

P a s c a l P l u s

# DATA STRUCTURES



**Fourth Edition**

.....

**Nell Dale**  
**Susan C. Lilly**

---

# **PASCAL PLUS DATA STRUCTURES, ALGORITHMS, AND ADVANCED PROGRAMMING**

---

**FOURTH EDITION**

**Nell Dale**

*The University of Texas, Austin*

**Susan C. Lilly**

*IBM*

**D. C. HEATH AND COMPANY**

Lexington, Massachusetts      Toronto

*Address editorial correspondence to:*

D. C. Heath and Company

125 Spring Street

Lexington, MA 02173

Acquisitions Editor: Randall Adams

Developmental Editor: Karen H. Myer

Production Editor: Rachel D'Angelo Wimberly

Production Coordinator: Charles Dutton

Cover: TK

This material in no way represents the opinion of IBM, nor does it reflect IBM's approval or disapproval.

Copyright © 1995 by D. C. Heath and Company.

Previous editions copyright © 1985, 1988, 1991 by D. C. Heath and Company.

All rights reserved. No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopy, recording, or any information storage or retrieval system, without permission in writing from the publisher.

Published simultaneously in Canada.

Printed in the United States of America.

International Standard Book Number: 0-669-34720-5

Library of Congress Catalog Card Number: 94-76241

10 9 8 7 6 5 4 3 2 1

*To my children, David, Joshua, Miriam, and Leah, and to my grandmothers, Mildred Lilly and Ida Schmidt, who remind me daily that no one is too young or too old to enjoy learning.*

*S.C.L.*

*To my family.*

*N.D.*

---

# Preface

---

Historically a course on data structures has been a mainstay of most Computer Science departments. However, over the last ten years the focus of this course has broadened considerably. The topic of data structures now has been subsumed under the broader topic of *abstract data types (ADTs)*—the study of classes of objects whose logical behavior is defined by a set of values and a set of operations.

Although the term abstract data type describes a comprehensive collection of data values and operations, the term data structures refers to the study of data and how to represent data objects within a program; that is, the implementation of structured relationships. The shift in emphasis is representative of the move towards more abstraction in Computer Science education. We now are interested in the study of the abstract properties of classes of data objects in addition to how these objects might be represented in a program. Johannes J. Martin puts it very succinctly: “. . . depending on the point of view, a data object is characterized by its type (for the user) or by its structure (for the implementer).”<sup>1</sup>

Although this book’s title is still *Pascal Plus Data Structures*, its emphasis also has been moving toward more abstraction over the last two editions. But in keeping with our respective backgrounds of academia and industry, this shift has been in conjunction with more attention to software engineering principles, techniques, and practice. Both of these trends are continued in this fourth edition.


## Emphasis

The focus is on abstract data types, their specification, their implementation, and their application. Within this focus, we continue to stress computer science theory and software engineering principles, including modularization, data encapsulation, information hiding, data abstraction, the top-down design of algorithms and data structures in parallel, the analysis of algorithms, and life-cycle software verification methods. We feel strongly that these principles should be introduced to computer science students early in their education so that they learn to practice good software techniques from the beginning. An understanding of theoretical concepts helps students put new ideas that they encounter into place, and practical advice allows them to apply what they have learned. Because we feel that these concepts can be taught to those with no formal mathematics, we consistently use intuitive explanations, even for topics that have a basis in mathematics, such as the analysis of algorithms. In all cases, our highest goal is to make our explanations as readable and as easily understandable as possible.

<sup>1</sup>Johannes J. Martin, *Data Types and Data Structures*, Prentice-Hall International Series in Computer Science, C. A. R. Hoare, Series Editor, 1986.

## New Material and Organizational Changes

In light of the trend toward more abstraction, we have made a number of structural changes and added new material. We now define the logical properties of an ADT by specifying the operations that can be performed on them. All operations on ADTs fall into three classes: constructor operations (operations that create or change a structure), observer operations (operations that observe the state of the structure), and iterator operations (operations that process each item in the structure). These operator classes are defined and discussed. Three new operations are defined on the List ADT—Length, ResetList, and GetNextItem. These operations allow the user to iterate through a list and process each item without having to access the list directly.

