

Korsh Data Structures, Algorithms, and Program

Data Structures, Algorithms, and Program Style

James F. Korsh



FT MEADE
GenColl

QA 76
.9
.D35 K67
1986

Data Structures, Algorithms, and Program Style

James F. Korsh

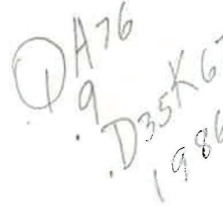
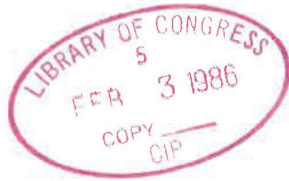
Temple University



PWS COMPUTER SCIENCE
Boston

PWS PUBLISHERS

Prindle, Weber & Schmidt • Duxbury Press • PWS Engineering • Bréton Publishers •
Staller Office Building • 20 Park Plaza • Boston, Massachusetts 02116



Copyright © 1986 by PWS Publishers.

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transcribed, in any form or by any means—electronic, mechanical, photocopying, recording, or otherwise—without the prior written permission of PWS Publishers.

PWS Publishers is a division of Wadsworth, Inc.

Library of Congress Cataloging-in-Publication Data

Korsh, James F.

Data structures, algorithms, and program style.

Includes index.

1. Data structures (Computer science) I. Title.

QA76.9.D35K67 1986 005.7'3 85-16873

ISBN 0-87150-936-9

ISBN 0-87150-936-9

Printed in the United States of America

86 87 88 89 90 — 10 9 8 7 6 5 4 3 2 1

Sponsoring Editor: Bob Prior

Signing Representative: Jack Fox

Editorial Assistant: Jane Parker

Developmental Editor: Veronica Dort

Production Editor: Ellie Connolly

Manuscript Editor: Cynthia Insolio Benn

Interior Design: Ellie Connolly

Cover Design: Julie Gecha

Interior Illustration: George Nichols

Typesetting: Beacon Graphics Corporation

Cover Printing: New England Book Components, Inc.

Printing and Binding: Halliday Lithographic

CIP EA16 6-28-85

BU02 3-18-86

One must learn by doing the thing,
for though you can think you know it
you have no certainty until you try it.

Sophocles

Preface

This text is intended for use in the just revised second programming course, CS2 in the curriculum of the Association for Computing Machinery, as well as data structures courses such as CS7 of the ACM curriculum, and independent study. A knowledge of the Pascal programming language is the only prerequisite. All programs appear in Pascal, although the concepts and methods presented are independent of the language used. The text emphasizes current techniques for the creation and testing of a program. The text stresses writing programs that are general enough to be readily adapted for use in the solution of related problems or more complex problems. This technique of program adaptation is consistently applied. Other basic programming paradigms presented are: the top-down approach, recursion (as a special case of the top-down approach), a search or traversal through the set of all solutions, and the direct construction of a solution. Sophisticated concepts are presented in a clear, informal way, theory always being coupled with concrete practical examples.

In terms of organization, the table of contents speaks for itself, but a few comments may prove helpful.

Chapters 1 to 6 deal with the fundamental concepts of program and data structure. Combined with selected topics from Chapter 7, they provide more than enough subject matter for CS2, the second programming course. Some instructors may wish to include or substitute topics from later chapters.

Chapters 7 to 12 deal with important and illustrative applications of the earlier concepts and introduce advanced data structures. They contain enough material for a solid data structures course. Many of the case studies from earlier chapters might profitably be presented in such a course. Chapters 8 to 12 may be covered in arbitrary order, with the preceding chapters serving as a quick review of the basics.

Considerable care has been taken to include sets of exercises that are graduated in degree of complexity or difficulty. Also included are suggested assignments that incorporate essential concepts of each chapter. You will find that each program is thoroughly discussed. For the most part, however, it is up to the reader to combine structural components to achieve an executable program. Thus a student must understand the program in order to use it.

