SCHAUM'S outlines

DATA STRUCTURES WITH C

SEYMOUR LIPSCHUTZ

- Implementation of algorithms and procedures using C
- Simplified presentation of Arrays, Recursion, Linked Lists, Queues, Trees, Graphs, Sorting & Searching Methods and Hashing
- Excellent pedagogy. Includes
  - 255 Solved examples and problems
  - 86 C Programs
  - 160 Supplementary problems
  - 100 Programming problems
  - 135 Multiple-choice questions

For sale in India, Nepal, Bangladesh, Sri Lanka and Bhutan only

Tata McGraw-Hill

Data Structures With C

Adapted in India by arrangement with The McGraw-Hill Companies, Inc., New York

Sales Territories: India, Nepal, Bangladesh, Sri Lanka and Bhutan

ISBN (13): 978-0-07-070198-4
ISBN (10): 0-07-070198-9

Vice President and Managing Director—McGraw-Hill Education, Asia Pacific Region: *Ajay Shukla*

Head—Higher Education Publishing and Marketing: *Vibha Mahajan*
Manager—Sponsoring (SEM & Tech. Ed.): *Shalini Jha*
Asst Sponsoring Editor: *Surabhi Shukla*
Development Editor: *Surbhi Suman*
Executive—Editorial Services: *Sohini Mukherjee*
Jr Manager—Production: *Anjali Razdan*
Dy Marketing Manager—SEM & Tech Ed: *Biju Ganesan*

General Manager—Production: *Rajender P Ghansela*
Asst General Manager—Production: *B L Dogra*

Typeset at Bukprint India, B-180A, Guru Nanak Pura, Laxmi Nagar, Delhi 110 092, and printed at
Lalit Offset Printer, 219, F.I.E., Patpar Ganj, Industrial Area, Delhi 110 092

Cover: SDR Printers

RQXLCRBZDLLBC

The McGraw·Hill Companies

Contents

A Word to the Readers of the Special Indian Edition

Data Structures is a subject of primary importance to the discipline of Computer Science and Engineering. It is a logical and mathematical model of storing and organizing data in a particular way in a computer, required for designing and implementing efficient algorithms and program development.

Different kinds of data structures like arrays, linked lists, stacks, queues, etc., are suited to different kinds of applications. Some specific data structures are essential ingredients of many efficient algorithms, and make possible the management of huge amounts of data, such as large databases and Internet indexing services. For example, B-trees are particularly well-suited for implementation of databases, while compiler implementations usually use hash tables to look up identifiers.

Nowadays, various programming languages like C, C++ and Java are used to implement the concepts of Data Structures, of which C remains the language of choice for programmers across the world. This book provides the implementation of algorithms and pseudocodes using C in every chapter, thereby, making it easier for the readers to comprehend the theory. Multiple-Choice Questions included in the text are aimed to help students practice the learnt concepts. Thus, we hope that this book will be an excellent self-teach and test-preparation material for beginners.

Salient Features

- Demonstrates the implementation of algorithms and procedures related to data-structure concepts using the C programming language
- Offers simplified presentation for important topics—Arrays, Recursion, Linked Lists, Queues, Trees, Graphs, Sorting and Searching Methods, Hashing
- ADT representation of Arrays, Strings, Linked Lists, Stacks and Queues
- Provide apt discussions on notations of Algorithm complexity, Representation of polynomials using arrays, and linked lists, Dynamic memory management, Josephus problem, Linked list and queue operations, Application of stacks, queues and trees, Spanning trees, AVL-trees, *m*-way trees, B-trees, B+-trees, Red-black trees, Sorting algorithm, Hash table
- Excellent pedagogical features:
  - 180 Solved Examples
  - 86 C Programs
  - 175 Solved Problems
  - 160 Supplementary Problems (unsolved)
  - 100 Programming Problems
  - 135 Multiple-Choice Questions

Chapter Highlights

Chapter 1 gives an overview of data structures and discusses classification of data structures, Abstract Data types (ADT), algorithm complexity while mathematical and algorithmic notations, control structures, subalgorithms, variables and datatypes are covered in **Chapter 2**.

String processing, string operations and strings as ADT are taken up in **Chapter 3**. Word or text processing are also discussed in this chapter.

Chapter 4 explains arrays, records and pointers. Topics like arrays as ADT, storage representations, representation of polynomials using arrays, addition of polynomials, dynamic memory management pointers, records and matrices are covered in this chapter.

Chapter 5 is on linked lists. It includes linked lists as ADT, operations using linked lists, header linked lists, doubly linked lists, circularly linked lists, garbage compaction, Josephus problem and its solution, representation and manipulations polynomials using linked lists and buddy systems (in brief).

Stacks, queues and recursion and their applications are discussed in **Chapter 6**. Array, linked list and ADT representation of stacks and queues, polish notations using stacks, queue operations, circular queues, dequeues, priority queues, maze problem, simulation of queues, categorizing data and decimal to binary conversion are dealt in this chapter.

Chapter 7 is on binary trees. It provides information on traversal of binary trees, threaded binary trees, binary search trees, balanced binary trees, AVL search trees, *m*-way search trees, B-trees, B+-trees, red-black trees and applications of trees (expression Trees; game Trees).

Chapter 8 presents the concepts of graphs and their applications. Spanning trees, Minimum spanning tree algorithms—Prim's and Kruskal's algorithms, Directed and bi-connected graphs are covered here.

Finally, sorting algorithms such as shell sort, *K*-way merge sort, balanced merge sort, polyphase merge sort, two-way merge sort, and efficiency considerations in searching and sorting are discussed in **Chapter 9**. Topics such as merging ordered and unordered files, sort order and sort stability, hash table and hash functions are also covered in this chapter.

The Schaum's Outlines Advantage

A high-performance study guide, **Schaum's Outlines** help you cut study time, hone problem-solving skills and achieve your personal best in exams. They give you the information your teachers expect you to know in a handy and succinct format—without overwhelming you with unnecessary details. You get a complete overview of the subject, plus plenty of practice exercises to test your skills. Schaum's is ideal for self-study at your own pace, equipping you to understand and recall all important facts that you need to remember.

Acknowledgements

A number of experts have taken out time from their busy schedules to provide valuable feedback about the book. Our heartfelt gratitude goes out to those whose names are given in the next page.

Manish Manoria
TRUBA Institute of Engineering and Information Technology, Bhopal, Madhya Pradesh

L K Sharma
Alwar Institute of Engineering and Technology, Alwar, Rajasthan

Rajiv Pandey
Amity University, Lucknow, Uttar Pradesh

Dilkeshwar Pandey
Academy of Business and Engineering Sciences (ABES), Ghaziabad, Uttar Pradesh

Mayank Aggarwal
Gurukul Kangri Vishwavidyalaya, Haridwar, Uttarakhand

Sanjay Kumar Pandey
United College of Engineering and Research, Allahabad, Uttar Pradesh

H N Verma
Sachdeva Institute of Technology, Mathura, Uttar Pradesh

Shashank Dwivedi
United College of Engineering and Research, Allahabad, Uttar Pradesh

Gurpreet Kaur
Indraprastha University, New Delhi

Prashant Lakkadwala
Chameli Devi Institute of Technology and Management, Indore, Madhya Pradesh

Sameer Bhave
Indore Professional Studies Academy(IPSA), Indore, Madhya Pradesh

Bhupesh Deka
Infosys Technologies Limited, Bhubaneshwar, Orissa

S R Biradar
Sikkim Manipal Institute of Technology, East Sikkim

Mahua Banerjee
Xavier Institute of Social Service, Ranchi, Jharkhand

R S Prasad
Vishwakarma Institute of Information Technology, Pune, Maharashtra

Ilango Krishnamurthi
Sri Krishna College of Engineering and Technology, Coimbatore, Tamil Nadu

P Sampath
Bannari Amman Institute of Technology, Erode, Tamil Nadu

D Lakshmi
Dr N G P Institute of Technology, Coimbatore, Tamil Nadu

T Ramesh
National Institute of Technology(NIT), Warangal, Andhra Pradesh

G Shobha
R V College of Engineering, Bangalore, Karnataka

Jibi Abraham
MSR Institute of Technology, Bangalore, Karnataka

Feedback

Helpful suggestions and constructive criticism always go a long way in enhancing any endeavour. We request all readers to email us their valuable comments/views/feedback for the betterment of the book at tmh.csefeedback@gmail.com, mentioning the title and author name in the subject line. Also, please feel free to report any piracy of the book spotted by you.

Preface

The study of data structures is an essential part of virtually every undergraduate and graduate program in computer science. This text, in presenting the more essential material, may be used as a textbook for a formal course in data structures or as a supplement to almost all current standard texts.

The chapters are mainly organised in increasing degree of complexity. **Chapter 1** is an introduction and overview of the material, and **Chapter 2** presents the mathematical background and notation for the presentation and analysis of our algorithms. **Chapter 3**, on pattern matching, is independent and tangential to the text and hence may be postponed or omitted on a first reading. **Chapters 4 through 8** contain the core material in any course on data structures. Specifically, Chapter 4 treats arrays and records, **Chapter 5** is on linked lists, **Chapter 6** covers stacks and queues and includes recursion **Chapter 7** is on binary trees and **Chapter 8** is on graphs and their applications. Although sorting and searching is discussed throughout the text within the context of specific data structures (e.g., binary search with linear arrays, quicksort with stacks and queues and heapsort with binary trees), **Chapter 9**, the last chapter, presents additional sorting and searching algorithms such as merge-sort and hashing.

Algorithms are presented in a form which is machine and language independent. Moreover, they are written using mainly IF-THEN-ELSE and REPEAT-WHILE modules for flow of control, and using an indentation pattern for easier reading and understanding. Accordingly, each of our algorithms may be readily translated into almost any standard programming language.

Adopting a deliberately elementary approach to the subject matter with many examples and diagrams, this book should appeal to a wide audience, and is particularly suited as an effective self-study guide. Each chapter contains clear statements of definitions and principles together with illustrative and other descriptive material. This is followed by graded sets of solved and supplementary problems. The solved problems illustrate and amplify the material, and the supplementary problems furnish a complete review of the material in the chapter.

I wish to thank many friends and colleagues for invaluable suggestions and critical review of the manuscript. I also wish to express my gratitude to the staff of the McGraw-Hill Schaum's Outline Series, especially Jeffrey McCartney, for their helpful cooperation. Finally, I join many other authors in explicitly giving credit to Donald E. Knuth who wrote the first comprehensive treatment of the subject of data structures, which has certainly influenced the writing of this and many other texts on the subject.

SEYMOUR LIPSCHUTZ

Chapter 1

Introduction and Overview

1.1 INTRODUCTION

This chapter introduces the subject of data structures and presents an overview of the content of the text. Basic terminology and concepts will be defined and relevant examples provided. An overview of data organization and certain data structures will be covered along with a discussion of the different operations which are applied to these data structures. Last, we will introduce the notion of an algorithm and its complexity, and we will discuss the time-space tradeoff that may occur in choosing a particular algorithm and data structure for a given problem.

1.2 BASIC TERMINOLOGY; ELEMENTARY DATA ORGANIZATION

Data are simply values or sets of values. A *data item* refers to a single unit of values. Data items that are divided into subitems are called *group items*; those that are not are called *elementary items*. For example, an employee's name may be divided into three subitems—first name, middle initial and last name—but the social security number would normally be treated as a single item.

Collections of data are frequently organized into a hierarchy of *fields*, *records* and *files*. In order to make these terms more precise, we introduce some additional terminology.

An *entity* is something that has certain *attributes* or properties which may be assigned values. The values themselves may be either numeric or nonnumeric. For example, the following are possible attributes and their corresponding values for an entity, an employee of a given organization:

Attributes:	Name	Age	Sex	Social Security Number
Values:	ROHLAND, GAIL	34	F	134-24-5533

Entities with similar attributes (e.g., all the employees in an organization) form an *entity set*. Each attribute of an entity set has a *range* of values, the set of all possible values that could be assigned to the particular attribute.

The term "information" is sometimes used for data with given attributes, or, in other words, meaningful or processed data.

The way that data are organized into the hierarchy of fields, records and files reflects the relationship between attributes, entities and entity sets. That is, a *field* is a single elementary unit of information representing an attribute of an entity, a *record* is the collection of field values of a given entity and a *file* is the collection of records of the entities in a given entity set.

Each record in a file may contain many field items, but the value in a certain field may uniquely determine the record in the file. Such a field K is called a *primary key*, and the values k_1, k_2, ... in such a field are called *keys* or *key values.*

Example 1.1

(a) Suppose an automobile dealership maintains an inventory file where each record contains the following data:

Serial Number, Type, Year, Price, Accessories

The Serial Number field can serve as a primary key for the file, since each automobile has a unique serial number.

(b) Suppose an organization maintains a membership file where each record contains the following data:

Name, Address, Telephone Number, Dues Owed

Although there are four data items, Name and Address may be group items. Here the Name field is a primary key. Note that the Address and Telephone Number fields may not serve as primary keys, since some members may belong to the same family and have the same address and telephone number.

Records may also be classified according to length. A file can have fixed-length records or variable-length records. In *fixed-length records*, all the records contain the same data items with the same amount of space assigned to each data item. In *variable-length records*, file records may contain different lengths. For example, student records usually have variable lengths, since different students take different numbers of courses. Usually, variable-length records have a minimum and a maximum length.

The above organization of data into fields, records and files may not be complex enough to maintain and efficiently process certain collections of data. For this reason, data are also organized into more complex types of structures. The study of such data structures, which forms the subject matter of this text, includes the following three steps:

1. Logical or mathematical description of the structure
2. Implementation of the structure on a computer
3. Quantitative analysis of the structure, which includes determining the amount of memory needed to store the structure and the time required to process the structure.

The next section introduces us to some of these data structures.

Remark: The second and third of the steps in the study of data structures depend on whether the data are stored (a) in the main (primary) memory of the computer or (b) in a secondary (external) storage unit. This text will mainly cover the first case. This means that, given the address of a memory location, the time required to access the content of the memory cell does not depend on the particular cell or upon the previous cell accessed. The second case, called *file management* or *data base management*, is a subject unto itself and lies beyond the scope of this text.

1.3 DATA STRUCTURES

Data may be organized in many different ways; the logical or mathematical model of a particular organization of data is called a *data structure*. The choice of a particular data model depends on two considerations. First, it must be rich enough in structure to mirror the actual relationships of the data in the real world. On the other hand, the structure should be simple enough that one can effectively process the data when necessary. This section will introduce us to some of the data structures which will be discussed in detail later in the text.

Classification of Data Structures

Data structures are generally classified into primitive and non-primitive data structures. Basic data types such as integer, real, character and boolean are known as primitive data structures. These data types consist of characters that cannot be divided, and hence they are also called simple data types.

The simplest example of non-primitive data structure is the processing of complex numbers. Very few computers are capable of doing arithmetic on complex numbers. Linked-lists, stacks, queues, trees and graphs are examples of non-primitive data structures. Figure 1.1 shows the classification of data structures.

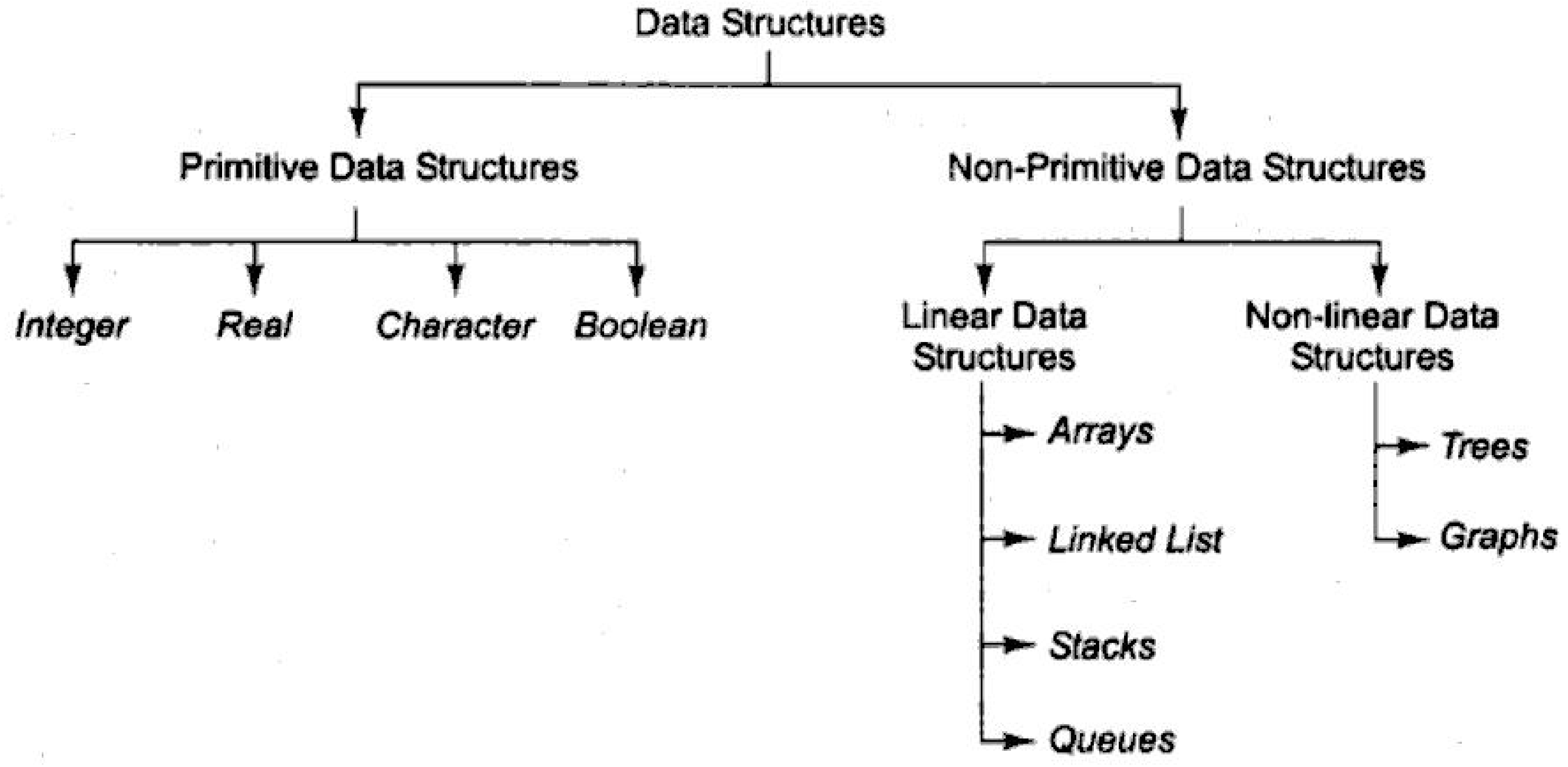


Fig. 1.1 Classification of Data Structures

Based on the structure and arrangement of data, non-primitive data structures are further classified into linear and non-linear.

A data structure is said to be *linear* if its elements form a sequence or a linear list. In linear data structures, the data is arranged in a linear fashion although the way they are stored in memory need not be sequential. Arrays, linked lists, stacks and queues are examples of linear data structures.

Conversely, a data structure is said to be *non-linear* if the data is not arranged in sequence. The insertion and deletion of data is therefore not possible in a linear fashion. Trees and graphs are examples of non-linear data structures.

Arrays

The simplest type of data structure is a *linear* (or *one-dimensional*) *array*. By a *linear array*, we mean a list of a finite number n of similar data elements referenced respectively by a set of n consecutive numbers, usually 1, 2, 3, ..., n. If we choose the name A for the array, then the elements of A are denoted by subscript notation

$$a_1, a_2, a_3, \ldots, a_n$$

or by the parenthesis notation

$$\text{A(1), A(2), A(3), ..., A(N)}$$

or by the bracket notation

$$\text{A[1], A[2], A[3], ..., A[N]}$$

Regardless of the notation, the number K in A[K] is called a *subscript* and A[K] is called a *subscripted variable.*

Remark: The parentheses notation and the bracket notation are frequently used when the array name consists of more than one letter or when the array name appears in an algorithm. When using this notation we will use ordinary uppercase letters for the name and subscripts as indicated above by the A and N. Otherwise, we may use the usual subscript notation of italics for the name and subscripts and lowercase letters for the subscripts as indicated above by the a and n. The former notation follows the practice of computer-oriented texts whereas the latter notation follows the practice of mathematics in print.

Example 1.2

A linear array STUDENT consisting of the names of six students is pictured in Fig. 1.2. Here STUDENT[1] denotes John Brown, STUDENT[2] denotes Sandra Gold, and so on.

Linear arrays are called one-dimensional arrays because each element in such an array is referenced by one subscript. A *two-dimensional array* is a collection of similar data elements where each element is referenced by two subscripts. (Such arrays are called *matrices* in mathematics, and *tables* in business applications.) Multidimensional arrays are defined analogously. Arrays will be covered in detail in Chapter 4.

	STUDENT
1	John Brown
2	Sandra Gold
3	Tom Jones
4	June Kelly
5	Mary Reed
6	Alan Smith

Fig. 1.2

Example 1.3

A chain of 28 stores, each store having 4 departments, may list its weekly sales (to the nearest dollar) as in Fig. 1.3. Such data can be stored in the computer using a two-dimensional array in which the first subscript denotes the store and the second subscript the department. If SALES is the name given to the array, then

SALES[1, 1] = 2872, SALES[1, 2] = 805, SALES[1, 3] = 3211, ..., SALES[28, 4] = 982

Store \ Dept.	1	2	3	4
1	2872	805	3211	1560
2	2196	1223	2525	1744
3	3257	1017	3686	1951
...	...	...	...	...
28	2618	931	2333	982

Fig. 1.3

The size of this array is denoted by 28 × 4 (read 28 by 4), since it contains 28 *rows* (the horizontal lines of numbers) and 4 *columns* (the vertical lines of numbers).

Linked Lists

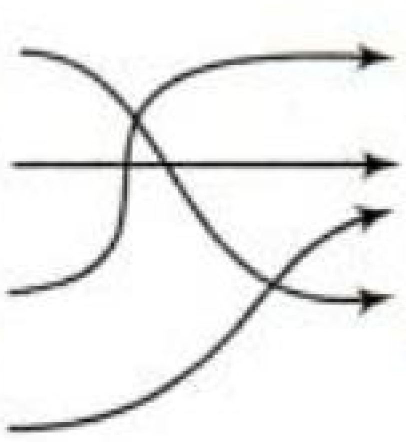
Linked lists will be introduced by means of an example. Suppose a brokerage firm maintains a file where each record contains a customer's name and his or her salesperson, and suppose the file contains the data appearing in Fig. 1.4. Clearly the file could be stored in the computer by such a table, i.e., by two columns of nine names. However, this may not be the most useful way to store the data, as the following discussion shows.

	Customer	Salesperson
1	Adams	Smith
2	Brown	Ray
3	Clark	Jones
4	Drew	Ray
5	Evans	Smith
6	Farmer	Jones
7	Geller	Ray
8	Hill	Smith
9	Infeld	Ray

Fig. 1.4

Another way of storing the data in Fig. 1.4 is to have a separate array for the salespeople and an entry (called a *pointer*) in the customer file which gives the location of each customer's salesperson. This is done in Fig. 1.5, where some of the pointers are pictured by an arrow from the location of the pointer to the location of the corresponding salesperson. Practically speaking, an integer used as a pointer requires less space than a name; hence this representation saves space, especially if there are hundreds of customers for each salesperson.

	Customer	Pointer
1	Adams	3
2	Brown	2
3	Clark	1
4	Drew	2
5	Evans	3
6	Farmer	1
7	Geller	2
8	Hill	3
9	Infeld	2



Salesperson	
Jones	1
Ray	2
Smith	3

Fig. 1.5

Suppose the firm wants the list of customers for a given salesperson. Using the data representation in Fig. 1.5, the firm would have to search through the entire customer file. One way to simplify such a search is to have the arrows in Fig. 1.5 point the other way; each salesperson would now have a set of pointers giving the positions of his or her customers, as in Fig. 1.6. The main disadvantage of this representation is that each salesperson may have many pointers and the set of pointers will change as customers are added and deleted.

	Salesperson	Pointer
1	Jones	3, 6
2	Ray	2, 4, 7, 9
3	Smith	1, 5, 8

Fig. 1.6

Another very popular way to store the type of data in Fig. 1.4 is shown in Fig. 1.7. Here each salesperson has one pointer which points to his or her first customer, whose pointer in turn points to the second customer, and so on, with the salesperson's last customer indicated by a 0. This is pictured with arrows in Fig. 1.7 for the salesperson Ray. Using this representation one can easily obtain the entire list of customers for a given salesperson and, as we will see in Chapter 5, one can easily insert and delete customers.

The representation of the data in Fig. 1.7 is an example of linked lists. Although the terms "pointer" and "link" are usually used synonymously, we will try to use the term "pointer" when an element in one list points to an element in a different list, and to reserve the term "link" for the case when an element in a list points to an element in that same list.

	Customer	Link
1	Adams	5
2	Brown	4
3	Clark	6
4	Drew	7
5	Evans	8
6	Farmer	0
7	Geller	9
8	Hill	0
9	Infeld	0

Salesperson	Pointer	
Jones	3	1
Ray	2	2
Smith	1	3

Fig. 1.7

Trees

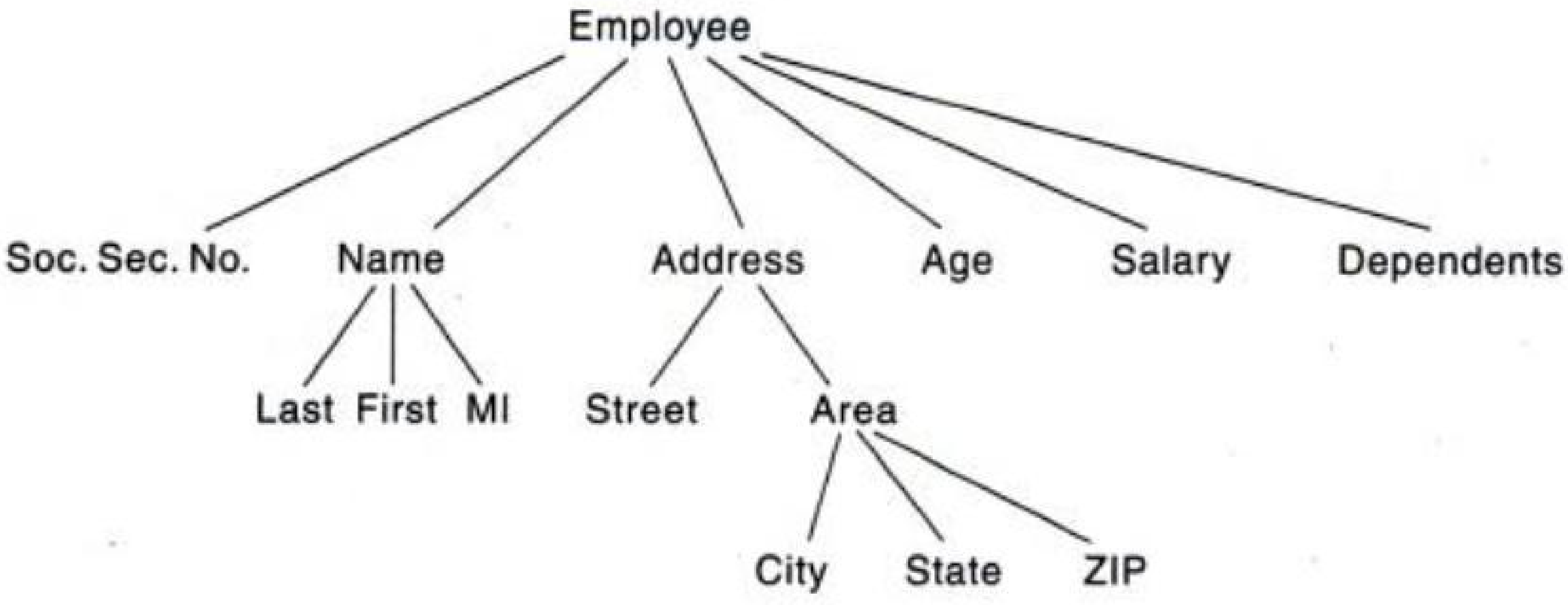
Data frequently contain a hierarchical relationship between various elements. The data structure which reflects this relationship is called a *rooted tree graph* or, simply, a *tree*. Trees will be defined and discussed in detail in Chapter 7. Here we indicate some of their basic properties by means of two examples.

Example 1.4 Record Structure

Although a file may be maintained by means of one or more arrays, a record, where one indicates both the group items and the elementary items, can best be described by means of a tree structure. For example, an employee personnel record may contain the following data items:

Social Security Number, Name, Address, Age, Salary, Dependents

However, Name may be a group item with the subitems Last, First and MI (middle initial). Also, Address may be a group item with the subitems Street address and Area address, where Area itself may be a group item having subitems City, State and ZIP code number. This hierarchical structure is pictured in Fig. 1.8(a). Another way of picturing such a tree structure is in terms of levels, as in Fig. 1.8(b).



(a)

```
01 Employee
      02 Social Security Number
      02 Name
            03 Last
            03 First
            03 Middle Initial
      02 Address
            03 Street
            03 Area
                  04 City
                  04 State
                  04 ZIP
      02 Age
      02 Salary
      02 Dependents
```

(b)

Fig. 1.8

Example 1.5 Algebraic Expressions

Consider the algebraic expression

$$(2x + y)(a - 7b)^3$$

Using a vertical arrow (↑) for exponentiation and an asterisk (*) for multiplication, we can represent tile expression by the tree in Fig. 1.9. Observe that the order in which the operations will be performed is reflected in the diagram: the exponentiation must take place after the subtraction, and the multiplication at the top of the tree must be executed last.

There are data structures other than arrays, linked lists and trees which we shall study. Some of these structures are briefly described below.

(a) *Stack*: A stack, also called a last-in first-out (LIFO) system, is a linear list in which insertions and deletions can take place only at one end, called the *top*. This structure is similar in its operation to a stack of dishes on a spring system, as pictured in Fig. 1.10(a). Note that new dishes are inserted only at the top of the stack and dishes can be deleted only from the top of the stack.

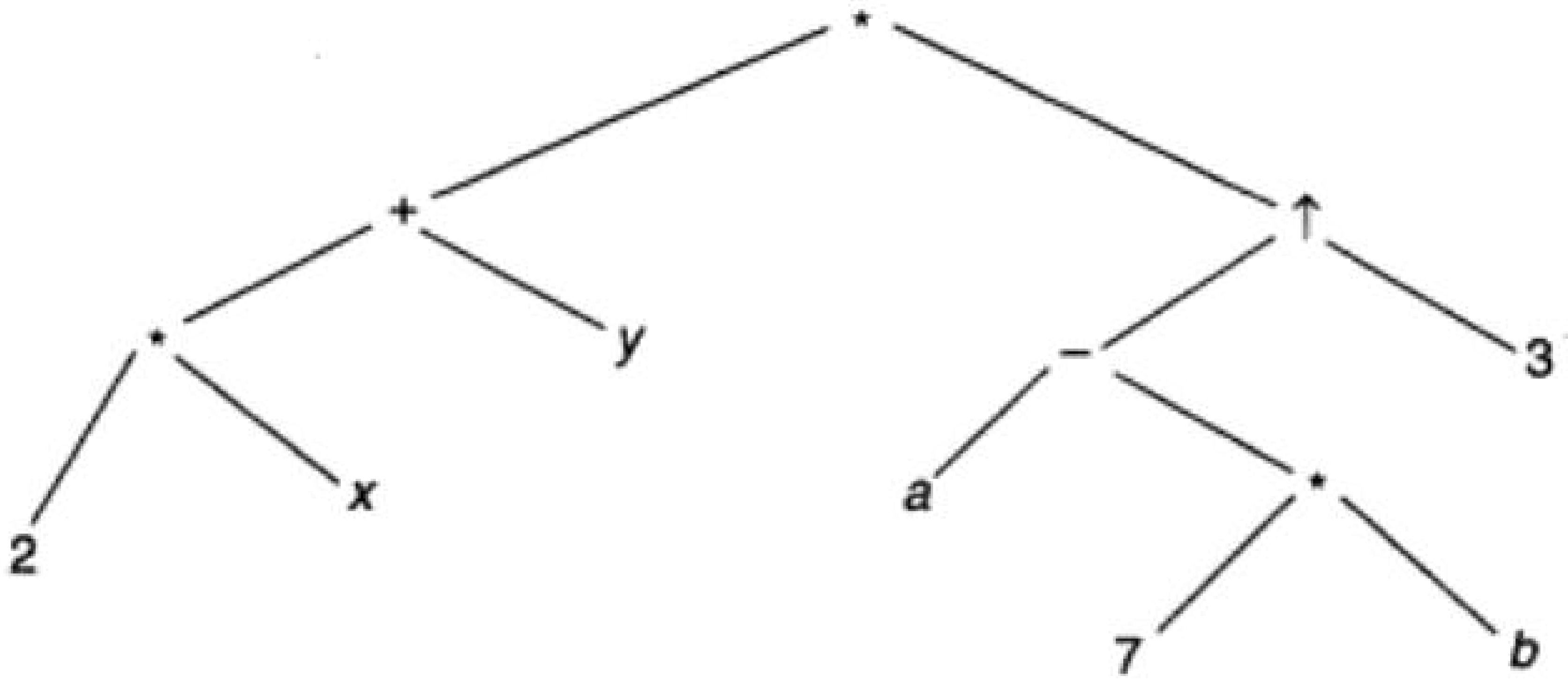
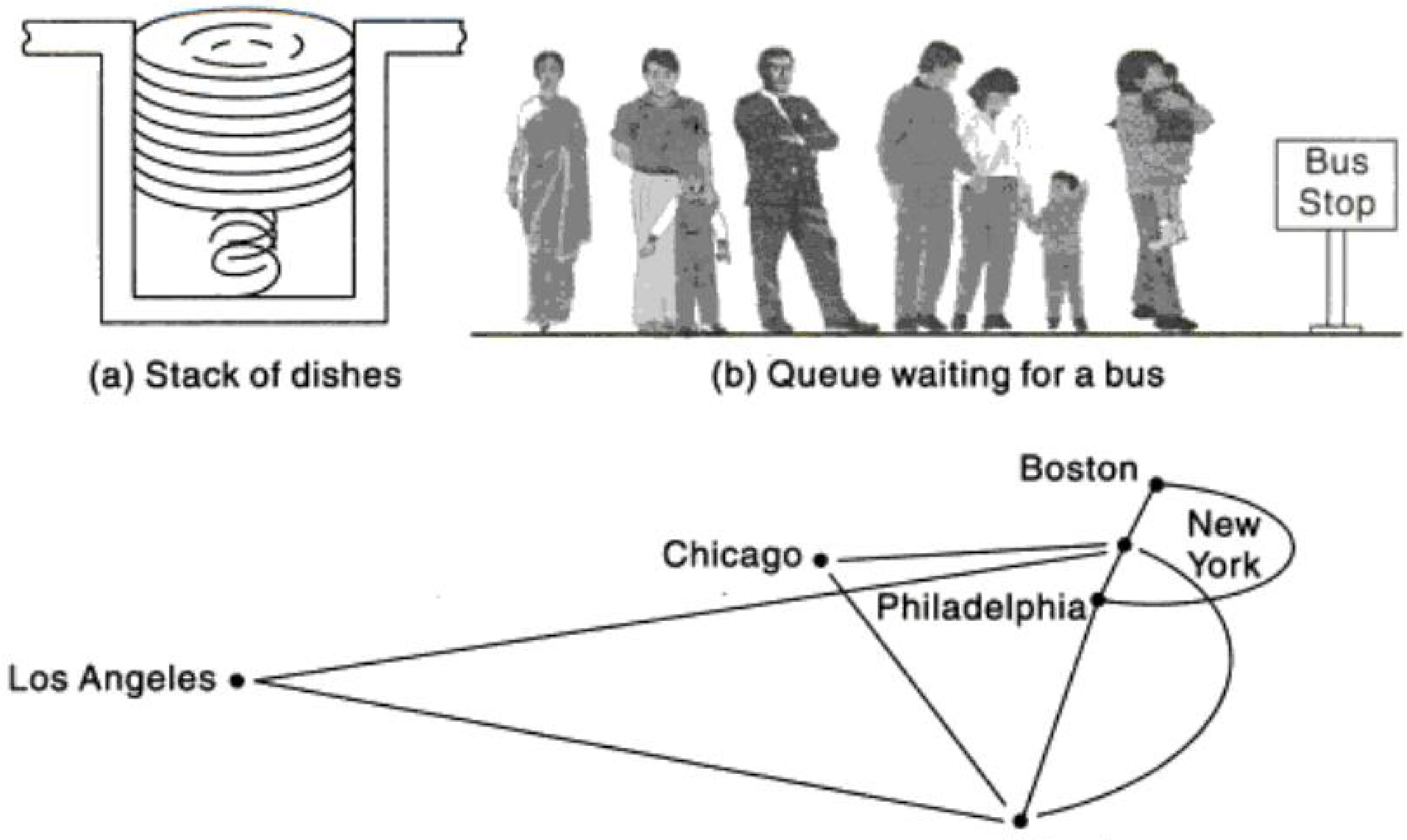


Fig. 1.9

(b) *Queue*: A queue, also called a first-in first-out (FIFO) system, is a linear list in which deletions can take place only at one end of the list, the "front" of the list, and insertions can take place only at the other end of the list, the "rear" of the list. This structure operates in much the same way as a line of people waiting at a bus stop, as pictured in Fig. 1.10(b): the first person in line is the first person to board the bus. Another analogy is with automobiles waiting to pass through an intersection—the first car in line is the first car through.

(c) *Graph*: Data sometimes contain a relationship between pairs of elements which is not necessarily hierarchical in nature. For example, suppose an airline flies only between the cities connected by lines in Fig. 1.10(c). The data structure which reflects this type of relationship is called a *graph*. Graphs will be formally defined and studied in Chapter 8.



(a) Stack of dishes

(b) Queue waiting for a bus

(c) Airline flights

Fig. 1.10

Remark: Many different names are used for the elements of a data structure. Some commonly used names are "data element," "data item," "item aggregate," "record," "node" and "data object." The particular name that is used depends on the type of data structure, the context in which the structure is used and the people using the name. Our preference shall be the term "data element," but we will use the term "record" when discussing files and the term "node" when discussing linked lists, trees and graphs.

1.4 DATA STRUCTURE OPERATIONS

The data appearing in our data structures are processed by means of certain operations. In fact, the particular data structure that one chooses for a given situation depends largely on the frequency with which specific operations are performed. This section introduces the reader to some of the most frequently used of these operations.

The following four operations play a major role in this text:

1. *Traversing:* Accessing each record exactly once so that certain items in the record may be processed. (This accessing and processing is sometimes called "visiting" the record.)
2. *Searching:* Finding the location of the record with a given key value, or finding the locations of all records which satisfy one or more conditions.
3. *Inserting:* Adding a new record to the structure.
4. *Deleting:* Removing a record from the structure.

Sometimes two or more of the operations may be used in a given situation; e.g., we may want to delete the record with a given key, which may mean we first need to search for the location of the record.

The following two operations, which are used in special situations, will also be considered:

1. *Sorting:* Arranging the records in some logical order (e.g., alphabetically according to some NAME key, or in numerical order according to some NUMBER key, such as social security number or account number)
2. *Merging:* Combining the records in two different sorted files into a single sorted file

Other operations, e.g. copying and concatenation, will be discussed later in the text.

Example 1.6

An organization contains a membership file in which each record contains the following data for a given member:

Name, Address, Telephone Number, Age, Sex

(a) Suppose the organization wants to announce a meeting through a mailing. Then one would traverse the file to obtain Name and Address for each member.

(b) Suppose one wants to find the names of all members living in a certain area. Again one would traverse the file to obtain the data.

(c) Suppose one wants to obtain Address for a given Name. Then one would search the file for the record containing Name.

(d) Suppose a new person joins the organization. Then one would insert his or her record into the file.

(e) Suppose a member dies. Then one would delete his or her record from the file.

(f) Suppose a member has moved and has a new address and telephone number. Given the name of the member, one would first need to search for the record in the file. Then one would perform the "update"—i.e., change items in the record with the new data.

(g) Suppose one wants to find the number of members 65 or older. Again one would traverse the file, counting such members.

1.5 ABSTRACT DATA TYPES (ADT)

An abstract data type (ADT) refers to a set of data values and associated operations that are specified accurately, independent of any particular implementation. With an ADT, we know what a specific data type can do, but how it actually does it is hidden. In broader terms, the ADT consists of a set of definitions that allow us to use the functions while hiding the implementation.

The properties of an abstract data type are emphasized through the following examples.

Example 1.7 List Representation

Consider a list *L* consisting of data items—1, 2, 3, 4, 5, 6, 7, 8, 9 as shown in Fig. 1.11(a). We can use any of four data structures to support *L*—a linear list, a matrix, a tree, or a graph, as given in Fig. 1.11(b), (c) and (d) respectively.

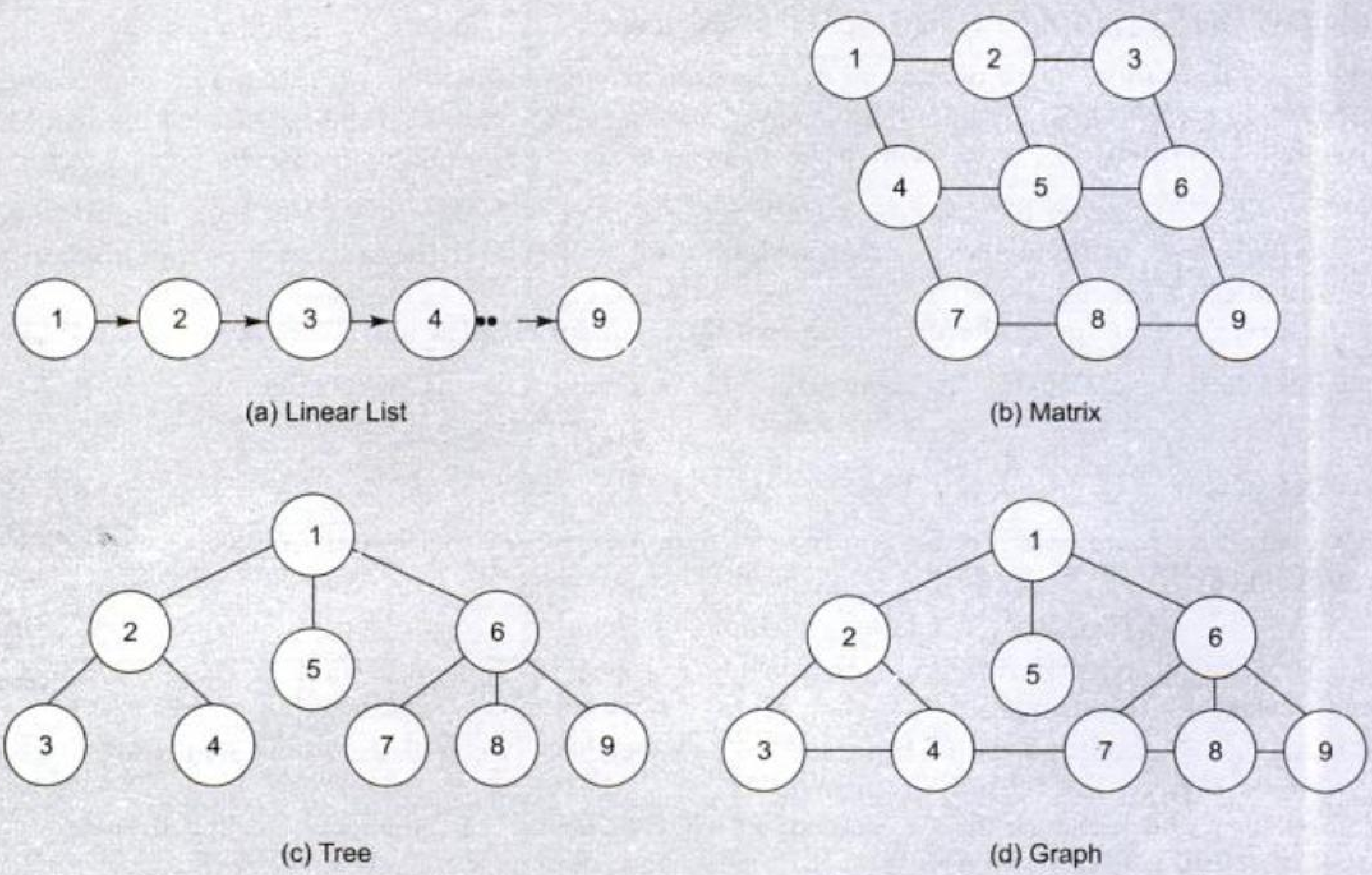


Fig. 1.11 Data structures which support a list

Assume that we place the list on an ADT. The users should not be aware of the structure that we use, i.e., whether it is a tree, or graph or something else. As long as they are able to insert and retrieve data, it does not make a difference as to how we store the data.

