers who offered many valuable constructive criticisms:

John Hinkel
Lander College, South Carolina

Andrew Bernat
University of Texas, El Paso

Nancy Wiegand
University of Wisconsin, Madison

Dean Arden
State University of New York, Albany

John Bertani
Oregon State University

Debra Trantina
Arizona State University

Luegina Mounfield
Louisiana State University, Baton Rouge

Krithi Ramamritham
University of Massachusetts, Amherst

Ben Moreland
University of Connecticut, Storrs

Anne Louise Radimsky
California State University, Sacramento

George Strawn
Iowa State University

Paul Schnare
Eastern Kentucky University

Ralph H. Bjork
University of Wisconsin, Platteville

My husband, George W. Shiflet, Jr., has been helpful in so many ways, including proofreading galleys and page proofs.

George and my parents, Isabell and Carroll Buzzett, have given me many years of love and encouragement along with their prayers. Without them this book would not be possible; and so, it is dedicated to those whom I love the most.

CONTENTS

CHAPTER

1

Programming Methodology

Introduction

Structures that hold data, operations that manipulate these structures, and algorithms that use these operations are the essence of data structures. Since we will be developing application programs employing data structures, we begin by studying an organized approach to the creation of programs and a measure of the efficiency of the methods we use.

In the first three sections of this chapter we examine the process of developing software with particular emphasis on the design, coding, and testing phases. Careful attention to these areas is essential to the production of good programs and software systems. Usually, we have a choice of techniques to use in the design of different parts, or modules, in a program. In the last section of this chapter we study a measure of the efficiency of algorithms. This measure gives us a basis of comparison so that we can select the most appropriate technique to use in a particular situation.

SECTION 1.1

Program Design

Data structures is a study of the various frameworks for storing data and the algorithms that implement and perform operations on these structures. Undoubtedly, you have already written a number of algorithms. An **algorithm** is a method for accomplishing something in a finite number of steps. The term is derived from the last name of a Persian mathematician, Abu Ja'far Mohammed ibn Mûsâ al-Khowârizmî, who wrote an important arithmetic textbook about 825 A.D. In the prerequisite programming course you covered algorithmic development as well as details of the Pascal language.

Before launching into a study of data structures, we will review the process of problem solving involved in software development. This process, called the **software life cycle**, has five major steps:

1. Analysis
2. Design
3. Coding
4. Testing
5. Maintenance

The first step of the software life cycle is to obtain a clear understanding of the problem through a detailed analysis. Only after a careful and thorough design process should we attempt to translate the solution into a programming language. Such attention to the design phase often will minimize the time needed for debugging. Problems revealed at a stage such as the coding or testing phase may require that we return to an earlier step and repeat part of the life cycle, making necessary changes. Many professional programmers spend a majority of their time maintaining existing systems of programs by correcting and modifying them.

In beginning computer science courses you are usually given the specifications for new problems so that your emphasis is on the design, coding, and testing steps of the software life cycle. Thus, in this chapter we cover suggestions for producing quality work in each of these three phases. The last section of the chapter covers the basis for evaluating and comparing the algorithms we develop.

In 1976 Edsger Dijkstra wrote of the importance of using **structured programming** techniques. Structured programs use only three basic constructs:

1. **Sequential structure,** with steps performed one after another in succession.
2. **Selection structure,** such as implemented with an *if-then-else* or *case* statement.
3. **Repetition structure,** such as implemented with a *for, while*, or *repeat* loop.

Every program can and should be written using only these three constructs. Structured programming has proved to be an effective technique in producing programs that are easier to write, read, test, debug, and maintain.

Another aspect of structured programming is **top-down design** or the development of a solution in a **modular** fashion starting at the highest level with repeated refinement. It is impossible to keep all aspects of a complex program in mind. We can, however, design the major steps of the solution as modules. In turn, each of these steps can be further broken down into more manageable portions or submodules. Within a program the modules are usually realized as procedures or functions with the main program invoking subprograms at the highest level of design. It is best for each subprogram to perform only one major task and usually to have no more than about 30 statements. Each of these modules can be tested and debugged independently. Consider how much easier it would be to find an error in 30 lines of code of a module as opposed to