

Classic Data Structures

D. SAMANTA

*School of Information Technology
Indian Institute of Technology Kharagpur*

Prentice-Hall of India Private Limited

New Delhi - 110001

2006

This One



LGBL-HD9-XG2F

Copyrighted material

Contents

<i>Preface</i>	ix
1 INTRODUCTION AND OVERVIEW	1–9
1.1 Definitions	1
1.2 Concept of Data Structures	4
1.3 Overview of Data Structures	6
1.4 Implementation of Data Structures	8
2 ARRAYS	10–33
2.1 Definition	10
2.2 Terminology	11
2.3 One-dimensional Array	11
2.3.1 Memory Allocation for an Array	12
2.3.2 Operations on Arrays	13
2.3.3 Application of Arrays	19
2.4 Multidimensional Arrays	20
2.4.1 Two-dimensional Arrays	20
2.4.2 Sparse Matrices	22
2.4.3 Three-dimensional and n -dimensional Arrays	28
2.5 Pointer Arrays	30
<i>Problems to Ponder</i>	31
<i>References</i>	33
3 LINKED LISTS	34–97
3.1 Definition	34
3.2 Single Linked List	34
3.2.1 Representation of a Linked List in Memory	35
3.2.2 Operations on a Single Linked List	36
3.3 Circular Linked List	48
3.4 Double Linked Lists	51
3.4.1 Operations on a Double Linked List	51
3.5 Circular Double Linked List	56
3.5.1 Operations on Circular Double Linked List	57
3.6 Application of Linked Lists	59
3.6.1 Sparse Matrix Manipulation	59
3.6.2 Polynomial Representation	63
3.6.3 Dynamic Storage Management	67

3.7	Memory Representation	68
3.7.1	Fixed Block Storage	68
3.7.2	Variable Block Storage	71
3.8	Boundary Tag System	72
3.9	Deallocation Strategy	77
3.10	Buddy System	82
3.10.1	Binary Buddy System	91
3.10.2	Comparison between Fibonacci and Binary Buddy System	92
3.10.3	Comparison of Dynamic Storage Allocation Systems	92
3.11	Compaction	92
	<i>Problems to Ponder</i>	95
	<i>References</i>	97

4 STACKS 98–145

4.1	Introduction	98
4.2	Definition	99
4.3	Representation of Stack	99
4.3.1	Array Representation of Stacks	99
4.3.2	Linked List Representation of Stacks	100
4.4	Operations on Stacks	101
4.5	Applications of Stack	103
4.5.1	Evaluation of Arithmetic Expressions	104
4.5.2	Code Generation for Stack Machines	113
4.5.3	Implementation of Recursion	115
4.5.4	Factorial Calculation	117
4.5.5	Quick Sort	118
4.5.6	Tower of Hanoi Problem	124
4.5.7	Activation Record Management	127
	<i>Problems to Ponder</i>	143
	<i>References</i>	145

5 QUEUES 146–180

5.1	Introduction	146
5.2	Definition	148
5.3	Representation of Queues	148
5.3.1	Representation of Queue using Array	148
5.3.2	Representation of Queue using Linked List	152
5.4	Various Queue Structures	152
5.4.1	Circular Queue	153
5.4.2	Deque	156
5.4.3	Priority Queue	159
5.5	Application of Queues	164
5.5.1	Simulation	164
5.5.2	CPU Scheduling in Multiprogramming Environment	174
5.5.3	Round Robin Algorithm	176
	<i>Problems to Ponder</i>	179
	<i>References</i>	180

6 TABLES	181–202
6.1 Rectangular Tables 182	
6.2 Jagged Tables 182	
6.3 Inverted Tables 185	
6.4 Hash Tables 185	
6.4.1 Hashing Techniques 186	
6.4.2 Collision Resolution Techniques 190	
6.4.3 Closed Hashing 191	
6.4.4 Open Hashing 196	
6.4.5 Comparison of Collision Resolution Techniques 198	
<i>Problems to Ponder</i> 201	
<i>References</i> 201	
7 TREES	203–355
7.1 Basic Terminologies 204	
7.2 Definition and Concepts 207	
7.2.1 Binary Trees 207	
7.2.2 Properties of Binary Tree 209	
7.3 Representation of Binary Tree 212	
7.3.1 Linear Representation of a Binary Tree 213	
7.3.2 Linked Representation of Binary Tree 216	
7.3.3 Physical Implementation of Binary Tree in Memory 218	
7.4 Operations on Binary Tree 220	
7.4.1 Insertion 220	
7.4.2 Deletion 224	
7.4.3 Traversals 227	
7.4.4 Merging of Two Binary Trees 237	
7.5 Types of Binary Trees 240	
7.5.1 Expression Tree 240	
7.5.2 Binary Search Tree 244	
7.5.3 Heap Trees 254	
7.5.4 Threaded Binary Trees 264	
7.5.5 Height Balanced Binary Tree 278	
7.5.6 Weighted Binary Tree 293	
7.5.7 Decision Trees 305	
7.6 Trees and Forests 308	
7.6.1 Representation of Trees 310	
7.7 B Trees 318	
7.7.1 B Tree Indexing 319	
7.7.2 Operations on a B Tree 321	
7.7.3 Lower and Upper Bounds of a B Tree 342	
7.8 B+ Tree Indexing 343	
7.9 Trie Tree Indexing 345	
7.9.1 Trie Structure 345	
7.9.2 Operations on Trie 346	
7.9.3 Applications of Tree Indexing 349	
<i>Problems to Ponder</i> 351	
<i>References</i> 354	

8 GRAPHS

356–433

- 8.1 Introduction 356
- 8.2 Graph Terminologies 358
- 8.3 Representation of Graphs 362
 - 8.3.1 Set Representation 363
 - 8.3.2 Linked Representation 363
 - 8.3.3 Matrix Representation 364
- 8.4 Operations on Graphs 370
 - 8.4.1 Operations on Linked List Representation of Graphs 371
 - 8.4.2 Operations on Matrix Representation of Graphs 383
- 8.5 Application of Graph Structures 391
 - 8.5.1 Shortest Path Problem 393
 - 8.5.2 Topological Sorting 404
 - 8.5.3 Minimum Spanning Trees 407
 - 8.5.4 Connectivity in a Graph 415
 - 8.5.5 Euler's and Hamiltonian Circuits 421
- 8.6 BDD and its Applications 425
 - 8.6.1 Conversion of Decision Tree into BDD 426
 - 8.6.2 Applications of BDD 428
- Problems to Ponder* 430
- References* 432

9 SETS

434–466

- 9.1 Definition and Terminologies 436
- 9.2 Representation of Sets 437
 - 9.2.1 List Representation of Set 437
 - 9.2.2 Hash Table Representation of Set 437
 - 9.2.3 Bit Vector Representation of Set 438
 - 9.2.4 Tree Representation of Set 439**
- 9.3 Operations of Sets 444
 - 9.3.1 Operation on List Representation of Set 445
 - 9.3.2 Operations on Hash Table Representation of Set 450
 - 9.3.3 Operations on Bit Vector Representation of Set 453
 - 9.3.4 Operation on Tree Representation of Set 455
- 9.4 Applications of Sets 460
 - 9.4.1 Spelling Checker 460
 - 9.4.2 Information System using Bit Strings 462
 - 9.4.3 Client-server Environment 464
- Problems to Ponder* 466
- References* 466

Preface

Data structures are commonly used in many program designs. The study of data structures, therefore, rightly forms the central course of any curriculum in computer science and engineering. Today, most curricula in computer science courses cover topics such as "Introduction to Computing", "Principles of Programming Languages", "Programming Methodologies", "Algorithms", etc. The study of these topics is not possible without first acquiring a thorough knowledge of data structures.

Today's computer world has become unimaginably fast. To use the full strength of computing power to solve all sorts of complicated problems through elegant program design, a good knowledge of the data structures is highly essential. To be more precise, writing efficient programs and managing different types of real and abstract data is an art; and data structure is the only ingredient to promote this art. In-depth concepts of data structures help in mastering their applications in real software projects.

In the last few years, there has been a tremendous progress in the field of data structures and its related algorithms. This book offers a deep understanding of the essential concepts of data structures. The book exposes the reader to different types of data structures such as arrays, linked lists, stacks, queues, tables, trees, graphs, and sets for a good grounding in each area. These data structures are known as classic data structures, as with the help of these any abstract data which fits with real world applications can be implemented.

The study of data structures remains incomplete if their computer representations and operational details are not covered. The books currently available on data structures present the operational details with raw codes, which many readers, especially those new to the field, find difficult to understand. In the present title, operations on data structures are described in English-like constructs, which are easy to comprehend by students new to the computer science discipline.

This text is designed primarily for use in undergraduate engineering courses, but its clear analytic explanations in simple language also make it suitable for study by polytechnic students. The book can also be used as a self-study course on data structures.

The book is designed to be both *versatile* and *complete*, in the sense that for each type of data structure three important topics are elucidated. First, various ways of representing a structure are explained. Second, the different operations to manage a structure are presented. Finally, the applications of a data structure with focus on its engineering issues are discussed. More than 300 figures have been used to make the discussions comprehensive and lucid. There are numerous section-wise exercises as "Assignment" in each chapter so that the readers can test their understanding of the subject. Also, the problems under the heading "Problems to Ponder" in each chapter, are planned for the advanced readers who can judge their grasp of the subject. A few references are included at the end of each chapter for advanced study.

As prerequisites, the students are expected to have experience of a programming language,

a little understanding of recursive procedures, introductory concepts of compiler, operating system, etc. The students are advised to go for generic implementation of algorithms with C++ so that once the data structures are programmed, they can be subsequently used in many different applications.

It is very difficult to avoid errors completely from a book of this nature. In spite of the immense amount of effort and attention to minute details, some errors might have still crept in. The author would welcome and greatly appreciate suggestions from the readers on making improvements to the book.

I would like to express my sincere gratitude to my friends and colleagues who contributed in many ways towards completion of the present work. I am grateful to the staff of North Eastern Regional Institute of Science and Technology (NERIST), Nirjuli, and Indian Institute of Technology Kharagpur who extended their help during the preparation of this book. My most heartfelt thanks go to Y. Usha for drawing all the figures in the book, and to N. Chetri and P. Kuli for typing the text.

I also wish to express my gratitude to the staff at Prentice-Hall of India, New Delhi for a masterful job of producing the finished volume.

Finally, I thank my wife Monalisa, a faculty member in the Department of Computer Science and Engineering, NERIST, who checked the manuscript with painstaking attention. Her sincere cooperation and involvement has made this work a reality. Our daughter Ananya grudgingly allowed me to sit with the computer to do the work. I affectionately dedicate this work to them.

D. SAMANTA

1.1 DEFINITIONS

Before going to study the actual subject, it will be a special advantage if we aware ourselves about the various terms involved in the subject. In the present section, basic terminologies and concepts are described with examples.

Data

Data means value or set of values. In both the singular and plural form, this term is data. Following are some examples of data:

- (i) 34
- (ii) 12/01/1965
- (iii) ISBN 81-203-0000-0
- (iv) 111
- (v) Pascal
- (vi) Æ
- (vii) 21, 25, 28, 30, 35, 37, 38, 41, 43

In all the above examples, some values are represented in different ways. Each value or collection of all such values is termed as data.

Entity

An *entity* is one that has certain attributes and which may be assigned values. For example, an employee in an organisation is an entity. The possible attributes and the corresponding values for an entity in the present example are:

Entity :	EMPLOYEE			
Attributes :	NAME	DOB	SEX	DESIGNATION
Values :	RAVI ANAND	30/12/61	M	DIRECTOR

All the employees in an organisation constitute an *entity set*. Each attribute of an entity set has a range of values, and is called the *domain* of attribute. Domain is the set of all possible values that could be assigned to the particular attribute. For example, in the EMPLOYEE entity, the attribute SEX has domain as {M, F}. It can be noted that, an entity after assigning its values results a composite data.

Information

The term *information* is used for data with its attribute(s). In other words, information can be

defined as meaningful data or processed data. For example, a set of data is presented during defining the term data, all these data become information if we impose the meaning as mentioned below related to a person:

Data	Meaning
34	Age of the person
12/01/1965	Date of birth of the person
ISBN 81-203-0000-0	Book number, recently published by the person
111	Number of awards achieved by the person in tally mark
Pascal	Nick name of the person
Æ	Signature of the person
21, 25, 28, 30, 35, 37, 38, 41, 43	Important ages of the person

Difference between data and information

From the definition of data and information it is evident that these two are not the same, however, there is a relation between them. Figure 1.1 shows the interrelation between data and information:

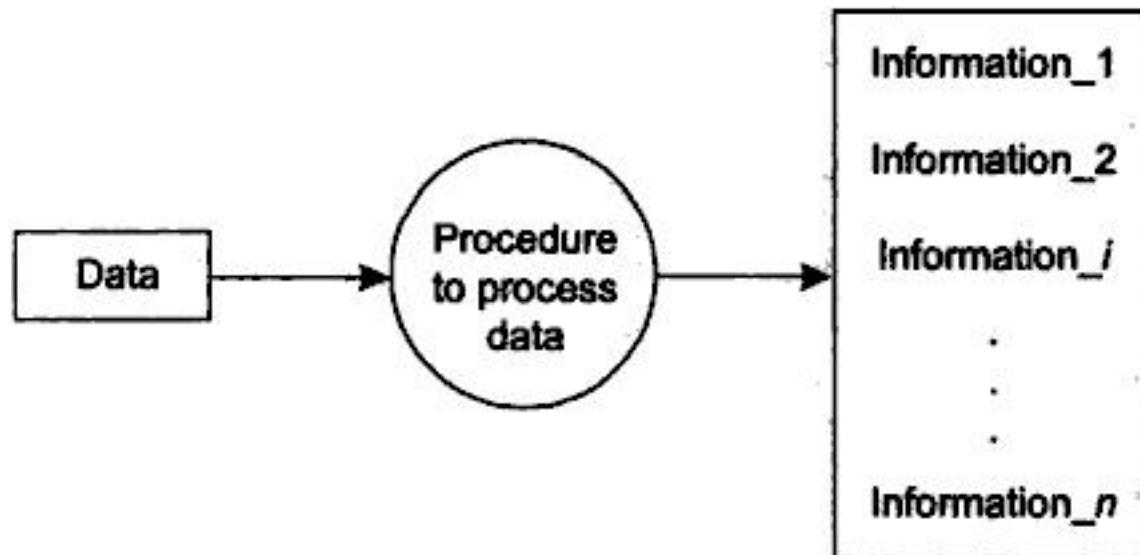


Fig. 1.1 Relation between data and information.

As an example, suppose there is a set of data comprising of the amount of milk consumed by a person in a month. From this given set of data information can be retrieved are as follows:

- (i) What is the total amount of milk consumed?
- (ii) In which day maximum milk is consumed?
- (iii) In which day minimum milk is consumed?
- (iv) What is the average amount of milk consumption per day?
- (v) What amount of carbohydrate is assimilated?
- (vi) What amount of protein is assimilated?
- (vii) What amount of fat is assimilated?
- (viii) What amount of mineral is assimilated?
- (ix) What amount of average calorie is gained per day?

and so on. To get these information from the given set of data, we are to define the processes

and then to apply the corresponding process on the data. (In fact, the terms, *data*, *entity* and *information* have no standard definition in computer science and they are often used interchangeably.)

Data type

A *data type* is a term which refers to the kind of data that may appear in computation. Following are a few well-known data types (see table):

<i>Data</i>	<i>Data type</i>
34	Numeric (integer)
12/01/1965	Date
ISBN 81-203-0000-0	Alphanumeric
	Graphics
Pascal	String
Æ	Image
21, 25, 28, 30, 35, 37, 38, 41, 43	Array of integers

Real, boolean, character, complex, etc., are also some more frequently used data types.

Built-in data type

With every programming language, there is a set of data types called *built-in data types*. For example, in C, FORTRAN and Pascal, the data types that are available as built-in are listed below:

C : int, float, char, double, enum etc.

FORTRAN : INTEGER, REAL, LOGICAL, COMPLEX, DOUBLE PRECISION, CHARACTER, etc.

Pascal : Integer, Real, Character, Boolean, etc.

With the built-in data types, programming languages allow users a lot of advantages regarding the processing of various types of data. For example, if a user declares a variable of type Real (say), then several things are automatically implied, such as how to store a value for that variable, what are the different operations possible on that type of data, what amount of memory is required to store, etc. All these things are taken care by the compiler or run-time system manager.

Abstract data type

When an application requires a special kind of data which is not available as built-in data type then it is the programmer's burden to implement his own kind of data. Here, the programmer has to give more effort regarding how to store value for that data, what are the operations that meaningfully manipulate variables of that kind of data, amount of memory require to store for a variable. The programmer has to decide all these things and accordingly implement it. Programmers' own data type is termed as *abstract data type*. Abstract data type is also alternatively termed as *user-defined data type*. For example, we want to process dates of the form dd/mm/yy. For

this, no built-in data type is known in C, FORTRAN, and Pascal. If a programmer wants to process dates, then an abstract data type, say Date, can be devised and various operations like: to add few days to a date to obtain another date, to find the days between two dates, to obtain a day for a given date, etc., have to be defined accordingly. Besides these, programmers should decide how to store the data, what amount of memory will be needed to represent a value of Date, etc. An abstract data type, in fact, can be built with the help of built-in data types and other abstract data type(s) already built by the programmer. Some programming languages allow facility to build abstract data type easily. For example, using struct/class in C/C++, and using record in Pascal, programmers can define their own data types.

1.2 CONCEPT OF DATA STRUCTURES

Digital computer can manipulate only primitive data, that is, data in terms of 0's and 1's. Manipulation of primitive data is inherent within the computer and need not require any extra effort from the user side. But in our real life applications, various kind of data other than the primitive data are involved. Manipulation of real-life data (can also be termed as user data) requires the following essential tasks:

1. Storage representation of user data: user data should be stored in such a way that computer can understand it.
2. Retrieval of stored data: data stored in a computer should be retrieved in such a way that user can understand it.
3. Transformation of user data: various operations which require to be performed on user data so that it can be transformed from one form to another.

Basic theory of computer science deals with the manipulation of various kinds of data, wherefrom the concept of data structures comes. In fact, data structure is the fundamentals in computer science. For a given kind of user data, its structure implies the following:

1. Domain (\mathcal{D}): This is the range of values that the data may have. This domain is also termed as data object.
2. Function (\mathcal{F}): This is the set of operations which may legally be applied to elements of data object. This implies that for a data structure we must specify the set of operations.
3. Axioms (\mathcal{A}): This is the set of rules with which the different operation belongs to \mathcal{F} actually can be implemented.

Now we can define the term data structure.

A *data structure* D is a triplet, that is, $D = (\mathcal{D}, \mathcal{F}, \mathcal{A})$ where \mathcal{D} is a set of data object, \mathcal{F} is a set of functions and \mathcal{A} is a set of rules to implement the functions. Let us consider an example.

We know for the integer data type (int) in C programming language the structure includes of the following type:

$$\mathcal{D} = (0, \pm 1, \pm 2, \pm 3, \dots)$$

$$\mathcal{F} = (+, -, *, /, \%)$$

$\mathcal{A} =$ (A set of binary arithmetics to perform addition, subtraction, division, multiplication, and modulo operations.)

It can be easily revealed that the triplet $(\mathcal{D}, \mathcal{F}, \mathcal{A})$ is nothing but an abstract data type. Also, the elements in set \mathcal{D} are not necessarily from primitive data it may contain from some other abstract data type. Alternatively, an implementation of a data structure D is a mapping from \mathcal{D} to a set of other data structures D_i , $i = 1, 2, \dots, n$, for some n . More precisely, this mapping specifies how every object of \mathcal{D} is to be represented by the objects of D_i , $i = 1, 2, \dots, n$. Every function of D must be written using the function of the implementing data structures D_i , $i = 1, 2, \dots, n$. The fact is that each of the implementing data structures is either primitive data type or abstract data type. We can conclude the discussion with another example.

Suppose, we want to implement a data type namely **complex** as abstract data type. Any variable of complex data type has two parts: real part and imaginary part. In our usual notation, if z is a complex number then $z = x + iy$, where x and y are real and imaginary parts respectively. Both x and y are of **Real** data type which is another abstract data type (available as built-in data type). So, abstract data type **complex** can be defined using the data structure **Real** as:

```
Complex z {
    x : Real
    y : Real
}
```

Now the set \mathcal{D} of **Complex** can be realised from the domain of x and y which is **Real** in this case. Let us specify the set of operations for the **Complex** data type, which are stated as \mathcal{F} :

$$\mathcal{F} = (\oplus, \ominus, \otimes, \oplus, \nabla, \parallel)$$

Assume that $z_1 = x_1 + iy_1$ and $z_2 = x_2 + iy_2$ are two data of **Complex** data type. Then we can define the rules for implementing the operations in \mathcal{F} giving thus axioms \mathcal{A} . In the current example, for **Complex** data type:

$$\begin{aligned} \mathcal{A} = & \{ \\ & z = z_1 \oplus z_2 = (x_1 + x_2) + i(y_1 + y_2) && \text{(complex addition)} \\ & z = z_1 \ominus z_2 = (x_1 - x_2) + i(y_1 - y_2) && \text{(complex subtraction)} \\ & z = z_1 \otimes z_2 = (x_1 \times x_2 - y_1 \times y_2) + i(x_1 \times y_2 + x_2 \times y_1) && \text{(complex multiplication)} \\ & z = z_1 \oplus z_2 \\ & \quad = \frac{x_1 \times x_2 + y_1 \times y_2}{x_2^2 + y_2^2} + i \frac{x_2 \times y_1 - x_1 \times y_2}{x_2^2 + y_2^2} && \text{(complex division)} \\ & z = \nabla z_1 = \frac{x_1}{x_1^2 + y_1^2} - i \frac{y_1}{x_1^2 + y_1^2} && \text{(complex conjugate)} \\ & Z = |z_1| = \sqrt{x_1^2 + y_1^2} && \text{(complex magnitude)} \\ & \} \end{aligned}$$

Note that, how different operations of **Complex** data type can be implemented using the operations $+$, $-$, \times , $/$, $\sqrt{}$, of implementing data structure, namely **Real**.

Assignment 1.1 Implement *Date* as an abstract data type which comprises of dd/mm/yy, where dd varies from 1 to 31, mm varies from 1 to 12 and yy is any integer with four digits.

Specify \mathcal{F} consisting of all possible operations on variables of type *Date* and then define \mathcal{A} to implement all the operations in \mathcal{F} . Assume the implementing data structure(s) which is/are necessary.

1.3 OVERVIEW OF DATA STRUCTURES

In computer science, several data structures are known depending on area of applications. Of them, few data structures are there which are frequently used almost in all application areas and with the help of which almost all complex data structure can be constructed. These data structures are known as *fundamental data structures* or *classic data structures*. Figure 1.2 gives a classification of all classic data structures.

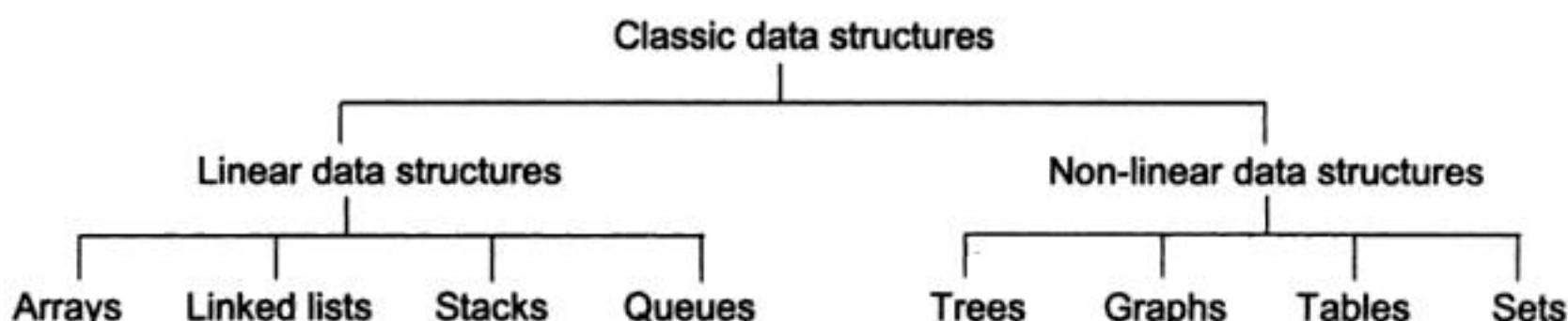


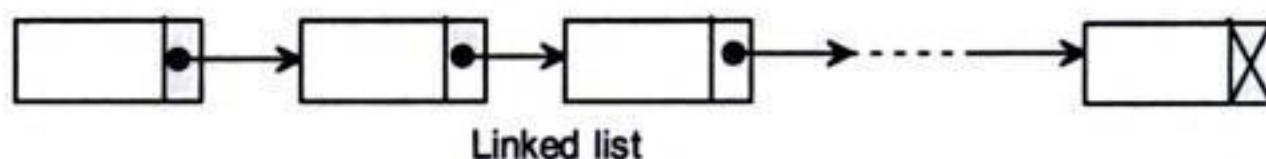
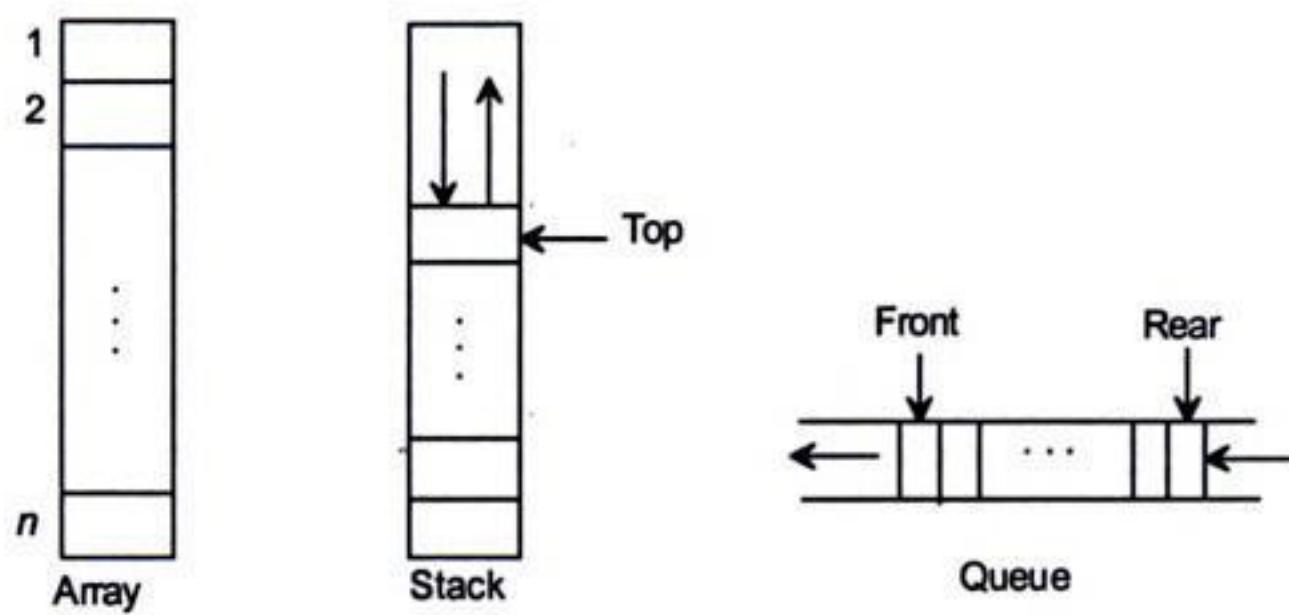
Fig. 1.2 Classification of classic data structures.

In addition to these classic data structures, other data structures such as lattice, Petri nets, neural nets, search graphs, semantic nets, etc., are known in various applications. These are known to be very complex data structures. (Discussion of all these complicated data structures is beyond the scope of this book.)

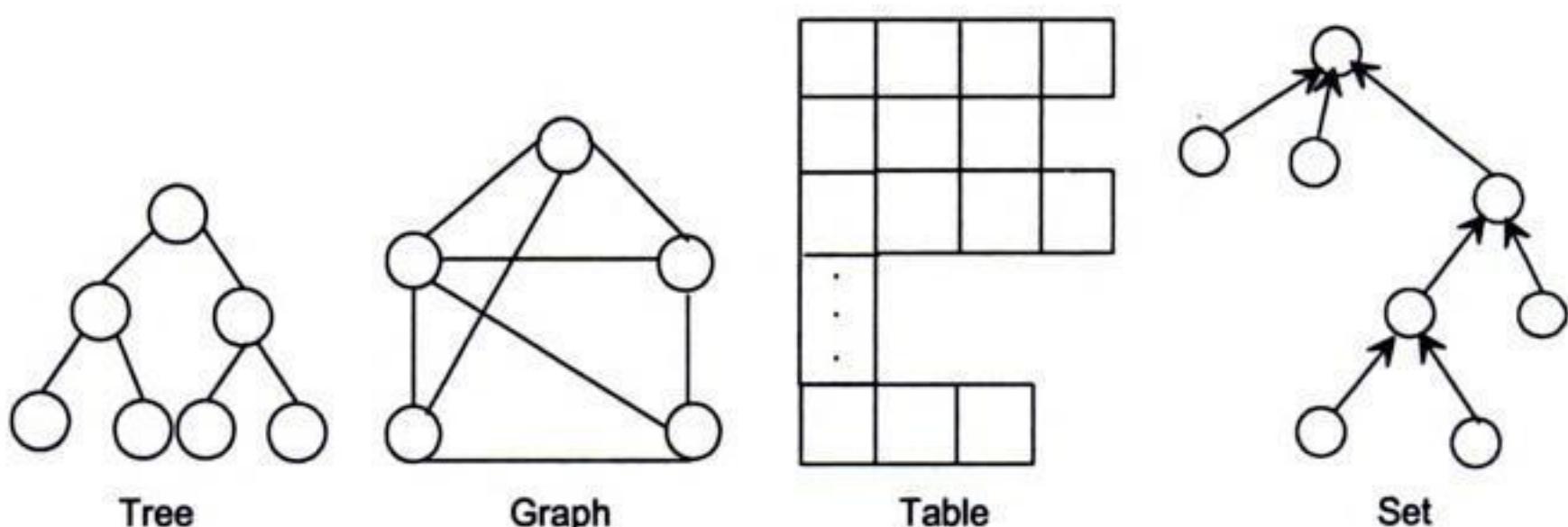
It can be observed that all the classic data structures are classified into two main classes: *linear data structures* and *non-linear data structures*. In case of linear data structures, all the elements form a sequence or maintain a linear ordering. On the other hand, no such sequence in elements, rather all the elements are distributed over a plane in case of non-linear data structures. Again within each classical data structure there are a number of variations as depicted in Figure 1.2. In this book, we have planned for eight chapters to discuss eight classic data structures separately. As an overview, all the classic data structures are primarily introduced which can be seen in Figure 1.3.

Assignment 1.2 Devise a data structure (say Graphic) to represent a graphics data as shown in Figure 1.4. Also, define the following operation on any data of this type:

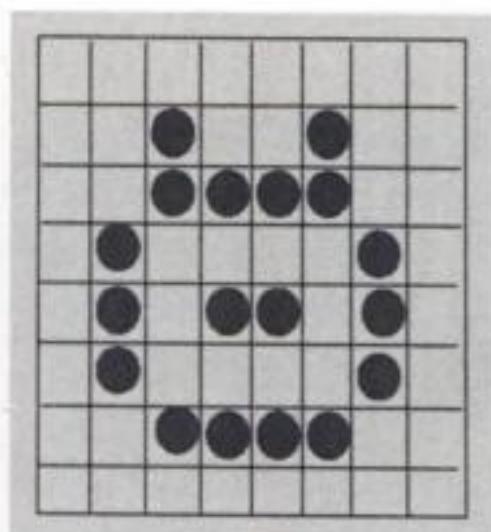
- (a) ScaleUp(P): To magnify the image by certain per cent P
- (b) ScaleDn(P): To reduce the image by certain per cent P
- (c) Rotate(R, A): To rotate the image either clockwise ($R = +$) or anticlockwise ($R = -$) by a certain angle A .



(a) Linear data structures



(b) Non-linear data structures

Fig. 1.3 Overview of classic data structures.**Fig. 1.4**

1.4 IMPLEMENTATION OF DATA STRUCTURES

We will implement all the classic data structures as mentioned in the previous section in two phases:

Phase 1

Storage representation: Here we have decided how a data structure can be stored in computer memory. This storage representation, in general, is based on the use of other data structures.

Phase 2

Algorithmic notation of various operations of the classic data structure: Function for manipulating a data structure is expressed in terms of algorithms so that details of the operation can be understood easily and reader can implement them with the help of any programming language. It will be preferable to use C/C++ programming because of their versatile features and modern programming approaches.

In order to express the algorithm for a given operation, we will assume different control structures and notations as stated below:

Algorithm <Name of the operation>(Input parameters; Output parameters)

Input: <Specification of input data for the operation>

Output: <Specification of output after the successful performance of the operation>

Data structure: <If the operation assumes other data structure for its implementation>

Steps:

1.
2. **If** <condition> **then**
 1.
 2.
 -
 -
3. **Else**
 1.
 2.
 -
 -
4. **EndIf**
5.
6. **While** <condition> **do**
 1.
 2.
 -
 -
7. **EndWhile**
8.

9. **For** <loop condition> **do**
 1.
 2.
 -
 -
10. **EndFor**
11. ...
12. **Stop**

Different statements that will be used in the algorithms are very much similar to the statement in C language. Sufficient comment will be introduced side by side against statements to indicate what step that a statement will perform. The convention of putting comment is similar to the convention of placing comments in C/C++ programming language.

Another convention that we will assume is the naming of variables. The variable names are either in capital or small letters. The variable names in capital letters will be treated as either constant or as input to the algorithm. All the variable names in small letters are local to the algorithm and will be treated as temporary variables.

'Problems to Ponder' at the end of each chapter includes problems related to the topics discussed in the concerned chapter. In addition to, a number of 'Assignments' are also provided during the discussion in each chapter. Students can improve their level of learning if they practise these assignments and solve problems.

2 *Arrays*

If we want to store a group of data together in one place then array is the one data structure we are looking for. This data structure enables us to arrange more than one element, that is why it is termed as composite data structure. In this data structure, all the elements are stored in contiguous locations of memory. Figure 2.1 shows an array of data stored in a memory block starting at location 453.

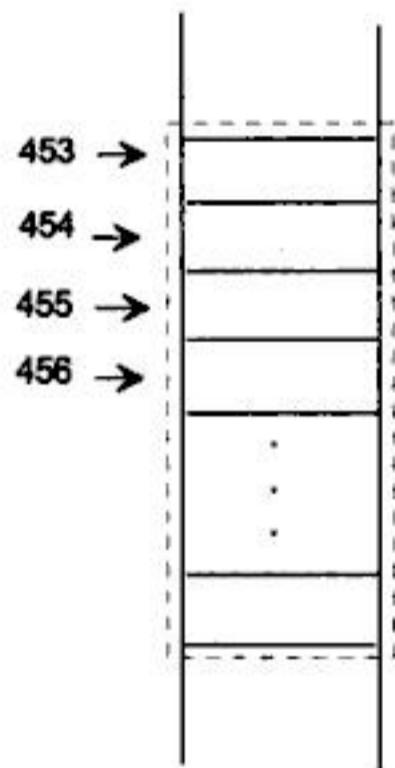


Fig. 2.1 Array of data.

2.1 DEFINITION

An *array* is a finite, ordered collection of homogeneous data elements. Array is finite because it contains only limited number of elements; and ordered, as all the elements are stored one by one in contiguous locations of computer memory in a linear ordered fashion. All the elements of an array are of the same data type (say, integer) only and hence it is termed as collection of homogeneous elements. Following are some examples:

1. An array of integers to store the age of all students in a class.
2. An array of strings (of characters) to store the name of all villagers in a village.

Note: 1. An array is known as linear data structure because, all elements of the array are stored in a linear order.

2. The data structure is generally a built-in data type in all programming languages. Declaration syntax for an array A, say, of 100 integers in BASIC, FORTRAN, Pascal and C are as below:

BASIC	: DIMENSION A[100]
FORTRAN	: DIM A[100]
Pascal	: A: ARRAY[1...100] of integer
C	: int A[100]

2.2 TERMINOLOGY

Size. Number of elements in an array is called the *size* of the array. It is also alternatively termed as *length* or *dimension*.

Type. *Type* of an array represents the kind of data type it is meant for. For example, array of integers, array of character strings, etc.

Base. *Base* of an array is the address of memory location where the first element in the array is located. For example, 453 is the base address of the array as mentioned in Figure 2.1.

Index. All the elements in an array can be referenced by a subscript like A_i or $A[i]$, this subscript is known as *index*. Index is always an integer value. As each array elements is identified by a subscript or index that is why an array element is also termed as *subscripted* or *indexed variable*.

Range of index. Indices of array elements may change from a lower bound (L) to an upper bound (U), which are called the boundaries of an array.

In a declaration of an array in FORTRAN (DIMENSION A[100]), range of index is 1 to 100. For the same array in C (int A[100]) the range of index is from 0 to 99. These are all default range of indices. However in Pascal, a user can define the range of index for any lower bound to upper bound, for example, for A: ARRAY[-5...19] of integer, the points of the range is -5, -4, -3, ..., 18, 19. Here, the index of i -th element is $-5 + i - 1$. In terms of L , the lower bound, this formula stands as:

$$\text{Index } (A_i) = L + i - 1$$

If the range of index varies from $L \dots U$ then the size of the array can be calculated as

$$\text{Size } (A) = U - L + 1$$

Word. Word denotes the size of an element. In each memory location, computer can store an element of word size w , say. This word size varies from machine to machine such as 1 byte (in IBM CPU-8085) to 8 bytes (in Intel Pentium PCs). Thus, if the size of an element is double the word size of a machine then to store such an element, it requires two consecutive memory locations.

2.3 ONE-DIMENSIONAL ARRAY

If only one subscript/index is required to reference all the elements in an array then the array will be termed as *one-dimensional* array or simply an array.

2.3.1 Memory Allocation for an Array

Memory representation of an array is very simple. Suppose, an array $A[100]$ is to be stored in a memory as in Figure 2.2. Let the memory location where the first element can be stored is M . If each element requires one word then the location for any element say $A[i]$ in the array can be obtained as:

$$\text{Address } (A[i]) = M + (i - 1)$$

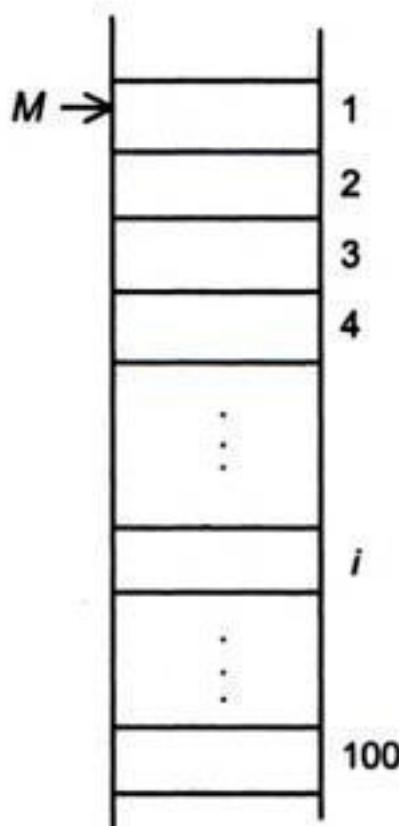


Fig. 2.2 Physical representation of a one-dimensional array.

Likewise, in general, an array can be written as $A[L \dots U]$, where L and U denote the lower and upper bounds for index. If it is stored starting from memory location M , and for each element it requires w number of words, then the address for $A[i]$ will be

$$\text{Address } (A[i]) = M + (i - L) \times w$$

The above formula is known as *indexing formula*; which is used to map the logical presentation of an array to physical presentation. By knowing the starting address of an array M , the location of i -th element can be calculated instead of moving towards i from M , Figure 2.3.

Assignment 2.1 Suppose, an array $A [-15 \dots 64]$ is stored in a memory whose starting address is 459. Assume that word size for each element is 2. Then obtain the following:

- How many number of elements are there in the array A ?
- If one word of the memory is equal to 2 bytes, then how much memory is required to store the entire array?
- What is the location for $A[50]$?
- What is the location of 10-th element?
- Which element is located at 589?

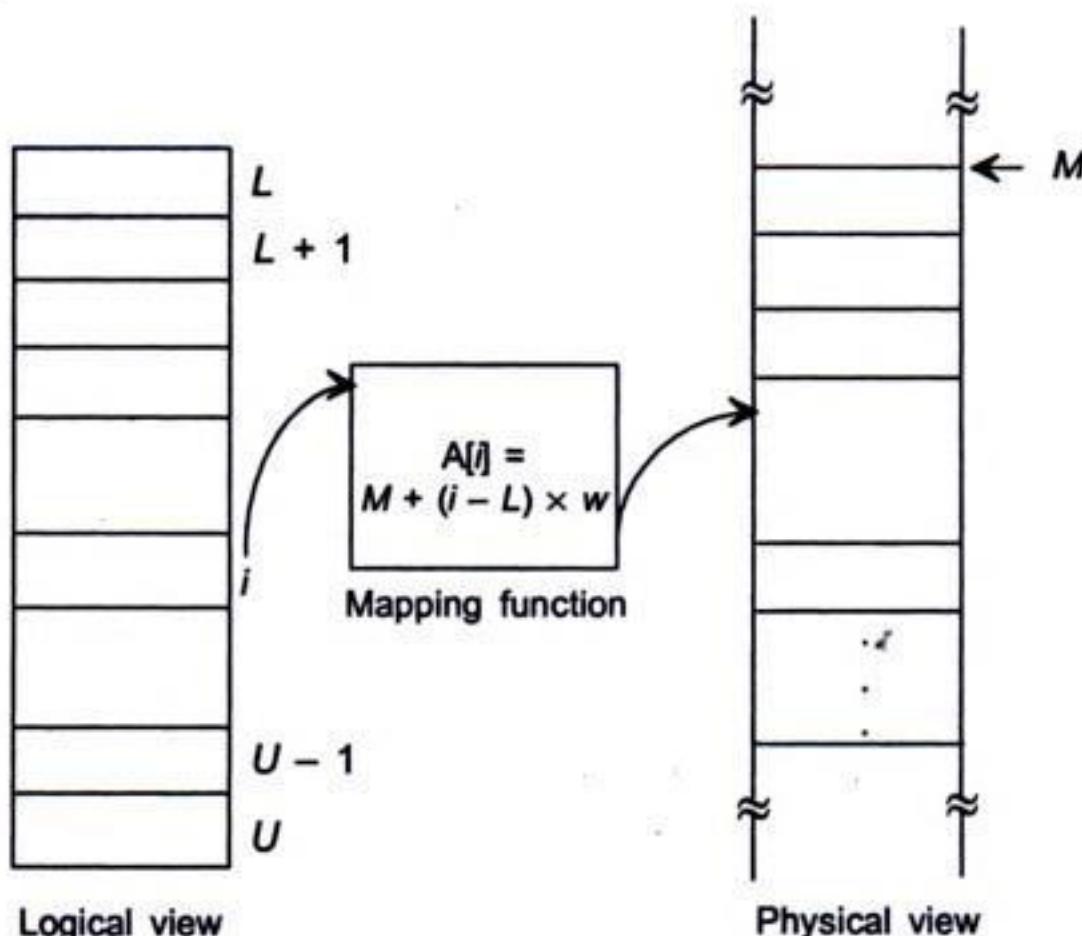


Fig. 2.3 Address mapping between logical and physical views of an array.