Standard Pascal is susceptible to the criticism that it does not support encapsulation, the hallmark of modern software engineering methodology, because the standard does not support separately compilable units or modules. Fortunately, however, almost all implementations of Pascal (including Turbo Pascal and Think Pascal) do support the unit construct, thus providing the encapsulation not present in the standard. Therefore, in this edition we stress encapsulation through the use of units. All ADTs are encapsulated into units. We do, however, demonstrate how to incorporate units into the code if the system being used does not support them. An icon representing a package () flags all references to units.

Many languages support the concept of a *generic* structure—a structure in which the operations on the data are defined, but the types of the data objects being manipulated are not. No Pascal implementation has this feature; however, it can be simulated. The documentation in the interface of a unit encapsulating an ADT tells the user to prepare an auxiliary unit that describes the types of the objects being manipulated; the ADT unit then accesses this auxiliary unit. This technique does not break the concept of abstraction because the user simply is tailoring the ADT to manipulate the program-dependent data objects.

We have removed the separate FindElement procedure that was internal to the List ADT in the last edition. Because experience has shown that this generalization confuses students when they first learn to manipulate linked lists, finding the element to manipulate is repeated within the code of each operation that needs it. In Chapter 7, after the students are comfortable with linked lists, we introduce this simplification.

Rather than letting the items in a list be accessed by a field named Key, the user of any ADT requiring a comparison of two items must provide Compare, a comparison function that takes two items of ItemType and returns Less, Equal, or Greater. We provide even more flexibility in ADT operations by letting the user pass procedures and functions as parameters, thus allowing data to be searched on multiple keys or a list to be ordered on different keys.

We also introduce the questions of visibility and accessibility. For example, should the structure be defined in the interface section of a unit and declared in the user program, or should the structure be defined and declared within the implementation section of the unit encapsulating the ADT?

To reinforce the concept of recursion, we first present the binary search tree algorithms in their recursive form and then translate them into iterative form. Lastly, we have combined Chapters 11 and 12 into Sorting and Searching and added radix sorting.



## Content and Organization

**Chapter 1** reviews the basic goals of high-quality software and the basic principles of software engineering for designing and implementing programs to meet these goals. Because there is more than one way to solve a problem, we discuss how competing solutions can be compared through the analysis of algorithms, using Big-O notation. The techniques for the top-down design of both programs and data structures are reviewed, with an emphasis on modularization, good programming style, and documentation. The separation of the design of problem solution from its implementation is stressed. The idea of making a schedule for completing a programming assignment is discussed.

*ListAndCount*, a sample application program in this chapter, produces a numbered program listing and source-line count of comment lines and executable lines—a tool that students can use throughout the rest of the course.

**Chapter 2** addresses what we see as a critical need in software education—the ability to design and implement correct programs and verify that they actually are correct. Topics covered in this chapter include the concept of “life-cycle” verification; designing for correctness using preconditions, postconditions, and loop invariants; the use of deskchecking and design/code walkthroughs and inspections to identify errors before testing; debugging techniques, data coverage (black box), and code coverage (clear or white box) approaches; unit testing, writing test plans, and structured integration testing using stubs and drivers.

The application section for this chapter shows how all these concepts can be applied to the development of a procedure that searches a list of items using the binary search algorithm. Students seem to think that an array and a list are synonyms. In order to make the distinction between them clear from the beginning, we encapsulate a list into a record with two fields:

Length (type Integer) and Data (type array).

The list is those elements in the array Data from 1 to Length. By making this difference explicit from the beginning, we trust that students will use the terms array and list appropriately (never interchangeably) throughout their careers.

**Chapter 3** presents data abstraction and encapsulation, the software engineering concepts that relate to the design of the data structures used in programs. We discuss three perspectives of data—abstraction, implementation, and application. Abstract data types are defined as the abstract view of data and operator classes for ADTs are discussed. We apply these perspectives to the built-in data structures that Pascal supports: arrays, records, variant records, and sets. We introduce the concept of dynamic allocation in this chapter along with the syntax for using Pascal pointer variables.