Example 1.8

A shop maintains the list of customers, sales assistants and the average transactions on a day as given in Fig. 1.12.

Suppose we need to write a program, which will help determine the number of sales assistants required to serve customers efficiently. Here, we will need to simulate the waiting line in the shop. This analysis will require the simulation of a queue. However, queues are not generally available in programming languages. Therefore, even if the queue type is available, we need some basic queue operations such as enqueuing and dequeuing, which are basically insertion and deletion operations, for the simulation.

Here is what we can do in this situation:

1. Write a program that simulates the queue, or
2. Write a queue ADT that can solve any queue problem.

If we choose the second option, we still need to write a program to simulate the shop application. However, doing that will actually be simpler and faster because we can concentrate on the application rather than the queue.

	Customers
1	Customers
2	Customers
3	Customers

	Sales Assistant
1	Ray
2	Reed
3	Kelly

Fig. 1.12

An abstract data type can thus be further defined as a data declaration packaged together with the operations that are meaningful for the data type. In other words, we encapsulate the data and the operations on the data, and then we hide them from the user.

The user need not know the data structure to use the ADT. Considering Example 1.8, the application program should have no knowledge of the data structure. All references to and manipulation of the data in the queue must be handled through defined interfaces in the structure. Allowing the application program to directly reference the data structure is a common fault in many implementations. This prevents the ADT from being fully portable to other applications.

Abstract Data Type Model

A representation of the ADT model is shown in Fig. 1.13. Notice that there are two different parts of the ADT model—functions (public and private) and data structures. Both are contained within the ADT model itself, and do not come within the scope of the application program. On the other hand, data structures are available to all of the ADTs functions as required, and a function may call on any other function to accomplish its task. This means that data structures and functions are within the scope of each other.

Data are entered, accessed, modified and deleted through the external

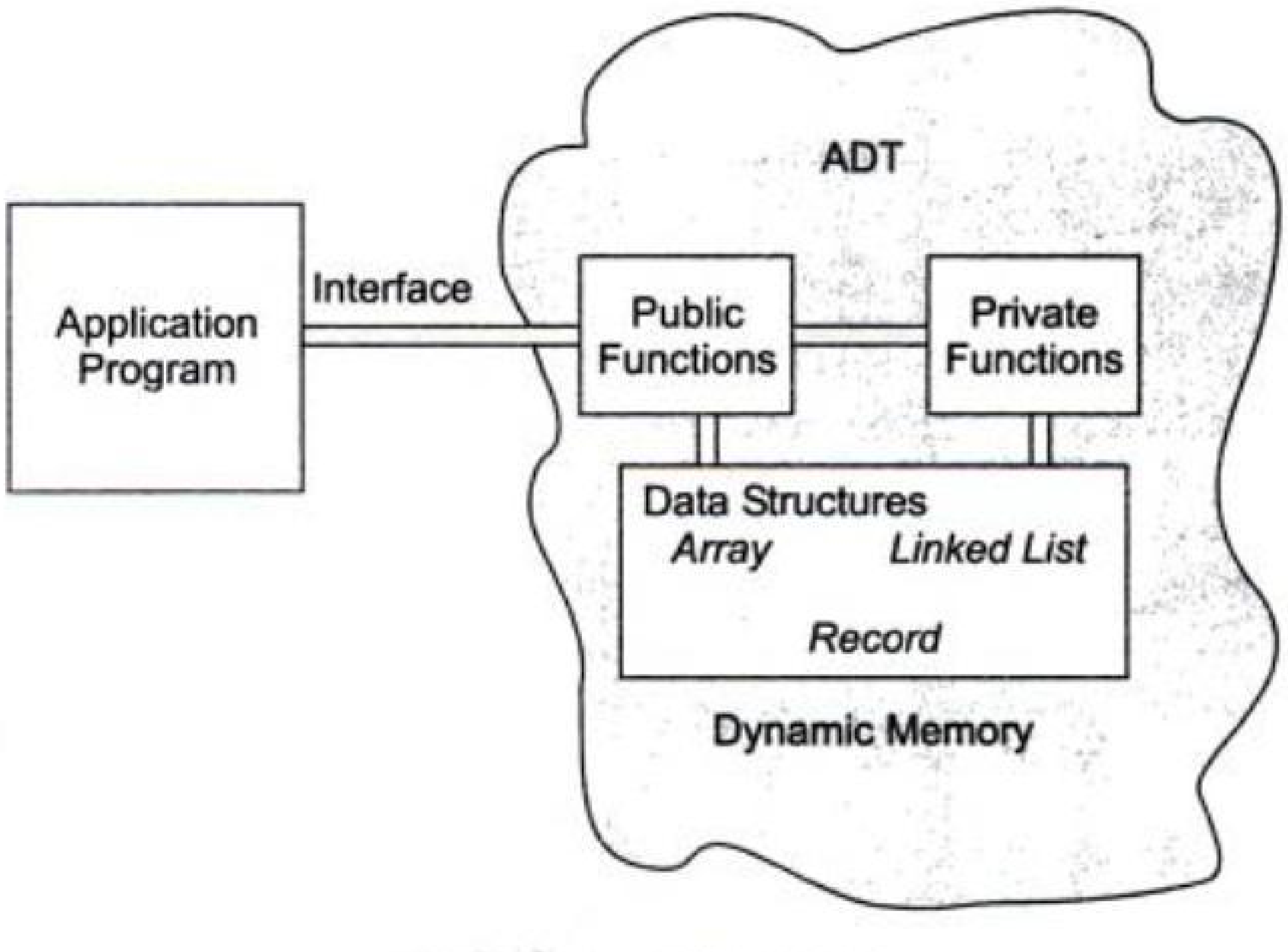


Fig. 1.13 ADT Model

application programming interface. This interface can only access the public functions. For each ADT operation, there is an algorithm that performs its specific task. The operation name and parameters are available to the application, and they provide the only interface to the application.

When a list is controlled entirely by the program, it is implemented using simple structures. Note that it is not enough if we just encapsulate the structure in an ADT, it is also necessary for multiple versions of the structure to coexist. Therefore, we must hide the implementation from the user, while being able to store different data at the same time.

Two basic structures, namely array and linked list, can be used to implement an ADT list.

1.6 ALGORITHMS: COMPLEXITY, TIME-SPACE TRADEOFF

An algorithm is a well-defined list of steps for solving a particular problem. One major purpose of this text is to develop efficient algorithms for the processing of our data. The time and space it uses are two major measures of the efficiency of an algorithm. The complexity of an algorithm is the function which gives the running time and/or space in terms of the input size. (The notion of complexity will be treated in Chapter 2.)

Each of our algorithms will involve a particular data structure. Accordingly, we may not always be able to use the most efficient algorithm, since the choice of data structure depends on many things, including the type of data and the frequency with which various data operations are applied. Sometimes the choice of data structure involves a time-space tradeoff: by increasing the amount of space for storing the data, one may be able to reduce the time needed for processing the data, or vice versa. We illustrate these ideas with two examples.

Searching Algorithms

Consider a membership file, as in Example 1.6, in which each record contains, among other data, the name and telephone number of its member. Suppose we are given the name of a member and we want to find his or her telephone number. One way to do this is to linearly search through the file, i.e., to apply the following algorithm:

Linear Search

Search each record of the file, one at a time, until finding the given Name and hence the corresponding telephone number.

First of all, it is clear that the time required to execute the algorithm is proportional to the number of comparisons. Also, assuming that each name in the file is equally likely to be picked, it is intuitively clear that the average number of comparisons for a file with n records is equal to $n/2$; that is, the complexity of the linear search algorithm is given by $C(n) = n/2$.

The above algorithm would be impossible in practice if we were searching through a list consisting of thousands of names, as in a telephone book. However, if the names are sorted alphabetically, as in telephone books, then we can use an efficient algorithm called binary search. This algorithm is discussed in detail in Chapter 4, but we briefly describe its general idea below.

Binary Search

Compare the given Name with the name in the middle of the list; this tells which half of the list contains Name. Then compare Name with the name in the middle of the correct half to determine which quarter of the list contains Name. Continue the process until finding Name in the list.

One can show that the complexity of the binary search algorithm is given by

$$C(n) = \log_2 n$$

Thus, for example, one will not require more than 15 comparisons to find a given Name in a list containing 25 000 names.

Although the binary search algorithm is a very efficient algorithm, it has some major drawbacks. Specifically, the algorithm assumes that one has direct access to the middle name in the list or a sublist. This means that the list must be stored in some type of array. Unfortunately, inserting an element in an array requires elements to be moved down the list, and deleting an element from an array requires element to be moved up the list.

The telephone company solves the above problem by printing a new directory every year while keeping a separate temporary file for new telephone customers. That is, the telephone company updates its files every year. On the other hand, a bank may want to insert a new customer in its file almost instantaneously. Accordingly, a linearly sorted list may not be the best data structure for a bank.

An Example of Time-Space Tradeoff

Suppose a file of records contains names, social security numbers and much additional information among its fields. Sorting the file alphabetically and running a binary search is a very efficient way to find the record for a given name. On the other hand, suppose we are given only the social security number of the person. Then we would have to do a linear search for the record, which is extremely time-consuming for a very large number of records. How can we solve such a problem? One way is to have another file which is sorted numerically according to social security number. This, however, would double the space required for storing the data. Another way, pictured in Fig. 1.14, is to have the main file sorted numerically by social security number and to have an auxiliary array with only two columns, the first column containing an alphabetized list of the names and the second column containing pointers which give the locations of the corresponding records in the main file. This is one way of solving the problem that is used frequently, since the additional space, containing only two columns, is minimal for the amount of extra information it provides.

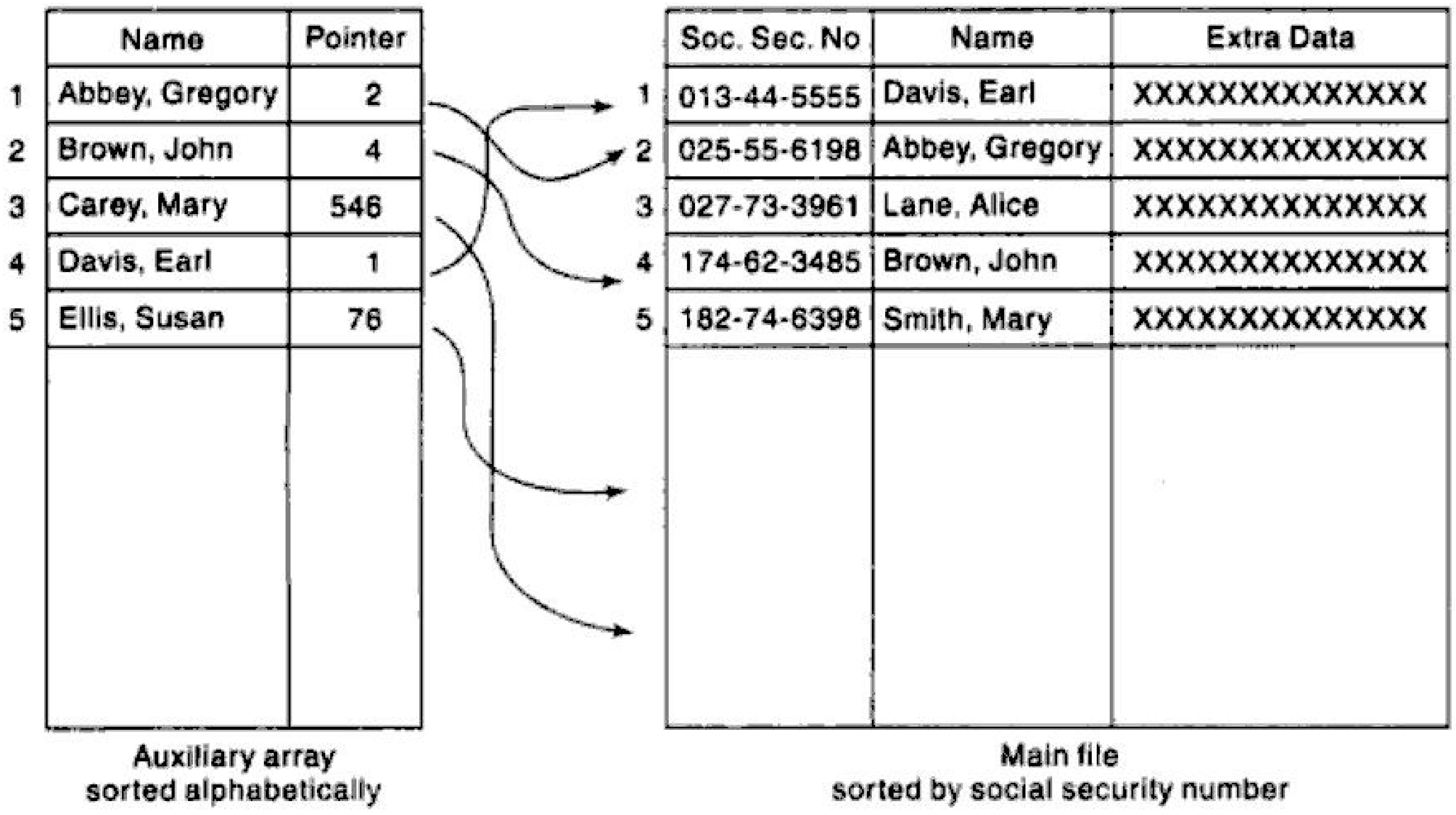


Fig. 1.14

Remark: Suppose a file is sorted numerically by social security number. As new records are inserted into the file, data must be constantly moved to new locations in order to maintain the sorted order. One simple way to minimize the movement of data is to have the social security number serve as the address of each record. Not only would there be no movement of data when records are inserted, but there would be instant access to any record. However, this method of storing data would require one billion (10^9) memory locations for only hundreds or possibly thousands of records. Clearly, this tradeoff of space for time is not worth the expense. An alternative method is to define a function H from the set K of key values—social security numbers—into the set L of addresses of memory cells. Such a function H is called a *hashing function.* Hashing functions and their properties will be covered in Chapter 9.

SOLVED PROBLEMS

Basic Terminology

1.1 A professor keeps a class list containing the following data for each student:

Name, Major, Student Number, Test Scores, Final Grade

(a) State the entities, attributes and entity set of the list.
(b) Describe the field values, records and file.
(c) Which attributes can serve as primary keys for the list?

(a) Each student is an entity, and the collection of students is the entity set. The properties, name, major, and so on, of the students are the attributes.
(b) The field values are the values assigned to the attributes, i.e., the actual names, test scores, and so on. The field values for each student constitute a record, and the collection of all the student records is the file.
(c) Either Name or Student Number can serve as a primary key, since each uniquely determines the student's record. Normally the professor uses Name as the primary key, but the registrar may use Student Number.

1.2 A hospital maintains a patient file in which each record contains the following data:

Name, Admission Date, Social Security Number, Room, Bed Number, Doctor

(a) Which items can serve as primary keys?
(b) Which pair of items can serve as a primary key?
(c) Which items can be group items?

(a) Name and Social Security Number can serve as primary keys. (We assume that no two patients have the same name.)
(b) Room and Bed Number in combination also uniquely determine a given patient.
(c) Name, Admission Date and Doctor may be group items.

1.3 Which of the following data items may lead to variable-length records when included as items in the record: (a) age, (b) sex, (c) name of spouse, (d) names of children, (e) education, (f) previous employers?

Since (d) and (f) may contain a few or many items, they may lead to variable-length records. Also, (e) may contain many items, unless it asks only for the highest level obtained.

1.4 Data base systems will be only briefly covered in this text. Why?

"Data base systems" refers to data stored in the secondary memory of the computer. The implementation and analysis of data structures in the secondary memory are very different from those in the main memory of the computer. This text is primarily concerned with data structures in main memory, not secondary memory.

Data Structures and Operations

1.5 Give a brief description of (a) traversing, (b) sorting and (c) searching.

(a) Accessing and processing each record exactly once
(b) Arranging the data in some given order
(c) Finding the location of the record with a given key or keys

1.6 Give a brief description of (a) inserting and (b) deleting.

(a) Adding a new record to the data structure, usually keeping a particular ordering
(b) Removing a particular record from the data structure

1.7 Consider the linear array NAME in Fig. 1.15, which is sorted alphabetically.
(a) Find NAME[2], NAME[4] and NAME[7].
(b) Suppose Davis is to be inserted into the array. How many names must be moved to new locations.
(c) Suppose Gupta is to be deleted from the array. How many names must be moved to new locations?

(a) Here NAME[K] is the *k*th name in the list. Hence,

NAME[2] = Clark, NAME[4] = Gupta, NAME[7] = Pace

(b) Since Davis will be assigned to NAME[3], the names Evans through Smith must be moved. Hence six names are moved.
(c) The names Jones through Smith must be moved up the array. Hence four names must be moved.

	NAME
1	Adam
2	Clark
3	Evans
4	Gupta
5	Jones
6	Lane
7	Pace
8	Smith

Fig. 1.15

1.8 Consider the linear array NAME in Fig. 1.16. The values of FIRST and LINK[K] in the figure determine a linear ordering of the names as follows. FIRST gives the location of the first name in the list, and LINK[K] gives the location of the name following NAME[K], with 0 denoting the end of the list. Find the linear ordering of the names.

FIRST: 5 → 5

	NAME	LINK
1	Rogers	7
2	Clark	8
3		
4	Hansen	10
5	Brooks	2
6	Pitt	1
7	Walker	0
8	Fisher	4
10	Leary	6

Fig. 1.16

The ordering is obtained as follows:
FIRST = 5, so the first name in the list is NAME[5], which is Brooks.
LINK[5] = 2, so the next name is NAME[2], which is Clark.
LINK[2] = 8, so the next name is NAME[8], which is Fisher.
LINK[8] = 4, so the next name is NAME[4], which is Hansen.
LINK[4] = 10, so the next name is NAME[10], which is Leary.
LINK[10] = 6, so the next name is NAME[6], which is Pitt.
LINK[6] = 1, so the next name is NAME[1], which is Rogers.
LINK[1] = 7, so the next name is NAME[7], which is Walker.
LINK[7] = 0, which indicates the end of the list.

Thus the linear ordering of the names is Brooks, Clark, Fisher, Hansen, Leary, Pitt, Rogers, Walker. Note that this is the alphabetical ordering of the names.

1.9 Consider the algebraic expression $(7x + y)(5a - b)^3$. (a) Draw the corresponding tree diagram as in Example 1.5. (b) Find the scope of the exponential operation. (The scope of a node v in a tree is the subtree consisting of v and the nodes following v.)

(a) Use a vertical arrow (↑) for exponentiation and an asterisk (*) for multiplication to obtain the tree in Fig. 1.17.

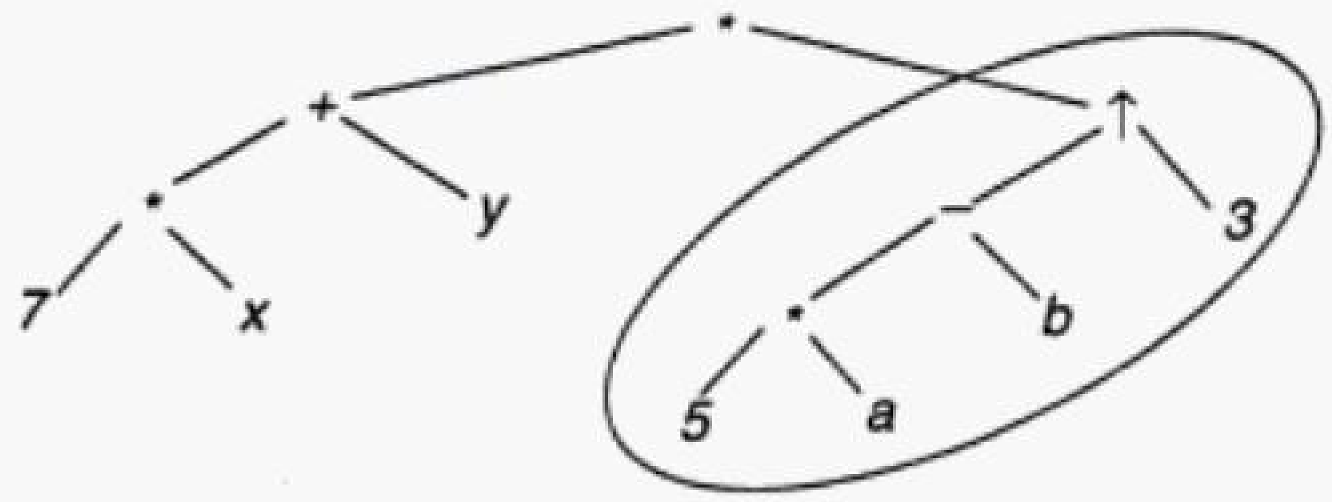


Fig. 1.17

(b) The scope of the exponentiation operation ↑ is the subtree circled in the diagram. It corresponds to the expression $(5a - b)^3$.

1.10 The following is a tree structure given by means of level numbers as discussed in Example 1.4:

01 Employee 02 Name 02 Number 02 Hours 03 Regular 03 Overtime 02 Rate

Draw the corresponding tree diagram.

The tree diagram appears in Fig. 1.18. Here each node v is the successor of the node which precedes v and has a lower level number than v.

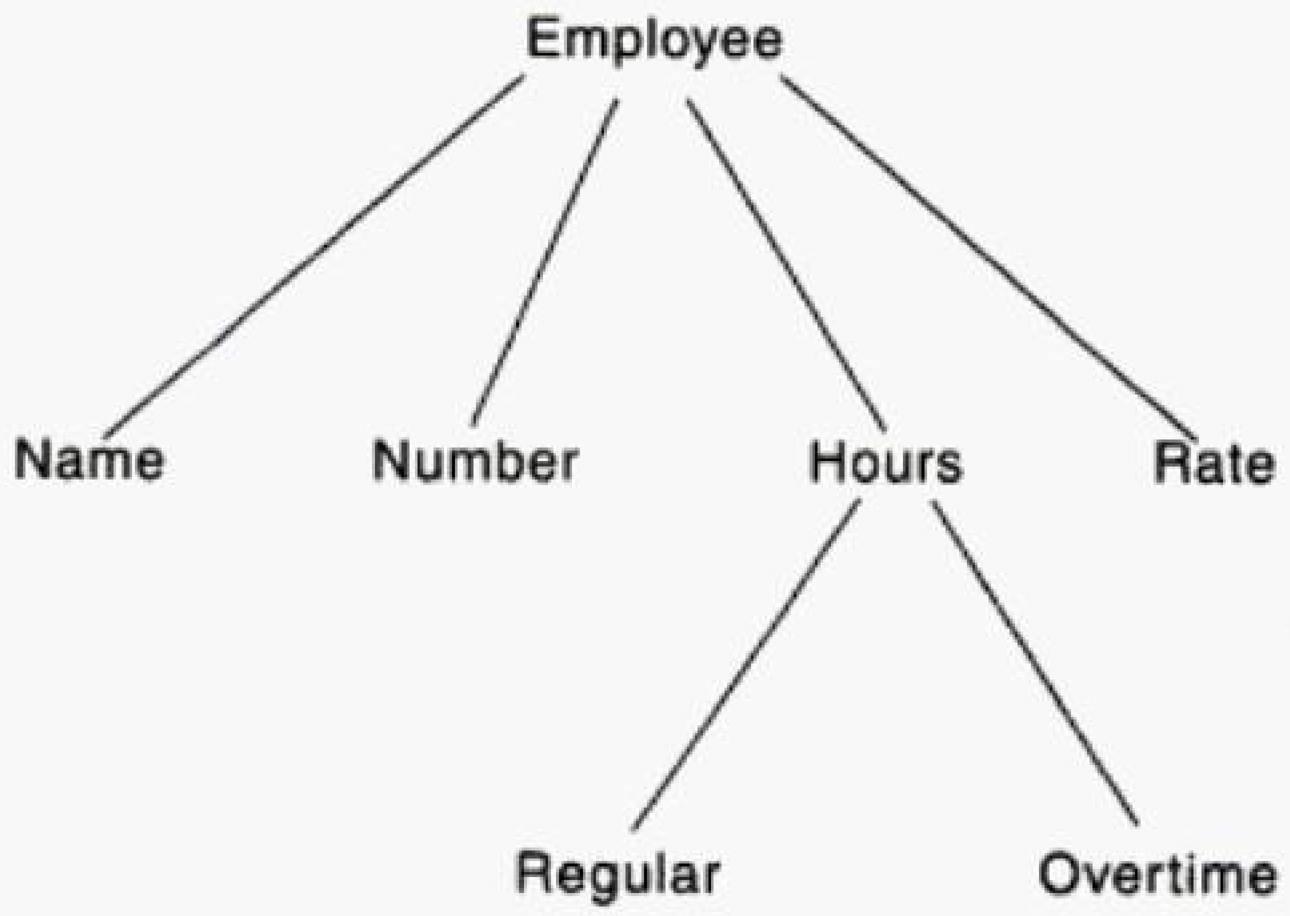


Fig. 1.18

1.11 Discuss whether a stack or a queue is the appropriate structure for determining the order in which elements are processed in each of the following situations.

(a) Batch computer programs are submitted to the computer center.
(b) Program A calls subprogram B which calls subprogram C, and so on.
(c) Employees have a contract which calls for a seniority system for hiring and firing.

(a) Queue. Excluding priority cases, programs are executed on a first come, first served basis.
(b) Stack. The last subprogram is executed first, and its results are transferred to the next-to-last program, which is then executed, and so on, until the original calling program is executed.
(c) Stack. In a seniority system, the last to be hired is the first to be discharged.

1.12 The daily flights of an airline company appear in Fig. 1.19. CITY lists the cities, and ORIG[K] and DEST[K] denote the cities of origin and destination, respectively, of the flight NUMBER[K]. Draw the corresponding directed graph of the data. (The graph is directed because the flight numbers represent flights from one city to another but not returning.)

The nodes of the graph are the five cities. Draw an arrow from city A to city B if there is a flight from A to B, and label the arrow with the flight number. The directed graph appears in Fig. 1.20.

	CITY
1	Atlanta
2	Boston
3	Chicago
4	Miami
5	Philadelphia

(a)

	NUMBER	ORIG	DEST
1	701	2	3
2	702	3	2
3	705	5	3
4	708	3	4
5	711	2	5
6	712	5	2
7	713	5	1
8	715	1	4
9	717	5	4
10	718	4	5

(b)

Fig. 1.19

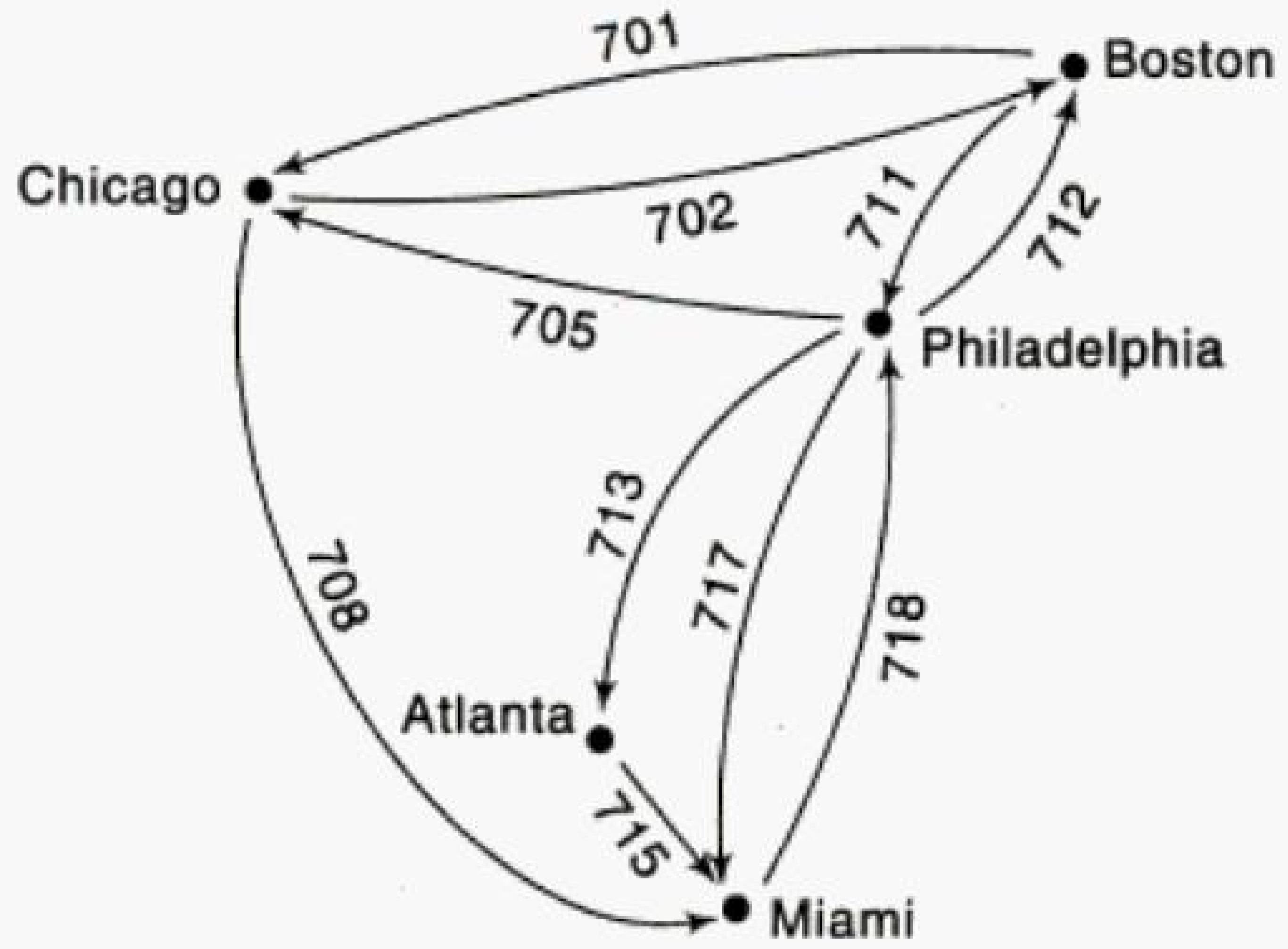


Fig. 1.20

Complexity; Space-Time Tradeoffs

1.13 Briefly describe the notions of (a) the complexity of an algorithm and (b) the space-time tradeoff of algorithms.

(a) The complexity of an algorithm is a function $f(n)$ which measures the time and/or space used by an algorithm in terms of the input size n.

(b) The space-time tradeoff refers to a choice between algorithmic solutions of a data processing problem that allows one to decrease the running time of an algorithmic solution by increasing the space to store the data and vice versa.

1.14 Suppose a data set S contains n elements.

(a) Compare the running time T_1 of the linear search algorithm with the running time T_2 of the binary search algorithm when (i) $n = 1000$ and (ii) $n = 10\ 000$.

(b) Discuss searching for a given item in S when S is stored as a linked list.

(a) Recall (Sec. 1.5) that the expected running of the linear search algorithm is $f(n) = n/2$ and that the binary search algorithm is $f(n) = \log_2 n$. Accordingly, (i) for $n = 1000$, $T_1 = 500$ but $T_2 = \log_2 1000 \approx 10$; and (ii) for $n = 10\ 000$, $T_1 = 5000$ but $T_2 = \log_2 10\ 000 \approx 14$.

(b) The binary search algorithm assumes that one can directly access the middle element in the set S. But one cannot directly access the middle element in a linked list. Hence one may have to use a linear search algorithm when S is stored as a linked list.

1.15 Consider the data in Fig. 1.19, which gives the different flights of an airline. Discuss different ways of storing the data so as to decrease the time in executing the following:

(a) Find the origin and destination of a flight, given the flight number.

(b) Given city A and city B, find whether there is a flight from A to B, and if there is, find its flight number.

(a) Store the data of Fig. 1.19(b) in arrays ORIG and DEST where the subscript is the flight number, as pictured in Fig. 1.21(a).
(b) Store the data of Fig. 1.19(b) in a two-dimensional array FLIGHT where FLIGHT[J, K] contains the flight number of the flight from CITY[J] to CITY[K], or contains 0 when there is no such flight, as pictured in Fig. 1.21(b).

	ORIG	DEST
701	2	3
702	3	2
703	0	0
704	0	0
705	5	3
706	0	0
⋮	⋮	⋮
715	1	4
716	0	0
717	5	4
718	4	5

(a)

FLIGHT	1	2	3	4	5
1	0	0	0	715	0
2	0	0	701	0	711
3	0	702	0	708	0
4	0	0	0	0	718
5	713	712	705	717	0

(b)

Fig. 1.21

1.16 Suppose an airline serves n cities with s flights. Discuss drawbacks to the data representations used in Fig. 1.21(a) and Fig. 1.21(b).

(a) Suppose the flight numbers are spaced very far apart; i.e. suppose the ratio of the number s of flights to the number of memory locations is very small, e.g. approximately 0.05. Then the extra storage space may not be worth the expense.
(b) Suppose the ratio of the number s of flights to the number n of memory locations in the array FLIGHT is very small, i.e. that the array FLIGHT is one that contains a large number of zeros (such an array is called a sparse matrix). Then the extra storage space may not be worth the expense.

1.17 List examples of linear data structures.

Arrays, linked lists, stacks and queues are examples of linear data structures.

1.18 Define Abstract Data Type. Explain it briefly.

An abstract data type can be defined as a data declaration packaged together with the operations that are meaningful for the data type. In other words, we encapsulate the data and the operations on the data, and then we hide them from the user.

MULTIPLE CHOICE QUESTIONS

1.1 ________ refers to a single unit of values.
(a) Group item (b) Data item
(c) Elementary item (d) Basic item

1.2 A __________ is something that has certain attributes or properties which may be assigned values.
(a) Field (b) Record
(c) Entity (d) File

1.3 ________ is the collection of records of the entities in a given entity set.
(a) Field (b) Record
(c) Entity (d) File

1.4 The value in a ________ field uniquely determines the record in a file.
(a) Primary key (b) Secondary key
(c) Key (d) Pointer

1.5 In ________ length records, file records may contain different lengths.
(a) Fixed (b) Primary
(c) Variable (d) Entity

1.6 ________ is the logical or mathematical model of a particular organization of data.
(a) Structure (b) Variable
(c) Function (d) Data Structures

1.7 Which of the following is not a primitive data structure?
(a) Boolean (b) Integer
(c) Arrays (d) Character

1.8 Which of the following is a non-linear data structure?
(a) Array (b) Linked List
(c) Stack (d) Graph

1.9 ________ is also called last-in-first-out (LIFO) system
(a) Queue (b) Stack
(c) Graph (d) Tree

1.10 __________ is also called first-in first-out (FIFO) system.
(a) Tree (b) Stack
(c) Queue (d) Graph

1.11 Which of the following operations accesses each record exactly once so that certain items may be processed?
(a) Inserting (b) Deleting
(c) Traversing (d) Searching

1.12 __________ is a data structure that contains a relationship between a pair of elements, which is not necessarily hierarchical in nature.
(a) Tree (b) Graph
(c) Array (d) String

1.13 __________ involves arranging the records in a logical order.
(a) Merging (b) Sorting
(c) Traversing (d) Searching

1.14 __________ is a set of data values and associated operations that are specified accurately, independent of any particular implementation.
(a) Stack
(b) Tree
(c) Abstract Data Type
(d) Graph

1.15 Which of the following operations combine records in two different sorted files into a single sorted file?
(a) Inserting (b) Sorting
(c) Searching (d) Merging

ANSWERS TO MULTIPLE CHOICE QUESTIONS

1.1 (b) **1.2** (c) **1.3** (d) **1.4** (a) **1.5** (a) **1.6** (d) **1.7** (c) **1.8** (d)
1.9 (d) **1.10** (c) **1.11** (c) **1.12** (b) **1.13** (b) **1.14** (c) **1.15** (d)

Chapter 2

Preliminaries

2.1 INTRODUCTION

The development of algorithms for the creation and processing of data structures is a major feature of this text. This chapter describes, by means of simple examples, the format that will be used to present our algorithms. The format we have selected is similar to the format used by Knuth in his well-known text *Fundamental Algorithms.* Although our format is language-free, the algorithms will be sufficiently well structured and detailed so that they can be easily translated into some programming language such as Pascal, C, etc. In fact going forward, the algorithms will be implemented using the C language throughout the text.

Algorithms may be quite complex. The computer programs implementing the more complex algorithms can be more easily understood if these programs are organized into hierarchies of modules similar to the one in Fig. 2.1. In such an organization, each program contains first a main module, which gives a general description of the algorithm; this main module refers to certain submodules, which contain more detailed information than the main module; each of the submodules may refer to more detailed submodules; and so on. The organization of a program into such a hierarchy of modules normally requires the use of certain basic flow patterns and logical structures which are usually associated with the notion of structured programming. These flow patterns and logical structures will be reviewed in this chapter.

The chapter begins with a brief outline and discussion of various mathematical functions which occur in the study of algorithms and in computer science in general, and the chapter ends with a discussion of the different kinds of variables that can appear in our algorithms and programs.

The notion of the complexity of an algorithm is also covered in this chapter. This important measurement of algorithms gives us a tool to compare different algorithmic solutions to a particular problem such as searching or sorting. The concept of an algorithm and its complexity is fundamental not only to data structures but also to almost all areas of computer science.

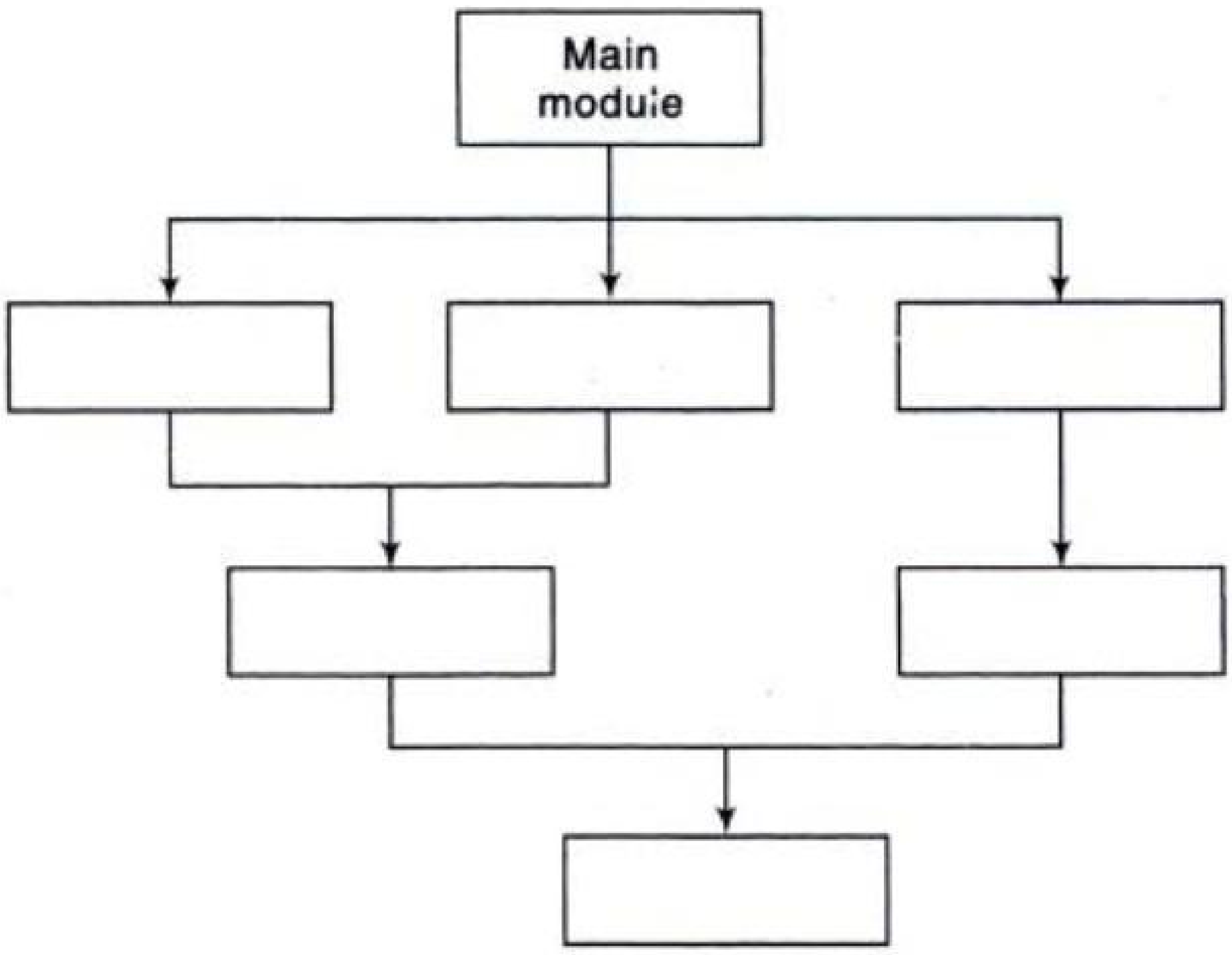


Fig. 2.1 A Hierarchy of Modules

2.2 MATHEMATICAL NOTATIONS AND FUNCTIONS

This section gives various mathematical functions which appear very often in the analysis of algorithms and in computer science in general, together with their notation.

Floor and Ceiling Functions

Let x be any real number. Then x lies between two integers called the floor and the ceiling of x. Specifically,

$\lfloor x \rfloor$, called the *floor* of x, denotes the greatest integer that does not exceed x.

$\lceil x \rceil$, called the *ceiling* of x, denotes the least integer that is not less than x.

If x is itself an integer, then $\lfloor x \rfloor = \lceil x \rceil$; otherwise $\lfloor x \rfloor + 1 = \lceil x \rceil$.

Example 2.1

$\lfloor 3.14 \rfloor = 3$, $\lfloor \sqrt{5} \rfloor = 2$, $\lfloor -8.5 \rfloor = -9$, $\lfloor 7 \rfloor = 7$

$\lceil 3.14 \rceil = 4$, $\lceil \sqrt{5} \rceil = 3$, $\lceil -8.5 \rceil = -8$, $\lceil 7 \rceil = 7$

Remainder Function; Modular Arithmetic

Let k be any integer and let M be a positive integer. Then

$$k \pmod{M}$$

(read k *modulo* M) will denote the integer remainder when k is divided by M. More exactly, $k \pmod{M}$ is the unique integer r such that

$$k = Mq + r \quad \text{where} \quad 0 \le r < M$$

When k is positive, simply divide k by M to obtain the remainder r. Thus

$$25 \pmod 7 = 4, \quad 25 \pmod 5 = 0, \quad 35 \pmod{11} = 2, \quad 3 \pmod 8 = 3$$

Problem 2.2(b) shows a method to obtain $k \pmod M$ when k is negative.

The term "mod" is also used for the mathematical congruence relation, which is denoted and defined as follows:

$$a \equiv b \pmod M \quad \text{if and only if} \quad M \text{ divides } b - a$$

M is called the *modulus*, and $a \equiv b \pmod M$ is read "a is congruent to b modulo M." The following aspects of the congruence relation are frequently useful:

$$0 \equiv M \pmod M \quad \text{and} \quad a \pm M \equiv a \pmod M$$

Arithmetic modulo M refers to the arithmetic operations of addition, multiplication and subtraction where the arithmetic value is replaced by its equivalent value in the set

$$\{0, 1, 2, \ldots, M - 1\}$$

or in the set

$$\{1, 2, 3, \ldots, M\}$$

For example, in arithmetic modulo 12, sometimes called "clock" arithmetic,

$$6 + 9 \equiv 3, \quad 7 \times 5 \equiv 11, \quad 1 - 5 \equiv 8, \quad 2 + 10 \equiv 0 \equiv 12$$

(The use of 0 or M depends on the application.)

Integer and Absolute Value Functions

Let x be any real number. The *integer value* of x, written INT(x), converts x into an integer by deleting (truncating) the fractional part of the number. Thus

$$\text{INT}(3.14) = 3, \quad \text{INT}(\sqrt{5}) = 2, \quad \text{INT}(-8.5) = -8, \quad \text{INT}(7) = 7$$

Observe that INT(x) = $\lfloor x \rfloor$ or INT(x) = $\lceil x \rceil$ according to whether x is positive or negative.