Assignment 2.2 In the indexing formula in Section 2.3.1, we have taken that the array is stored from lower region of the memory to the higher region. But in some computer the convention is just reverse, that is, starting an array i at higher region and hence Address (A_i) > Address (A_{i+1}) for all $L < i < U$. Modify the indexing formula for this case.

2.3.2 Operations on Arrays

Various operations that can be performed on an array are: traversing, sorting, searching, insertion, deletion, merging.

Traversing

This operation is used visiting all elements in an array. A simplified algorithm is presented as below:

Algorithm TRAVERSE ARRAY()

Input: An array A with elements.

Output: According to PROCESS().

Data structures: Array $A[L \dots U]$.

//*L* and *U* are the lower and upper bound
//of array index

Steps:

- ```

1. $i = L$ //Start from first location L
2. While $i \leq U$ do
 1. PROCESS($A[i]$)
 2. $i = i + 1$ //Move to the next location
3. EndWhile
4. Stop

```

**Note:** Here PROCESS( ) is a procedure which when called for an element can perform an action. For example, display the element on the screen, determine whether  $A[i]$  is empty or not, etc. PROCESS( ) also can be used to manipulate some special operations like count the special element of interest (for example, negative numbers in an integer array), updating of each element (say, increase each by 10%) and so on.

### Sorting

This operation, if performed on an array, will sort it in a specified order (ascending/descending). The following algorithm is used to store the elements of an integer array in ascending order.

#### Algorithm SORT\_ARRAY()

**Input:** An array with integer data.

**Output:** An array with sorted elements in an order according to ORDER( ).

**Data structures:** An integer array  $A[L \dots U]$ . // $L$  and  $U$  are the lower and upper bound of //array index

#### Steps:

1.  $i = U$
2. While  $i \geq L$  do
  1.  $j = L$  //Start comparing from first
  2. While  $j < i$  do
    1. If  $\text{ORDER}(A[j], A[j + 1]) = \text{FALSE}$  //If  $A[j]$  and  $A[j + 1]$  are not in order
      1.  $\text{SWAP}(A[j], A[j + 1])$  //Interchange the elements (see Figure 2.4)
    2. EndIf
    3.  $j = j + 1$  //Go to next element
  3. EndWhile
  4.  $i = i - 1$
3. EndWhile
4. Stop

Here, ORDER(...) is a procedure to test whether two elements are in order and SWAP(...) is a procedure to interchange the elements between two consecutive locations.

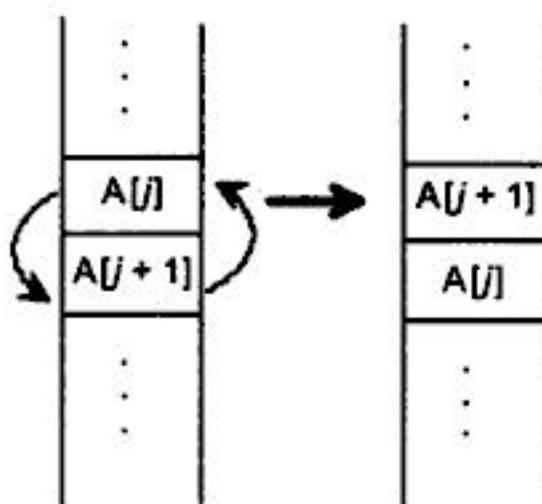


Fig. 2.4 Swapping of two elements in an array.

**Assignment 2.3** Write algorithms to

- Sort an array of integers in ascending order.
- Sort an array of integers in descending order.
- Sort an array of string of characters in lexicographic order.
- Sort an array of an abstract data type.

*Hint:* You have to think of only little modification of procedure ORDER(...) in each case.

## Searching

This operation is applied to search an element of interest in an array. The simplified version of the algorithm is as follows:

### Algorithm SEARCH\_ARRAY(KEY)

**Input:** KEY is the element to be searched.

**Output:** Index of KEY in A or a message on failure.

**Data structures:** An array A[L ... U]. //L and U are the lower and upper bound of array index

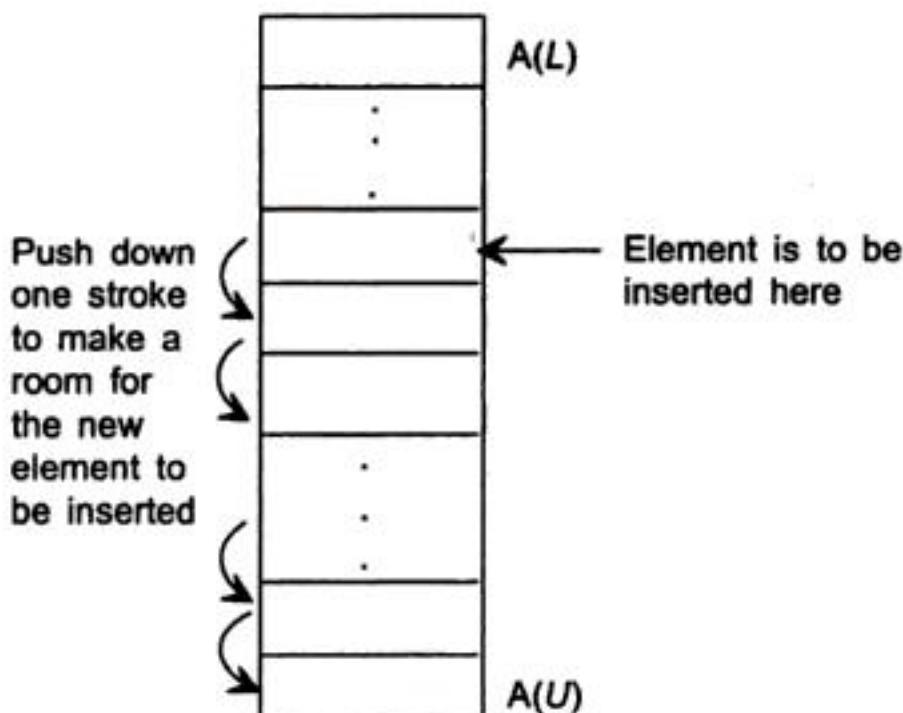
#### Steps:

- i = L, found = 0, location = 0* //found = 0 indicates search is not finished and unsuccessful
- While (*i ≤ U*) and (*found = 0*) do // Continue if all or any one condition do(es) not satisfy
  - If COMPARE(A[*i*], KEY) = TRUE then //If key is found
    - found = 1* //Search is finished and successful
    - location = i*
  - Else
    - i = i + 1* //Move to the next
  - EndIf
- EndWhile
- If *found = 0* then
  - Print "Search is unsuccessful : KEY is not in the array"
- Else
  - Print "Search is successful : KEY is in the array at location", *location*
- EndIf
- Return(*location*)
- Stop

**Assignment 2.4** In the algorithm discussed above assume that elements are randomly distributed. Modify the algorithm to give more faster algorithm when the elements are already in sorted (either ascending or descending) order.

## Insertion

This operation is used to insert an element into an array provided that the array is not full (see Figure 2.5). Let us observe the algorithmic description for the insertion operation.

**Fig. 2.5** Insertion of an element to an array.**Algorithm INSERT(KEY, LOCATION)**

**Input:** KEY is the item, LOCATION is the index of the element where it is to be inserted.

**Output:** Array enriched with KEY.

**Data structures:** An array  $A[L \dots U]$ . //L and U are the lower and upper bound of array index

**Steps:**

1. If  $A[U] \neq \text{NULL}$  then
  1. Print "Array is full: No insertion possible"
  2. Exit //End of execution
2. Else
  1.  $i = U$  //Start pushing from bottom
  2. While  $i > \text{LOCATION}$  do
    1.  $A[i + 1] = A[i]$
    2.  $i = i - 1$
  3. EndWhile
  4.  $A[\text{LOCATION}] = \text{KEY}$  //Put the element at desired location
  5.  $U = U + 1$  //Update the upper index of the array
3. EndIf
4. Stop

**Assignment 2.5** The algorithm INSERT only checks the last element for vacancy. But an array may be empty from any  $i$ -th position ( $L \leq i \leq U$ ); in that case number of push down can be reduced instead of pushing down the entire trailing part. Modify the above algorithm INSERT when the last element is at  $i$ -th location. ( $i \leq U$ ).

**Assignment 2.6** How can an empty array be defined? Verify that whether all the algorithms discussed so far in this chapter can work with empty array. If not, modify algorithm(s) so that they can work even for empty array(s).

**Deletion**

This operation is used to delete a particular element from an array. The element will be deleted by overwriting it with its subsequent element and this subsequent element then is to be deleted. In other words, push the tail (the part of the array after the elements which are to be deleted) one stroke up. Consider the following algorithm to delete an element from an array:

**Algorithm DELETE(KEY)**

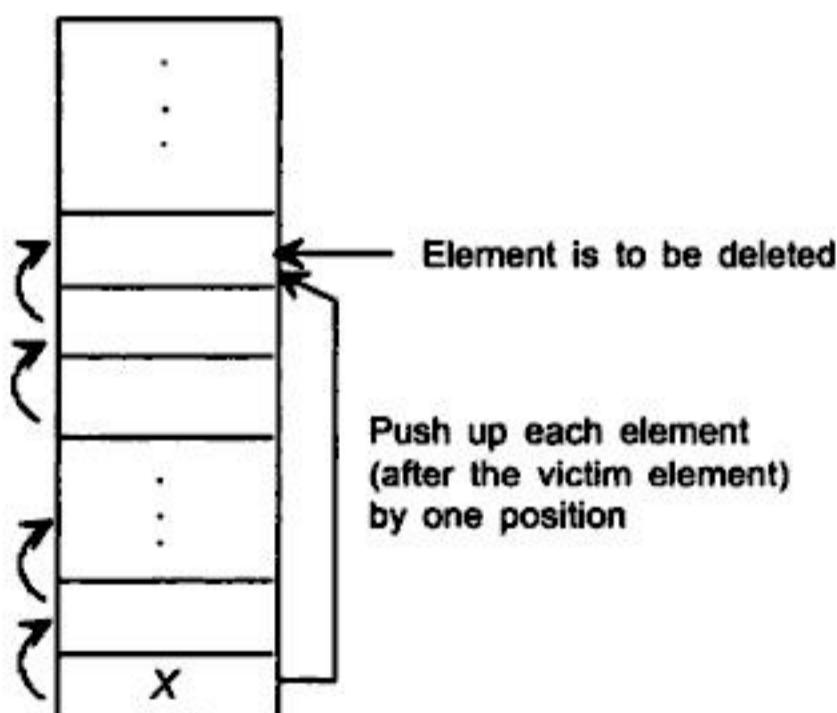
**Input:** KEY the element to be deleted.

**Output:** Slimed array without KEY.

**Data structures:** An array A[L...U]. //L and U are the lower and upper bound of //array index

**Steps:**

1.  $i = \text{SEARCH\_ARRAY}(A, \text{KEY})$  //Perform the search operation on A and return  
//the location
2. If ( $i = 0$ ) then
  1. Print "KEY is not found: No deletion"
  2. Exit //Exit the program
3. Else
  1. While  $i < U$  do
    1.  $A[i] = A[i + 1]$  //Replace the element by its successor
    2.  $i = i + 1$
  2. EndWhile
4. EndIf
5.  $A[U] = \text{NULL}$  //The bottom most element is made empty  
//(see Figure 2.6)
6.  $U = U - 1$  //Updated the upper bound now
7. Stop

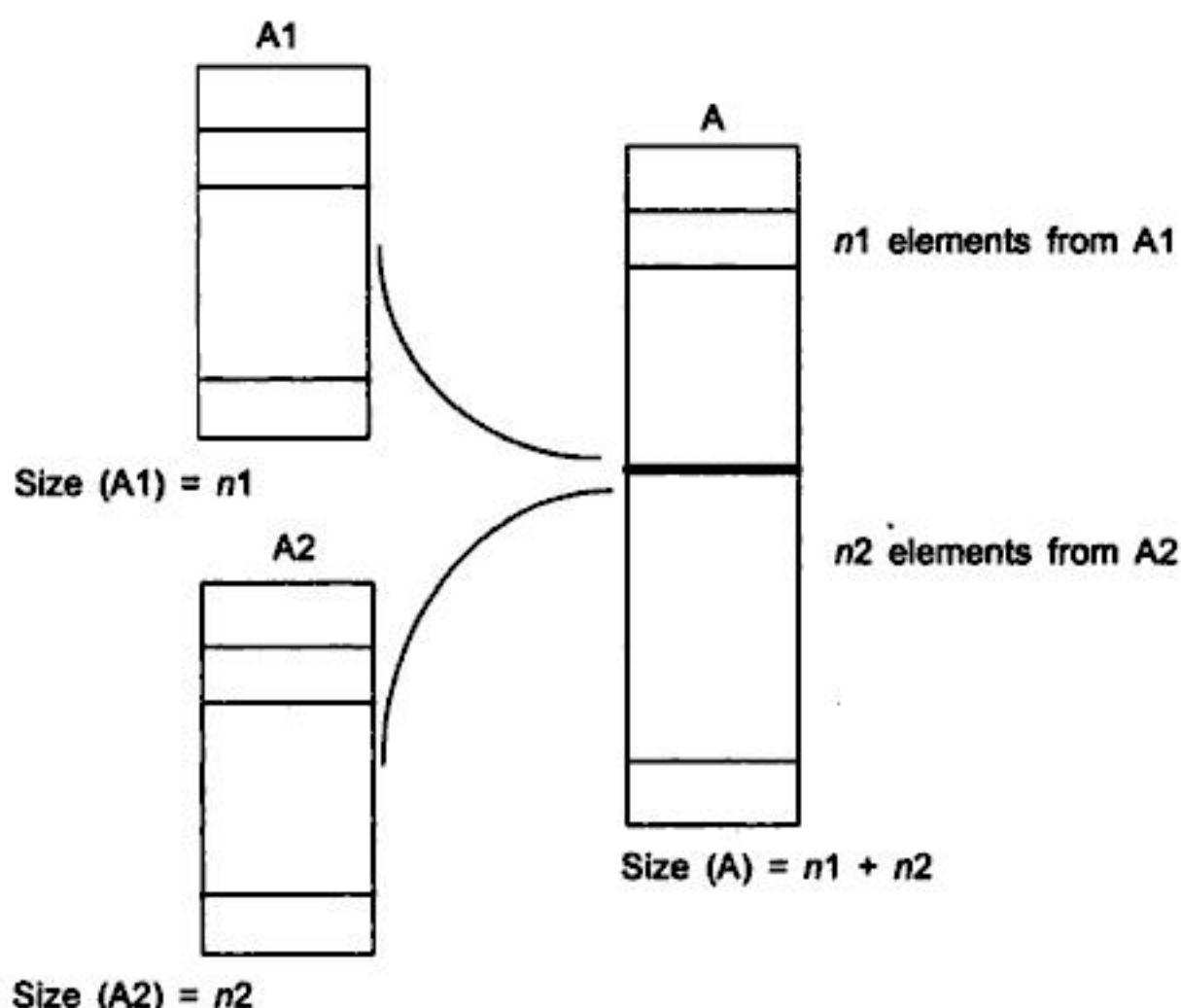


**Fig. 2.6** Deletion of an element from an array.

**Note:** It is a general practice that no intermediate location will be made empty, that is, an array should be packed and empty locations are at the tail of an array.

### Merging

Merging is an important operation when we need to compact the elements from two different arrays into a single array (see Figure 2.7).



**Fig. 2.7** Merging of A<sub>1</sub> and A<sub>2</sub> to A.

#### Algorithm MERGE(A<sub>1</sub>, A<sub>2</sub>; A)

**Input:** Two arrays A<sub>1</sub>[L<sub>1</sub> ... U<sub>1</sub>], A<sub>2</sub>[L<sub>2</sub> ... U<sub>2</sub>].

**Output:** Resultant array A[L ... U], where L = L<sub>1</sub>, and U = U<sub>1</sub> + (U<sub>2</sub> - L<sub>2</sub> + 1) when A<sub>1</sub> is appended after A<sub>2</sub>.

**Data structures:** Array structure.

#### Steps:

1.  $i_1 = L_1$ ,  $i_2 = L_2$ , //Initialization of control variables
2.  $L = L_1$ ,  $U = U_1 + U_2 - L_2 + 1$  //Initialization of lower and upper bounds of resultant array A
3.  $i = L$
4. Allocate memory for A[L ... U]
5. While  $i_1 \leq U_1$  do //To copy array A<sub>1</sub> into the first part of A
  1.  $A[i] = A_1[i_1]$
  2.  $i = i + 1$ ,  $i_1 = i_1 + 1$
6. EndWhile
7. While  $i_2 \leq U_2$  do //To copy the array A<sub>2</sub> into last part of A
  1.  $A[i] = A_2[i_2]$
  2.  $i = i + 1$ ,  $i_2 = i_2 + 1$
8. EndWhile
9. Stop

**Assignment 2.7** Modify the algorithm MERGE for the following cases of (i) if both  $A_1$  and  $A_2$  are empty; (ii) either  $A_1$  or  $A_2$  is empty.

### 2.3.3 Application of Arrays

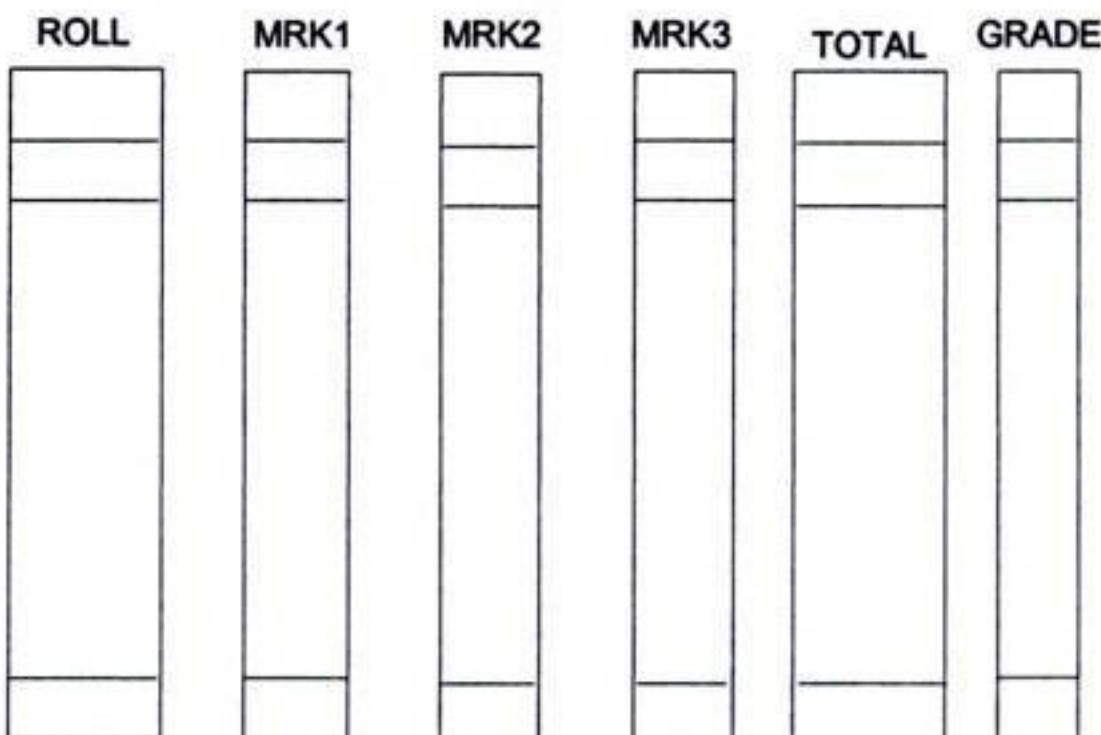
There are wide applications of arrays in computation. This is why almost every programming language include this data type as a built-in data type. A simple example that can be cited is to store some data before performing their manipulation. Another simple example is mentioned as below:

Suppose, we want to store records of all students in a class. The record structure is given by

STUDENTS

| ROLL NO<br>(Alphanumeric) | MARK1<br>(Numeric) | MARK2<br>(Numeric) | MARK3<br>(Numeric) | TOTAL<br>(Numeric) | GRADE<br>(Character) |
|---------------------------|--------------------|--------------------|--------------------|--------------------|----------------------|
|---------------------------|--------------------|--------------------|--------------------|--------------------|----------------------|

If sequential storage of records is not an objection, then we can store the records by maintaining 6 arrays whose size is specified by the total number of students in the class, as in Figure 2.8.



**Fig. 2.8** Storing the students' records.

#### Assignment 2.8 (Practical)

1. Declare 6 arrays as required to store the students records each of size say, 30.
2. Initialize each array with appropriate data.
3. Use the following operations on your data structures
  - (a) List all the records on the screen
  - (b) Search for a record whose Roll No is say, X
  - (c) Delete a record whose Roll No is say, X
  - (d) Insert a new records just after the record, whose Roll No = X

- (e) Sort all the records according to the descending order of TOTAL  
 (f) Suppose, we want to merge the records of two classes. Obtain the merging in such a case.

*Note:* Carry out all the above tasks using the operations those already defined for array.

4. Give an idea how a polynomial with three variables, say  $X$ ,  $Y$  and  $Z$  can be represented using a one-dimensional array.

## 2.4 MULTIDIMENSIONAL ARRAYS

So far we have discussed the one dimensional arrays. But multidimensional arrays are also important. Matrix (2-dimensional array), 3-dimensional array are two examples of multidimensional arrays. The following sections describe the multidimensional arrays.

### 2.4.1 Two-dimensional Arrays

Two-dimensional arrays (alternatively termed as matrices) are the collection of homogeneous elements where the elements are ordered in a number of rows and columns. An example of an  $m \times n$  matrix where  $m$  denotes number of rows and  $n$  denotes number of columns is as follows:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & \dots & a_{1n} \\ a_{21} & a_{22} & a_{23} & a_{24} & \dots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & a_{m3} & a_{m4} & \dots & a_{mn} \end{bmatrix}_{m \times n}$$

The subscripts of any arbitrary element, say  $(a_{ij})$  represent the  $i$ -th row and  $j$ -th column.

#### **Memory representation of a matrix**

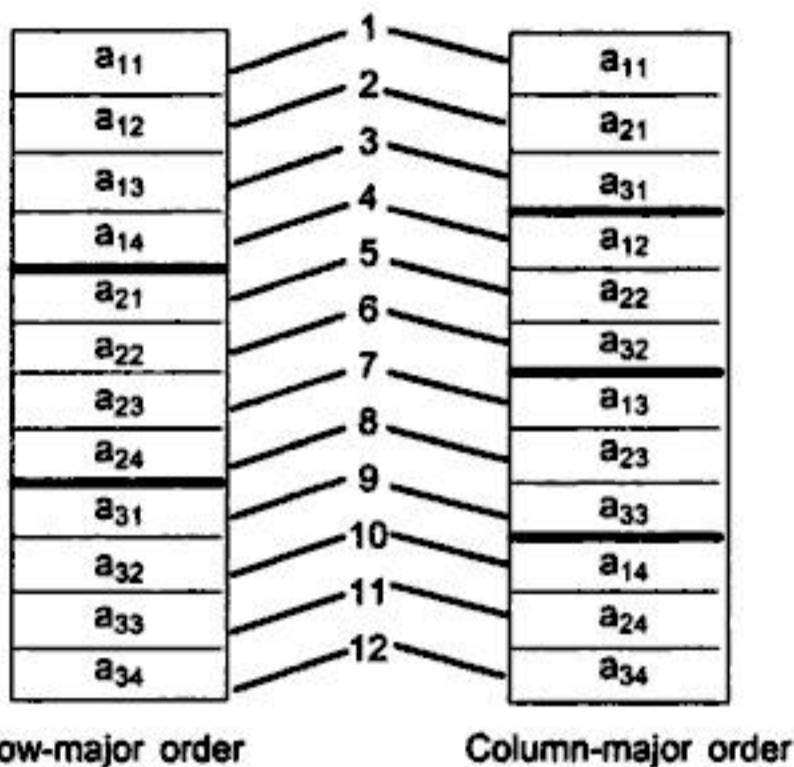
Like one-dimensional array, matrices are also stored in contiguous memory locations. There are two conventions of storing any matrix in memory:

1. Row-major order
2. Column-major order.

In row-major order, elements of a matrix are stored on a row-by-row basis, that is, all the elements in first row, then in second row and so on. On the other hand, in column-major order, elements are stored column-by-column, that is, all the elements in first column are stored in their order of rows, then in second column, third column and so on. For example, consider a matrix A of order  $3 \times 4$ :

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \end{bmatrix}_{3 \times 4}$$

This matrix can be represented in memory as shown in Figure 2.9.



**Fig. 2.9** Memory representation of  $A_{3 \times 4}$  matrix.

### Reference of elements in a matrix

Logically, a matrix appears as two-dimensional; but physically it is stored in a linear fashion. So, in order to map from logical view to physical structure, we need indexing formula. Obviously, the indexing formula for different order will be different. The indexing formulae, for different orders are stated as below:

**Row-major order.** Assume that the base address is the first location of the memory, that is,

1. So, the address of  $a_{ij}$  will be obtained as

$$\begin{aligned} \text{Address } (a_{ij}) &= \text{Storing all the elements in first } (i-1)\text{-th rows} \\ &\quad + \text{The number of elements in } i\text{-th row up to } j\text{-th column.} \\ &= (i-1) \times n + j \end{aligned}$$

So, for the matrix  $A_{3 \times 4}$ , the location of  $a_{32}$  will be calculated as 10 (see Figure 2.9). Instead of considering the base address to be 1, if it is at  $M$ , then the above formula can be easily modified as:

$$\text{Address } (a_{ij}) = M + (i-1) \times n + j - 1$$

**Column-major order.** Let us first consider the starting location of matrix is at memory location 1. Then

$$\begin{aligned} \text{Address } (a_{ij}) &= \text{Storing all the elements in first } (j-1)\text{-th columns} \\ &\quad + \text{The number of elements in } j\text{-th column up to } i\text{-th rows.} \\ &= (j-1) \times m + i \end{aligned}$$

And considering the base address at  $M$  instead of 1, the above formula will stand as

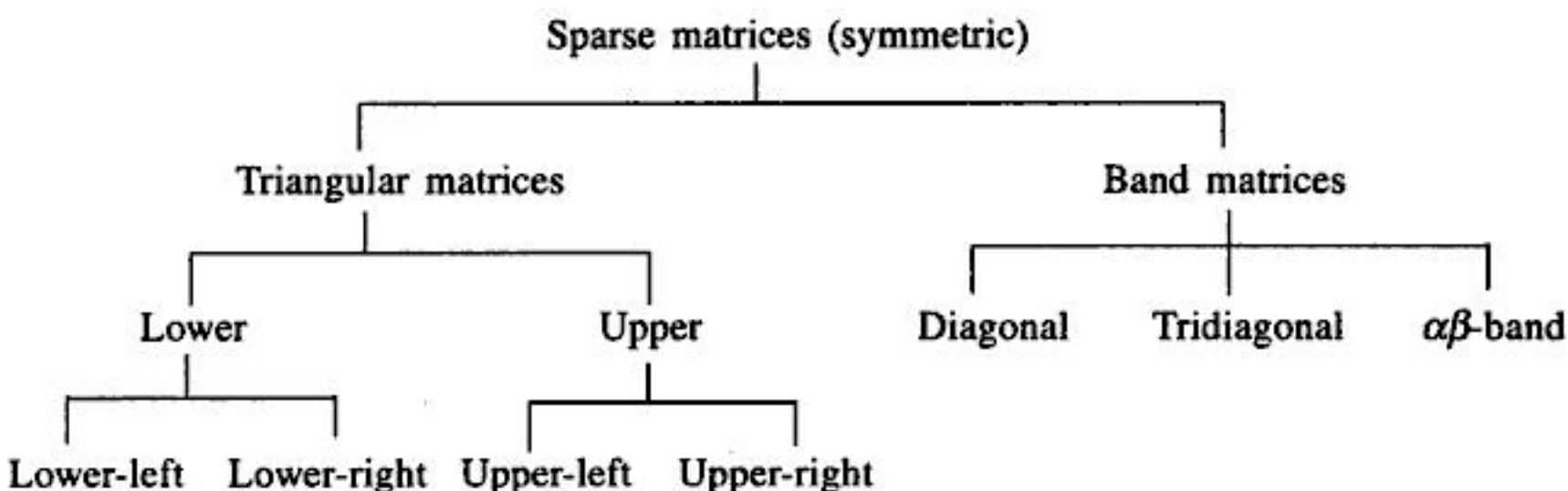
$$\text{Address } (a_{ij}) = M + (j-1) \times m + i - 1$$

### 2.4.2 Sparse Matrices

A *sparse matrix* is a two-dimensional array having the value of majority elements as null. Following is a sparse matrix where '\*' denotes the elements having non-null values.

$$\begin{bmatrix} \cdots & * & \cdots & * & \cdots & * & \cdots \\ \cdots & * & \cdots & * & \cdots & \cdots & \cdots \\ \cdots & \cdots & * & * & * & \cdots & \cdots \\ * & \cdots & * & \cdots & * & \cdots & * \\ \cdots & * & \cdots & * & \cdots & * & \cdots \\ \cdots & * & \cdots & * & \cdots & * & \cdots \\ \cdots & * & \cdots & * & \cdots & * & \cdots \\ \cdots & * & \cdots & * & \cdots & * & \cdots \end{bmatrix}$$

In large number of applications, sparse matrices are involved. So far storage of a sparse matrix is concerned, storing of null elements is nothing but wastage of memory. So we should devise a technique such that only non-null elements will be stored. One approach is to use the linked list (this will be discussed in Chapter 3, Section 3.6.1). Some well-known sparse matrices which are symmetric in form can be classified as follows:

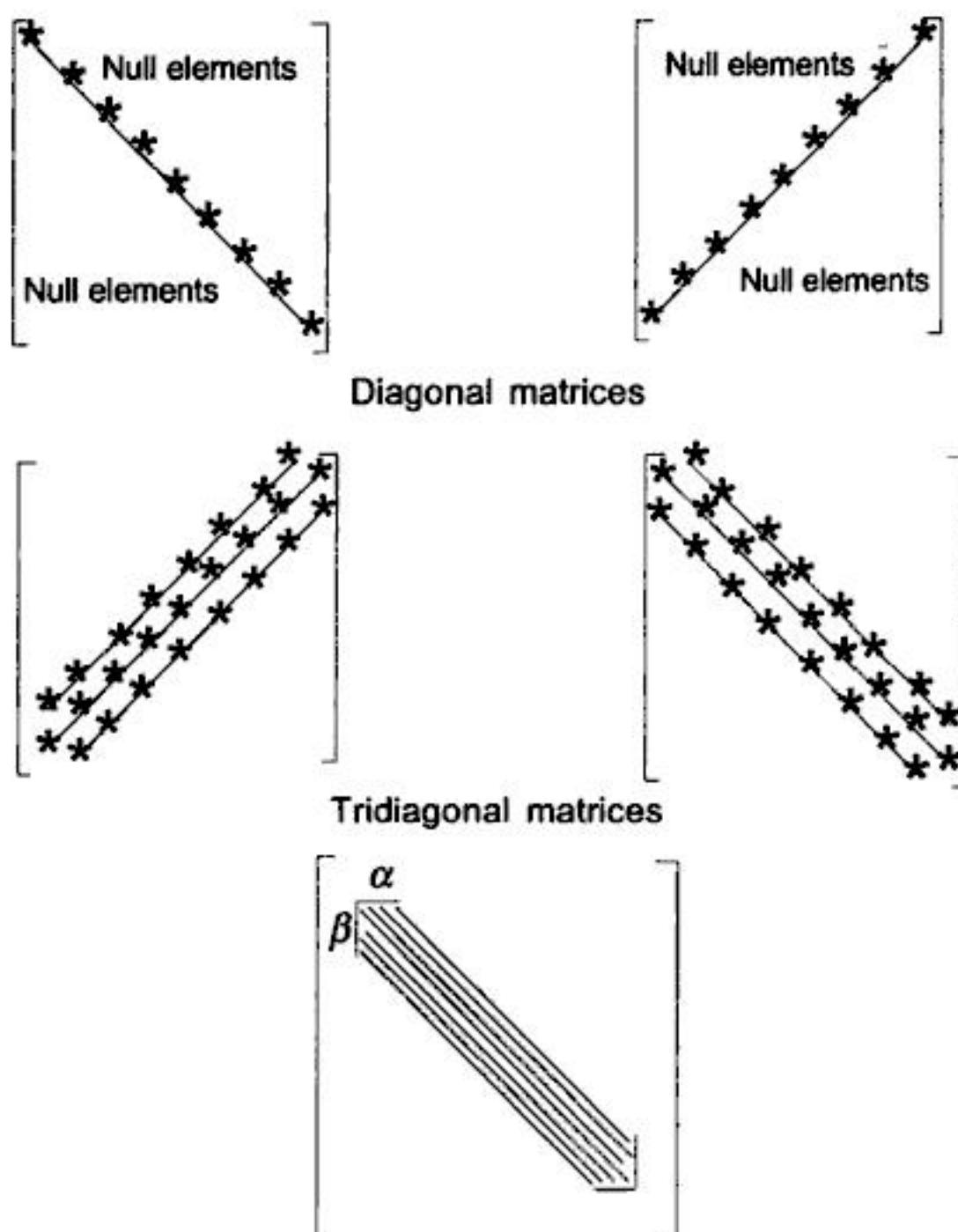
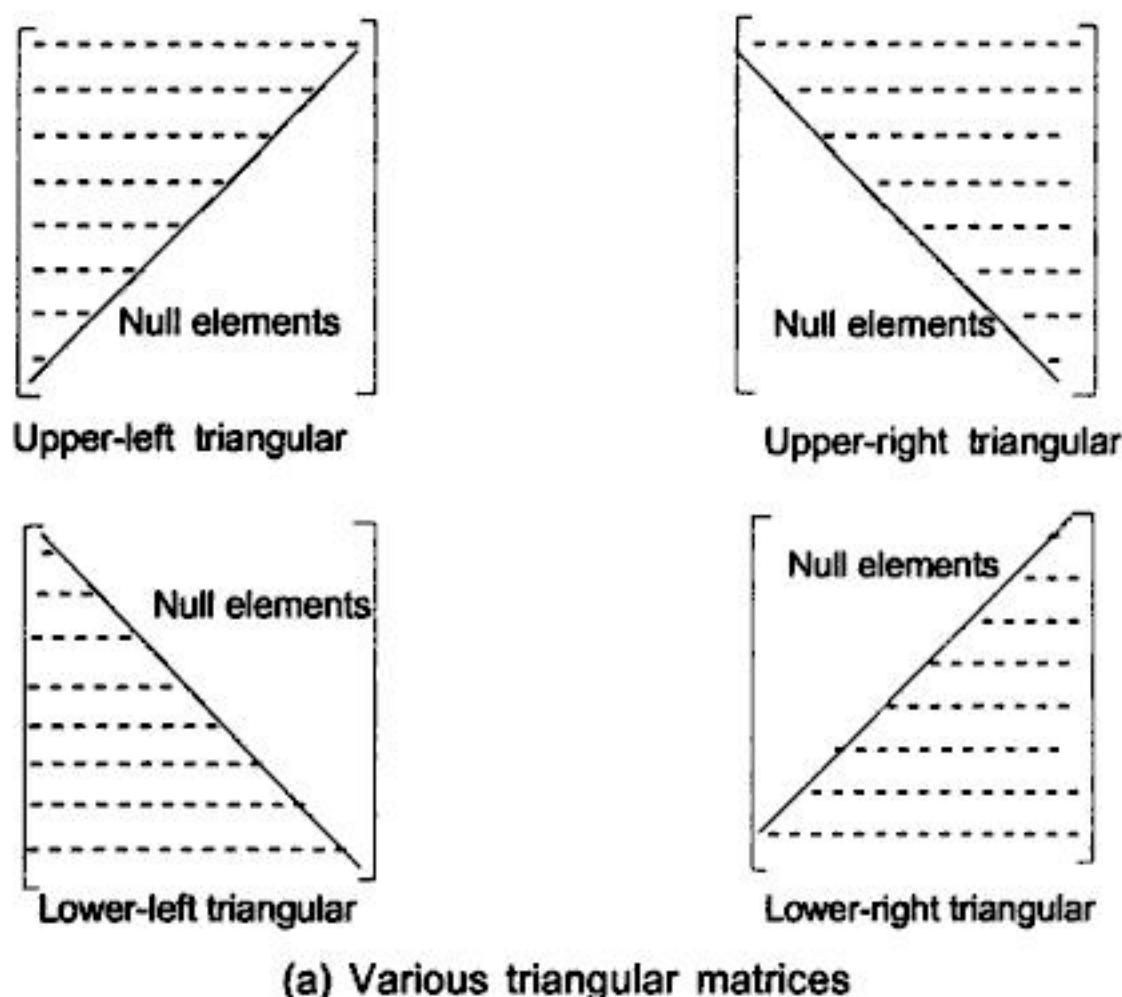


Figures 2.10(a) and 2.10(b) show the general views of the above mentioned sparse matrices. Let us discuss the memory representation some of the above forms.

#### **Memory representation of lower-triangular matrix**

Consider the following lower-triangular matrix:

$$\begin{bmatrix} a_{11} & & & & \\ a_{21} & a_{22} & & & \\ a_{31} & a_{32} & a_{33} & & \\ \vdots & \vdots & \vdots & & \\ a_{n1} & a_{n2} & a_{n3} & \dots & a_{nn} \end{bmatrix}_{n \times n}$$



$\alpha\beta$ -band matrix: non-null elements are only on  $\alpha$ -upper diagonal and  $\beta$ -lower diagonal matrix

(b) Various diagonal matrices

Fig. 2.10 Different symmetric sparse matrices.

**Row-major order.** According to row-major order, the address of any element  $a_{ij}$ ,  $1 \leq i, j \leq n$  can be obtained as

$$\begin{aligned}\text{Address } (a_{ij}) &= \text{Number of elements up to } a_{ij} \text{ element} \\ &= \text{Total number of elements in first } i - 1 \text{ rows} \\ &\quad + \text{Number of elements up to } j\text{-th column in the } i\text{-th row} \\ &= 1 + 2 + 3 + \dots + (i - 1) + j \\ &= \frac{i(i - 1)}{2} + j\end{aligned}$$

If the starting location of the first element, that is, of  $a_{11}$  is  $M$ , then the address of  $a_{ij}$ ,  $1 \leq i, j \leq n$  will be

$$\text{Address } (a_{ij}) = M + \frac{i(i - 1)}{2} + j - 1$$

**Column-major order.** According to column-major order, the address of any element  $a_{ij}$ ,  $1 \leq i, j \leq n$  can be obtained as

$$\begin{aligned}\text{Address } (a_{ij}) &= \text{Number of elements up to } a_{ij} \text{ element} \\ &= \text{Total number of elements in first } j - 1 \text{ columns} \\ &\quad + \text{Number of elements up to } i\text{-th row in the } j\text{-th column} \\ &= [n + (n - 1) + (n - 2) + \dots + (n - j + 2)] + (i - j + 1) \\ &= \{n \times (j - 1) - [1 + 2 + 3 + \dots + (j - 2) + (j - 1)] + i\} \\ &= n \times (j - 1) - \frac{j(j - 1)}{2} + i \\ &= (j - 1) \times \left(n - \frac{j}{2}\right) + i\end{aligned}$$

If the starting location of the first element (that is, of  $a_{11}$ ) is  $M$  then the address of  $a_{ij}$ ,  $1 \leq i, j \leq n$  will be

$$\text{Address } (a_{ij}) = M + (j - 1) \times \left(n - \frac{j}{2}\right) + i - 1$$

### Memory representation of upper-triangular matrix

As an another example, consider the following form of a upper-triangular matrix:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ & a_{22} & a_{23} & \dots & a_{2n} \\ & & a_{33} & \dots & a_{3n} \\ & & & \vdots & \\ & & & & a_{nn} \end{bmatrix}_{n \times n}$$

**Row-major order.** According to row-major order, the address of any element  $a_{ij}$ ,  $1 \leq i, j \leq n$  can be obtained as

$$\begin{aligned}
 \text{Address } (a_{ij}) &= \text{Number of elements up to } a_{ij} \text{ element} \\
 &= \text{Total number of elements in first } (i - 1) \text{ rows} \\
 &\quad + \text{Number of elements up to } j\text{-th column in the } i\text{-th row} \\
 &= n + (n - 1) + (n - 2) + \dots + (n - i + 2) + (j - i + 1) \\
 &= n \times (i - 1) - [1 + 2 + 3 + \dots + (i - 2) + (i - 1)] + j \\
 &= n \times (i - 1) - \frac{i(i - 1)}{2} + i \\
 &= (i - 1) \times \left(n - \frac{i}{2}\right) + j
 \end{aligned}$$

If the starting location of the first element, i.e. of  $a_{11}$  is  $M$ , then the address of  $a_{ij}$ ,  $1 \leq i, j \leq n$  will be

$$\text{Address } (a_{ij}) = M + (i - 1) \times \left(n - \frac{i}{2}\right) + j - 1$$

**Column-major order.** According to column-major order, the address of any element  $a_{ij}$ ,  $1 \leq i, j \leq n$  can be obtained as

$$\begin{aligned}
 \text{Address } (a_{ij}) &= \text{Number of elements up to } a_{ij} \text{ element} \\
 &= \text{Total number of elements in first } (j - 1) \text{ columns} \\
 &\quad + \text{Number of elements up to } i\text{-th row in the } j\text{-th column} \\
 &= [1 + 2 + 3 + \dots + (j - 1)] + i \\
 &= \frac{j(j - 1)}{2} + i
 \end{aligned}$$

If the starting location of the first element, i.e. of  $a_{11}$  is  $M$ , then the address of  $a_{ij}$ ,  $1 \leq i, j \leq n$  will be

$$\text{Address } (a_{ij}) = M + \frac{j(j - 1)}{2} + i - 1$$

### Assignment 2.9

- (a) How many elements are there in a triangular sparse matrix of order  $n \times n$ .
- (b) Obtain the indexing formula for lower-right and upper-left triangular matrices using row-major and column-major order, consider the cases of
  - (i) square matrix of order  $n \times n$
  - (ii) non-square matrix of order  $m \times n$ ,  $m \neq n$

### Memory representation of diagonal matrix

In the sparse matrices having the elements only on diagonal following points are evident:

Number of elements in a  $n \times n$  square diagonal matrix =  $n$ .

Any element  $a_{ij}$  can be referred in memory using the formula

$$\text{Address } (a_{ij}) = i[\text{or } j]$$

One can easily verify that the above formula is same in both row-major and column-major order.