Several analogies are useful in explaining structured programming. Knowing the grammar of a language does not ensure quality writing. Learning to write well



entails internalizing principles of good style and is an advanced task beyond simply learning a language. In the same way, learning to write good programs requires skill in the use of a wide range of tools, and involves much more than acquiring a language. When the master mechanic opens the hood of a well-designed automobile, he doesn't see just a jumble of wires and metal, but essential components properly linked and aligned. The knowledgeable programmer should be able to see at a glance the major components and linkages of a finely crafted program. Such a program consists of many threads carefully interwoven to support the whole. A weak thread can have serious consequences, yet the replacement of a thread must not cause the structure to unravel. Program structure enhances or obscures a program's intelligibility, while efficiency is determined by the details of a program's parts.

This book demonstrates the tools needed to craft fine programs and to solve substantial problems. These tools, although few in number, must be used with care and forethought. The tool kit includes the basic paradigms for problem solving, for structuring programs, and for structuring data. Their proper use involves data abstraction, modularization, and the intelligent choice of data structure. This text stresses the entire process of program development, and will enable you to write a well-designed program and to recognize one when you see it. At the same time, you will develop insight into program analysis, learning how to analyze programs in order to determine their correctness and efficiency.

Many individuals contributed to this book. Pam Bennett typed the manuscript. Colleagues and students deserving special thanks are Frank Friedman, Len Garrett, Mordechai Halpern, Giorgio Ingargiola, Elliot Koffman, Seymour Lipschutz, Alex Pelin, Artur Ramer, Jan Stone, Judy Weiner, Brad Wilson, Sharon Doherty, Dan Joyce, Bob Mulholland, Regis McKenzie, Ken Sall, Judy Stull, and Jane Turk.

The staff of PWS Publishers, including Ellie Connolly, Veronica Dort, Karin Ellison, and Bob Prior, all provided valuable support and guidance. The technical editing of Cynthia Insolio Benn was a particular blessing. It is also a delight to acknowledge the painstaking and helpful work of the reviewers: John Beidler, University of Scranton; Sudesh Bhatia, Middlesex Community College; John Davenport, Georgia Southern College; Michael Donlavey, Boston College; Gary Ford, University of Colorado; William Ford, University of the Pacific; John Herman, Western Michigan University; Clyde Kruskal, University of Illinois; Steven Reiss, Brown University; Richard Schlichting, University of Arizona; Jan Stone and Harold Stone, The Interfactor, Inc.; Helen Takacs, Mississippi State University; Alan Tharp, North Carolina State University; Brad Wilson, Bowling Green State University; and Scott Woodfield, Arizona State University.

Harold Stone was kind enough to contribute the historical notes in Chapter 1 and Roger Webster provided the data for the table of sorting times in Chapter 7.

Finally, without the support of my wife, Nina, and my children, Eric, Aaron, and Joanna, it would have been impossible ever to finish the task of writing this book. Nina is also responsible for numerous improvements in the presentation. Any errors, of course, are due to the devil in us all.

Contents

FUNDAMENTALS ☐☐☐☐☐☐☐☐☐☐☐☐☐☐☐☐☐☐☐

- Preface iii
- 1 Programming Structure 1
 - 1.1 Structure: Data and Program 1
 - 1.2 A Look Ahead 3
 - 1.3 The Need for Structure 3
 - 1.3.1 Using Structure 6
 - 1.3.2 A Prime Example 7
 - 1.4 Basic Constructs 12
 - 1.5 Top-Down Design 14
 - 1.6 High-Level Languages 15
 - 1.6.1 Data Types 16
 - 1.6.2 Data Representation 16
 - 1.7 Elements of Style 17
 - 1.7.1 Expressing Purpose 18
 - 1.7.2 Communication between Modules 19
 - 1.7.3 Ensuring against Data Errors: Defensive Programming 20
 - 1.7.4 Changing Programs 21
 - 1.7.5 Verification and Debugging 22
 - 1.7.6 In a Nutshell 24
 - 1.8 Time and Storage Requirements 24
 - 1.8.1 Analyzing the Prime Example 25
 - 1.8.2 Limits on Time and Storage 26
 - 1.9 Case Study: Bowling Scores 27
 - 1.9.1 Algorithm 27
 - 1.9.2 Program 31
 - 1.9.3 Debugging 36
 - 1.10 Historical Notes 44
 - 1.11 Summary 46
 - Exercises 46