The *absolute value* of the real number x, written ABS(x) or $|x|$, is defined as the greater of x or $-x$. Hence ABS(0) = 0, and, for $x \neq 0$, ABS(x) = x or ABS(x) = $-x$, depending on whether x is positive or negative. Thus

$$|-15| = 15, \quad |7| = 7, \quad |-3.33| = 3.33, \quad |4.44| = 4.44, \quad |-0.075| = 0.075$$

We note that $|x| = |-x|$ and, for $x \neq 0$, $|x|$ is positive.

Summation Symbol; Sums

Here we introduce the summation symbol Σ (the Greek letter sigma). Consider a sequence $a_1, a_2, a_3, \ldots$. Then the sums

$$a_1 + a_2 + \cdots + a_n \quad \text{and} \quad a_m + a_{m+1} + \cdots + a_n$$

will be denoted, respectively, by

$$\sum_{j=1}^{n} a_j \quad \text{and} \quad \sum_{j=m}^{n} a_j$$

The letter j in the above expressions is called a *dummy index* or *dummy variable*. Other letters frequently used as dummy variables are i, k, s and t.

Example 2.2

$$\sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$$

$$\sum_{j=2}^{5} j^2 = 2^2 + 3^2 + 4^2 + 5^2 = 4 + 9 + 16 + 25 = 54$$

$$\sum_{j=1}^{n} j = 1 + 2 + \cdots + n$$

The last sum in Example 2.2 will appear very often. It has the value $n(n + 1)/2$. That is,

$$1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$$

Thus, for example,

$$1 + 2 + \cdots + 50 = \frac{50(51)}{2} = 1275$$

Factorial Function

The product of the positive integers from 1 to n, inclusive, is denoted by $n!$ (read "n factorial"). That is,

$$n! = 1 \cdot 2 \cdot 3 \cdots (n-2)(n-1)n$$

It is also convenient to define $0! = 1$.

Example 2.3

(a) $2! = 1 \cdot 2 = 2;\quad 3! = 1 \cdot 2 \cdot 3 = 6;\quad 4! = 1 \cdot 2 \cdot 3 \cdot 4 = 24$

(b) For $n > 1$, we have $n! = n \cdot (n-1)!$ Hence

$$5! = 5 \cdot 4! = 5 \cdot 24 = 120;\quad 6! = 6 \cdot 5! = 6 \cdot 120 = 720$$

Permutations

A *permutation* of a set of n elements is an arrangement of the elements in a given order. For example, the permutations of the set consisting of the elements a, b, c are as follows:

$$abc,\quad acb,\quad bac,\quad bca,\quad cab,\quad cba$$

One can prove: *There are $n!$ permutations of a set of n elements.* Accordingly, there are $4! = 24$ permutations of a set with 4 elements, $5! = 120$ permutations of a set with 5 elements, and so on.

Exponents and Logarithms

Recall the following definitions for integer exponents (where m is a positive integer):

$$a^m = a \cdot a \cdots a \ (m \text{ times}), \quad a^0 = 1, \quad a^{-m} = \frac{1}{a^m}$$

Exponents are extended to include all rational numbers by defining, for any rational number m/n,

$$a^{m/n} = \sqrt[n]{a^m} = \left(\sqrt[n]{a}\right)^m$$

For example,

$$2^4 = 16, \qquad 2^{-4} = \frac{1}{2^4} = \frac{1}{16}, \quad 125^{2/3} = 5^2 = 25$$

In fact, exponents are extended to include all real numbers by defining, for any real number x,

$$a^x = \lim_{r \to x} a^r \qquad \text{where } r \text{ is a rational number}$$

Accordingly, the exponential function $f(x) = a^x$ is defined for all real numbers.

Logarithms are related to exponents as follows. Let b be a positive number. The logarithm of any positive number x to the base b, written

$$\log_b x$$

represents the exponent to which b must be raised to obtain x. That is,

$$y = \log_b x \quad \text{and} \quad b^y = x$$

are equivalent statements. Accordingly,

$$\log_2 8 = 3 \ \text{ since } \ 2^3 = 8; \qquad \log_{10} 100 = 2 \ \text{ since } \ 10^2 = 100$$
$$\log_2 64 = 6 \ \text{ since } \ 2^6 = 64; \qquad \log_{10} 0.001 = -3 \ \text{ since } \ 10^{-3} = 0.001$$

Furthermore, for any base b,

$$\log_b 1 = 0 \ \text{ since } \ b^0 = 1$$
$$\log_b b = 1 \ \text{ since } \ b^1 = b$$

The logarithm of a negative number and the logarithm of 0 are not defined.

One may also view the exponential and logarithmic functions

$$f(x) = b^x \qquad \text{and} \qquad g(x) = \log_b x$$

as inverse functions of each other. Accordingly, the graphs of these two functions are related. (See Solved Problem 2.5.)

Frequently, logarithms are expressed using approximate values. For example, using tables or calculators, one obtains

$$\log_{10} 300 = 2.4771 \quad \text{and} \quad \log_e 40 = 3.6889$$

as approximate answers. (Here $e = 2.718281 \cdots$.)

Logarithms to the base 10 (called *common logarithms*), logarithms to the base *e* (called *natural logarithms*) and logarithms to the base 2 (called *binary logarithms*) are of special importance. Some texts write:

$\ln x$	instead of	$\log_e x$
$\lg x$ or $\text{Log } x$	instead of	$\log_2 x$

This text on data structures is mainly concerned with binary logarithms. Accordingly,

> The term $\log x$ shall mean $\log_2 x$ unless otherwise specified.

Frequently, we will require only the floor or the ceiling of a binary logarithm. This can be obtained by looking at the powers of 2. For example,

$$\lfloor \log_2 100 \rfloor = 6 \quad \text{since} \quad 2^6 = 64 \quad 2^7 = 128$$
$$\lceil \log_2 1000 \rceil = 10 \quad \text{since} \quad 2^9 = 512 \quad \text{and} \quad 2^{10} = 1024$$

and so on.

2.3 ALGORITHMIC NOTATIONS

An algorithm, intuitively speaking, is a finite step-by-step list of well-defined instructions for solving a particular problem. The formal definition of an algorithm, which uses the notion of a Turing machine or its equivalent, is very sophisticated and lies beyond the scope of this text. This section describes the format that is used to present algorithms throughout the text. This algorithmic notation is best described by means of examples.

Example 2.4

An array DATA of numerical values is in memory. We want to find the location LOC and the value MAX of the largest element of DATA. Given no other information about DATA, one way to solve the problem is as follows:

Initially begin with LOC = 1 and MAX = DATA[1]. Then compare MAX with each successive element DATA[K] of DATA. If DATA[K] exceeds MAX, then update LOC and MAX so that LOC = K and MAX = DATA[K]. The final values appearing in LOC and MAX give the location and value of the largest element of DATA.

A formal presentation of this algorithm, whose flow chart appears in Fig. 2.2, follows.

Algorithm 2.1: (Largest Element in Array) A nonempty array DATA with N numerical values is given. This algorithm finds the location LOC and the value MAX of the largest element of DATA. The variable K is used as a counter.

Step 1. [Initialize.] Set K : = 1, LOC : = 1 and MAX : = DATA[1].
Step 2. [Increment counter.] Set K : = K + 1.
Step 3. [Test counter.] If K > N, then:
Write: LOC, MAX, and Exit.
Step 4. [Compare and update.] If MAX < DATA[K], then:
Set LOC : = K and MAX : = DATA[K].
Step 5. [Repeat loop.] Go to Step 2.

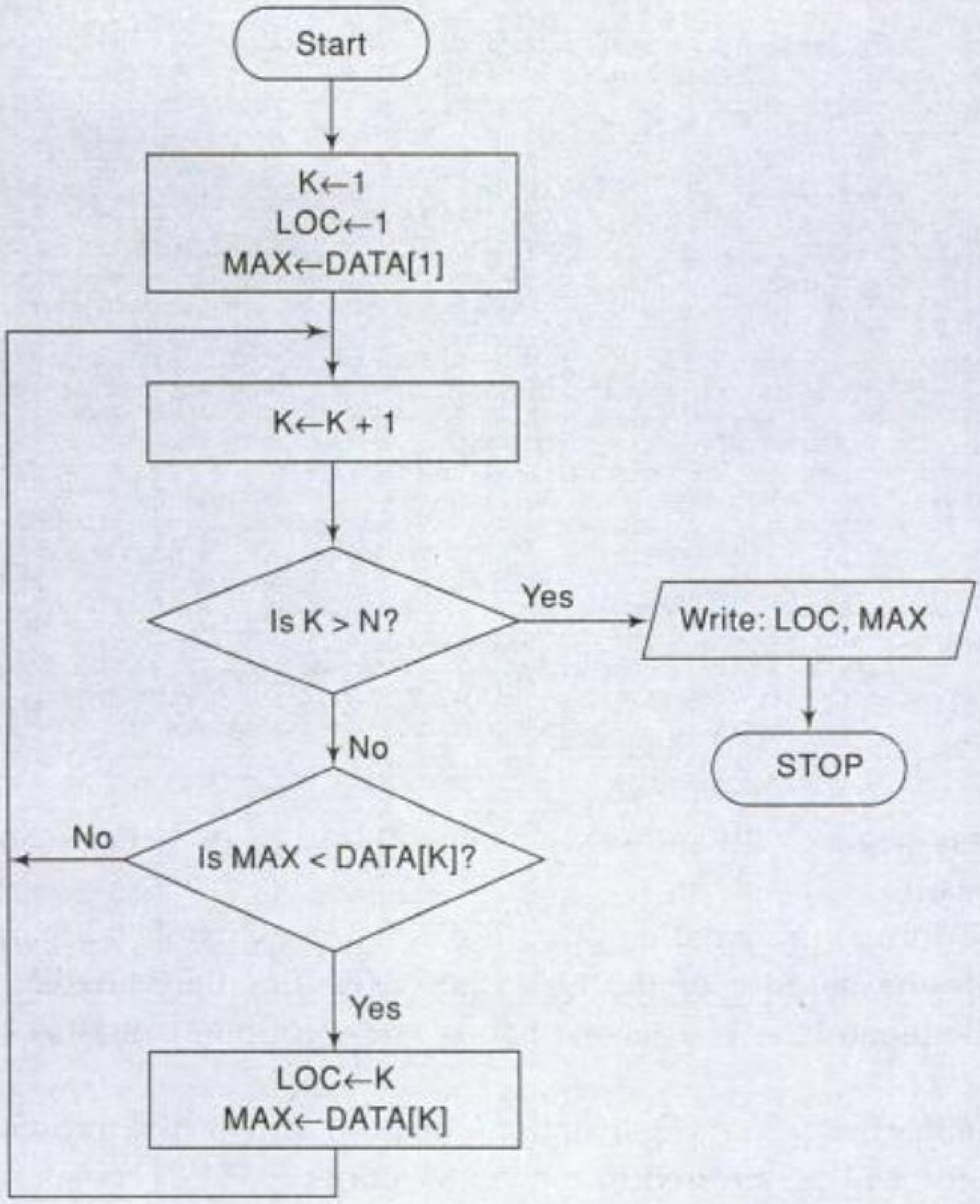


Fig. 2.2

Program 2.1

```
/* C implementation of Algorithm 2.1 */
#include <stdio.h>
#include <conio.h>

void main()
{
int DATA[10]={22,65,1,99,32,17,74,49,33,2};
int N, LOC, MAX, K;
N=10;
K=0;
LOC=0;
MAX=DATA[0];
clrscr();
```

```
loop:
K=K+1;
if(K==N)
{
 printf("LOC = %d, MAX= %d",LOC,MAX);
 getch();
 exit();
}
if(MAX<DATA[K])
{
 LOC=K;
 MAX=DATA[K];
}
goto loop;
}
```

Output:

```
LOC = 3, MAX= 99
```

Remark: In C, an array begins with an index value of 0 instead of 1. For instance, an array A[10] will have index values 0, 1, 2,....8, 9.

The format for the formal presentation of an algorithm consists of two parts. The first part is a paragraph which tells the purpose of the algorithm, identifies the variables which occur in the algorithm and lists the input data. The second part of the algorithm consists of the list of steps that is to be executed.

The following summarizes certain conventions that we will use in presenting our algorithms. Some control structures will be covered in the next section.

Identifying Number

Each algorithm is assigned an identifying number as follows: Algorithm 4.3 refers to the third algorithm in Chapter 4; Algorithm P5.3 refers to the algorithm in Solved Problem 5.3 in Chapter 5. Note that the letter "P" indicates that the algorithm appears in a problem.

Steps, Control, Exit

The steps of the algorithm are executed one after the other, beginning with Step 1, unless indicated otherwise. Control may be transferred to Step *n* of the algorithm by the statement "Go to Step *n*." For example, Step 5 transfers control back to Step 2 in Algorithm 2.1. Generally speaking, these Go to statements may be practically eliminated by using certain control structures discussed in the next section.

If several statements appear in the same step, e.g.,

Set K := 1, LOC := 1 and MAX := DATA[1].

then they are executed from left to right.

The algorithm is completed when the statement

Exit.

is encountered. This statement is similar to the STOP statement used in FORTRAN and in flowcharts.

Comments

Each step may contain a comment in brackets which indicates the main purpose of the step. The comment will usually appear at the beginning or the end of the step.

Variable Names

Variable names will use capital letters, as in MAX and DATA. Single-letter names of variables used as counters or subscripts will also be capitalized in the algorithms (K and N, for example), even though lowercase may be used for these same variables (*k* and *n*) in the accompanying mathematical description and analysis. (Recall the discussion of italic and lowercase symbols in Sec. 1.3 of Chapter 1, under "Arrays.")

Assignment Statement

Our assignment statements will use the dots-equal notation := that is used in Pascal. For example,

Max := DATA[1]

assigns the value in DATA[1] to MAX. In C language, we use the equal sign = for this operation.

Input and Output

Data may be input and assigned to variables by means of a Read stateme.. with the following form:

Read: Variables names.

Similarly, messages, placed in quotation marks, and data in variables may be output by means of a Write or Print statement with the following form:

Write: Messages and/or variable names.

Procedures

The term "procedure" will be used for an independent algorithmic module which solves a particular problem. The use of the word "procedure" or "module" rather than "algorithm" for a given problem is simply a matter of taste. Generally speaking, the word "algorithm" will be reserved for the solution of general problems. The term "procedure" will also be used to describe a certain type of subalgorithm which is discussed in Sec. 2.6.

2.4 CONTROL STRUCTURES

Algorithms and their equivalent computer programs are more easily understood if they mainly use self-contained modules and three types of logic, or flow of control, called

1. Sequence logic, or sequential flow
2. Selection logic, or conditional flow
3. Iteration logic, or repetitive flow

These three types of logic are discussed below, and in each case we show the equivalent flowchart.

Sequence Logic (Sequential Flow)

Sequence logic has already been discussed. Unless instructions are given to the contrary, the modules are executed in the obvious sequence. The sequence may be presented explicitly, by means of numbered steps, or implicitly, by the order in which the modules are written. (See Fig. 2.3.) Most processing, even of complex problems, will generally follow this elementary flow pattern.

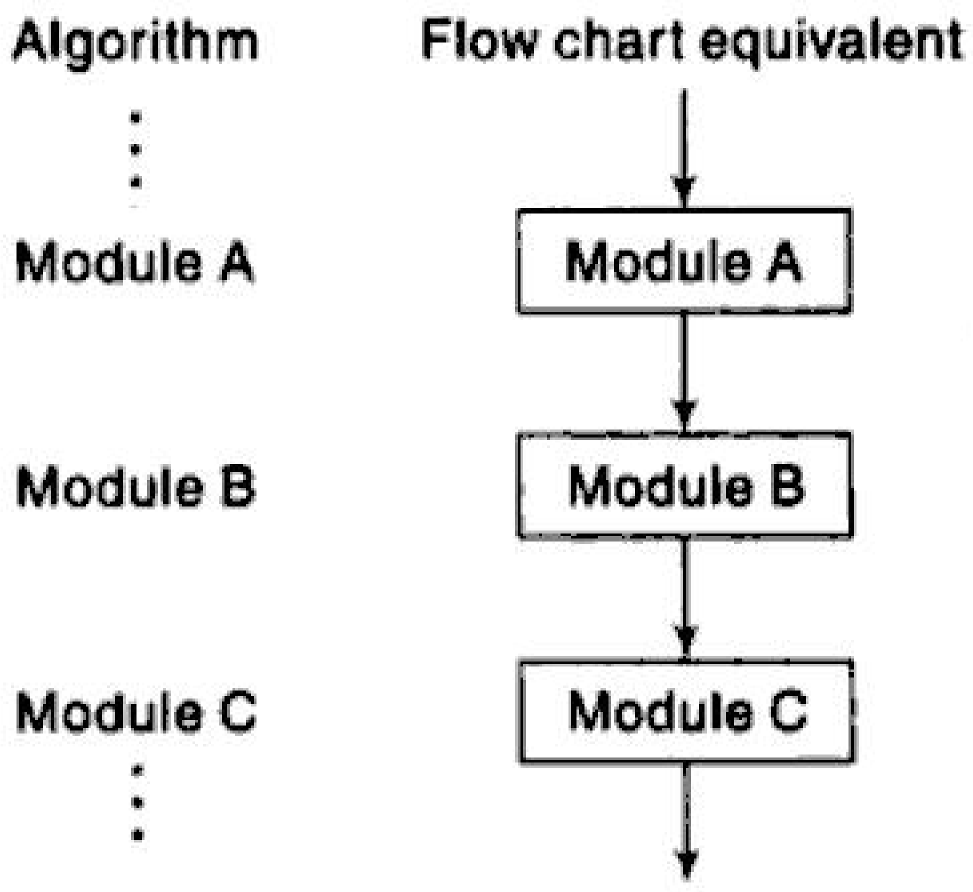


Fig. 2.3 Sequence Logic

Selection Logic (Conditional Flow)

Selection logic employs a number of conditions which lead to a selection of one out of several alternative modules. The structures which implement this logic are called conditional structures or If structures. For clarity, we will frequently indicate the end of such a structure by the statement

[End of If structure.]

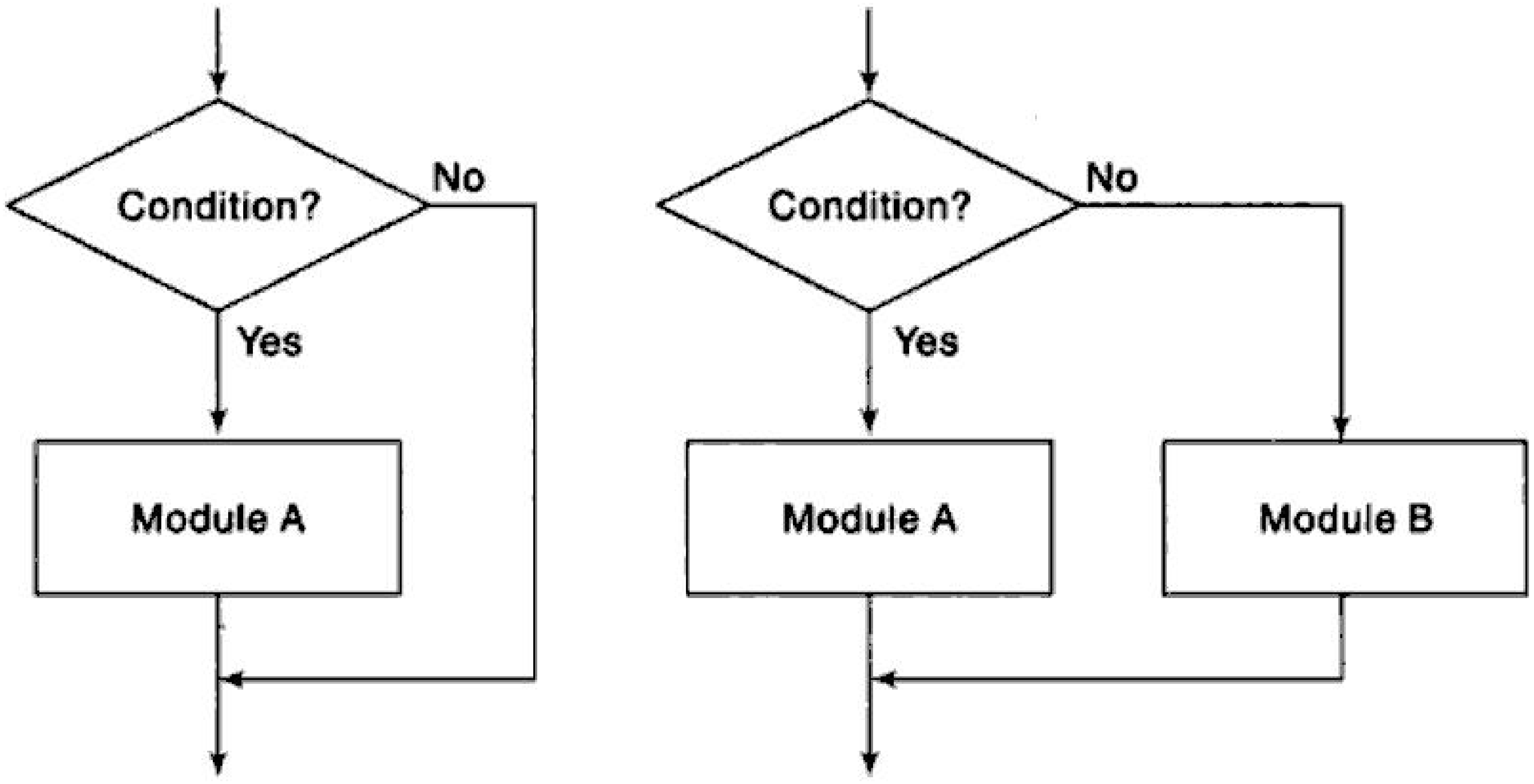
or some equivalent.

These conditional structures fall into three types, which are discussed separately.

1. **Single Alternative.** This structure has the form

If condition, then:
[Module A]
[End of If structure.]

The logic of this structure is pictured in Fig. 2.4(a). If the condition holds, then Module A, which may consist of one or more statements, is executed; otherwise Module A is skipped and control transfers to the next step of the algorithm.



(a) Single alternative. (b) Double alternative.

Fig. 2.4

2. **Double Alternative.** This structure has the form

If condition, then:
    [Module A]
Else:
    [Module B]
[End of If structure.]

The logic of this structure is pictured in Fig. 2.4(b). As indicated by the flow chart, if the condition holds, then Module A is executed; otherwise Module B is executed.

3. **Multiple Alternatives.** This structure has the form

If condition(1), then:
    [Module A_1]
Else if condition(2), then:
    [Module A_2]
    ⋮
Else if condition(M), then:
    [Module A_M]
Else:
    [Module B]
[End of If structure.]

The logic of this structure allows only one of the modules to be executed. Specifically, either the module which follows the first condition which holds is executed, or the module which follows the final Else statement is executed. In practice, there will rarely be more than three alternatives.

Example 2.5

The solutions of the quadratic equation

$$ax^2 + bx + c = 0$$

where $a \neq 0$, are given by the quadratic formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

The quantity $D = b^2 - 4ac$ is called the *discriminant* of the equation. If D is negative, then there are no real solutions. If $D = 0$, then there is only one (double) real solution, $x = -b/2a$. If D is positive, the formula gives the two distinct real solutions. The following algorithm finds the solutions of a quadratic equation.

Algorithm 2.2: (Quadratic Equation) This algorithm inputs the coefficients A, B, C of a quadratic equation and outputs the real solutions, if any.

Step 1. Read: A, B, C.
Step 2. Set D: = B^2 − 4AC.
Step 3. If D > 0, then:
    (a) Set X1 := $(-B + \sqrt{D})/2A$ and
        X2 := $(-B - \sqrt{D})/2A$.

```
        (b) Write: X1, X2.
    Else if D = 0, then:
        (a) Set X := -B/2A.
        (b) Write: 'UNIQUE SOLUTION', X.
    Else:
        Write: 'NO REAL SOLUTIONS'
    [End of If structure.]
Step 4. Exit.
```

Remark: Observe that there are three mutually exclusive conditions in Step 3 of Algorithm 2.2 that depend on whether D is positive, zero or negative. In such a situation, we may alternatively list the different cases as follows:

```
Step 3.  1. If D > 0, then:
               .........
         2. If D = 0, then:
               .........
         3. If D < 0, then:
               .........
```

Program 2.2

```
/* C implementation of Algorithm 2.2*/
#include <stdio.h>
#include <conio.h>
#include <math.h>

void main()
{
int A,B,C,D;
float X,X1,X2;
clrscr();

printf("Enter the values of A, B and C: ");
scanf("%d %d %d",&A,&B,&C);
D=B*B-4*A*C;

if(D>0)
{
 X1=((-1)*B+sqrt(D))/2*A;
 X2=((-1)*B-sqrt(D))/2*A;
 printf("X1= %.2f, X2= %.2f", X1,X2);
}
else if(D==0)
{
 X=(-1)*B/2*A;
 printf("UNIQUE SOLUTION X=%.2f",X);
}
else
printf("NO REAL SOLUTIONS");
```

```
getch();
}

Output:
Enter the values of A, B and C: 1
-3
-4
X1= 4.00, X2= -1.00
```

Iteration Logic (Repetitive Flow)

The third kind of logic refers to either of two types of structures involving loops. Each type begins with a Repeat statement and is followed by a module, called the *body of the loop*. For clarity, we will indicate the end of the structure by the statement

[End of loop.]

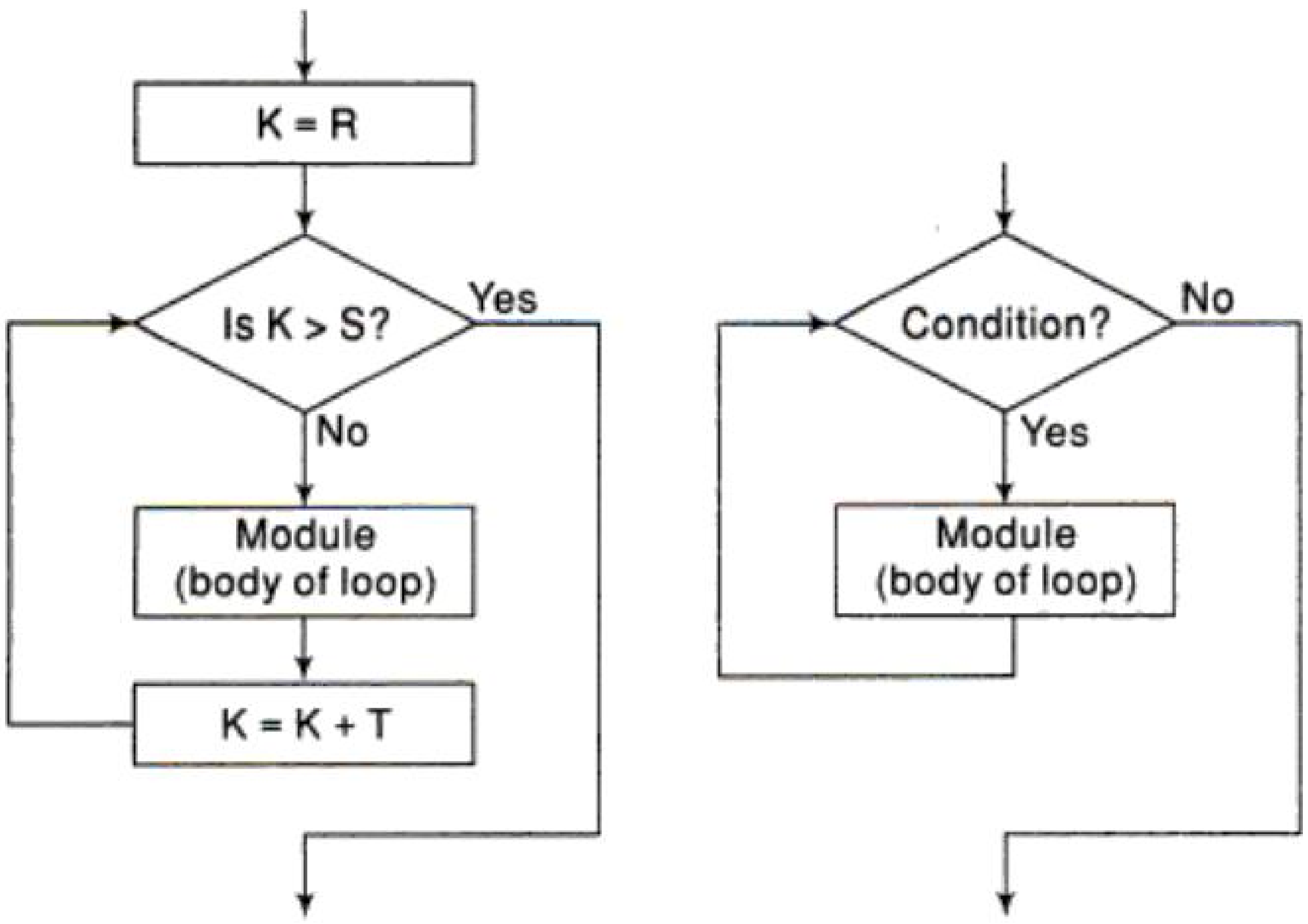
or some equivalent.

Each type of loop structure is discussed separately.

The *repeat-for loop* uses an index variable, such as K, to control the loop. The loop will usually have the form:

Repeat for K = R to S by T:
[Module]
[End of loop.]

The logic of this structure is pictured in Fig. 2.5(a). Here R is called the *initial value*, S the *end value* or *test value*, and T the *increment*. Observe that the body of the loop is executed fitst with K = R, then with K = R + T, then with K = R + 2T, and so on. The cycling ends when K > S. The flow chart assumes that the increment T is positive; if T is negative, so that K decreases in value, then the cycling ends when K < S.



(a) Repeat-For structure. (a) Repeat-While structure.

Fig. 2.5

The *repeat-while loop* uses a condition to control the loop. The loop will usually have the form

Repeat while condition:
[Module]
[End of loop.]

The logic of this structure is pictured in Fig. 2.5(b). Observe that the cycling continues until the condition is false. We emphasize that there must be a statement before the structure that initializes the condition controlling the loop, and in order that the looping may eventually cease, there must be a statement in the body of the loop that changes the condition.

Example 2.6

Algorithm 2.1 is rewritten using a repeat-while loop rather than a Go to statement:

Algorithm 2.3: (Largest Element in Array) Given a nonempty array DATA with N numerical values, this algorithm finds the location LOC and the value MAX of the largest element of DATA.

1. [Initialize.] Set K := 1, LOC := 1 and MAX := DATA[1].
2. Repeat Steps 3 and 4 while K ≤ N:
3. If MAX < DATA[K], then:
   Set LOC := K and MAX := DATA[K].
   [End of If structure.]
4. Set K := K + 1.
   [End of Step 2 loop.]
5. Write: LOC, MAX.
6. Exit.

Program 2.3

```
/* C implementation of Algorithm 2.3*/
#include <stdio.h>
#include <conio.h>

void main()
{
int  DATA[10]={22,65,1,99,32,17,74,49,33,2};
int N, LOC, MAX, K;
N=10;
K=0;
LOC=0;
MAX=DATA[0];
clrscr();

while(K<N)
{
  if(MAX<DATA[K])
```

```
    {
      LOC=K;
      MAX=DATA[K];
    }
   K=K+1;
  }

  printf("LOC= %d, MAX= %d",LOC,MAX);
  getch();
  }

  Output:
  LOC = 3, MAX= 99
```

Algorithm 2.3 indicates some other properties of our algorithms. Usually we will omit the word "Step." We will try to use repeat structures instead of Go to statements. The repeat statement may explicitly indicate the steps that form the body of the loop. The "End of loop" statement may explicitly indicate the step where the loop begins. The modules contained in our logic structures will normally be indented for easier reading. This conforms to the usual format in structured programming.

Any other new notation or convention either will be self-explanatory or will be explained when it occurs.

2.5 COMPLEXITY OF ALGORITHMS

The analysis of algorithms is a major task in computer science. In order to compare algorithms, we must have some criteria to measure the efficiency of our algorithms. This section discusses this important topic.

Suppose M is an algorithm, and suppose n is the size of the input data. The time and space used by the algorithm M are the two main measures for the efficiency of M. The time is measured by counting the number of key operations—in sorting and searching algorithms, for example, the number of comparisons. That is because key operations are so defined that the time for the other operations is much less than or at most proportional to the time for the key operations. The space is measured by counting the maximum of memory needed by the algorithm.

The *complexity* of an algorithm M is the *function* $f(n)$ which gives the running time and/or storage space requirement of the algorithm in terms of the size n of the input data. Frequently, the storage space required by an algorithm is simply a multiple of the data size n. Accordingly, unless otherwise stated or implied, the term "complexity" shall refer to the running time of the algorithm.

The following example illustrates that the function $f(n)$, which gives the running time of an algorithm, depends not only on the size n of the input data but also on the particular data.

Example 2.7

Suppose we are given an English short story TEXT, and suppose we want to search through TEXT for the first occurrence of a given 3-letter word W. If W is the 3-letter word "the," then it is likely that W occurs near the beginning of TEXT, so $f(n)$ will be small. On the other hand, if W is the 3-letter word "zoo," then W may not appear in TEXT at all, so $f(n)$ will be large.

The above discussion leads us to the question of finding the complexity function $f(n)$ for certain cases. The two cases one usually investigates in complexity theory are as follows:

1. *Worst case:* the maximum value of $f(n)$ for any possible input
2. *Average case:* the expected value of $f(n)$

Sometimes we also consider the minimum possible value of $f(n)$, called the *best case.*

The analysis of the average case assumes a certain probabilistic distribution for the input data; one such assumption might be that all possible permutations of an input data set are equally likely. The average case also uses the following concept in probability theory. Suppose the numbers $n_1, n_2, \ldots, n_k$ occur with respective probabilities $p_1, p_2, \ldots, p_k$. Then the *expectation* or *average value* E is given by

$$E = n_1p_1 + n_2p_2 + \cdots + n_kp_k$$

These ideas are illustrated in the following example.

Example 2.8 Linear Search

Suppose a linear array DATA contains *n* elements, and suppose a specific ITEM of information is given. We want either to find the location LOC of ITEM in the array DATA, or to send some message, such as LOC = 0, to indicate that ITEM does not appear in DATA. The linear search algorithm solves this problem by comparing ITEM, one by one, with each element in DATA. That is, we compare ITEM with DATA[1], then DATA[2], and so on, until we find LOC such that ITEM = DATA[LOC]. A formal presentation of this algorithm follows.

Algorithm 2.4: (Linear Search) A linear array DATA with N elements and a specific ITEM of information are given. This algorithm finds the location LOC of ITEM in the array DATA or sets LOC = 0.

```
1.  [Initialize] Set K := 1 and LOC := 0.
2.  Repeat Steps 3 and 4 while LOC = 0 and K ≤ N.
3.       If ITEM = DATA[K], then: Set LOC: = K.
4.       Set K := K + 1. [Increments counter.]
    [End of Step 2 loop.]
5.  [Successful?]
    If LOC = 0, then:
         Write: ITEM is not in the array DATA.
    Else:
         Write: LOC is the location of ITEM.
    [End of If structure.]
6.  Exit.
```

Program 2.4

```
/* C implementation of Algorithem 2.4*/
#include <stdio.h>
#include <conio.h>

void main()
{
int DATA[10]={22,65,1,99,32,17,74,49,33,2};
```

```
int ITEM=17;
int N, LOC, K;
N=10;
K=0;
LOC=-1;
clrscr();

while(LOC==-1 && K<N)
{
  if(ITEM==DATA[K])
   LOC=K;
  K=K+1;
}

if(LOC==-1)
 printf("ITEM is not in the array DATA");
else
 printf("%d is the location of ITEM",LOC);

getch();
}

Output:
5 is the location of ITEM
```

The complexity of the search algorithm is given by the number C of comparisons between ITEM and DATA[K]. We seek $C(n)$ for the worst case and the average case.

Worst Case

Clearly the worst case occurs when ITEM is the last element in the array DATA or is not there at all. In either situation, we have

$$C(n) = n$$

Accordingly, $C(n) = n$ is the worst-case complexity of the linear search algorithm.

Average Case

Here we assume that ITEM does appear in DATA, and that it is equally likely to occur at any position in the array. Accordingly, the number of comparisons can be any of the numbers 1, 2, 3, ..., n, and each number occurs with probability $p = 1/n$. Then

$$\begin{aligned} C(n) &= 1 \cdot \frac{1}{n} + 2 \cdot \frac{1}{n} + \cdots + n \cdot \frac{1}{n} \\ &= (1 + 2 + \cdots + n) \cdot \frac{1}{n} \\ &= \frac{n(n+1)}{2} \cdot \frac{1}{n} = \frac{n+1}{2} \end{aligned}$$

This agrees with our intuitive feeling that the average number of comparisons needed to find the location of ITEM is approximately equal to half the number of elements in the DATA list.

Remark: The complexity of the average case of an algorithm is usually much more complicated to analyze than that of the worst case. Moreover, the probabilistic distribution that one assumes for the average case may not actually apply to real situations. Accordingly, unless otherwise stated or implied, the complexity of an algorithm shall mean the function which gives the running time of the worst case in terms of the input size. This is not too strong an assumption, since the complexity of the average case for many algorithms is proportional to the worst case.

Rate of Growth; Big *O* Notation

Suppose M is an algorithm, and suppose n is the size of the input data. Clearly the complexity $f(n)$ of M increases as n increases. It is usually the rate of increase of $f(n)$ that we want to examine. This is usually done by comparing $f(n)$ with some standard function, such as

$$\log_2 n, \quad n \log_2 n, \quad n^2, \quad n^3, \quad 2^n$$

The rates of growth for these standard functions are indicated in Fig. 2.6, which gives their approximate values for certain values of n. Observe that the functions are listed in the order of their rates of growth: the logarithmic function $\log_2 n$ grows most slowly, the exponential function 2^n grows most rapidly, and the polynomial functions n^c grow according to the exponent c. One way to compare the function $f(n)$ with these standard functions is to use the functional O notation defined as follows:

n \ $g(n)$	$\log n$	n	$n \log n$	n^2	n^3	2^n
5	3	5	15	25	125	32
10	4	10	40	100	10^3	10^3
100	7	100	700	10^4	10^6	10^{30}
1000	10	10^3	10^4	10^6	10^9	10^{300}

Fig. 2.6 Rate of Growth of Standard Functions

Suppose $f(n)$ and $g(n)$ are functions defined on the positive integers with the property that $f(n)$ is bounded by some multiple of $g(n)$ for almost all n. That is, suppose there exist a positive integer n_0 and a positive number M such that, for all $n > n_0$, we have

$$|f(n)| \leq M|g(n)|$$

Then we may write

$$f(n) = O(g(n))$$

which is read "$f(n)$ is of order $g(n)$." For any polynomial $P(n)$ of degree m, we show in Solved Problem 2.10 that $P(n) = O(n^m)$; e.g.,

$$8n^3 - 576n^2 + 832n - 248 = O(n^3)$$

We can also write

$$f(n) = h(n) + O(g(n)) \quad \text{when} \quad f(n) - h(n) = O(g(n))$$

(This is called the "big O" notation since $f(n) = o(g(n))$ has an entirely different meaning.)

To indicate the convenience of this notation, we give the complexity of certain well-known searching and sorting algorithms:

(a) Linear search: $O(n)$
(b) Binary search: $O(\log n)$
(c) Bubble sort: $O(n^2)$
(d) Merge-sort: $O(n \log n)$

These results are discussed in Chapter 9, on sorting and searching.

2.6 OTHER ASYMPTOTIC NOTATIONS FOR COMPLEXITY OF ALGORITHMS Ω, Θ, o

The "big O" notation defines an upper bound function $g(n)$ for $f(n)$ which represents the time/space complexity of the algorithm on an input characteristic n. There are other asymptotic notations such as Ω, Θ, o which also serve to provide bounds for the function $f(n)$.

Omega Notation (Ω)

The omega notation is used when the function $g(n)$ defines a lower bound for the function $f(n)$.

Definition

$f(n) = \Omega(g(n))$ (read as f of n is omega of g of n), iff there exists a positive integer n_0 and a positive number M such that $|f(n)| \geq M|g(n)|$, for all $n \geq n_0$.

For $f(n) = 18n + 9$, $f(n) > 18n$ for all n, hence $f(n) = \Omega(n)$. Also, for $f(n) = 90n^2 + 18n + 6$, $f(n) > 90n^2$ for $n \geq 0$ and therefore $f(n) = \Omega(n^2)$.

For $f(n) = \Omega(g(n))$, $g(n)$ is a lower bound function and there may be several such functions, but it is appropriate that the function which is almost as large a function of n as possible such that the definition of Ω is satisfied, is chosen as $g(n)$. Thus for example, $f(n) = 5n + 1$ leads to both $f(n) = \Omega(n)$ and $f(n) = \Omega(1)$. However, we never consider the latter to be correct, since $f(n) = \Omega(n)$ represents the largest possible function of n satisfying the definition of Ω and hence is more informative.

Theta Notation (Θ)

The theta notation is used when the function $f(n)$ is bounded both from above and below by the function $g(n)$.

Definition

$f(n) = \Theta(g(n))$ (read as f on n is theta of g of n) iff there exist two positive constants c_1 and c_2, and a positive integer n_0 such that $c_1|g(n)| \leq |f(n)| \leq c_2|g(n)|$ for all $n \geq n_0$.

From the definition it implies that the function $g(n)$ is both an upper bound and a lower bound for the function $f(n)$ for all values of n, $n \geq n_0$. In other words, $f(n)$ is such that, $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

For $f(n) = 18n + 9$, since $f(n) > 18n$ and $f(n) \leq 27n$ for $n \geq 1$, we have $f(n) = \Omega(n)$ and $f(n) = O(n)$ respectively, for $n \geq 1$. Hence $f(n) = \Theta(n)$. Again, $16n^2 + 30n - 90 = \Theta(n^2)$ and $7.2^n + 30n = \Theta(2^n)$.

Little Oh Notation (o)

Definition

$f(n) = o(g(n))$(read as f of n is little oh of g of n) iff $f(n) = O(g(n))$ and $f(n) \neq \Omega(g(n))$.

For $f(n) = 18n + 9$, we have $f(n) = O(n^2)$ but $f(n) \neq \Omega(n^2)$. Hence $f(n) = o(n^2)$. However, $f(n) \neq o(n)$.

2.7 SUBALGORITHMS

A *subalgorithm* is a complete and independently defined algorithmic module which is used (or *invoked* or *called*) by some main algorithm or by some other subalgorithm. A subalgorithm receives values, called *arguments*, from an originating (calling) algorithm; performs computations; and then sends back the result to the calling algorithm. The subalgorithm is defined independently so that it may be called by many different algorithms or called at different times in the same algorithm. The relationship between an algorithm and a subalgorithm is similar to the relationship between a main program and a subprogram in a programming language.

The main difference between the format of a subalgorithm and that of an algorithm is that the subalgorithm will usually have a heading of the form

$$\text{NAME}(\text{PAR}_1, \text{PAR}_2, \ldots, \text{PAR}_K)$$

Here NAME refers to the name of the subalgorithm which is used when the subalgorithm is called, and PAR_1, PAR_2, ..., PAR_K refer to parameters which are used to transmit data between the sub algorithm and the calling algorithm.

Another difference is that the subalgorithm will have a Return statement rather than an Exit statement; this emphasizes that control is transferred back to the calling program when the execution of the subalgorithm is completed.

Subalgorithms fall into two basic categories: *function* subalgorithms and *procedure* subalgorithms. The similarities and differences between these two types of subalgorithms will be examined below by means of examples. One major difference between the subalgorithms is that the function subalgorithm returns only a single value to the calling algorithm, whereas the procedure subalgorithm may send back more than one value.

Example 2.9

The following function subalgorithm MEAN finds the average AVE of three numbers A, B and C.

Function 2.5: MEAN(A, B, C)

1. Set AVE := (A + B + C)/3.
2. Return(AVE).

Note that MEAN is the name of the subalgorithm and A, B and C are the parameters. The Return statement includes, in parentheses, the variable AVE, whose value is returned to the calling program.