In the application section we reinforce the ideas of data abstraction and encapsulation by creating a user-defined data type, the string. Students learn to write two different kinds of documentation in the context of this application. The functional specification of the ADT is documentation for the user of the structure; it is implementation independent. The documentation within the code of the ADT operations is for the maintainer of the code; it describes the algorithm and how it is implemented.

**Chapter 4** introduces the stack data type. The stack first is considered from its abstract perspective, and the idea of recording the logical abstraction in an ADT specification is stressed. We implement the set of stack operations using an array-based structure and a linked structure; then we compare these implementations in terms of memory use and operator efficiency using Big-O.

Because the ability to encapsulate an abstract data type in a separately compiled package is so important, we introduce the concept of a unit in this chapter and use units throughout the rest of the book. The unit has three parts: the interface section, which defines the function of the operations in the unit; the implementation section, which contains the code that implements the operations; and the initialization section. The functional specification for the stack becomes the interface section of the unit encapsulating the Stack ADT. The implementation section contains the code, either the array-based or linked implementation. It is important for the students to note that the documentation for the interface section is the same for both implementations.

The application section illustrates the use of a stack in a program that evaluates infix expressions. The unit encapsulating the Stack ADT is brought into the application program through a USES clause; the only access to the elements on the stack is through the operations provided.

**Chapter 5** introduces the FIFO queue. In addition to discussing the logical properties of the Queue ADT, this chapter gives a detailed look at the design considerations of selecting among multiple implementation choices. We examine techniques for testing the Queue ADT using a batch test driver. The technique of allowing the user of an ADT to define the type of item on the structure and provide a function comparing two items is presented. Because of the generality that this technique provides, it is used throughout the rest of the text.

The application in this chapter is a generic multiple-server/single-queue queuing system simulation. The concepts of abstraction are enforced by the use of the Queue ADT.

**Chapter 6** reviews operator classes within the context of determining the specification for the List ADT. The new operations Length, ResetList, and GetNextItem are introduced and used to write three procedures—PrintList, CreateListFromFile, and WriteListToFile—that access the elements in a list without having access to the structure itself. The concept of a nontext (binary) file is presented in this chapter and used in the application. We discuss and compare an array-based implementation and a linked implementation.

The application program *AdManager* in this chapter makes use of the List ADT, the procedures written using the List ADT, and the Queue ADT, thus reinforcing the ideas of abstraction.

**Chapter 7** introduces a variety of programming techniques. We explore a number of variations of linked structures—circular linked lists, linked lists with headers and trailers, and doubly linked lists. We also introduce the concept of passing a procedure or function as a parameter. The issues of visibility and accessibility are presented in this chapter. Finally, we cover the use of static allocation to implement a linked structure.



To demonstrate incremental software development, we complete program *AdManager*.

**Chapter 8** presents the principles of recursion in an intuitive manner and then shows how recursion can be used to solve programming problems. Guidelines for writing recursive procedures and functions are illustrated with many examples. We present a simple three-question technique for verifying the correctness of recursive procedures and functions and use it throughout the rest of the book.

The application in this chapter is a recursive solution to a maze problem. The implementation is compared to a nonrecursive (stack-based) approach to demonstrate how recursion can simplify the solution to some kinds of problems.

**Chapter 9** introduces binary search trees as a way to arrange data, giving the flexibility of a linked structure with order  $\log N$  search time. In order to build on the previous chapter and exploit the inherent recursive structure of binary trees, the algorithms first are presented recursively. After all of the operations have been implemented recursively, we code *InsertTreeElement* and *DeleteTreeElement* iteratively to show the flexibility of binary search trees.

The application modifies program *ListAndCount* (from Chapter 1) to include a cross-reference generator. Here the students see how an existing program can be modified—one of the most common real-world programming assignments.

**Chapter 10** presents a collection of other branching structures: heaps, priority queues (implemented with heaps), and graphs. Binary Expression Trees (deleted in the third edition) are brought back by popular demand. The graph algorithms make use of stacks, queues, and priority queues thus both reinforcing earlier material and demonstrating how general these structures are.