2 Records, Arrays, and Pointers 48

- 2.1 Bricks 48
 - 2.1.1 Pascal Records 49
 - 2.1.2 Arrays in Pascal 50
 - 2.1.3 Representation of Records in Arrays 56
 - 2.1.4 Variable-Length Records 60
 - 2.1.5 Variant Records 62
 - 2.1.6 Address Calculation without Access 62
- 2.2 Mortar 63
 - 2.2.1 Pointers 63
 - 2.2.2 Pointer Variables and the Pascal Heap 65
- 2.3 Representations of Two-Dimensional Arrays 67
 - 2.3.1 Rowwise and Columnwise Representation 68
 - 2.3.2 Symmetric Array Representation 70
 - 2.3.3 Pointer Array Representation 71
- 2.4 Advantages and Disadvantages of the Techniques 73
- 2.5 Case Study: The Stable Marriage Problem 74
 - 2.5.1 The Algorithm 75
 - 2.5.2 The Program 78
- Exercises 81
- Suggested Assignment 83

3 Lists 86

- 3.1 Why Are Lists Needed? 86
- 3.2 Keeping Track of List Pointers 88
 - 3.2.1 Insertion and Deletion of Records in Lists 90
- 3.3 Expanding and Contracting Lists 91
 - 3.3.1 Insertion 91
 - 3.3.2 Special Cases 92
 - 3.3.3 Header Records 93
 - 3.3.4 Deletion 93
- 3.4 Traversal of Lists 95
 - 3.4.1 List Reversal Using a Loop 95
 - 3.4.2 A General List Traversal Using a Loop 98
 - 3.4.3 The Merits of Modularization 99
- 3.5 Using the TRAVERSE Procedure for Lists 99
- 3.6 Implementing Lists 108
 - 3.6.1 Lists Stored in the Pascal Heap 108
 - 3.6.2 Lists Stored in an Array of Records 109
 - 3.6.3 Lists Stored in Languages without Records 110
- 3.7 Keeping Track of Available Records 111
 - 3.7.1 Why Heap? (Using Dynamic Storage Management) 112
 - 3.7.2 How to Heap 113
- 3.8 Sequential Arrays versus Lists for Record Storage 116
- 3.9 Case Study: Merging and the Perfect Shuffle 117
 - 3.9.1 The Perfect Shuffle 117

3.9.2 Array Implementation of the Perfect Shuffle	118
3.9.3 List Implementation of the Perfect Shuffle	120
3.9.4 Merging Sequences of Entries	124
Exercises	126
Suggested Assignment	129

4 Introduction to Recursion, Stacks, and Queues 130

4.1 What Is Recursion?	130
4.2 Using Recursion	131
4.2.1 The Towers of Hanoi	131
4.2.2 Verifying and Simulating a Recursive Program	134
4.2.3 The Length of a List	137
4.2.4 Copying a List	138
4.2.5 var versus Non- var Parameters	139
4.2.6 Counting Squares	142
4.2.7 Permutations	143
4.3 A Close Look at the Execution of Recursive Programs	150
4.4 Implementing Recursive Programs	153
4.4.1 A Sample Implementation of TOWERS	154
4.4.2 A Sample Implementation of PERMUTATIONS	158
4.5 Stacks	161
4.5.1 Array Implementation of a Stack	162
4.5.2 List Implementation of the Stack with an Array of Records	164
4.5.3 List Implementation of the Stack with Records in the Pascal Heap	165
4.6 Case Study: Checking Sequences for Proper Nesting	166
4.7 Queues	172
4.7.1 Array and Circular Array Implementation of the Queue	172
4.7.2 List Implementation of the Queue with an Array of Records	174
4.7.3 List Implementation of the Queue with Records in the Pascal Heap	174
Exercises	175
Suggested Assignment	178