The subalgorithm MEAN is invoked by an algorithm in the same way as a function subprogram is invoked by a calling program. For example, suppose an algorithm contains the statement

$$\text{Set TEST} := \text{MEAN}(T_1, T_2, T_3)$$

where T_1, T_2 and T_3 are test scores. The argument values T_1, T_2 and T_3 are transferred to the parameters A, B, C in the subalgorithm, the subalgorithm MEAN is executed, and then the value of AVE is returned to the program and replaces MEAN(T_1, T_2, T_3) in the statement. Hence the average of T_1, T_2 and T_3 is assigned to TEST.

The following C program uses the MEAN function to calculate the average AVE of three numbers A, B and C:

Program 2.5

```
#include <stdio.h>
#include <conio.h>

void main()
{
int A,B,C;
float MEAN(int,int,int);
clrscr();

printf("Enter the values of A, B and C: ");
scanf("%d %d %d",&A,&B,&C);

printf("The average of %d, %d and %d is: %.2f",A,B,C,MEAN(A,B,C));
getch();
}

float MEAN(int T1,int T2,int T3)
{
 float AVE;
 AVE=(T1+T2+T3)/3;
 return(AVE);
}
```

Output:

```
Enter the values of A, B and C: 22
36
8
The average of 22, 36 and 8 is: 22.00
```

Example 2.10

The following procedure SWITCH interchanges the values of AAA and BBB.

Procedure 2.6: SWITCH(AAA, BBB)
1. Set TEMP := AAA, AAA := BBB and BBB := TEMP.
2. Return.

The procedure is invoked by means of a Call statement. For example, the Call statement

Call SWITCH(BEG, AUX)

has the net effect of interchanging the values of BEG and AUX. Specifically, when the procedure SWITCH is invoked, the argument of BEG and AUX are transferred to the parameters AAA and BBB, respectively; the procedure is executed, which interchanges the values of AAA and BBB; and then the new values of AAA and BBB are transferred back to BEG and AUX, respectively.

Remark: Any function subalgorithm can be easily translated into an equivalent procedure by simply adjoining an extra parameter which is used to return the computed value to the calling algorithm. For example, Function 2.1 may be translated into a procedure

MEAN(A, B, C, AVE)

where the parameter AVE is assigned the average of A, B, C. Then the statement

Call MEAN(T_1, T_2 and T_3, TEST)

also has the effect of assigning the average of T_1, T_2 and T_3 to TEST. Generally speaking, we will use procedures rather than function subalgorithms.

2.8 VARIABLES, DATA TYPES

Each variable in any of our algorithms or programs has a data type which determines the code that is used for storing its value. Four such data types follow:

1. *Character.* Here data are coded using some character code such as EBCDIC or ASCII. The 8-bit EBCDIC code of some characters appears in Fig. 2.7. A single character is normally stored in a byte.
2. *Real* (or *floating point*). Here numerical data are coded using the exponential form of the data.
3. *Integer* (or *fixed point*). Here positive integers are coded using binary representation, and negative integers by some binary variation such as 2's complement.
4. *Logical.* Here the variable can have only the value true or false; hence it may be coded using only one bit, 1 for true and 0 for false. (Sometimes the bytes 1111 1111 and 0000 0000 may be used for true and false, respectively.)

The data types of variables in our algorithms will not be explicitly stated as with computer programs but will usually be implied by the context.

Example 2.11

Suppose a 32-bit memory location X contains the following sequence of bits:

0110 1100 1100 0111 1101 0110 0110 1100

Char.	Zone	Numeric	Hex	Char.	Zone	Numeric	Hex	Char.	Zone	Numeric	Hex
				S	1110	0010	E2	*blank*	0100	0000	40
A	1100	0001	C1	T		0011	E3	.		1011	4B
B		0010	C2	U		0100	E4	<		1100	4C
C		0011	C3	V		0101	E5	(	↓	1101	4D
D		0100	C4	W		0110	E6	+	0100	1110	4E
E		0101	C5	X		0111	E7	&	0101	0000	50
F		0110	C6	Y	↓	1000	E8	$		1011	5B
G		0111	C7	Z	1110	1001	E9	*		1100	5C
H	↓	1000	C8					)	↓	1101	5D
I	1100	1001	C9	0	1111	0000	F0	;	0101	1110	5E
J	1101	0001	D1	1		0001	F1	—	0110	0000	60
K		0010	D2	2		0010	F2	/		0001	61
L		0011	D3	3		0011	F3	,		1011	6B
M		0100	D4	4		0100	F4	%		1100	6C
N		0101	D5	5		0101	F5	>	↓	1110	6E
O		0110	D6	6		0110	F6	?	0110	1111	6F
P		0111	D7	7		0111	F7	:	0111	1010	7A
Q	↓	1000	D8	8	↓	1000	F8	#		1011	7B
R	1101	1001	D9	9	1111	1001	F9	@		1100	7C
								=	0111	1110	7E

Fig. 2.7 Part of the EBCDIC Code

There is no way to know the content of the cell unless the data type of X is known.

(a) Suppose X is declared to be of character type and EBCDIC is used. Then the four characters %GO% are stored in X.

(b) Suppose X is declared to be of some other type, such as integer or real. Then an integer or real number is stored in X.

The following C program demonstrates how the same value is interpreted differently based on different associated data types:

Program 2.6

```
#include <stdio.h>
#include <conio.h>

void main()
{
char c1='1';
int c2=1;
clrscr();

printf("c1 (char) = %c \nc1's ASCII value = %d\nc2 (int) = %d",c1,c1,c2);
getch();
}

Output:
c1 (char) = 1
c1's ASCII value = 49
c2 (int) = 1
```

In the above program, variable c1 treats 1 as a character while variable c2 treats 1 as an integer.

Local and Global Variables

The organization of a computer program into a main program and various subprograms has led to the notion of local and global variables. Normally, each program module contains its own list of variables, called *local variables*, which can be accessed only by the given program module. Also, subprogram modules may contain parameters, variables which transfer data between a subprogram and its calling program.

Example 2.12

Consider the procedure SWITCH(AAA, BBB) in Example 2.10. The variables AAA and BBB are parameters; they are used to transfer data between the procedure and a calling algorithm. On the other hand, the variable TEMP in the procedure is a local variable. It "lives" only in the procedure; i.e., its value can be accessed and changed only during the execution of the procedure. In fact, the name TEMP may be used for a variable in any other module and the use of the name will not interfere with the execution of the procedure SWITCH.

Program 2.7

```
/* C implementation of the SWITCH procedure */
#include <stdio.h>
#include <conio.h>

int AAA=10;
int BBB=20;
void SWITCH(void);

void main()
{
clrscr();

printf("AAA = %d BBB = %d",AAA,BBB);
SWITCH();
printf("\nAfter calling SWITCH procedure, AAA = %d BBB = %d",AAA,BBB);
getch();
}

void SWITCH(void)
{
 int TEMP;
 TEMP=AAA;
 AAA=BBB;
 BBB=TEMP;
 return;
}
```

Output:

```
AAA = 10 BBB = 20
After calling SWITCH procedure, AAA = 20 BBB = 10
```

Language designers realized that it would be convenient to have certain variables which can be accessed by some or even all the program modules in a computer program. Variables that can be accessed by all program modules are called *global* variables, and variables that can be accessed by some program modules are called *nonlocal* variables. Each programming language has its own syntax for declaring such variables. For example, FORTRAN uses a COMMON statement to declare global variables, and Pascal uses scope rules to declare global and nonlocal variables.

Accordingly, there are two basic ways for modules to communicate with each other:

1. *Directly*, by means of well-defined parameters
2. *Indirectly*, by means of non local and global variables

The indirect change of the value of a variable in one module by another module is called a *side effect.* Readers should be very careful when using nonlocal and global variables, since errors caused by side effects may be difficult to detect.

SOLVED PROBLEMS

Mathematical Notations and Functions

2.1 Find (a) $\lfloor 7.5 \rfloor$, $\lfloor -7.5 \rfloor$, $\lfloor -18 \rfloor$, $\lfloor \sqrt{30} \rfloor$, $\lfloor \sqrt[3]{30} \rfloor$, $\lfloor \pi \rfloor$; and (b) $\lceil 7.5 \rceil$, $\lceil -7.5 \rceil$, $\lceil -18 \rceil$, $\lceil \sqrt{30} \rceil$, $\lceil \sqrt[3]{30} \rceil$, $\lceil \pi \rceil$.

(a) By definition, $\lfloor x \rfloor$ denotes the greatest integer that does not exceed x, called the floor of x. Hence,

$$\lfloor 7.5 \rfloor = 7 \quad \lfloor -7.5 \rfloor = -8 \quad \lfloor -18 \rfloor = -18$$

$$\lfloor \sqrt{30} \rfloor = 5 \quad \lfloor \sqrt[3]{30} \rfloor = 3 \quad \lfloor \pi \rfloor = 3$$

(b) By definition, $\lceil x \rceil$ denotes the least integer that is not less than x, called the ceiling of x. Hence,

$$\lceil 7.5 \rceil = 8 \quad \lceil -7.5 \rceil = -7 \quad \lceil -18 \rceil = -18$$

$$\lceil \sqrt{30} \rceil = 6 \quad \lfloor \sqrt[3]{30} \rfloor = 4 \quad \lceil \pi \rceil = 4$$

2.2 (a) Find 26 (mod 7), 34 (mod 8), 2345 (mod 6), 495 (mod 11).
(b) Find –26 (mod 7), –2345 (mod 6), –371 (mod 8), –39 (mod 3).
(c) Using arithmetic modulo 15, evaluate 9 + 13, 7 + 11, 4 – 9, 2 – 10.

(a) Since k is positive, simply divide k by the modulus M to obtain the remainder r. Then $r = k \pmod{M}$

Thus

$$5 = 26 \pmod 7 \quad 2 = 34 \pmod 8 \quad 5 = 2345 \pmod 6 \quad 0 = 495 \pmod{11}$$

(b) When k is negative, divide $|k|$ by the modulus to obtain the remainder r'. Then $k \equiv -r' \pmod{M}$. Hence $k \pmod{M} = M - r'$ when $r' \neq 0$. Thus

$$-26 \pmod 7 = 7 - 5 = 2 \qquad -371 \pmod 8 = 8 - 3 = 5$$
$$-2345 \pmod 6 = 6 - 5 = 1 \qquad -39 \pmod 3 = 0$$

(c) Use $a \pm M \equiv a \pmod{M}$:

$$9 + 13 = 22 \equiv 22 - 15 = 7 \qquad 7 + 11 = 18 \equiv 18 - 15 = 3$$
$$4 - 9 = -5 \equiv -5 + 15 = 10 \qquad 2 - 10 = -8 \equiv -8 + 15 = 7$$

2.3 List all the permutations of the numbers 1, 2, 3, 4.

Note first that there are $4! = 24$ such permutations:

1234	1243	1324	1342	1423	1432
2134	2143	2314	2341	2413	2431
3124	3142	3214	3241	3412	3421
4123	4132	4213	4231	4312	4321

Observe that the first row contains the six permutations beginning with 1, the second row those beginning with 2, and so on.

2.4 Find: (a) 2^{-5}, $8^{2/3}$, $25^{-3/2}$; (b) $\log_2 32$, $\log_{10} 1000$, $\log_2 (1/16)$; (c) $\lfloor \log_2 1000 \rfloor$, $\lfloor \log_2 0.01 \rfloor$.

(a) $2^{-5} = 1/2^5 = 1/32$; $8^{2/3} = (\sqrt[3]{8}\,)^2 = 2^2 = 4$; $25^{-3/2} = 1/25^{3/2} = 1/5^3 = 1/125$.

(b) $\log_2 32 = 5$ since $2^5 = 32$; $\log_{10} 1000 = 3$ since $10^3 = 1000$; $\log_2(1/16) = -4$ since $2^{-4} = 1/2^4 = 1/16$.

(c) $\lfloor \log_2 1000 \rfloor = 9$ since $2^9 = 512$ but $2^{10} = 1024$;
$\lfloor \log_2 0.01 \rfloor = -7$ since $2^{-7} = 1/128 < 0.01 < 2^{-6} = 1/64$.

2.5 Plot the graphs of the exponential function $f(x) = 2^x$, the logarithmic function $g(x) = \log_2 x$ and the linear function $h(x) = x$ on the same coordinate axis. (a) Describe a geometric property of the graphs $f(x)$ and $g(x)$. (b) For any positive number c, how are $f(c)$, $g(c)$ and $h(c)$ related?

Figure 2.8 pictures the three functions.

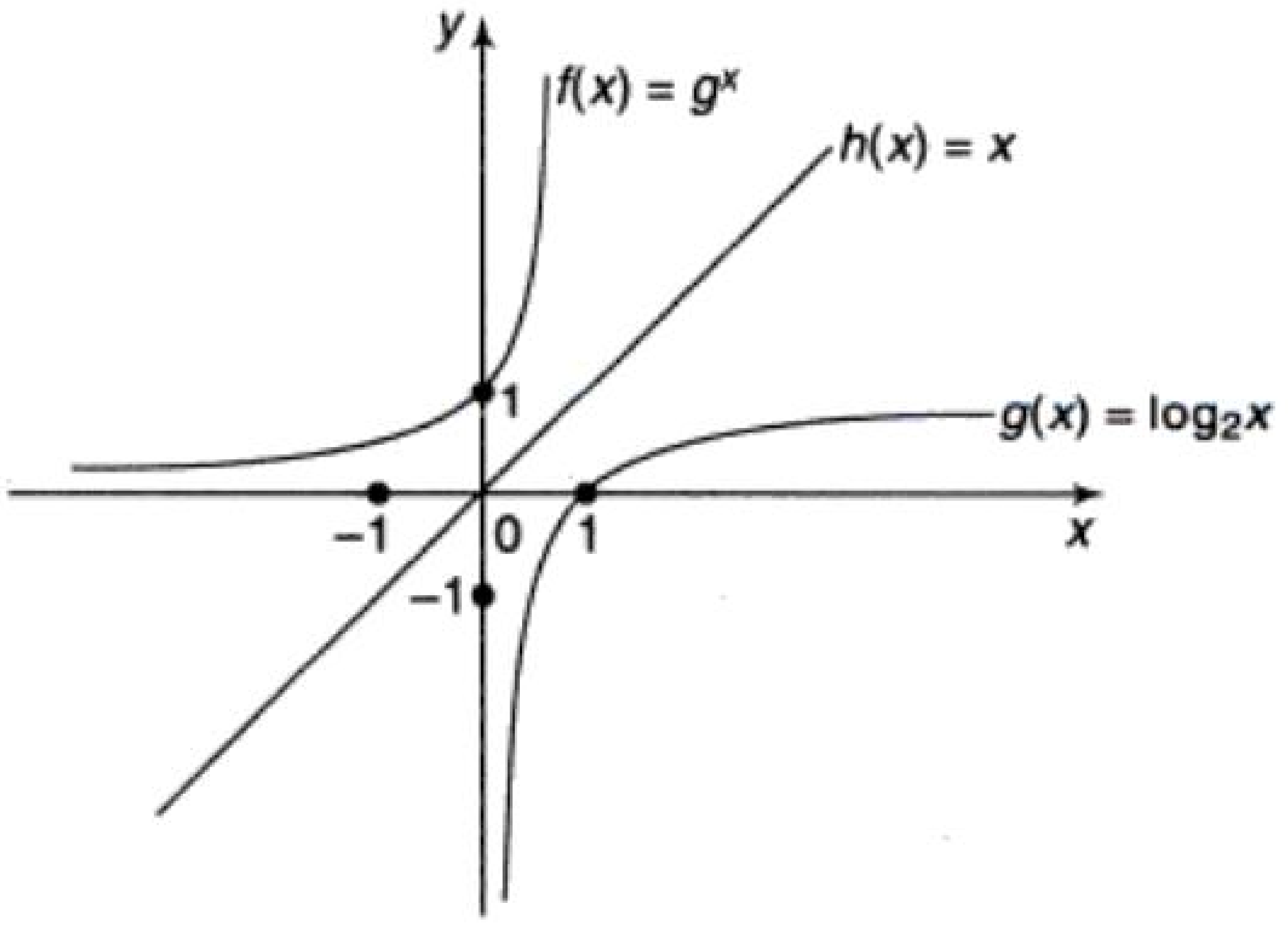


Fig. 2.8

(a) Since $f(x) = 2^x$ and $g(x) = \log_2 x$ are inverse functions, they are symmetric with respect to the line $y = x$.

(b) For any positive number c, we have.

$$g(c) < h(c) < f(c)$$

In fact, as c increases in value, the vertical distances between the functions,

$$h(c) - g(c) \qquad \text{and} \qquad f(c) - h(c),$$

increase in value. Moreover, the logarithmic function $g(x)$ grows very slowly compared with the linear function $h(x)$, and the exponential function $f(x)$ grows very quickly compared with $h(x)$.

Algorithms, Complexity

2.6 Consider Algorithm 2.3, which finds the location LOC and the value MAX of the largest element in an array DATA with n elements. Consider the complexity function $C(n)$, which measures the number of times LOC and MAX are updated in Step 3. (The number of comparisons is independent of the order of the elements in DATA.)

(a) Describe and find $C(n)$ for the worst case.

(b) Describe and find $C(n)$ for the best case.

(c) Find $C(n)$ for the average case when $n = 3$, assuming all arrangements of the elements in DATA are equally likely.

(a) The worst case occurs when the elements of DATA are in increasing order, where each comparison of MAX with DATA[K] forces LOC and MAX to be updated. In this case, $C(n) = n - 1$.

(b) The best case occurs when the largest element appears first and so when the comparison of MAX with DATA[K] never forces LOC and MAX to be updated. Accordingly, in this case, $C(n) = 0$.

(c) Let 1, 2 and 3 denote, respectively, the largest, second largest and smallest elements of DATA. There are six possible ways the elements can appear in DATA, which correspond to the $3! = 6$ permutations of 1, 2, 3. For each permutation p, let n_p denote the number of times LOC and MAX are updated when the algorithm is executed with input p. The six permutations p and the corresponding values n_p follow:

Permutation p:	123	132	213	231	312	321
Value of n_p:	0	0	1	1	1	2

Assuming all permutations p are equally likely,

$$C(3) = \frac{0+0+1+1+1+2}{6} = \frac{5}{6}$$

(The evaluation of the average value of $C(n)$ for arbitrary n lies beyond the scope of this text. One purpose of this problem is to illustrate the difficulty that may occur in finding the complexity of the average case of an algorithm.)

2.7 Suppose Module A requires M units of time to be executed, where M is a constant. Find the complexity $C(n)$ of each algorithm, where n is the size of the input data and b is a positive integer greater than 1.

(a) **Algorithm P2.7A:**

```
1. Repeat for I = 1 to N:

2. Repeat for J = 1 to N:
3.       Repeat for K = I to N:
4.             Module A.
```

[End of Step 3 loop.]
[End of Step 2 loop.]
[End of Step 1 loop.]
5. Exit.

(b) **Algorithm P2.7B:**

1. Set J := 1.
2. Repeat Steps 3 and 4 while J ≤ N:
3. Module A.
4. Set J := B × J.
[End of Step 2 loop.]
5. Exit.

Observe that the algorithms use N for n and B for b.)

(a) Here $C(n) = \sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{k=1}^{n} M$

The number of times M occurs in the sum is equal to the number of triplets (i, j, k), where i, j, k are integers from 1 to n inclusive. There are n^3 such triplets. Hence

$$C(n) = M'n^3 = O(n^3)$$

(b) Observe that the values of the loop index J are the powers of b:

$$1, b, b^2, b^3, b^4, \ldots$$

Therefore, Module A will be repeated exactly T times, where T is the first exponent such that

$$b^T > n$$

Hence, $T = \lfloor \log_b n \rfloor + 1$

Accordingly, $C(n) = MT = O(\log_b n)$

2.8 (a) Write a procedure FIND(DATA, N, LOC1, LOC2) which finds the location LOC1 of the largest element and the location LOC2 of the second largest element in an array DATA with $n > 1$ elements.

(b) Why not let FIND also find the values of the largest and second largest elements?

(a) The elements of DATA are examined one by one. During the execution of the procedure, FIRST and SECOND will denote, respectively, the values of the largest and second largest elements that have already been examined. Each new element DATA[K] is tested as follows. If

$$\text{SECOND} \le \text{FIRST} < \text{DATA[K]}$$

then FIRST becomes the new SECOND element and DATA[K] becomes the new FIRST element. On the other hand, if

$$\text{SECOND} < \text{DATA[K]} \le \text{FIRST}$$

then DATA[K] becomes the new SECOND element. Initially, set FIRST := DATA[1] and SECOND := DATA[2], and check whether or not they are in the right order. A formal presentation of the procedure follows:

Procedure P2.8: FIND(DATA, N, LOC1, LOC2)

1. Set FIRST := DATA[1], SECOND := DATA[2], LOC1 := 1, LOC2 := 2.
2. [Are FIRST and SECOND initially correct?]
   If FIRST < SECOND, then:
   (a) Interchange FIRST and SECOND,
   (b) Set LOC1 := 2 and LOC2 := 1.

   [End of If structure.]
3. Repeat for K = 3 to N:
   If FIRST < DATA[K], then:
   (a) Set SECOND := FIRST and FIRST := DATA[K].
   (b) Set LOC2 := LOC1 and LOC1 := K.
   Else if SECOND < DATA[K], then:
   Set SECOND := DATA[K] and LOC2 := K.
   [End of If structure.]
   [End of loop.]
4. Return.

(b) Using additional parameters FIRST and SECOND would be redundant, since LOC1 and LOC2 automatically tell the calling program that DATA[LOC1] and DATA[LOC2] are, respectively, the values of the largest and second largest elements of DATA.

Program 2.8

```
/* C implementation of Procedure P 2.8 */
#include <stdio.h>
#include <conio.h>

int DATA[10]={22,65,1,99,32,17,74,49,33,2};
int N, LOC1, LOC2, FIRST, SECOND;

void main()
{
void FIND(int [],int,int,int);
clrscr();
N=10;
LOC1=-1;
LOC2=-1;

FIND(DATA,N,LOC1,LOC2);
```

```
printf("FIRST = %d, LOC1 = %d, SECOND = %d, LOC2 = %d",FIRST,LOC1,SECOND,
LOC2);

getch();
}

void FIND(int LIST[],int LEN,int L1,int L2)
{
int TEMP,K;
FIRST=LIST[0];
SECOND=LIST[1];
L1=0;
L2=1;

if(FIRST<SECOND)
{
 TEMP=FIRST;
 FIRST=SECOND;
 SECOND=TEMP;
 L2=0;
 L1=1;
}
for(K=2;K<LEN;K++)
{
 if(FIRST<LIST[K])
 {
  SECOND=FIRST;
  FIRST=LIST[K];
  L2=L1;
  L1=K;
 }
 else if(SECOND<LIST[K])
 {
  SECOND=LIST[K];
  L2=K;
 }
}
LOC1=L1;
LOC2=L2;
}
```

Output:

```
FIRST = 99, LOC1 = 3, SECOND = 74, LOC2 = 6
```

2.9 An integer $n > 1$ is called a *prime* number if its only positive divisors are 1 and n; otherwise, n is called a *composite* number. For example, the following are the prime numbers less than 20:

$$2, 3, 5, 7, 11, 13, 17, 19$$

If $n > 1$ is not prime, i.e., if n is composite, then n must have a divisor $k \neq 1$ such that $k \leq \sqrt{n}$ or, in other words, $k^2 \leq n$.

Suppose we want to find all the prime numbers less than a given number m, such as 30. This can be done by the "sieve method," which consists of the following steps. First list the 30 numbers:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15
16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30

Cross out 1 and the multiples of 2 from the list as follows:

~~1~~, 2, 3, ~~4~~, 5, ~~6~~, 7, ~~8~~, 9, ~~10~~, 11, ~~12~~, 13, ~~14~~, 15
~~16~~, 17, ~~18~~, 19, ~~20~~, 21, ~~22~~, 23, ~~24~~, 25, ~~26~~, 27, ~~28~~, 29, ~~30~~

Since 3 is the first number following 2 that has not been eliminated, cross out the multiples of 3 from the list as follows:

~~1~~, 2, 3, ~~4~~, 5, ~~6~~, 7, ~~8~~, ~~9~~, ~~10~~, 11, ~~12~~, 13, ~~14~~, ~~15~~
~~16~~, 17, ~~18~~, 19, ~~20~~, ~~21~~, ~~22~~, 23, ~~24~~, 25, ~~26~~, ~~27~~, ~~28~~, 29, ~~30~~

Since 5 is the first number following 3 that has not been eliminated, cross out the multiples of 5 from the list as follows:

~~1~~, 2, 3, ~~4~~, 5, ~~6~~, 7, ~~8~~, ~~9~~, ~~10~~, 11, ~~12~~, 13, ~~14~~, ~~15~~
~~16~~, 17, ~~18~~, 19, ~~20~~, ~~21~~, ~~22~~, 23, ~~24~~, ~~25~~, ~~26~~, ~~27~~, ~~28~~, 29, ~~30~~

Now 7 is the first number following 5 that has not been eliminated, but $7^2 > 30$. This means the algorithm is finished and the numbers left in the list are the primes less than 30:

$$2, 3, 5, 7, 11, 13, 17, 19, 23, 29$$

Translate the sieve method into an algorithm to find all prime numbers less than a given number n.

First define an array A such that

$$A[1] = 1, \quad A[2] = 2, \quad A[3] = 3, \quad A[4] = 4, \ \ldots, \ A[N - 1] = N - 1, \quad A[N] = N$$

We cross out an integer L from the list by assigning A[L] = 1. The following procedure CROSSOUT tests whether A[K] = 1, and if not, it sets

A[2K] = 1, A[3K] = 1, A[4K] = 1, ...

That is, it eliminates the multiples of K from the list

Procedure P2.9A: CROSSOUT(A, N, K)

1. If A[K] = 1, then: Return.
2. Repeat for L = 2K to N by K:
   Set A[L]:= 1.
   [End of loop.]
3. Return.

The sieve method can now be simply written:

Algorithm P2.9B: This algorithm prints the prime numbers less than N.

1. [Initialize array A.] Repeat for K = 1 to N:
   Set A[K] := K.
2. [Eliminate multiples of K.] Repeat for K = 2 to $\sqrt{N}$.
   Call CROSSOUT(A, N, K).
3. [Print the primes.] Repeat for K = 2 to N:
   If A[K] ≠ 1, then: Write: A[K].
4. Exit.

Program 2.9

```
/* C implementation of Algorithm P2.9B*/
#include <stdio.h>
#include <conio.h>
#include <math.h>

int A[100];
void CROSSOUT(int,int);

void main()
{
int K,N;
clrscr();

printf("Enter the value of N: ");
scanf("%d",&N);

A[0]=-1;
for(K=1;K<=N;K++)
A[K]=K;

for(K=2;K<=sqrt(N);K++)
CROSSOUT(N,K);

for(K=2;K<=N;K++)
if(A[K]!=1)
```

```
printf("%d ",A[K]);
getch();
}

void CROSSOUT(int n,int k)
{
 int L;
 if(A[k]==1)
 return;
 for(L=2*k;L<=n;L=L+k)
 A[L]=1;
 return;
}
```

```
Output:
Enter the value of N: 20
2 3 5 7 11 13 17 19
```

2.10 Suppose $P(n) = a_0 + a_1n + a_2n^2 + \cdots + a_mn^m$; that is, suppose degree $P(n) = m$. Prove that $P(n) = O(n^m)$.

Let $b_0 = |a_0|$, $b_1 = |a_1|$, ..., $b_m = |a_m|$. Then, for $n \geq 1$,

$$P(n) \leq b_0 + b_1n + b_2n^2 + \cdots + b_mn^m$$

$$= \left(\frac{b_0}{n^m} + \frac{b_1}{n^{m-1}} + \cdots + b_m\right)n^m$$

$$\leq (b_0 + b_1 + \cdots + b_m)n^m = Mn^m$$

where $M = |a_0| + |a_1| + \cdots + |a_m|$. Hence $P(n) = O(n^m)$.

For example, $5x^3 + 3x = O(x^3)$ and $x^5 - 4\ 000\ 000x^2 = O(x^5)$.

2.11 Suppose that $T_1(n)$ and $T_2(n)$ are the time complexities of two program fragments P_1 and P_2 where $T_1(n) = O(f(n))$ and $T_2(n) = O(g(n))$, what is the time complexity of program fragment P_1 followed by P_2?

The time complexity of program fragment P_1 followed by P_2 is given by $T_1(n) + T_2(n)$. To obtain $T_1(n) + T_2(n)$, we have

$T_1(n) \leq c \cdot f(n)$ for some positive number c and positive integer n_1, such that $n \geq n_1$ and

$T_2(n) \leq d \cdot g(n)$ for some positive number d and positive integer n_2, such that $n \geq n_2$

Let $n_0 = \max(n_1, n_2)$. Then,

$T_1(n) + T_2(n) \leq c \cdot f(n) + d \cdot g(n)$, for $n > n_0$

(i.e.) $T_1(n) + T_2(n) \leq (c + d) \max(f(n), g(n))$ for $n > n_0$

Hence $T_1(n) + T_2(n) = O(\max(f(n), g(n)))$. (This result is referred to as Rule of Sums of O notation)

2.12 Given: $T_1(n) = O(f(n))$ and $T_2(n) = O(g(n))$. Find $T_1(n) \cdot T_2(n)$.

$T_1(n) \le c \cdot f(n)$ for some positive number c and positive integer n_1, such that $n \ge n_1$ and $T_2(n) \le d \cdot g(n)$ for some positive number d and positive integer n_2 such that $n \ge n_2$.

Hence, $T_1(n) \cdot T_2(n) \le c \cdot f(n) \cdot d \cdot g(n)$
$\le k \cdot f(n) \cdot g(n)$

Therefore $T_1(n) \cdot T_2(n) = O(f(n) \cdot g(n))$ (This result is referred to as Rule of Products of O notation)

Variables, Data Types

2.13 Describe briefly the difference between local variables, parameters and global variables.

Local variables are variables which can be accessed only within a particular program or subprogram. Parameters are variables which are used to transfer data between a subprogram and its calling program. Global variables are variables which can be accessed by all of the program modules in a computer program. Each programming language which allows global variables has its own syntax for declaring them.

2.14 Suppose NUM denotes the number of records in a file. Describe the advantages in defining NUM to be a global variable. Describe the disadvantages in using global variables in general.

Many of the procedures will process all the records in the file using some type of loop. Since NUM will be the same for all these procedures, it would be advantageous to have NUM declared a global variable. Generally speaking, global and nonlocal variables may lead to errors caused by side effects, which may be difficult to detect.

2.15 Suppose a 32 bit memory location AAA contains the following sequence of bits:

0100 1101 1100 0001 1110 1001 0101 1101

Determine the data stored in AAA.

There is no way of knowing the data stored in AAA unless one knows the data type of AAA. If AAA is a character variable and the EBCDIC code is used for storing data, then (AZ) is stored in AAA. If AAA is an integer variable, then the integer with the above binary representation is stored in AAA.

2.16 Mathematically speaking, integers may also be viewed as real numbers. Give some reasons for having two different data types.

The arithmetic for integers, which are stored using some type of binary representation, is much simpler than the arithmetic for real numbers, which are stored using some type of exponential form. Also, certain round-off errors occurring in real arithmetic do not occur in integer arithmetic.

SUPPLEMENTARY PROBLEMS

Mathematical Notations and Functions

2.1 Find (a) $\lfloor 3.4 \rfloor$, $\lfloor -3.4 \rfloor$, $\lfloor -7 \rfloor$, $\lfloor \sqrt{75} \rfloor$, $\lfloor \sqrt[3]{75} \rfloor$, $\lfloor e \rfloor$; (b) $\lceil 3.4 \rceil$, $\lceil -3.4 \rceil$, $\lceil -7 \rceil$, $\lceil \sqrt{75} \rceil$, $\lceil \sqrt[3]{75} \rceil$, $\lceil e \rceil$.

2.2 (a) Find 48 (mod 5), 48 (mod 7), 1397 (mod 11), 2468 (mod 9).
(b) Find –48 (mod 5), –152 (mod 7), –358 (mod 11), –1326 (mod 13).
(c) Using arithmetic modulo 13, evaluate

$$9 + 10, \quad 8 + 12, \quad 3 + 4, \quad 3 - 4, \quad 2 - 7, \quad 5 - 8$$

2.3 Find (a) $|3 + 8|$, $|3 - 8|$, $|-3 + 8|$, $|-3 - 8|$; (b) 7!, 8!, 14!/12!, 15!/16!

2.4 Find (a) 3^{-4}, $4^{7/2}$, $27^{-2/3}$; (b) $\log_2 64$, $\log_{10} 0.001$, $\log_2 (1/8)$; (c) $\lfloor \lg 1\,000\,000 \rfloor$, $\lfloor \lg 0.001 \rceil$.

Algorithms, Complexity

2.5 Consider the complexity function $C(n)$ which measures the number of times LOC is updated in Step 3 of Algorithm 2.3. Find $C(n)$ for the average case when $n = 4$, assuming all arrangements of the given four elements are equally likely. (Compare with Solved Problem 2.6.)

2.6 Consider Procedure P2.8, which finds the location LOC1 of the largest element and the location LOC2 of the second largest element in an array DATA with $n > 1$ elements. Let $C(n)$ denote the number of comparisons during the execution of the procedure.
(a) Find $C(n)$ for the best case.
(b) Find $C(n)$ for the worst case.
(c) Find $C(n)$ for the average case for $n = 4$, assuming all arrangements of the given elements in DATA are equally likely.

2.7 Repeat Supplementary Problem 2.6, except now let $C(n)$ denote the number of times the values of FIRST and SECOND (or LOC1 and LOC2) must be updated.

2.8 Suppose the running time of a Module A is a constant M. Find the order of magnitude of the complexity function $C(n)$ which measures the execution time of each of the following algorithms, where n is the size of the input data (denoted by N in the algorithms).

(a) **Procedure P2.8A:**

```
1. Repeat for I = 1 to N:
2.     Repeat for J = 1 to I:
3.         Repeat for K = 1 to J:
4.             Module A.
           [End of Step 3 loop.]
       [End of Step 2 loop.]
   [End of Step 1 loop.]
5. Exit.
```

(b) **Procedure P2.8B:**

1. Set J := N.
2. Repeat Steps 3 and 4 while J > 1.
3. Module A.
4. Set J := J/2.
   [End of Step 2 loop.]
5. Return.

2.9. Find the order of complexity of the following program.

```
fun(n)
  {if(n<=2)return (1); else
   return ((fun(n-1)*fun(n-2));}
```

PROGRAMMING PROBLEMS

2.1 Write a function subprogram DIV(J, K), where J and K are positive integers such that DIV(J, K) = 1 if J divides K but otherwise DIV(J, K) = 0. (For example, DIV(3, 15) = 1 but DIV(3, 16) = 0.)

2.2 Write a program using DIV(J, K) which reads a positive integer N > 10 and determines whether or not N is a prime number. (*Hint:* N is prime if (i) DIV(2, N) = 0 (i.e., N is odd) and (ii) DIV(K, N) = 0 for all odd integers K where $1 < K^2 \leq N$.)

2.3 Translate Procedure P2.8 into a C program; i.e., write a program which finds the location LOC1 of the largest element and the location LOC2 of the second largest element in an array DATA with N > 1 elements. Test the program using 70, 30, 25, 80, 60, 50, 30, 75, 25, and 60.

2.4 Translate the sieve method for finding prime numbers, described in Solved Problem 2.9, into a C program to find the prime numbers less than N. Test the program using (a) N = 1000 and (b) N = 10 000.

2.5 Let C denote the number of times LOC is updated using Algorithm 2.3 to find the largest element in an array A with N elements.

(a) Write a subprogram COUNT(A, N, C) which finds C.

(b) Write a Procedure P2.27 which (i) reads N random numbers between 0 and 1 into an array A and (ii) uses COUNT(A, N, C) to find the value of C.

(c) Write a program which repeats Procedure P2.27 1000 times and finds the average of the 1000 C's.

(i) Test the program for N = 3 and compare the result with the value obtained in Solved Problem 2.6.

(ii) Test the program for N = 4 and compare the result with the value in Supplementary Problem 2.5.

2.6 Write a pseudocode for an algorithm that receives an integer, prints the number of digits and the sum of digits in the integer.

MULTIPLE CHOICE QUESTIONS

2.1 ______ of a set of n elements is an arrangement of the elements in a given order.
(a) Combination (b) Permutation
(c) Exponent (d) Logarithm

2.2 There are _______ permutations of a set of n elements.
(a) $n!$ (b) n
(c) $n2$ (d) $n+1$

2.3 Logarithms to the base 10 are called _______ logarithms.
(a) Natural (b) Simple
(c) Common (d) Binary

2.4 The first part of an algorithm tells the _______ of the algorithm.
(a) Logic (b) Process
(c) Purpose (d) Steps

2.5 Each step of an algorithm may contain its _______ in brackets.
(a) Purpose (b) Functions
(c) Steps (d) Comments

2.6 The term _______ will be used for an independent algorithmic module which solves a particular problem.
(a) Program (b) Logic
(c) Procedure (d) Name

2.7 _______ logic employs a number of conditions which lead to a selection of one out of several alternative modules.
(a) Selection (b) Sequential
(c) Iteration (d) Procedural

2.8 A structure is of the form:
If condition, then:
[Module A]
Else:
[Module B]
[End of if structures]
What is this structure?
(a) Multiple Alternative
(b) Double Alternative
(c) Single Alternative
(d) None of the above

2.9 _______ loop uses a condition to control the loop.
(a) Repeat-for (b) Repeat
(c) Continue (d) Repeat-while

2.10 In complexity theory, _______ case refers to the expected value of $f(n)$.
(a) Average (b) Best
(c) Worst (d) Good

2.11 $O(n^2)$ is the complexity of which searching and sorting algorithm?
(a) Binary search (b) Linear search
(c) Merge sort (d) Bubble sort

2.12 The _______ notation is used when the function $g(n)$ defines a lower bound for the function $f(n)$.
(a) Omega (b) Big O
(c) Theta (d) Little Oh

2.13 Each program module contains its own list of variables called _______
(a) Global (b) Local
(c) Search (d) Binary

2.14 _______ function of C is used to allocate a block of memory.
(a) malloc() (b) calloc()
(c) free (d) realloc()

2.15 Variables that can be accessed by some program modules are called _______ variables.
(a) Global (b) Local
(c) Search (d) Nonlocal

ANSWERS TO MULTIPLE CHOICE QUESTIONS

2.1 (b) **2.2** (a) **2.3** (c) **2.4** (b) **2.5** (d) **2.6** (c) **2.7** (a)
2.8 (b) **2.9** (d) **2.10** (a) **2.11** (d) **2.12** (a) **2.13** (b) **2.14** (a)
2.15 (d)

Chapter 3

String Processing

3.1 INTRODUCTION

Historically, computers were first used for processing numerical data. Today, computers are frequently used for processing nonnumerical data, called *character data*. This chapter discusses how such data are stored and processed by the computer.

One of the primary applications of computers today is in the field of word processing. Such processing usually involves some type of pattern matching, as in checking to see if a particular word S appears in a given text T. We discuss this pattern matching problem in detail and, moreover, present two different pattern matching algorithms. The complexity of these algorithms is also investigated.

Computer terminology usually uses the term "string" for a sequence of characters rather than the term "word," since "word" has another meaning in computer science. For this reason, many texts sometimes use the expression "string processing," "string manipulation" or "text editing" instead of the expression "word processing."

The material in this chapter is essentially tangential and independent of the rest of the text. Accordingly, the reader or instructor may choose to omit this chapter on a first reading or cover this chapter at a later time.

3.2 BASIC TERMINOLOGY

Each programming language contains a *character set* that is used to communicate with the computer. This set usually includes the following:

Alphabet:	A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
Digits:	0 1 2 3 4 5 6 7 8 9
Special characters:	+ − / * ( ) , . $ = ' □

The set of special characters, which includes the blank space, frequently denoted by □, varies somewhat from one language to another.

A finite sequence S of zero or more characters is called a *string*. The number of characters in a string is called its *length*. The string with zero characters is called the *empty string* or the *null string*. Specific strings will be denoted by enclosing their characters in single quotation marks. The quotation marks will also serve as string delimiters. Hence

'THE END' 'TO BE OR NOT TO BE' '□□'

are strings with lengths 7, 18, 0 and 2, respectively. We emphasize that the blank space is a character and hence contributes to the length of the string. Sometimes the quotation marks may be omitted when the context indicates that the expression is a string.

Let S_1 and S_2 be strings. The string consisting of the characters of S_1 followed by the characters of S_2 is called the *concatenation* of S_1 and S_2; it will be denoted by $S_1//S_2$. For example,

'THE' // 'END' = 'THEEND' but 'THE' // ' □ ' // ' END = 'THE END'

Clearly the length of $S_1//S_2$ is equal to the sum of the lengths of the strings S_1 and S_2.

A string Y is called a *substring* of a string S if there exist strings X and Z such that

$$S = X//Y//Z$$

If X is an empty string, then Y is called an *initial substring* of S, and if Z is an empty string then Y is called a *terminal substring* of S. For example,

'BE OR NOT' is a substring of 'TO BE OR NOT TO BE'
'THE' is an initial substring of 'THE END'

Clearly, if Y is a substring of S, then the length of Y cannot exceed the length of S.

Remark: Characters are stored in the computer using either a 6-bit, a 7-bit or an 8-bit code. The unit equal to the number of bits needed to represent a character is called a *byte*. However, unless otherwise stated or implied, a byte usually means 8 bits. A computer which can access an individual byte is called a *byte-addressable machine*.

3.3 STORING STRINGS

Generally speaking, strings are stored in three types of structures: (1) fixed-length structures, (2) variable-length structures with fixed maximums and (3) linked structures. We discuss each type of structure separately, giving its advantages and disadvantages.

Record-Oriented, Fixed-Length Storage

In fixed-length storage each line of print is viewed as a record, where all records have the same length, i.e., where each record accommodates the same number of characters. Since earlier systems used to input on terminals with 80-column images or using 80-column cards, we will assume our records have length 80 unless otherwise stated or implied.

Example 3.1

Suppose the input consists of the program in Fig. 3.1. Using a record-oriented, fixed-length storage medium, the input data will appear in memory as pictured in Fig. 3.2, where we assume that 200 is the address of the first character of the program.