### **Memory representation of tridiagonal matrix**

Let us consider the following tridiagonal matrix:

$$\begin{bmatrix} a_{11} & a_{12} & & & \\ a_{21} & a_{22} & a_{23} & & \\ & a_{32} & a_{33} & a_{34} & \\ & a_{43} & a_{44} & a_{45} & \\ & & \vdots & & \\ & & & a_{(n-1)(n-2)} & a_{(n-1)(n-1)} & a_{(n-1)n} \\ & & & & a_{n(n-1)} & a_{nn} \end{bmatrix}$$

**Row-major order.** According to row-major order, the address of any element  $a_{ij}$ ,  $1 \leq i, j \leq n$  can be obtained as

$$\begin{aligned} \text{Address } (a_{ij}) &= \text{Number of elements up to } a_{ij} \text{ element} \\ &= \text{Total number of elements in first } (i - 1) \text{ rows} \\ &\quad + \text{Number of elements up to } j\text{-th column in the } i\text{-th row} \\ &= \{2 + [3 + 3 + \dots + \text{up to } (i - 2) \text{ terms}]\} + (j - i + 2) \\ &= 2 + (i - 2) \times 3 + j - (i - 2) \\ &= 2 + 2 \times (i - 2) + j \end{aligned}$$

If the starting location of the first element, i.e. of  $a_{11}$  is  $M$  then the address of  $a_{ij}$ ,  $1 \leq i, j \leq n$  will be

$$\text{Address } (a_{ij}) = M + 2 \times (i - 2) + j + 1$$

**Column-major order.** According to column-major order, the address of any element  $a_{ij}$ ,  $1 \leq i, j \leq n$  can be obtained as

$$\begin{aligned} \text{Address } (a_{ij}) &= \text{Number of elements up to } a_{ij} \text{ element} \\ &= \text{Total number of elements in first } (j - 1) \text{ columns} \\ &\quad + \text{Number of elements up to } i\text{-th row in the } j\text{-th column} \\ &= \{2 + [3 + 3 + \dots + \text{up to } (j - 2) \text{ terms}]\} + (i - j + 2) \\ &= 2 + (j - 2) \times 3 + i - (j - 2) \\ &= 2 + 2 \times (j - 2) + i \end{aligned}$$

If the starting location of the first element, i.e. of  $a_{11}$  is  $M$  then the address of  $a_{ij}$ ,  $1 \leq i, j \leq n$  will be

$$\text{Address } (a_{ij}) = M + 2 \times (j - 2) + i + 1$$

**Note:** The formulas for row-major/column-major order is symmetric and one can be obtained from the other by interchanging  $i$  and  $j$ .

#### Assignment 2.10

- (a) Obtain the number of elements in a  $(n \times n)$  square tridiagonal matrix.
- (b) What will be the indexing formula for other form of tridiagonal matrix?

#### Memory representation for $\alpha\beta$ -band matrix

Let us consider a matrix as in Figure 2.11:

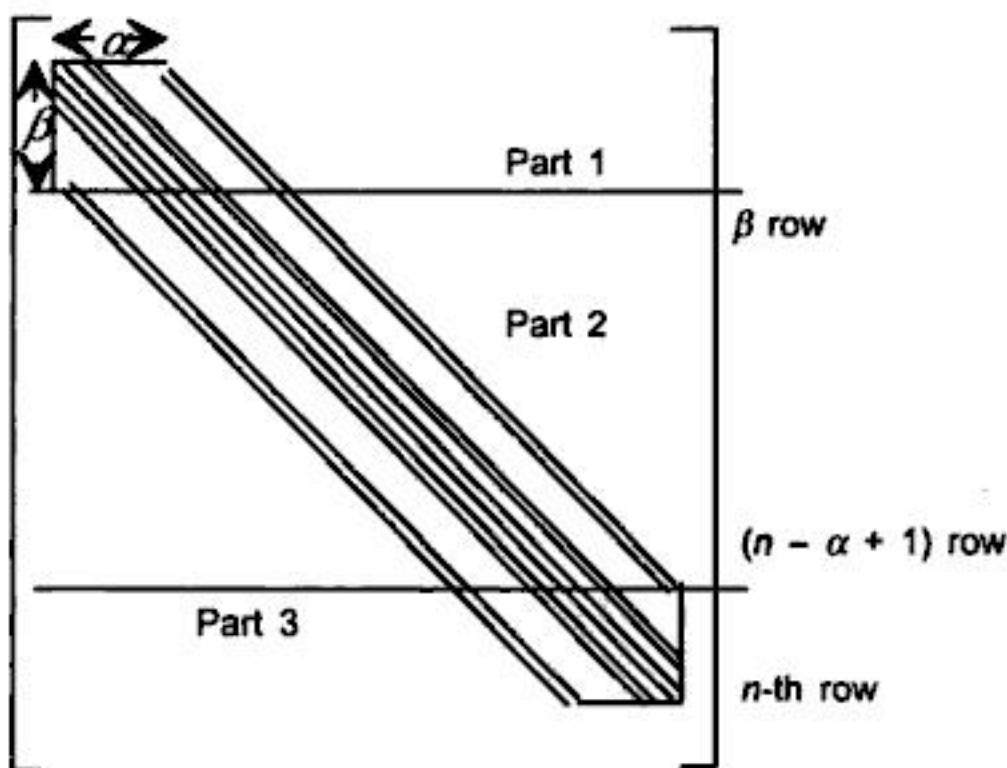


Fig. 2.11  $\alpha\beta$ -band matrix.

Note that there are  $(\alpha - 1)$  and  $(\beta - 1)$  sub-diagonals above and below the main diagonal respectively. As there are three possible arrangements so elements in each arrangement can be referred by three indexing formulae corresponding to Part 1, Part 2, and Part 3 as shown in the Figure 2.11.

Considering the row-major ordering for the memory allocation, the indexing formula is explained as below:

**Case 1:**  $1 \leq i \leq \beta$

$$\begin{aligned} \text{Address } (a_{ij}) &= \text{Number of elements in first } (i-1)\text{-th rows} \\ &\quad + \text{Number of elements in } i\text{-th row up to } j\text{-th column} \\ &= \alpha + (\alpha + 1) + (\alpha + 2) + \dots + (\alpha + i - 2) + j \\ &= \alpha \times (i - 1) + [1 + 2 + 3 + \dots + (i - 2)] + j \\ &= \alpha \times (i - 1) + \frac{(i - 1)(i - 2)}{2} + j \end{aligned}$$

**Case 2:**  $\beta < i \leq n - \alpha + 1$

$$\begin{aligned}
 \text{Address } (a_{ij}) &= \text{Number of elements in first } \beta \text{ rows} \\
 &\quad + \text{Number of elements between } (\beta + 1)\text{-th row and } (i - 1)\text{-th row} \\
 &\quad + \text{Number of elements in } i\text{-th row} \\
 &= \alpha + (\alpha + 1) + (\alpha + 2) + \dots + (\alpha + \beta - 1) + (\alpha + \beta - 1) \times (i - \beta - 1) \\
 &\quad + j - i + \beta \\
 &= \alpha\beta + \frac{\beta(\beta - 1)}{2} + (\alpha + \beta - 1)(i - \beta - 1) + j - i + \beta
 \end{aligned}$$

**Case 3:**  $n - \alpha + 1 < i$

$$\begin{aligned}
 \text{Address } (a_{ij}) &= \text{Number of elements in first } (n - \alpha + 1) \text{ rows} \\
 &\quad + \text{Number of elements after } (n - \alpha + 1)\text{-th row and up to } (i - 1)\text{-th row} \\
 &\quad + \text{Number of elements in } i\text{-th row and up to } j\text{-th column} \\
 &= \alpha\beta + \frac{\beta(\beta - 1)}{2} + (\alpha + \beta - 1)(n - \alpha - \beta + 1) + (\alpha + \beta - 2) \\
 &\quad + (\alpha + \beta - 3) + \dots + \{\alpha + \beta - [(i - 1) - (n - \alpha + 1)]\} + j - i + \alpha \\
 &= \alpha\beta + \frac{\beta(\beta - 1)}{2} + (\alpha + \beta - 1)(n - \alpha - \beta + 1) + (\alpha + \beta)(i - n + \alpha - 1) \\
 &\quad - \{1 + 2 + 3 + \dots + [(i - 1) - (n - \alpha + 1)]\} + 1 \\
 &= \alpha\beta + \frac{\beta(\beta - 1)}{2} + (\alpha + \beta - 1)(n - \alpha - \beta + 1) + (\alpha + \beta)(i - n + \alpha - 1) \\
 &\quad - \frac{(i - n + \alpha - 1) \times (i - n + \alpha - 2)}{2} + 1
 \end{aligned}$$

#### Assignment 2.11

- Find the number of elements in the  $\alpha\beta$ -band matrix.
- Show that the indexing formula for  $\alpha\beta$ -band matrix is a general formula of tridiagonal matrix.
- Obtain the indexing formula for column-major ordering of  $\alpha\beta$ -band matrix.

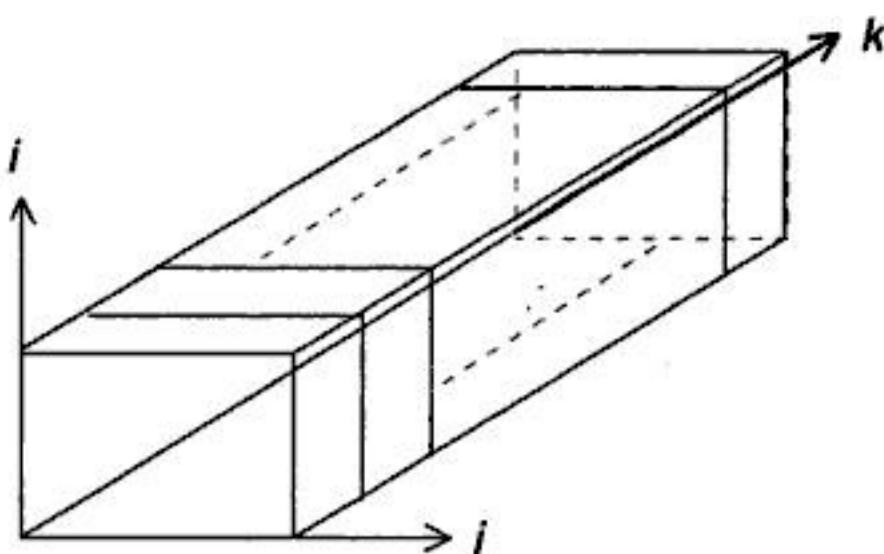
### 2.4.3 Three-dimensional and $n$ -dimensional Arrays

So far we have discussed one-dimensional and two-dimensional arrays. In some advanced applications, more than two-dimensional arrays are also used. Here we will discuss multidimensional arrays from the theoretical point of view.

#### Three-dimensional (3-D) array

A three-dimensional array will look as shown in Figure 2.12.

A three-dimensional array can be compared with a book whereas two-dimensional and one-dimensional arrays can be compared with a page and a line respectively. Here, three major dimensions can be termed as row, column and page. Let, for a three-dimensional array, the following specifications are known:



**Fig. 2.12** A three-dimensional array.

Number of rows =  $x$  (number of elements in a column)

Number of columns =  $y$  (number of elements in a row) and

Number of pages =  $z$

Assume that  $a_{ijk}$  is an element in  $i$ -th row,  $j$ -th column and  $k$ -th page. Now, we are interested to find the memory location of element  $a_{ijk}$  in the array.

Storing a 3-D array means, storing the pages one by one. Again storing a page is same as storing a 2-D array. Thus, if the elements in a page stored in row-major order then we term that 3-D array is also in row-major order. The following formula is taking into consideration of row-major order of a 3-D array:

$$\begin{aligned} \text{Address } (a_{ijk}) &= \text{Number of elements in first } (k - 1) \text{ pages} \\ &\quad + \text{Number of elements in } k\text{-th page up to } (i - 1) \text{ rows} \\ &\quad + \text{Number of elements in } k\text{-th page, in } i\text{-th row up to } j\text{-th column} \\ &= xy(k - 1) + (i - 1)y + j \end{aligned}$$

Instead of index starting from  $l$  for all indices (as assumed in the above mentioned formula), if we assume that  $i$  changes between  $l_x$  to  $u_x$ ,  $j$  changes between  $l_y$  to  $u_y$  and  $k$  changes between  $l_z$  to  $u_z$ . So that

$$x = u_x - l_x + 1, \quad y = u_y - l_y + 1 \quad \text{and} \quad z = u_z - l_z + 1$$

Also assuming the word size of each element is  $w$  instead of  $l$  then the above indexing formula for 3-D array can be restated in general form as:

$$\text{Address } (a_{ijk}) = M + [xy(k - l_z) + (i - l_x)y + (j - l_y)] \times w$$

where,  $M$  denotes the base address of the array.

### ***n*-dimensional array**

From the representation of 2-D and 3-D, we can extend our idea to store an  $n$ -dimensional array. Here, to identify an element, we need  $n$  indices,  $i_1, i_2, \dots, i_n$ .

Let  $x_j$  be the number of elements in  $j$ -th dimension and range of index for  $i_j$ , say, varies between  $l_j \dots u_j$ , where  $1 \leq j \leq n$ . So, the total number of elements in the array is

$$\prod_{j=1}^n x_j = \prod_{j=1}^n (u_j - l_j + 1), \quad 1 \leq j \leq n$$

Storing this  $n$ -dimensional array in memory, any element can be referenced using the formula:

$$\begin{aligned}\text{Address } (a_{i_1 i_2 \dots i_n}) &= (i_n - l_n) x_{n-1} x_{n-2} x_{n-3} \dots x_3 x_2 x_1 + (i_{n-1} - l_{n-1}) x_{n-2} x_{n-3} \dots x_3 x_2 x_1 \\ &\quad + \dots + (i_2 - l_2) x_1 + (i_1 - l_1)\end{aligned}$$

in row-major order and

$$\begin{aligned}\text{Address } (a_{i_1 i_2 \dots i_n}) &= (i_1 - l_1) x_2 x_3 \dots x_{n-2} x_{n-1} x_n + (i_2 - l_2) x_3 x_4 \dots x_{n-2} x_{n-1} x_n \\ &\quad + (i_{n-1} - l_{n-1}) x_{n-2} x_{n-1} x_n + (i_n - l_n)\end{aligned}$$

in column-major order.

## 2.5 POINTER ARRAYS

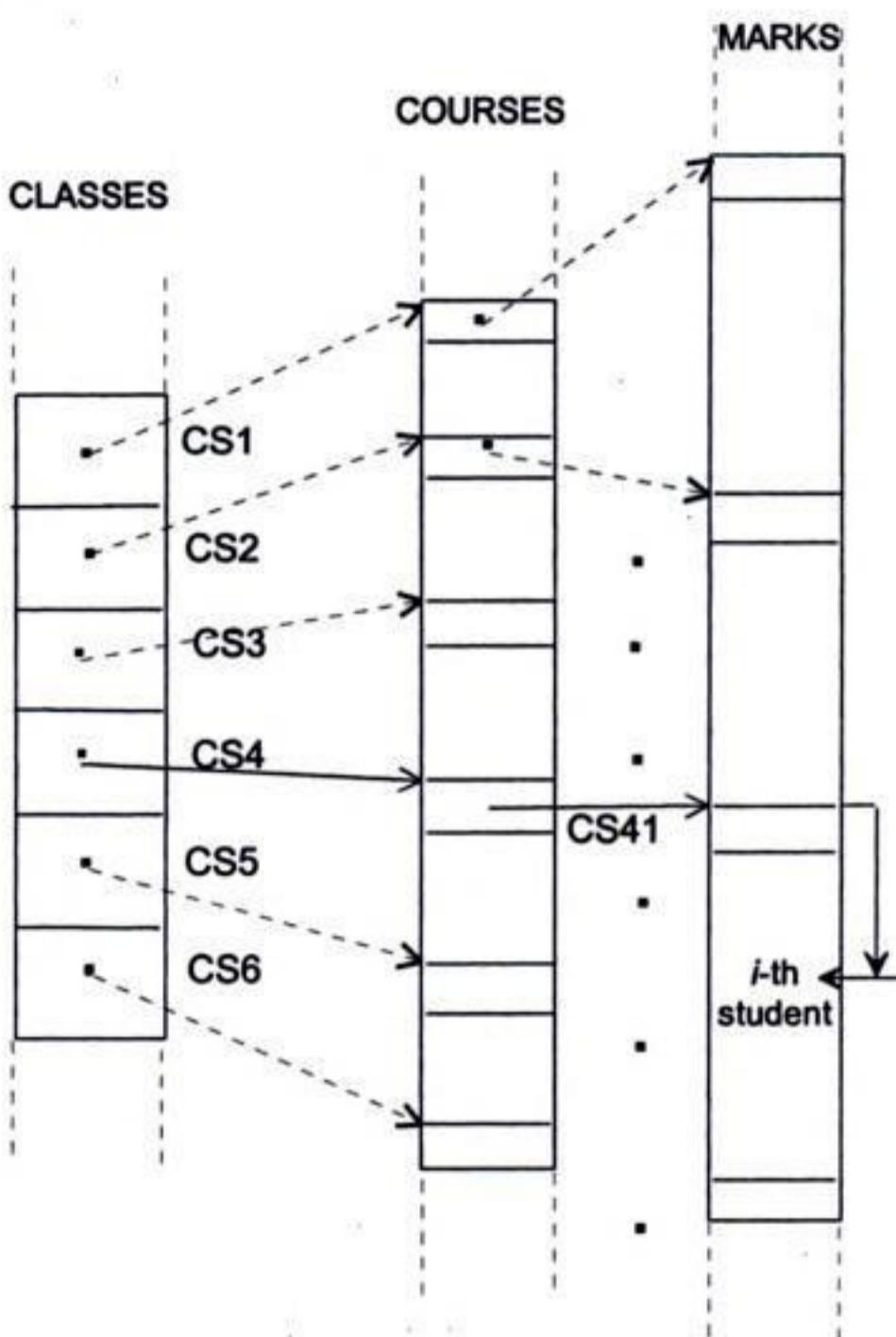
Some integer values, real numbers, string of characters, etc., are simple elements that arrays may contain. In addition to these, another kind of array is also well known in the theory of computer science which is in use to store address of some memory variables or the address of another arrays. Address of memory variable or array is known as *pointer* and an array containing pointers as its elements is known as *pointer array*. Let us consider the following example to illustrate a pointer array.

**Example:** Suppose, we want to store the marks of all students of Computer Science and Engineering (CSE) department for a year. There are six classes, say, CS-1, CS-2, ..., CS-6. And for each class, there are at most 5 courses; for example, course of  $i$ -th class can be denoted as  $\text{CS}_{ij}$  (where  $j$  varies from 1 to 5). We will assume that there is at most 30 students in each class.

We should maintain an array (one-dimensional) of size  $6 \times 5 \times 30 (= 900)$  to store the marks. We will use pointer arrays to keep track of the mark of  $i$ -th year student in a course  $j(s_{ij})$  and the starting location where the marks of the course  $\text{CS}_i$  begin. The idea is illustrated in Figure 2.13.

Here, we have maintained 3 arrays, viz. CLASSES, COURSES, and MARKS having sizes 6, 30 and 900 respectively. The MARKS array stores the marks obtained by the various students in different courses. The COURSES array stores the starting location for the marks of different courses. In Figure 2.13, starting location for all the marks of CS41 course is shown by the pointed arrow. The CLASSES array points the starting location (at COURSES) for various courses under a class. For example, the location of CS41 (the first course) under the 4-th year class (CS4) is shown by the pointed arrow.

Thus, if we want to retrieve the marks obtained by the  $k$ -th student in  $j$ -th course of class  $\text{CS}_i$ , then we have to search first the array CLASSES to obtain the starting location of all courses under class  $\text{CS}_i$ . After knowing the starting location of class  $\text{CS}_i$  we shall move to that location in array COURSES. The information for  $j$ -th course can be obtained by shifting sequentially  $j$  steps in the array COURSES, where the starting location (at MARKS) for the  $j$ -th course will be obtained. With that address, one can move to the referred location at MARKS array and again sequentially moving  $k$  steps the desired marks will be retrieved.



**Fig. 2.13 Pointer array.**

#### Assignment 2.12

- Using only a single array of marks having size 900, give an idea how the same information can be maintained. You must consider the case that there may be less than 5 courses in a class or may be less than 30 students in a course.
- Other than one-dimensional array, 2-D or 3-D can be employed? How?

#### PROBLEMS TO PONDER

**2.1** Following are the three known searching methods when data are stored in an array.

- Sequential search
- Binary search
- Interpolation search
- Address calculation search (closed hashing)
  - Write generic implementation for the above searching methods so that they can be used with any type of data stored in the array.
  - Obtain a time comparison of the above methods and then conclude which is/are to be taken as best searching methods.

**2.2** Following are the important sorting algorithms which can be applied on data stored in an array.

- (a) Bubble sort
- (b) Selection sort
- (c) Insertion sort
- (d) Shell sort
- (e) Quick sort

- (i) Store a set of data into a *generic array*. Obtain a sorted order of data either in ascending or descending order using the above sorting methods.
- (ii) For a given set of data, apply the above sorting methods and then note the time required for sorting in each case. Then conclude which is/are the best sorting method(s).

*Note:* An array is generic means it can hold any data irrespective of its type.

**2.3** A multiplication table is a matrix of order  $m \times n$  where an entry in  $i$ -th row and  $j$ -th column is the product  $x \times y$ , where  $x$  and  $y$  are the numbers in  $i$ -th row and  $j$ -th column respectively. Figure 2.14 shows a multiplication table from 3 to 7.

|   | 3  | 4  | 5  | 6  | 7  |
|---|----|----|----|----|----|
| 3 | 9  | 12 | 15 | 18 | 21 |
| 4 | 12 | 16 | 20 | 24 | 28 |
| 5 | 15 | 20 | 25 | 30 | 35 |
| 6 | 18 | 24 | 30 | 36 | 42 |
| 7 | 21 | 28 | 35 | 42 | 49 |

**Fig. 2.14** A multiplication table from 3 to 7.

Write a program to display multiplication table from  $x$  to  $y$ .

**2.4** A magic square is a square matrix of integers such that the sum of every row, the sum of every column and sum of each of the diagonal are equal. Such a magic square is shown in Figure 2.15.

|    |    |    |    |
|----|----|----|----|
| 4  | 15 | 14 | 1  |
| 9  | 6  | 7  | 12 |
| 5  | 10 | 11 | 8  |
| 16 | 3  | 2  | 13 |

**Fig. 2.15** A magic square with SUM = 34.

- (a) Write a program to read a set of integers for a square matrix and then decide whether the matrix represents a magic square or not.
- (b) Write a game program as follows:
  - (i) Read the size of the square matrix,  $N \times N$

- (ii) Display a square matrix (now it is blank) of  $N \times N$
- (iii) Allow the player to insert data into the matrix as displayed (you should give a chance to the user to confirm the entry and to alter the previous entries, if desire).
- (iv) After the completion of all the entries from the player, count the score as follows:

Score = 0 (zero) if it is not a magic square. Otherwise score =  $T + P*100$ , where  $T$  is the time required in second and  $P$  is the number of alteration of entries.

The player's performance will be judged by the minimum score achieved, other than zero.

## REFERENCES

1. Ellis Horowitz and Sartaj Sahni, *Fundamentals of Data Structures*, Computer Science Press, Rockville, Maryland, 1985.
2. Gotlieb, C.C. and Gotlieb, L.R., *Data Types and Structures*, Prentice Hall, Englewood Cliffs, New Jersey, 1986.
3. Jean Paul Tremblay and Paul G. Sorenson, *An Introduction to Data Structures with Applications*, McGraw-Hill, New York, 1987.
4. Robert L. Kruse, Bruce P. Leung and Clovis L. Tondo, *Data Structures and Program Design in C*, Prentice Hall, Englewood Cliffs, New Jersey, 1995.

# 3 *Linked Lists*

Array is a data structure where elements are stored in consecutive memory locations. In order to occupy the adjacent space, block of memory that is required for the array should be allocated before hand. Once memory is located it cannot be extended any more. This is why array is known as *static data structure*. In contrast to this, linked list is called *dynamic data structure* where amount of memory required can be varied during its use. In linked list, adjacency between the elements are maintained by means of *links* or *pointers*. A link or pointer actually is the address (memory location) of the subsequent element. Thus, in a linked list, data (actual content) and link (to point to the next data) both are required to be maintained. An element in a linked list is specially termed as *node*, which can be viewed as shown in Figure 3.1. A node consists of two fields: DATA (to store the actual information) and LINK (to point to the next node).

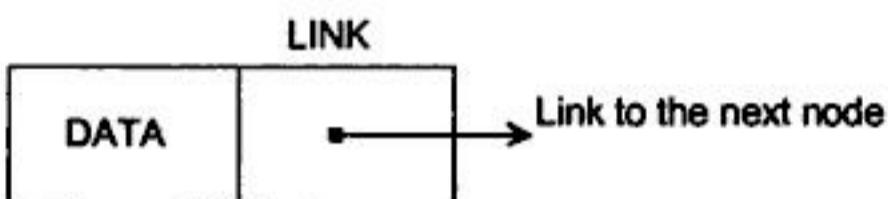


Fig. 3.1 Node: an element in a linked list.

## 3.1 DEFINITION

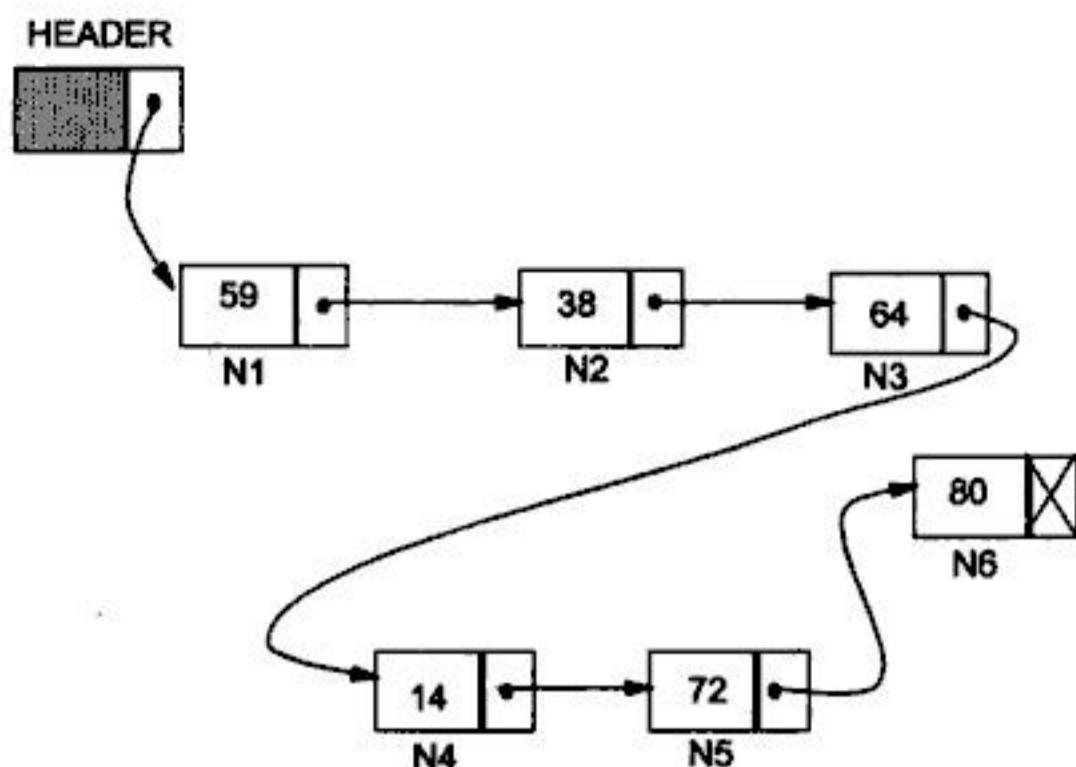
A *linked list* is an ordered collection of finite, homogeneous data elements called *nodes* where the linear order is maintained by means of links or pointers.

Depending on the requirements the pointers are maintained, and accordingly linked list can be classified into three major groups: single linked list, circular linked list, and double linked list.

## 3.2 SINGLE LINKED LIST

In a single linked list each node contains only one link which points the subsequent node in the list. Figure 3.2 shows a linked list with 6 nodes.

Here, N<sub>1</sub>, N<sub>2</sub>, ..., N<sub>6</sub> are the constituent nodes in the list. HEADER is an empty node (having data content NULL) and only used to store a pointer to the first node N<sub>1</sub>. Thus, if one knows the address of the HEADER node from the link field of this node, next node can be traced and so on. This means that starting from the first node one can reach to the last node whose link field does not contain any address rather a null value. Note that in a single linked list one can move from left to right only; this is why a single linked list is also alternatively termed as *one way* list.



**Fig. 3.2** A single linked list with 6 nodes.

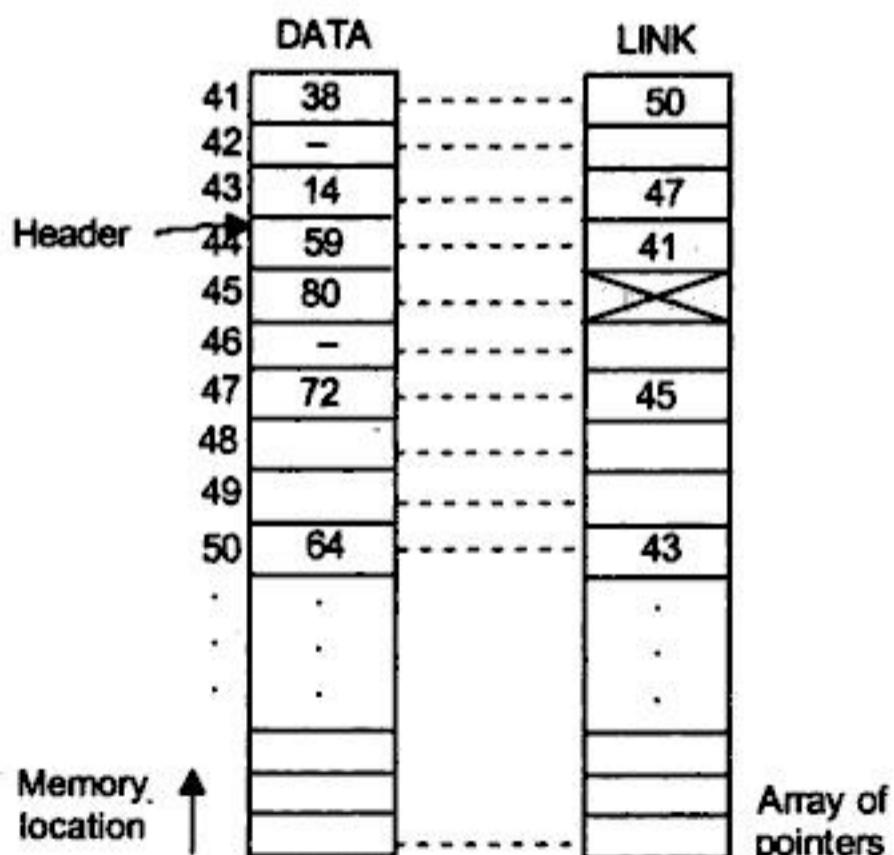
### 3.2.1 Representation of a Linked List in Memory

There are two ways to represent a linked list in memory:

1. Static representation using array
2. Dynamic representation using free pool of storage

#### **Static representation**

Static representation of a single linked list maintains two arrays: one array for data and other for links. Static representation for the linked list as shown in the Figure 3.2 is given in Figure 3.3.



**Fig. 3.3** Static representation of a single linked list using arrays.

Two parallel arrays of equal size are allocated which should be sufficient to store the entire linked list. Nevertheless this contradicts the idea of linked list (that is non-contiguous location of elements). But in some programming languages, for example, ALGOL, FORTRAN, BASIC, etc., such a representation is the only representation to manage a linked list.

### **Dynamic representation**

The efficient way of representing a linked list is using free pool of storage. In this method, there is a *memory bank* (which is nothing but a collection of free memory spaces), and a *memory manager* (a program, in fact). During the creation of linked list, whenever a node is required the request is placed to the memory manager; memory manager will then search the memory bank for the block requested and if found grants a desired block to the caller. Again, there is also another program called *garbage collector*, it plays whenever a node is no more in use; it returns the unused node to the memory bank. It may be noted that memory bank is also a list of memory space that is available to a programmer. Such a memory management is known as *dynamic* memory management. Dynamic representation of linked list uses the dynamic memory management policy.

The mechanism of dynamic representation of single linked list is illustrated as shown in Figures 3.4(a) and 3.4(b). A list of available memory space is there whose pointer is stored in AVAIL. For a request of a node, the list AVAIL is searched for the block of right size. If AVAIL is null or the block of desired size is not found memory manager will return a message accordingly. Suppose, the block is found and let it be XY. Then memory manager will return the pointer of XY to the caller in a temporary buffer, say NEW. The newly availed node XY then can be inserted at any position in the linked list by changing the pointers of the concerned nodes. In Figure 3.4(a), the node XY is inserted at the end and change of pointers are shown by the dotted arrows. Figure 3.4(b) explains the mechanism of how a node is to be returned to the memory bank.

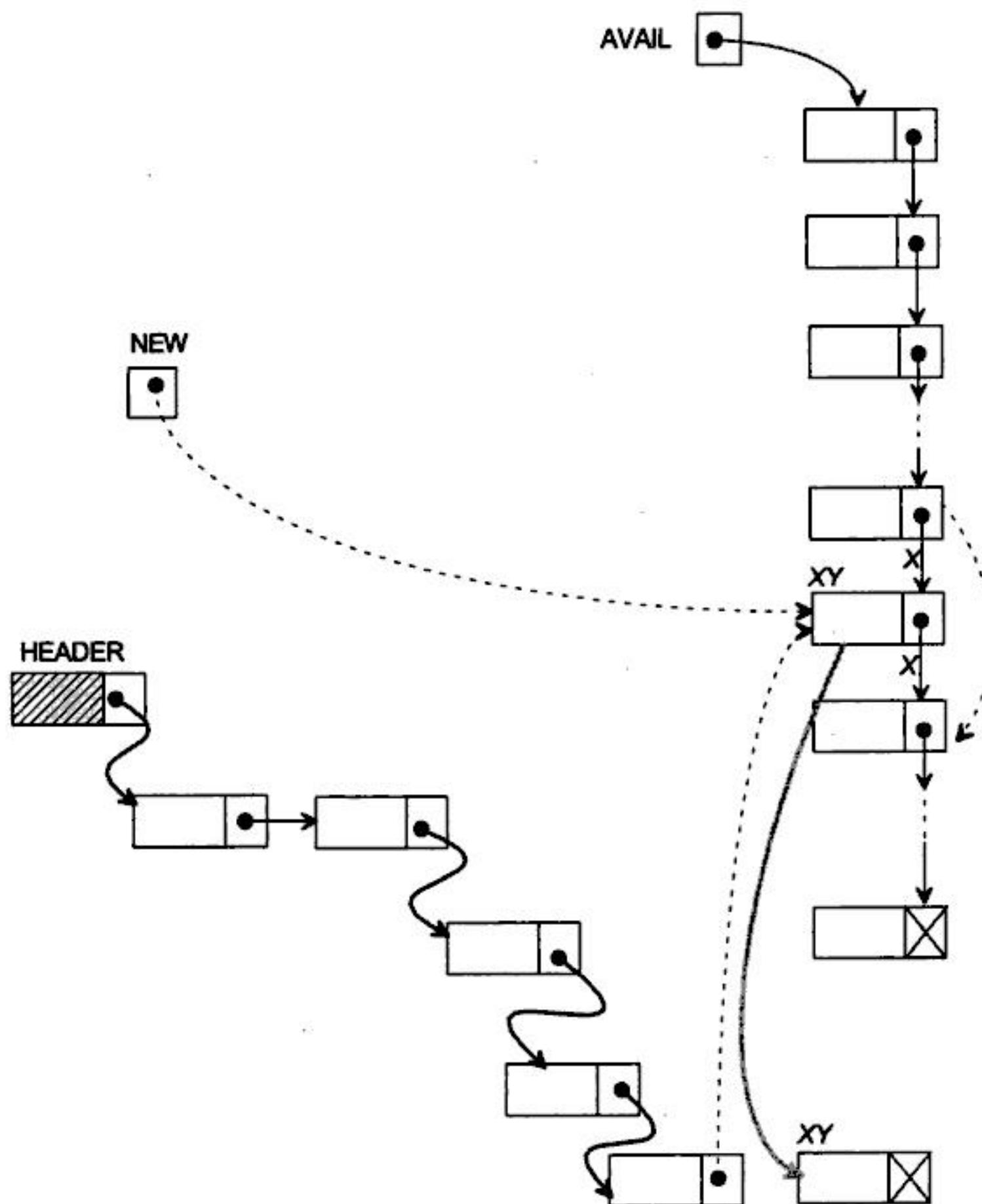
The pointers that are required to be manipulated while returning a node is shown by dotted arrows. Note that such allocation or deallocation is carried out by changing the pointers only.

### **3.2.2 Operations on a Single Linked List**

Possible operations on a single linked list are listed as below:

- *Traversing* a list
- *Insertion* of a node into a list
- *Deletion* of a node from a list
- *Copy* a linked list to make a duplicate
- *Merging* two linked lists into a larger list
- *Searching* for an element in a list.

We will assume the following convention in our present discussion: say X is a node. The values in the DATA field and LINK field will be denoted as X.DATA and X.LINK respectively. We will write NULL to imply that value in the field like DATA, LINK is nil.



**Fig. 3.4(a)** Allocation of a node from memory bank to a linked list.

### **Traversing a single linked list**

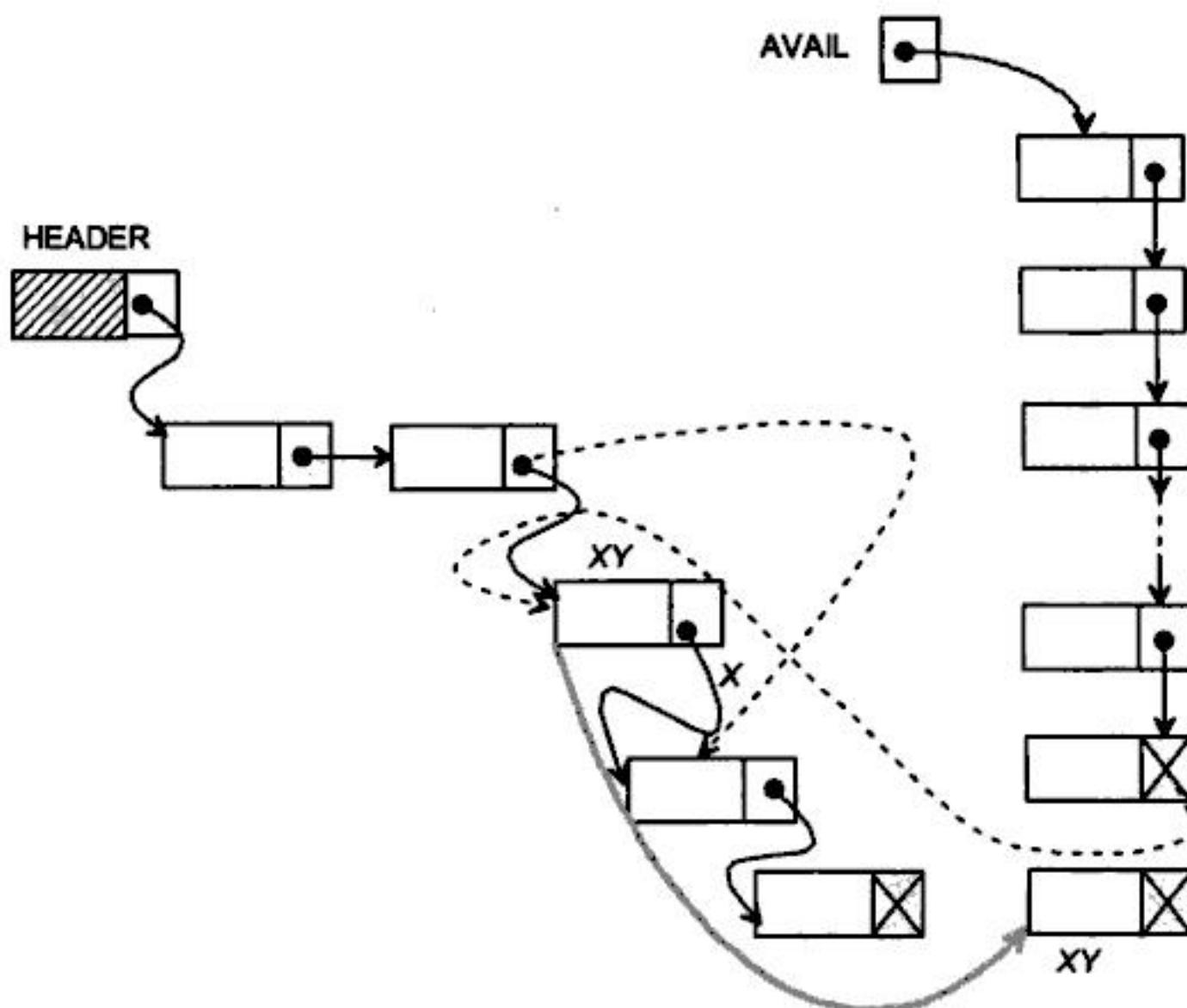
To traverse a single linked list we mean to visit every node in the list starting from the first node to the last node. Following is the algorithm TRAVERSE\_SL for the same.

#### **Algorithm TRAVERSE\_SL (HEADER)**

**Input:** HEADER is the pointer to the header node.

**Output:** According to the PROCESS( )

**Data structures:** A single linked list whose address of the starting node is known from HEADER.



**Fig. 3.4(b)** Returning a node from list to memory bank.

**Steps:**

- |                                         |                                                 |
|-----------------------------------------|-------------------------------------------------|
| 1. <code>ptr = HEADER.LINK</code>       | //ptr is to store the pointer to a current node |
| 2. While ( <code>ptr ≠ NULL</code> ) do | //Continue till the last node                   |
| 1. <code>PROCESS(ptr)</code>            | //Perform PROCESS( ) on the current node        |
| 2. <code>ptr = ptr.LINK</code>          | //Move to the next node                         |
| 3. EndWhile                             |                                                 |
| 4. Stop                                 |                                                 |

*Note:* A simple operation of `PROCESS( )` may be thought as to print the data content of a node.

#### **Insertion of a node into a single linked list**

There are various positions where a node can be inserted:

- Insert at front (as a first element)
- Insert at end (as a last element)
- Insert at any position.

Before we discuss these insertions, let us assume a procedure `GETNODE(NODE)` to get a pointer of a memory block which suits the type `NODE`. The procedure may be defined as below:

**Procedure GETNODE(NODE)**

1. If (AVAIL = NULL) //AVAIL is the pointer to the pool of free storage
  1. Return (NULL)
  2. Print "Insufficient memory: Unable to allocate memory"
2. Else //Sufficient memory is available
  1. ptr = AVAIL
  2. While (SIZEOF(ptr) ≠ SIZEOF(NODE)) and (ptr ≠ NULL)
    - //Till the desired block is not found or search reaches //at the end of the pool
    1. ptr1 = ptr //To keep the track of the previous block
    2. ptr = ptr.LINK //Move to the next block
  3. EndWhile
  4. If (SIZEOF(ptr) = SIZEOF(NODE)) //Memory block of right size is found
    1. ptr1.LINK = ptr.LINK //Update the AVAIL List
    2. Return(ptr)
  5. Else
    1. Print "The memory block is too large to fit"
    2. Return(NULL)
  6. EndIf
3. EndIf
4. Stop

**Insertion of a node into a single linked list at the front.** The algorithm INSERT\_SL\_FRONT is to insert a node into a single linked list at the front of the list.