5 More Complex Lists 179

5.1 Imposing List-structure on Data	179
5.2 Traversal of List-structures	183
5.2.1 An Iterative Procedure	185
5.2.2 A Recursive Procedure	187
5.3 Using the TRAVERSE Procedures for List-structures	189
5.4 Implementing List-structures	194
5.5 Case Study: Information Retrieval	195
5.5.1 Family Relationships	196

5.5.2 A First Solution	196
5.5.3 A Better Solution	197
5.5.4 Updating the System	198
5.5.5 Retrieving Information from the System	204
5.5.6 Other System Components	205
Exercises	205
Suggested Assignment	208

6 Trees 211

6.1 Branching Out	211
6.1.1 Depth versus Capacity	215
6.2 Trees of Records	216
6.2.1 Insertion and Deletion of Records in Binary Trees	216
6.2.2 Climbing Binary Trees	218
6.2.3 Three Preorder Traversals of a Binary Tree	220
6.3 Using the Traverse Procedures for Binary Trees	223
6.4 Implementing Binary Trees	226
6.4.1 Sequential Representation	226
6.4.2 Linked Representation	227
6.4.3 List-structure Representation	229
6.5 Trees	229
6.5.1 Implementing Trees as Binary Trees	229
6.6 Traversals of Trees	232
6.6.1 Obtaining the Binary Tree for a Tree	233
6.6.2 Backtracking: The N -queens Problem	237
6.6.3 Depth-First	240
6.6.4 Breadth-First	245
6.6.5 Branch and Bound	245
6.7 More on Recursion, Trees, and Stacks	246
6.7.1 Balanced Binary Trees	246
6.7.2 Trading Storage for Time	250
Exercises	252
Suggested Assignment	257

APPLICATIONS ☐ ☐ ☐ ☐ ☐ ☐ ☐ ☐ ☐ ☐ ☐ ☐ ☐ ☐ ☐ ☐ ☐ ☐

7 Searching and Sorting 258

7.1 Overview	258
7.2 Elementary Searches	259
7.2.1 Linear Search	260
7.2.2 Saving Time: A Neat Trick	260
7.2.3 Binary Search	262

7.2.4	Timing the Binary Search	264
7.2.5	Interpolation Search	264
7.3	Elementary Sorts	265
7.3.1	Maximum Entry Sort	265
7.3.2	Bubble Sort	267
7.3.3	Timing the Bubble Sort	268
7.3.4	Insertion Sort	269
7.3.5	Timing the Insertion Sort	270
7.3.6	Attempted Improvements	271
7.4	Heapsort: A Faster Sort	272
7.4.1	Heapsorts	273
7.4.2	Creating a Heap	275
7.4.3	Timing the Heap Creation	275
7.4.4	Better Heap Creation	277
7.4.5	Some Details of Heap Implementation	280
7.4.6	Reheaping	280
7.4.7	An Implementation for Heapsort	281
7.5	Quicksort: Another Fast Sort	283
7.5.1	Two Quicksort Procedures	285
7.5.2	The Partition Module	287
7.5.3	Analyzing Iterative Quicksort	290
7.5.4	The Worst and Best Cases	291
7.5.5	Distributive Partitioning	292
7.6	Priority Queues	292
7.6.1	Simple Implementations	293
7.6.2	Using a Heap	293
7.6.3	Using Leftist Trees	294
7.7	Binary Search Trees	296
7.7.1	Searching the Search Tree	297
7.7.2	Growing the Search Tree "Simply"	298
7.7.3	The Shape of Simple Binary Search Trees	299
7.7.4	Deleting a Record "Simply"	300
7.7.5	A Balancing Act	300
7.7.6	Maintaining Balance	308
7.7.7	Random Access in AVL Trees	317
7.8	Hash Tables	318
7.8.1	Building a Hash Table	319
7.8.2	Searching a Hash Table	320
7.8.3	Random Hashing	322
7.8.4	Other Collision Resolution Policies	323
7.8.5	Searching in Ordered Hash Tables	331
7.8.6	Deletion from Hash Tables	331
7.9	Simulation of an Algorithm	332
7.10	Synopsis of Search and Sort Efficiencies	335
	Exercises	337
	Suggested Assignment	340