**Chapter 11** covers sorting and searching. We have included radix sort after searching. We placed it at the end of the chapter to emphasize that it is not a comparison sort like the others and to be able to refer to hashing in the discussion. As an added bonus, queues are used in radix sorting, giving us one more reinforcement for the generality of the ADTs presented in this book.

## Additional Features

**Chapter Goals** A set of goals is presented at the beginning of each chapter to help the students assess what they have learned. These goals are tested in the exercises at the end of each chapter.


**Chapter Exercises** Most chapters have more than 35 exercises, new or revised for this edition. The exercises have varying levels of difficulty, including short programming problems, the analysis of algorithms, and problems that test the student's understanding of concepts. For chapters that contain application programs, there are sets of exercises that specifically pertain to the material in the application section of the chapter. These exercises are designed to motivate the students to read the appli-

cations carefully. There are also exercises marked for Turbo Pascal programmers. Approximately one-third of the exercises are answered in the back of the text; the answer key for the remaining exercises is in the *Instructor's Guide*. The numbers of those exercises answered in the text are bold faced so they can be easily identified.

**Application Programs** There are nine completely implemented applications that demonstrate a start-to-finish approach to designing a computer program from program specifications, including:

- the problem statement through to the formal specification
- the design of each data structure, using ADTs
- the source code for the program (in the text and on disk)
- additional topics (for example, testing approaches, error checking, alternate implementations, suggestions for enhancements)

Program reading is an essential skill for software professionals, but few books include programs of sufficient length for students to get this experience. There is a set of exercises and programming assignments based on each application.

**Program Disk** The source code for both data structures and application programs is included on a *disk provided with the text*. Having the source code for the ADTs on disk encourages the students to think in terms of reusable code. The source code for the larger application programs is provided to give students practice in modifying programs, without having to spend time rekeying the original program. Throughout the text a computer disk icon (  ) appears alongside a description of a program and its file name.

**Programming Assignments** A set of recommended programming assignments for each chapter is included at the end of the book. The assignments represent a range of difficulty levels and were carefully chosen to illustrate the techniques described in the text. These assignments, which include modifications and enhancements to the programs in the application sections of Chapters 1–9, give the students experience in program modification and program “maintenance.” A large selection of additional programming assignments is also available in the *Instructor's Guide*.

**Instructor's Guide** An *Instructor's Guide* is available that includes the following sections for each chapter.

- Goals
- Outline
- Teaching Notes: suggestions for how to teach the material covered in the chapter
- Workouts: suggestions for in-class activities, discussion questions, and short exercises
- Quickie Quiz Questions: additional short-answer questions that can be used in class to test student comprehension
- Exercise Key: answers to those questions that are not in the back of the book
- Programs: suitable programs in ready-to-copy format

**Transparency Masters** Figures from the text, ADT specifications, and algorithms are provided in a form ready to use as transparency masters.

## Acknowledgments

We would like to thank the following people who took the time to answer our questionnaire: Thomas Bennet, University of Missouri, Columbia; Daniel Brekke, Moorhead State University; John T. Buono, Cochise College, Sierra Vista Campus; Debra L. Burton, Texas A&M University, Corpus Christi; Saad Harous, Sultan Qaboos University (formerly of Case Western Reserve University); Wayne B. Hewitt, Johnson County Community College; Debbie Kaneko, Hampton University; Herbert Mapes, Gallaudet University; Robert E. Matthews, Armstrong State College; M. Dee Medley, Augusta College; Thomas Meyer, California State University, Bakersfield; Paul M. Mullins, Youngstown State University; Elaine Rhodes, Illinois Central College; James Robergé, Illinois Institute of Technology; and Mark A. Taylor, University of Auckland.

Anyone who has ever written a textbook—or is related to someone who has—knows the amount of time and effort that such a project takes. In theory, revised editions should not be as time consuming, but somehow they always are. Special thanks to Al, my husband and gourmet cook, and to all my children and grandchildren for their love and support. Thanks also to Lorinda who keeps the house clean and office in order. Thanks to Maggie who sleeps at my feet and Bear who thinks she must protect me from delivery people—especially those bringing packages from the publisher. Finally, thanks to all my friends who politely ignore mental lapses and sudden dashes to the computer.