**Algorithm INSERT\_SL\_FRONT(HEADER, X)**

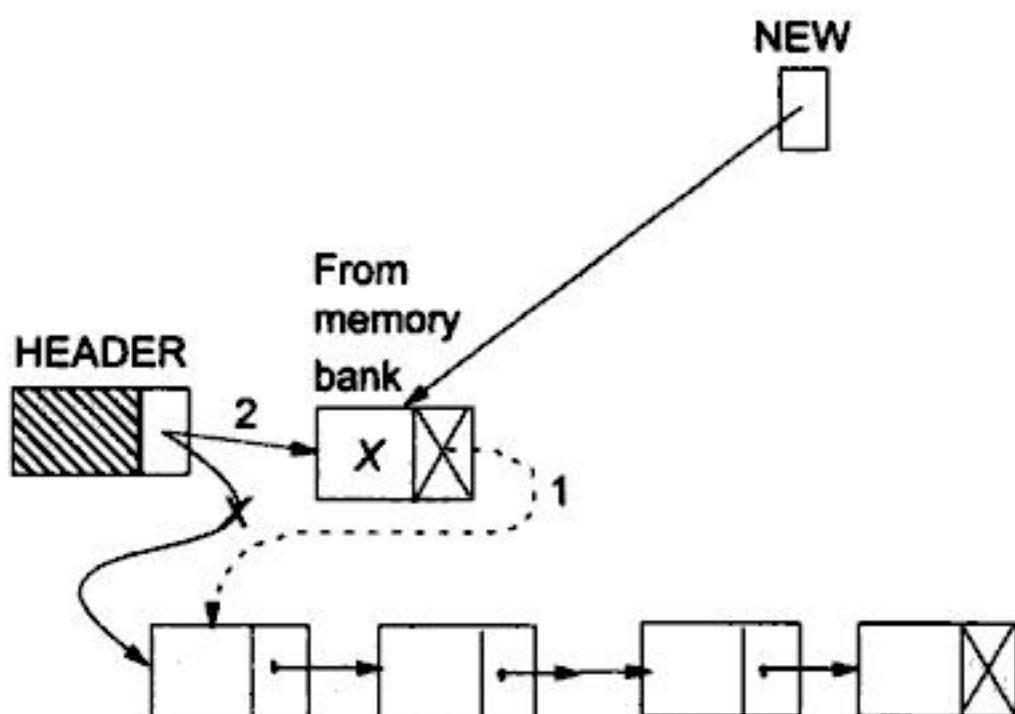
**Input:** HEADER is the pointer to the header node and X is the data of the node to be inserted.

**Output:** A single linked list with newly inserted node in the front of the list.

**Data structures:** A single linked list whose address of the starting node is known from HEADER.

**Steps:**

1. new = GETNODE(NODE) //Get a memory block of type NODE and store //its pointer in new
2. If (new = NULL) then //Memory manager returns NULL on searching //the memory bank
  1. Print "Memory underflow: No insertion"
  2. Exit //Quit the program
3. Else
  1. new.LINK = HEADER.LINK //Change of pointer 1 as shown in Figure 3.5(a)
  2. new.DATA = X //Copy the data X to newly availed node
  3. HEADER.LINK = new //Change of pointer 2 as shown in Figure 3.5(a)
4. EndIf
5. Stop



**Fig. 3.5(a)** Insertion of a node at the front of a single linked list.

**Insertion of a node into a single linked list at the end.** The algorithm **INSERT\_SL\_END** is to insert a node into a single linked list at the end of the list.

**Algorithm INSERT\_SL\_END (HEADER, X)**

**Input:** HEADER is the pointer to the header node and X is the data of the node to be inserted.

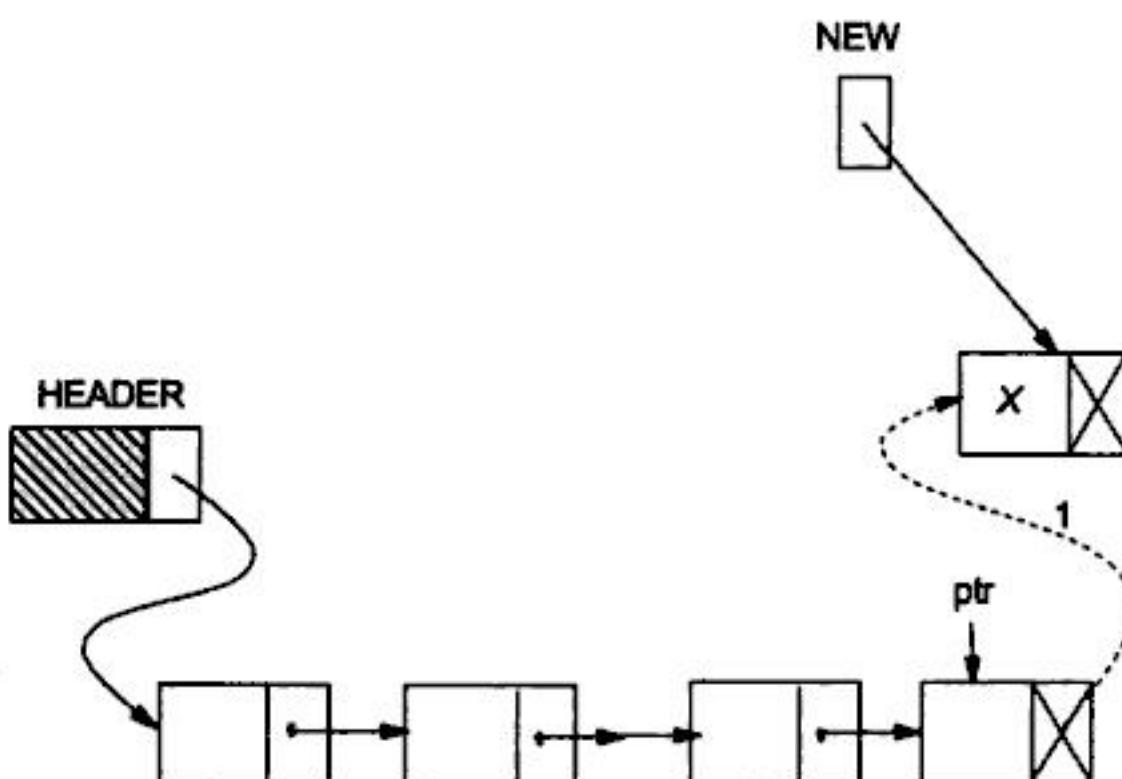
**Output:** A single linked list with newly inserted node having data X at end.

**Data structures:** A single linked list whose address of the starting node is known from HEADER.

**Steps:**

1. new = GETNODE(NODE) //Get a memory block of type NODE and store its pointer in new
2. If (new = NULL) then //Unable to allocate memory for a node
  1. Print "Memory is insufficient: Insertion is not possible"
  2. Exit //Quit the program
3. Else //Move to the end of the given list and then insert
  1. ptr = HEADER //Start from the HEADER node
  2. While (ptr.LINK ≠ NULL) do //Move to the end
    1. ptr = ptr.LINK //Change pointer to the next node
  3. EndWhile
  4. ptr.LINK = new //Change the link field of last node: Pointer 1 as in Figure 3.5(b)
  5. new.DATA = X //Copy the content X into new node
4. EndIf
5. Stop

**Insertion of a node into a single linked list at any position in the list.** The algorithm **INSERT\_SL\_ANY** is to insert a node into a single linked list at any position in the list.



**Fig. 3.5(b)** Insertion at the end of a single linked list.

#### Algorithm INSERT\_SL\_ANY(HEADER, X, KEY)

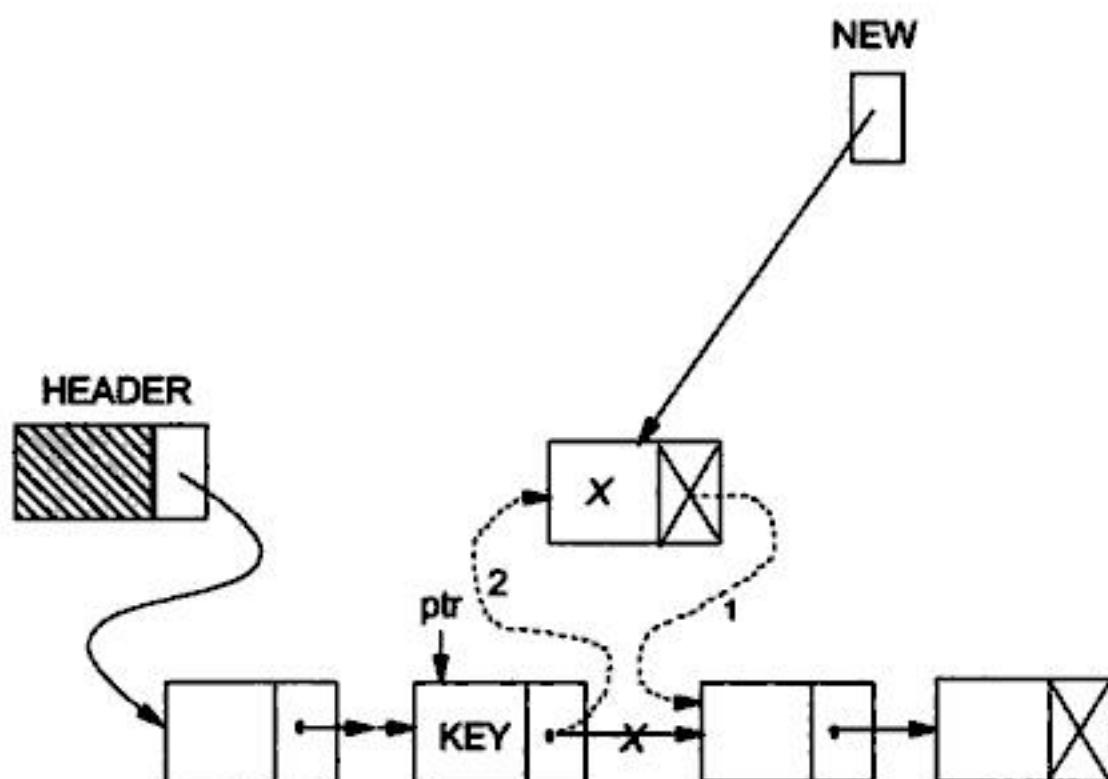
**Input:** HEADER is the pointer to the header node, X is the data of the node to be inserted, and KEY being the data of the key node after which the node has to be inserted.

**Output:** A single linked list enriched with newly inserted node having data X after the node with data KEY.

**Data structures:** A single linked list whose address of the starting node is known from HEADER.

#### Steps:

1. new = GETNODE(NODE) //Get a memory block of type NODE and store its pointer in new
2. If (new = NULL) then //Unable to allocate memory for a node
  1. Print "Memory is insufficient: Insertion is not possible"
  2. Exit //Quit the program
3. Else
  1. ptr = HEADER //Start from the HEADER node
  2. While (ptr.DATA ≠ KEY) and (ptr.LINK ≠ NULL) do //Move to the node having data as KEY or at the end if KEY is not in the list
    1. ptr = ptr.LINK
  3. EndWhile
  4. If (ptr.LINK = NULL) then //Search fails to find the KEY
    1. Print "KEY is not available in the list"
    2. Exit
  5. Else
    1. new.LINK = ptr.LINK //Change the pointer 1 as shown in Figure 3.5(c)
    2. new.DATA = X //Copy the content into the new node
    3. ptr.LINK = new //Change the pointer 2 as shown in Figure 3.5(c)
  6. EndIf
4. EndIf
5. Stop



**Fig. 3.5(c)** Insert at any position in a single linked list.

### **Deletion of a node from a single linked list**

Like insertions, there are also various cases of deletion:

- (i) Deletion at the front of the list
- (ii) Deletion at the end of the list
- (iii) Deletion at any position in the list.

Let us consider the procedure for various cases of deletions. We will assume a procedure namely, RETURNNODE(PTR) which returns a node having pointer PTR to the free pool of storage. The procedure RETURNNODE(PTR) may be defined as follows:

#### **Procedure RETURNNODE(PTR)**

1. ptr1 = AVAIL
2. While (ptr1.LINK ≠ NULL) do
  1. ptr1 = ptr1.LINK
3. EndWhile
4. ptr1.LINK = PTR
5. PTR.LINK = NULL
6. Stop

//PTR is the pointer of a node to be returned

//Start from the beginning of the free pool

//Insert the node at the end

//Node inserted is the last node

*Note:* The procedure RETURNNODE( ) inserts the free node at the end of the pool of free storage whose header address is AVAIL. Alternatively, we can insert the free node at the front of the AVAIL list which is left as an exercise.

**Deletion of a node from a single linked list at the front.** The algorithm DELETE\_SL\_FRONT is to delete a node from a single linked list at the front of the list.

#### **Algorithm DELETE\_SL\_FRONT(HEADER)**

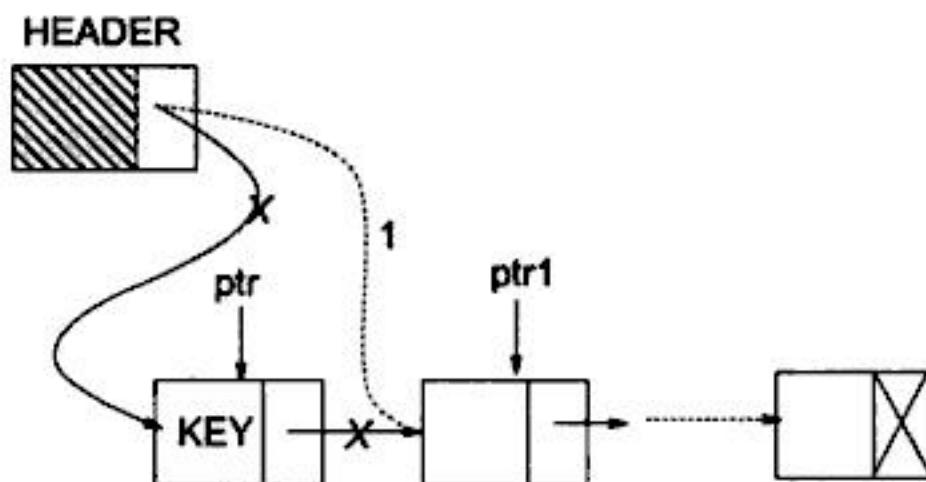
**Input:** HEADER is the pointer to the header node.

**Output:** A single linked list eliminating the node at the front.

**Data structures:** A single linked list whose address of the starting node is known from HEADER.

**Steps:**

1. `ptr = HEADER.LINK` //Pointer to the first node
2. If (`ptr = NULL`) //If the list is empty
  1. Print "The list is empty: No deletion"
  2. Exit //Quit the program
3. Else //The list is not empty
  1. `ptr1 = ptr.LINK` //ptr1 is the pointer to the second node, if any
  2. `HEADER.LINK = ptr1` //Next node becomes the first node as in Figure 3.6(a)
  3. `RETURNNNODE(ptr)` //Deleted node is freed to the memory bank for future use
4. EndIf
5. Stop



**Fig. 3.6(a) Deletion of a node from a single linked list at the front.**

**Deletion of a node from a single linked list at the end.** The algorithm `DELETE_SL_END` is to delete a node at the end from a single linked list.

**Algorithm `DELETE_SL_END(HEADER)`**

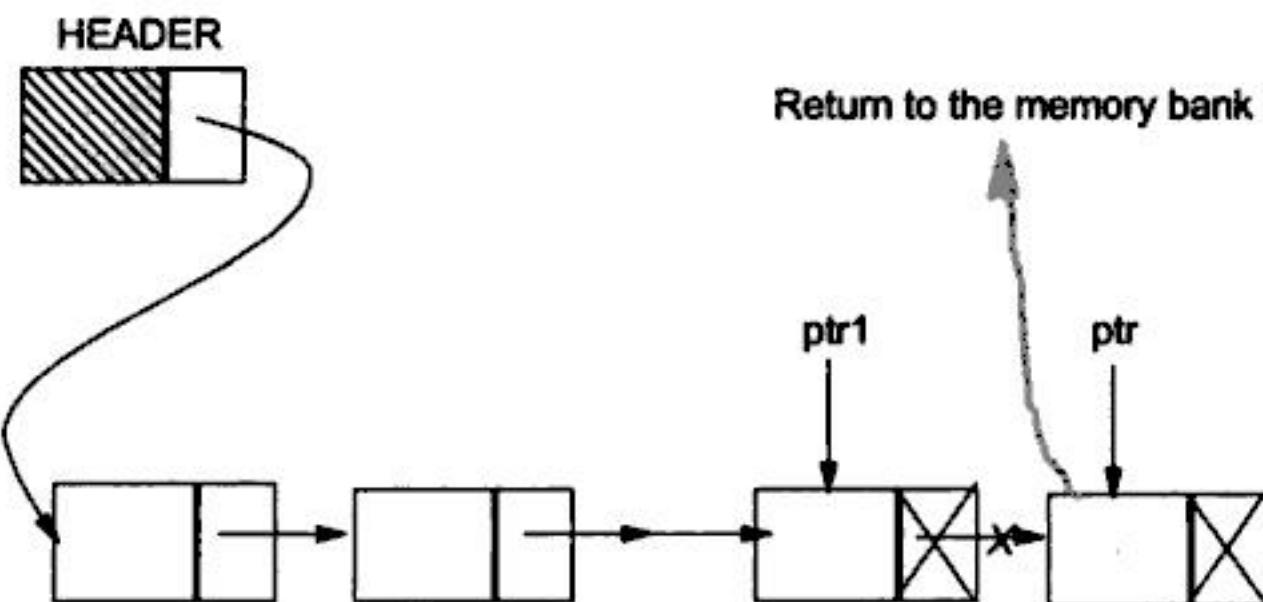
**Input:** `HEADER` is the pointer to the header node.

**Output:** A single linked list eliminating the node at the end.

**Data structures:** A single linked list whose address of the starting node is known from `HEADER`.

**Steps:**

1. `ptr = HEADER` //Move from the header node
2. If (`ptr.LINK = NULL`) then
  1. Print "The list is empty: No deletion possible"
  2. Exit //Quit the program
3. Else
  1. While (`ptr.LINK ≠ NULL`) do
    1. `ptr1 = ptr` //To store the previous pointer
    2. `ptr = ptr.LINK` //Move to the next
  2. EndWhile
  3. `ptr1.LINK = NULL` //Last but one node become the last node as in //Figure 3.6(b)
  4. `RETURNNNODE(ptr)` //Deleted node is returned to the memory bank //for future use
4. EndIf
5. Stop



**Fig. 3.6(b)** Deletion of a node from a single linked list at the end.

**Deletion of a node from a single linked list at any position in the list.** The next algorithm **DELETE\_SL\_ANY** is to delete a node from any position in the single linked list.

#### Algorithm **DELETE\_SL\_ANY**

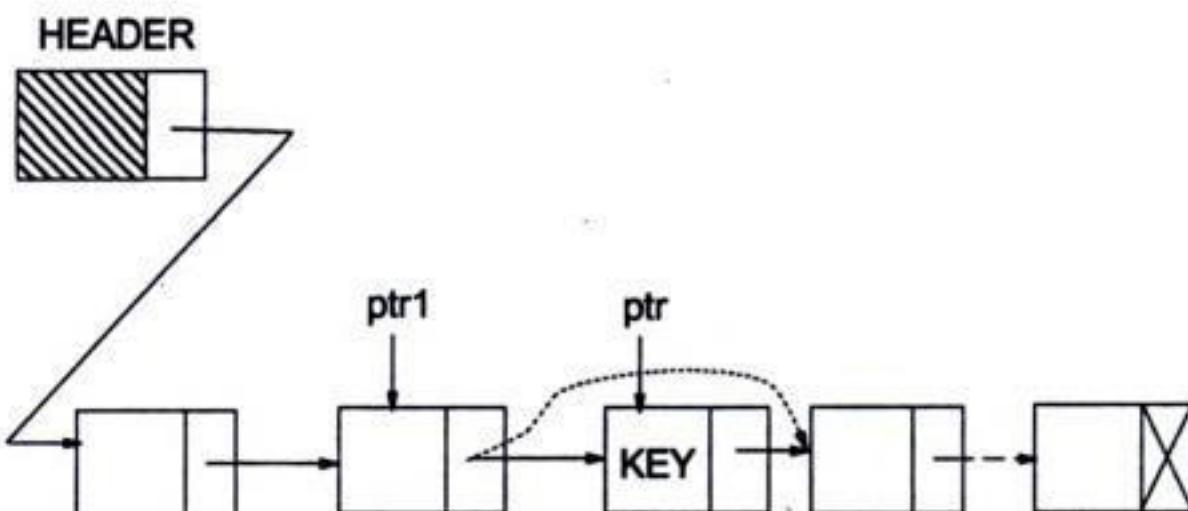
**Input:** HEADER is the pointer to the header node, KEY is the data content of the node to be deleted.

**Output:** A single linked list except the node with data content as KEY.

**Data structures:** A single linked list whose address of the starting node is known from HEADER.

#### Steps:

1.  $\text{ptr1} = \text{HEADER}$  //Start from the header node
2.  $\text{ptr} = \text{ptr1.LINK}$  //This points to the first node, if any
3. While ( $\text{ptr} \neq \text{NULL}$ ) do
  1. If ( $\text{ptr.DATA} \neq \text{KEY}$ ) then
    1.  $\text{ptr1} = \text{ptr}$  //If not found the key
    2.  $\text{ptr} = \text{ptr.LINK}$  //Keep a track of the pointer of the previous node
  2. Else
    1.  $\text{ptr1.LINK} = \text{ptr.LINK}$  //Shift to the next
    2. **RETURNNODE(ptr)** //The node is found
    3. Exit //Link field of the predecessor is to point
3. EndIf //the successor of node under deletion, see Figure 3.6(c)
4. EndWhile //Return the deleted node to the memory bank
5. If ( $\text{ptr} = \text{NULL}$ ) then //When the desired node is not available in the list
  1. Print "Node with KEY does not exist: No deletion"
6. EndIf //Exit the program
7. Stop



**Fig. 3.6(c) Deletion of a node at any position in a single linked list.**

### Assignment 3.1

- It can be noted that in all our previous algorithms on single linked lists, we have maintained a header node whose data field contains null value and link field points the first node in the list. This simply means that even in an empty list there is a node. However, maintenance of this special node is just a convention. So, instead of making the data field to null value, if we store actual data, that is, header node itself becomes the first node then what modification you would suggest for all the algorithms as mentioned for insertion and deletion operations. [Hint: Only little change is required.]
- An alternative implementation for the algorithm DELETE\_SL\_ANY is proposed as given below:

#### Algorithm DELETE\_SL\_ANY\_ALTERNATIVE(HEADER, KEY)

**Input:** HEADER is the pointer to the header node, KEY is the data content of the node to be deleted.

**Output:** A single linked list except the node with data content as KEY.

**Data structures:** A single linked list whose address of the starting node is known from HEADER.

#### Steps:

1.  $\text{ptr} = \text{HEADER.LINK}$
2. While( $\text{ptr.DATA} \neq \text{KEY}$ ) and ( $\text{ptr} \neq \text{NULL}$ ) do
  1.  $\text{ptr1} = \text{ptr}$
  2.  $\text{ptr} = \text{ptr.LINK}$
3. EndWhile
4. If ( $\text{ptr.DATA} = \text{KEY}$ ) then
  1.  $\text{ptr1.LINK} = \text{ptr.LINK}$
  2. RETURNNODE(ptr)
5. Else
  1. Print "Node does not exist : Deletion is unsuccessful"
6. EndIf
7. Stop

With respect to the above algorithm, answer the following:

- (i) Is the algorithm works for an empty list (the list containing only header node)?
- (ii) If the key element is at the first position or last position then will it work correctly?
- (iii) Find the bugs, if any, in it and then rectify for correct operation.

***Copy of a single linked list***

For a given list we can copy it into another list by duplicating the content of each node into newly allocated node. Following is an algorithm to copy an entire single linked list.

**Algorithm COPY\_SL(HEADER; HEADER1)**

**Input:** HEADER is the pointer to the header node of the list.

**Output:** HEADER1 is the pointer to the duplicate list.

**Data structures:** Single linked list structure.

**Steps:**

1.  $\text{ptr} = \text{HEADER}$  //Current position in the master list
2.  $\text{HEADER1} = \text{GETNODE(NODE)}$  //Get a node for the header of a duplicate list
3.  $\text{ptr1} = \text{HEADER1}$  //ptr1 is the current position in the duplicated list
4.  $\text{ptr1.DATA} = \text{NULL}$  //Header node does not contain any data
5. While ( $\text{ptr} \neq \text{NULL}$ ) do //Till the traversal of master node is finished
  1.  $\text{new} = \text{GETNODE(NODE)}$  //Get a new node from memory bank
  2.  $\text{new.DATA} = \text{ptr.DATA}$  //Copy the content
  3.  $\text{ptr1.LINK} = \text{new}$  //Insert the node at the end of the duplicate list
  4.  $\text{new.LINK} = \text{NULL}$
  5.  $\text{ptr1} = \text{new}$
  6.  $\text{ptr} = \text{ptr.LINK}$  // Move to the next node in the master list
6. EndWhile
7. Stop

***Merging of two single linked lists into one list***

Suppose two single linked lists, namely, L1 and L2 are there and we want to merge the list L2 after L1. Also assume that, HEADER1 and HEADER2 are the header nodes of the lists L1 and L2 respectively. Merging can be done by setting the pointer of the link field of the last node in the list L1 with the pointer of the first node in L2. The header node in the list L2 should be returned to the pool of free storage. Merging two single linked lists into one list is illustrated in Figure 3.7.

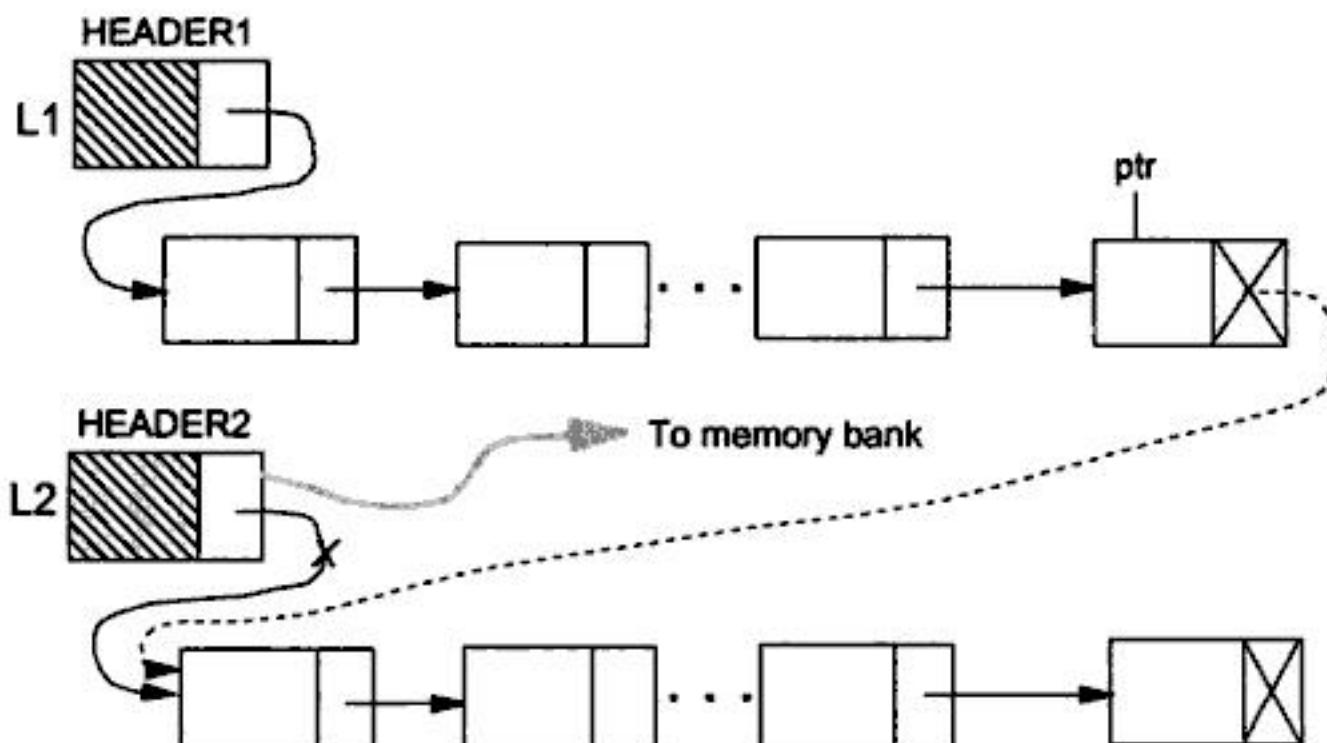


Fig. 3.7 Merging of two single linked lists into one single linked list.

Following is the algorithm MERGE\_SL to merge two single linked lists into one single linked list:

**Algorithm MERGE\_SL(HEADER1, HEADER2; HEADER)**

**Input:** HEADER1 and HEADER2 are pointers to header nodes of lists (L1 and L2, respectively) to be merged.

**Output:** HEADER is the pointer to the resultant list.

**Data structures:** Single linked list structure.

**Steps:**

1. ptr = HEADER1
2. While (ptr.LINK ≠ NULL) do //Move to the last node in the list L1
  1. ptr = ptr.LINK
3. EndWhile
4. ptr.LINK = HEADER2.LINK //Last node in L1 points to the first node in L2
5. RETURNNODE(HEADER2) //Return the header node to the memory bank
6. HEADER = HEADER1 //HEADER becomes the header node of the merged list
7. Stop

**Assignment 3.2** Perform the following tasks on linked lists using single linked list representation:

- (a) Obtain insertion and deletion operations on a list which is stored in an array.
- (b) Find the maximum, minimum and average for some numeric data as element stored in single linked list.
- (c) For a given set of data, insert them in a sorted fashion.
- (d) A list containing unsorted data is known, make it as a sorted list.
- (e) Suppose a list is no more required, dispose the entire list into the pool of free storage.
- (f) Suppose two lists are known; compare two lists to furnish the following:
  - (i) Exact replica or not
  - (ii) Number of matching occurs between the lists
  - (iii) Two lists are sorted in the same order or opposite order etc.
- (g) For a given list obtain decatenation (sublist). Decide yourself a suitable criteria for splitting.
- (h) Store the element in reverse order that of a given list (don't make another list).

**Searching for an element in a single linked list**

The algorithm SEARCH\_SL( ) is given below to search an item in a single linked list.

**Algorithm SEARCH\_SL(Key; LOCATION)**

**Input:** KEY, the item to be searched.

**Output:** LOCATION, the pointer to a node where the KEY belongs to or an error messages.

**Data structures:** A single linked list whose address of the starting node is known from HEADER.

**Step:**

1. ptr = HEADER.LINK //Start from the first node
2. flag = 0, LOCATION = NULL

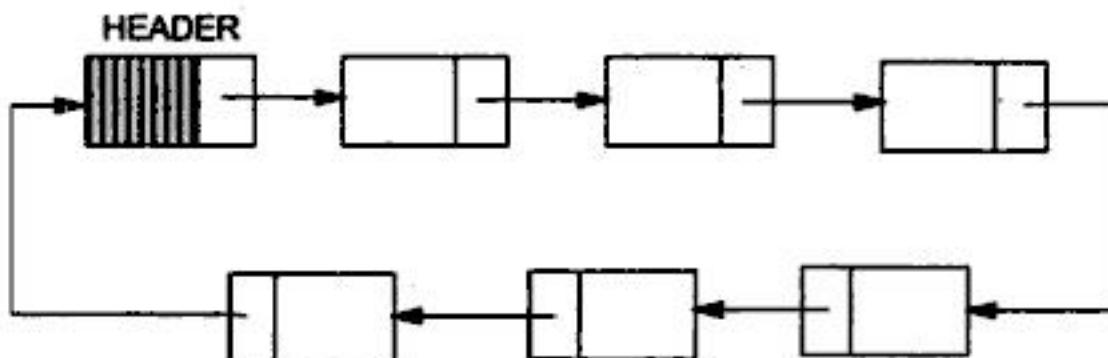
```

3. While (ptr ≠ NULL) and (flag = 0) do
 1. If (ptr.DATA = KEY) then
 1. flag = 1 //Search is finished
 2. LOCATION = ptr
 3. Return (LOCATION)
 2. Else
 1. ptr = ptr.LINK //Move to the next node
 3. EndIf
4. EndWhile
5. If (ptr = NULL) then
 1. Print "Search is unsuccessful"
6. EndIf
7. Stop

```

### 3.3 CIRCULAR LINKED LIST

In our previous discussion, we have noticed that in single linked list, the link field of the last node is null (hereafter a single linked list may be read as ordinary linked list), but a number of advantages can be gained if we utilize this link field to store the pointer of the header node. A linked list where the last node points the header node is called *circular* linked list. Figure 3.8 shows a pictorial representation of a circular linked list.



**Fig. 3.8** A circular linked list.

Circular linked lists have certain advantages over ordinary linked lists. Few advantages of circular linked lists are discussed as below:

#### **Accessibility of a member node in the list**

In an ordinary list, a member node is accessible from a particular node, that is, from the header node only. But in a circular linked list, every member node is accessible from any node by merely chaining through the list.

**Example:** Suppose, we are manipulating some information which are stored in a list. Also, think of a case, where for a given data we want to find the earlier occurrence(s) as well as post occurrence(s). Post occurrence(s) can be traced out by chaining through the list from the current node irrespective of whether the list is maintained as circular linked or ordinary linked list.

In order to find all the earlier occurrences, in case of ordinary linked lists, we have to start our traversing from the header node at the cost of maintaining pointer for header in addition to the pointers for current node and another for chaining. But in case of circular linked list, one can trace out the same without maintaining the header information; rather maintaining only two pointers. Note that, in ordinary linked lists, one can chain through left to right only where it is virtually in the both direction for circular linked lists.

### **Null link problem**

Null value in the link field may create some problem during the execution of programs if a proper care is not taken. This is illustrated by mentioning two algorithms to perform search on ordinary linked list and circular linked list.

#### **Algorithm SEARCH\_SL(KEY)**

**Input:** KEY the item to be searched.

**Output:** Location, the pointer to a node where KEY belongs or an error message.

**Data structures:** A single linked list whose address of the starting node is known from HEADER.

#### **Steps:**

1. **ptr = HEADER.LINK**
2. **While (ptr ≠ NULL) do**
  1. **If (ptr.DATA ≠ KEY) then**
    1. **ptr = ptr.LINK**
  2. **Else**
    1. **Return(ptr)**
    2. **Exit**
    3. **EndIf**
  3. **EndWhile**
  4. **If (ptr = NULL) then**
    1. **Print "The entire list has traversed but KEY is not found"**
  5. **EndIf**
  6. **Stop**

Note that, here two tests in step 2 (which control the loop of searching) cannot be placed together as **While (ptr ≠ NULL) AND (ptr.DATA ≠ KEY) do** because in that case there will be an execution error for **ptr.DATA** since it is not defined when **ptr = NULL**. But with circular linked list very simple implementation is possible without any special care for NULL pointer. As an illustration the searching of an element in a circular linked list is given below:

#### **Algorithm SEARCH\_CL(KEY)**

**Input:** KEY the item of search.

**Output:** Location, the pointer to a node where KEY belongs or an error message.

**Data structures:** A circular linked list whose address to the starting node is known from HEADER.

#### **Steps:**

1. **ptr = HEADER.LINK**
2. **While (ptr.DATA ≠ KEY) and (ptr ≠ HEADER) do**
  1. **ptr = ptr.LINK**
3. **EndWhile**
4. **If (ptr.DATA = KEY)**
  1. **Return (ptr)**
5. **Else**
  1. **Print "Entire list is searched: KEY node is not found"**
6. **EndIf**
7. **Stop**

### Some easy-to-implement operations

Some operation can easily be implemented with circular linked list than ordinary linked list. Operations like merging (concatenation) splitting (decatenation), deletion, dispose of an entire list, etc., can easily be performed on circular linked list. The merging operation, as in Figure 3.9, is explained in algorithm MERGE\_CL as under:

#### Algorithm MERGE\_CL(HEADER1, HEADER2)

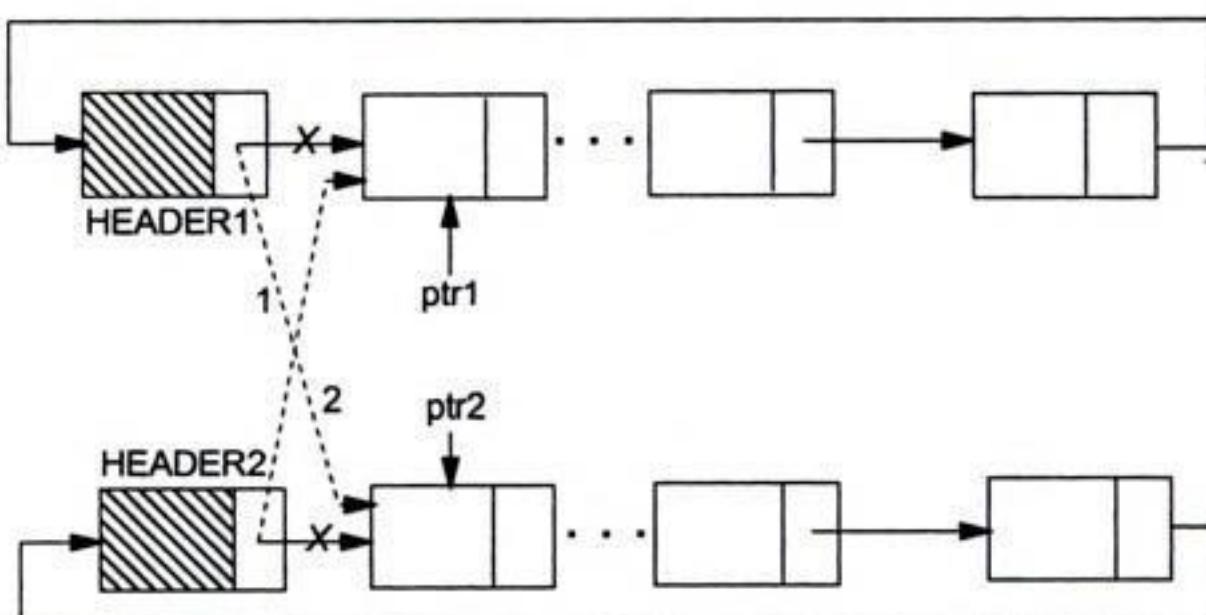
**Input:** HEADER1 and HEADER2 are the two pointers to header nodes.

**Output:** A larger circular linked list containing all the nodes from lists HEADER1 and HEADER2.

**Data structures:** Circular linked list structure.

#### Steps:

1.  $\text{ptr1} = \text{HEADER1.LINK}$
2.  $\text{ptr2} = \text{HEADER2.LINK}$
3.  $\text{HEADER1.LINK} = \text{ptr2}$
4.  $\text{HEADER2.LINK} = \text{ptr1}$
5. Stop



**Fig. 3.9** Concatenation of two circular linked lists.

One can easily compare the algorithm MERGE\_CL with MERGE\_SL where an entire list is required to be traversed in order to locate the last node, and at the cost of some time.

It can be noted that the algorithm MERGE\_CL keeps two header node thus wasting some memory space; in fact the maintenance of header node in circular list has no significance (truly speaking it is a burden here) unlike ordinary linked lists.

#### Assignment 3.3

- (a) Modify algorithm MERGE\_CL so that it will contain only one header node instead of two header nodes.
- (b) By writing algorithms for disposing the entire list with (i) ordinary linked list and (ii) circular linked list, show that the algorithm for circular linked list is better than ordinary linked list.
- (c) Write a recursive algorithm to invert (pointer will point in reverse direction) a circular linked list.
- (d) Write algorithms to convert an ordinary single linked list into a circular linked list and vice versa.



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.

```

3. EndWhile
4. new = GETNODE(NODE) //Get a new node from the pool of free storage
5. If (new = NULL) then //When the memory is not available
 1. Print "Memory is not available"
 2. Exit //Quit the program
6. EndIf
7. If (ptr.RLINK = NULL) //If the KEY node is at the end or not found in the list
 1. new.LLINK = ptr
 2. ptr.RLINK = new //Insert at the end
 3. new.RLINK = NULL
 4. new.DATA = X //Copy the information to the newly inserted node
8. Else
 1. ptr1 = ptr.RLINK //The KEY is available
 2. new.LLINK = ptr
 3. new.RLINK = ptr1 //Next node after the key node
 4. ptr.RLINK = new //Change the pointer shown as 2 in Figure 3.11(c)
 5. ptr1.LLINK = new //Change the pointer shown as 4 in Figure 3.11(c)
 6. ptr = new //Change the pointer shown as 1 in Figure 3.11(c)
 7. new.DATA = X //Change the pointer shown as 3 in Figure 3.11(c)
 //This becomes the current node
 8. new.RLINK = NULL //Copy the content to the newly inserted node
9. EndIf
10. Stop

```

*Note:* Observe that the algorithm INSERT\_DL\_ANY will insert a node even the key node does not exist. In that case the node will be inserted at the end of the list. Also, note that the algorithm will work even if the list is empty.

### ***Deletion in a double linked list***

Deletion of a node from a double linked list may take place from any position in the list, which are depicted as shown in Figure 3.12. Let us consider each of the cases separately.

**Deletion of a node at the front of a double linked list.** The algorithm is as under:

#### **Algorithm DELETE\_DL\_FRONT( )**

**Input:** A double linked list with data.

**Output:** A reduced double linked list.

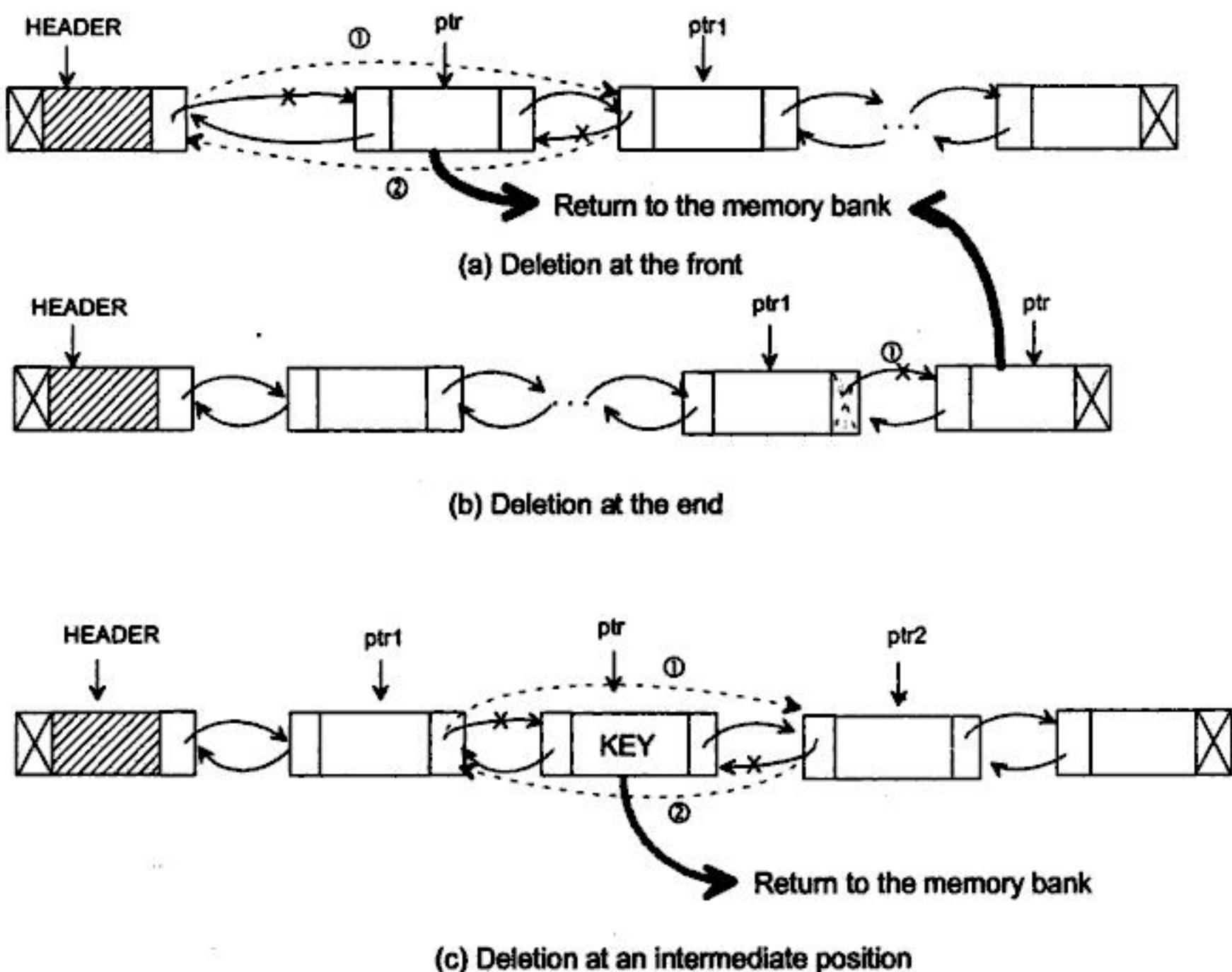
**Data structure:** Double linked list structure whose pointer to the header node is HEADER.