The main advantages of the above way of storing strings are:

1. The ease of accessing data from any given record
2. The ease of updating data in any given record (as long as the length of the new data does not exceed the record length)

```
 /*PROGRAM PRINTING TWO INTEGERS IN DECREASING ORDER*/
void main ()
{ int J,K;
scanf("%d %d",&J,&K);
if(J>K)
printf("%d %d\n",J,K);
if(J<K)
printf("%d %d\n",K,J);
}
```

Fig. 3.1 Input Data

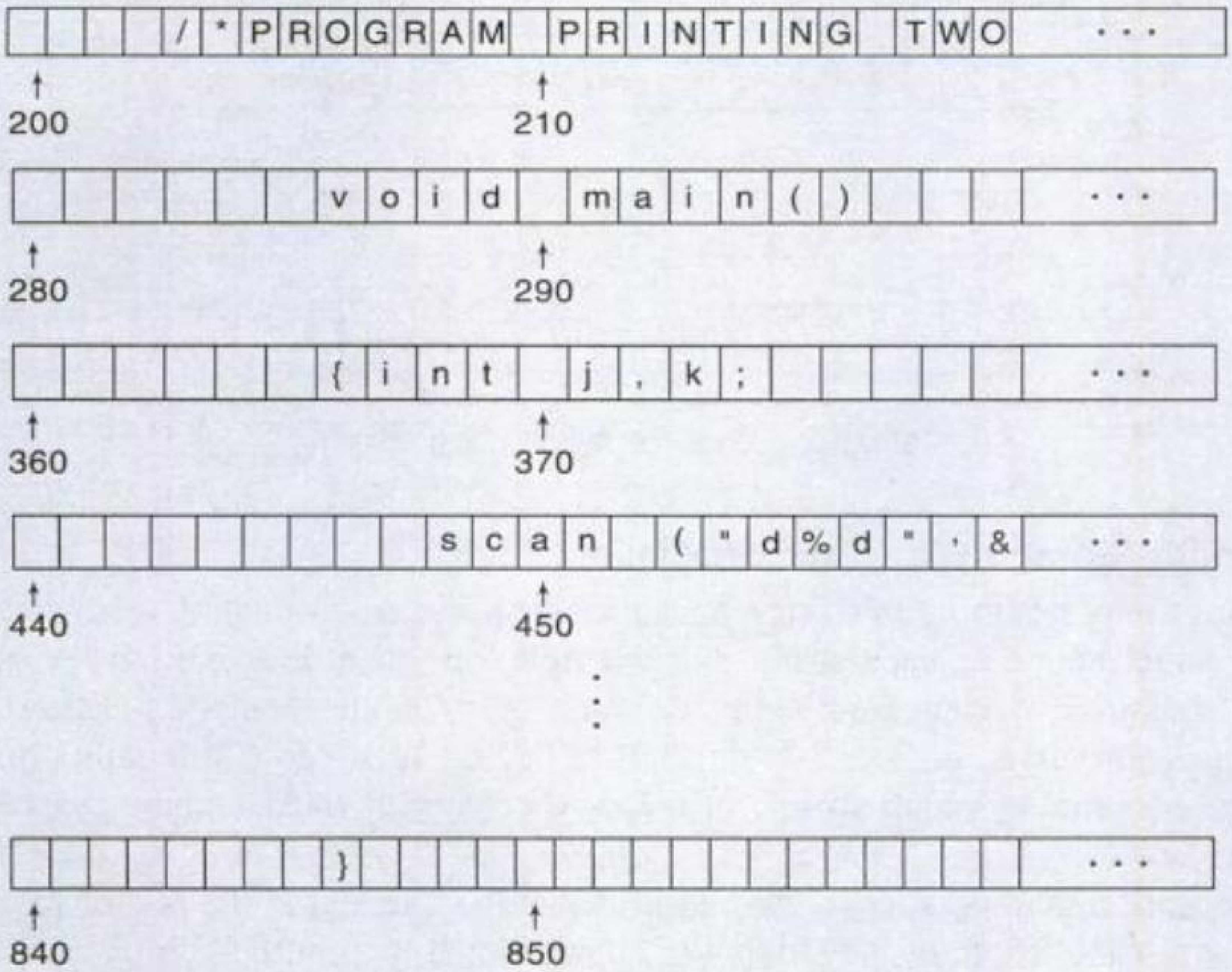


Fig. 3.2 Records Stored Sequentially in the Computer

The main disadvantages are:

1. Time is wasted reading an entire record if most of the storage consists of inessential blank spaces.
2. Certain records may require more space than available.
3. When the correction consists of more or fewer characters than the original text, changing a misspelled word requires the entire record to be changed.

Remark: Suppose we wanted to insert a new record in Example 3.1. This would require that all succeeding records be moved to new memory locations. However, this disadvantage can be easily remedied as indicated in Fig. 3.3. That is, one can use a linear array POINT which gives the address of each successive record, so that the records need not be stored in consecutive locations in memory. Accordingly, inserting a new record will require only an updating of the array POINT.

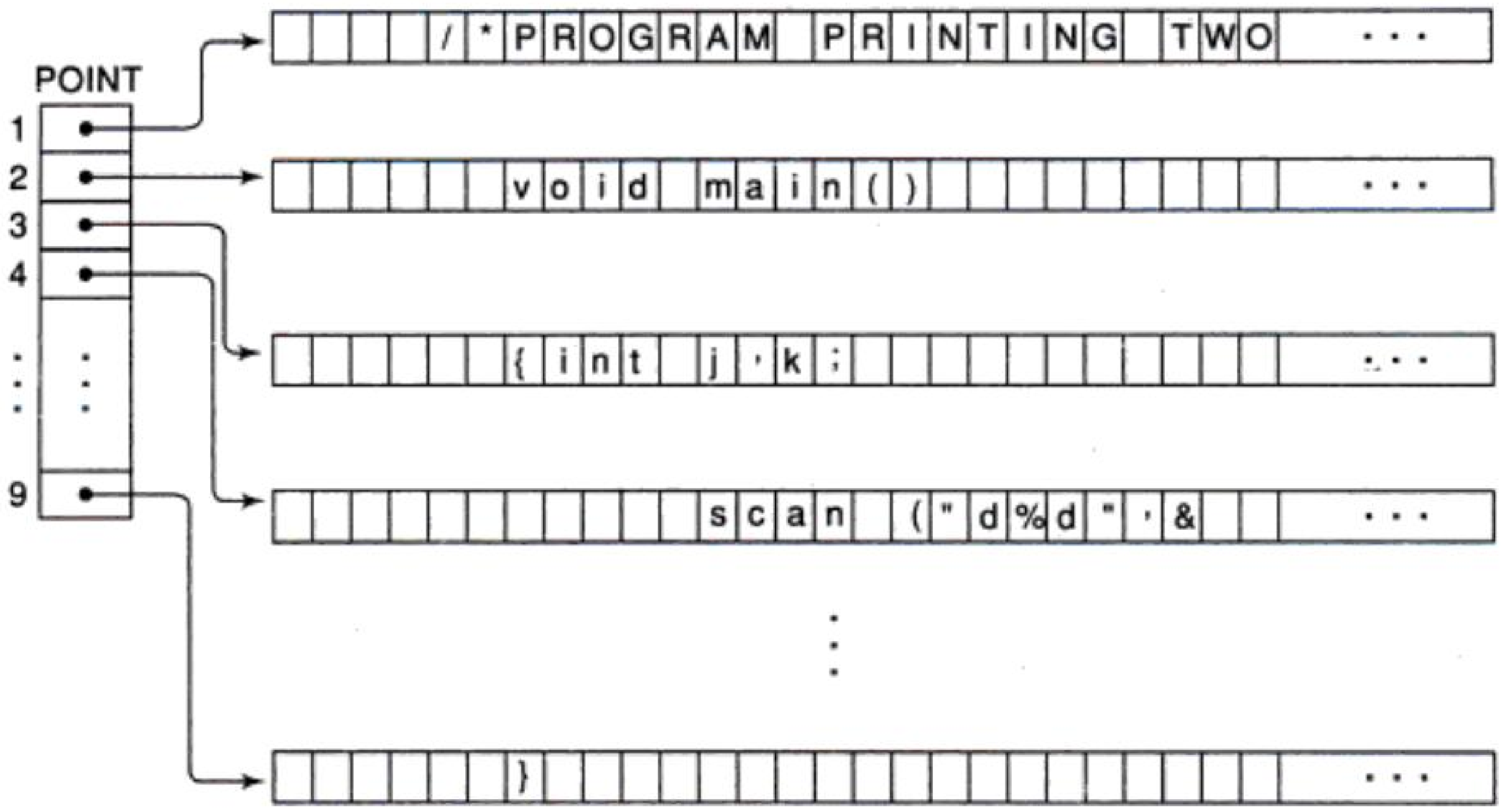


Fig. 3.3 Records Stored Using Pointers

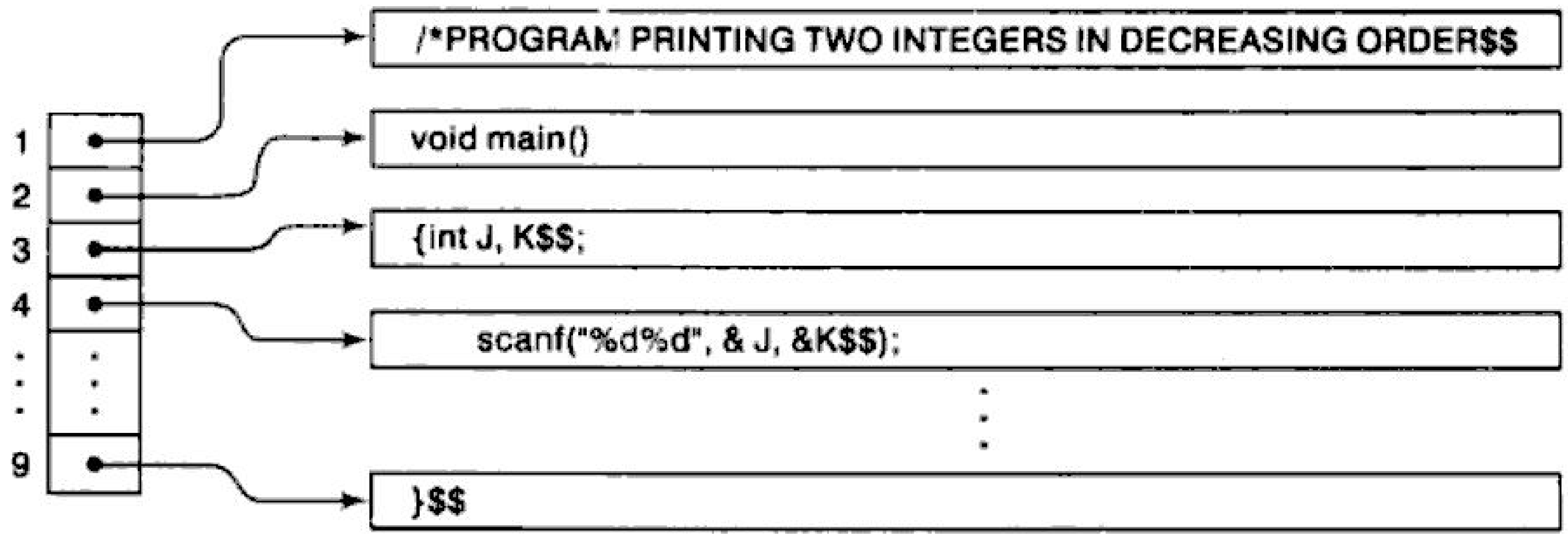
Variable-Length Storage with Fixed Maximum

Although strings may be stored in fixed-length memory locations as above, there are advantages in knowing the actual length of each string. For example, one then does not have to read the entire record when the string occupies only the beginning part of the memory location. Also, certain string operations (discussed in Sec. 3.4) depend on having such variable-length strings.

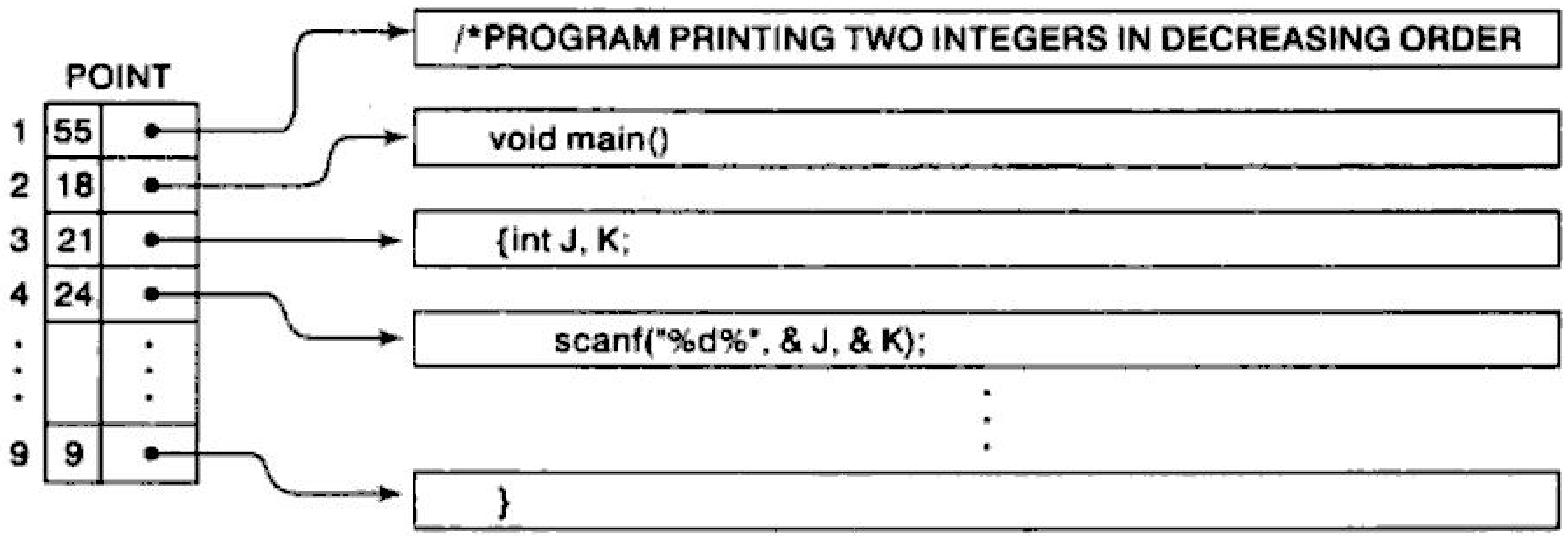
The storage of variable-length strings in memory cells with fixed lengths can be done in two general ways:

1. One can use a marker, such as two dollar signs ($$), to signal the end of the string.
2. One can list the length of the string—as an additional item in the pointer array, for example.

Using the data in Fig. 3.1, the first method is pictured in Fig. 3.4(a) and the second method is pictured in Fig. 3.4(b).



(a) Records with sentinels.

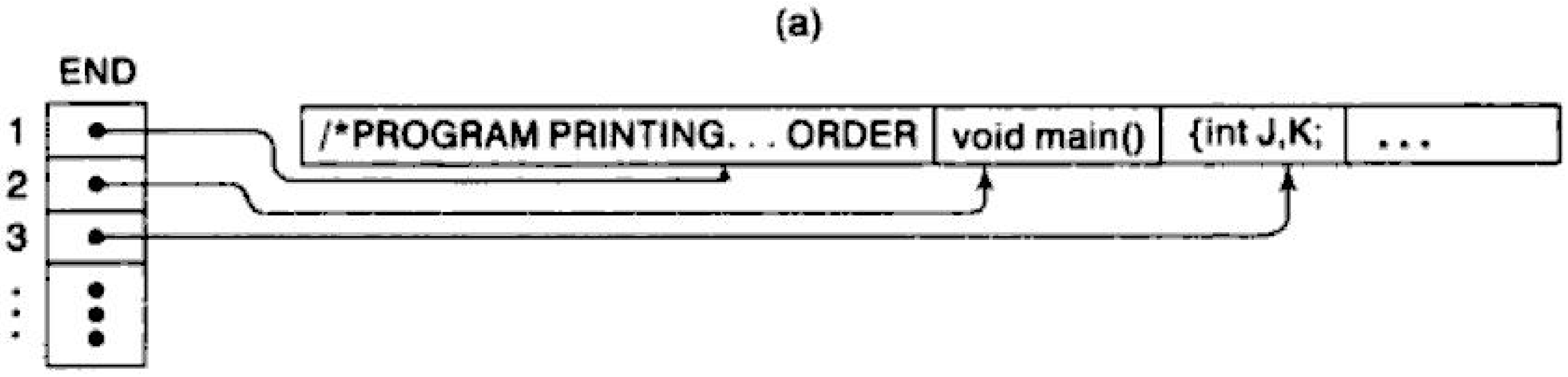


(b) Record whose lengths are listed.

Fig. 3.4

Remark: One might be tempted to store strings one after another by using some separation marker, such as the two dollar signs ($$) in Fig. 3.5(a), or by using a pointer array giving the location of the strings, as in Fig. 3.5(b). These ways of storing strings will obviously save space and are

(a)



(b)

Fig. 3.5 Records Stored One after Another sometimes used in secondary memory when records are relatively permanent and require little change. However, such methods of storage are usually inefficient when the strings and their lengths are frequently being changed.

Linked Storage

Computers are being used very frequently today for word processing, i.e., for inputting, processing and outputting printed matter. Therefore, the computer must be able to correct and modify the printed matter, which usually means deleting, changing and inserting words, phrases, sentences and even paragraphs in the text. However, the fixed-length memory cells discussed above do not easily lend themselves to these operations. Accordingly, for most extensive word processing applications, strings are stored by means of linked lists. Such linked lists, and the way data are inserted and deleted in them, are discussed in detail in Chapter 5. Here we simply look at the way strings appear in these data structures.

By a (one-way) linked list, we mean a linearly ordered sequence of memory cells, called *nodes*, where each node contains an item, called a *link*, which points to the next node in the list (i.e., which contains the address of the next node). Figure 3.6 is a schematic diagram of such a linked list.

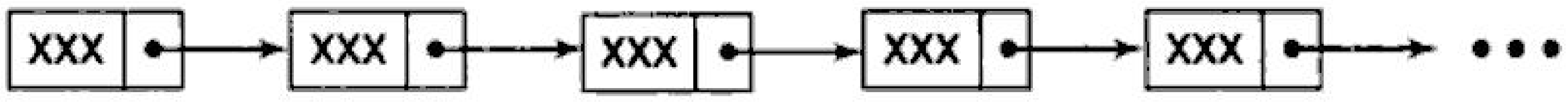
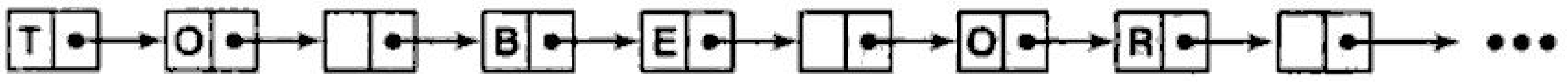


Fig. 3.6 Linked List

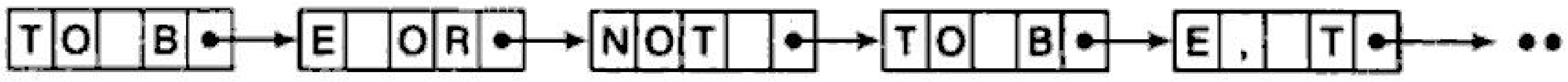
Strings may be stored in linked lists as follows. Each memory cell is assigned one character or a fixed number of characters, and a link contained in the cell gives the address of the cell containing the next character or group of characters in the string. For example, consider this famous quotation:

To be or not to be, that is the question.

Figure 3.7(a) shows how the string would appear in memory with one character per node, and Fig. 3.7(b) shows how it would appear with four characters per node.



(a) One character per node.



(b) Four characters per node.

Fig. 3.7

3.4 CHARACTER DATA TYPE

This section gives an overview of the way various programming languages handle the *character* data type. As noted in the preceding chapter (in Sec. 2.7), each data type has its own formula for decoding a sequence of bits in memory.

Constants

Many programming languages denote string constants by placing the string in either single or double quotation marks. For example,

'THE END' and 'TO BE OR NOT TO BE'

are string constants of lengths 7 and 18 characters respectively. Our algorithms will also define character constants in this way.

Variables

Each programming language has its own rules for forming character variables. However, such variables fall into one of three categories: static, semistatic and dynamic. By a *static* character variable, we mean a variable whose length is defined before the program is executed and cannot change throughout the program. By a *semistatic* character variable, we mean a variable whose length may vary during the execution of the program as long as the length does not exceed a maximum value determined by the program before the program is executed. By a *dynamic* character variable, we mean a variable whose length can change during the execution of the program. These three categories correspond, respectively, to the ways the strings are stored in the memory of the computer as discussed in the preceding section.

Example 3.2

In C, variables are declared using alphanumeric characters. The only special character that is allowed inside a variable name is an underscore (_). Further, a variable name must comply with certain rules; for instance:

- A variable name must always begin with a letter
- A variable name can not be same as a system keyword
- No white spaces are allowed inside a variable name

Following are some examples of character variable declaration in C:

```
char a;          //declares a single character variable a
char str[20];    //declares a character array (string) of length 20
```

3.5 STRINGS AS ADT

Most languages have strings as a built-in data type or a standard library, and a set of operations defined on that type. Therefore, there is actually no need for us to create our own string ADT.

However, we can implement our own string data type, if required. The use of a data type in string processing applications should not depend on how it is implemented, or whether it is built-in or user defined. We only need to know what operations are allowed on the string.

A string data type typically should have operations to:

- Return the n^{th} character in a string.
- Set the n^{th} character in a string to c.
- Find the length of a string.
- Concatenate two strings.
- Copy a string.

- Delete part of a string.
- Modify and compare strings in other ways.

The following is a set of operations we might want to do on strings.

GETCHAR(*str*, *n*)	Returns the n^{th} character in the string
PUTCHAR(*str*, *n*, *c*)	Sets the n^{th} character in the string to *c*
LENGTH(*str*)	Returns the number of characters in the string
POS(*str*1, *str*2)	Returns the position of the first occurrence of *str*2 found in *str*1, or 0 if no match
CONCAT(*str*1, *str*2)	Returns a new string consisting of characters in *str*1 followed by characters in *str*2
SUBSTRING(*str*1, *i*, *m*)	Returns a substring of length *m* starting at position *i* in string *str*
DELETE(*str*, *i*, *m*)	Deletes *m* characters from *str* starting at position *i*
INSERT(*str*1, *str*2, *i*)	Changes *str*1 into a new string with *str*2 inserted in position *i*
COMPARE(*str*1, *str*2)	Returns an integer indicating whether *str*1 > *str*2

Let S1 be a string. There are a number of ways in which this string can be implemented as shown in Example 3.3.

Example 3.3

Suppose S1 = 'JANICE'.

(a) In this case, as the length is known, it can be implemented as a fixed length array, where the first element denotes the length of the string as shown below.

[6, J, A, N, I, C, E]

(b) If the length is not known, it can be implemented as an array but with the end of the string indicated using a special 'NULL' character denoted by '\0' as shown below. Memory can then be dynamically allocated for the string once we know its length.

[J, A, N, I, C, E, \0,.....]

The first implementation has the disadvantages of all fixed length array implementations. However, some operations are efficient, for instance, finding the length.

The second implementation has the advantages of dynamic allocation of space; modifying the string also may be more efficient, as we need not recalculate size.

3.6 STRING OPERATIONS

Although a string may be viewed simply as a sequence or linear array of characters, there is a fundamental difference in use between strings and other types of arrays. Specifically, groups of consecutive elements in a string (such as words, phrases and sentences), called *substrings,* may be units unto themselves. Furthermore, the basic units of access in a string are usually these substrings, not individual characters.

Consider, for example, the string

'TO BE OR NOT TO BE'

We may view the string as the 18-character sequence T, O, □, B, ..., E. However, the substrings TO, BE, OR, ... have their own meaning.

On the other hand, consider an 18-element linear array of 18 integers, 4, 8, 6, 15, 9, 5, 4, 13, 8, 5, 11, 9, 9, 13, 7, 10, 6, 11

The basic unit of access in such an array is usually an individual element. Groups of consecutive elements normally do not have any special meaning.

For the above reason, various string operations have been developed which are not normally used with other kinds of arrays. This section discusses these string-oriented operations. The next section shows how these operations are used in word processing. (Unless otherwise stated or implied, we assume our character-type variables are dynamic and have a variable length determined by the context in which the variable is used.)

Substring

Accessing a substring from a given string requires three pieces of information: (1) the name of the string or the string itself, (2) the position of the first character of the substring in the given string and (3) the length of the substring or the position of the last character of the substring. We call this operation SUBSTRING. Specifically, we write

SUBSTRING(string, initial, length)

to denote the substring of a string S beginning in a position K and having a length L.

Example 3.4

(a) Using the above function we have:

```
SUBSTRING('TO BE OR NOT TO BE', 4, 7) = 'BE OR N'
SUBSTRING('THE END' , 4, 4) = '□END'
```

Program 3.1

```
/* Code showing the implementation of SUBSTRING function in C */
#include <stdio.h>
#include <conio.h>

void main()
{
char S[80]={"TO BE OR NOT TO BE"};
char *SUBSTR(char*,int,int);
clrscr();

printf("STRING = %s",S);
printf("\n\nSUBSTRING(S,4,7) = %s",SUBSTR(S,4,7));
getch();
}

char *SUBSTR(char *STR,int i,int j)
{
 int k,m=0;
```

```
  char STRRES[80];
  for(k=i-1;k<=i+j-1-1;k++)
  {
   STRRES[m]=STR[k];
   m=m+1;
  }
   STRRES[m]='\0';
  return(STRRES);
}

Output:
STRING = TO BE OR NOT TO BE

SUBSTRING(S,4,7) = BE OR N
```

Indexing

Indexing, also called *pattern matching,* refers to finding the position where a string pattern P first appears in a given string text T. We call this operation INDEX and write

`INDEX(text, pattern)`

If the pattern P does not appear in the text T, then INDEX is assigned the value 0. The arguments "text" and "pattern" can be either string constants or string variables.

Example 3.5

(a) Suppose T contains the text

'HIS FATHER IS THE PROFESSOR'

Then,

INDEX(T, 'THE'), INDEX(T, 'THEN') and INDEX(T, '□THE□')

have the values 7, 0 and 14, respectively.

Program 3.2

```
/* Code showing the implementation of INDEX function in C */
#include <stdio.h>
#include <conio.h>
#include <string.h>

void main()
{
char T[80]={"HIS FATHER IS THE PROFESSOR"};
int INDEX(char*,char*);
clrscr();
```

```
printf("T = %s",T);
printf("\n\nINDEX(T,'THE') = %d",INDEX(T,"THE"));
printf("\n\nINDEX(T,'THEN') = %d",INDEX(T,"THEN"));
printf("\n\nINDEX(T,' THE ') = %d",INDEX(T," THE "));
getch();
}

int INDEX(char *STR1,char *STR2)
{
 int m,n;
 int index,flag;

 for(m=0;m<strlen(STR1);m++)
 {
  index=m;
  flag=1;
  for(n=0;n<strlen(STR2);n++)
  {
   if(STR1[m+n]==STR2[n])
    ;
   else
   {
    flag=0;
    break;
   }
  }
  if(flag==0)
   continue;
  else
   return(index);
 }
 if(m==strlen(STR1))
  return(-1);
}
```

Output:

```
T = HIS FATHER IS THE PROFESSOR

INDEX(T,'THE') = 6

INDEX(T,'THEN') = -1

INDEX(T,' THE ') = 13
```

Remark: Since 0 is a valid index location in C, we have used –1 to denote instances where pattern does not match the text.

Concatenation

Let S_1 and S_2 be strings. Recall (Sec. 3.2) that the *concatenation* of S_1 and S_2, which we denote by $S_1//S_2$, is the string consisting of the characters of S_1 followed by the characters of S_2.

Example 3.6

(a) Suppose S_1 = 'MARK' and S_2 = 'TWAIN'. Then:

$S_1//S_2$ = 'MARKTWAIN' but $S_1//$ '□'$//S_2$ = 'MARK TWAIN'

(b) Concatenation is performed in C language using the strcat function, as shown below:

```
strcat(S1,S2);      //Concatenates strings S1 and S2 and stores
                    the result in S1
```

strcat () function is part of the string.h header file; hence it must be included at the time of pre-processing.

Program 3.3

```
/* Code showing string concatenation in C */
#include <stdio.h>
#include <conio.h>
#include <string.h>

void main()
{
char S1[80]={"MARK"};
char S2[80]={"TWAIN"};
clrscr();

printf("S1 = %s",S1);
printf("\nS2 = %s",S2);
printf("\nS1//S2 = %s",strcat(S1,S2));
strcpy(S1,"MARK");
printf("\nS1//' '//S2 = %s",strcat(strcat(S1," "),S2));

getch();
}
```

Output:

```
S1 = MARK
S2 = TWAIN
S1//S2 = MARKTWAIN
S1//' '//S2 = MARK TWAIN
```

Length

The number of characters in a string is called its length. We will write

LENGTH(string)

for the length of a given string.

Example 3.7

(a) Suppose S = 'COMPUTER'. Then:

LENGTH (S) = 8

Similarly,

LENGTH ('MARC TWAIN') = 10

(b) String length is determined in C language using the strlen function, as shown below:

```
X = strlen("Sunrise");    //strlen function returns an integer value
                          7 and assigns it to the variable X
```

Similar to strcat, strlen is also a part of string.h, hence the header file must be included at the time of pre-processing.

Program 3.4

```
/* Code showing the use of strlen () in C */
#include <stdio.h>
#include <conio.h>
#include <string.h>

void main()
{
char S1[80]={"COMPUTER"};
char S2[80]={"MARC"};
clrscr();

printf("Length(%s) = %d",S1,strlen(S1));
printf("\nLength(%s) = %d",S2,strlen(S2));

getch();
}

Output:
Length(COMPUTER) = 8
Length(MARC) = 4
```

3.7 WORD/TEXT PROCESSING

In earlier times, character data processed by the computer consisted mainly of data items, such as names and addresses. Today the computer also processes printed matter, such as letters, articles and reports. It is in this latter context that we use the term "word processing."

Given some printed text, the operations usually associated with word processing are the following:

(a) *Replacement.* Replacing one string in the text by another.

(b) *Insertion.* Inserting a string in the middle of the text.

(c) *Deletion.* Deleting a string from the text.

The above operations can be executed by using the string operations discussed in the preceding section. This we show below when we discuss each operation separately. Many of these operations are built into or can easily be defined in most of the programming languages.

Insertion

Suppose in a given text T we want to insert a string S so that S begins in position K. We denote this operation by

INSERT(text, position, string)

For example,

INSERT ('ABCDEFG', 3, 'XYZ') = 'ABXYZCDEFG'
INSERT ('ABCDEFG', 6, 'XYZ') = 'ABCDEXYZFG'

This INSERT function can be implemented by using the string operation defined in the previous section as follows:

INSERT(T, K, S) = SUBSTRING(T, 1, K − 1) //S// SUBSTRING(T, K, LENGTH(T) − K + 1)

That is, the initial substring of T before the position K, which has length K − 1, is concatenated with the string S, and the result is concatenated with the remaining part of T, which begins in position K and has length LENGTH(T) − (K − 1) = Length (T) − K + 1. (We are assuming implicitly that T is a dyamic variable and that the size of T will not become too large.)

Deletion

Suppose in a given text T we want to delete the substring which begins at position K and has length L. We denote this operation by

DELETE(text, position, length)

For example,

DELETE(' ABCDEFG ', 4, 2) = ' ABCFG '
DELETE(' ABCDEFG ', 2, 4) = ' AFG '

We assume that nothing is deleted if position K = 0. Thus

DELETE(' ABCDEFG ', 0, 2) = ' ABCDEFG '

The importance of this "zero case" is seen later.

The DELETE function can be implemented using the string operations given in the preceding section as follows:

DELETE(T, K, L) =
SUBSTRING(T, 1, K − 1)//SUBSTRING(T, K + L, LENGTH(T) − K − L + 1)

That is, the initial substring of T before position K is concatenated with the terminal substring of T beginning at position K + L. The length of the initial substring is K − 1, and the length of the terminal substring is:

LENGTH(T) – (K + L – 1) = LENGTH(T) – K – L + 1

We also assume that DELETE(T, K, L) = T when K = 0.

Now suppose text T and pattern P are given and we want to delete from T the first occurrence of the pattern P. This can be accomplished by using the above DELETE function as follows:

DELETE(T, INDEX(T, P), LENGTH(P))

That is, in the text T, we first compute INDEX(T, P), the position where P first occurs in T, and then we compute LENGTH(P), the number of characters in P. Recall that when INDEX(T, P) = 0 (i.e., when P does not occur in T) the text T is not changed.

Example 3.8

(a) Suppose T = 'ABCDEFG' and P = 'CD'. Then INDEX(T, P) = 3 and LENGTH(P) = 2. Hence

DELETE ('ABCDEFG', 3, 2) = 'ABEFG'

(b) Suppose T = 'ABCDEFG' and P = 'DC'. Then INDEX(T, P) = 0 and LENGTH(P) = 2. Hence, by the "zero case,"

DELETE('ABCDEFG ', 0, 2) = ' ABCDEFG '

as expected.

Suppose after reading into the computer a text T and a pattern P, we want to delete every occurrence of the pattern P in the text T. This can be accomplished by repeatedly applying

DELETE(T, INDEX(T, P), LENGTH(P))

until INDEX(T, P) = 0 (i.e., until P does not appear in T). An algorithm which accomplishes this follows.

Algorithm 3.1: A text T and a pattern P are in memory. This algorithm deletes every occurrence of P in T.

1. [Find index of P.] Set K := INDEX(T, P).
2. Repeat while K ¹ 0:
    (a) [Delete P from T.]
        Set T := DELETE(T, INDEX(T, P), LENGTH(P))
    (b) [Update index.] Set K := INDEX(T, P).
    [End of loop.]
3. Write : T.
4. Exit.

We emphasize that after each deletion, the length of T decreases and hence the algorithm must stop. However, the number of times the loop is executed may exceed the number of times P appears in the original text T, as illustrated in the following example.

Example 3.9

(a) Suppose Algorithm 3.1 is run with the data

T = XABYABZ, P = AB

Then the loop in the algorithm will be executed twice. During the first execution, the first occurrence of AB in T is deleted, with the result that T = XYABZ. During the second execution, the remaining occurrence of AB in T is deleted, so that T = XYZ. Accordingly, XYZ is the output.

(b) Suppose Algorithm 3.1 is run with the data

T = XAAABBBY, P = AB

Observe that the pattern AB occurs only once in T but the loop in the algorithm will be executed three times. Specifically, after AB is deleted the first time from T we have T = XAABBY, and hence AB appears again in T. After AB is deleted a second time from T, we see that T = XABY and AB still occurs in T. Finally, after AB is deleted a third time from T, we have T = XY and AB does not appear in T, and thus INDEX(T, P) = 0. Hence XY is the output.

The above example shows that when a text T is changed by a deletion, patterns may occur that did not appear originally.

Replacement

Suppose in a given text T we want to replace the first occurrence of a pattern P_1 by a pattern P_2. We will denote this operation by

$$\text{REPLACE(text, pattern}_1\text{, pattern}_2\text{)}$$

For example

REPLACE('XABYABZ', 'AB', 'C') = 'XCYABZ'
REPLACE('XABYABZ' , 'BA', 'C') = 'XABYABZ'

In the second case, the pattern BA does not occur, and hence there is no change.

We note that this REPLACE function can be expressed as a deletion followed by an insertion if we use the preceding DELETE and INSERT functions. Specifically, the REPLACE function can be executed by using the following three steps:

K := INDEX(T, P_1)
T := DELETE(T, K, LENGTH(P_1))
INSERT(T, K, P_2)

The first two steps delete P_1 from T, and the third step inserts P_2 in the position K from which P_1 was deleted.

Suppose a text T and patterns P and Q are in the memory of a computer. Suppose we want to replace every occurrence of the pattern P in T by the pattern Q. This might be accomplished by repeatedly applying

REPLACE(T, P, Q)

until INDEX(T, P) = 0 (i.e., until P does not appear in T). An algorithm which does this follows.

Algorithm 3.2: A text T and patterns P and Q are in memory. This algorithm replaces every occurrence of P in T by Q.

1. [Find index of P.] Set K := INDEX(T, P).
2. Repeat while K ^{1} 0:
   - (a) [Replace P by Q.] Set T := REPLACE(T, P, Q).
   - (b) [Update index.] Set K := INDEX(T, P).

   [End of loop.]
3. Write: T.
4. Exit.

Warning: Although this algorithm looks very much like Algorithm 3.1, there is no guarantee that this algorithm will terminate. This fact is illustrated in Example 3.10(b). On the other hand, suppose the length of Q is smaller than the length of P. Then the length of T after each replacement decreases. This guarantees that in this special case where Q is smaller than P the algorithm must terminate.

Example 3.10

(a) Suppose Algorithm 3.2 is run with the data

T = XABYABZ, P = AB, Q = C

Then the loop in the algorithm will be executed twice. During the first execution, the first occurrence of AB in T is replaced by C to yield T = XCYABZ. During the second execution, the remaining AB in T is replaced by C to yield T = XCYCZ. Hence XCYCZ is the output.

(b) Suppose Algorithm 3.2 is run with the data

T = XAY, P = A, Q = AB

Then the algorithm will never terminate. The reason for this is that P will always occur in the text T, no matter how many times the loop is executed. Specifically,

T = XABY at the end of the first execution of the loop
T = XAB^{2}Y at the end of the second execution of the loop
....
T = XAB^{n}Y at the end of the *n*th execution of the loop

(The infinite loop arises here since P is a substring of Q.)

The following program shows the implementation of INSERTION, DELETION and REPLACEMENT algorithms in C:

Program 3.5

```
#include <stdio.h>
#include <conio.h>
```

```
char*  SUBSTR(char*,int,int);
int  INDEX(char*,char*);
char*  INSERT(char*,int,char*);
char*  DELETE(char*,int,int);
char*  REPLACE(char*,char*,char*);

void main()
{
char  S[80]={"ABCDEFG"};
char  R1[80],R2[80],R3[80],R4[80],R5[80],R6[80];
clrscr();

printf("STRING = %s",S);
strcpy(R1,INSERT(S,3,"XYZ"));
strcpy(R2,INSERT(S,6,"XYZ"));
printf("\n\nINSERT('ABCDEFG',3,'XYZ')  =  %s",R1);
printf("\n\nINSERT('ABCDEFG',6,'XYZ')  =  %s",R2);
strcpy(R3,DELETE(S,4,2));
strcpy(R4,DELETE(S,2,4));
printf("\n\nDELETE('ABCDEFG',4,2)  =  %s",R3);
printf("\n\nDELETE('ABCDEFG',2,4)  =  %s",R4);
strcpy(R5,REPLACE("XABYABZ","AB","C"));
strcpy(R6,REPLACE("XABYABZ","BA","C"));
printf("\n\nREPLACE('XABYABZ','AB','C')  =  %s",R5);
printf("\n\nREPLACE('XABYABZ','BA','C')  =  %s",R6);

getch();
}

char*  INSERT(char*  S1,int  K,char*S2)
{
 char  RESULT[80];
 strcpy(RESULT,SUBSTR(S1,1,K-1));
 strcat(RESULT,S2);
 strcat(RESULT,SUBSTR(S1,K,strlen(S1)-K+1));
 return(RESULT);
}

char*  DELETE(char*  S1,int  K,int  L)
{
 char  RESULT[80];
 strcpy(RESULT,SUBSTR(S1,1,K-1));
 strcat(RESULT,SUBSTR(S1,K+L,strlen(S1)-K-L+1));
 return(RESULT);
}
```

```
char* REPLACE(char* S1,char* S2,char* S3)
{
 int K;
 char RES1[80];
 char RES2[80];
 if(INDEX(S1,S2)!=-1)
   K=INDEX(S1,S2)+1;
 else
 return(S1);
 strcpy(RES1,DELETE(S1,K,strlen(S2)));
 strcpy(RES2,INSERT(RES1,K,S3));
 return(RES2);
}

char *SUBSTR(char *STR,int i,int j)
{
 int k,m=0;
 char STRRES[80];
 for(k=i-1;k<=i+j-1-1;k++)
 {
   STRRES[m]=STR[k];
   m=m+1;
 }
   STRRES[m]='\0';
 return(STRRES);
}

int INDEX(char *STR1,char *STR2)
{
 int m,n;
 int index,flag;

 for(m=0;m<strlen(STR1);m++)
 {
  index=m;
  flag=1;
  for(n=0;n<strlen(STR2);n++)
  {
    if(STR1[m+n]==STR2[n])
     ;
    else
    {
     flag=0;
     break;
```

```
      }
    }
    if(flag==0)
      continue;
    else
      return(index);
  }
  if(m==strlen(STR1))
    return(-1);
}
```

Output:

```
STRING = ABCDEFG

INSERT('ABCDEFG',3,'XYZ') = ABXYZCDEFG

INSERT('ABCDEFG',6,'XYZ') = ABCDEXYZFG

DELETE('ABCDEFG',4,2) = ABCFG

DELETE('ABCDEFG',2,4) = AFG

REPLACE('XABYABZ','AB','C') = XCYABZ

REPLACE('XABYABZ','BA','C') = XABYABZ
```

3.8 PATTERN MATCHING ALGORITHMS

Pattern matching is the problem of deciding whether or not a given string pattern P appears in a string text T. We assume that the length of P does not exceed the length of T. This section discusses two pattern matching algorithms. We also discuss the complexity of the algorithms so we can compare their efficiencies.

Remark: During the discussion of pattern matching algorithms, characters are sometimes denoted by lowercase letters (*a, b, c,* ...) and exponents may be used to denote repetition; e.g.,

$$a^2b^3ab^2 \text{ for } aabbbabb \quad \text{and} \quad (cd)^3 \text{ for } cdcdcd$$

In addition, the empty string may be denoted by Λ, the Greek letter lambda, and the concatenation of strings X and Y may be denoted by $X \cdot Y$ or, simply, XY.

First Pattern Matching Algorithm

The first pattern matching algorithm is the obvious one in which we compare a given pattern P with each of the substrings of T, moving from left to right, until we get a match. In detail, let

$$W_K = \text{SUBSTRING(T, K, LENGTH(P))}$$

That is, let W_K denote the substring of T having the same length as P and beginning with the Kth character of T. First we compare P, character by character, with the first substring, W_1. If all the characters are the same, then P = W_1 and so P appears in T and INDEX(T, P) = 1. On the other hand, suppose we find that some character of P is not the same as the corresponding character of W_1. Then P ≠ W_1 and we can immediately move on to the next substring, W_2. That is, we next compare P with W_2. If P ≠ W_2, then we compare P with W_3, and so on. The process stops (a) when we find a match of P with some substring W_K and so P appears in T and INDEX(T, P) = K, or (b) when we exhaust all the W_K's with no match and hence P does not appear in T. The maximum value MAX of the subscript K is equal to LENGTH(T) – LENGTH(P) + 1.

Let us assume, as an illustration, that P is a 4-character string and that T is a 20-character string, and that P and T appear in memory as linear arrays with one character per element. That is,

$$P = P[1]P[2]P[3]P[4] \quad \text{and} \quad T = T[1]T[2]T[3] \cdots T[19]T[20]$$

Then P is compared with each of the following 4-character substrings of T:

$$W_1 = T[1]T[2]T[3]T[4], \quad W_2 = T[2]T[3]T[4]T[5], \quad \ldots, \quad W_{17} = T[17]T[18]T[19]T[20]$$

Note that there are MAX = 20 – 4 + 1 = 17 such substrings of T.

A formal presentation of our algorithm, where P is an *r*-character string and T is an *s*-character string, is shown in Algorithm 3.3.

Observe that Algorithm 3.3 contains two loops, one inside the other. The outer loop runs through each successive R-character substring

$$W_K = T[K]T[K + 1] \cdots T[K + R - 1]$$

of T. The inner loop compares P with W_K, character by character. If any character does not match, then control transfers to Step 5, which increases K and then leads to the next substring of T. If all the R characters of P do match those of some W_K, then P appears in T and K is the INDEX of P in T. On the other hand, if the outer loop completes all of its cycles, then P does not appear in T and so INDEX = 0.

Algorithm 3.3: (Pattern Matching) P and T are strings with lengths R and S, respectively, and are stored as arrays with one character per element. This algorithm finds the INDEX of P in T.

1. [Initialize.] Set K := 1 and MAX : = S – R + 1.
2. Repeat Steps 3 to 5 while K £ MAX:
3. Repeat for L = 1 to R: [Tests each character of P.]
   If P[L] ¹ T[K + L – 1], then: Go to Step 5.
   [End of inner loop.]
4. [Success.] Set INDEX = K, and Exit.
5. Set K := K + 1.
   [End of Step 2 outer loop.]
6. [Failure.] Set INDEX = 0.
7. Exit.

Program 3.6

```
/* C implementation of Algorithm 3.3*/
#include <stdio.h>
#include <conio.h>

void main()
{
char P[80]={"bab"};
char T[80]={"aabbbabb"};
int R,S,K,L,MAX,INDEX;
clrscr();

R=strlen(P);
S=strlen(T);
K=0;
MAX=S-R;

while(K<=MAX)
{
 for(L=0;L<R;L++)
 if(P[L]!=T[K+L])
  break;

 if(L==R)
 {
  INDEX=K;
  break;
 }
 else
  K=K+1;
}
if(K>MAX)
 INDEX=-1;

printf("P = %s",P);
printf("\n\nT = %s",T);

if(INDEX!=-1)
 printf("\n\nIndex of P in T is %d",INDEX);
else
 printf("\n\nP does not exist in T");

getch();
}
```

Output:

```
P = bab

T = aabbbabb

Index of P in T is 4
```

The complexity of this pattern matching algorithm is measured by the number C of comparisons between characters in the pattern P and characters of the text T. In order to find C, we let N_k denote the number of comparisons that take place in the inner loop when P is compared with W_K. Then

$$C = N_1 + N_2 + \cdots + N_L$$

where L is the position L in T where P first appears or L = MAX if P does not appear in T. The next example computes C for some specific P and T where LENGTH(P) = 4 and LENGTH(T) = 20 and so MAX = 20 − 4 + 1 = 17.