⌘ **Contents**

8 Files 341

- 8.1 The File Data Structure 341
- 8.2 Internal and External Memory 342
 - 8.2.1 Operating Systems 342
 - 8.2.2 Filters 342
- 8.3 Organization of Files 344
 - 8.3.1 Sequential Files 345
 - 8.3.2 Text Files 351
- 8.4 Sequential Access in External Memory 352
- 8.5 Sorting Tape Files 354
 - 8.5.1 Straight Merge 354
 - 8.5.2 Natural Merge 356
 - 8.5.3 Replacement Selection 357
 - 8.5.4 Polyphase Sort 358
- 8.6 Direct Access in External Memory 360
- 8.7 Sequential and Random Access of Disk Files 362
 - 8.7.1 Sequential Access 362
 - 8.7.2 Random Access 364
 - 8.7.3 Indexed Sequential Access 364
 - 8.7.4 B-trees for Sequential and Random Access 366
- 8.8 Summary 372
- Exercises 373
- Suggested Assignment 374

9 Topological Sorting: An Archetypal Solution 376

- 9.1 Background
 - 9.1.1 Binary Relations and Partial Orders 377
 - 9.1.2 Graphic Representation of Partial Orders 378
 - 9.1.3 Topological Sorts: Consistent Rankings 379
- 9.2 A Searching Solution 381
- 9.3 A Constructed Solution 381
 - 9.3.1 Correctness 382
 - 9.3.2 An Initial Implementation 384
 - 9.3.3 A Better Implementation 385
- 9.4 Analysis of TOPSORT 395
- 9.5 Behavior for Replicated Pairs or Loops 396
- 9.6 Final Comments on TOPSORT 397
 - 9.6.1 An Input Validation Module 398
- 9.7 Reviewing Methodology 398
- Exercises 398
- Suggested Assignment 400

10 Compilers 402

- 10.1 How Compilers Work 402

10.2	Language Specification	403
10.2.1	Derivations and Derivation Trees	404
10.2.2	Syntax and Semantics	406
10.2.3	Parsing	408
10.2.4	Lexical Analysis and Symbol Tables	412
10.2.5	Translation	414
10.3	Evaluation and Translation of Expressions	415
10.3.1	Postfix Expressions	416
10.3.2	Infix Expressions	418
10.3.3	Translating Infix to Postfix	422
10.4	Top-down and Bottom-up Compilers	422
10.5	A Simple Compiler	424
10.6	Compiler Conclusions	431
	Exercises	431
	Suggested Assignment	432

11 Huffman Coding and Optimal and Nearly Optimal Binary Search Trees 433

11.1	Techniques for Compressing Text or Storing Records	433
11.2	Weighted Path Length	434
11.3	Huffman Coding	435
11.3.1	The Huffman Algorithm	438
11.3.2	Representation of Huffman Trees	440
11.3.3	Implementation	440
11.3.4	A Proof	442
11.4	Optimal Binary Search Trees	444
11.4.1	Finding Optimal Binary Search Trees	445
11.5	Nearly Optimal Binary Search Trees	449
11.5.1	Greedy Binary Search Trees	450
11.5.2	Greedy Construction	452
11.5.3	Implementation	454
11.5.4	Alphabetic Codes	455
11.6	Conclusion	455
	Exercises	457
	Suggested Assignment	459