#### **Steps:**

```

1. ptr = HEADER.RLINK //Pointer to the first node
2. If (ptr = NULL) then //If the list is empty
 1. Print "List is empty: No deletion is made"
 2. Exit
3. Else
 1. ptr1 = ptr.RLINK //Pointer to the second node
 2. HEADER.RLINK = ptr1 //Change the pointer shown as 1 in Figure 3.12(a)
 3. If (ptr1 ≠ NULL)
 1. ptr1.LLINK = HEADER //If the list contains a node after the first node
 //of deletion
 //Change the pointer shown as 2 in Figure 3.12(a)

```



**Fig. 3.12** Deletion at various position in a double linked list.

```

4. EndIf
5. RETURNNODE (ptr) //Return the deleted node to the memory bank
4. EndIf
5. Stop

```

Note that the above algorithm will work even if the list is empty.

**Deletion of a node at the end of a double linked list.** The algorithm is as under:

**Algorithm DELETE\_DL\_END( )**

**Input:** A double linked list with data.

**Output:** A reduced double linked list.

**Data structure:** Double linked list structure whose pointer to the header node is HEADER.

**Steps:**

1.  $\text{ptr} = \text{HEADER}$
2. While ( $\text{ptr.RLINK} \neq \text{NULL}$ ) do //Move to the last node
  1.  $\text{ptr} = \text{ptr.RLINK}$
3. EndWhile



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.

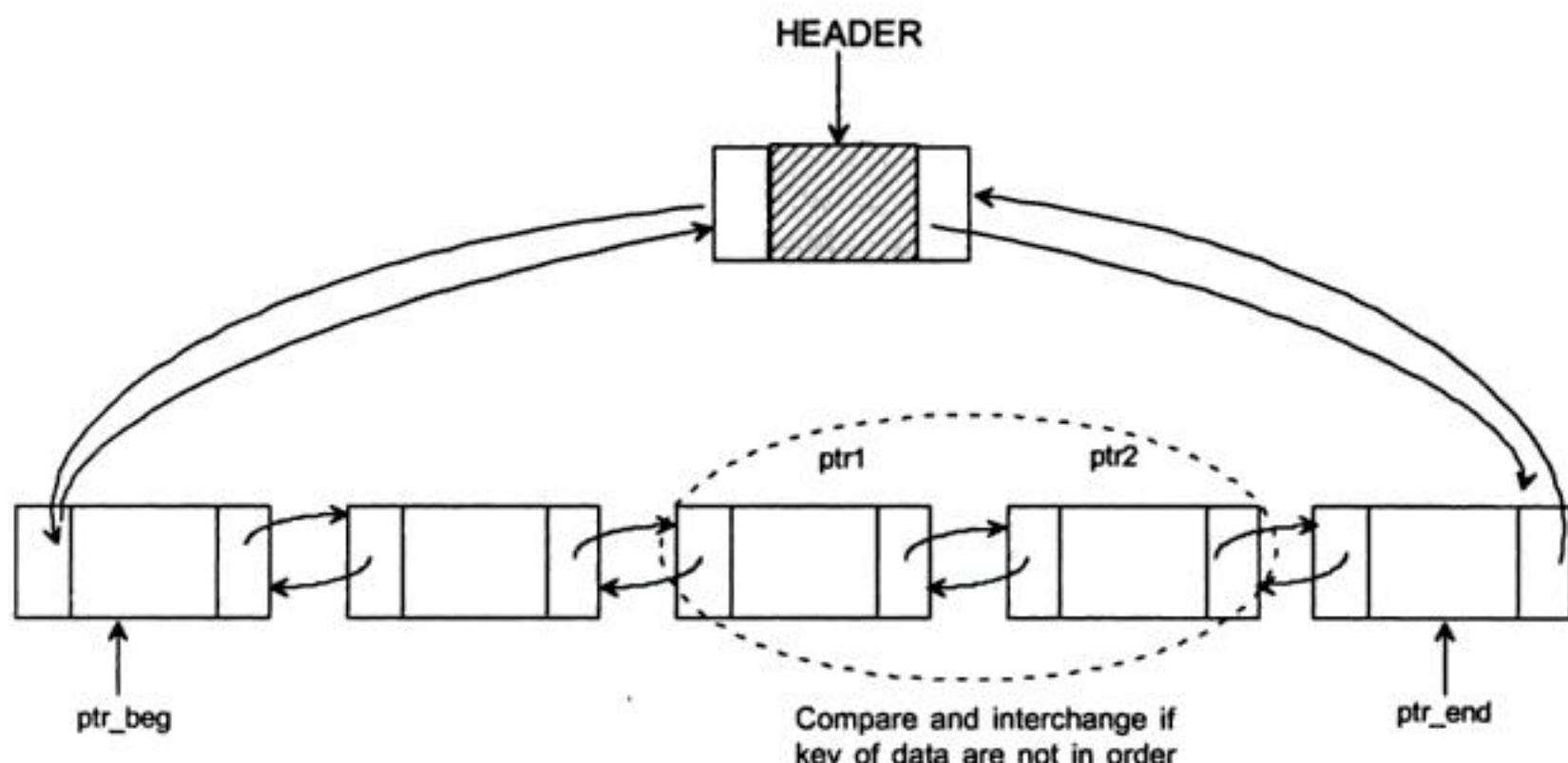
```

3. While (ptr2 ≠ ptr_end) do //For compare and interchange—inner loop
 1. If ORDER(ptr1.DATA, ptr2.DATA) = FALSE then
 1. SWAP (ptr1, ptr2) //Interchange the data content at ptr1 and ptr2
 2. EndIf
 3. ptr1 = ptr1.RLINK //Move the first pointer to the next
 4. ptr2 = ptr2.RLINK //Move the second pointer to the next
 4. EndWhile
 5. ptr_end = ptr_end.LLINK //Right most node is now sorted out
4. EndWhile
5. Stop

```

In the above algorithm, we have assumed the procedure ORDER(data1, data2). To test whether two data are in a required order or not; it will return TRUE if they are in order else FALSE. We also assume another procedure SWAP(ptr1, ptr2) to interchange the data content at the nodes pointed by the pointers ptr1 and ptr2. These procedures are very easy to implement.

The above algorithm uses the bubble sorting technique. Execution of each outer loop bubbles up the largest node towards the right end of sorting (say, in ascending order) and each inner loop is to compare between the successive nodes and push up the largest towards the right if they are not in order. Figure 3.15 illustrates the sorting procedure.



**Fig. 3.15** Sorting operation and use of various pointers.

**Assignment 3.5** A college uses the following structure for a graduate class:

1. Student (30)
  2. Name
    3. LAST (12 Characters)
    3. FIRST (12 Characters)
    3. MIDDLE (12 Characters)
  2. RollNo (15 Alphanumeric characters)



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.

```

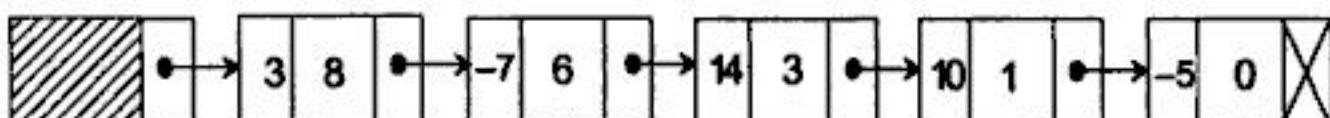
//Get header nodes for column header//
6. ptr = HEADER
7. For col = 1 to n do
 1. new = GETNODE(NODE)
 2. new.i = 0, new.j = col, new.DATA = NULL
 3. ptr.COLLINK = new
 4. new.COLLINK = HEADER, new.ROWLINK = new
 5. ptr = new
8. EndFor
//Get header nodes for row header//
9. ptr = HEADER
10. For row = 1 to m
 1. new = GETNODE(NODE)
 2. new.i = row, new.j = 0, new.DATA = NULL
 3. ptr.ROWLINK = new
 4. new.ROWLINK = HEADER, new.COLLINK = new
 5. ptr = new
11. EndFor
//Insertion of elements a_{ij} into the sparse matrix//
12. Read (data, row, col) //Read the element to be inserted and its position
13. rowheader = HEADER.COLLINK, colheader = HEADER.ROWLINK
 //Go to the row header of the i-th row//
14. While (row < rowheader.i)
 1. rowheader = rowheader.COLLINK
15. EndWhile
 //Go to the column header of the j-th column//
16. While (col < colheader.j)
 1. colheader = colheader.ROWLINK
17. EndWhile
 //Find the position for insertion//
18. rowptr = rowheader //ON ROW
19. While (rowptr.j < col) do
20. 1. ptr1 = rowptr //This is to find the predecessor and successor
 2. rowptr = rowptr.ROWLINK //ptr1 points the predecessor i.e. the node at
 3. If (rowptr = rowheader) then
 1. Break //j-th column
 2. colptr = colheader //rowptr is the pointer to successor
 3. EndIf
21. EndWhile
22. colptr = colheader //On Column
23. While (colptr.i < row)
 1. ptr2 = colptr //This is to find the predecessor and successor
 2. colptr = colptr.COLLINK //ptr2 points the predecessor, i.e. the node in
 3. If (colptr = coolheaded) //i - 1-th row
 1. Break //colptr is the pointer to successor
 2. EndIf
24. EndWhile

```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Considering the single linked list representation, a node should have three fields: COEFF (to store the coefficient  $a_i$ ), EXP (to store the exponent  $e_i$ ) and a LINK (to store the pointer to the next node representing next term). It is evident that number of nodes required to represent a polynomial is the same as the number of terms in the polynomial. An additional node may be considered for a header. As an example, let us consider the single linked list representation of the polynomial  $P(x) = 3x^8 - 7x^6 + 14x^3 + 10x - 5$  would be stored as shown in Figure 3.18.



**Fig. 3.18** Linked list representation of a polynomial (single variable).

Note that, the terms whose coefficients are zero, are not stored here. Next let us consider two basic operations, viz., addition and multiplication of two polynomials with this representation.

**Polynomial addition.** In order to add two polynomials, say, P and Q to get a resultant polynomial R, we have to compare their terms starting at the first nodes and moving towards the end one-by-one. Two pointers Pptr and Qptr are used to move along the terms of P and Q. There may be three cases during the comparison between terms in two polynomials.

(i) **Case 1:** Exponents of two terms are equal. In this case coefficients in two nodes are to be added and a new terms will be created with the values

$$\text{Rptr.COEFF} = \text{Pptr.COEFF} + \text{Qptr.COEFF}$$

and

$$\text{Rptr.EXP} = \text{Pptr.EXP}$$

(ii) **Case 2:**  $\text{Pptr.EXP} > \text{Qptr.EXP}$ , i.e. the exponent of the current term in P is greater than the exponent of the current term in Q. Then, a duplicate of the current term in P is created and to be inserted in the polynomial R.

(iii) **Case 3:**  $\text{Pptr.EXP} < \text{Qptr.EXP}$ , i.e. the case when exponent of the current term in P is less than the exponent of the current term in Q. In this case, a duplicate of the current term of Q is created and to be inserted in the polynomial R. The algorithm POLYNOMIAL\_ADD is described as below:

#### Algorithm POLYNOMIAL\_ADD(PHEADER, QHEADER; RHEADER)

**Input:** Two polynomials P and Q whose header pointers are PHEADER and QHEADER

**Output:** A polynomial R is the sum of P and Q having header as RHEADER.

**Data structure:** Single linked list structure for representing a term in single variable polynomial.

#### Steps:

1.  $\text{Pptr} = \text{PHEADER.LINK}, \text{Qptr} = \text{QHEADER.LINK}$   
//Get a header node for resultant polynomial//
2.  $\text{RHEADER} = \text{GETNODE(NODE)}$
3.  $\text{RHEADER.LINK} = \text{NULL}, \text{RHEADER.EXP} = \text{NULL}, \text{RHEADER.COEFF} = \text{NULL}$
4.  $\text{Rptr} = \text{RHEADER}$  //Current pointer to the resultant polynomial R



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

denotes the current term in P and Qptr be that of in Q. For each term of P we have to visit all the terms in Q; the exponent values in two terms are added ( $R.EXP = P.EXP + Q.EXP$ ), coefficient values are multiplied ( $R.COEFF = P.COEFF \times Q.COEFF$ ), and these values are included into R in such a way that if there is no term in R whose exponent value is the same as exponent value as obtained by adding the exponents from P and Q, then create a new node and insert it to R with the values so obtained (that is  $R.COEFF$ , and  $R.EXP$ ); on the other hand if a node is found in R having same exponent value  $R.EXP$ , then update the coefficient value of it by adding the resultant coefficient ( $R.COEFF$ ) into it. The algorithm POLY\_MULTIPLY is described as below:

**Algorithm POLY\_MULTIPLY(PHEADER, QHEADER; RHEADER)**

**Input:** Two polynomials P and Q having their headers as PHEADER, QHEADER.

**Output:** A polynomial R storing the result of multiplication of P and Q.

**Data structure:** Single linked list structure for representing a term in single variable polynomial.

**Steps:**

1. Pptr = PHEADER, Qptr = QHEADER //Get a node for the header of R//
2. RHEADER = GETNODE(NODE)
3. RHEADER.LINK = NULL, RHEADER.COEFF = NULL, RHEADER.EXP = NULL
4. If (Pptr.LINK = NULL) or (Qptr.LINK = NULL) then
  1. Exit //No valid operation possible
5. EndIf
6. Pptr = Pptr.LINK
7. While (Pptr ≠ NULL) do //For each term of P
  1. While (Qptr ≠ NULL) do
    1. C = Pptr.COEFF × Qptr.COEFF
    2. X = Pptr.EXP + Qptr.EXP
//Search for the equal exponent value in R//
   - 3. Rptr = RHEADER
  - 4. While (Rptr ≠ NULL) and (Rptr.EXP > X) do
    1. Rptr1 = Rptr
    2. Rptr = Rptr.LINK
    3. If (Rptr.EXP = X) then
      1. Rptr.COEFF = Rptr.COEFF + C
    4. Else //Add a new node at correct position in R
      1. new = GETNODE(NODE)
      2. new.EXP = X, new.COEFF = C
      3. If (Rptr.LINK = NULL) then
        1. Rptr.LINK = new //Append the node at end
        2. new.LINK = NULL
      4. Else
        1. Rptr1.LINK = new //Insert the node in ordered position
        2. new.LINK = Rptr
    5. EndIf
    5. EndIf
    5. EndWhile
  - 2. EndWhile



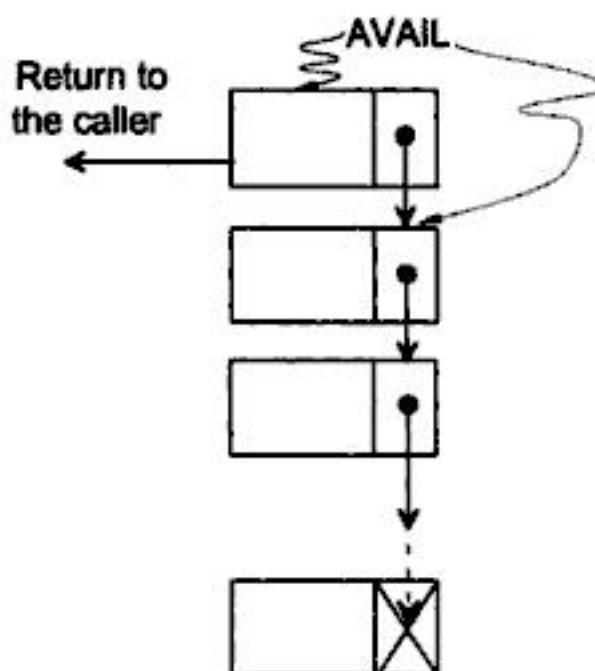
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



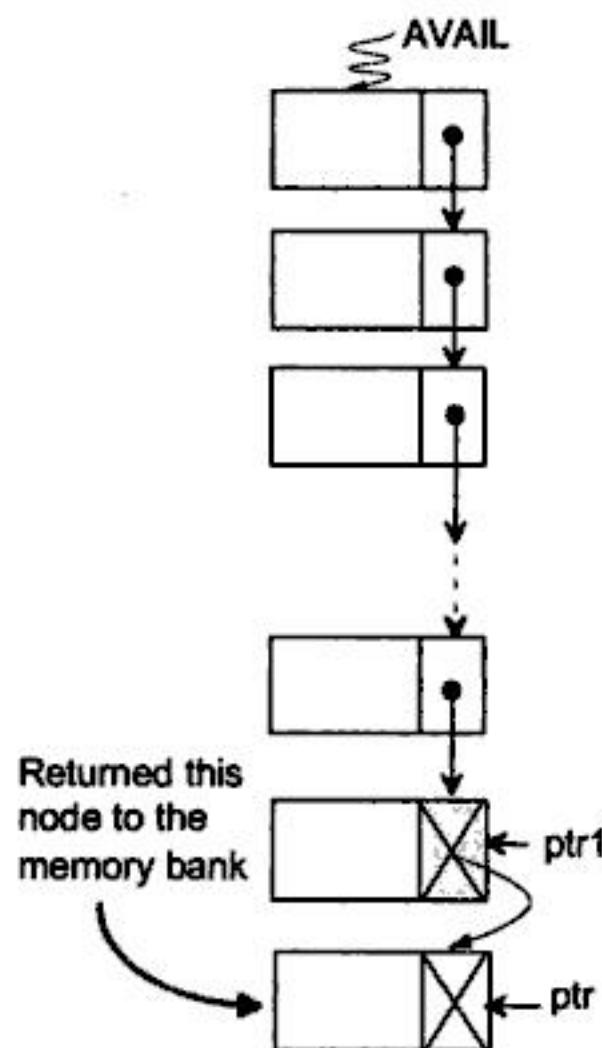
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



**Fig. 3.20** Getting a block from the memory bank.

3. EndWhile
4. `ptr1.LINK = PTR`
5. `PTR.LINK = NULL`
6. Stop

The procedure `RETURNNODE( )` append a returned block (bearing pointer `PTR`) at the end of the pool of free storage pointed by `AVAIL`. Change in pointers can be seen in the Figure 3.21 as dotted line.



**Fig. 3.21** Returning a block to the memory bank.



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.

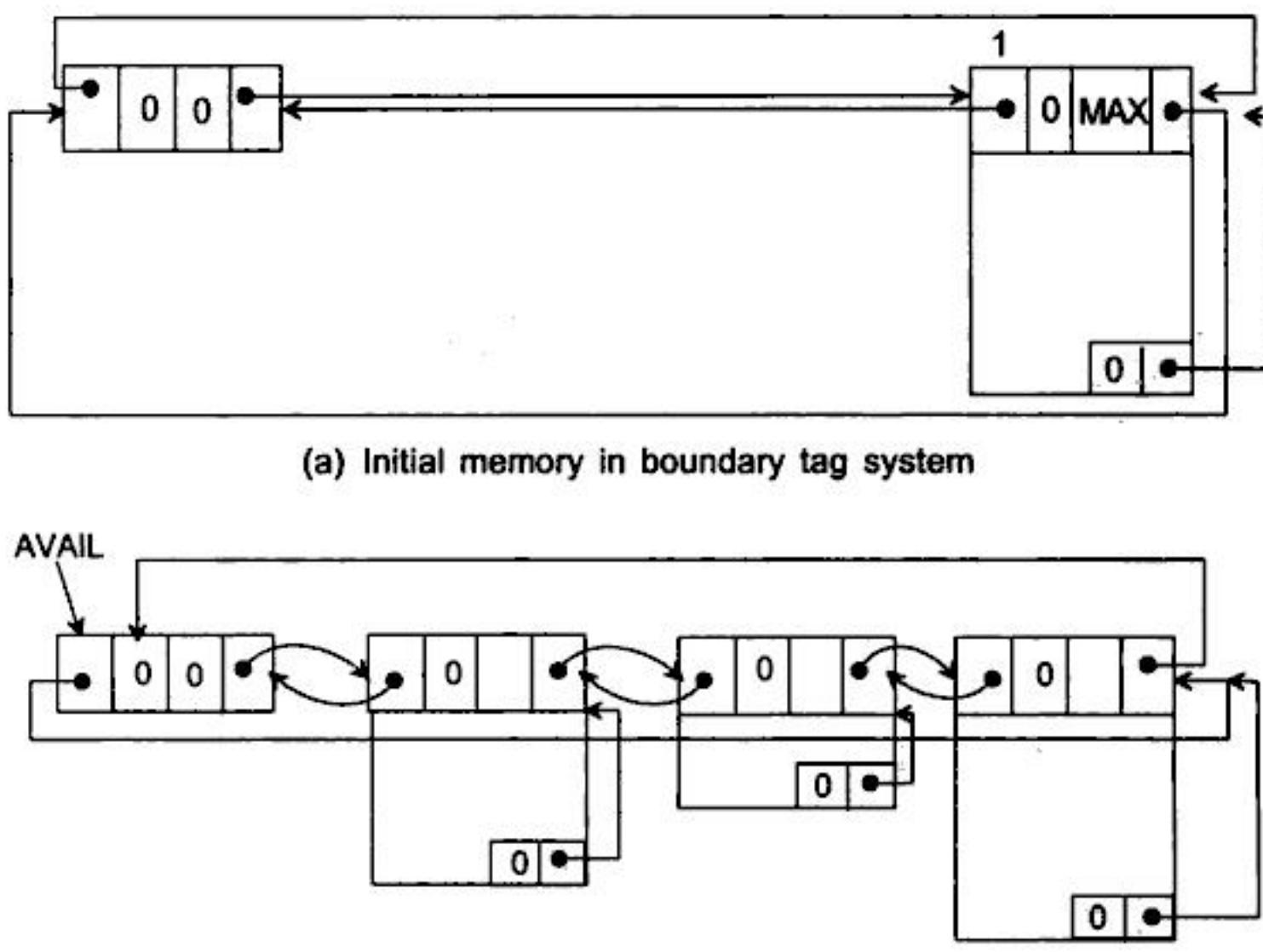


Fig. 3.25 Pool of free storages in boundary tag system.

During the description of algorithms for GETNODE(NODE) and RETURNNODE(ptr) in boundary tag system, we will assume that the memory addresses is numbered from 1 to MAX, and that  $\text{ptr.TAG} = 1$  if  $\text{ptr} < 1$  or  $\text{ptr} > \text{MAX}$ . This assumption is just a convention in order to simplify our algorithms. Before going to describe the algorithms, let us first discuss the allocation and deallocation strategies used in this system.

### **Storage allocation strategies**

After describing the list structure, now let us see how a request for a memory block of given size can be serviced. Following are various known strategies:

- (a) First-fit allocation
- (b) Best-fit allocation
- (c) Worst-fit allocation
- (d) Next-fit allocation.

Let us discuss all these allocation strategies assuming that the memory system has to serve a request for a block of size  $N$ .

**First-fit storage allocation.** This is the most simplest storage allocation strategy. Here the list of available storages will be searched and as soon as a free storage block of size  $\geq N$  will be found pointer of that block will be sent to the calling program after retaining the residue space. Thus, for example, for a block of size 2 K, if the first-fit strategy found a block of 7 K, then after retaining a storage block of size 5 K, 2 K memory will be sent to the caller.



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.

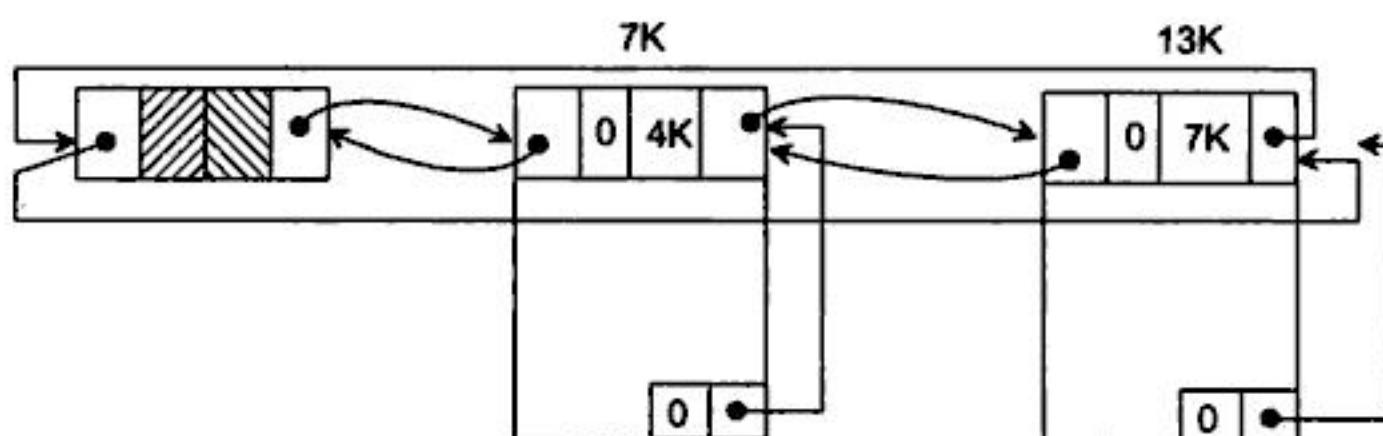
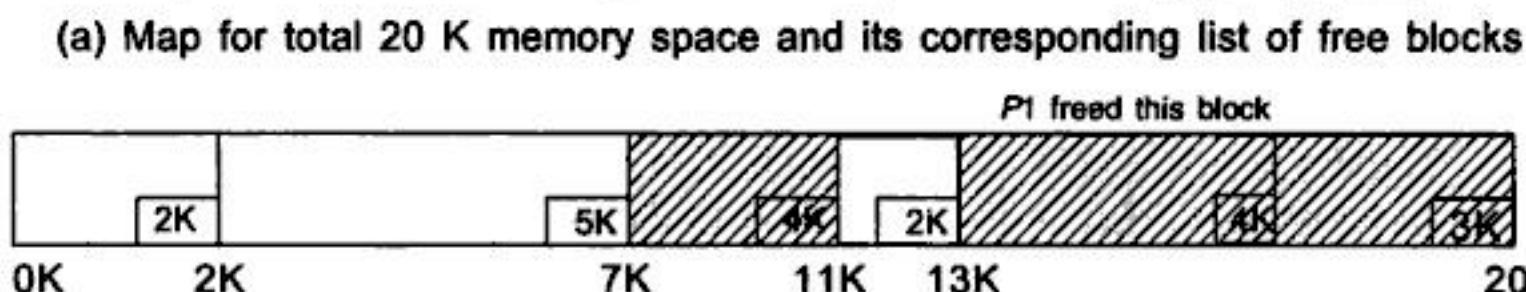
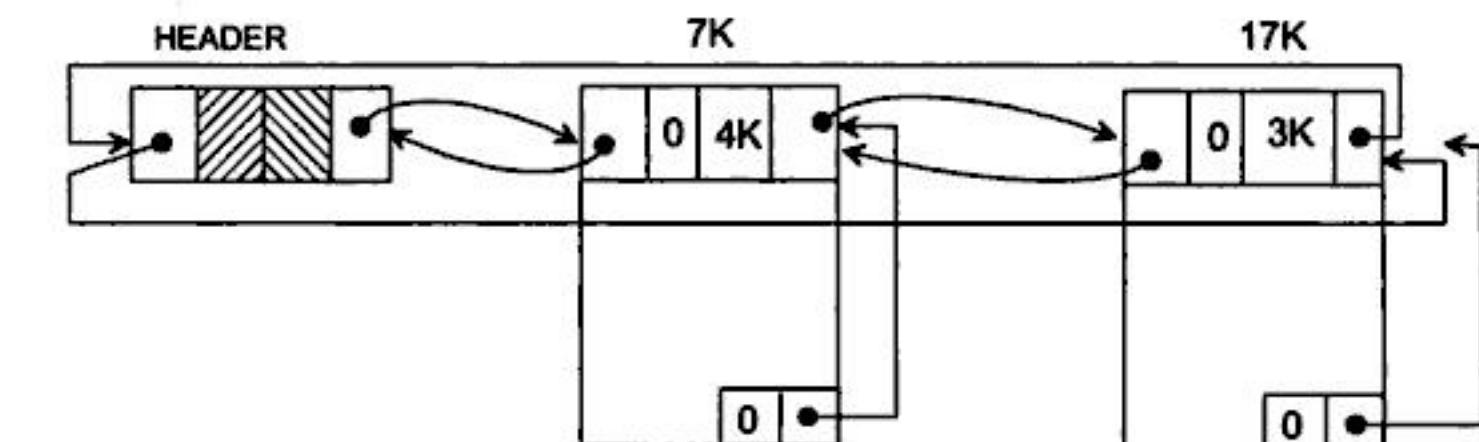
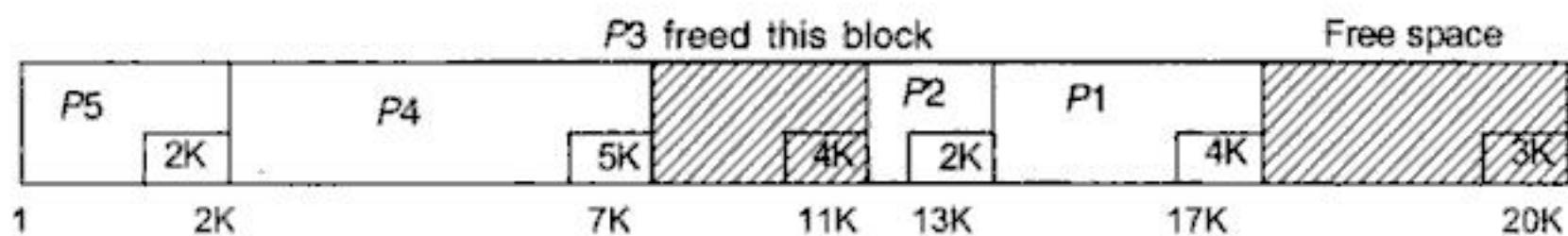
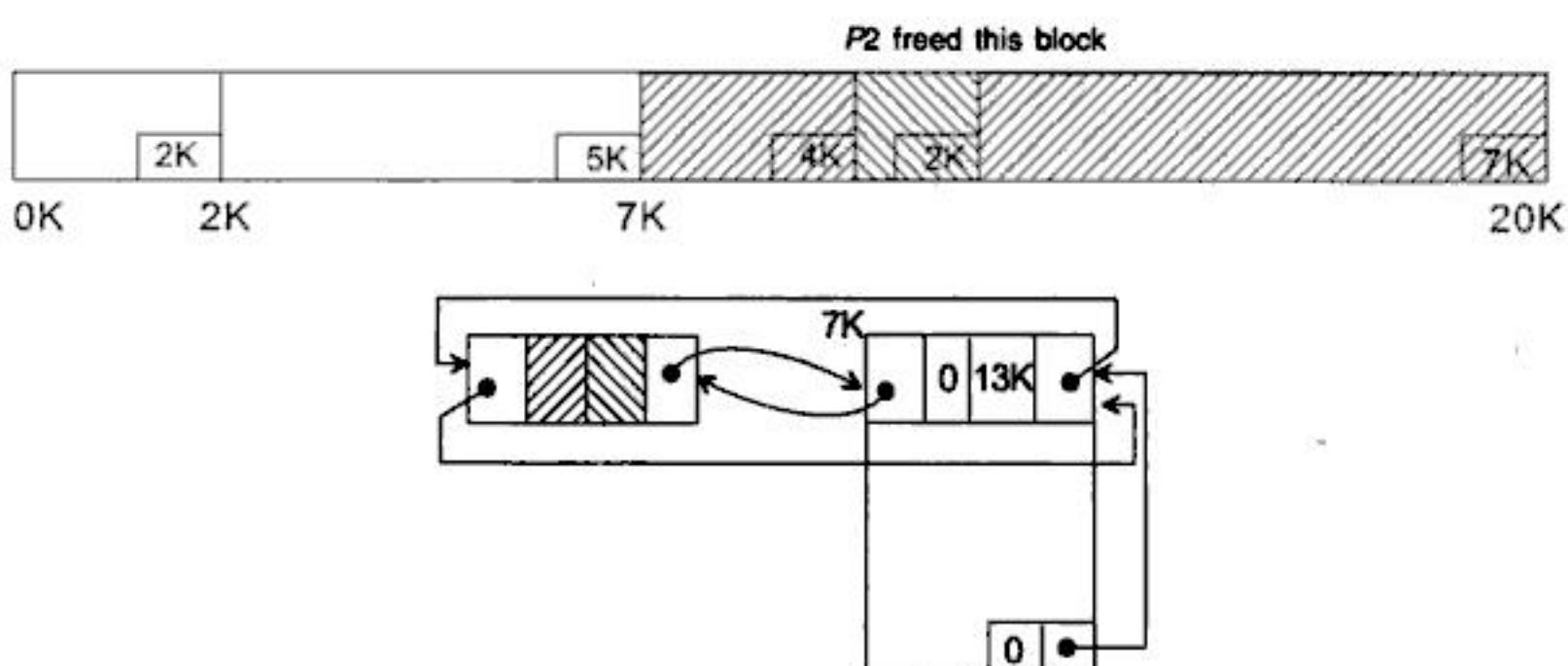
(b) Free storage list when  $P_1$  releases the block of 4 K(c) Free storage space when  $P_2$  releases a block of size 2 K

Fig. 3.26 De-allocation of blocks in boundary tag system.



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.

```

** Append new free block after left adjacent block and right adjacent block after new
block**
1. ptr1 = (PTR - 1).UPLINK //Pointer to the left adjacent block
2. ptr2 = ptr1.RLINK //Pointer to the right adjacent block
3. ptr3 = ptr2.RLINK
4. ptr1.RLINK = ptr3 //Change of pointer as shown (1) in Figure 3.27(d)
5. ptr3.LLINK = ptr1 //Change of pointer as shown (2) in Figure 3.27(d)
6. ptr1.SIZE = ptr1.SIZE + n + ptr2.SIZE
 //Change the size of the newly combined block
7. (ptr1 + ptr1.SIZE - 1).TAG = 0 //Set the TAG field of the newly combined block
8. (ptr1 + ptr1.SIZE - 1).UPLINK = ptr1 //Set the UPLINK field as shown (3) in Figure 3.27(d)
6. Stop

```

#### Assignment 3.9

1. What modification you would suggest in order to make the algorithm GETNODE\_BTS\_NEXTFIT as GETNODE\_BTS\_FIRSTFIT based on the first-fit allocation strategy?
2. Write an algorithm for availing a node based on the best-fit storage allocation strategy.
3. Procedure RETURNNODE\_BTS maintains a list of free storages. For a given request, this list is to be searched from header node whose pointer is AVAIL. One modification is suggested as “Recently freed block is for the consideration of next service”. What necessary change you should incorporate in the procedure RETURNNODE\_BTS and GETNODE\_BTS to attain the suggested modification?

## 3.10 BUDDY SYSTEM

So far we have discussed, the block sizes are either fixed or completely arbitrary. *Buddy system* is the another storage management system which restricts the sizes of blocks to some fixed set of sizes. These blocks of restricted sizes are maintained in a linked list. Whenever a request for a block of size  $N$  comes, the number  $M$ , the smallest of the fixed sizes but equal to or larger than  $N$ , is determined, and a block of size  $M$  is allocated, if available on the list. If a block of size  $M$  is not available, then a larger block, if available, is split into two sub-blocks (known as *buddies*), each of them are also of fixed sizes, and this process is repeated until a block of size  $M$  is produced.

Buddy system specifies the restricted sizes as  $F_0, F_1, \dots, F_{\text{MAX}}$  for blocks according to some pattern. One principle for the specification of block sizes is the use of recurrence relation:

$$F_n = F_{n-1} + F_{n-k}, k \leq n \leq \text{MAX}$$

for a given  $k$  and  $\text{MAX}$

$$F_0 = C_0, F_1 = C_1, \dots, F_{k-1} = C_{k-1}$$

with initial conditions.

For example, if  $k = 1$  and  $F_0 = 8$ , then the block sizes are 8, 16, 32, 64, 128, .... That is, the block sizes are successive powers of 2; and the buddy system based on such fixed sizes is called *binary buddy system*. Another buddy system, which restricts the fixed sizes to the Fibonacci



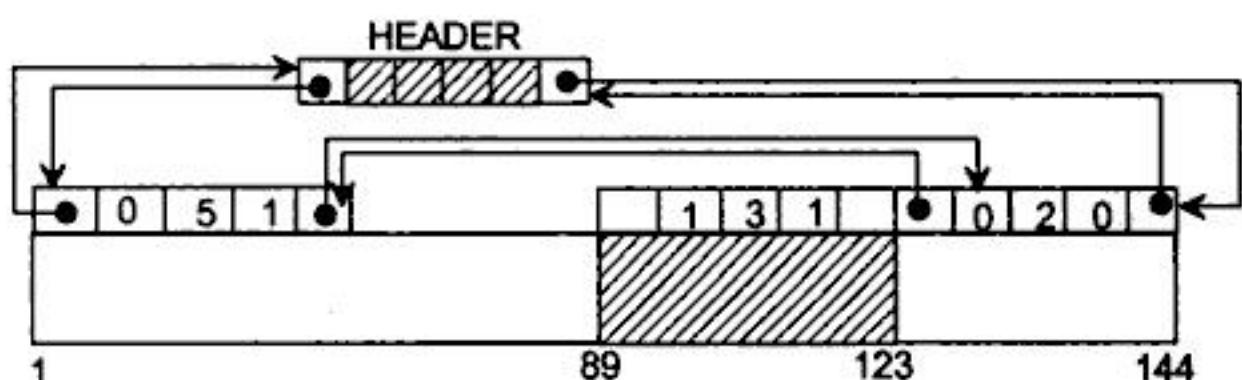
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



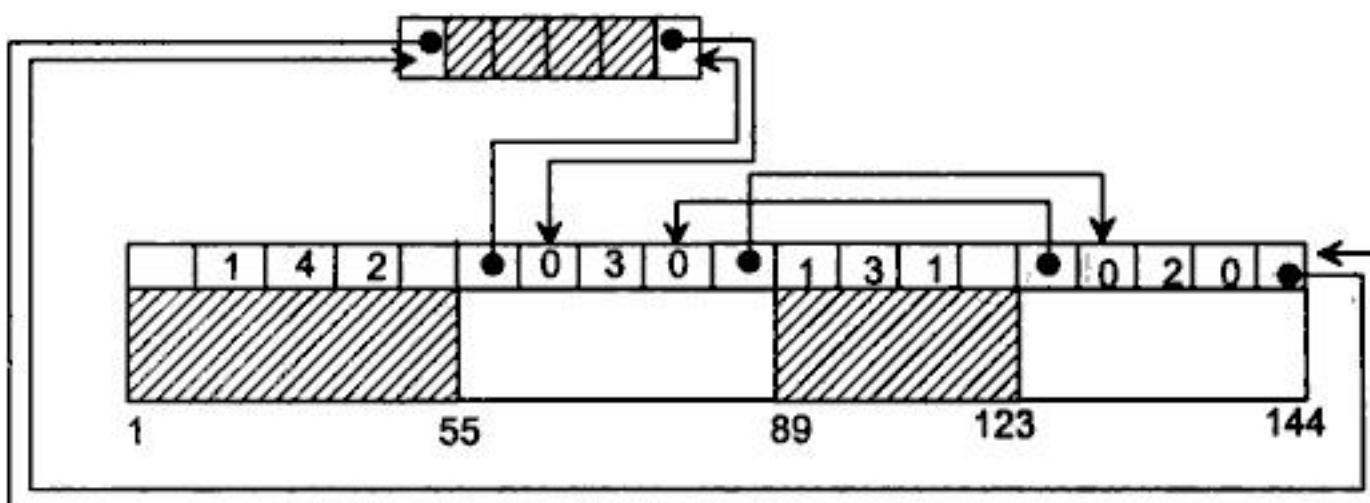
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



(a) Initial list of free blocks in buddy system, after allocation of block of size 34 from the main block



(b) After allocating the next request for a block of size 50

**Fig. 3.30 Allocation strategy in buddy system.**

The algorithm **GETNODE\_BUDDY\_BESTFIT** is an implementation plan for the allocation of a block in Buddy system. We assume the free blocks are maintained in a circularly double linked list whose pointer to the header node is known from **AVAIL**. We will maintain an array  $F[0, \dots, MAX]$ , which records the block sizes  $F_i$ ,  $0 \leq i \leq MAX$ , as determined by the recurrence relation. We will also assume our memory reference is from 1 to  $F[MAX]$ .

**Algorithm GETNODE\_BUDDY\_BESTFIT( $N$ )**

**Input:**  $N$  be the size of the block under request.

**Output:** Return a pointer of the block if available else an error message.

**Data structure:** Linked list structure for buddy system with **AVAIL** is the header to the list.  $F[0, \dots, MAX]$  is an array containing the size of the blocks.

**Steps:**

1. If ( $N > F[MAX]$ ) then
  1. Print "Request is too big: Unable to allocate memory"
  2. Exit
2. EndIf
3.  $i = 0$  //To determine the size of the block
4. While ( $N > F[i]$ ) do
  1.  $i = i + 1$
5. EndWhile
6.  $size = i$  //This is the value of the SIZE field of the required block
7.  $flag = 0$  //Indicates that the desired block is yet to be allocated



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.