Example 3.11

(a) Suppose P = *aaba* and T = *cdcd* ⋯ *cd* = $(cd)^{10}$. Clearly P does not occur in T. Also, for each of the 17 cycles, $N_k = 1$, since the first character of P does not match W_K. Hence

$$C = 1 + 1 + 1 + \cdots + 1 = 17$$

(b) Suppose P = *aaba* and T = *ababaaba*.... Observe that P is a substring of T. In fact, P = W_5 and so $N_5 = 4$. Also, comparing P with W_1 = *abab*, we see that $N_1 = 2$, since the first letters do match; but comparing P with W_2 = *baba*, we see that $N_2 = 1$, since the first letters do not match. Similarly, $N_3 = 2$ and $N_4 = 1$. Accordingly,

$$C = 2 + 1 + 2 + 1 + 4 = 10$$

(c) Suppose P = *aaab* and T = *aa* ⋯ *a* = a^{20}. Here P does not appear in T. Also, every W_K = *aaaa*; hence every $N_k = 4$, since the first three letters of P do match. Accordingly,

$$C = 4 + 4 + \cdots + 4 = 17 \cdot 4 = 68$$

In general, when P is an r-character string and T is an s-character string, the data size for the algorithm is

$$n = r + s$$

The worst case occurs when every character of P except the last matches every substring W_K, as in Example 3.10(c). In this case, $C(n) = r(s - r + 1)$. For fixed n, we have $s = n - r$, so that

$$C(n) = r(n - 2r + 1)$$

The maximum value of $C(n)$ occurs when $r = (n + 1)/4$. (See Problem 3.19.) Accordingly, substituting this value for r in the formula for $C(n)$ yields

$$C(n) = \frac{(n+1)^2}{8} = O(n^2)$$

The complexity of the average case in any actual situation depends on certain probabilities which are usually unknown. When the characters of P and T are randomly selected from some finite alphabet, the complexity of the average case is still not easy to analyze, but the complexity of the average case is still a factor of the worst case. Accordingly, we shall state the following: *The complexity of this pattern matching algorithm is equal to* $O(n^2)$. In other words, the time required to execute this algorithm is proportional to n^2. (Compare this result with the one on page 3.28.)

Second Pattern Matching Algorithm

The second pattern matching algorithm uses a table which is derived from a particular pattern P but is independent of the text T. For definiteness, suppose

$$P = aaba$$

First we give the reason for the table entries and how they are used. Suppose $T = T_1T_2T_3 \ldots$, where T_i denotes the ith character of T; and suppose the first two characters of T match those of P; i.e., suppose $T = aa\ldots$. Then T has one of the following three forms:

(i) $T = aab\ldots$, (ii) $T = aaa\ldots$, (iii) $T = aax$

where x is any character different from a or b. Suppose we read T_3 and find that $T_3 = b$. Then we next read T_4 to see if $T_4 = a$, which will give a match of P with W_1. On the other hand, suppose $T_3 = a$. Then we know that $P \neq W_1$; but we also know that $W_2 = aa\ldots$, i.e., that the first two characters of the substring W_2 match those of P. Hence we next read T_4 to see if $T_4 = b$. Last, suppose $T_3 = x$. Then we know that $P \neq W_1$, but we also know that $P \neq W_2$ and $P \neq W_3$, since x does not appear in P. Hence we next read T_4 to see if $T_4 = a$, i.e., to see if the first character of W_4 matches the first character of P.

There are two important points to the above procedure. First, when we read T_3 we need only compare T_3 with those characters which appear in P. If none of these match, then we are in the last case, of a character x which does not appear in P. Second, after reading and checking T_3, we next read T_4; we do not have to go back again in the text T.

Figure 3.8(a) contains the table that is used in our second pattern matching algorithm for the pattern $P = aaba$. (In both the table and the accompanying graph, the pattern P and its substrings Q will be represented by italic capital letters.) The table is obtained as follows. First of all, we let Q_i denote the initial substring of P of length i; hence

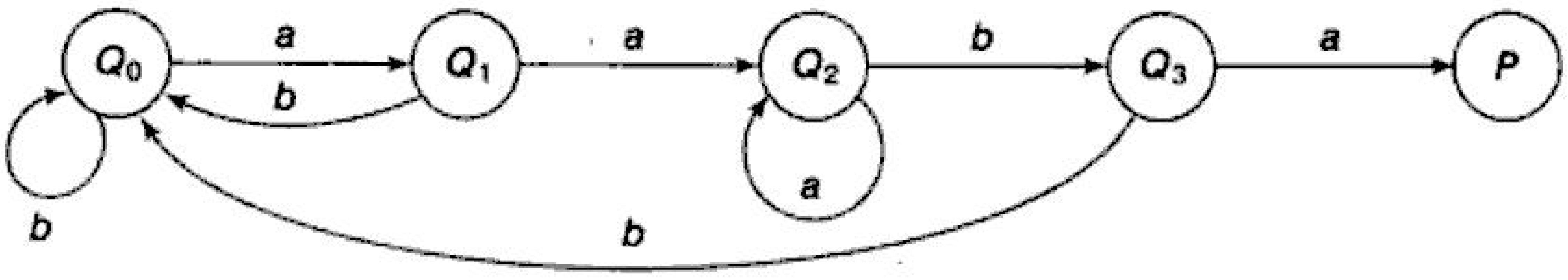
$$Q_0 = \Lambda, \quad Q_1 = a, \quad Q_2 = a^2, \quad Q_3 = a^2b, \quad Q_4 = a^2ba = P$$

(Here $Q_0 = \Lambda$ is the empty string.) The rows of the table are labeled by these initial substrings of P, excluding P itself. The columns of the table are labeled a, b and x, where x represents any character that doesn't appear in the pattern P. Let f be the function determined by the table; i.e., let

$$f(Q_i, t)$$

	a	b	x
Q_0	Q_1	Q_0	Q_0
Q_1	Q_2	Q_0	Q_0
Q_2	Q_2	Q_3	Q_0
Q_3	P	Q_0	Q_0

(a) Pattern matching table



(a) Pattern matching graph

Fig. 3.8

denote the entry in the table in row Q_i and column t (where t is any character). This entry $f(Q_i, t)$ is defined to be the largest Q that appears as a terminal substring in the string Q_it, the concatenation of Q_i and t. For example,

a^2 is the largest Q that is a terminal substring of $Q_2a = a^3$, so $f(Q_2, a) = Q_2$
Λ is the largest Q that is a terminal substring of $Q_1b = ab$, so $f(Q_1, b) = Q_0$
a is the largest Q that is a terminal substring of $Q_0a = a$, so $f(Q_0, a) = Q_1$
Λ is the largest Q that is a terminal substring of $Q_3a = a^3bx$, so $f(Q_3, x) = Q_0$

and so on. Although $Q_1 = a$ is a terminal substring of $Q_2a = a^3$, we have $f(Q_2, a) = Q_2$ because Q_2 is also a terminal substring of $Q_2a = a^3$ and Q_2 is larger than Q_1. We note that $f(Q_i, x) = Q_0$ for any Q, since x does not appear in the pattern P. Accordingly, the column corresponding to x is usually omitted from the table.

Our table can also be pictured by the labeled directed graph in Fig. 3.8(b). The graph is obtained as follows. First, there is a node in the graph corresponding to each initial substring Q_i of P. The Q's are called the *states* of the system, and Q_0 is called the *initial* state. Second, there is an arrow (a directed edge) in the graph corresponding to each entry in the table. Specifically, if

$$f(Q_i, t) = Q_j$$

then there is an arrow labeled by the character t from Q_i to Q_j. For example, $f(Q_2, b) = Q_3$, so there is an arrow labeled b from Q_2 to Q_3. For notational convenience, we have omitted all arrows labeled x, which must lead to the initial state Q_0.

We are now ready to give the second pattern matching algorithm for the pattern P = *aaba*. (Note that in the following discussion capital letters will be used for all single-letter variable names that appear in the algorithm.) Let $T = T_1T_2T_3 \cdots T_N$ denote the n-character-string text which is searched for the pattern P. Beginning with the initial state Q_0 and using the text T, we will obtain a sequence of states $S_1, S_2, S_3, \ldots$ as follows. We let $S_1 = Q_0$ and we read the first character T_1. From either the table or the graph in Fig. 3.8, the pair (S_1, T_1) yields a second state S_2; that is, $F(S_1, T_1) = S_2$. We read the next character T_2. The pair (S_2, T_2) yields a state S_3, and so on. There are two possibilities:

1. Some state S_K = P, the desired pattern. In this case, P does appear in T and its index is K − LENGTH(P).
2. No state $S_1, S_2, \ldots, S_{N+1}$ is equal to P. In this case, P does not appear in T.

We illustrate the algorithm with two different texts using the pattern P = *aaba*.

Example 3.12

(a) Suppose T = *aabcaba*. Beginning with Q_0, we use the characters of T and the graph (or table) in Fig. 3.8 to obtain the following sequence of states:

$$Q_0 \xrightarrow{Ca} Q_1 \xrightarrow{Ca} Q_2 \xrightarrow{Cb} Q_3 \xrightarrow{Cc} Q_0 \xrightarrow{Ca} Q_1 \xrightarrow{Cb} Q_0 \xrightarrow{Ca} Q_1$$

We do not obtain the state P, so P does not appear in T.

(b) Suppose T = *abcaabaca*. Then we obtain the following sequence of states:

$$Q_0 \xrightarrow{Ca} Q_1 \xrightarrow{Cb} Q_0 \xrightarrow{Cc} Q_0 \xrightarrow{Ca} Q_1 \xrightarrow{Ca} Q_2 \xrightarrow{Cb} Q_3 \xrightarrow{Ca} P$$

Here we obtain the pattern P as the state S_8. Hence P does appear in T and its index is 8 − LENGTH(P) = 4.

The formal statement of our second pattern matching algorithm follows:

Algorithm 3.4: (Pattern Matching). The pattern matching table $F(Q_1, T)$ of a pattern P is in memory, and the input is an N-character string $T = T_1T_2 \ldots T_N$. This algorithm finds the INDEX of P in T.

1. [Initialize.] Set K := 1 and $S_1 = Q_0$
2. Repeat Steps 3 to 5 while S_K ¹ P and K £ N.
3. Read T_K.
4. Set $S_{K+1} := F(S_K, T_K)$. [Finds next state.]
5. Set K := K + 1. [Updates counter.]

   [End of Step 2 loop.]
6. [Successful?]

   If S_K = P, then:

   INDEX = K – LENGTH(P).

   Else:

   INDEX = 0.

   [End of If structure.]
7. Exit

Program 3.7

```
/* C implementation of Algorithm 3.4*/
#include <stdio.h>
#include <conio.h>
char F(char,char);
int state[4][3];

void main()
{
char P[80]={"aaba"};
char T[80]={"abcaabaca"};
int N,K,S,I,INDEX;
clrscr();

state[0][0]=1;
state[0][1]=0;
state[0][2]=0;
state[1][0]=2;
state[1][1]=0;
state[1][2]=0;
state[2][0]=2;
state[2][1]=3;
state[2][2]=0;
state[3][0]=-1;
```

```
state[3][1]=0;
state[3][2]=0;
N=strlen(T);
K=0;
S=0;

while(K<N && S!=-1)
{
 if(T[K]=='a')
 I=0;
 if(T[K]=='b')
 I=1;
 if(T[K]=='x')
 I=2;

 S=F(S,I);
 K=K+1;
}

if(S==-1)
 INDEX=K-strlen(P);
else
 INDEX=-1;

printf("P = %s",P);
printf("\n\nT = %s",T);

if(INDEX!=-1)
 printf("\n\nIndex of P in T is %d",INDEX);
else
 printf("\n\nP does not exist in T");

getch();
}


char F(char SK,char TK)
{
 return(state[SK][TK]);
}
```

Output:

```
P = aaba

T = abcaabaca

Index of P in T is 3
```

The running time of the above algorithm is proportional to the number of times the Step 2 loop is executed. The worst case occurs when all of the text T is read, i.e., when the loop is executed n = LENGTH(T) times. Accordingly, we can state the following: *The complexity of this pattern matching algorithm is equal to $O(n)$.*

Remark: A combinatorial problem is said to be *solvable in polynomial time* if there is an algorithmic solution with complexity equal to $O(n^m)$ for some m, and it is said to be *solvable in linear time* if there is an algorithmic solution with complexity equal to $O(n)$, where n is the size of the data. Thus the second of the two pattern matching algorithms described in this section is solvable in linear time. (The first pattern matching algorithm was solvable in polynomial time.)

SOLVED PROBLEMS

Terminology; Storage of Strings

3.1 Let W be the string ABCD. (a) Find the length of W. (b) List all substrings of W. (c) List all the initial substrings of W.

(a) The number of characters in W is its length, so 4 is the length of W.

(b) Any subsequence of characters of W is a substring of W. There are 11 such substrings:

Substrings:	ABCD,	ABC, BCD,	AB, BC, CD,	A, B, C, D,	Λ
Lengths:	4	3	2	1	0

(Here Λ denotes the empty string.)

(c) The initial substrings are ABCD, ABC, AB, A, Λ; that is, both the empty string and those substrings that begin with A.

3.2 Assuming a programming language uses at least 48 characters—26 letters, 10 digits and a minimum of 12 special characters—give the minimum number and the usual number of bits to represent a character in the memory of the computer.

Since $2^5 < 48 < 2^6$, one requires at least a 6-bit code to represent 48 characters. Usually a computer uses a 7-bit code, such as ASCII, or an 8-bit code, such as EBCDIC, to represent characters. This allows many more special characters to be represented and processed by the computer.

3.3 Describe briefly the three types of structures used for storing strings.

(a) Fixed-length-storage structures. Here strings are stored in memory cells that are all of the same length, usually space for 80 characters.

(b) Variable-length storage with fixed maximums. Here strings are also stored in memory cells all of the same length; however, one also knows the actual length of the string in the cell.

(c) Linked-list storage. Here each cell is divided into two parts; the first part stores a single character (or a fixed small number of characters), and the second part contains the address of the cell containing the next character.

3.4 Find the string stored in Fig. 3.9, assuming the link value 0 signals the end of the list.

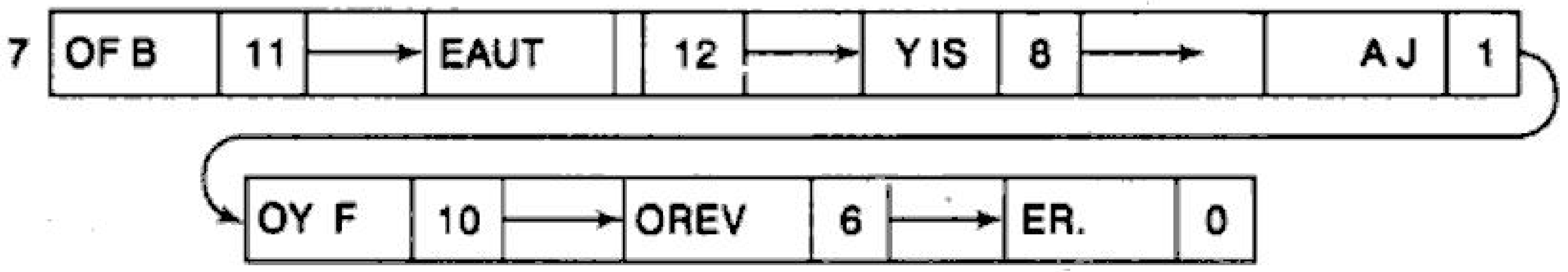
Here the string is stored in a linked-list structure with 4 characters per node. The value of START gives the location of the first node in the list:

4 | A TH | 2 |

The link value in this node gives the location of the next node in the list:

2 | ING | 7 |

Continuing in this manner, we obtain the following sequence of nodes:



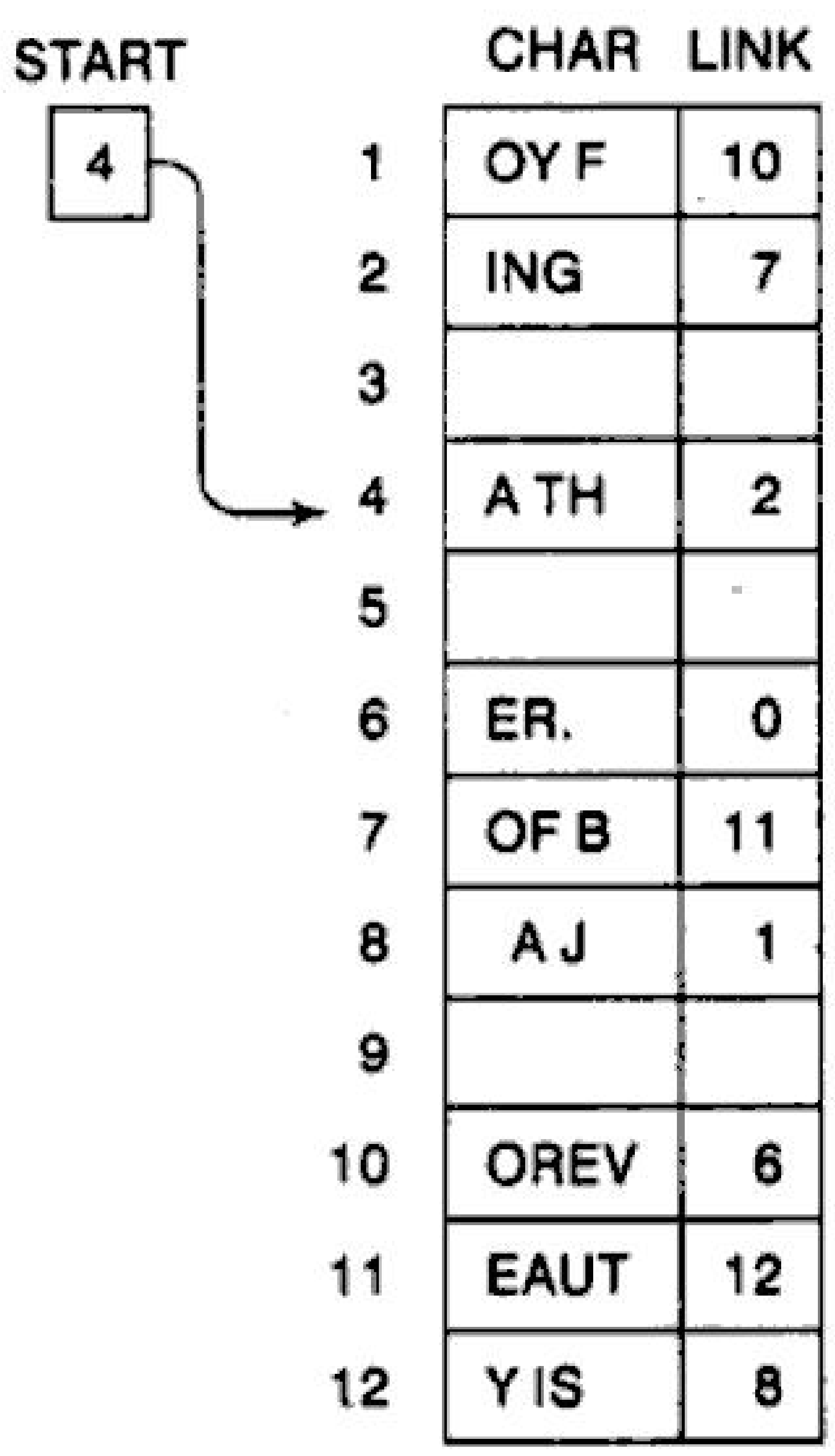


Fig. 3.9

Thus the string is:

A THING OF BEAUTY IS A JOY FOREVER.

3.5 Give some (a) advantages and (b) disadvantages of using linked storage for storing strings.

(a) One can easily insert, delete, concatenate and rearrange substrings when using linked storage.
(b) Additional space is used for storing the links. Also, one cannot directly access a character in the middle of the list.

3.6 Describe briefly the meaning of (a) static, (b) semistatic and (c) dynamic character variables.

(a) The length of the variable is defined before the program is executed and cannot change during the execution of the program.
(b) The length of the variable may vary during the execution of the program, but the length cannot exceed a maximum value defined before the program is executed.
(c) The length of the variable may vary during the execution of the program.

3.7 Suppose MEMBER is a character variable with fixed length 20. Assume a string is stored left-justified in a memory cell with blank spaces padded on the right or with the right-most characters truncated. Describe MEMBER (a) if 'JOHN PAUL JONES' is assigned to MEMBER and (b) if 'ROBERT ANDREW WASHINGTON' is assigned to MEMBER.

The data will appear in MEMBER as follows:

(a) MEMBER

J	O	H	N		P	A	U	L		J	O	N	E	S					

(b) MEMBER

R	O	B	E	R	T		A	N	D	R	E	W		W	A	S	H	I	N

String Operations

In Solved Problems 3.8 to 3.11 and 3.13, let S and T be character variables such that

S = 'JOHN PAUL JONES'
T = 'A THING OF BEAUTY IS A JOY FOREVER.'

3.8 Recall that we use LENGTH(string) for the length of a string.
(a) How is this function denoted in C?
(b) Find LENGTH(S) and LENGTH(T).

(a) In C, the length of a string can be determined using the strlen library function, as shown below:

```
strlen(string); // this function call will return an integer value
                   equal to the length of the string
```

(b) Assuming there is only one blank space character between words,

LENGTH(S) = 15 and LENGTH(T) = 35

3.9 Recall that we use SUBSTRING(string, position, length) to denote the substring of *string* beginning in a given *position* and having a given *length*. Determine (a) SUBSTRINGS(S, 4, 8) and (b) SUBSTRING(T, 10, 5).

(a) Beginning with the fourth character and recording 8 characters, we obtain

SUBSTRING(S, 4, 8) = 'N□PAUL□J'

(b) Similarly, SUBSTRING(T, 10, 5) = ' F□BEAU'

3.10 Recall that we use INDEX(text, pattern) to denote the position where a pattern first appears in a text. This function is assigned the value 0 if the pattern does not appear in the text. Determine (a) INDEX(S, 'JO'), (b) INDEX(S, 'JOY'), (c) INDEX(S, '□JO'), (d) INDEX(T, 'A'), (e) INDEX(T, '□A□') and (f) INDEX(T, 'THE').

(a) INDEX(S, 'JO') = 1, (b) INDEX(S, 'JOY') = 0, (c) INDEX(S, ' □JO') = 10, (d) INDEX(T, 'A') = 1, (e) INDEX(T, '□A□') = 21 and (f) INDEX(T, 'THE') = 0. (Recall that □ is used to denote a blank space.)

3.11 Recall that we use $S_1//S_2$ to denote the concatenation of strings S_1 and S_2.

(a) How is this function denoted in C?
(b) Find (i) 'THE' // 'END' and (ii) 'THE' // '□'//'END'.s
(c) Find (i) SUBSTRING(S, 11, 5)//',□'//SUBSTRING(S, 1, 9) and (ii) SUBSTRING (T, 28, 3)//'GIVEN'.

(a) In C, the concatenation of strings is performed using the strcat libary function, as shown below:

```
strcat(S1,S2);  // this function call will concatenate strings S1
                   and S2 and store the result in S1
```

(b) $S_1//S_2$ refers to the string consisting of the characters of S_1 followed by the characters of S_2, Hence, (i) THEEND and (ii) THE END.
(c) (i) JONES, JOHN PAUL and (ii) FORGIVEN.

3.12 Recall that we use INSERT(text, position, string) to denote inserting a *string* S in a given *text* T beginning in *position* K.

(a) Find (i) INSERT('AAAAA', 1, 'BBB'), (ii) INSERT('AAAAA', 3, 'BBB') and (iii) INSERT('AAAAA', 6, 'BBB').
(b) Suppose T is the text 'THE STUDENT IS ILL.' Use INSERT to change T so that it reads: (i) The student is very ill. (ii) The student is ill today. (iii) The student is very ill today.

(a) (i) BBBAAAAA, (ii) AABBBAAA and (iii) AAAAABBB.
(b) Be careful to include blank spaces when necessary. (i) INSERT(T, 15, '□VERY'). (ii) INSERT(T, 19, '□TODAY'). (iii) INSERT(INSERT(T, 19, '□TODAY'), 15, '□VERY') or INSERT(INSERT(T, 15, '□VERY'), 24, '□TODAY').

3.13 Find

(a) DELETE('AAABBB', 2, 2) and DELETE('JOHN PAUL JONES', 6, 5)
(b) REPLACE('AAABBB', 'AA', 'BB') and
REPLACE('JOHN PAUL JONES', 'PAUL', 'DAVID')

(a) DELETE(T, K, L) deletes from a text T the substring which begins in position K and has length L. Hence the answers are

ABBB and JOHN JONES

(b) REPLACE(T, P_1, P_2) replaces in a text T the first occurrence of the pattern P_1 by the pattern P_2. Hence the answers are

BBABBB and JOHN DAVID JONES

Word/Text Processing

In Solved Problems 3.14 to 3.17, S is a short story stored in a linear array LINE with *n* elements such that each LINE[K] is a static character variable storing 80 characters and representing a line of the story. Also, LINE[1], the first line, contains only the title of the story, and LINE[N], the last line, contains only the name of the author. Furthermore, each paragraph begins with 5 blank spaces, and there is no other indention except possibly the title in LINE[1] or the name of the author in LINE[N].

3.14 Write a procedure which counts the number NUM of paragraphs in the short story S.

Beginning with LINE[2] and ending with LINE[N – 1], count the number of lines beginning with 5 blank spaces. The procedure follows.

Procedure P3.14: PAR(LINE, N, NUM)
1. Set NUM := 0 and BLANK := '□□□□□'
2. [Initialize counter.] Set K := 2.
3. Repeat Steps 4 and 5 while K ≤ N – 1.
4. [Compare first 5 characters of each line with BLANK.]
   If SUBSTRING(LINE[K], 1, 5) = BLANK, then:
      Set NUM := NUM + 1.
   [End of If structure.)
5. Set K := K + 1. [Increments counter.]
   [End of Step 3 loop.]
6. Return.

Program 3.8

```
/* C implementation of Procedure P3.14 */
#include <stdio.h>
#include <conio.h>
#include <string.h>

char S[10][80]={{"This is the story of a boy"},
     {"who lived in Delhi"},
     {"     His name was Rohan"},
     {"He was studying engineering"},
     {"     His favorite subject was DS"},
     {"He was good in string handling"}};
int PAR(char[][80],int,int);
char* SUBSTR(char*,int,int);
```

```
void main()
{
int NUM,N;
clrscr();

NUM=0;
N=6;

printf("The number of paragraphs in short story S are %d",PAR(S,N,NUM));
getch();
}

int PAR(char S1[][80],int N1,int NUM1)
{
 int K;
 int i;
 char BLANK[6]={"     "};
 char TEMP[80];

 K=0;
 while(K<N1-1)
 {
  i=0;

  while(S1[K][i]!='\0')
  {
    TEMP[i]=S1[K][i];
    i=i+1;
  }

  if(strcmp(SUBSTR(TEMP,1,5),BLANK)==0)
   NUM1=NUM1+1;
  K=K+1;
 }
 return(NUM1);
}

char* SUBSTR(char *STR,int i,int j)
{
 int k,m=0;
 char STRRES[80];
 for(k=i-1;k<=i+j-1-1;k++)
 {
  STRRES[m]=STR[k];
  m=m+1;
```

```
  }
   STRRES[m]='\0';
  return(STRRES);
}
```

Output:

```
The number of paragraphs in short story S are 2
```

3.15 Write a procedure which counts the number NUM of times the word "the" appears in the short story S. (We do not count "the" in "mother," and we assume no sentence ends with the word "the."

Note that the word "the" can appear as THE☐ at the beginning of a line, as ☐THE at the end of a line, or as ☐THE☐ elsewhere in a line. Hence we must check these three cases for each line. The procedure follows.

Procedure P3.15: COUNT(LINE, N, NUM)

1. Set WORD := 'THE' and NUM := 0.
2. [Prepare for the three cases.]
   Set BEG := WORD//'☐', END := '☐'//WORD and
   MID := '☐' //WORD// '☐'.
3. Repeat Steps 4 through 6 for K = 1 to N:
4.   [First case.] If SUBSTRING(LINE[K], 1, 4) = BEG, then:
           Set NUM := NUM + 1.
5.   [Second case.] If SUBSTRING(LINE[K], 77, 4) = END, then:
           Set NUM := NUM + 1.
6.   [General case.] Repeat for J = 2 to 76.
           If SUBSTRING(LINE[K], J, 5) = MID, then:
               Set NUM := NUM + 1.
           [End of If structure.]
       [End of Step 6 loop.]
   [End of Step 3 loop.]
7. Return.

Program 3.9

```
/* C implementation of Procedure 3.15 */
#include <stdio.h>
#include <conio.h>
#include <string.h>

char S[10][80]={{"This is the story of a boy"},
     {"who lived in Delhi"},
     {"     His name was Rohan"},
     {"THE Dummy Text: Checking THE use of THE string THE"},
```

```
    {"       His favorite subject was DS"},
    {"He was good in string handling"}};
int COUNT(char[][80],int,int);
char* SUBSTR(char*,int,int);

void main()
{
int NUM,N;
clrscr();

NUM=0;
N=6;

printf("The number of instances of WORD in short story S are
%d",COUNT(S,N,NUM));
getch();
}

int COUNT(char S1[][80],int N1,int NUM1)
{
 int K;
 int i,J;
 char BEG[10]={"THE "};
 char END[10]={" THE"};
 char MID[10]={" THE "};
 char TEMP[80];

 K=0;
 while(K<N1)
 {
  i=0;

  while(S1[K][i]!='\0')
  {
    TEMP[i]=S1[K][i];
    i=i+1;
  }
  TEMP[i]='\0';
  if(strcmp(SUBSTR(TEMP,1,4),BEG)==0)
   NUM1=NUM1+1;
  if(strcmp(SUBSTR(TEMP,strlen(TEMP)-3,4),END)==0)
   NUM1=NUM1+1;
  for(J=2;J<strlen(TEMP)-5;J++)
  if(strcmp(SUBSTR(TEMP,J,5),MID)==0)
   NUM1=NUM1+1;
```

```
      K=K+1;
   }
   return(NUM1);
}

char* SUBSTR(char *STR,int i,int j)
{
   int k,m=0;
   char STRRES[80];
   for(k=i-1;k<=i+j-1-1;k++)
   {
     STRRES[m]=STR[k];
     m=m+1;
   }
     STRRES[m]='\0';
   return(STRRES);
}
```

Output:

```
The number of instances of WORD in short story S are 4
```

3.16 Discuss the changes that must be made in Procedure P3.15 if one wants to count the number of occurrences of an aribitrary word W with length R.

There are three basic types of changes.

(a) Clearly, 'THE' must be changed to W in Step 1.

(b) Since the length of W is *r* and not 3, appropriate changes must be made in Steps 3 to 6.

(c) One must also consider the possibility that W will be followed by some punctuation, e.g.,

W, W; W. W?

Hence more than the three cases must be treated.

3.17 Outline an algorithm which will interchange the *k*th and *l*th paragraphs in the short story S.

The algorithm reduces to two procedures:

Procedure A. Find the values of arrays BEG and END where

LINE[BEG[K]] and LINE[END[K]]

contain, respectively, the first and last lines of paragraph K of the story S.

Procedure B. Using the values of BEG[K] and END[K] and the values of BEG[L] and END[L], interchange the block of lines of paragraph K with the block of lines of paragraph L.

Pattern Matching

3.18 For each of the following patterns P and texts T, find the number C of comparisons to find the INDEX of P in T using the "slow" algorithm, Algorithm 3.3:
(a) P = abc, T = $(ab)^5$ = $ababababab$ (b) P = abc, T = $(ab)^{2n}$
(c) P = aaa, T = $(aabb)^3$ = $aabbaabbaabb$ (d) P = aaa, T = $abaabbaaabbbaaaabbbb$

Recall that $C = N_1 + N_2 + \cdots + N_k$ where N_k denotes the number of comparisons that take place in the inner loop when P is compared with W_K.

(a) Note first that there are

$$\text{LENGTH(T)} - \text{LENGTH(P)} + 1 = 10 - 3 + 1 = 8$$

substrings W_K. We have

$$C = 2 + 1 + 2 + 1 + 2 + 1 + 2 + 1 = 4(3) = 12$$

and INDEX(T, P) = 0, since P does not appear in T.

(b) There are $2n - 3 + 1 = 2(n - 1)$ subwords W_K. We have

$$C = 2 + 1 + 2 + 1 + \cdots + 2 + 1 = (n + 1)(3) = 3n + 3$$

and INDEX(T, P) = 0.

(c) There are $12 - 3 + 1 = 10$ subwords W_K. We have

$$C = 3 + 2 + 1 + 1 + 3 + 2 + 1 + 1 + 3 + 2 = 19$$

and INDEX(T, P) = 0.

(d) We have

$$C = 2 + 1 + 3 + 2 + 1 + 1 + 3 = 13$$

and INDEX(T, P) = 7.

3.19 Suppose P is an r-character string and T is an s-character string, and suppose $C(n)$ denotes the number of comparisons when Algorithm 3.3 is applied to P and T. (Here $n = r + s$.)
(a) Find the complexity $C(n)$ for the best case.
(b) Prove that the maximum value of $C(n)$ occurs when $r = (n + 1)/4$.

(a) The best case occurs when P is an initial substring of T, or, in other words, when INDEX(T, P) = 1. In this case $C(n) = r$. (We assume $r \leq s$.)

(b) By the discussion in Sec. 3.7,

$$C = C(n) = r(n - 2r + 1) = nr - 2r^2 + r$$

Here n is fixed, so $C = C(n)$ may be viewed as a function of r. Calculus tells us that the maximum value of C occurs when $C' = dc/dr = 0$ (here C' is the derivative of C with respect to r). Using calculus, we obtain:

$$C' = n - 4r + 1$$

Setting $C' = 0$ and solving for r gives us the required result.

3.20 Consider the pattern $P = aaabb$. Construct the table and the corresponding **labeled directed** graph used in the "fast," or second pattern matching, algorithm.

First list the initial segments of P:

$$Q_0 = \Lambda,\quad Q_1 = a,\quad Q_2 = a^2,\quad Q_3 = a^3,\quad Q_4 = a^3b,\quad Q_5 = a^3b^2$$

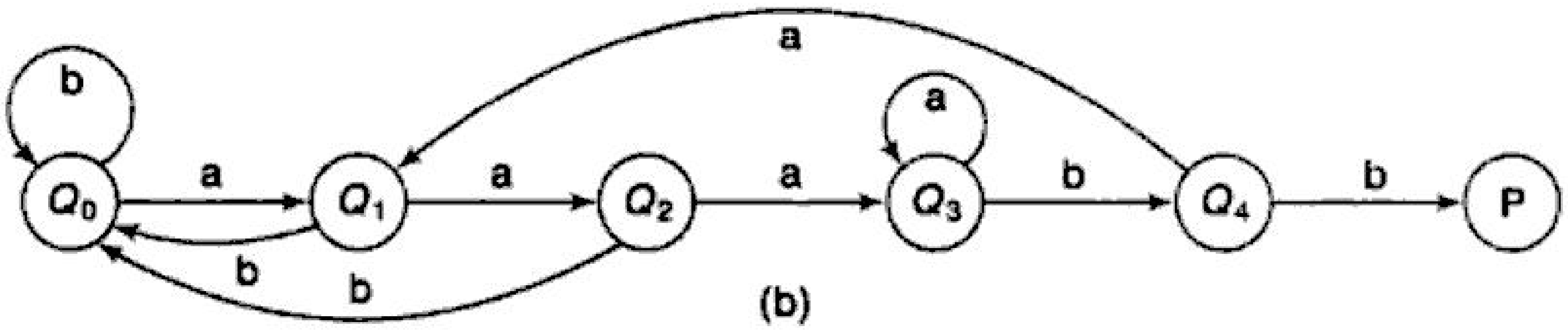
For each character t, the entry $f(Q_i, t)$ in the table is the largest Q which appears as a terminal substring in the string Q_it. We compute:

$$f(\Lambda, a) = a,\quad f(a, a) = a^2,\quad f(a^2, a) = a^3,\quad f(a^3, a) = a^3,\quad f(a^3b, a) = a$$
$$f(\Lambda, b) = \Lambda,\quad f(a, b) = \Lambda,\quad f(a^2, b) = \Lambda,\quad f(a^3, b) = a^3b,\quad f(a^3b, b) = P$$

Hence the required table appears in Fig. 3.10(a). The corresponding graph appears in Fig. 3.10(b), where there is a node corresponding to each Q and an arrow from Q_i to Q_j labeled by the character t for each entry $f(Q_i, t) = Q_j$ in the table.

	a	b
Q_0	Q_1	Q_0
Q_1	Q_2	Q_0
Q_2	Q_3	Q_0
Q_3	Q_3	Q_4
Q_4	Q_1	P

(a)



(b)

Fig. 3.10

3.21 Find the table and corresponding graph for the second pattern matching algorithm where the pattern is $P = ababab$.

The initial substrings of P are:

$$Q_0 = \Lambda,\quad Q_1 = a,\quad Q_2 = ab,\quad Q_3 = aba,\quad Q_4 = abab,\quad Q_5 = ababa,\quad Q_6 = ababab = P$$

The function f giving the entries in the table follows.

$$f(\Lambda, a) = a \qquad f(\Lambda, b) = \Lambda$$
$$f(a, a) = a \qquad f(a, b) = ab$$
$$f(ab, a) = aba \qquad f(ab, b) = \Lambda$$
$$f(aba, a) = a \qquad f(aba, b) = abab$$
$$f(abab, a) = ababa \qquad f(abab, b) = \Lambda$$
$$f(ababa, a) = a \qquad f(ababa, b) = P$$

The table appears in Fig. 3.11(a) and the corresponding graph appears in Fig. 3.11(b).

3.4 Suppose STATE is a character variable with fixed length 12. Describe the contents of STATE after the assignment (a) STATE := 'NEW YORK', (b) STATE := 'SOUTH CAROLINA' and (c) STATE := 'PENNSYLVANIA'.

String Operations

In Supplementary Problems 3.5 to 3.10, let S and T be character variables such that

S = 'WE THE PEOPLE' and T = 'OF THE UNITED STATES'

Implement the problems using C programs

3.5 Find the length of S and T.

3.6 Find (a) SUBSTRING(S, 4, 8) and (b) SUBSTRING(T, 10, 5).

3.7 Find (a) INDEX(S, 'P'), (b) INDEX(S, 'E'), (c) INDEX(S, 'THE'), (d) INDEX(T, 'THE'), (e) INDEX(T, 'THEN') and (f) INDEX(T, 'TE').

3.8 Using $S_1//S_2$ to stand for the concatenation of S_1 and S_2, find (a) 'NO'//'EXIT', (b) 'NO'// '□' // 'EXIT' and (c) SUBSTRING(S, 4, 10)//'□ARE□' //SUBSTRING(T, 8, 6).

3.9 Find (a) DELETE('AAABBB', 3, 3), (b) DELETE('AAABBB', 1, 4), (c) DELETE(S, 1, 3) and (d) DELETE(T, 1, 7).

3.10 Find (a) REPLACE('ABABAB', 'B', 'BAB'), (b) REPLACE(S, 'WE', 'ALL') and (c) REPLACE(T, 'THE', 'THESE').

3.11 Find (a) INSERT('AAA', 2, 'BBB'), (b) INSERT('ABCDE', 3, 'XYZ') and (c) INSERT ('THE BOY', 5, 'BIG□').

3.12 Suppose U is the text 'MARC STUDIES MATHEMATICS'. Use INSERT to change U so that it reads: (a) MARC STUDIES ONLY MATHEMATICS. (b) MARC STUDIES MATHEMATICS AND PHYSICS. (c) MARC STUDIES APPLIED MATHEMATICS.

Pattern Matching

3.13 Consider the pattern $P = abc$. Using the "slow" pattern matching algorithm, Algorithm 3.3, find the number C comparisons to find the INDEX of P in each of the following texts T: (a) a^{10}, (b) $(aba)^{10}$, (c) $(cbab)^{10}$, (d) d^{10} and (e) d^n where $n > 3$.

3.14 Consider the pattern $P = a^5b$. Repeat Problem 3.34 with each of the following texts T: (a) a^{20}, (b) a^n where $n > 6$; (c) d^{20} and (d) d^n where $n > 6$.

3.15 Consider the pattern $P = a^3ba$. Construct the table and the corresponding labeled directed graph used in the "fast" pattern matching algorithm.

3.16 Repeat Problem 3.15 for the pattern $P = aba^2b$.

PROGRAMMING PROBLEMS

In Programming Problems 3.1 to 3.3, assume the preface of this text is stored in a linear array LINE such that LINE[K] is a static character variable storing 80 characters and represents a line of the preface. Assume that each paragraph begins with 5 blank spaces and there is no other indention. Also, assume there is a variable NUM which gives the number of lines in the preface.

3.1 Write a program which defines a linear array PAR such that PAR[K] contains the location of the Kth paragraph, and which also defines a variable NPAR which contains the number of paragraphs.

3.2 Write a program which reads a given WORD and then counts the number *C* of times WORD occurs in LINE. Test the program using (a) WORD = 'THE' and (b) WORD = 'HENCE'.

3.3 Write a program which interchanges the Jth and Kth paragraphs. Test the program using J = 2 and K = 4.

In Programming Problems 3.4 to 3.9, assume the preface of this text is stored in a single character variable TEXT. Assume 5 blank spaces indicates a new paragraph.

3.4 Write a program which constructs a linear array PAR such that PAR[K] contains the location of the Kth paragraph in TEXT, and which finds the value of a variable NPAR which contains the number of paragraphs. (Compare with Programming Problem 3.1.)

3.5 Write a program which reads a given WORD and then counts the number *C* of times WORD occurs in TEXT. Test the program using (a) WORD = 'THE' and (b) WORD = 'HENCE'. (Compare with Programming Problem 3.2.)

3.6 Write a program which interchanges the Jth and Kth paragraphs in TEXT. Test the program using J = 2 and K = 4. (Compare with Programming Problem 3.3.)

3.7 Write a program which reads words WORD1 and WORD2 and then replaces each occurrence of WORD1 in TEXT by WORD2. Test the program using WORD1 = 'HENCE' and WORD2 = 'THUS'

3.8 Write a subprogram INST(TEXT, NEW, K) which inserts a string NEW into TEXT beginning at TEXT[K].

3.9 Write a subprogram PRINT(TEXT, K) which prints the character string TEXT in lines with at most K characters. No word should be divided in the middle and appear on two lines, so some lines may contain trailing blank spaces. Each paragraph should begin with its own line and be indented using 5 blank spaces. Test the program using (a) K = 800, (b) K = 70 and (c) K = 60.

3.10 Write a program to find the distance between two character strings.

3.11 Write a program with three short strings, about 6 characters each, and use strcpy to copy one, two, and three into them. Concatenate the three strings into one string and print the result out 10 times.

MULTIPLE CHOICE QUESTIONS

3.1 Computers are used for processing numerical data called ______ data.
(a) Float (b) Local
(c) Character (d) Nonlocal

3.2 Each programming language contains a ______ set that is used to communicate with the computer.
(a) Character (b) Integer
(c) Float (d) Numeric

3.3 Finite sequence S of zero or more characters is called ______.
(a) Array (b) List
(c) String (d) Block

3.4 String with zero characters is called ______ string.
(a) NULL (b) Binary
(c) Totalled (d) List

3.5 A computer which can access an

Chapter 4

Arrays, Records and Pointers

4.1 INTRODUCTION

Data structures are classified as either linear or nonlinear. A data structure is said to be linear if its elements form a sequence, or, in other words, a linear list. There are two basic ways of representing such linear structures in memory. One way is to have the linear relationship between the elements represented by means of sequential memory locations. These linear structures are called *arrays* and form the main subject matter of this chapter. The other way is to have the linear relationship between the elements represented by means of pointers or links. These linear structures are called *linked lists*; they form the main content of Chapter 5. Nonlinear structures such as trees and graphs are treated in later chapters.

The operations one normally performs on any linear structure, whether it be an array or a linked list, include the following:

(a) *Traversal*. Processing each element in the list.
(b) *Search*. Finding the location of the element with a given value or the record with a given key.
(c) *Insertion*. Adding a new element to the list.
(d) *Deletion*. Removing an element from the list.
(e) *Sorting*. Arranging the elements in some type of order.
(f) *Merging*. Combining two lists into a single list.

The particular linear structure that one chooses for a given situation depends on the relative frequency with which one performs these different operations on the structure.