Anyone who has ever written a textbook also knows that the publisher can be of great help or hindrance. Fortunately, we know of hindrance only from hearsay. We are grateful to all our Heath staff—Randall Adams, our acquisitions editor; Karen Myer, our developmental editor; Rachel Wimberly, our production editor; and Heather Monahan, our answerer of questions and review arranger. You have all been a superb team—as usual.

N.D.  
S.L.

---

# Brief Contents

---

1	<i>Programming Tools</i>	1
2	<i>Verifying, Debugging, and Testing</i>	59
3	<i>Data Design</i>	119
4	<i>Stacks</i>	189
5	<i>FIFO Queues</i>	267
6	<i>Linear Lists</i>	339
7	<i>Lists Plus</i>	413
8	<i>Programming with Recursion</i>	481
9	<i>Binary Search Trees</i>	545
10	<i>Trees Plus</i>	617
11	<i>Sorting and Searching Algorithms</i>	685
	<i>Appendixes</i>	A1
	<i>Glossary</i>	A16
	<i>Answers to Selected Exercises</i>	A27
	<i>Programming Assignments</i>	A75
	<i>Index</i>	A112

---

# Contents

---

<b>1</b>	<b>Programming Tools</b>	<b>1</b>
	<i>Goals</i>	<i>1</i>
	<i>Beyond Programming</i>	<i>2</i>
	<i>A Programmer's Toolbox</i>	<i>3</i>
	Hardware	3
	Software	3
	Ideaware	3
	<i>The Goal: Quality Software</i>	<i>4</i>
	Goal 1: Quality Software Works	4
	Goal 2: Quality Software Can Be Read and Understood	5
	Goal 3: Quality Software Can Be Modified, if Necessary, Without Excruciating Time and Effort	5
	Goal 4: Quality Software Is Completed on Time and Within Budget	6
	<i>Getting Started: Understanding the Problem</i>	<i>7</i>
	The First Step	7
	Writing Detailed Specifications	7
	<i>Focus on: Cookies for Uncle Sam</i>	<i>8</i>
	<i>The Next Step: Solving the Problem</i>	<i>10</i>
	Comparing Algorithms	11
	Big-O	13
	Some Common Orders of Magnitude	14
	<i>Focus on: Family Laundry</i>	<i>16</i>
	<i>Top-Down Design</i>	<i>17</i>
	<i>Focus on: Top-Down Design for English Majors</i>	<i>18</i>
	<i>Focus on: A Note on the Algorithm "Language"</i>	<i>19</i>
	Information Hiding	24
	<i>Designing Data Structures</i>	<i>24</i>
	<i>Focus on: When Ignorance Is Bliss</i>	<i>25</i>

## ***Implementing the Solution***      27

Comments	27
Declarations	27
Program Structures	28
Procedure or Function Calls	28
Nonobvious Code	28
Self-Documenting Code	28
Prettyprinting	29
Using Constants	30

## ***All That . . . and on Schedule***      30

### ***Summary***      31

### ***Application: Software Development Tools 1: List and Count***      32

Reviewing the Specifications	32
------------------------------	----

### ***Focus on: Estimating Lines of Code***      33

Processing Source Lines	34
Top-Down Design	35
The GetFiles Module	36
The TerminateProcessing Module	38
The ProcessProgramFile Module	38
The StringType Module	41
The Program	44
Standards for Program Formatting	50

### ***Exercises***      54

## **2 Verifying, Debugging, and Testing**      59

### ***Goals***      59

### ***Where Do Bugs Come From?***      61

Errors in the Specifications and Design	61
Compile-Time Errors	63

### ***Focus on: Use of Semicolons in Pascal***      64

Run-Time Errors	66
-----------------	----

### ***Designing for Correctness***      68

Assertions and Program Design	68
Preconditions and Postconditions	69
Loop Invariants	70