3. Case: (ptr1 = AVAIL) and (ptr2 = AVAIL) //List is empty, no free blocks
  1. PTR.LLINK = AVAIL
  2. PTR.RLINK = AVAIL
  3. AVAIL.LLINK = PTR
  4. AVAIL.RLINK = PTR
  5. Exit //Insertion is completed
4. Case: (ptr1 = ptr2) and (ptr1 ≠ AVAIL) //List contains a single block
  1. If (PTR < ptr1) then //Insert as left block
    1. PTR.RLINK = ptr1
    2. ptr1.LLINK = PTR
    3. AVAIL.RLINK = PTR
    4. PTR.LLINK = AVAIL
    5. Exit //Insertion is completed
  2. Else //Insert as right block
    1. ptr1.RLINK = PTR
    2. PTR.LLINK = ptr1
    3. AVAIL.LLINK = ptr1
    4. PTR.RLINK = AVAIL
    5. Exit //Insertion is completed
5. Case: (ptr1 ≠ ptr2)
  1. While (PTR > ptr1) do //Move to the predecessor block
    1. ptr1 = ptr1.RLINK
    2. EndWhile
  3. While (PTR < ptr2) do //Move to the successor block
    1. ptr2 = ptr2.LLINK
    2. EndWhile
  5. If (PTR < ptr1) then //Insert at front
    1. PTR.RLINK = ptr1
    2. ptr1.LLINK = ptr1
    3. PTR.LLINK = AVAIL
    4. AVAIL.RLINK = PTR
  6. Else
    1. If (PTR > ptr2) then //Insert at end
      1. PTR.LLINK = ptr2
      2. ptr2.RLINK = PTR
      3. PTR.RLINK = AVAIL
      4. AVAIL.LLINK = PTR
    2. Else //Insert between ptr1 and ptr2. ptr2 is now left //to PTR and ptr1 is right to PTR
      1. PTR.RLINK = ptr2
      2. ptr2.LLINK = PTR
      3. PTR.LLINK = ptr1
      4. ptr1.RLINK = PTR
    3. EndIf
  7. EndIf
6. Stop



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.



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.

## 4.4 OPERATIONS ON STACKS

Basic operations required to manipulate a stack are:

**PUSH :** To insert an item into the stack

**POP :** To remove an item from a stack

**STATUS :** To know the present state of a stack

Let us define all these operations of stack. First we will consider the above mentioned operations for a stack represented with an array.

### Algorithm PUSH\_A(ITEM)

**Input:** The new item ITEM to be pushed onto it.

**Output:** A stack with newly pushed ITEM at the TOP position.

**Data structure:** An array A with TOP as the pointer.

#### Steps:

1. If  $\text{TOP} \geq \text{SIZE}$  then
  1. Print "Stack is full"
2. Else
  1.  $\text{TOP} = \text{TOP} + 1$
  2.  $\text{A}[\text{TOP}] = \text{ITEM}$
3. EndIf
4. Stop

Here, we have assumed that, array index varies from 1 to SIZE and TOP points the location of the current top-most item in the stack. Following operation POP\_A defines POP an item from a stack which is represented using an array A.

### Algorithm POP\_A()

**Input:** A stack with elements.

**Output:** Removes an ITEM from the top of the stack if it is not empty.

**Data structure:** An array A with TOP as the pointer.

#### Steps:

1. If  $\text{TOP} < 1$ 
  1. Print "Stack is empty"
2. Else
  1.  $\text{ITEM} = \text{A}[\text{TOP}]$
  2.  $\text{TOP} = \text{TOP} - 1$
3. EndIf
4. Stop

Next operation is to test the various states of a stack like whether it is full or empty, how many items are right now in it, and read the current element at the top without removing it.

### Algorithm STATUS\_A()

**Input:** A stack with elements.

**Output:** States whether it is empty or full, available free space and item at TOP.

**Data structure:** An array A with TOP as the pointer.



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.



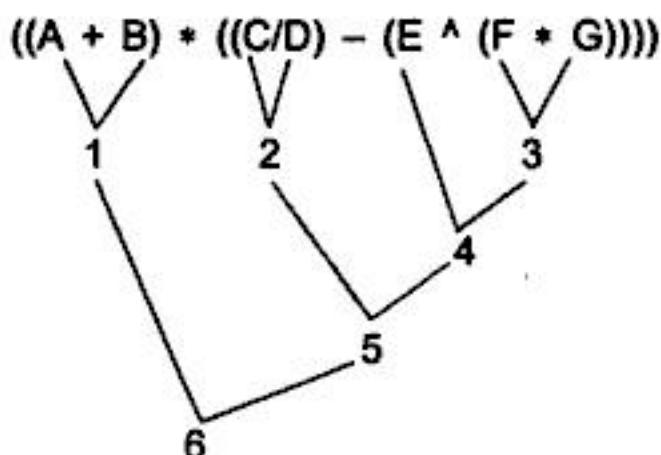
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Note that, the above rules for precedence and associativity vary from one programming language to another.

Another way of fixing the order of evaluation is to fully parenthesize the expression; this allows one to override the rule for precedence and associativity. Following is the parenthesized version of an expression:

Input:  $((A + B) * ((C/D) - (E \wedge (F * G))))$   
(A fully parenthesized expression).

With these parenthesization, the innermost parenthesis part (called subexpression) will be evaluated first, then the next innermost and so on; such a sequence is shown as below:



Whatever the way we specify the order of evaluations, the problem is that we must scan the expression from left to right repeatedly, this process, hence inefficient because of the repeated scanning that must be done. Another problem is that how compiler can generate correct code for a given expression. The last problem mainly occurs for a partially parenthesized expression. These problems can be solved with the following two steps:

1. Conversion of a given expression into a special notation
2. Evaluation/production of object code using stack.

### **Notations for arithmetic expressions**

There are three notations to represent an arithmetic expression, viz. infix, prefix and postfix (or suffix). The conventional way of writing an expression is called infix. For example,

$A + B, \quad C - D, \quad E * F, \quad G/H$ , etc.

Here, the notation is

$\langle \text{operand} \rangle \langle \text{operator} \rangle \langle \text{operand} \rangle$ .

This is called *infix* because the operators come in between the operands. The *prefix* notation, on the other hand, uses the convention:

$\langle \text{operator} \rangle \langle \text{operand} \rangle \langle \text{operand} \rangle$

Here, operators come before the operands. Following are simple expressions in prefix notation:

$+AB, \quad -CD, \quad *EF, \quad /GH$ , etc.

This notation was introduced by Polish mathematician Jan Lukasiewicz and hence also termed



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.

| <i>Read symbol</i> | <i>Stack</i> | <i>Output</i>       |
|--------------------|--------------|---------------------|
| Initial            | (            |                     |
| 1                  | ((           |                     |
| 2                  | ((           | A                   |
| 3                  | ((+          | A                   |
| 4                  | ((+          | AB                  |
| 5                  | (            | AB+                 |
| 6                  | (^           | AB+                 |
| 7                  | (^           | AB + C              |
| 8                  | ( -          | AB + C ^            |
| 9                  | ( - (        | AB + C ^            |
| 10                 | ( - (        | AB + C ^ D          |
| 11                 | ( - ( *      | AB + C ^ D          |
| 12                 | ( - ( *      | AB + C ^ DE         |
| 13                 | ( -          | AB + C ^ DE *       |
| 14                 | ( - /        | AB + C ^ DE *       |
| 15                 | ( - /        | AB + C ^ DE * F     |
| 16                 |              | AB + C ^ DE * F / - |

**Output: A B + C ^ DE \* F / - (postfix form)**

The above procedure assumes the input infix expressions are according to the right syntax. So, if the input expression is not correct, its postfix form will not be correct. Extending the same idea, we can incorporate relational, boolean and unary operators in the above procedure.

#### Assignment 4.2

- (a) Verify the algorithm INFIX\_TO\_POSTFIX for producing the postfix notations of the following as infix notations.
  - (i)  $(A + B) ^ C ^ D * E - F / G$
  - (ii)  $A + * B - C$
  - (iii)  $A + B ) * ( C - D )$
  - (iv)  $A * ( + B ) - C / D$
- (b) Modify the algorithm INFIX\_TO\_POSTFIX to convert an infix expression containing boolean operators and relational operators and unary operators to their equivalent postfix form.
- (c) For the algorithm INFIX\_TO\_POSTFIX, we assume the input expression to be correct. That is, there is no error checking. But the input expression may not be well formed always: expressions having two operands together or two binary operator together or unmatched parentheses are some common mistakes. Write a procedure, so that it will check for the correct expression at the time of producing postfix form.



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.

|   |      |                                                               |                           |
|---|------|---------------------------------------------------------------|---------------------------|
| * | T2   | $x = C, y = T1$<br>PRODUCE_CODE(T1, C, 'MUL', T2)<br>PUSH(T2) | LDA T1<br>MUL C<br>STA T2 |
| D | T2 D | PUSH(D)                                                       |                           |
| / | T3   | $x = D, y = T2$<br>PRODUCE_CODE(T2, D, 'DIV', T3)<br>PUSH(T3) | LDA T2<br>DIV D<br>STA T3 |
| # | T3   | Stop                                                          |                           |

#### Assignment 4.3

- (a) Consider expressions which contains relational and boolean operators. Formulate the precedence values required for these operators to convert such an expression to reverse Polish notation.
- (b) Modify the algorithm POSTFIX\_TO\_CODE so that it will also handle the unary operators.
- (c) Modify the algorithm POSTFIX\_TO\_CODE for the six relational operators  $<$ ,  $>$ ,  $\leq$ ,  $\geq$ ,  $\neq$ ,  $=$ ) and four boolean operators (AND, OR, NOT, XOR).

#### 4.5.2 Code Generation for Stack Machines

In Section 4.5.1, we have discussed the generation of code for a given arithmetic expression in postfix form. The codes are of the type called *single address* codes. These codes are for those machine which maintains a number of registers; registers are to store the temporaries. One problem using machines which have very limited number of registers is how to handle the storage of intermediate results.

Some machine architectures are known which use a stack instead of registers to store temporaries or intermediate results; obviously stack size in this case should be adequate enough to handle a large expression. Here, we will present the description of a simple hypothetical machine to illustrate the concept of code generation for postfix form of arithmetic expressions.

Let us assume the following set of instructions (given in mnemonic forms) of the stack machine.

**PUSH <name>** To load from memory onto stack. This instruction loads an operand from the memory location named <name> and places the content of <name> into the stack.

**POP <name>** To store the top element of the stack in memory. The content of the top of the stack are removed and stored in the memory location referenced by <name>.

Let us assume that our machine performs the following arithmetic operations only:

ADD, SUB, MUL, DIV

Assume the operations can be stated with zero-address operation code. Operands are explicitly mentioned as top-two elements in the stack. The operations are as stated below:



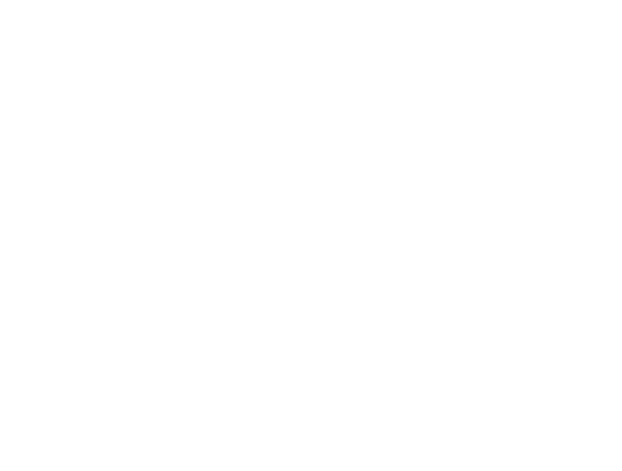
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



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.

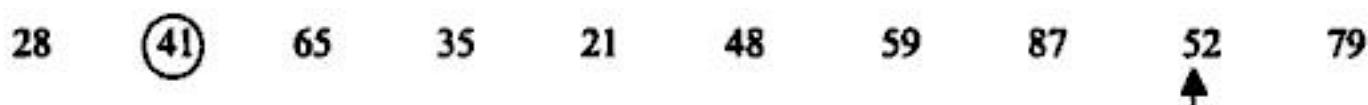


You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

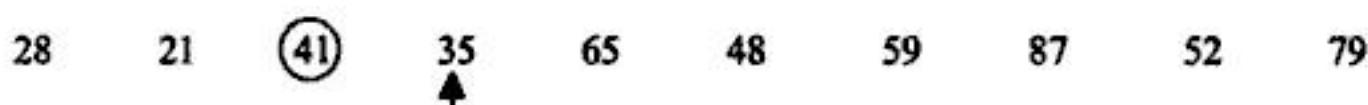
Now, pivot element is at right side. Again compare this with the elements at left side and next to the element just swapped. Again swap will occur if comparison yields that they are not in order. With this, the next situation appears as:



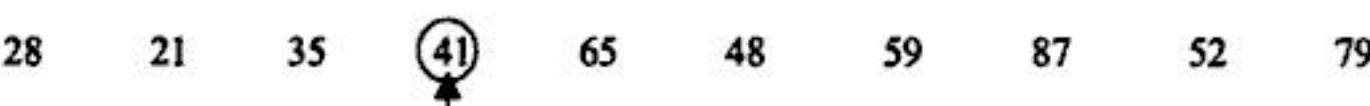
Repeatedly applying the above steps one can get the following observation. Comparison with the pivot will shift the right marker from 52 to 87, 87 to 59, 59 to 48 and 48 to 21 at last when pivot element is found larger than 21. So swap will take place:



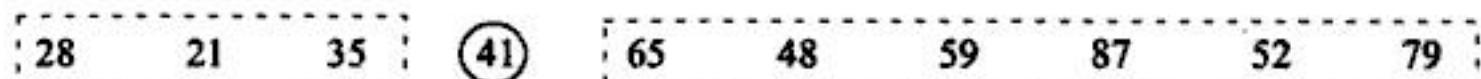
Compare from left, we get the following change:



Another comparison from right yields:



Now, we should stop as 41 is now placed in its final position, and observe that we get two sub-lists one containing all the element at the left of 41 which are smaller than 41 and another list at the right of 41 containing elements which are larger than 41.



After getting two sub-lists we have to recursively apply the same procedure on each sub-list. A sub-list containing no elements or single elements is the terminal stage of the repetition. Now, here is the application of stack. Out of two sub-lists, one sub-list has to be pushed onto the stack before sorting the other list. When taking care of this list, pop the next list to be considered and procedure will continue till the stack is empty.

The quick sort algorithm now can be defined recursively as below: Let  $L$  be the original list and  $FL, EL$  are location of front and end elements of  $L$  respectively.

#### QUICK\_SORT( $FL, EL$ )

1. DIVIDE( $L, L_1, L_2$ ) //Divide  $L$  into two sub-lists  $L_1, L_2$
2. If  $\text{SIZEOF}(L_1) > 1$  //If the left-most list  $L_1$  contains more than one element
  1. QUICK\_SORT( $FL_1, EL_1$ ) //Apply quick sort on  $L_1$
3. If  $\text{SIZEOF}(L_2) > 1$  //If the right-most list  $L_2$  contains more than one element
  1. QUICK\_SORT( $FL_2, EL_2$ )
4. Stop

Note that a sub-list can be identified by the location of its two extreme elements. Here,  $\text{SIZEOF}(L)$  is assumed to determine the number of elements in  $L$ .



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.

Now both the lists contain only one element so no more push for them. Top pointer of the stack is NULL. So quick sort has reached to its end. The list in sorted order is shown as:

|   | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 |
|---|----|----|----|----|----|----|----|----|----|----|
| A | 21 | 28 | 35 | 41 | 48 | 52 | 59 | 65 | 79 | 87 |

Fig. 4.7 Illustration of recursive execution of quick sort using stacks.

In the above algorithm QUICK\_SORT, we assume PUSH(FL, EL) as to push FL and EL into FRONT and END of stacks respectively. Similarly, POP(FL, EL) is to POP the items from two stacks FRONT and END and they are stored as FL and EL, respectively. The details of quick sort method is illustrated through an example as shown in Figure 4.7. Note that element with dotted circle indicates that the element is placed at final position. The circled element is the present pivot element. A 'X' in the stack pointer position indicates the deletion of the entry that is a POP.

#### 4.5.6 Tower of Hanoi Problem

Another complex recursive problem is the tower of Hanoi problem. This problem has a historical basis in the ritual of ancient Vietnam. The problem can be described as below:

Suppose, there are three pillars *A*, *B* and *C*. There are *N* discs of decreasing size so that no two discs are of the same size. Initially all the discs are stacked on one pillar in their decreasing order of size. Let this pillar be *A*. Other two pillars are empty. The problem is to move all the discs from one pillar to other using third pillar as auxiliary so that

- Only one disc may be moved at a time.
- A disc may be moved from any pillar to another.
- At no time can a larger disc be placed on a smaller disc.

Figure 4.8 represents the initial and final stages of the tower of Hanoi problem for *N* = 5 discs. Solution of this problem can be stated recursively as follows:

Move *N* discs from pillar *A* to *C* via the pillar *B* means

Move first (*N* - 1) discs from pillar *A* to *B*

Move the disc from pillar *A* to *C*

Move all (*N* - 1) discs from pillar *B* to *C*.

The above solution can be described by writing a function say, MOVE(*N*, ORG, INT, DES), where *N* is the number of discs, ORG, INT and DES are origin (from pillar), intermediate (via pillar) and destination (to pillar) respectively.

Thus, with this notation, MOVE(5, *X*, *Z*, *Y*) means move 5 discs from pillar *X* to pillar *Y* taking the intermediate pillar as *Z*. With this definition, the problem can be solved with recursion as:



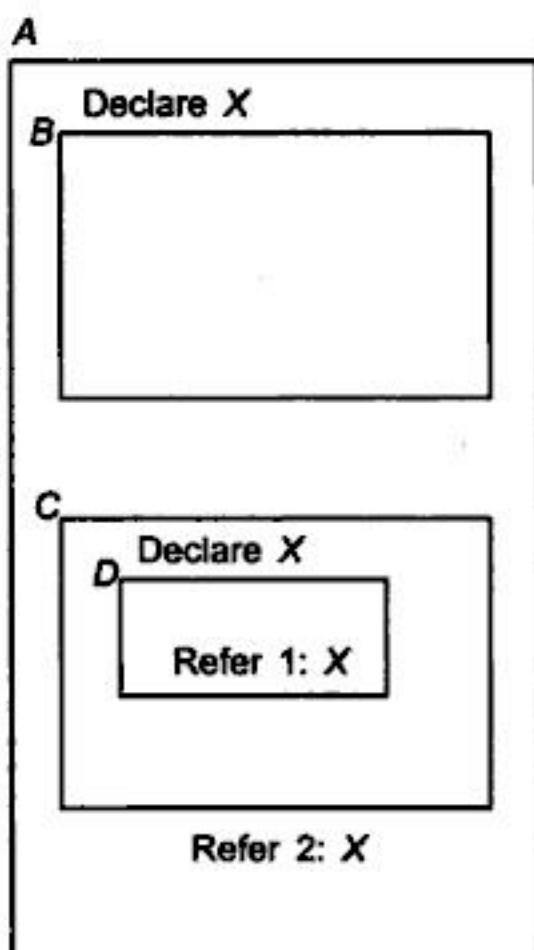
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



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.



**Fig. 4.10** A block structured program.

The static scope rule can be defined as below:

- The scope of a variable declared in a particular block consists of that block, exclusive of any block nested within it that declares the same identifier.
- If a variable is not declared within a block, then it obtains the declaration from the next outer block; if not there then next outer block and so on until a declaration is found. Such a rule is known as ‘most closely nested rule’.

Thus, with this rule, reference of a variable at Refer 2 (see Figure 4.10) be resolved from the declaration in block A, whereas reference at Refer 1 will be resolved from the declaration in block C.

In dynamic scope rule, on the other hand, reference of an identifier is resolved during the execution of the program and same variable name may be defined at several point within the same program, that is, variable name may change its definition as execution proceeds. This is why, dynamic scope rule is also alternatively termed as ‘fluid binding’. This rule is stated as follows.

The declaration of a variable is referred from the most recently occurring and still active definition of the name during execution of the program.

For an example, consider the program structure as shown in Figure 4.11. P is the main program. During its execution, at a certain point, procedure ‘call A’ occurs. Here procedure A is in currently active block P. So, for any reference, X in A will be obtained from declaration in P.

For other procedure ‘call2 C’, as it itself has declaration for X (Declare 3) so any reference of X will be resolved from that only. Now, suppose B is on execution. B in turn call Procedure A (as ‘call4 A’). Here, reference of X will be obtained from Declare2 as B is the most recently occurring block. Thus, for a given reference of X in A, its declaration once resolved from Declare1 and another from Declare2 in the same program.



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.

```
01 Program MAIN
02 A, B, C: integer
03 Procedure Q
04 Begin
05 A = A + 2
06 C = C + 2
07 End
08 Procedure R
09 C: integer
10 Begin
11 C = 2;
12 call Q;
13 B = A + B
14 End
15 Procedure S
16 B, C: integer;
17 Procedure Q
18 Begin
19 A = A + 1
20 C = C + 1
21 End
22 Begin
23 B = 3
24 C = 1
25 call Q
26 call R
27 End
28 Begin
29 A = 1
30 B = 2
31 C = 3
32 call R
33 call S
34 End
```

Fig. 4.14 A block structured program.



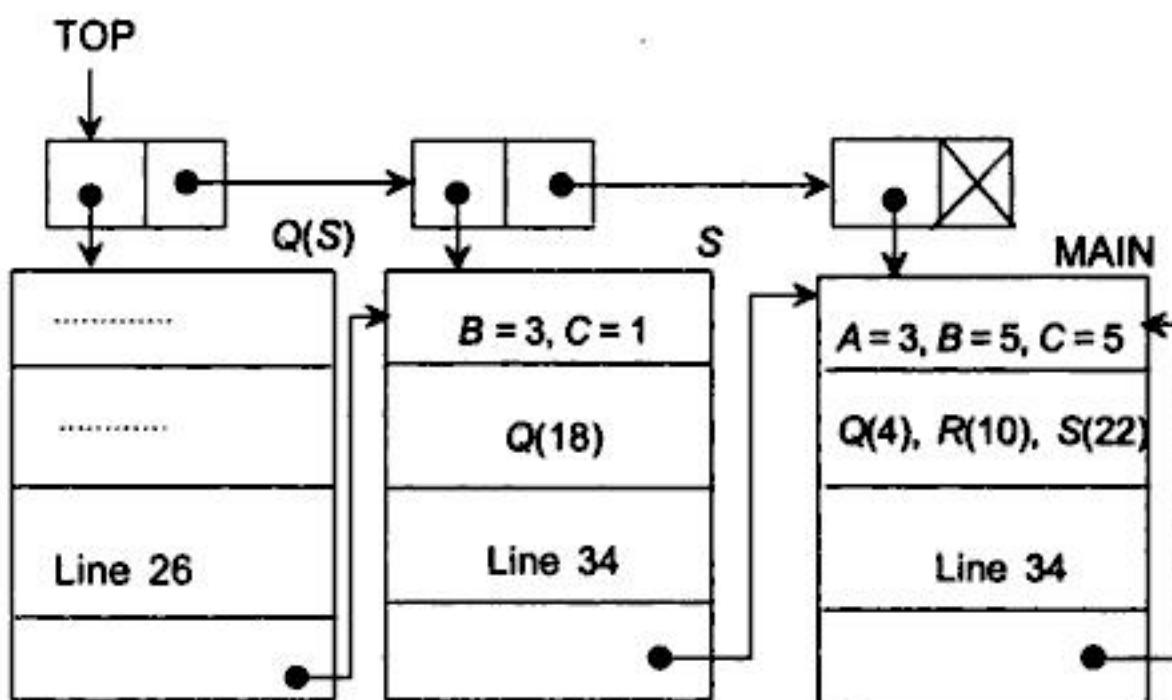
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



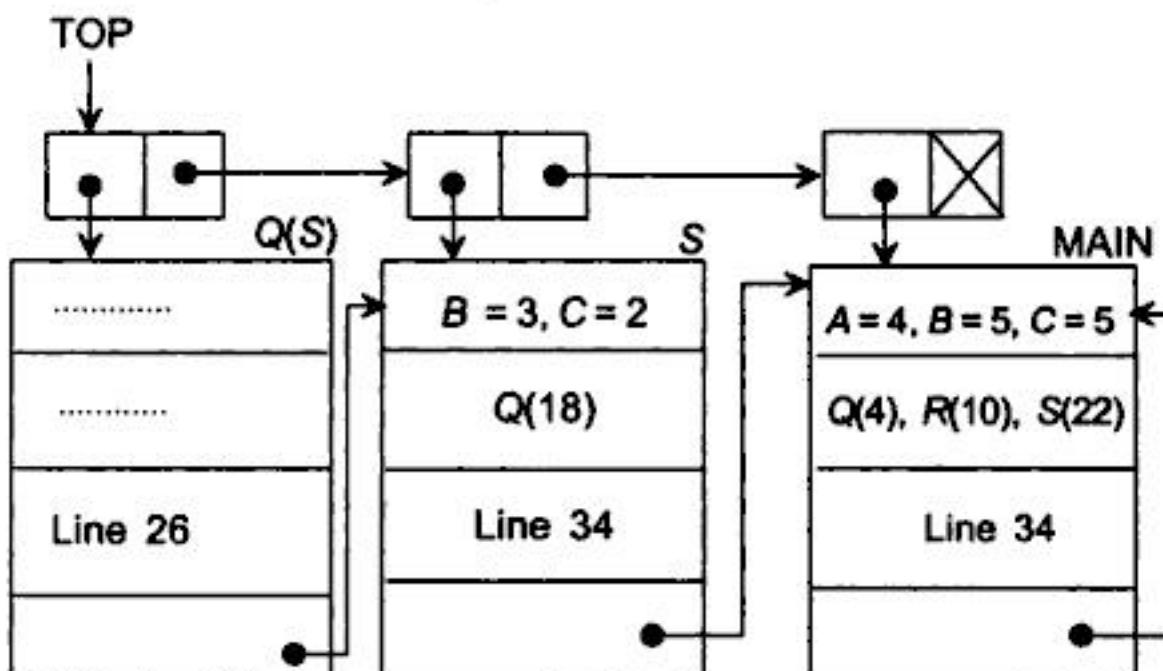
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



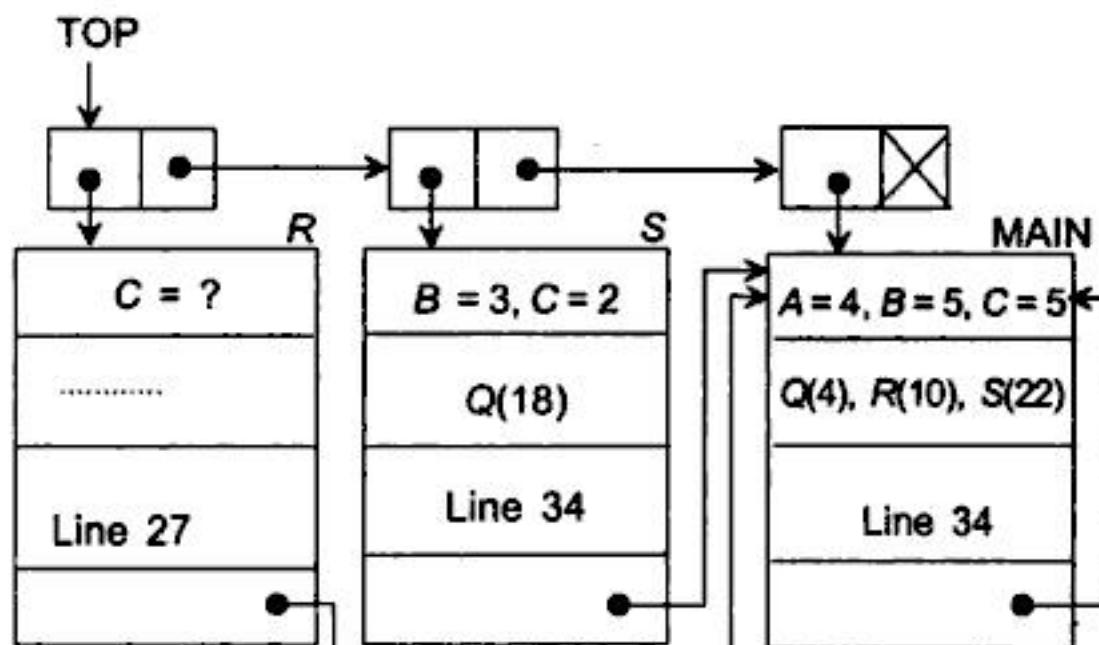
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



(f) Procedure S begins its execution at line 22 and when control reaches at line 25 it invokes the procedure Q. The reference of Q is resolved from the current activation record, that is, of S and that activation record of Q is then pushed onto the stack.



(g) Execution of Q is started and references of A and C at lines 19 and 20 are resolved from S and MAIN respectively.



(h) When Q finished its execution controls returns to line 26, its activation is then popped and procedure R is invoked. This R is resolved from the activation record of MAIN.

Fig. 4.17 Continued.



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.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

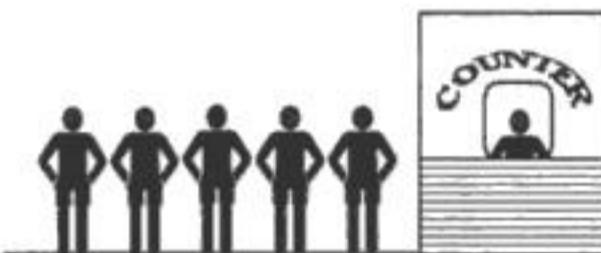
# 5 Queues

## 5.1 INTRODUCTION

*Queue* is a simple but very powerful data structure to solve numerous computer applications. Like stacks, queues are also useful to solve various system programs. Let us visit some simple applications of queues in our everyday life as well as in computer science before going to study this data structure.

### ***Queuing in front of a counter***

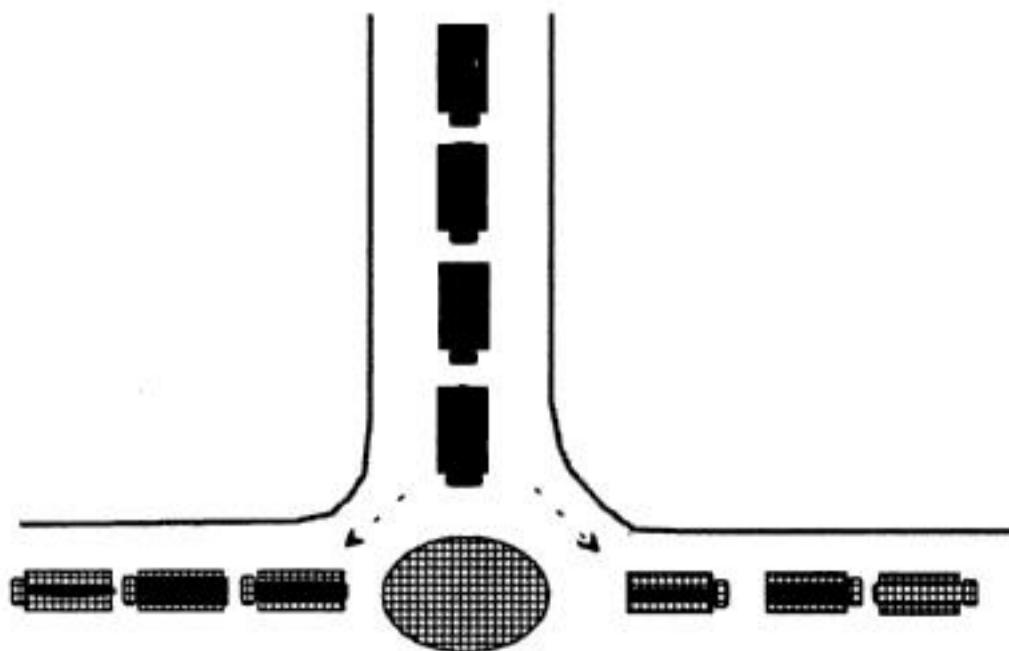
Suppose there are a number of customer in front of a counter to get service (say, to collect tickets or to withdraw/deposit money in a teller of a bank), Figure 5.1(a). The customers are forming a queue and they will be served in the order they arrived, that is, a customer who comes first will be served first.



**Fig. 5.1(a) Queue of customers.**

### ***Traffic control at a turning point***

Suppose there is a turning point in a highway where the traffics has to turn, Figure 5.1(b). All



**Fig. 5.1(b) Traffic passing at a turning point.**



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.

**Steps:**

1. If (FRONT = 0) then
  1. Print "Queue is empty"
  2. Exit
2. Else
  1. ITEM =  $Q[FRONT]$  //Get the element
  2. If (FRONT = REAR)
    1. REAR = 0 //When queue contains single element
    2. FRONT = 0 //The queue becomes empty
  3. Else
    1. FRONT = FRONT + 1
    4. EndIf
  3. EndIf
  4. Stop

Let us trace the above two algorithms with a queue of size = 10. Suppose, the current state of the queue is FRONT = 8, REAR = 9. Ten operations are requested as under:

- |            |            |            |            |             |
|------------|------------|------------|------------|-------------|
| 1. DEQUEUE | 2. ENQUEUE | 3. ENQUEUE | 4. DEQUEUE | 5. DEQUEUE  |
| 6. DEQUEUE | 7. ENQUEUE | 8. ENQUEUE | 9. DEQUEUE | 10. DEQUEUE |

Figure 5.4 presents the status of the queue when these operations are carried out.

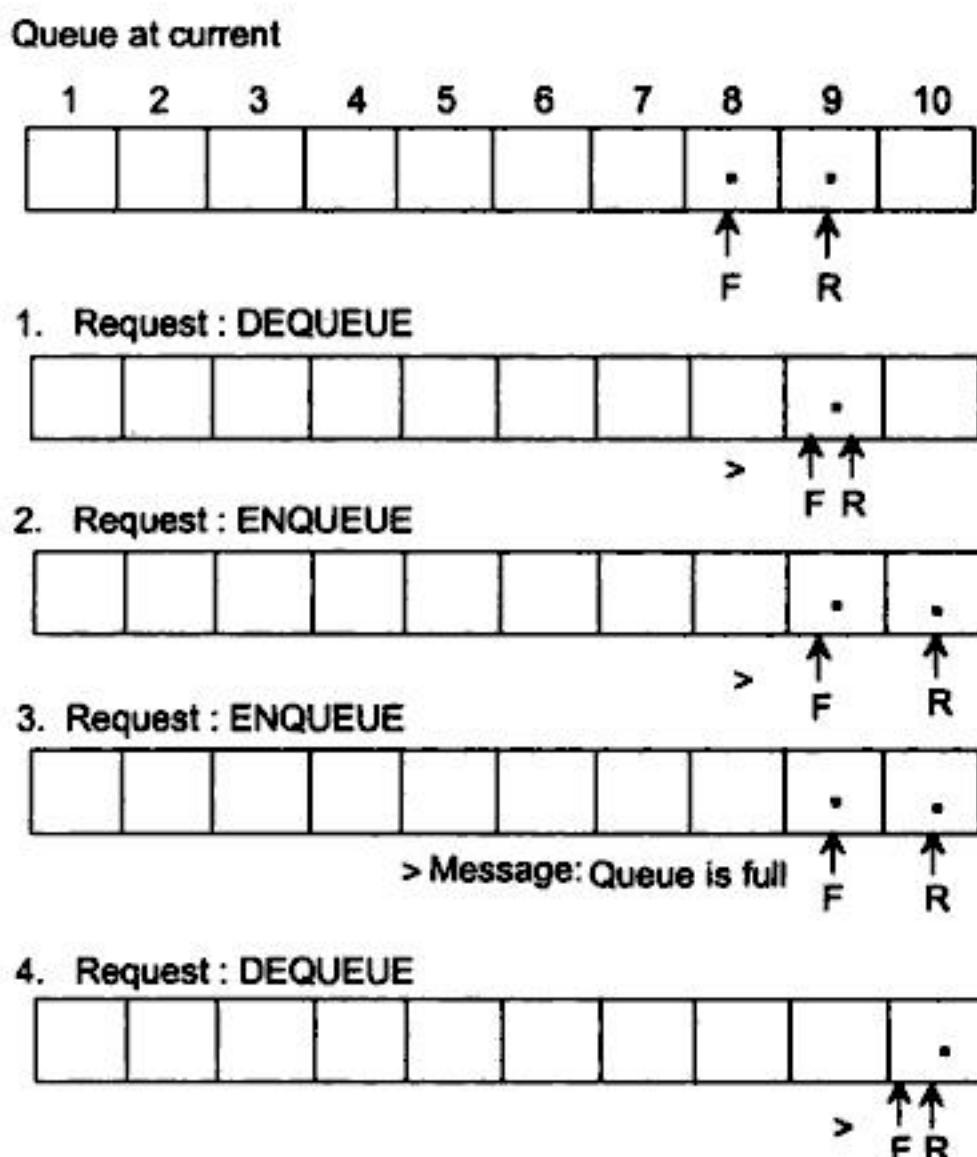


Fig. 5.4 Continued.



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.

**Steps:**

1. If (FRONT = 0) then //When queue is empty
  1. FRONT = 1
  2. REAR = 1
  3. CQ[FRONT] = ITEM
2. Else //Queue is not empty
  1. next = (REAR MOD LENGTH) + 1
  2. If (next ≠ FRONT) then //If queue is not full
    1. REAR = next
    2. CQ[REAR] = ITEM
  3. ELSE
    1. Print "Queue is full"
  4. EndIf
3. EndIf
4. Stop

**Algorithm DECQUEUE( )**

**Input:** A queue CQ with elements. Two pointers FRONT and REAR are known.

**Output:** The deleted element is ITEM if the queue is not empty.

**Data structures:** CQ is the array representation of circular queue.

**Steps:**

1. If (FRONT = 0) then
  1. Print "Queue is empty"
  2. Exit
2. Else
  1. ITEM = CQ[FRONT]
  2. If (FRONT = REAR) then //If the queue contains single element
    1. FRONT = 0
    2. REAR = 0
  3. Else
    1. FRONT = (FRONT MOD LENGTH) + 1
  4. EndIf
3. EndIf
4. Stop

In order to trace these two algorithms let us consider a circular queue of LENGTH = 4. Following operations are requested. Different states of the queue under processing these requests is illustrated in Figure 5.8.

- |                 |                 |
|-----------------|-----------------|
| 1. ENCQUEUE (A) | 2. ENCQUEUE (B) |
| 3. ENCQUEUE (C) | 4. ENCQUEUE (D) |
| 5. DECQUEUE     | 6. ENCQUEUE (E) |
| 7. DECQUEUE     | 8. ENCQUEUE (F) |
| 9. DECQUEUE     | 10. DECQUEUE    |
| 11. DECQUEUE    | 12. DECQUEUE    |

Assume that initially queue is empty, that is, FRONT = REAR = 0.



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.

**Steps:**

1. If (FRONT = 0) then
  1. Print "Deque is empty"
  2. Exit
2. Else
  1. If (FRONT = REAR) then //The deque contains single element
    1. ITEM = DQ[REAR]
    2. FRONT = REAR = 0 //Deque becomes empty
  2. Else
    1. If (REAR = 1) then //REAR is at extreme left
      1. ITEM = DQ[REAR]
      2. REAR = LENGTH
    2. Else
      1. If (REAR = LENGTH) then //REAR is at extreme right
        1. ITEM = DQ[REAR]
        2. REAR = 1
      2. Else //REAR is at an intermediate position
        1. ITEM = DQ[REAR]
        2. REAR = REAR - 1
      3. EndIf
      3. EndIf
    3. EndIf
    3. EndIf
  4. Stop

There are, however, two variations of deques known:

1. Input restricted deque, and
2. Output restricted deque.

These two types are actually intermediate between a queue and a deque. Specifically, an *input-restricted deque* is a deque which allows insertions at one end (say REAR end) only but allows deletions at both ends. Similarly, an *output-restricted deque* is a deque where deletions take place at one end (say FRONT end) only but allows insertions at both ends. Figure 5.10 represents two such variations of deques.

**Assignment 5.4**

1. Using the linked list representation of a deque, obtain the four operations PUSHQ( ), POPQ( ), INJECT( ), and EJECT( ).
2. Using the circular array representation of a deque, obtain the following:
  - (a) Insertion operation into an input-restricted deque.
  - (b) Deletion operations from an input-restricted deque.
3. Repeat problem 2 for an output-restricted deque.



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.

It is clear from Figure 5.15(a) that for each priority value a simple queue is to be maintained. An element will be added into a particular queue depending on its priority value.

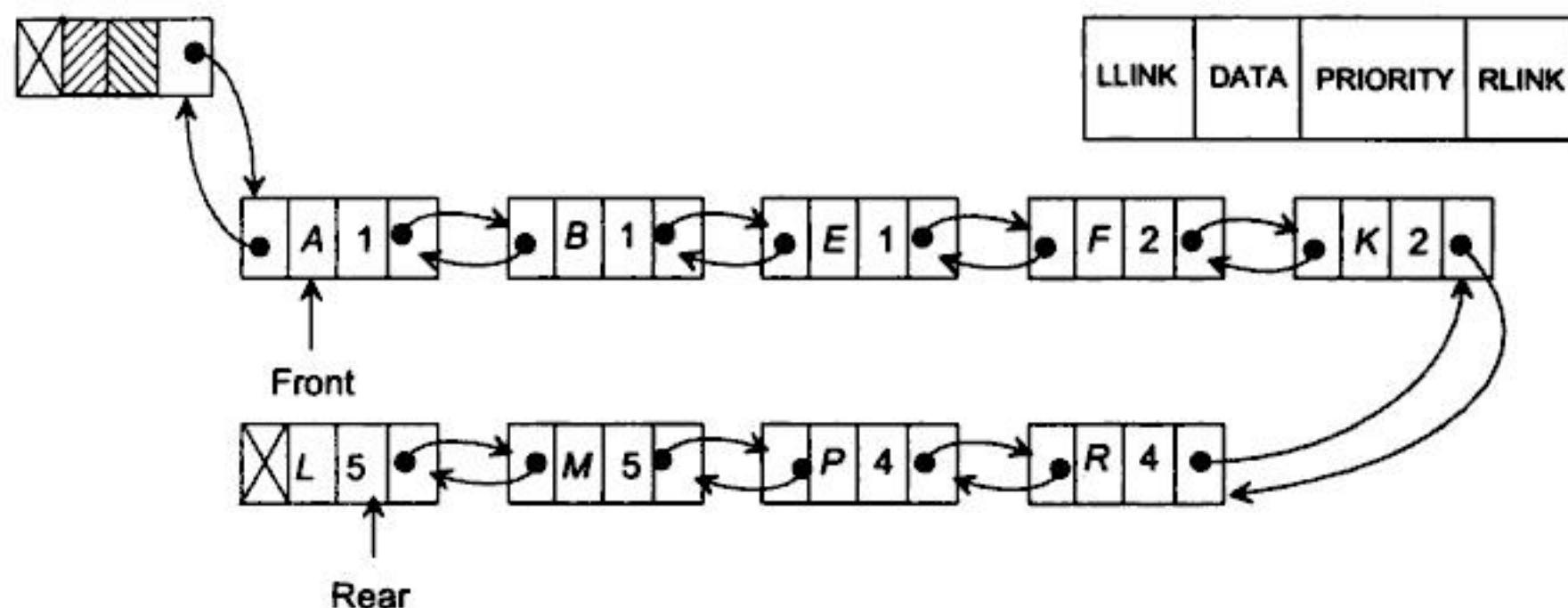
The priority queue as shown in Figure 5.15(b) is some way better than the multi-queue with multiple queues. Here one can get rid off maintaining several pointers for FRONT and REAR in several queues. Multiqueue with multiple queues has one advantage that one can have different queues of arbitrary length. In some application, it is seen that, number of occurrence of elements with some priority value is much larger than other value, thus demanding a queue of larger size.

Both the above representations are not economic from the memory utilization point of view; majority of the memory space remains vacant.

Algorithms for insertion and deletion operations for multiqueue implementation are left as exercises.