12 Some Pointers on Storage Management 460

12.1	The Need for Storage Management	460
12.2	The Heap	462
12.3	The List of Available Space for Variable-Length Records	463
12.3.1	Two-Way Circular Lists	465
12.3.2	The Buddy System	466
12.3.3	Indexing Methods	468
12.4	Shared Storage	469

xii **Contents**

12.5 Storage Reclamation Techniques	470
12.5.1 Garbage Collection	470
12.5.2 The Reference Counter Method	471
12.6 Stackless Traversals	475
12.6.1 Threaded Trees	477
12.6.2 Link Inversion	478
12.6.3 Robson Traversal	480
12.7 Pitfalls: Garbage Generation and Dangling References	483
12.8 Conclusion	486
Exercises	486
Suggested Assignment	488

Bibliography 489

Index 494

For this implementation the null pointer value is zero. The implementation for the modules corresponding to those of Section 3.6.1, except for AVAIL, are

```

function NEXT(POINTER : listpointer) : listpointer;
begin
    NEXT := RECORDS[POINTER].LINK
end;

procedure SETNULL(var POINTER : listpointer);
begin
    POINTER := 0
end;

procedure SETLINK(POINTER1, POINTER2 : listpointer);
begin
    RECORDS[POINTER1].LINK := POINTER2
end;

procedure SETINFO(POINTER : listpointer; INFORMATION : infofield);
begin
    RECORDS[POINTER].INFO := INFORMATION
end;

```

This implementation assumes that RECORDS is declared nonlocally to these modules, so that they may reference it. With this implementation, the RECORDS array plays the role of the heap. Since the procedure NEW, which allocates storage from the heap, is not usable here, it is necessary to write an AVAIL procedure that, in effect, is equivalent to NEW. AVAIL must keep track of those slots in RECORDS that are not in use, and hence can be allocated when record storage is requested.

3.6.3 Lists Stored in Languages without Records

In some languages, such as FORTRAN, there are no heaps, nor are there pointer variables or records. In such a language, the records themselves must be implemented by the programmer. They may be stored in single or multiple arrays as discussed in Chapter 2. Since the records now have an additional link field indicating their successor, they need not be stored sequentially and may be located anywhere in the array or multiarrays, as long as they do not overlap. An additional variable is needed for the head of the list. Once these arrays are set up, the programmer may write functions to access each field of a record and procedures to copy one record's value into another record, or to copy a record's field value into another record's field. These might be

```

NEXT(POINTER)
Returns a copy of the link field value of the record pointed to by POINTER.

INFO(POINTER)
Returns a copy of the information field value of the record pointed to by
POINTER.

NAME(POINTER)
Returns a copy of the name field value of the record pointed to by POINTER.

```

SETLINK(POINTER1, POINTER2)

Sets the value of the link field of the record pointed to by POINTER1 to POINTER2.

SETINFO(POINTER1, POINTER2)

Sets the value of the information field of the record pointed to by POINTER1 to the value of the information field of the record pointed to by POINTER2.

SETNAME(POINTER1, POINTER2)

Sets the value of the name field of the record pointed to by POINTER1 to the value of the name field of the record pointed to by POINTER2.

In FORTRAN, the array or multiarray in which the records are kept would be declared to be stored in the FORTRAN common area or else must be passed as parameters to these modules. Once they are written, for all intents and purposes, the language has records. These records can be accessed, copied, or modified by using these modules. Although the programmer must do all this work initially, from this point on the situation is conceptually the same as in Section 3.6.2. That is, the actual implementation of the records may be forgotten; they are treated as if stored in an array of records just as in Pascal. The problem of writing an AVAIL procedure still remains, and it is addressed in the next section.

In FORTRAN, although storage for records kept in lists must be managed, the programmer does not need to manage storage for individual elements or for arrays of elements. To refer in a program to an element *N*, or an array *DATA*, simply declare *N* to be an element, and *DATA* to be an array. Then use the operations available in FORTRAN to enter specific values into *N* or *DATA*. This works because individual elements and arrays are data structures directly available in FORTRAN. Lists, on the other hand, must be built up explicitly by storing their records in arrays. Consequently, the programmer must determine where storage is available to create a record when the need arises.