This chapter discusses a very common linear structure called an array. Since arrays are usually easy to traverse, search and sort, they are frequently used to store relatively permanent collections of data. On the other hand, if the size of the structure and the data in the structure are constantly changing, then the array may not be as useful a structure as the linked list, discussed in Chapter 5.

to begin the index set with 1932 so that

AUTO[K] = number of automobiles sold in the year K

Then LB = 1932 is the lower bound and UB = 1984 is the upper bound of AUTO. By Eq. (4.1),

Length = UB − LB + 1 = 1984 − 1930 + 1 = 55

That is, AUTO contains 55 elements and its index set consists of all integers from 1932 through 1984.

Program 4.1

```
/*
Defining Arrays in C*/
#include <stdio.h>
main()
{
  int a[10]; //1
  for(int i = 0;i<10;i++)
  {
    a[i]=i;
  }
  printaray(a);
}
void printaray(int a[])
{
  for(int i = 0;i<10;i++)
  {
    printf("Value in the array %d\n",a[i]);
  }
}
```

The above program helps to define an array. Statement 1 defines an array of integers of the size 10, which means you can store 10 integers. When we define the array, the size should be known. Subscripts are used to refer the elements of the array where 0 is considered to be the lowest subscript always and the highest subscript is (size −1), which is 9 in this case. We can refer to any element as a[0], a[1], a[2], etc.

An array can also be processed using a for loop. The consecutive memory locations of the array are allocated and the element size is the same. We should always keep in mind that among all the operators used the subscript of this array must have the highest precedence.

Each programming language has its own rules for declaring arrays. Each such declaration must give, implicitly or explicitly, three items of information: (1) the name of the array, (2) the data type of the array and (3) the index set of the array.

Example 4.2

(a) Suppose DATA is a 6-element linear array containing real values. C language declares such an array as follows:

```
float DATA[6];
```

In each stage we can see that either r is decreased or y is increased, thus showing the algorithm is finite and the length |X| of the region X, which is $r + 1 - y$ decreases each time, and the 'sorted' condition

B < Y < X < R

remains true. When X = 0, the algorithm terminates, and thus we get our sorted list. The loop executed N number of times there are |B| + |R| calls to the routine swap.

4.4 REPRESENTATION OF LINEAR ARRAYS IN MEMORY

Let LA be a linear array in the memory of the computer. Recall that the memory of the computer is simply a sequence of addressed locations as pictured in Fig. 4.3. Let us use the notation

LOC(LA[K]) = address of the element LA[K] of the array LA

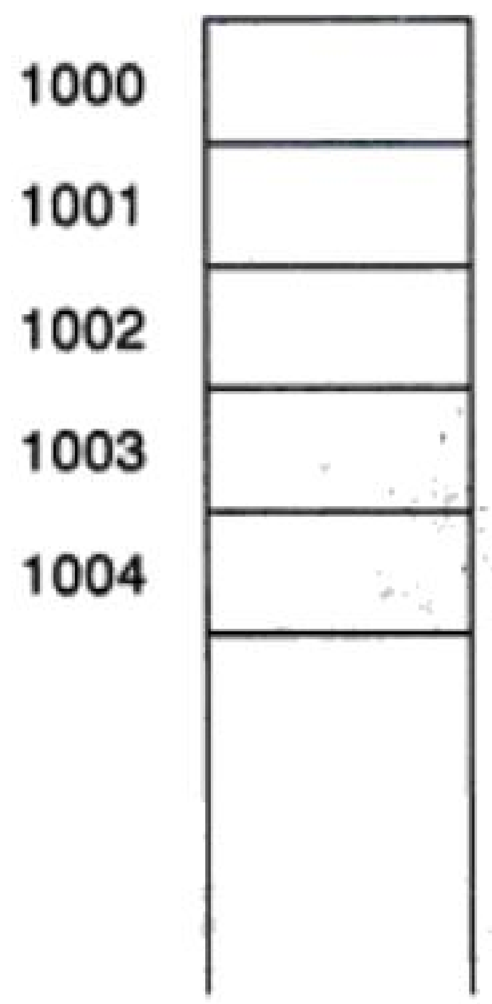


Fig. 4.3 Computer Memory

As previously noted, the elements of LA are stored in successive memory cells. Accordingly, the computer does not need to keep track of the address of every element of LA, but needs to keep track only of the address of the first element of LA, denoted by

Base(LA)

and called the *base address* of LA. Using this address *Base*(LA), the computer calculates the address of any element of LA by the following formula:

$$LOC\ (LA[K]) = Base(LA) + w(K - \text{lower bound}) \qquad (4.2)$$

where w is the number of words per memory cell for the array LA. Observe that the time to calculate LOC(LA[K]) is essentially the same for any value of K. Furthermore, given any subscript K, one can locate and access the content of LA[K] without scanning any other element of LA.

Example 4.4

Consider the array AUTO in Example 4.1(b), which records the number of automobiles sold each year from 1932 through 1984. Suppose AUTO appears in memory as pictured in Fig. 4.4. That is, *Base*(AUTO) = 200, and w = 4 words per memory cell for AUTO. Then

$$\text{LOC(AUTO[1932])} = 200, \quad \text{LOC(AUTO[1933])} = 204, \quad \text{LOC(AUTO[1934])} = 208, \ldots$$

The address of the array element for the year K = 1965 can be obtained by using Eq. (4.2):

$$\begin{aligned}\text{LOC(AUTO[1965])} &= \textit{Base}\text{(AUTO)} + w(1965 - \text{lower bound})\\ &= 200 + 4(1965 - 1932) = 332\end{aligned}$$

Again we emphasize that the contents of this element can be obtained without scanning any other element in array AUTO.

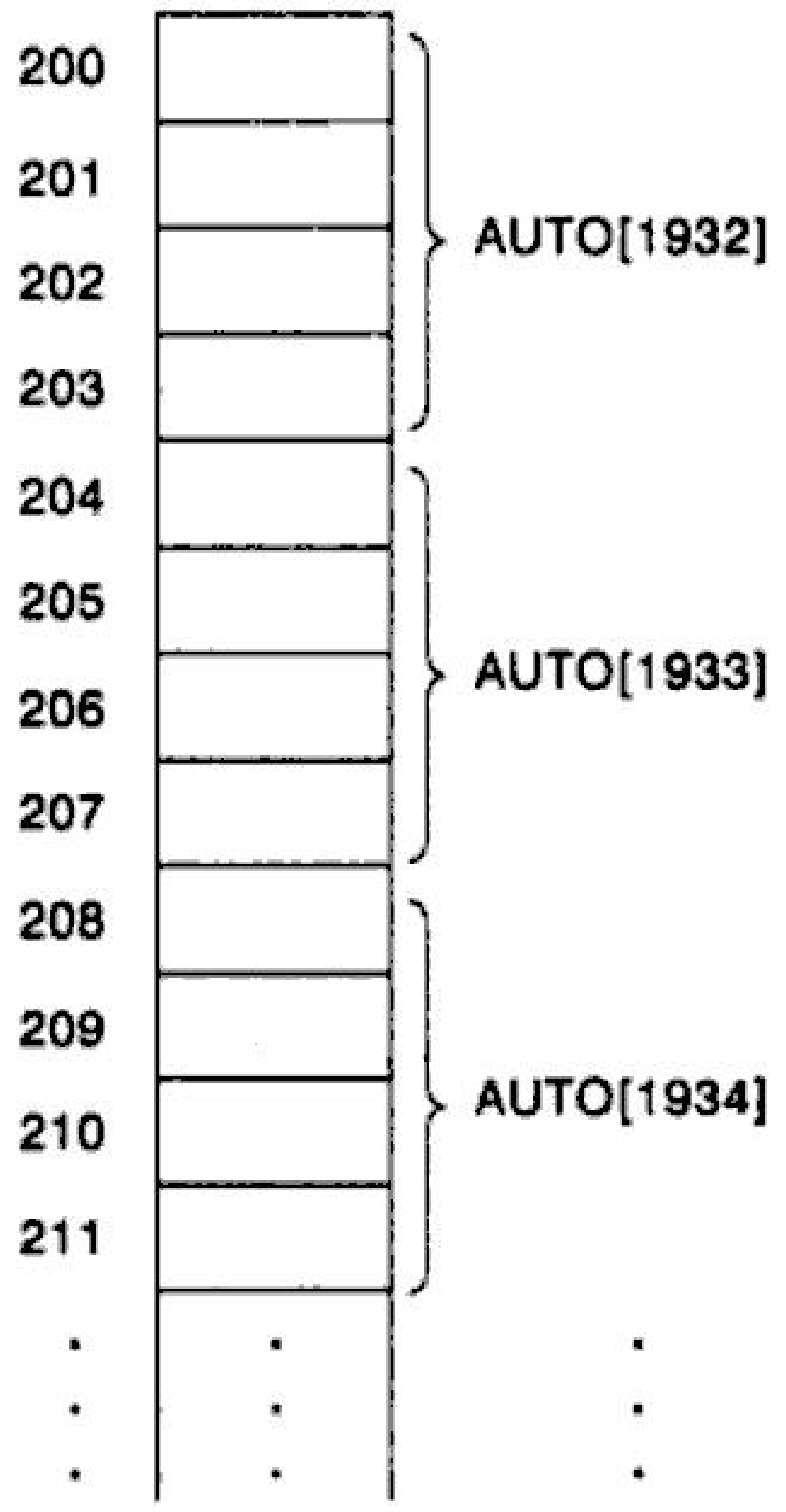


Fig. 4.4

Program 4.3

There is a memory address for all the elements of an array. In the following program, we can see how an array limit value and an array element address are printed.

```
#include <stdio.h>
void printaray(int x[]);
main()
```

```
{
  int x[15];
  for(int i = 0;i<15;i++)
  {
    x[i]=i;
  }
  printaray(x);
}
void printaray(int x[])
{
  for(int i = 0;i<15;i++)
  {
    printf("Value in the array %d\n",x[i]);
  }
}
void printdetail(int x[])
{
  for(int i = 0;i<15;i++)
  {
    printf("Value in the array %d and address is %16lu\n",x[i],&x[i]);
  }
}
```

In the above program, the function 'printaray' is used to print values of each element in the array 'aray'. To print the value and address of all the elements and its address the printdetail function is used.

Now that we know all the elements are integer type, the difference between address is 2.

All the elements in the array are placed in consecutive memory spaces. The memory addresses can be printed using place holders %16lu or %p.

Remark: A collection A of data elements is said to be *indexed* if any element of A, which we shall call A_K, can be located and processed in a time that is independent of K. The above discussion indicates that linear arrays can be indexed. This is very important property of linear arrays. In fact, linked lists, which are covered in the next chapter, do not have this property.

4.5 TRAVERSING LINEAR ARRAYS

Let A be a collection of data elements stored in the memory of the computer. Suppose we want to print the contents of each element of A or suppose we want to count the number of elements of A with a given property. This can be accomplished by *traversing* A, that is, by accessing and processing (frequently called *visiting)* each element of A exactly once.

The following algorithm traverses a linear array LA. The simplicity of the algorithm comes from the fact that LA is a linear structure. Other linear structures, such as linked lists, can also be easily traversed. On the other hand, the traversal of nonlinear structures, such as trees and graphs, is considerably more complicated.

Example 4.5

Consider the array AUTO in Example 4.1(b), which records the number of automobiles sold each year from 1932 through 1984. Each of the following modules, which carry out the given operation, involves traversing AUTO.

(a) Find the number NUM of years during which more than 300 automobiles were sold.

```
1. [Initialization step.] Set NUM := 0.
2. Repeat for K = 1932 to 1984:
        If AUTO[K] > 300, then: Set NUM := NUM + 1.
   [End of loop.]
3. Return.
```

(b) Print each year and the number of automobiles sold in that year.

```
1. Repeat for K = 1932 to 1984:
        Write: K, AUTO[K].
   [End of loop.]
2. Return.
```

(Observe that (a) requires an initialization step for the variable NUM before traversing the array AUTO.)

4.6 INSERTING AND DELETING

Let A be a collection of data elements in the memory of the computer. "Inserting" refers to the operation of adding another element to the collection A, and "deleting" refers to the operation of removing one of the elements from A. This section discusses inserting and deleting when A is a linear array.

Inserting an element at the "end" of a linear array can be easily done provided the memory space allocated for the array is large enough to accommodate the additional element. On the other hand, suppose we need to insert an element in the middle of the array. Then, on the average, half of the elements must be moved downward to new locations to accommodate the new element and keep the order of the other elements.

Similarly, deleting an element at the "end" of an array presents no difficulties, but deleting an element somewhere in the middle of the array would require that each subsequent element be moved one location upward in order to "fill up" the array.

Remark: Since linear arrays are usually pictured extending downward, as in Fig. 4.1, the term "downward" refers to locations with larger subscripts, and the term "upward" refers to locations with smaller subscripts.

Example 4.6

Suppose TEST has been declared to be a 5-element array but data have been recorded only for TEST[1], TEST[2] and TEST[3]. If X is the value of the next test, then one simply assigns

TEST[4] := X

to add X to the list. Similarly, if Y is the value of the subsequent test, then we simply assign

TEST[5] := Y

to add Y to the list. Now, however, we cannot add any new test scores to the list.

Example 4.7

Suppose NAME is an 8-element linear array, and suppose five names are in the array, as in Fig. 4.5(a). Observe that the names are listed alphabetically, and suppose we want to keep the array names alphabetical at all times. Suppose Ford is added to the array. Then Johnson, Smith and Wagner must each be moved downward one location, as in Fig. 4.5(b). Next suppose Taylor is added to the array; then Wagner must be moved, as in Fig. 4.5(c). Last, suppose Davis is removed from the array. Then the five names Ford, Johnson, Smith, Taylor and Wagner must each be moved upward one location, as in Fig. 4.5(d). Clearly such movement of data would be very expensive if thousands of names were in the array.

	NAME (a)	NAME (b)	NAME (c)	NAME (d)
1	Brown	Brown	Brown	Brown
2	Davis	Davis	Davis	Ford
3	Johnson	Ford	Ford	Johnson
4	Smith	Johnson	Johnson	Smith
5	Wagner	Smith	Smith	Taylor
6		Wagner	Taylor	Wagner
7			Wagner	
8				

Fig. 4.5

The following algorithm inserts a data element ITEM into the Kth position in a linear array LA with N elements. The first four steps create space in LA by moving downward one location each element from the Kth position on. We emphasize that these elements are moved in reverse order—i.e., first LA[N], then LA[N – 1], …, and last LA[K]; otherwise data might be erased. (See Solved Problem 4.3) In more detail, we first set J := N and then, using J as a counter, decrease J each time the loop is executed until J reaches K. The next step, Step 5, inserts ITEM into the array in the space just created. Before the exit from the algorithm, the number N of elements in LA is increased by 1 to account for the new element.

Algorithm 4.2: (Inserting into a Linear Array) INSERT (LA, N, K, ITEM)
Here LA is a linear array with N elements and K is a positive integer such that $K \le N$. This algorithm inserts an element ITEM into the Kth position in LA.

1. [Initialize counter.] Set J : = N.
2. Repeat Steps 3 and 4 while $J \ge K$.
3.   [Move Jth element downward.] Set LA[J + 1] := LA[J].
4.   [Decrease counter.] Set J := J – 1.
   [End of Step 2 loop.]
5. [Insert element.] Set LA[K] := ITEM.
6. [Reset N.] Set N := N + 1.
7. Exit.

The following algorithm deletes the Kth element from a linear array LA and assigns it to a variable ITEM.

Algorithm 4.3: (Deleting from a Linear Array) DELETE(LA, N, K, ITEM)
Here LA is a linear array with N elements and K is a positive integer such that $K \leq N$. This algorithm deletes the Kth element from LA.

1. Set ITEM := LA[K].
2. Repeat for J = K to N – 1:
   [Move J + 1st element upward.] Set LA[J] := LA[J + 1].
   [End of loop.]
3. [Reset the number N of elements in LA.] Set N: = N – 1.
4. Exit.

Remark: We emphasize that if many deletions and insertions are to be made in a collection of data elements, then a linear array may not be the most efficient way of storing the data.

The following C program implements Algorithms 4.2 and 4.3:

Program 4.5

```
#include <stdio.h>
#include <conio.h>
# define UB 10

int array[UB]={21,2,43,14,-5,46,87,8};
int insert_item(int LA[], int N, int k, int item);
int delete_item(int LA[], int N, int k);

void main()
{
int ITEM, LOC;
int i, size=8;
int choice;
clrscr();

printf("Array: ");
for(i=0;i<size;i++)
printf("%d ",array[i]);

printf("\n\nEnter your choice: \n\n1. Insert an element\n2. Delete an
element\n  ");
scanf("%d",&choice);

if(choice!=1 && choice !=2)
{
 printf("\nInvalid Choice");
 getch();
```

```
 int j=N;
 while(j>=k-1)
 {
  LA[j+1]=LA[j];
  j--;
 }
 LA[k-1]=item;
 return(N+1);
}

int delete_item(int LA[], int N, int k)
{
 int j, item;
 item=LA[k-1];
 printf("\nItem %d deleted from location %d\n",item,k);
 for(j=k-1;j<N-1;j++)
  LA[j]=LA[j+1];
 return(N-1);
}
```

Output (Insertion):

```
Array: 21 2 43 14 -5 46 87 8
Enter your choice:
1. Insert an element
2. Delete an element
  1

Enter the element to be inserted in the array: 99
Enter the location where element 99 is to be inserted: 4
Modified array: 21 2 43 99 14 -5 46 87 8
```

Output (Deletion)

```
Array: 21 2 43 14 -5 46 87 8
Enter your choice:
1.  Insert an element
2.  Delete an element
    2

Enter the location from where element is to be deleted: 6
Item 46 deleted from location 6
Modified array: 21 2 43 14 -5 87 8
```

Remark: Some programmers use a bubble sort algorithm that contains a 1-bit variable FLAG (or a *logical* variable FLAG) to signal when no interchange takes place during a pass. If FLAG = 0 after any pass, then the list is already sorted and there is no need to continue. This may cut down on the number of passes. However, when using such a flag, one must initialize, change and test the variable FLAG during each pass. Hence the use of the flag is efficient only when the list originally is "almost" in sorted order.

4.8 SEARCHING; LINEAR SEARCH

Let DATA be a collection of data elements in memory, and suppose a specific ITEM of information is given. *Searching* refers to the operation of finding the location LOC of ITEM in DATA, or printing some message that ITEM does not appear there. The search is said to be *successful* if ITEM does appear in DATA and *unsuccessful* otherwise.

Frequently, one may want to add the element ITEM to DATA after an unsuccessful search for ITEM in DATA. One then uses a *search and insertion* algorithm, rather than simply a *search* algorithm; such search and insertion algorithms are discussed in the problem sections.

There are many different searching algorithms. The algorithm that one chooses generally depends on the way the information in DATA is organized. Searching is discussed in detail in Chapter 9. This section discusses a simple algorithm called *linear search,* and the next section discusses the well-known algorithm called *binary search.*

The complexity of searching algorithms is measured in terms of the number $f(n)$ of comparisons required to find ITEM in DATA where DATA contains n elements. We shall show that linear search is a linear time algorithm, but that binary search is a much more efficient algorithm, proportional in time to $\log_2 n$. On the other hand, we also discuss the drawback of relying only on the binary search algorithm.

Linear Search

Suppose DATA is a linear array with n elements. Given no other information about DATA, the most intuitive way to search for a given ITEM in DATA is to compare ITEM with each element of DATA one by one. That is, first we test whether DATA[1] = ITEM, and then we test whether DATA[2] = ITEM, and so on. This method, which traverses DATA sequentially to locate ITEM, is called *linear search* or *sequential search.*

To simplify the matter, we first assign ITEM to DATA[N + 1], the position following the last element of DATA. Then the outcome

$$\text{LOC} = \text{N} + 1$$

where LOC denotes the location where ITEM first occurs in DATA, signifies the search is unsuccessful. The purpose of this initial assignment is to avoid repeatedly testing whether or not we have reached the end of the array DATA. This way, the search must eventually "succeed."

A formal presentation of linear search is shown in Algorithm 4.5.

Observe that Step 1 guarantees that the loop in Step 3 must terminate. Without Step 1 (see Algorithm 2.4), the Repeat statement in Step 3 must be replaced by the following statement, which involves two comparisons, not one:

Repeat while LOC $\leq$ N and DATA[LOC] $\neq$ ITEM:

On the other hand, in order to use Step 1, one must guarantee that there is an unused memory

(b) Suppose ITEM = 85. The binary search for ITEM is pictured in Fig. 4.8. Here BEG, END and MID will have the following successive values:

1. Again initially, BEG = 1, END = 13, MID = 7 and DATA[MID] = 55.
2. Since 85 > 55, BEG has its value changed by BEG = MID + 1 = 8. Hence

   MID = INT[(8 + 13)/2] = 10 and so DATA[MID] = 77
3. Since 85 > 77, BEG has its value changed by BEG = MID + 1 = 11. Hence

   MID = INT[(11 + 13)/2] = 12 and so DATA[MID] = 88
4. Since 85 < 88, END has its value changed by END = MID - 1 = 11. Hence

   MID = INT[(11 + 11)/2] = 11 and so DATA[MID] = 80

   (Observe that now BEG = END = MID = 11.)

Since 85 > 80, BEG has its value changed by BEG = MID + 1 = 12. But now BEG > END. Hence ITEM does not belong to DATA.

(1) 11, 22, 30, 33, 40, 44, 55, 60, 66, 77, 80, 88, 99
(2) 11, 22, 30, 33, 40, 44, 55, 60, 66, 77, 80, 88, 99
(3) 11, 22, 30, 33, 40, 44, 55, 60, 66, 77, 80, 88, 99
(4) 11, 22, 30, 33, 40, 44, 55, 60, 66, 77, 80, 88, 99 [Unsuccessful]

Fig. 4.8 Binary Search for ITEM = 85

Complexity of the Binary Search Algorithm

The complexity is measured by the number $f(n)$ of comparisons to locate ITEM in DATA where DATA contains n elements. Observe that each comparison reduces the sample size in half. Hence we require at most $f(n)$ comparisons to locate ITEM where

$$2^{f(n)} > n \qquad \text{or equivalently} \qquad f(n) = \lfloor \log_2 n \rfloor + 1$$

That is, the running time for the worst case is approximately equal to $\log_2 n$. One can also show that the running time for the average case is approximately equal to the running time for the worst case.

Example 4.11

Suppose DATA contains 1 000 000 elements. Observe that

$$2^{10} = 1024 > 1000 \qquad \text{and hence} \qquad 2^{20} > 1000^2 = 1\ 000\ 000$$

Accordingly, using the binary search algorithm, one requires only about 20 comparisons to find the location of an item in a data array with 1 000 000 elements.

Limitations of the Binary Search Algorithm

Since the binary search algorithm is very efficient (e.g., it requires only about 20 comparisons with an initial list of 1 000 000 elements), why would one want to use any other search algorithm?

Program 4.10

```
#include <stdio.h>
#include <conio.h>

void main()
{
int A[6];
int B[6];
int C[6];
int i,flag=0;
clrscr();

for(i=0;i<6;i++)
 A[i]=B[i]=0;

A[1]=1;
A[2]=4;
A[5]=-7;

B[0]=-14;
B[1]=10;
B[2]=6;
B[3]=5;
B[4]=3;

printf("Polynomial 1 = ");
for(i=5;i>=0;i—)
{
 if(A[i]!=0 && i>0)
 {
   printf("%d.(x)%d ",A[i],i);
   flag=1;
 }

 if(i>0 && A[i-1]>0 && flag==1)
    printf("+ ");

 if(A[i]!=0 && i==0)
  printf("%d",A[i],i);
}

printf("\n\n");
flag=0;

printf("Polynomial 2 = ");
```

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

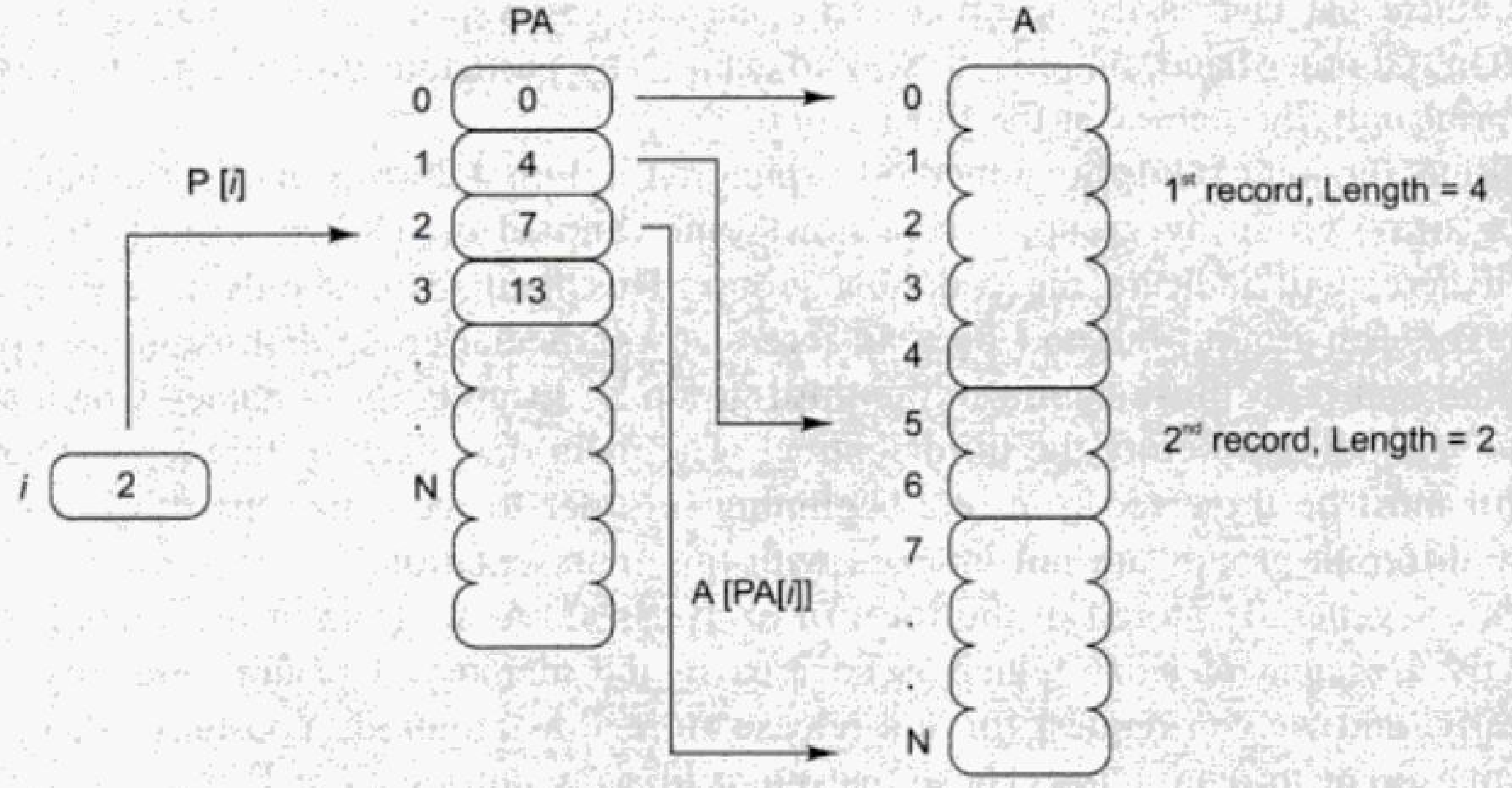


Fig. 4.19 Pointer Array PA, with Pointers Stored in A

Selecting the $(i + 1)^{th}$ record can be done by accessing PA[i] and following the pointer found there to the entry of A in which the $(i + 1)^{th}$ record begins. A[PA[i]] is the name assigned to the base of this record We should always ensure that the selection process is done first PA followed by A. where each array takes a fixed time and hence the $(i + 1)^{th}$ record is selected. Traversing through the records becomes easy by making use of PA, which can be seen in the following program.

```
traverse(A,PA,N)
/* Traverses through the N records stored in A which are pointed to by
indices stored in the pointer array PA */
int A[],PA[],N;
{
inti;
for (i=0; i<N; i++)
process(A,PA,i);
}
```

When pointer arrays are used it leads to additional storage but gives a solution for the variable-length records. All these records stored in an array have related pointers which are easy to use on violating language typing constraints.

Pointer Arrays

The two space-efficient data structures in Fig. 4.20 can be easily modified so that the individual groups can be indexed. This is accomplished by using a pointer array (here, GROUP) which contains the locations of the different groups or, more specifically, the locations of the first elements in the different groups. Figure 4.21 shows how Fig. 4.20(a) is modified. Observe that GROUP[L] and GROUP[L + 1] – 1 contain, respectively, the first and last elements in group L. (Observe that GROUP[5] points to the sentinel of the list and that GROUP[5] – 1 gives us the location of the last element in Group 4.)

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Analogously, the age of the father of the sixth newborn may be referenced by writing

Newborn.Father.Age[6]   or simply   Father.Age[6]

(c) Consider the record structure Student in Example 4.18. Since Student is declared to be a file with 20 students, all items automatically become 20-element arrays. Furthermore, Test becomes a two-dimensional array. In particular, the second test of the sixth student may be referenced by writing

Student.Test[6, 2]   or simply   Test[6,2]

The order of the subscripts corresponds to the order of the qualifying identifiers. For example,

Test[3, 1]

does not refer to the third test of the first student, but to the first test of the third student.

Remark: Texts sometimes use functional notation instead of the dot notation to denote qualifying identifiers. For example, one writes

```
Age(Father(Newborn))    instead  of    Newborn.Father.Age
```

and

```
First(Name(Student[8]))    instead  of    Student.Name.First[8]
```

Observe that the order of the qualifying identifiers in the functional notation is the reverse of the order in the dot notation.

4.15 REPRESENTATION OF RECORDS IN MEMORY; PARALLEL ARRAYS

Since records may contain nonhomogeneous data, the elements of a record cannot be stored in an array. C language allows the storage of such nonhomogeneous data records with the help of structures.

Example 4.23

Consider the record structure Newborn in Example 4.20. One can store such a record in C by the following declaration, which defines a data aggregate called a *structure*:

```
struct NEWBORN
{
        char NAME[20];
        char SEX[1];
        struct BIRTHDAY
        {
                int MONTH;
                int DAY;
                int YEAR;
        }B;
        struct FATHER
        {
                char NAME[20];
                int AGE;
```

Example 4.26

(a) Suppose

$$A = \begin{pmatrix} 1 & -2 & 3 \\ 0 & 4 & 5 \end{pmatrix} \quad \text{and} \quad B = \begin{pmatrix} 3 & 0 & -6 \\ 2 & -3 & 1 \end{pmatrix}$$

Then:

$$A + B = \begin{pmatrix} 1+3 & -2+0 & 3+(-6) \\ 0+2 & 4+(-3) & 5+1 \end{pmatrix} = \begin{pmatrix} 4 & -2 & -3 \\ 2 & 1 & 6 \end{pmatrix}$$

$$3A = \begin{pmatrix} 3\cdot 1 & 3\cdot(-2) & 3\cdot 3 \\ 3\cdot 0 & 3\cdot 4 & 3\cdot 5 \end{pmatrix} = \begin{pmatrix} 3 & -6 & 9 \\ 0 & 12 & 15 \end{pmatrix}$$

(b) Suppose $U = (1, -3, 4, 5)$, $V = (2, -3, -6, 0)$ and $W = (3, -5, 2, -1)$. Then:

$$U \cdot V = 1 \cdot 2 + (-3) \cdot (-3) + 4 \cdot (-6) + 5 \cdot 0 = 2 + 9 - 24 + 0 = -13$$
$$U \cdot W = 1.3 + (-3) \cdot (-5) + 4 \cdot 2 + 5 \cdot (-1) = 3 + 15 + 8 - 5 = 21$$

(c) Suppose

$$A = \begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix} \quad \text{and} \quad B = \begin{pmatrix} 2 & 0 & -4 \\ 3 & 2 & 6 \end{pmatrix}$$

The product matrix AB is defined and is a 2×3 matrix. The elements in the first row of AB are obtained, respectively, by multiplying the first row of A by each of the columns of B:

$$\begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix}\begin{pmatrix} 2 & 0 & -4 \\ 3 & 2 & 6 \end{pmatrix} = \begin{pmatrix} 1\cdot 2+3\cdot 3 & 1\cdot 0+3\cdot 2 & 1\cdot(-4)+3\cdot 6 \\ & & \end{pmatrix} = \begin{pmatrix} 11 & 6 & 14 \\ & & \end{pmatrix}$$

Similarly, the elements in the second row of AB are obtained, respectively, by multiplying the second row of A by each of the columns of B:

$$\begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix}\begin{pmatrix} 2 & 0 & -4 \\ 3 & 2 & 6 \end{pmatrix} = \begin{pmatrix} 11 & 6 & 14 \\ 2\cdot 2+4\cdot 3 & 2\cdot 0+4\cdot 2 & 2\cdot(-4)+4\cdot 6 \end{pmatrix} = \begin{pmatrix} 11 & 6 & 14 \\ 16 & 8 & 16 \end{pmatrix}$$

That is, $$AB = \begin{pmatrix} 11 & 6 & 14 \\ 16 & 8 & 16 \end{pmatrix}$$

The following algorithm finds the product AB of matrices A and B, which are stored as two-dimensional arrays. (Algorithms for matrix addition and matrix scalar multiplication, which are very similar to algorithms for vector addition and scalar multiplication, are left as exercises for the reader.)

Algorithm 4.8: (Matrix Multiplication) MATMUL(A, B, C, M, P, N)
Let A be an M × P matrix array, and let B be a P × N matrix array. This algorithm stores the product of A and B in an M × N matrix array C.

1. Repeat Steps 2 to 4 for I = 1 to M:
2. Repeat Steps 3 and 4 for J = 1 to N:

3. $a(f - h)$
4. $d(g - e)$
5. $(a + b)h$
6. $(c - a)(e + f)$
7. $(b - d)(g + h)$

Certain versions of the programming language BASIC have matrix operations built into the language. Specifically, the following are valid BASIC statements where A and B are two-dimensional arrays that have appropriate dimensions and K is a scalar:

```
MAT C = A +B
MAT D = (K)*A
MAT E = A*B
```

Each statement begins with the keyword MAT, which indicates that matrix operations will be performed. Thus C will be the matrix sum of A and B, D will be the scalar product of the matrix A by the scalar K, and E will be the matrix product of A and B.

4.17 SPARSE MATRICES

Matrices with a relatively high proportion of zero entries are called *sparse matrices.* Two general types of n-square sparse matrices, which occur in various applications, are pictured in Fig. 4.26. (It is sometimes customary to omit blocks of zeros in a matrix as in Fig. 4.26.) The first matrix, where all entries above the main diagonal are zero or, equivalently, where nonzero entries can only occur on or below the main diagonal, is called a *(lower) triangular matrix.* The second matrix, where nonzero entries can only occur on the diagonal or on elements immediately above or below the diagonal, is called a *tridiagonal matrix.*

$$\begin{pmatrix} 4 & & & & \\ 3 & -5 & & & \\ 1 & 0 & 6 & & \\ -7 & 8 & -1 & 3 & \\ 5 & -2 & 0 & 2 & -8 \end{pmatrix}$$

(a) Triangular matrix

$$\begin{pmatrix} 5 & -3 & & & & & \\ 1 & 4 & 3 & & & & \\ & 9 & -3 & 6 & & & \\ & & 2 & 4 & -7 & & \\ & & & 3 & -1 & 0 & \\ & & & & 6 & -5 & 8 \\ & & & & & 3 & -1 \end{pmatrix}$$

(b) Tridiagonal matrix

Fig. 4.26

The natural method of representing matrices in memory as two-dimensional arrays may not be suitable for sparse matrices. That is, one may save space by storing only those entries which may be nonzero. This is illustrated for triangular matrices in the following example. Other cases will be discussed in the solved problems.

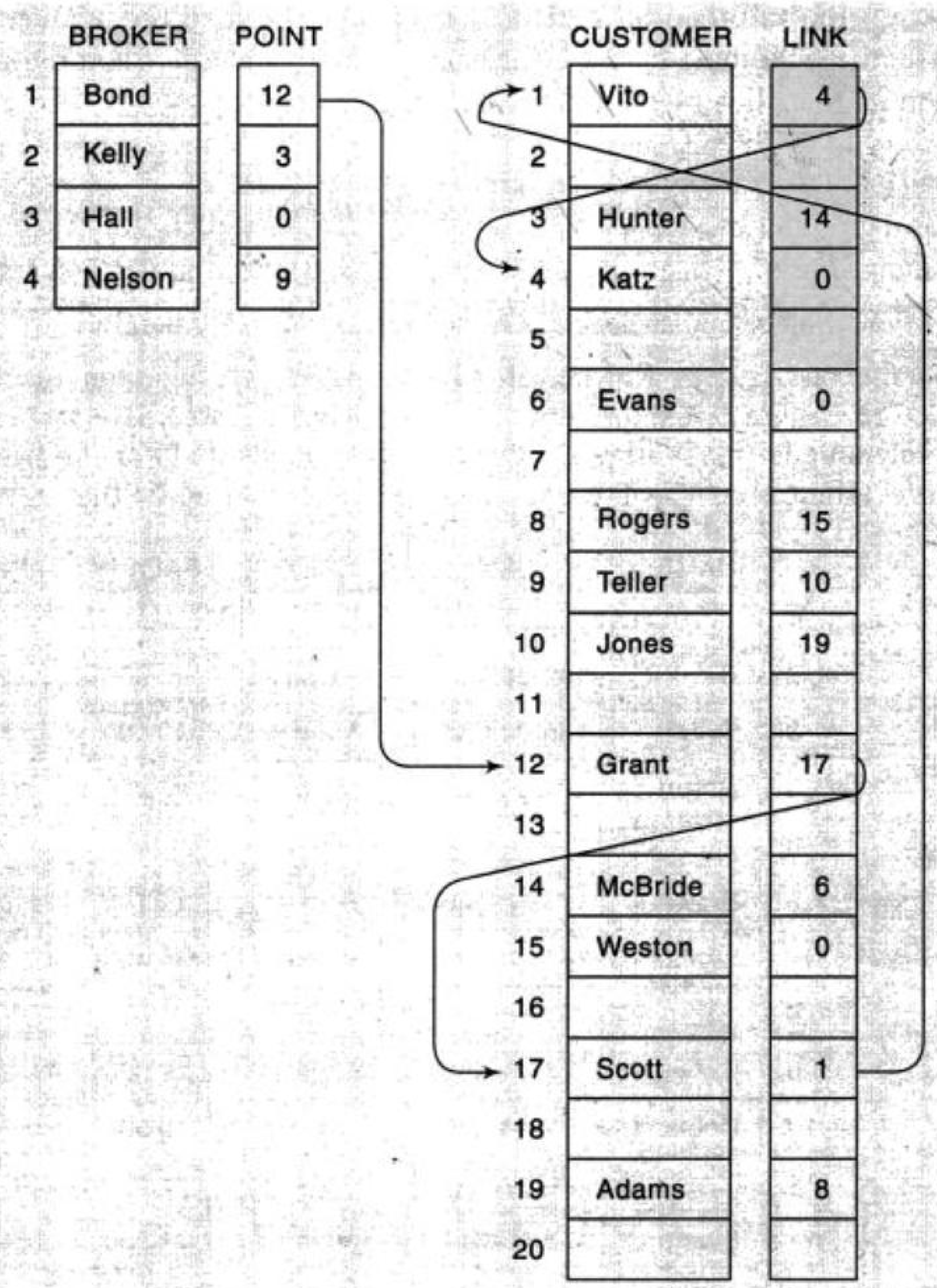


Fig. 5.7

Accordingly, Bond's list of customers, as indicated by the arrows, consists of

Grant, Scott, Vito, Katz

Similarly, Kelly's list consists of

Hunter, McBride, Evans

and Nelson's list consists of

Teller, Jones, Adams, Rogers, Weston

Hall's list is the null list, since the null pointer 0 appears in POINT[3].

```
  PTR=LINK[PTR];
 }

 PTR=START;
 printf("\n\nLIST After Traversal: \n");
 while(PTR!=-1)
 {
  printf("%d\t",LIST[PTR]);
  PTR=LINK[PTR];
 }
 getch();
}

void PROCESS(int P1)
{
  LIST[P1]=LIST[P1]*10;
}
```

```
Output:
Initial LIST:
2       55      22      19      5       50      87      79      33      99
29      8

LIST After Traversal:
20      550     220     190     50      500     870     790     330     990
290     80
```

Remark: In the above program, the PROCESS operation multiplies each element in the list with 10.

Example 5.7

The following procedure finds the number NUM of elements in a linked list.

Procedure: COUNT(INFO, LINK, START, NUM)

1. Set NUM : = 0. [Initializes counter.]
2. Set PTR : = START. [Initializes pointer.]
3. Repeat Steps 4 and 5 while PTR ≠ NULL.
4. Set NUM : = NUM + 1. [Increases NUM by 1.]
5. Set PTR : = LINK[PTR]. [Updates pointer.]
   [End of Step 3 loop.]
6. Return.

Observe that the procedure traverses the linked list in order to count the number of elements; hence the procedure is very similar to the above traversing algorithm, Algorithm 5.1. Here, however, we require an initialization step for the variable NUM before traversing the list. In other words, the procedure could have been written as follows:

```
  int L;
  while(P!=-1)
  {
  if(I>INFO[P])
   P=LINK[P];
  else if (I==INFO[P])
  {
   L=P;
   return(L);
  }
  else
  {
   L=-1;
   return(L);
  }
  }
L=-1;
return(L);
}

Output:
LIST:
2        5        8        19        22        29        33        50
55        79        87        99

Enter the ITEM to be searched: 99

ITEM 99 present at INDEX location 14 in the LIST
```

5.6 MEMORY ALLOCATION; GARBAGE COLLECTION

The maintenance of linked lists in memory assumes the possibility of inserting new nodes into the lists and hence requires some mechanism which provides unused memory space for the new nodes. Analogously, some mechanism is required whereby the memory space of deleted nodes becomes available for future use. These matters are discussed in this section, while the general discussion of the inserting and deleting of nodes is postponed until later sections.

Together with the linked lists in memory, a special list is maintained which consists of unused memory cells. This list, which has its own pointer, is called the *list of available space* or the *free-storage list* or the *free pool.*

Suppose our linked lists are implemented by parallel arrays as described in the preceding sections, and suppose insertions and deletions are to be performed on our linked lists. Then the unused memory cells in the arrays will also be linked together to form a linked list using AVAIL as its list pointer variable. (Hence this free-storage list will also be called the AVAIL list.) Such a data structure will frequently be denoted by writing

LIST(INFO, LINK, START, AVAIL)

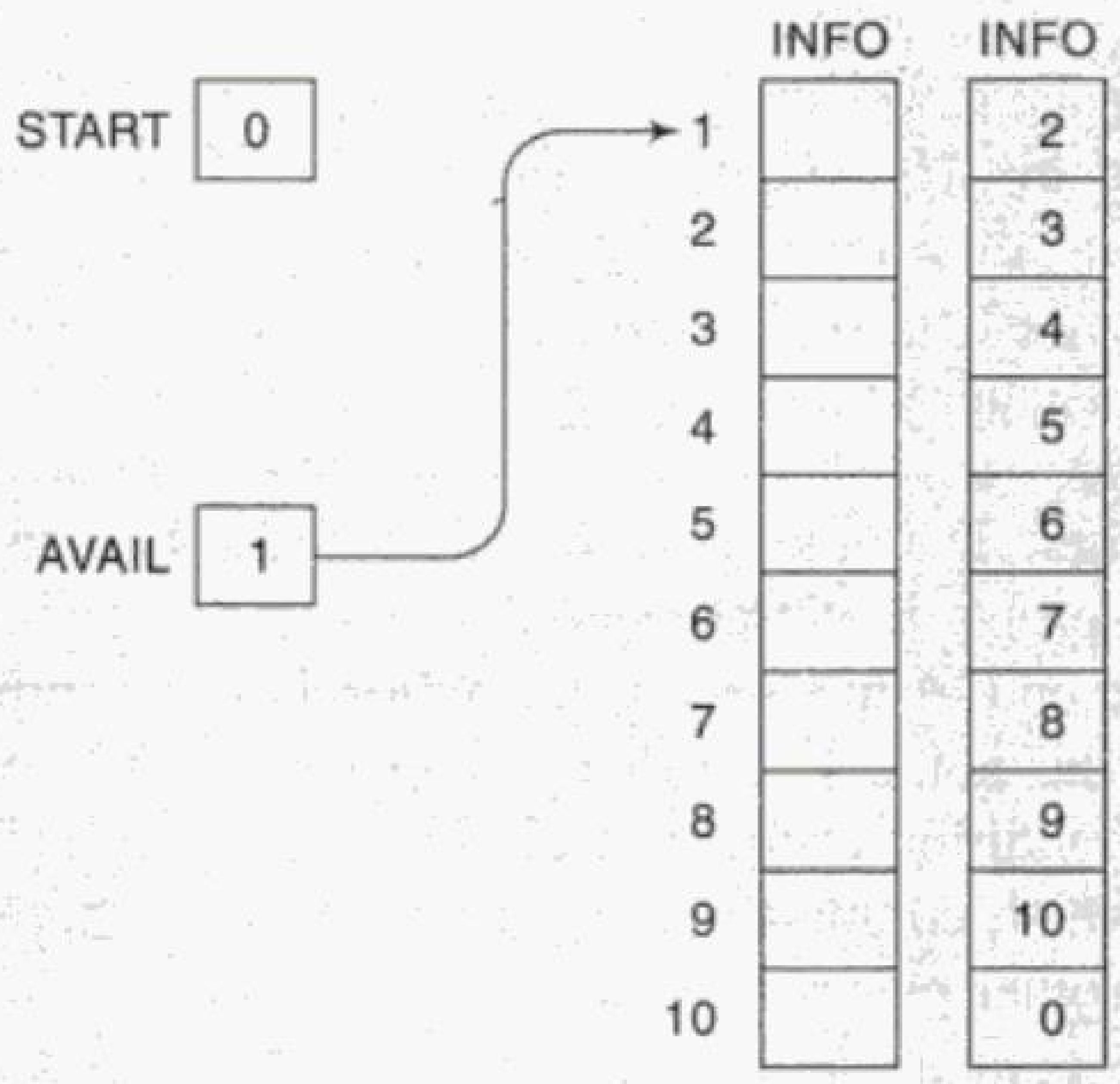


Fig. 5.14

Garbage Collection

Suppose some memory space becomes reusable because a node is deleted from a list or an entire list is deleted from a program. Clearly, we want the space to be available for future use. One way to bring this about is to immediately reinsert the space into the free-storage list. This is what we will do when we implement linked lists by means of linear arrays. However, this method may be too time-consuming for the operating system of a computer, which may choose an alternative method, as follows.