<b><i>Deskchecking, Walk-Throughs, and Inspections</i></b>	<b>75</b>
<b><i>Program Testing</i></b>	<b>77</b>
<b><i>Debugging with a Plan</i></b>	<b>82</b>
<b><i>Focus on: Advanced Debugging Techniques with Turbo Pascal</i></b>	<b>84</b>
<b><i>Developing a Testing Goal</i></b>	<b>85</b>
Data Coverage	85
Code Coverage	87
<b><i>Test Plans</i></b>	<b>89</b>
<b><i>Structured Integration Testing</i></b>	<b>90</b>
Top-Down Testing	90
Bottom-Up Testing	92
Mixed Testing Approaches	95
<b><i>Practical Considerations</i></b>	<b>95</b>
<b><i>Summary</i></b>	<b>96</b>
<b><i>Application: The Binary Search and Its Test Driver</i></b>	<b>97</b>
Searching	99
The Binary Search Algorithm	99
The Code—Version 1	103
Developing a Test Driver	103
Developing a Test Plan	107
The Code—Version 2	109
The Code—Final Version	111
<b><i>Exercises</i></b>	<b>112</b>

### **3 Data Design 119**

<b><i>Goals</i></b>	<b>119</b>
<b><i>Data from the Top Down</i></b>	<b>120</b>
<b><i>What Do We Mean by Data?</i></b>	<b>120</b>
<b><i>Data Abstraction</i></b>	<b>120</b>
<b><i>Data Structures</i></b>	<b>123</b>
<b><i>Focus on: Object-Oriented Programming</i></b>	<b>124</b>
Abstract Data Type Operator Classes	128

**Built-in Composite Data Types 129**

One-Dimensional Arrays	129
Two-Dimensional Arrays	134
Last Words on Arrays	140
Records	141
Other Pascal Built-in Data Types	143
Variant Records	143
Sets	150

**Pointers and Dynamic Memory Allocation 153**

Pointer Variables	154
Using NIL	155
Using Procedure New	156

**Focus on: Crash Protection 157**

Accessing Data Through Pointers	158
Using Dispose	159

**Summary 160****Application: Strings 161**

The Logical Level: The String ADT	161
The Implementation Level	162
Implementing the String Operations	164
Length Operation	165
CharAt Operation	166
MakeEmpty Operation	166
Append Operation	167
ReadLine Operation	168
PrintString Operation	170
Substring Operation	171
Concat Operation	173
CompareString Operation	176
Documentation in the String Package	180
Error Checking in the String Package	180

**Exercises 181****4 Stacks 189****Goals 189****The Logical Level 190**

What Is a Stack?	190
Operations on Stacks	191

<b><i>The User Level</i></b>	<b>194</b>
<b><i>The Implementation Level</i></b>	<b>200</b>
The Implementation of a Stack as a Static Array	200
Stack Operations with the Array Implementation	201
A More General Implementation	206
The Implementation of a Stack as a Linked Structure	207
Implementing Procedure Push	207
Implementing Procedure Pop	217
Implementing the Other Stack Operations	220
Comparing the Stack Implementations	222
Encapsulating a Data Structure: the Unit	223
<b><i>Debugging Hints for Pascal Pointers</i></b>	<b>229</b>
Avoiding Compile-Time Problems	229
Avoiding Run-Time Problems	230
<b><i>Stack Applications</i></b>	<b>231</b>
<b><i>Summary</i></b>	<b>232</b>
<b><i>Application: Expression Evaluation</i></b>	<b>233</b>
Data Structures	235
Top-Down Design	237
The Program	247
Error Checking	253
Other Notations for Expressions	255
Prefix Notation	255
Postfix Notation	257
<b><i>Exercises</i></b>	<b>257</b>
 <b>5 FIFO Queues</b>	 <b>267</b>
<b><i>Goals</i></b>	<b>267</b>
<b><i>The Logical Level</i></b>	<b>268</b>
What Is a Queue?	268
Operations on FIFO Queues	268
<b><i>The User Level</i></b>	<b>272</b>
<b><i>The Implementation Level</i></b>	<b>274</b>
The Implementation of a Queue as a Static Array	274
Another Queue Design	276
Comparing Array Implementations	282
The Implementation of a Queue as a Linked Structure	282