In newer languages, such as Pascal, it is not necessary to explicitly implement records of lists in arrays, as done here. It is also not necessary to be concerned with the problems of storage allocation or reclamation. Instead, dynamic data structures may be used. A data structure is dynamic if it is built during execution of a program and storage is allocated for its components at that time. Dynamic data structures are built simply by asking for storage to be allocated for a record in the heap. A pointer will then be assigned to point to that storage. Using the pointer, appropriate values may be assigned to the fields of the new record. These fields may be referred to directly, using naming conventions in the language. Special functions to store or access information from the fields of the record need not be written.

Records still need to be inserted properly in a dynamic data structure. Commands directly available in the language may be used to create or reclaim records, or reclamation may be left to implicit storage reclamation provided by the language. Thus the definition, creation, reclamation, and manipulation of dynamic data structures becomes easier.

3.7 Keeping Track of Available Records

Keeping track of and allocating storage that is needed during the execution of a program is called *dynamic storage management*. In some languages, notably FORTRAN,

storage for arrays is allocated storage before the program's execution, and no dynamic storage management is necessary. In block-structured languages like Pascal, storage is allocated to the local variables of a component (or block) at the start of its execution and remains assigned until the component (or block) completes its execution. The space can then be reassigned to the next executing component (or block). In effect, it disappears as far as the component (or block) is concerned. This is why values of local variables are not preserved between invocations of a module. Management of this storage during the execution of a program is not what is meant by dynamic storage management. Contrast this situation to heap storage, which is assigned or taken away from records in the heap during the execution of a component (or block) itself. This captures the true flavor of dynamic storage allocation.

If the array in which records are to be kept (as in Sections 3.6.2 and 3.6.3) is to function as a heap, then allocating storage in the array for a record requires dynamic storage management. In fact, this amounts to an implementation of a Pascal-like heap. But although storage can thus be managed explicitly for a specific program, the heap may manage storage for many programs and is hence more efficient use of storage. The difference is that one large array may function as a heap to be used by all the programs, instead of dedicating separate smaller arrays to each program.

3.7.1 Why Heap? (Using Dynamic Storage Management)

Why is storage efficiency important? Why not find storage for a new record in the array simply by placing it sequentially below the last new record's storage? After all, this would be very easy to do and would certainly use the storage as efficiently as possible. The answer is that this simple solution is a great idea—provided that all the storage needed to declare the array to be of sufficient length is available. The catch is that this proviso is rarely the case. Often, the amount required isn't known in advance, and even when it is known, not enough storage may be available.

Reconsider the prime number example of Chapter 1. It is probably evident by now that the list is a good data structure to use for the collection C . The program to generate all the primes between 2 and N uses very little storage besides that needed for the list. But how large should you make the array in which to store the list? It is possible to modify the program so it need not actually store all integers between 2 and N in the list, only those that are prime. Still, how many primes are there no greater than N ? It is even possible to write such a program so it efficiently generates and prints all primes $\leq N$ yet stores only those $\leq \sqrt{N}$ in the list. But again, how many of these are there?

Suppose you are a programmer asked to store the contents of ten manuscripts in ten lists. If each can have a maximum length of 50,000 words, then storage for 500,000 words is needed. How long is the longest word? You see the problem. Even if the amount of storage could be estimated, sufficient storage may not be available. You might either conclude that the task cannot be done, or you might use secondary storage (magnetic tape, disks). But because secondary storage is not random access, it is slow, and it requires different techniques than those described so far. One possibility remains. Maybe the manuscripts to be stored, perhaps even more than 10, are compatible in the sense that they are not all of great length simultaneously. Perhaps the lengths will vary as the manuscripts are processed by a program, so that at any one time the manuscripts being stored can fit. This means they may be growing