The operating system of a computer may periodically collect all the deleted space onto the free-storage list. Any technique which does this collection is called *garbage collection*. Garbage collection usually takes place in two steps. First the computer runs through all lists, tagging those cells which are currently in use, and then the computer runs through the memory, collecting all untagged space onto the free-storage list. The garbage collection may take place when there is only some minimum amount of space or no space at all left in the free-storage list, or when the CPU is idle and has time to do the collection. Generally speaking, the garbage collection is invisible to the programmer. Any further discussion about this topic of garbage collection lies beyond the scope of this text.

Overflow and Underflow

Sometimes new data are to be inserted into a data structure but there is no available space, i.e., the free-storage list is empty. This situation is usually called *overflow*. The programmer may handle overflow by printing the message OVERFLOW. In such a case, the programmer may then modify the program by adding space to the underlying arrays. Observe that overflow will occur with our linked lists when AVAIL = NULL and there is an insertion.

Analogously, the term *underflow* refers to the situation where one wants to delete data from a data structure that is empty. The programmer may handle underflow by printing the message UNDERFLOW. Observe that underflow will occur with our linked lists when START = NULL and there is a deletion.

location of the new node, this step can be implemented by the pair of assignments (in this order)

NEW := AVAIL,   AVAIL := LINK[AVAIL]

(c) Copying new information into the new node. In other words,

INFO[NEW] := ITEM

The schematic diagram of the latter two steps is pictured in Fig. 5.18.

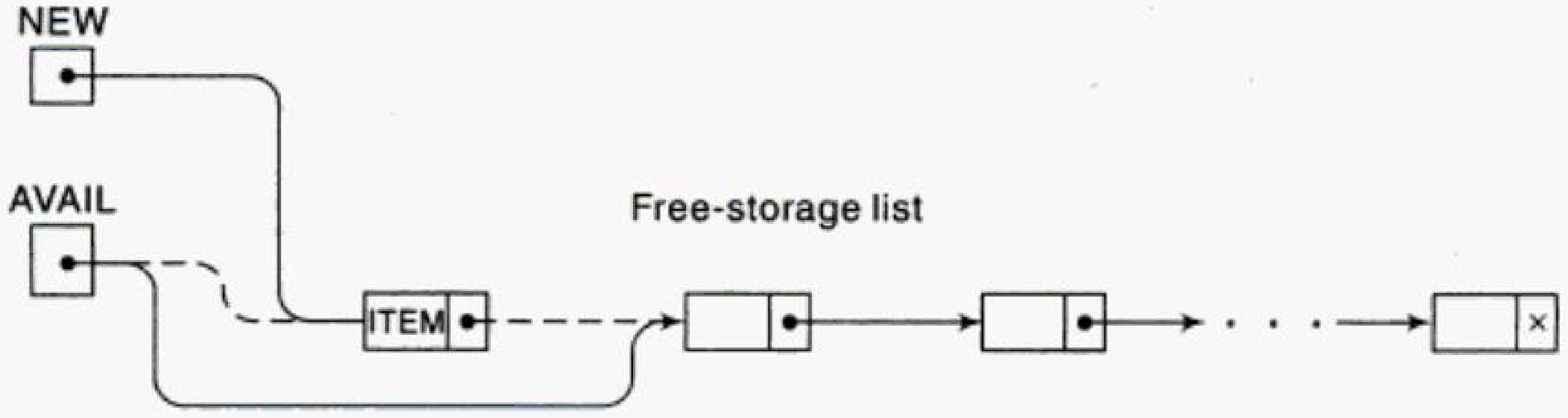


Fig. 5.18

Inserting at the Beginning of a List

Suppose our linked list is not necessarily sorted and there is no reason to insert a new node in any special place in the list. Then the easiest place to insert the node is at the beginning of the list. An algorithm that does so follows.

Algorithm 5.4: INSFIRST(INFO, LINK, START, AVAIL, ITEM)
This algorithm inserts ITEM as the first node in the list.
1. [OVERFLOW?] If AVAIL = NULL, then: Write: OVERFLOW, and Exit.
2. [Remove first node from AVAIL list.]
   Set NEW := AVAIL and AVAIL := LINK[AVAIL].
3. Set INFO[NEW] := ITEM. [Copies new data into new node]
4. Set LINK[NEW] := START. [New node now points to original first node.]
5. Set START := NEW. [Changes START so it points to the new node.]
6. Exit.

Steps 1 to 3 have already been discussed, and the schematic diagram of Steps 2 and 3 appears in Fig. 5.18. The schematic diagram of Steps 4 and 5 appears in Fig. 5.19.

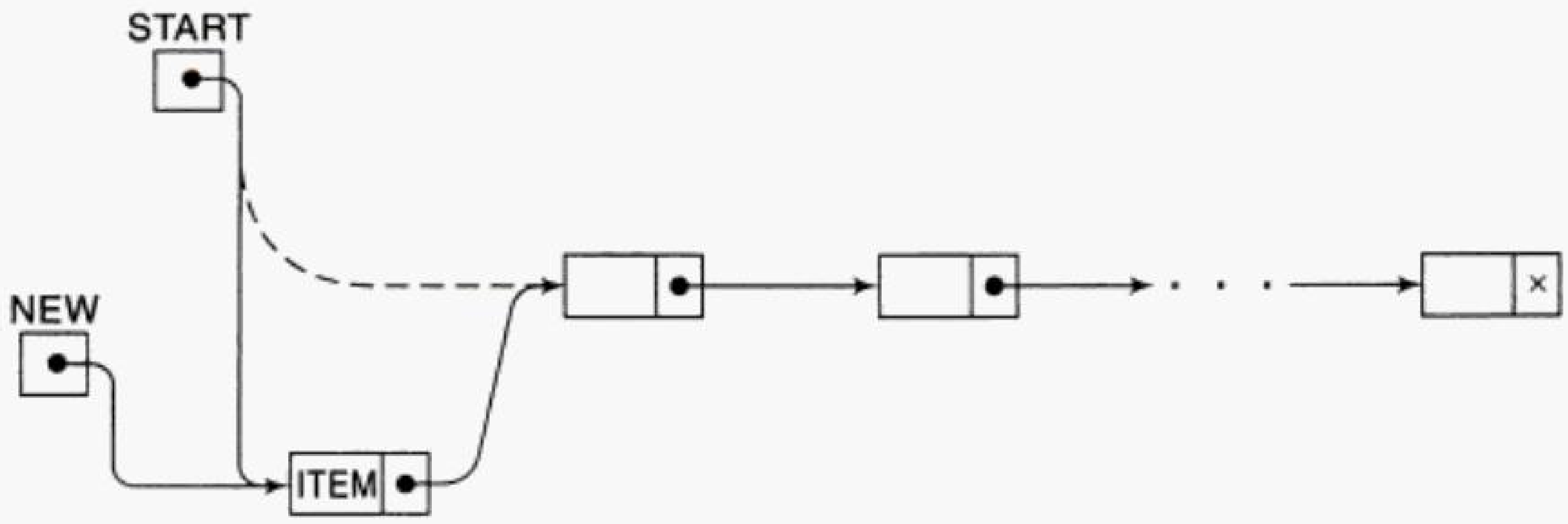


Fig. 5.19 Insertion at the Beginning of a List

3. SAVE = 5 and PTR = LINK[5] = 3.
4. Steps 5 and 6 are repeated as follows:
   (a) BED[3] = Dean < Jones, so SAVE = 3 and PTR = LINK[3] = 11.
   (b) BED[11] = Fields < Jones, so SAVE = 11 and PTR = LINK[11] = 8.
   (c) BED[8] = Green < Jones, so SAVE = 8 and PTR = LINK[8] = 1.
   (d) Since BED[1] = Kirk > Jones, we have:
   LOC = SAVE = 8 and Return.

(b) INSLOC(BED, LINK, START, AVAIL, LOC, ITEM) [Here LOC = 8.]

1. Since AVAIL ≠ NULL, control is transferred to Step 2.
2. NEW = 10 and AVAIL = LINK[10] = 2.
3. BED[10] = Jones.
4. Since LOC ≠ NULL we have:
   LINK[10] = LINK[8] = 1 and LINK[8] = NEW = 10.
5. Exit.

Figure 5.22 shows the data structure after Jones is added to the patient list. We emphasize that only three pointers have been changed, AVAIL, LINK[10] and LINK[8].

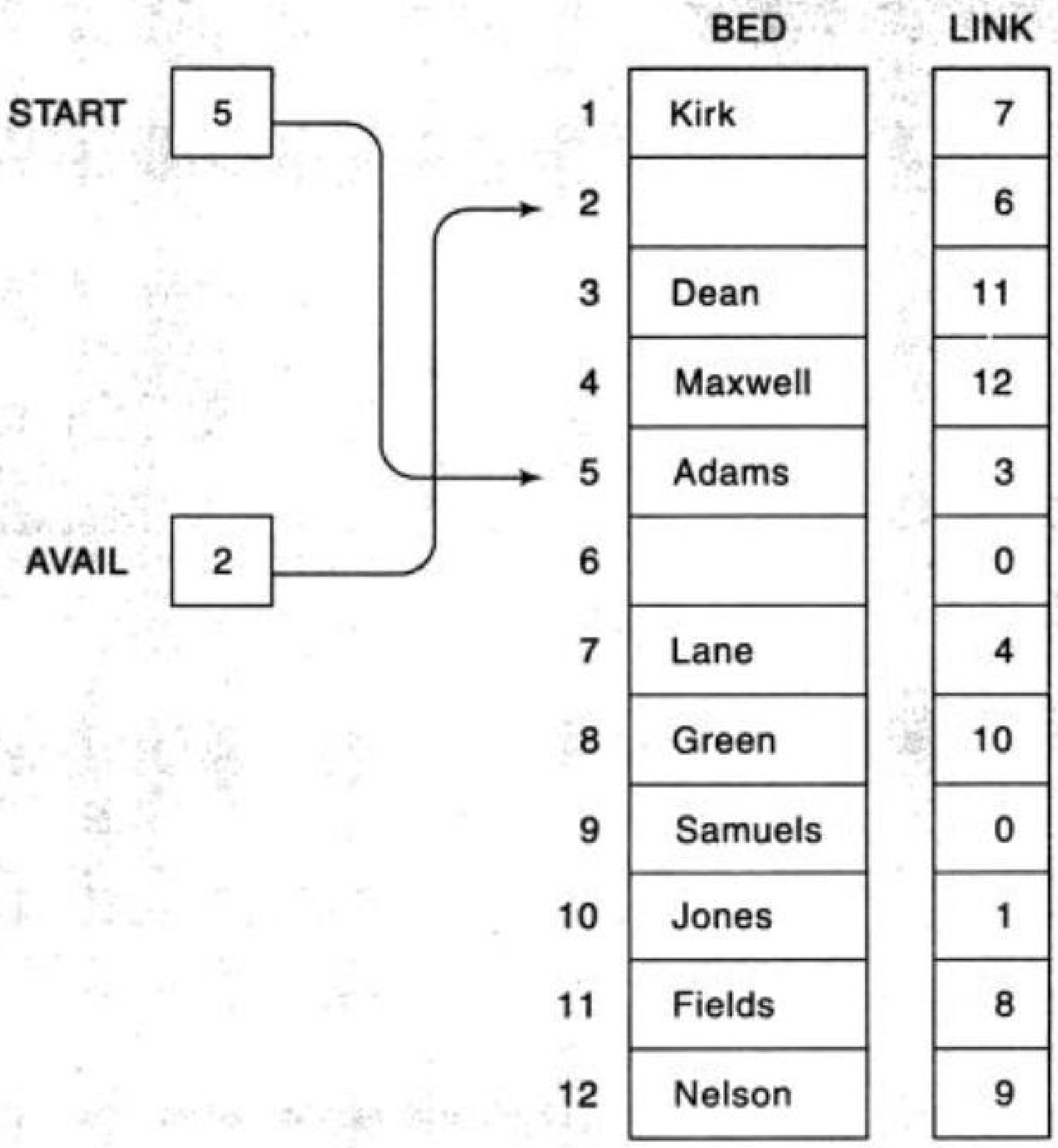


Fig. 5.22

The following is the C implementation of Algorithm 5.7 to insert an element into a sorted list:

Figure 5.23 does not take into account the fact that, when a node N is deleted from our list, we will immediately return its memory space to the AVAIL list. Specifically, for easier processing, it will be returned to the beginning of the AVAIL list. Thus a more exact schematic diagram of such a deletion is the one in Fig. 5.24. Observe that three pointer fields are changed as follows:

1. The nextpointer field of node A now points to node B, where node N previously pointed.
2. The nextpointer field of N now points to the original first node in the free pool, where AVAIL previously pointed.
3. AVAIL now points to the deleted node N.

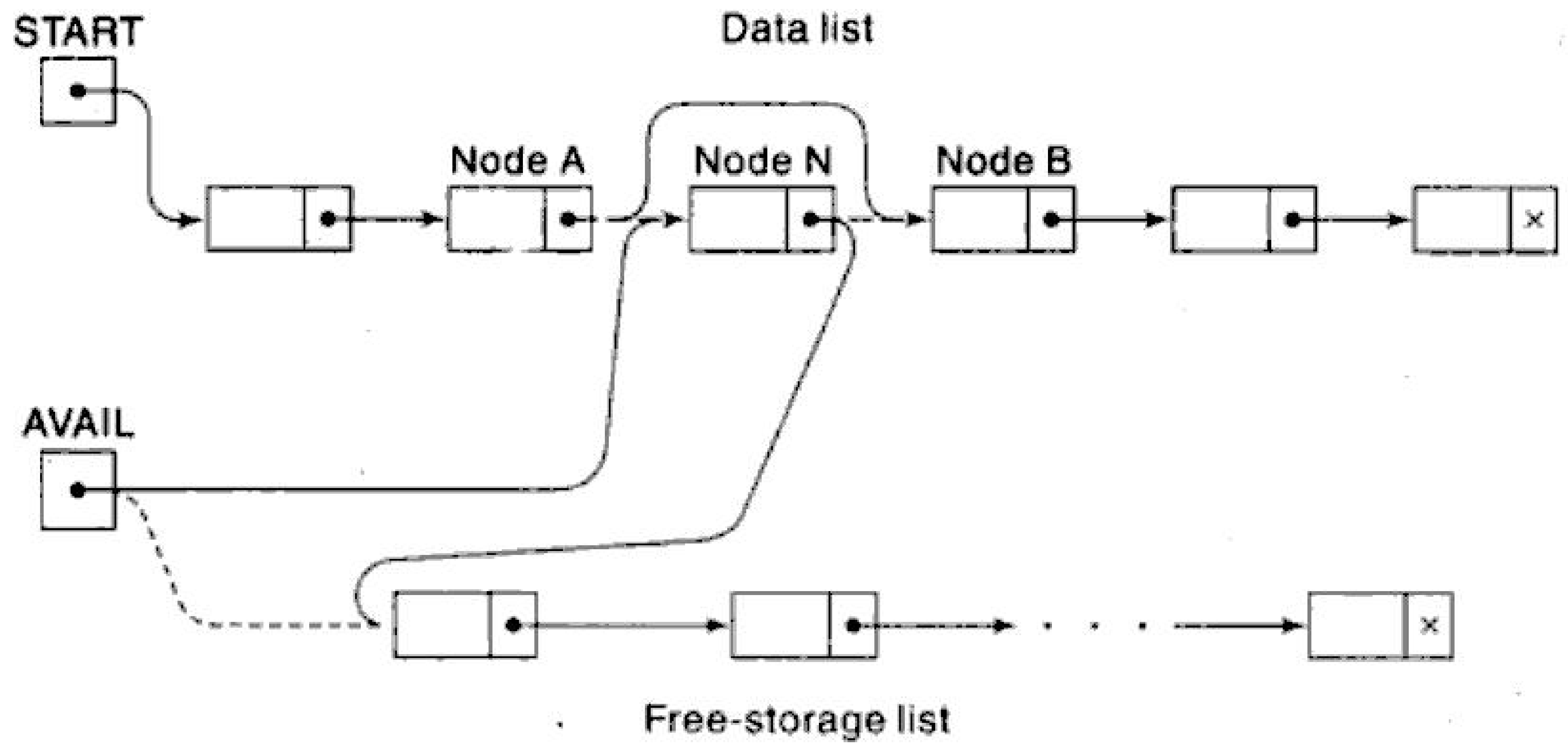


Fig. 5.24

There are also two special cases. If the deleted node N is the first node in the list, then START will point to node B; and if the deleted node N is the last node in the list, then node A will contain the NULL pointer.

Example 5.16

(a) Consider Fig. 5.22, the list of patients in the hospital ward. Suppose Green is discharged, so that BED[8] is now empty. Then, in order to maintain the linked list, the following three changes in the pointer fields must be executed:

LINK[11] = 10 LINK[8] = 2 AVAIL = 8

By the first change, Fields, who originally preceded Green, now points to Jones, who originally followed Green. The second and third changes add the new empty bed to the AVAIL list. We emphasize that, before making the deletion, we had to find the node BED[11], which originally pointed to the deleted node BED[8].

(b) Consider Fig. 5.13, the list of brokers and their customers. Suppose Teller, the first customer of Nelson, is deleted from the list of customers. Then, in order to maintain the linked lists, the following three changes in the pointer fields must be executed:

POINT[4] = 10 LINK[9] = 11 AVAIL = 9

By the first change, Nelson now points to his original second customer, Jones. The second and third changes add the new empty node to the AVAIL list.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

3. If INFO[PTR] = ITEM, then:
   Set LOC := PTR and LOCP := SAVE.
   Else:
   Set LOC := NULL and LOCP := SAVE.
   [End of If structure.]
4. Exit.

Observe the simplicity of this procedure compared with Procedure 5.9. Here we did not have to consider the special case when ITEM appears in the first node, and here we can perform at the same time the two tests which control the loop.

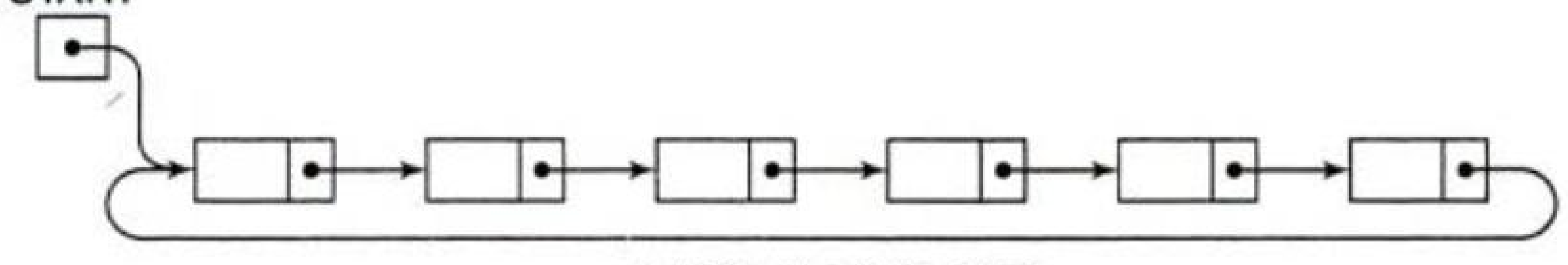
(c) Algorithm 5.10 deletes the first node N which contains ITEM when LIST is an ordinary linked list. The following is such an algorithm when LIST is a circular header list.

Algorithm 5.14: DELLOCHL(INFO, LINK, START, AVAIL, ITEM)
1. [Use Procedure 5.15 to find the location of N and its preceding node.]
   Call FINDBHL(INFO, LINK, START, ITEM, LOC, LOCP).
2. If LOC = NULL, then: Write: ITEM not in list, and Exit.
3. Set LINK[LOCP] := LINK[LOC]. [Deletes node.]
4. [Return deleted node to the AVAIL list.]
   Set LINK[LOC]:= AVAIL and AVAIL:= LOC.
5. Exit.

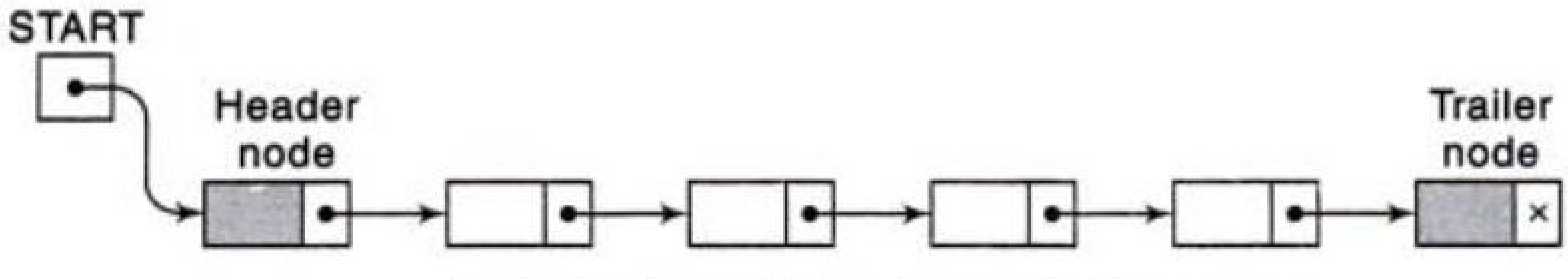
Again we did not have to consider the special case when ITEM appears in the first node, as we did in Algorithm 5.10.

Remark: There are two other variations of linked lists which sometimes appear in the literature:
1. A linked list whose last node points back to the first node instead of containing the null pointer, called a *circular list*
2. A linked list which contains both a special header node at the beginning of the list and a special trailer node at the end of the list

Figure 5.32 contains schematic diagrams of these lists.



(a) Circular linked list



(b) Linked list with header and trailer nodes

Fig. 5.32

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
Enter the ITEM to be deleted: 79
Modified LIST:

55        22        19        5         50        87        33        99
29        8
```

Polynomials

Header linked lists are frequently used for maintaining polynomials in memory. The header node plays an important part in this representation, since it is needed to represent the zero polynomial. This representation of polynomials will be presented in the context of a specific example.

Example 5.20

Let $p(x)$ denote the following polynomial in one variable (containing four nonzero terms):

$$p(x) = 2x^8 - 5x^7 - 3x^2 + 4$$

Then $p(x)$ may be represented by the header list pictured in Fig. 5.35(a), where each node corresponds to a nonzero term of $p(x)$. Specifically, the information part of the node is divided into two fields representing, respectively, the coefficient and the exponent of the corresponding term, and the nodes are linked according to decreasing degree.

Observe that the list pointer variable POLY points to the header node, whose exponent field is assigned a negative number, in this case −1. Here the array representation of the list will require three linear arrays, which we will call COEF, EXP and LINK. One such representation appears in Fig. 5.35(b).

Representation of Polynomials Using Linked Lists

We can represent a polynomial using array or a linked list. This is done by simply storing the coefficient and exponent of all the terms. The linked list representation is easy for operations like polynomial addition or subtraction, and polynomial multiplication. We should also keep in mind that in a polynomial, all the terms might not be present, especially for polynomials with a higher order. Look at the following example.

$$8x^{13} + 5x^6 + 2x^8 + x^5 + x^{10} + 17x$$

This 13[th] order polynomial does not have all the 14 terms (including the constant term). Thus, it is very easy to represent the polynomial with the help of a linked list structure. Here every node can retain information pertaining to a single term of the polynomial. All the nodes store three things:

- variable x
- exponent
- coefficient for each term

Whatever be the equation, it does not matter if the polynomial is in x or y. This is a very important information that needs to be kept in mind when performing operations on polynomials. Thus it is better if we define a node structure which holds 2 integers— exp and coff.

Let us compare this representation with an array structure storing the same polynomials. In

```
int LOCP,LOC;
int START, AVAIL;
int SRCHHL(int);
void FINDBHL(int);
void DELLOCHL(int);

void main()
{
int PTR,ITEM;
clrscr();

INFO[0]=22;INFO[2]=5;INFO[3]=19;INFO[5]=87;INFO[7]=29;INFO[8]=79;
INFO[9]=33;INFO[13]=50;INFO[14]=8;INFO[16]=55;INFO[18]=99;

LINK[0]=3;LINK[1]=17;LINK[2]=13;LINK[3]=2;LINK[4]=12;LINK[5]=8;
LINK[6]=-1;LINK[7]=14;LINK[8]=9;LINK[9]=18;LINK[10]=6;LINK[11]=16;
LINK[12]=10;LINK[13]=5;LINK[14]=11;LINK[15]=1;LINK[17]=19;LINK[18]=7;
LINK[19]=4;

START=11;
AVAIL=15;

printf("LIST: \n\n");
PTR=START;
printf("%d\t",INFO[PTR]);
PTR=LINK[PTR];
while(PTR!=START)
{
 printf("%d\t",INFO[PTR]);
 PTR=LINK[PTR];
}

printf("\n\nTraversing the LIST and applying PROCESS to each
node..\n\n");
PTR=START;
printf("%d\t",INFO[PTR]*10);
PTR=LINK[PTR];
while(PTR!=START)
{
 printf("%d\t",INFO[PTR]*10);
 PTR=LINK[PTR];
}

printf("\n\nEnter the ITEM to be searched: ");
scanf("%d",&ITEM);
```

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Generally, on giving the address of a size $2k$ node which needs to be returned, there is a change in its $(k + 1)^{th}$ bit from 1 to 0 or vice versa and the output is its buddy's address. We can come to know if a node is in use by making use of the tag field.

Example 5.26

Suppose m is 6 and requests are made for nodes of size 8, 16, 4, and 6. We will see how the buddy system responds.

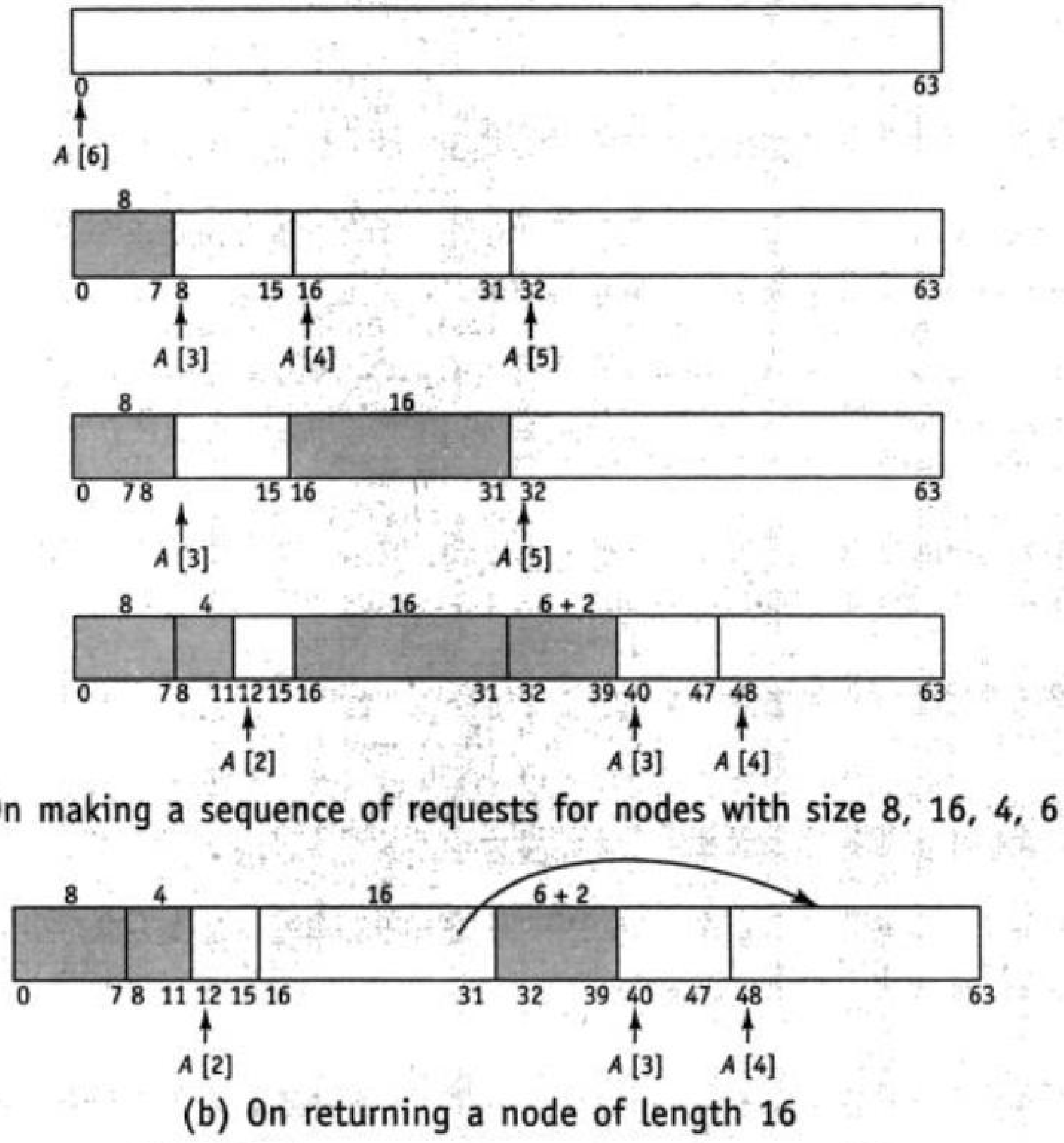


Fig. 5.43 Sequence of Memory Configurations

Figure 5.43(a) shows the sequence of memory configurations. If the node of length 16 is returned, the configuration becomes as shown in Fig. 5.43(b).

A[4] has two nodes and is the head of the list, the nodes are of length 16 each. The node which is released had a length 16 which starts at the address 16 having a node f length 16 which starts at the address 0 as its buddy. It cannot merge together as it is not free. It also can't merge with the node having a length 4 beginning with the address 12 because it is not its buddy.

Buddies are usually the successor nodes of the same parent. Hence we can come to a conclusion that [12, 15] and [16, 31] cannot be considered as buddies even though they have an adjacent memory space.And hence we can also say that they cannot be merged either.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

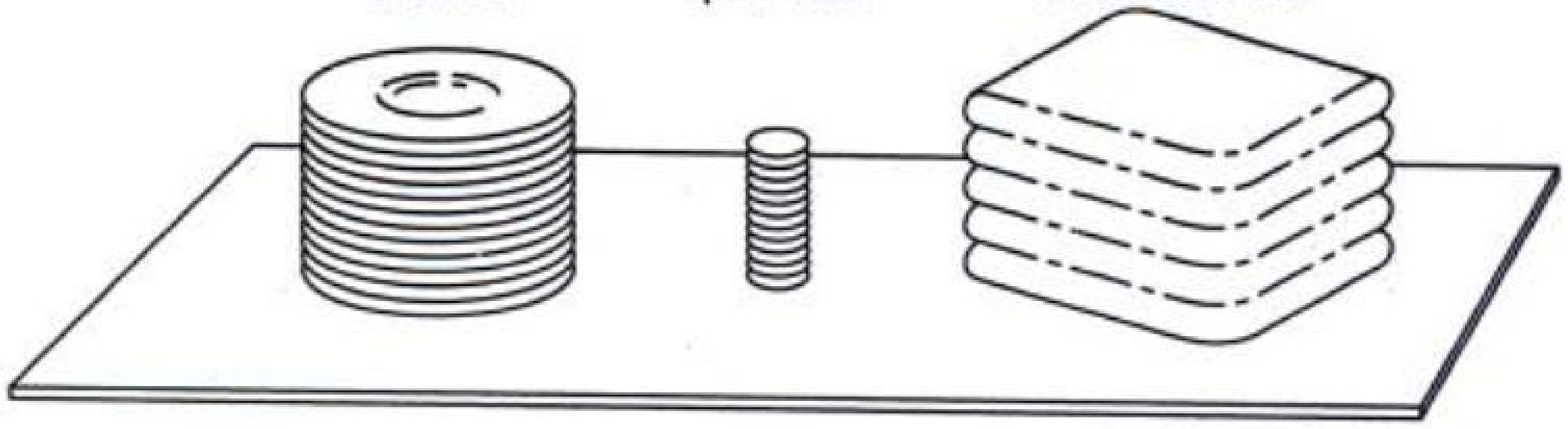


Fig. 6.1

Fig. 6.2 Queue Waiting for a Bus

6.2 STACKS

A *stack* is a list of elements in which an element may be inserted or deleted only at one end, called the *top* of the stack. This means, in particular, that elements are removed from a stack in the reverse order of that in which they were inserted into the stack.

Special terminology is used for two basic operations associated with stacks:

(a) "Push" is the term used to insert an element into a stack.
(b) "Pop" is the term used to delete an element from a stack.

We emphasize that these terms are used only with stacks, not with other data structures.

Example 6.1

Suppose the following 6 elements are pushed, in order, onto an empty stack:

AAA, BBB, CCC, DDD, EEE, FFF

Figure 6.3 shows three ways of picturing such a stack. For notational convenience, we will frequently designate the stack by writing:

STACK: AAA, BBB, CCC, DDD, EEE, FFF

The implication is that the right-most element is the top element. We emphasize that, regardless of the way a stack is described, its underlying property is that insertions and deletions can occur only at the top of the stack. This means EEE cannot be deleted before FFF is deleted, DDD cannot be deleted before EEE and FFF are deleted, and so on. Consequently, the elements may be popped from the stack only in the reverse order of that in which they were pushed onto the stack.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
#define Max 15
#include <stdlib.h>
void push(int newstack[], int *top, int val)
{
 if(*top < Max )
{
 *top = *top + 1;
 newstack[*top] = val;
}
else
{
 printf("No value can be pushed as stack is full\n");
 exit(0);
}
}
void pop(int newstack[], int *top, int * val)
{
 if(*top >= 0 )
{
 *val = newstack[*top];
 *top = *top - 1;
}
else
{
 printf("No value can be popped as a stack is empty\n");
 exit(0);
}
}
void main()
{
 int newstack[MAX];
 int top = -1;
 int n,val;
 do
{
 do
{
 printf("The element to be pushed is\n");
 scanf("%d",&val);
 push(newstack,&top,val);
 printf("To continue enter 1\n");
 scanf("%d",&n);
} while(n == 1);
 printf("To pop an element enter 1\n");
 scanf("%d",&n);
 while( n == 1)
{
```

with the need to maintain the MAXSTK variable and consequently on the checking of OVERFLOW of the linked stack during a push operation.

Procedure 6.3: PUSH_LINKSTACK(INFO, LINK, TOP, AVAIL, ITEM)
This procedure pushes an ITEM into a linked stack

1. [Available space?] If AVAIL = NULL, then Write OVERFLOW and Exit
2. [Remove first node from AVAIL list]
   Set NEW := AVAIL and AVAIL := LINK[AVAIL].
3. Set INFO[NEW] := ITEM [Copies ITEM into new node]
4. Set LINK[NEW] := TOP [New node points to the original top node in the stack]
5. Set TOP = NEW [Reset TOP to point to the new node at the top of the stack]
6. Exit.

Procedure 6.4: POP_LINKSTACK(INFO, LINK, TOP, AVAIL, ITEM)
This procedure deletes the top element of a linked stack and assigns it to the variable ITEM

1. [Stack has an item to be removed?]
   IF TOP = NULL then Write: UNDERFLOW and Exit.
2. Set ITEM := INFO[TOP] [Copies the top element of stack into ITEM]
3. Set TEMP := TOP and TOP = LINK[TOP]
   [Remember the old value of the TOP pointer in TEMP and reset TOP to point to the next element in the stack]
4. [Return deleted node to the AVAIL list]
   Set LINK[TEMP] = AVAIL and AVAIL = TEMP.
5. Exit.

However, the condition TOP = NULL may be retained in the pop procedure to prevent deletion from an empty linked stack and the condition AVAIL = NULL to check for available space in the free-storage list.

Example 6.4

Consider the linked stack shown in Fig. 6.7, the snapshots of the stack structure on execution of the following operations are shown in Fig. 6.10:
(i) Push BBB (ii) Pop (iii) Pop (iv) Push MMM
Original linked stack:

to be a success which is shown by returning true. If the heap is full, we report that the push failed by returning false. It is then the calling function's responsibility to detect and respond to an overflow.

Pop Stack

The data at the top of the stack is returned by the Pop stack operation. The node is then deleted and recycled after adjusting the count and subtracting it by 1, the function returns to the caller. Note the way underflow is reported. In statement 'dataOutPtr = NULL', we set the data pointer to NULL. If the stack is empty, when we return the data pointer in statement 'return dataOutPtr' as NULL. The code is shown below:

Program 6.5

Pop Stack: This function is used to pop an item on on top of stack.STACK is a pointer to a stack. The program is used for returning a pointer to the user's data when it is successful. It returns a NULL when there is an underflow.

```
void* popStack (STACK* newstack)
{
 void* dataOutPntr;
 STACK_NODE* temp;
 if(newstack->count == 0)
      dataOutPntr = NULL;
 else
        {
          temp  = newstack->top;
          dataOutPntr = newstack->top->dataPntr;
          newstack->top = newstack->top->link;
          free(temp);
          (newstack->count)--;
        }
 return dataOutPntr
}
```

In this case, two local pointers are required—one for the data pointer to be returned to the caller, and one that is used to free the deleted node. Its only concern is to ensure that is follows the procedure to keep a track of when the last node is deleted, thus resulting in an empty stack. There is no special logic which is made use of, except that an automatic creation of an empty stack since the last node has a null pointer, which on assigning to the top shows that the stack is empty.

Because a null pointer is false and a pointer with an address is true, we don't need a separate success flag—we just return the pointer with the address allocated by the new function. If memory was allocated successfully, it contains an address, which is true. If the allocation failed, it contains a null pointer, which is false.

Note that it takes two levels of indirection to access the link field of the top node.

On deleting a node from the stack, the pop stack releases its memory.

```
int TOP, AVAIL;
void PUSH(int);
int POP(void);

void main()
{
int i,num,A,B,RESULT;
clrscr();

LINK[0]=3;LINK[1]=17;LINK[2]=13;LINK[3]=2;LINK[4]=12;LINK[5]=8;
LINK[6]=11;LINK[7]=14;LINK[8]=9;LINK[9]=18;LINK[10]=6;LINK[11]=16;
LINK[12]=10;LINK[13]=5;LINK[14]=-1;LINK[15]=1;LINK[17]=19;LINK[18]=7;
LINK[19]=4;

TOP=-1;
AVAIL=15;
printf("Postfix Expression: %s",P);

strcat(P,")");
i=0;
while(P[i]!=')')
{
   if(isdigit(P[i]))
   {
    num=P[i]-'0';
    PUSH(num);
   }
  if(P[i]== '+' || P[i]== '-' ||  P[i]== '*' || P[i]== '/' || P[i]==
'^')
  {
   A=POP();
   B=POP();

   if(P[i]== '+')
    RESULT=B+A;

   if(P[i]== '-')
    RESULT=B-A;

   if(P[i]== '*')
    RESULT=B*A;

   if(P[i]== '/')
    RESULT=B/A;

   if(P[i]== '^')
    RESULT=pow(B,A);
```

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

The following program demonstrates implementation of Hash function in C using subtraction method as the basis:

Program 9.8

```
#include <stdio.h>
#include <conio.h>

void main()
{
   int HASH(int);
   int i;
   clrscr();

   printf("Press any key to generate the Hash Table for Employee
   Code (keys) 2000-2020 ");
   getch();
   printf("\n\n*****HASH TABLE*****\n");
   printf("\nKey\tAddress");
   for(i=2000;i<=2020;i++)
      printf("\n%d\t   %d",i,HASH(i));

   getch();
}

int HASH(int k)
{
   return(k-2000);
}
```

Output:

```
Press any key to generate the Hash Table for Employee Code (keys)
2000-2020

*****HASH TABLE*****

Key       Address
2000        0
2001        1
2002        2
2003        3
2004        4
2005        5
2006        6
2007        7
2008        8
```

key k. The efficiency depends mainly on the load factor λ. Specifically, we are interested in the following two quantities:

$S(\lambda)$ = average number of probes for a successful search

$U(\lambda)$ = average number of probes for an unsuccessful search

These quantities will be discussed for our collision procedures.

Collision Resolution Techniques

There are two broad ways of collision resolution:

(i) Open Addressing, where an array-based implemented.

(ii) Separate Chaining, where an array of linked list implemented.

Open Addressing includes:

- Linear probing (linear search)
- Quadratic probing (nonlinear search), and
- Double hashing (uses two hash functions).

(i) Open Addressing: Linear Probing and Modifications

Suppose that a new record R with key k is to be added to the memory table T, but that the memory location with hash address $H(k) = h$ is already filled. One natural way to resolve the collision is to assign R to the first available location following $T[h]$. (We assume that the table T with m locations is circular, so that $T[1]$ comes after $T[m]$.) Accordingly, with such a collision procedure, we will search for the record R in the table T by linearly searching the locations $T[h]$, $T[h + 1]$, $T[h + 2]$, ... until finding R or meeting an empty location, which indicates an unsuccessful search.

The above collision resolution is called *linear probing*. The average numbers of probes for a successful search and for an unsuccessful search are known to be the following respective quantities:

$$S(\lambda) = \frac{1}{2}\left(1 + \frac{1}{1-\lambda}\right) \quad \text{and} \quad U(\lambda) = \frac{1}{2}\left(1 + \frac{1}{(1-\lambda)^2}\right)$$

(Here $\lambda = n/m$ is the load factor.)

The table below summarizes the characteristics of the various open addressing probing sequences.

Table 9.3 Characteristics of the open addressing probing sequences

probing sequence	*primary clustering*	*capacity limit*	*size restriction*
linear probing	yes	none	none
quadratic probing	no	$\lambda < \frac{1}{2}$	M must be prime
double hashing	no	none	M must be prime

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Index

E

F

G

H

U

V

W

SCHAUM'S ouTlines

The High Performance Study Guide

Schaum's helps you

- Hone problem-solving skills
- Find answers fast
- Cut study time
- Brush up before tests
- Achieve high performance

Related titles in Schaum's Outlines

Computing

Data Structures

Programming with C, 3e

Programming with C++, 2e

Data Structures with Java, 2e

Computer Architecture and Organization, 2e

Computer Graphics, 2e

Operating Systems

Computer Networking

Discrete Mathematics, 3e

Database Management Systems

www.tatamcgrawhill.com

The McGraw·Hill Companies

Mc Graw Hill Higher Education

ISBN 13:978-0-07-070198-4
ISBN 10:0-07-070198-9

9 780070 701984