### **Linked list representation of priority queue**

This representation assumes the node structure as shown in Figure 5.16. LLINK and RLINK are two usual link fields, DATA is to store the actual content and PRIORITY is to store the priority value of the item. We will consider FRONT and REAR as two pointers pointing the first and last node in the queue, respectively. Here all the nodes are in sorted order according to the priority values of the items in the nodes. Following is an instance of priority queue:



**Fig. 5.16** Linked list representation of a priority queue.

With this structure, to delete an item having priority  $p$ , the list will be searched starting from the node under pointer REAR and the first occurring node with PRIORITY =  $p$  will be deleted. Similarly, to insert a node containing an item with priority  $p$ , the search will begin from node under pointer FRONT and node will be inserted before a node found first with priority value  $p$ , or if not found then before the node with the next priority value. The following two algorithms INSERT\_PQ and DELETE\_PQ are used to implement the insertion and deletion operations on a priority queue.

#### **Algorithm INSERT\_PQ(ITEM, P)**

**Input:** The ITEM and its priority  $P$  value of a node that is to be inserted.

**Output:** A new node inserted.

**Data structures:** Linked list structure of priority queue; HEADER as the pointer to the header.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



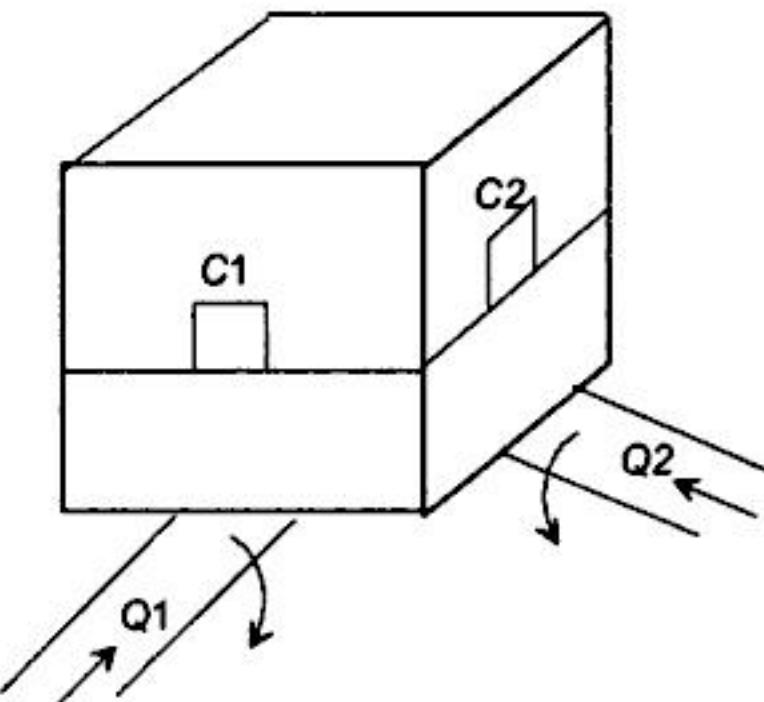
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

driven simulation, systems change its states with the change of time and in event-driven simulation, system changes its state whenever a new event arrives to the system or exits from the system.

Now let us consider a system, its model for simulation study and then application of queues in it. Consider a system as a ticket selling centre. There are two kind of tickets available, namely,  $T_1$  and  $T_2$ , which customers are to purchase. Two counters  $C_1$  and  $C_2$  are there (Figure 5.18). Also assume that time required to issue a ticket of  $T_1$  and  $T_2$  are  $t_1$  and  $t_2$  respectively. Two queues  $Q_1$  and  $Q_2$  are possible for the counters  $C_1$  and  $C_2$  respectively. With this description of the system, two models are proposed:



**Fig. 5.18** A ticket selling counter.

#### Model 1

- Any counter can issue both type of tickets.
- A customer when arrives goes to the queue which has lesser number of customers; if both are equally crowded, then to  $Q_1$ , the queue of counter  $C_1$ .

#### Model 2

- Two counters are earmarked, say,  $C_1$  for selling  $T_1$  and  $C_2$  for selling  $T_2$  only.
- A customer on arrival goes to either queue  $Q_1$  or  $Q_2$  depending on the ticket, that is, customer for  $T_1$  will be on  $Q_1$  and that for  $T_2$  will be on  $Q_2$ .

To simplify the simulation model, the underlying assumptions are made:

1. Queue lengths are infinite.
2. One customer in a queue is allowed for one ticket only.
3. Let  $\lambda_1$  and  $\lambda_2$  are the mean arrival rates of customers for ticket  $T_1$  and  $T_2$  respectively. The values for  $\lambda_1$  and  $\lambda_2$  will be provided by the system analyst.
4. Let us consider the *discrete probability distribution* (also called *Poisson distribution*) for the arrival of the customers to the centre. Poisson distribution gives a probability function

$$P(t) = 1 - e^{-\lambda t}$$

where  $P(t)$  = the probability that the next customer arrives at time  $t$ , and  $\lambda$  = the mean arrival rate. Thus, if we assume  $N$  be the total population of customers in a day, then



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.

```

3. While ($N_1 > 0$) do //Add remaining customer for ticket of type T_2
 1.a $c = \text{SETCUSTOMER1}()$
 1.b If ($l_1 \leq l_2$) then //Add to the smaller queue
 1. ENQUEUE1(c)
 2. $l_1 = l_1 + 1$
 1.c Else
 1. ENQUEUE2(c)
 2. $l_2 = l_2 + 1$
 1.d EndIf
 1.e $N_1 = N_1 - 1$
4. EndWhile
6. Else //Customer for T_2 is more than T_1 , that is,
 // $N_2 > N_1$
 1. While ($N_1 > 0$) do //One customer from T_1 type and other from
 // T_2 type
 1.a $c = \text{SETCUSTOMER1}()$
 1.b If ($l_1 < l_2$) then
 1. ENQUEUE1(c)
 2. $l_1 = l_1 + 1$
 1.c Else
 1. ENQUEUE2(c)
 2. $l_2 = l_2 + 1$
 1.d EndIf
 1.e $N_1 = N_1 - 1$
 1.f $c = \text{SETCUSTOMER2}()$
 1.g If ($l_1 < l_2$) then
 1. ENQUEUE1(c)
 2. $l_1 = l_1 + 1$
 1.h Else
 1. ENQUEUE2(c)
 2. $l_2 = l_2 + 1$
 1.i EndIf
 1.j $N_2 = N_2 - 1$
 2. EndWhile
3. While ($N_2 > 0$) do
 1.a $c = \text{SETCUSTOMER2}()$
 1.b If ($l_1 \leq l_2$) then
 1. ENQUEUE1(c)
 2. $l_1 = l_1 + 1$
 1.c Else
 1. ENQUEUE2(c)
 2. $l_2 = l_2 + 1$
 1.d EndIf
 1.e $N_2 = N_2 - 1$
7. EndWhile

```



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.

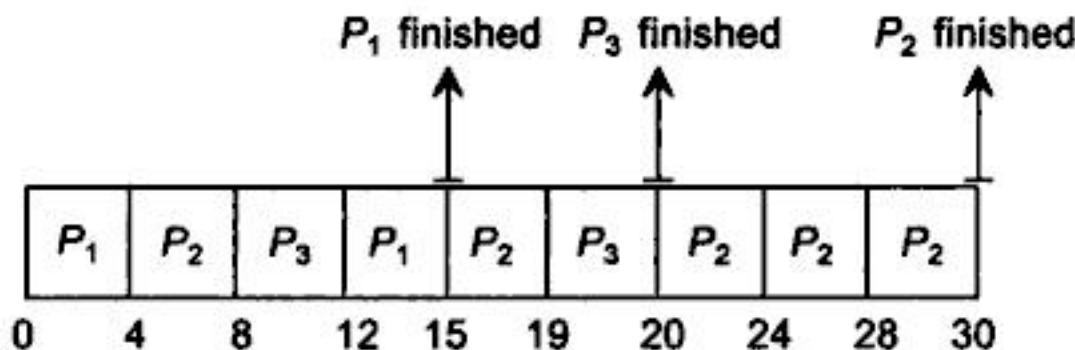
RR algorithm first decides a small unit of time, called a *time quantum* or *time slice*,  $\tau$ . A time quantum is generally from 10 to 100 milliseconds. CPU starts services with  $P_1$ .  $P_1$  gets CPU for  $\tau$  instant of time, afterwards CPU switches to  $P_2$  and so on. When CPU reaches the end of time quantum of  $P_n$  it returns to  $P_1$  and the same process will be repeated. Now, during time sharing, if a process finishes its execution before the finishing of its time quantum, the process then simply releases the CPU and the next process waiting will get the CPU immediately.

For an illustration, consider Table 5.2 for the set of processes:

**Table 5.2** Table for Process and Burst Time

| Process | Burst time |
|---------|------------|
| $P_1$   | 7          |
| $P_2$   | 18         |
| $P_3$   | 5          |

The total CPU time required is 30 unit. Let us assume a time quantum of 4 unit. The RR scheduling for this will be as shown in Figure 5.22.



**Fig. 5.22** RR scheduling.

The advantages of this kind of scheduling is reducing the average turn around time (not necessarily true always). Turn around time of a process is the time of its completion – time of its arrival. Thus, using FCFS strategy,

$$\text{Average turn around time} = \frac{7 + (7 + 18) + (7 + 18 + 5)}{3} = \frac{62}{3} = 20.66 \text{ unit}$$

Whereas, using RR algorithm,

$$\text{Average turn around time} = \frac{15 + 30 + 20}{3} = \frac{65}{3} = 21.66 \text{ unit}$$

See the result by repeating the calculations but using the sequence of processes as  $P_2$ ,  $P_1$  and  $P_3$ .

In time sharing systems any process may arrive at any instant of time. Generally, all the process currently under executions, are maintained in a queue. When a process finishes its execution this process is deleted from the queue and whenever a new process arrives



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.

### 6.3 INVERTED TABLES

It will be judicious if we discuss the concept of inverted tables with the following example.

Suppose, a telephone company maintains records of all the subscribers of a telephone exchange as shown in Table 6.1. These records can be used to serve several purpose. One of them requires the alphabetical ordering of the name of the subscriber (say, in order to put the name of the subscriber into telephone directory). Second, it requires the lexicographical ordering of the address of subscriber (say, it is required for routine maintenance). Third, it also requires the ascending order of the telephone numbers (say, in order to estimate the cabling charge from the telephone exchange to the location of telephone connection) etc. To serve all these purpose, the telephone company should maintain three sets of records: one in alphabetical order of the NAME, second, the lexicographical ordering of the ADDRESS and third, the ascending order of the phone numbers. But this way of maintaining records leads to the following serious drawbacks:

1. Requirement of extra storage: three times of the actual memory.
2. Difficult in modification of records: if a subscriber change his address then we have to modify this in three storage otherwise consistency in information will be lost.

However, using the concept of inverted tables, we can avoid the multiple sets of records, and we can still retrieve the records by any of the three keys almost as quickly as if the records are fully sorted by that key. Therefore, we should maintain an inverted table. In this case, this table comprise of three columns: NAME, ADDRESS, and PHONE as shown in Table 6.1(b). Each column contains the index numbers of records in the order based on the sorting of the corresponding key. This inverted table, therefore, can be consulted to retrieve information.

**Table 6.1 Multi-key Access and Its Inverted Table**

| (a) Records of a Telephone Exchange |                  |                      |        | (b) Inverted Table |         |       |
|-------------------------------------|------------------|----------------------|--------|--------------------|---------|-------|
| Index                               | Name             | Address              | Phone  | Name               | Address | Phone |
| 1                                   | K.R. Narayana    | Maker Towers #6      | 257696 | 2                  | 7       | 8     |
| 2                                   | A.B. Vajpayee    | 9 Vivekananda Road   | 257459 | 6                  | 6       | 4     |
| 3                                   | L.K. Advani      | 11 Von Kasturba Marg | 257583 | 1                  | 1       | 2     |
| 4                                   | Mamta Banerjee   | 342 Patel Avenue     | 257423 | 3                  | 8       | 5     |
| 5                                   | Y. Sinha         | 5 SBI Road           | 257504 | 4                  | 9       | 6     |
| 6                                   | D. Kulkarni      | 369 Faculty Colony   | 257564 | 8                  | 4       | 7     |
| 7                                   | T. Krishnamurthy | 185 Faculty Colony   | 257579 | 7                  | 5       | 3     |
| 8                                   | N. Puranjay      | 409 Medical Colony   | 257409 | 9                  | 2       | 1     |
| 9                                   | Tadi Tabi        | Officers Mess #52    | 257871 | 5                  | 3       | 9     |

### 6.4 HASH TABLES

There are another type of tables which help us to retrieve information very efficiently. The ideal *hash table* is merely an array of some constant size, the size depends on the application where



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.

### **Digit analysis method**

The basic idea of this hashing function is to form hash addresses by extracting and/or shifting the extracted digits or bits of the original key. As an example, given a key value, say 6732541, it can be transformed to the hash address 427 by extracting the digits in even positions and then reversing it. For a given set of keys, the position in the keys and same rearrangement pattern must be used consistently. The decision for extraction and then rearrangement is based on some analysis. To do this, an analysis is performed to determine which key positions should be used in forming hash addresses. For each criteria, hash addresses are calculated and then a graph is plotted, then that criteria will be selected which produce the most uniform distributions, that is with the smallest peaks and valleys.

This method is particularly useful in the case of static files where the key values of all the records are known in advance.

We have assumed the key values as integers in our previous discussions, but it need not be necessary always. In fact, any key value can be represented with a string of characters and then ASCII values of its constituent characters can be taken to convert it into numeric value. Thus, assuming that a key value  $k = k_1k_2k_3 \dots k_n$ , where each  $k_i$  is the constituent character in  $k$ . The hash function using division method is stated as below in the algorithm HASH\_DIVISION.

#### **Algorithm HASH\_DIVISION( $K$ ; INDEX)**

**Input:**  $K$ , the key value in the form of a string of characters whose hash address is to be calculated.

**Output:** INDEX, a positive integer as the hash address.

**Data structure:** Hash table in the form of an array  $H$ , the size of the hash table which will be used for modulo arithmetic operation.

#### **Steps:**

1.  $i = 1$  // $i$  is the pointer to the string  $K$
2.  $keyVal = 0$  //To store the keyvalue of  $K$
3. While ( $K[i] \neq \text{NULL}$ ) do
  1.  $keyVal = keyVal + K[i]$  //Add the ASCII value of  $K$
  2.  $i = i + 1$  //Move to the next character
4. EndWhile
5.  $INDEX = keyVal \bmod H + 1$  //Find the remainder modulo
6. Return (INDEX)
7. Stop

The algorithms for other hash function can be designed which is left as an exercise.

#### **Assignment 6.2**

1. Write the algorithms HASH\_MIDSQUARE( $K$ ), HASH\_FOLDING\_PURE( $K$ ), HASH\_FOLDING\_SHIFT( $K$ ), HASH\_FOLDING\_BOUNDARY( $K$ ), for the midsquare method and different variation of folding method respectively for a given key value  $K$  in the form of a string of characters.
2. Suppose a file contains 100 records. What should be the size of the hash table and hence  $h$ . Create an array of key values of the records as hash table. Generate 100 random numbers and assume them as key values. Apply different hash functions to calculate hash addresses and load the key values into the hash table.



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.



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.

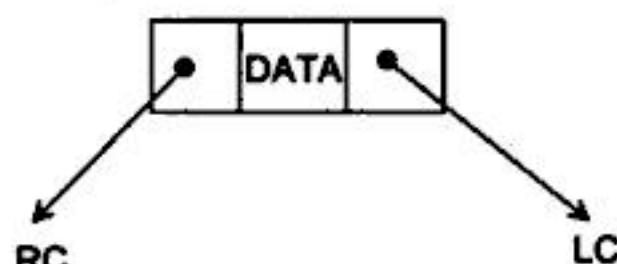


You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

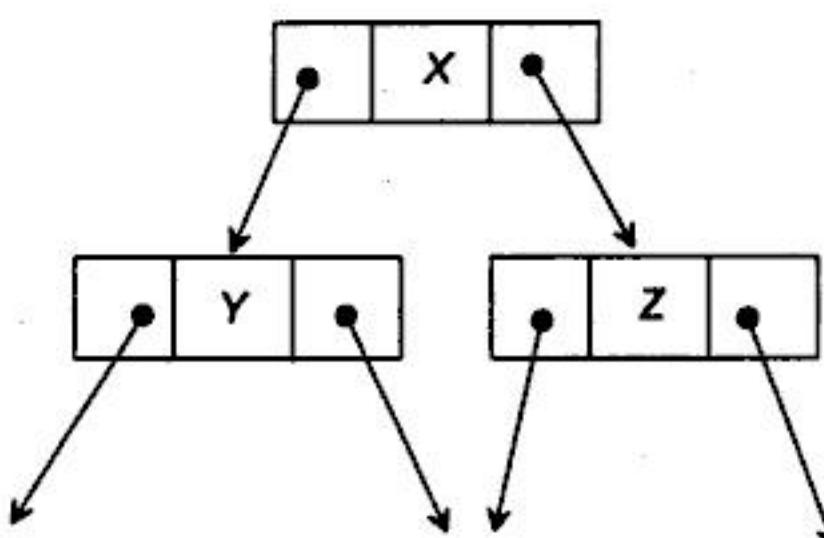
**Child.** If the immediate predecessor of a node is the parent of the node then all immediate successors of a node are known as *child*. For example, in Figure 7.4(b), Y and Z are the two child of X. Child which is at the left side called the *left child* and that of at the right side is called the *right child*.

**Link.** This is a pointer to a node in a tree. For example, as shown in Figure 7.4(a), LC and RC are two *links* of a node. Note that there may be more than two links of a node.

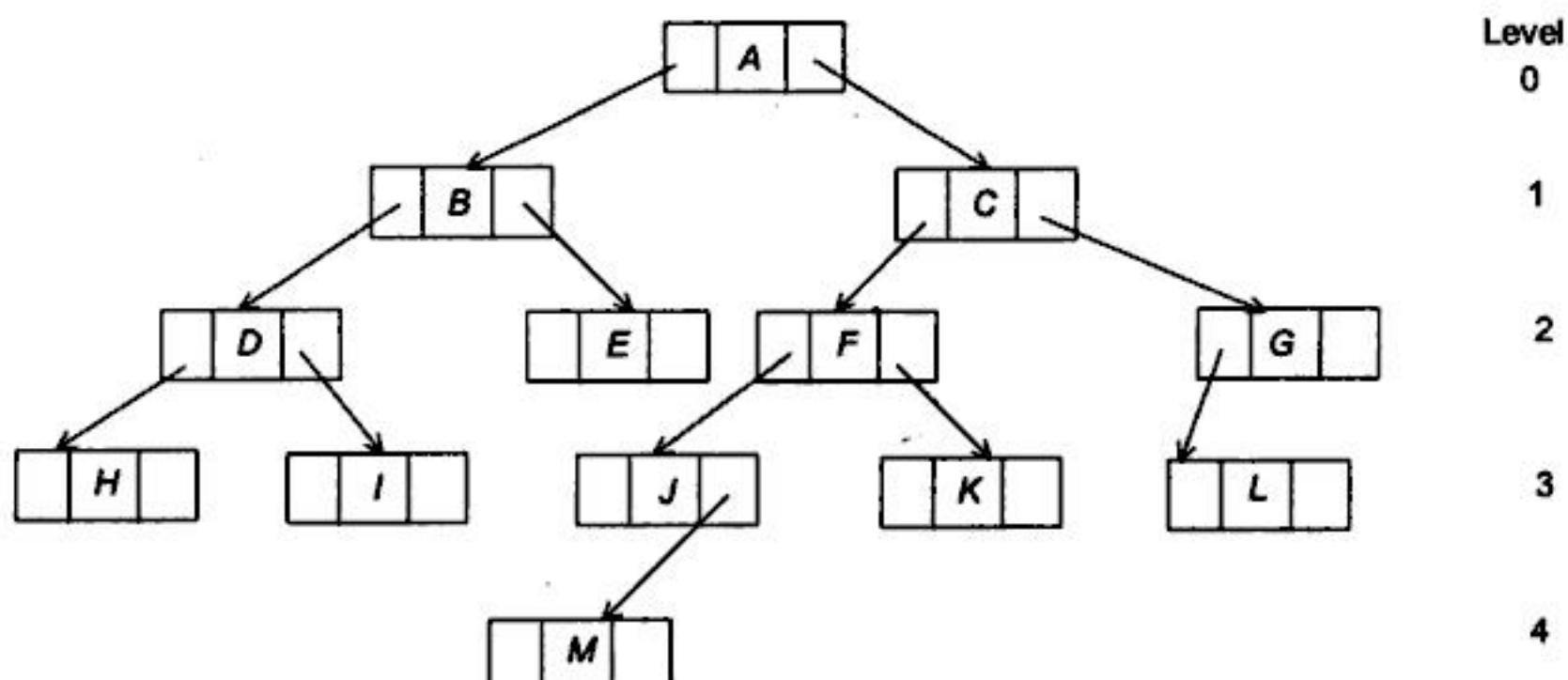
**Root.** This is a specially designated node which has no parent. In Figure 7.4(c), A is the *root* node.



(a) Structure of a node in a tree



(b) Parent, left child and right child of a node



(c) A simple tree with 13 nodes

Fig. 7.4 A tree and its various components.



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.



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.

**Algorithm BUILD\_TREE1(*I*, ITEM) //A binary tree currently with node at *I*****Input:** ITEM is the data content of the node *I*.**Output:** A binary tree with two sub-trees of node *I*.**Data structure:** Array representation of tree.**Steps:**

1. If (*I* ≠ 0) then //If the tree is not empty
  1. *A*[*I*] = ITEM //Store the content of the node *I* into the array *A*
  2. Node *I* has left sub-tree (Give option = Y/N)?
    3. If (option = Y) then //If node *I* has left sub tree
      1. BUILD\_TREE1(2 \* *I*, NEWL) //Then it is at 2\*I with next item as NEWL
    4. Else
      1. BUILD\_TREE1(0, NULL) //Empty sub-tree
  5. EndIf
  6. Node *I* has right sub-tree (Give option = Y/N)?
    7. If (option = Y) //If node *I* has right sub-tree
      1. BUILD\_TREE1 (2\*I+1, NEWR) //Then it is at 2\*I+1 with next item as NEWR
    8. Else
      1. BUILD\_TREE1(0, NULL) //Empty sub-tree
  9. EndIf
2. EndIf
3. Stop

**Algorithm BUILD\_TREE2(PTR, ITEM)****Input:** ITEM is the content of the node with pointer PTR.**Output:** A binary tree with two sub-trees of node PTR.**Data structure:** Linked list structure of binary tree.**Steps:**

1. If (PTR ≠ NULL) then //If the tree is not empty
  1. PTR.DATA = ITEM //Store the content of node at PTR
  2. Node PTR has left sub-tree (Give option = Y/N)?
    3. If (option = Y) then
      1. lcptr = GETNODE(NODE) //Allocate memory for the left child
      2. PTR.LC = lcptr //Assign it to Left link
      3. BUILD\_TREE2(lcptr, NEWL) //Build left sub-tree with next item as NEWL
    4. Else
      1. lcptr = NULL
      2. PTR.LC = NULL //Assign for an empty left sub-tree
      3. BUILD\_TREE2(lcptr, NULL) //Empty sub-tree
  5. EndIf
  6. Node PTR has right sub-tree (give option = Y/N)?
    7. If (option = Y) then
      1. rcptr = GETNODE(NODE) //Allocate memory for the right child
      2. PTR.RC = rcptr //Assign it to Right link
      3. BUILD\_TREE2(rcptr, NEWR) //Build right sub-tree with next item as NEWR



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.



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.

Suppose,  $T_1$  and  $T_2$  are two binary trees.  $T_2$  can be merged with  $T_1$  if all the nodes from  $T_2$ , one by one, is inserted into the binary tree  $T_1$  (insertion may be as internal node when it has to maintain certain property or may be as external nodes).

Another way, when the entire tree  $T_2$  (or  $T_1$ ) is included as a sub-tree of  $T_1$  (or  $T_2$ ). For this, obviously we need that in either (or both) tree there must be at least one null sub-tree. We will consider in our subsequent discussion, this second case of merging.

Before, going to perform the merging, we have to test for compatibility. If in both the trees, root node have both the left and right sub-trees then merge will fail; otherwise if  $T_1$  has left sub-tree (or right sub-tree) empty then  $T_2$  will be added as the left sub-tree (or right sub-tree) of the  $T_1$  and vice versa. For example, as in Figure 7.27,  $T_1$  has right sub-tree empty hence  $T_2$  is added as the right sub-tree of  $T_1$ . Note that if  $T(N)$  denotes the number of nodes  $n$  in tree  $T$  then

$$T(n_1 + n_2) = T_1(n_1) + T_2(n_2)$$

where  $T$  is the resultant tree of merging  $T_1$  and  $T_2$ .

Following is the algorithm MERGE to define the merge operation.

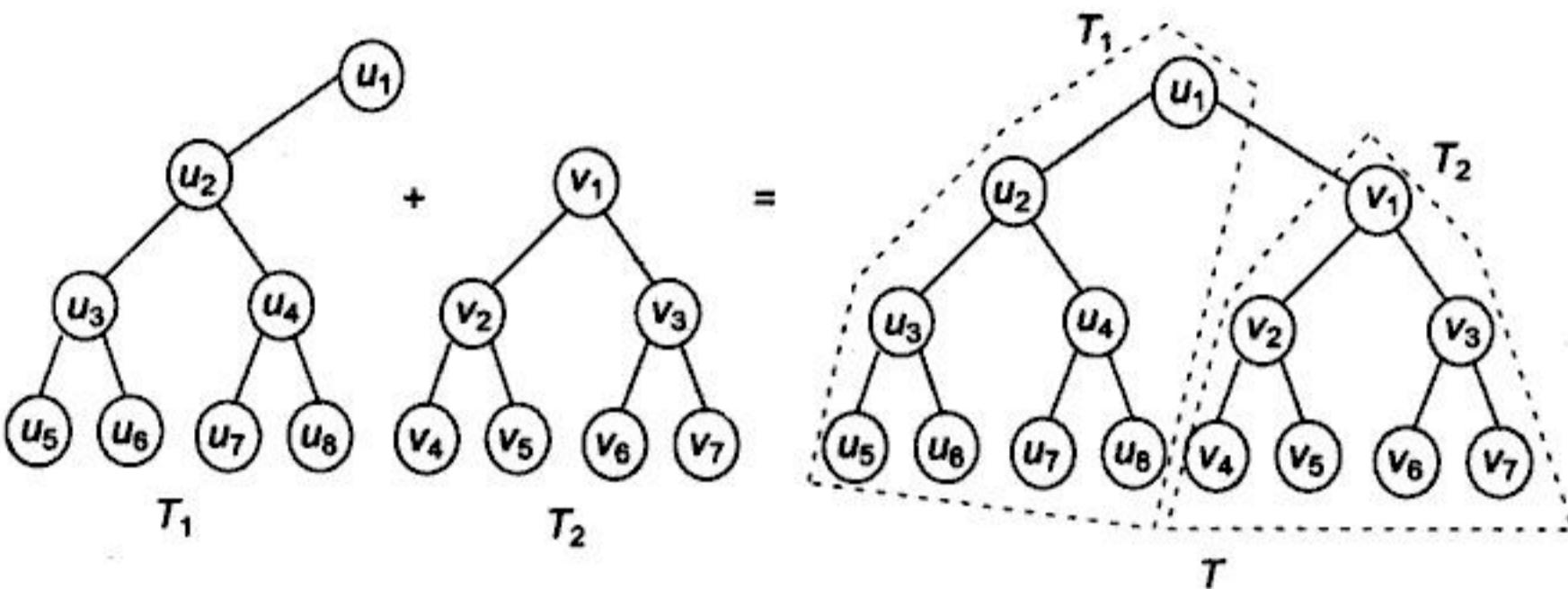


Fig. 7.27 Merging of two binary trees.

#### Algorithm MERGE(ROOT<sub>1</sub>, ROOT<sub>2</sub>; ROOT)

**Input:** Two pointers ROOT<sub>1</sub> and ROOT<sub>2</sub> are the roots of two binary tree  $T_1$  and  $T_2$  respectively with linked structure.

**Output:** A binary tree containing all the nodes of  $T_1$  and  $T_2$  having pointer to the root as ROOT.

**Data structure:** Linked structure of binary tree.

#### Steps:

1. If (ROOT<sub>1</sub> = NULL) then
    1. ROOT = ROOT<sub>2</sub>
    2. Exit
  2. Else
    1. If (ROOT<sub>2</sub> = NULL) then
      1. ROOT = ROOT<sub>1</sub>
      2. Exit
- //If  $T_1$  is empty then  $T_2$  is the result  
//Tree  $T_2$  is the result
- //If  $T_2$  is empty then  $T_1$  is a result



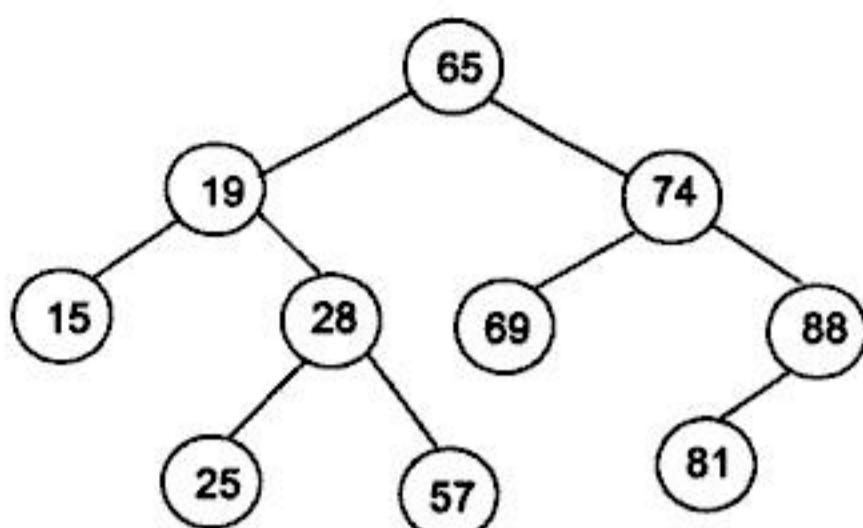
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



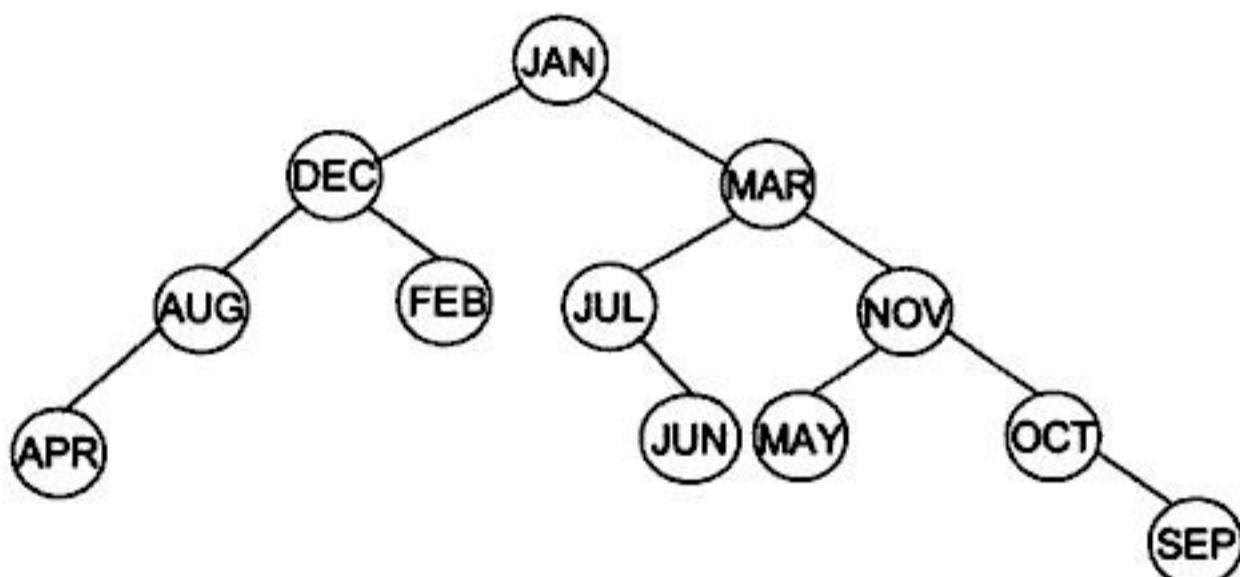
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



(a) Binary search tree with numeric data



(b) Binary search tree with alphabetic data

Fig. 7.32 Binary search trees.

Observe that in Figure 7.32(b), the lexicographical ordering is taken among the data whereas in Figure 7.32(a), numerical ordering is taken.

Now, let us discuss the possible operations on any binary search tree. Operations those are generally encountered to manipulate this data structures are:

- Searching for a data
- Inserting any data into it
- Deleting any data from it
- Traversing the tree.

### **Searching in a binary search tree**

Searching for a data in a binary search tree is much faster than in arrays or linked lists. This is why, in the applications where frequent searching operations are to be performed, this data structure is used to store data. In this section, we will discuss how this operation can be defined.

Suppose, in a binary search tree  $T$ , ITEM be the item of search. We will assume that, the tree is represented using linked structure.

We start from the root node  $R$ . Then, if ITEM is less than the value in the root node  $R$ , we



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.



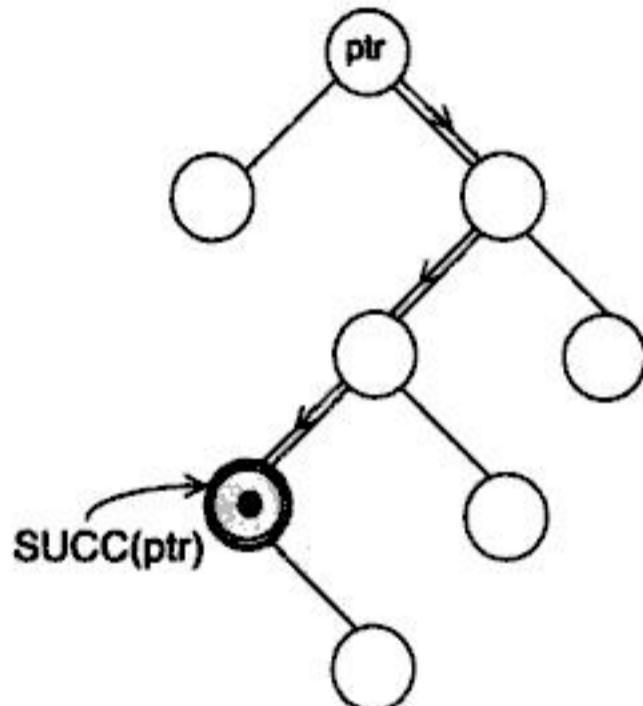
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```

 2. EndIf
 3. EndIf
 4. RETURN_NODE(ptr) //Return deleted node to the memory bank
12. EndIf
13. If (case = 3) //When contains both child
 1. ptr1 = SUCC(ptr) //Find the in order successor of the node
 2. item1 = ptr1.DATA
 3. BST_DELETE (item1) //Delete the inorder successor
 4. ptr.DATA = ITEM //Replace the data with the data of in
 //order successor
14. EndIf
15. Stop

```

Note that, we assume the function SUCC(ptr) which returns pointer to the inorder successor of the node ptr. It can be verified that inorder successor of ptr always occurs in the right sub-tree of ptr, and inorder successor of ptr does not have left child. Figure 7.38 shows an instance.



**Fig. 7.38** Inorder successor of a node ptr.

Finding the inorder successor of any node is very easy and is stated as below:

#### Algorithm SUCC(PTR)

**Input:** Pointer to a node PTR whose inorder successor is to be known.

**Output:** Pointer to the inorder successor of ptr.

**Data structure:** Linked structure of binary tree.

#### Steps:

```

1. ptr1 = PTR.RCHILD //Move to the right sub-tree
2. If (ptr1 ≠ NULL) then //If right sub-tree is not empty
 1. While (ptr1.LCHILD ≠ NULL) do //Move to the left-most end
 1. ptr1 = ptr1.LCHILD
 2. EndWhile
3. EndIf
4. Return(ptr) //Return the pointer of the inorder successor
5. Stop

```



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.

tree. Next, we have to compare it with the data in its parent; if the value is greater than that at parent then interchange the values. This will continue between two nodes on path from the newly inserted node to the root node till we get a parent whose value is greater than its child or we reached at the root.

For illustration, 19 is added as the left child of 25, Figure 7.42(b). Its value is compared with its parent's value, and to be a max heap, parent's value > child's value is satisfied, hence interchange as well as further comparison is no more required.

As an another illustration, let us consider the case of inserting 111 into the resultant heap tree. First, 111 will be added as right child of 25, when 111 is compared with 25 it requires interchange. Next, 111 is compared with 85, another interchange takes place. At the last, 111 is compared with 95 and then another interchange occurs. Now, our process stops here, as 111 is now in root node. The path on which these comparisons and interchanges have taken places are shown by dashed line, Figure 7.42(c).

Now, we are in a position to describe the algorithm MAX\_HEAP\_INSERT to insert a data into a max heap tree. Let us assume that  $A[1\dots\text{SIZE}]$  be an array of data where the heap tree is stored and its maximum capacity is SIZE.  $N$  denotes the number of nodes currently in the heap tree. For an empty heap tree,  $N = 0$ .

#### **Algorithm MAX\_HEAP\_INSERT(ITEM)**

**Input:** ITEM, the data to be inserted;  $N$ , the strength of nodes.

**Output:** ITEM is inserted into the heap tree.

**Data structure:** Array  $A[1\dots\text{SIZE}]$  stores the heap tree;  $N$  being the number of node in the tree.

#### **Steps:**

1. If ( $N \geq \text{SIZE}$ ) then
  1. Print "Heap tree is saturated: Insertion is void"
  2. Exit
2. Else
  1.  $N = N + 1$
  2.  $A[N] = \text{ITEM}$  //Adjoin the data in the complete binary tree
  3.  $i = N$  //Last node is the current node
  4.  $p = i \text{ div } 2$  //Its parent node is at  $p$
  5. While ( $p > 0$ ) and ( $A[p] < A[i]$ ) do //Continue till the root is reached or out of order
    1.  $\text{temp} = A[i]$
    2.  $A[i] = A[p]$
    3.  $A[p] = \text{temp}$
    4.  $i = p$  //Parent becomes child
    5.  $p = p \text{ div } 2$  //Parent of parent becomes new parent
  6. EndWhile
3. EndIf
4. Stop



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.

**Steps:**

```

1. CREATE_MAX_HEAP(A) //To create the heap H from the set of data
2. i = N
3. While (i > 1) do //Continue until heap contains single element
 1. SWAP (A[1], A[i]) //Delete the root node and replace it by the last node
 2. i = i - 1 //Pointer to the last node shifted towards the left
 3. j = 1 //To rebuild the heap
 4. While (j < i) do
 1. lchild = 2 * j //Left child
 2. rchild = 2 * j + 1 //Right child
 3. If (A[j] < A[lchild]) and (A[lchild] > A[rchild]) then
 1. SWAP(A[j], A[lchild])
 2. j = lchild
 4. Else
 1. If (A[j] < A[rchild]) and (A[rchild] > A[lchild]) then
 1. SWAP(A[j], A[rchild])
 2. j = rchild
 2. Else
 1. Break() //Rebuild is completed: break the loop
 3. EndIf
 5. EndIf
 5. EndWhile
4. EndWhile
5. Stop

```

In the above algorithm, we assume a procedure CREATE\_MAX\_HEAP( ) to build a heap from a given set of data, which is nothing but the repeated insertions into the heap starting from an empty heap using the procedure MAX\_HEAP\_INSERT( ). Other procedure, like SWAP(...), bears the usual meaning.

**Assignment 7.23** The HEAP\_SORT( ) procedure as discussed is used to sort data in ascending order. Only a few, modifications are required on it for reverse order sorting.

Obtain the necessary modification, so that using heap tree, data can be sorted in descending order.

### **Priority queue implementation using heap tree**

In Section 5.4.3, we have discussed how a priority queue can be implemented using circular array, linked list, etc. Another simplified implementation is possible using heap tree; the heap, however, can be represented using an array. This implementation is therefore free from the complexities of circular array and linked list but getting the advantages of simplicities of array.

It may be recalled that, heap trees allow the duplicity of data in it. Elements associated with their priority values are to be stored in the form of heap tree, which can be formed based on their priority values. The top priority element that has to be processed first is at the root; so it can be deleted and heap can be rebuilt to get the next element to be processed, and so on.



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.

- Any node can be accessible from any other node. Threads are usually more to upward whereas links are downward. Thus in a threaded tree, one can move in either direction and nodes are in fact circularly linked. This is not possible in unthreaded counter part because there we can move only in downward direction starting from root.
- Insertions into and deletions from a threaded tree are although time consuming operations (since we have to manipulate both links and threads) but these are very easy to implement.

### **Various operations on threaded binary trees**

Let us investigate the various operations possible on threaded binary trees. We will discuss the operations on inorder threaded binary tree as a representative example.

**To find the inorder successor of any node.** Given the address of any node in an inorder threaded binary tree we are to find its inorder successors. Finding inorder successor is very straightforward. It can be easily revealed from Figure 7.48(b) that:

- If  $N$  does not have a right child (that is, if the field RTAG implies a thread) then the thread points to its inorder successor.
- Else
- If  $N$  has a right sub-tree (that is, if RTAG implies a link) then the left-most node in this right sub-tree which does not have any left child is the inorder successor of  $N$ .

Thus, with these, one can easily verify the following [with reference to Figure 7.48(b)]:

- Inorder successor of 'B' is 'A'
- Inorder successor of 'A' is 'G'
- Inorder successor of 'D' is 'F'
- Inorder successor of 'C' is the HEADER, etc.

Now, we can formulate the procedure INSUCC( $N$ ) to find the inorder successor of any node  $N$  in a threaded binary tree. This is given in the following algorithm:

#### **Algorithm INSUCC(PTR)**

**Input:** PTR is the pointer to any node whose inorder successor has to be found.

**Output:** Pointer to the inorder successor of the node PTR.

**Data structure:** Linked structure of threaded binary tree.

#### **Steps:**

1. succ = PTR.RCHILD //Get the right child of the node
2. If (PTR.RTAG = FALSE) then //If the node has right sub-tree
  1. While (succ.LTAG = FALSE) //Move to the left-most node without left child
    1. succ = succ.LCHILD
  2. EndWhile
3. EndIf
4. Return(succ) //Return the pointer to the inorder successor
5. Stop

**To find the inorder predecessor of any node.** Finding the inorder predecessor of a given node is similar to the previous one and can be obtained by simply interchanging the RCHILD by LCHILD and LTAG by RTAG, algorithm INPRED () is as:



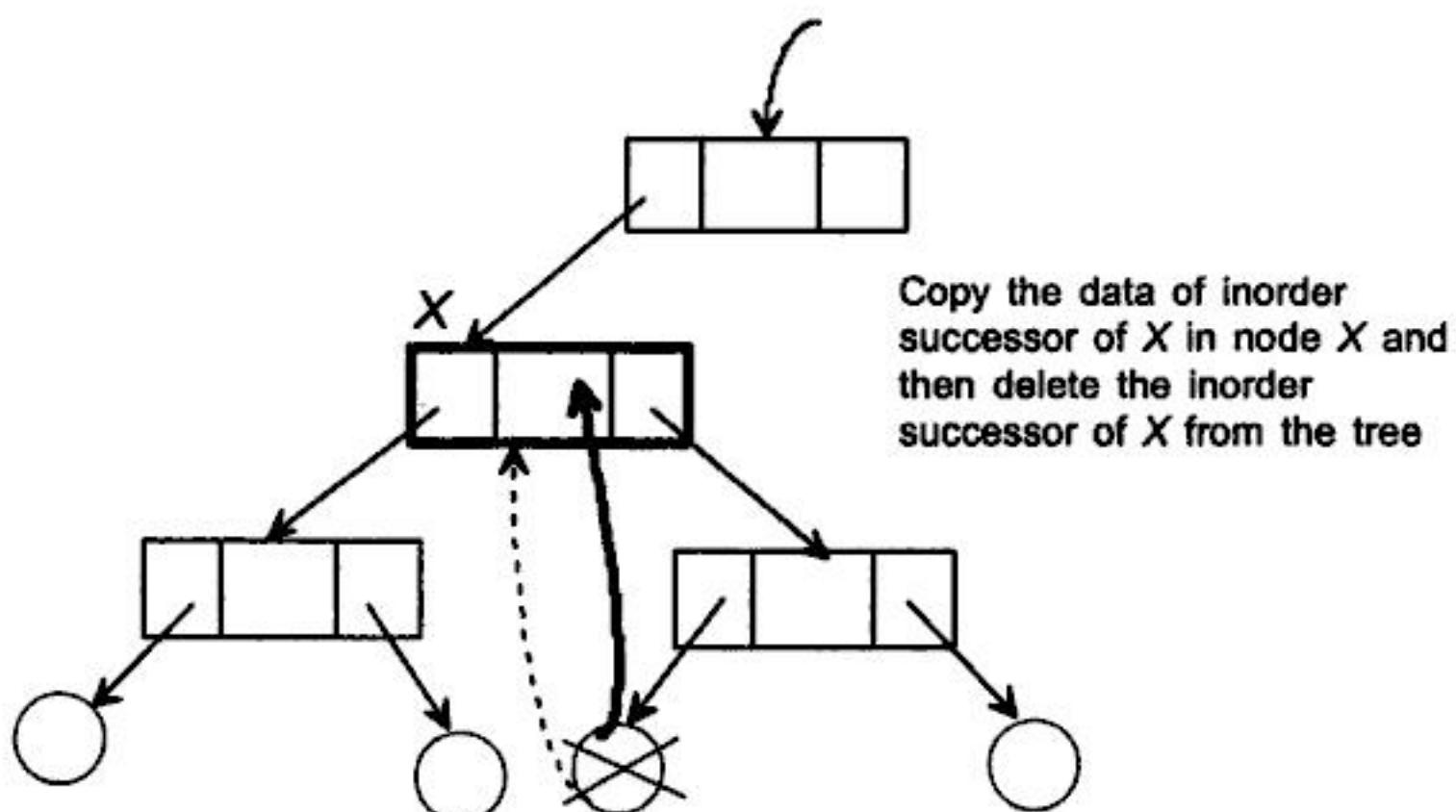
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



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.



(c) Case 3: Deletion of a node having both the sub-trees.

**Fig. 7.51(a)–(c)** Deletion of a node from threaded binary search tree.

In Case 2, we have to consider the modification in both the link and thread pointers. The link fields of the parent of the deleted node are modified by the links of respective sub-trees. If the deleted node is an inorder predecessor (successor) of some nodes then the parent of the deleted node becomes the inorder predecessor (successor) of that node.

Case 3, in fact, is an iterative procedure of deletion. Here, the node to be deleted is replaced by the data content of its inorder successor and then inorder successor has to be deleted either as Case 1 or Case 2. Note that here deletion will terminate either with Case 1 or Case 2.

Writing the procedure for deletion from a threaded binary search tree is now easy; if the pointer of node that has to be deleted is given then, we are to identify its parent first. Following is the procedure PARENT(PTR) to find the pointer of parent node of the node having pointer PTR in threaded binary tree.

#### **Algorithm PARENT(PTR)**

**Input:** The pointer PTR of a node whose parent is to be known.

**Output:** The pointer PARENT to the parent node of a given node PTR.

**Data structure:** Linked structure of threaded binary tree.

#### **Steps:**

1. varptr = PTR
2. While (varptr, RTAG = 0) do //Move to a node till it has a right thread
  1. varptr = varptr.RCHILD
3. EndWhile
4. pred = varptr.RCHILD //Move to a predecessor node
5. If (pred.LCHILD = ptr) then
  1. PARENT = pred
6. Else //Move to the immediate predecessor node
  1. pred = pred.LCHILD



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

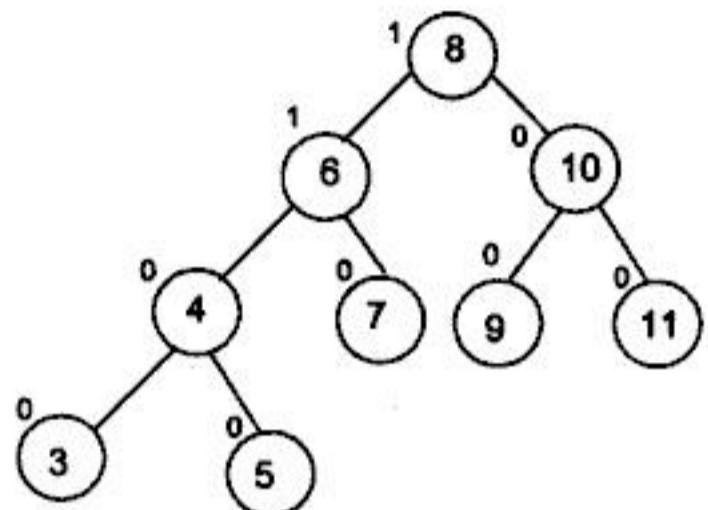
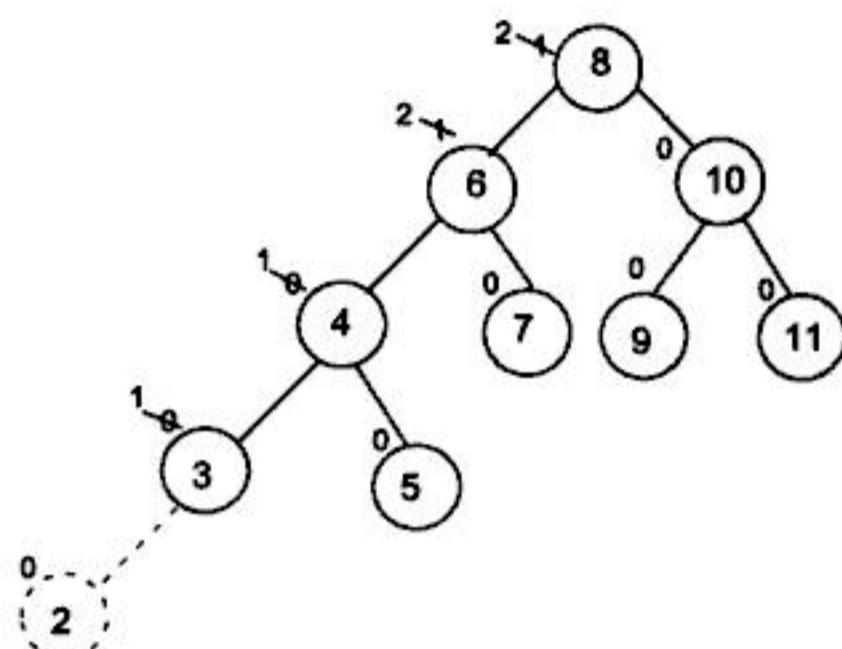


You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

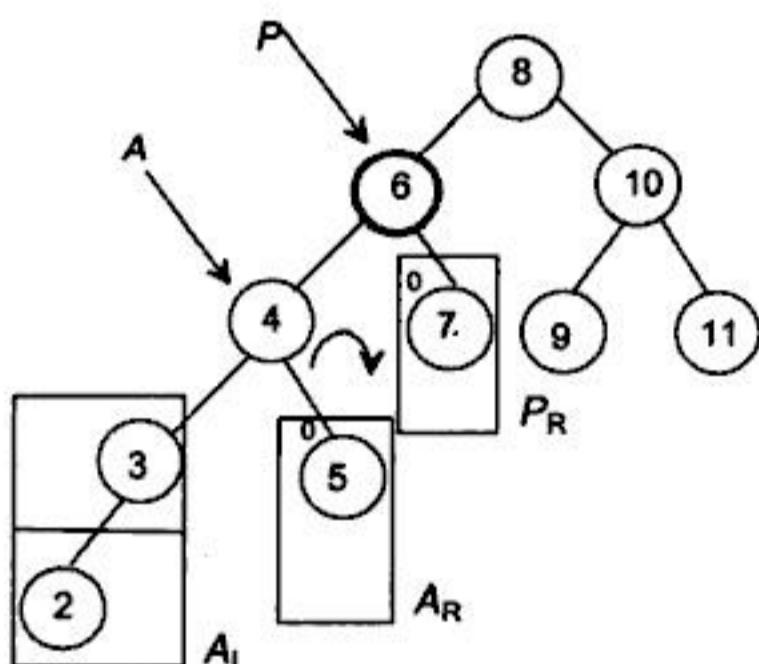
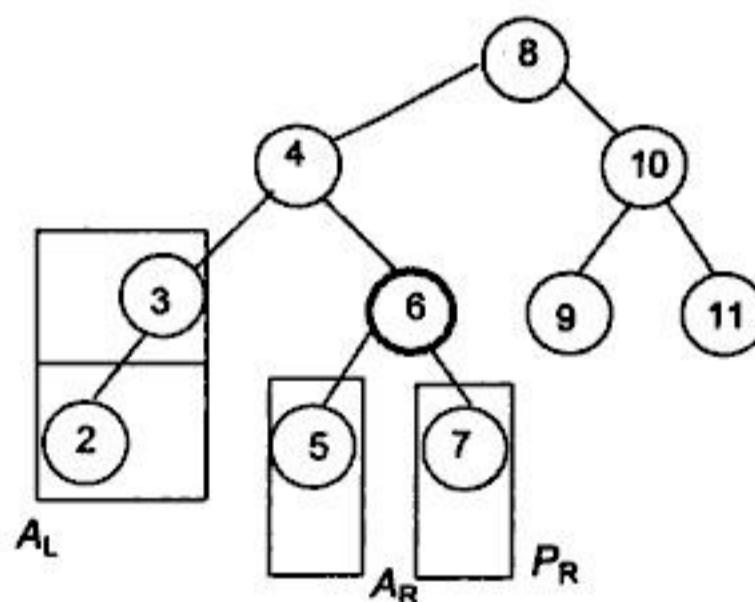


You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

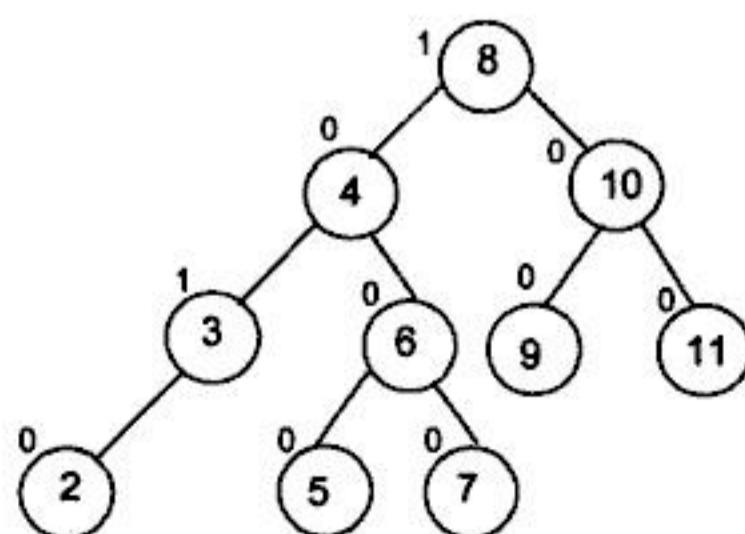
**Illustration.** Consider the illustration as shown in Figure 7.55. Here, 6 is the pivot node because this is the nearest node from the newly inserted node whose balance factor is switched from 1 to 2 in Figure 7.55(b). This insertion corresponds to the case of LEFT-TO-LEFT insertion. AVL rotation required is depicted as in Figures 7.55(c)–(d). Final height balanced tree is shown as in Figure 7.55(e) with balance factor of each node.

(a) A height balanced binary tree as  $|bf| \leq 1$ 

(b) After the insertion of 2 into the tree

(c) AVL rotation as per the LEFT-TO-LEFT  
(case 1) insertion

(d) After AVL rotation



(e) Height balanced tree with newly inserted node 2 and after AVL rotation

Fig. 7.55(a)–(e) LEFT-TO-LEFT insertion and AVL rotation to make the tree height balanced.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



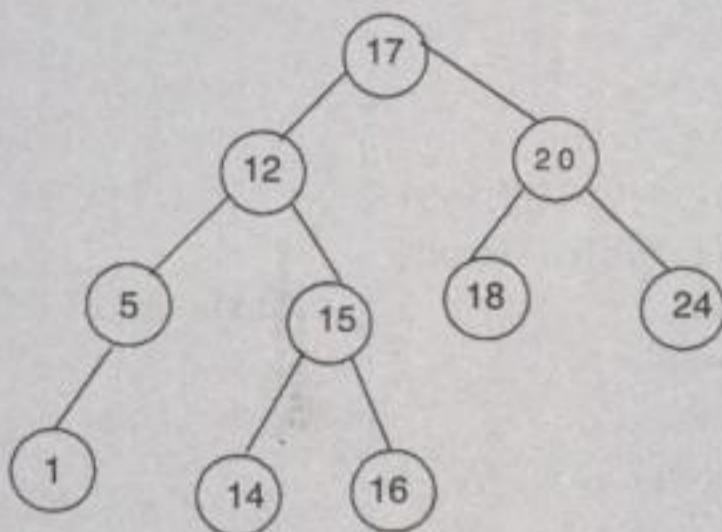
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

**Assignment 7.27**

1. Figure 7.62 represents a binary search tree.
  - (a) Mark the balance factor of each node and state whether it is a height balanced tree or not. If it is not a balanced tree, make it a height balanced tree.
  - (b) Insert 13 into the above tree and recompute the balance factor of each node. Show that insertion of 13 makes it an unbalanced tree. Carry out the necessary AVL rotation to make it height balanced.

**Fig. 7.62** A binary search tree (Assignment 7.27)

2. Starting from an empty height balanced tree, insert the following data one-by-one in the sequence as given; ensure that insertion of each node results a height balanced tree.

Set 1: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12

Set 2: 8, 9, 10, 2, 1, 5, 6, 4, 7, 11, 12, 3

3. Obtain the height balanced tree by the repeated insertions of data given below in their order of occurrence.

jan, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec

Show that the final height balanced tree resembles with the tree as given in Figure 7.52(f).

***Implementation for height balancing a tree***

Implementation of a height balance tree is very straightforward. Suppose, we have to construct a height balanced binary tree. (Starting from an empty tree, repeated insertion will result the height balanced tree.) We will follow the recursive approach of the insertion for the purpose.

In order to make the understanding easy, we will modularize the whole procedure into several modules documented as below. We will assume the following node structure to maintain a height balanced tree.

| LCHILD | DATA | HEIGHT | RCHILD |
|--------|------|--------|--------|
|--------|------|--------|--------|



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.

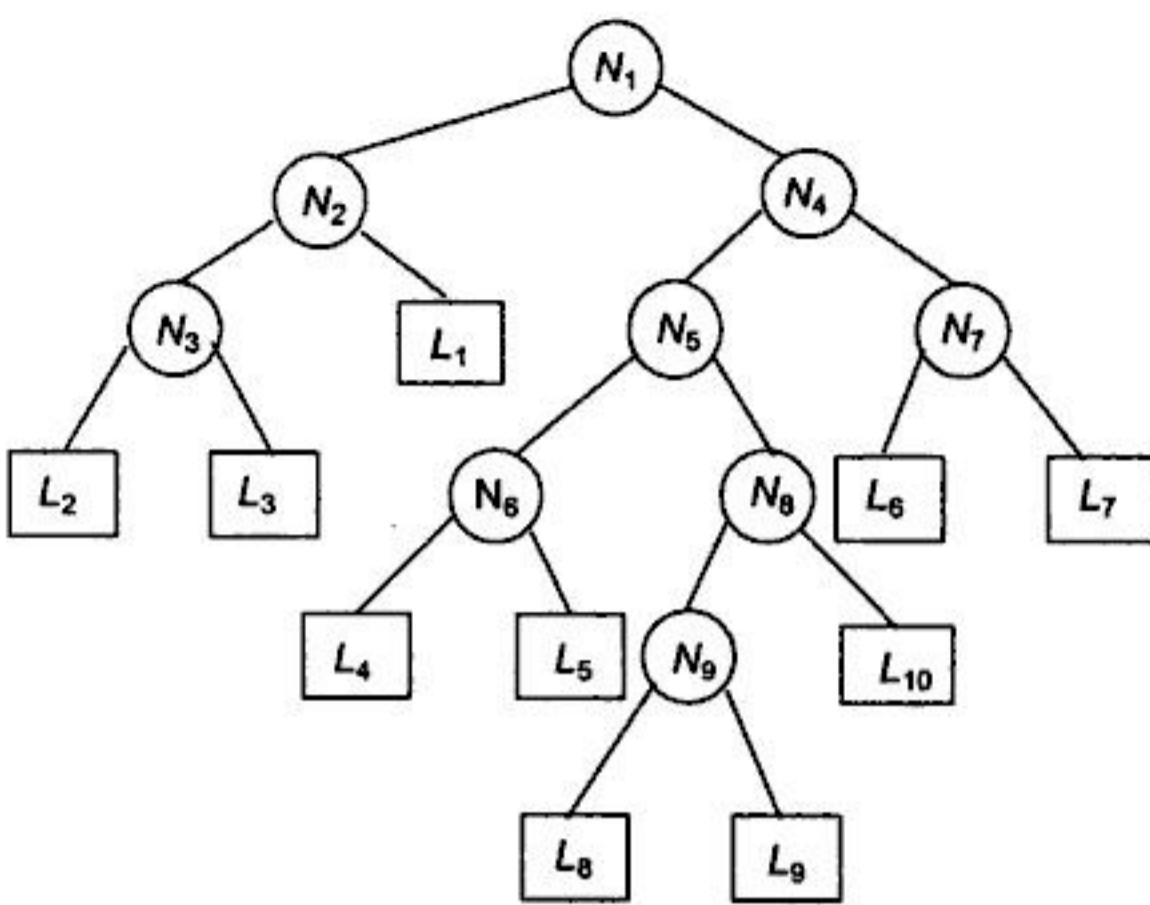


Fig. 7.68 An extended binary tree.

It is already discussed in Section 7.2.2 vide Lemma 7.5 that if  $N_E$  and  $N_I$  denote the number of external and internal nodes then  $N_E = N_I + 1$ . It can be verified from Figure 7.68, where  $N_E = 10$ ,  $N_I = 9$ , and thus  $N_E = N_I + 1$ .

Path length of a node as defined can be verified from Figure 7.68:

$$\text{Path length of } N_1 = 0$$

$$\text{Path length of } N_9 = 4$$

$$\text{Path length of } L_5 = 4$$

$$\text{Path length of } L_8 = 5$$

and so on. Now, we will define additional two terms: *external path length* and *internal path length*.

**External path length.** External path length (let it be denoted as  $E$ ) of a binary tree is defined as the sum of all path lengths, summed over each path from the root node of the tree to an external node. Alternatively,

$$E = \sum_{i=1}^{N_E} P_i$$

where,  $P_i$  denotes the path length of the  $i$ -th external node.

**Internal path length.** Internal path length (let it be denoted as  $I$ ) of a binary tree can be defined analogously as the sum of path lengths of all internal nodes in the tree. Alternatively,

$$I = \sum_{i=1}^{N_I} P_i$$

where,  $P_i$  denotes the path length of the  $i$ -th internal node.



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.

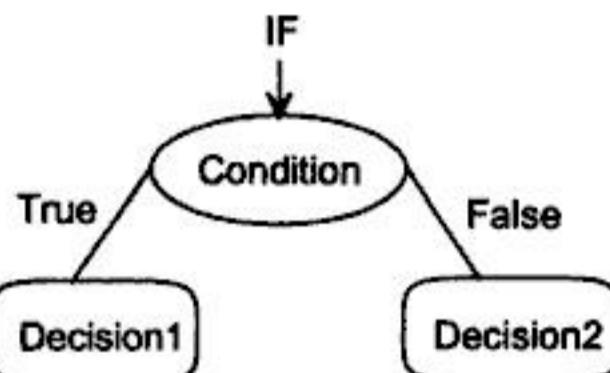
### 7.5.7 Decision Trees

*Decision tree* is a binary tree where a node represents some decision and edges emanating from a node represent the outcome of the decision. External nodes represent the ultimate decisions. In fact, all the internal nodes represent the local decision **IF condition is true THEN decision1 ELSE decision2** towards a global/final decision. Figure 7.76(a) shows the same.

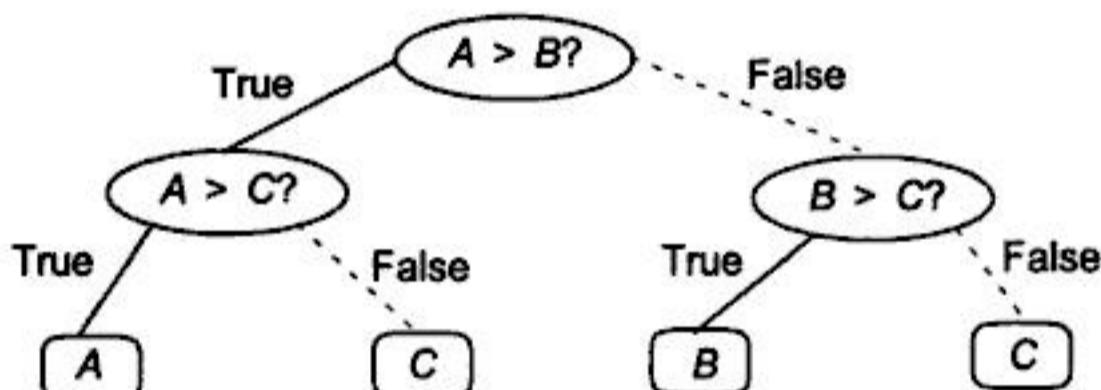
Let us observe a few decision trees. As illustrated in Figure 7.76(b), which depicts how we can proceed in order to decide the greatest number among three numbers, say, A, B, C.

In Figure 7.76(c), another decision tree is illustrated giving the details of flow in order to decide the ascending order of three numbers A, B, C.

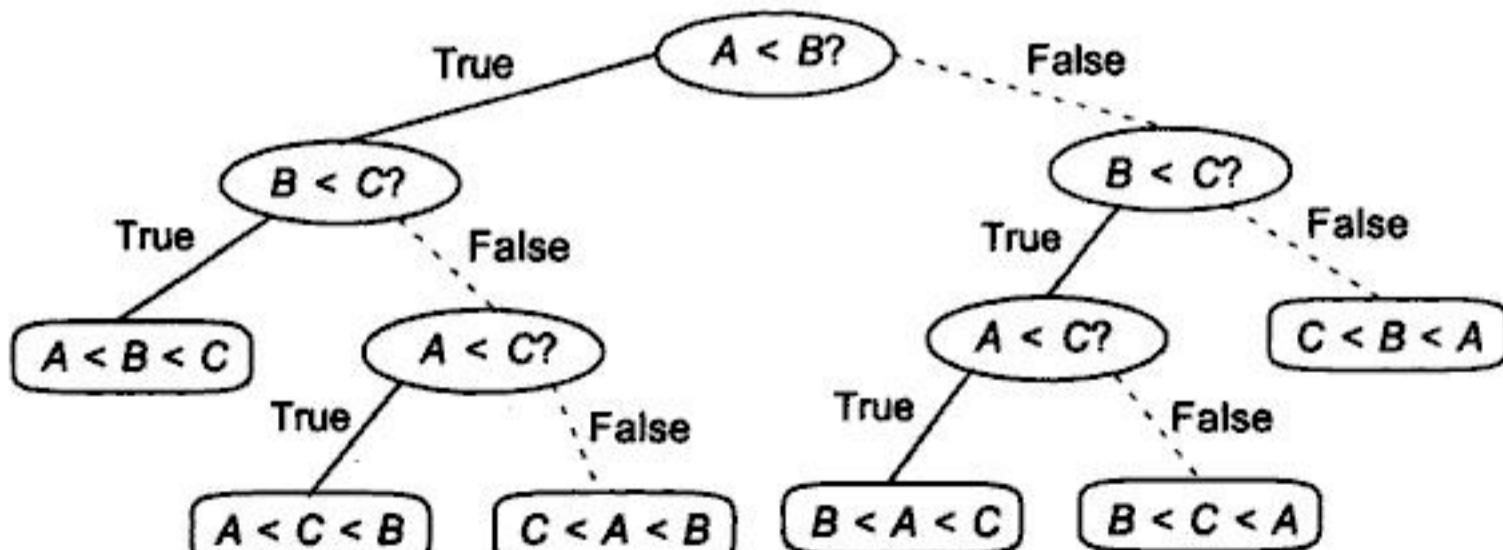
How a Boolean function can be expressed through a decision tree is shown in Figure 7.76(d). Such a decision tree is also alternatively termed as *Boolean tree*.



(a) IF-THEN-ELSE node in decision tree



(b) A decision tree for finding the greatest number among A, B, C



(c) A decision tree for deciding the ascending order among three numbers A, B, C

Fig. 7.76 Continued.



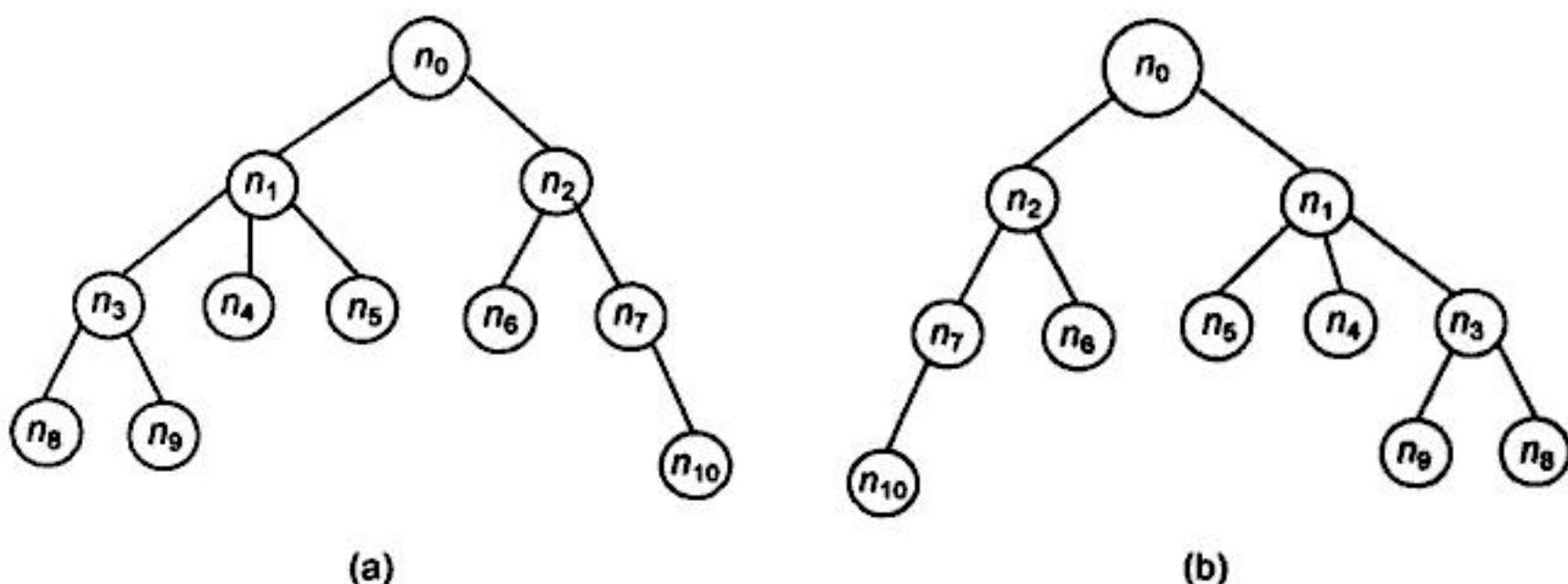
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



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.



**Fig. 7.79** Two general trees.

### 7.6.1 Representation of Trees

Since, in a general tree, a node may have any number of children, the implementation of a general tree is more complex than that of a binary tree. We will discuss the following three ways of representation: linked, array and binary tree representation.

## **Linked representation of trees**

With this representation, we will restrict the maximum number of children for each node. Suppose, the maximum number of children allowed is  $m$ . Then the structure of node will be appeared as shown in Figure 7.80.

| DATA  |       |     |       |
|-------|-------|-----|-------|
| LINK1 | LINK2 | ... | LINKm |

**Fig. 7.80** Node structure for a node in linked representation of a tree.

It is often simpler to write algorithms for a data representation where the node size is fixed. Using DATA and LINK fields, we can represent a tree using the fixed node size linked structure as discussed in Section 7.3.2. This strategy, however, has the disadvantage of wastage of memory space occupied by null links. Suppose, we are to represent a  $m$ -ary tree (that is, a tree of degree  $m$ ) with  $n$  nodes. Then Lemma 7.12 will show, how this representation has overhead of wastage of space.

**Lemma 7.12** If  $T$  is an  $m$ -ary tree with  $n$  nodes, then  $n(m - 1) + 1$  of the  $nm$  link fields are null,  $n \geq 1$ .

*Proof* The proof is very straightforward and left as an exercise.

Lemma 7.12 implies that, for a 3-ary tree more than  $2/3$  of the link fields are null. This wastage of space increases as the degree of the tree increases and proportion approaches to 1.

One alternative to this is to use variable size nodes. Here, in contrast to the fixed size nodes, the size of each node is decided by the number of children it has. This plan requires extra



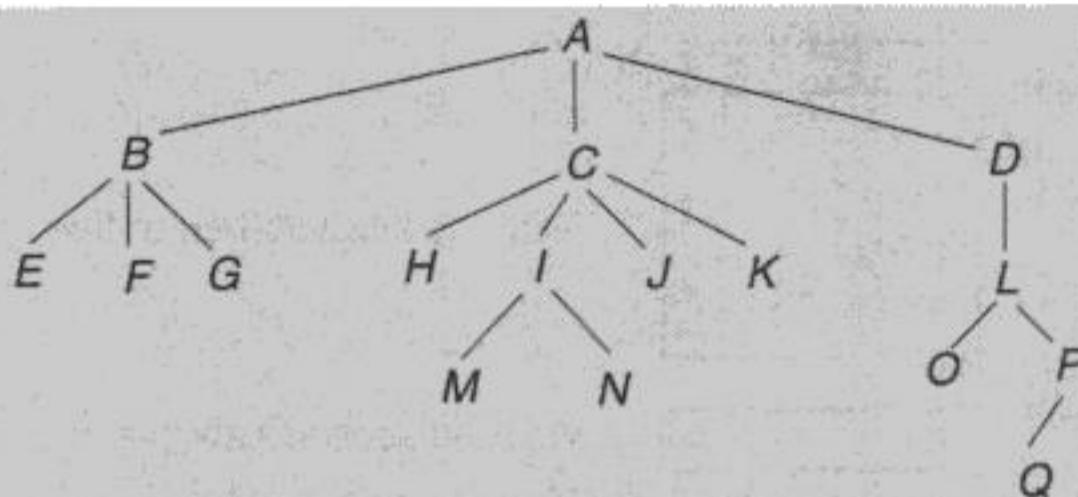
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



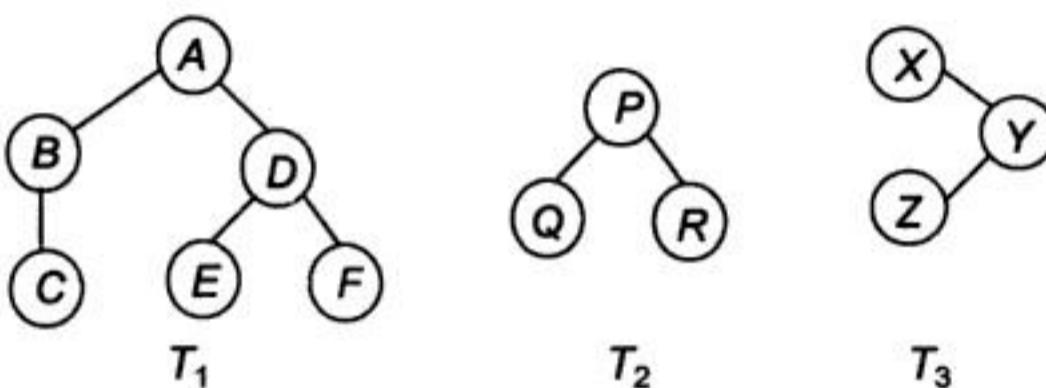
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Fig. 7.84 A general tree  $T$ .**Forest and its binary tree representation**

A *forest* is nothing but a collection of disjoint trees. Figure 7.85 represents a forest.

Fig. 7.85 A forest  $F$ .

Here,  $F$  is a forest which consists of three trees  $T_1$ ,  $T_2$  and  $T_3$ .

So far computer representation of this forest is concerned, this can be represented through its binary tree equivalent. Let us see how such a forest can be transformed into a binary tree.

Suppose, there is a forest  $F$  which contains  $T_1, T_2, \dots, T_n$ ,  $n$  disjoint trees. Then this forest  $F$  can be converted into its corresponding binary tree if we follow the following steps:

- Convert each tree  $T_i$  ( $i = 1, 2, \dots, n$ ) into its corresponding binary tree  $T'_i$ . Note that in such  $T'_i$ , the right link is empty.
- Assume that the root of  $T_1$ , will be the root of binary tree equivalent of forest. Then Add  $T'_{i+1}$  as the right sub-tree of  $T_i$ , for all  $i = 1, 2, \dots, n - 1$ .

As an illustration, let us consider the transformation of forest  $F$  (as given in Figure 7.85) into its corresponding binary tree  $T'$ . The whole transformation is illustrated in Figure 7.86.

As the selection of first tree and then the inclusion of remaining trees are arbitrary hence, the transformation procedure does not provide a unique binary tree.

Inorder and preorder traversals of the corresponding binary tree  $T$  of the forest  $F$  have a natural correspondence with traversals on  $F$ . Inorder traversal of  $T$  is equivalent to visiting the nodes of  $F$  in tree inorder which is defined recursively as given below:

- If  $F$  is empty then return, else
- Traverse the sub-trees of the first tree in tree inorder



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.

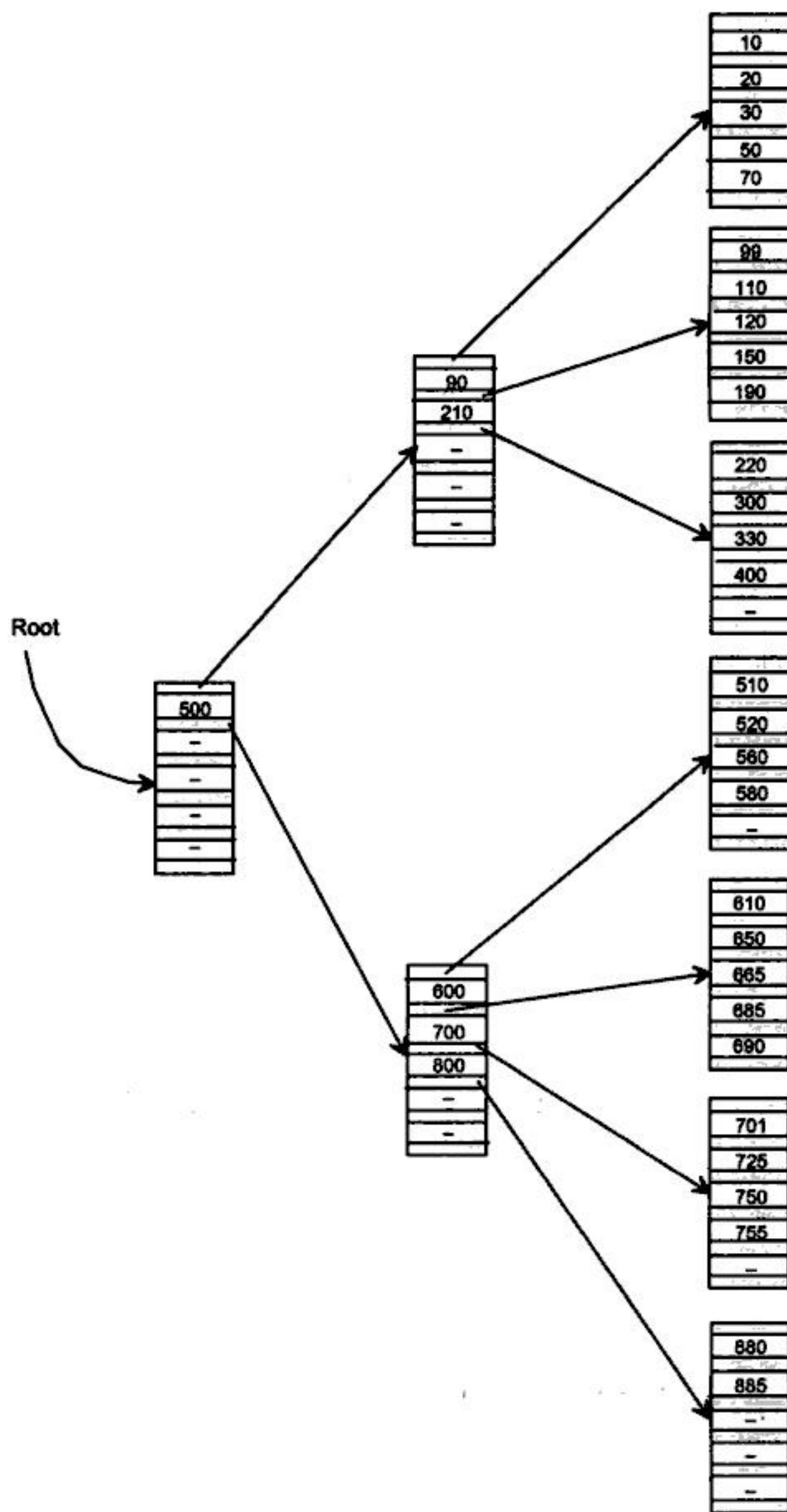


Fig. 7.89 A B tree.



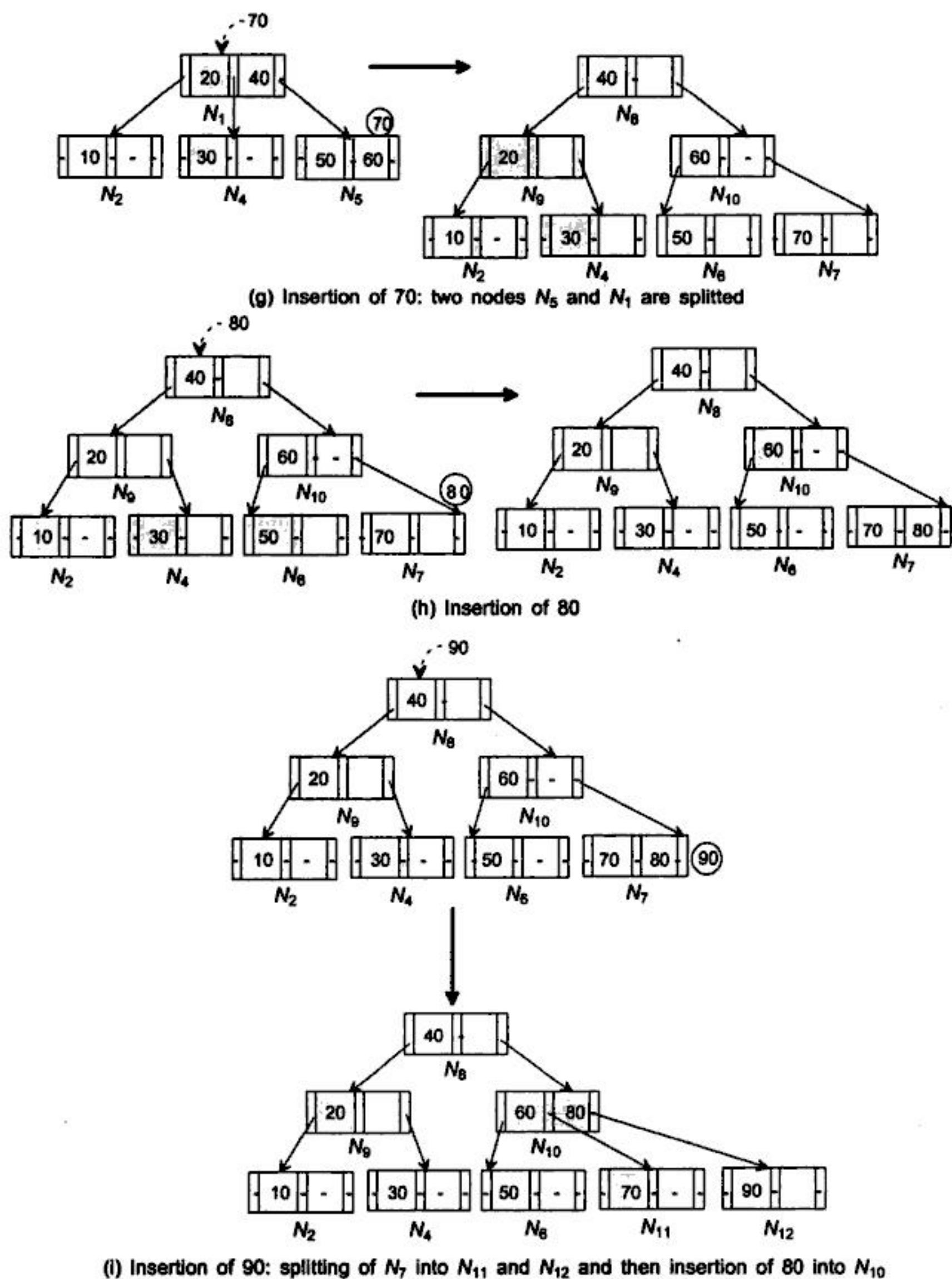
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



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.



**Fig. 7.90** Construction of a B tree of order 3.



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.

**Algorithm BUILD\_BTREE(K)**

**Input:**  $K = [K_1, K_2, \dots, K_n]$  a set of key values

**Output:** A B tree of order  $m$ , whose pointer to the root node is BTR containing the set of values  $K_1, K_2, \dots, K_n$ .

**Steps:**

1. BTR = GETNODE(BTREE\_NODE) //Get a node for the root node
2. For  $i = 0$  to  $(m - 1)$  do //Initially all pointers are null for a node
  1. BTR. $P_i$  = NULL
3. EndFor
4. For  $i = 1$  to  $(m - 1)$  do //Initially all key values are null
  1. BTR. $K_i$  = NULL
5. For  $i = 1$  to  $n$  do //Repeated application of insertion operation
  1. INSERT\_BTREE(BTR,  $K_i$ )
6. EndFor
7. Return (BTR) //Return the pointer to the root node
8. Stop

After the construction of B tree, we will discuss how to delete any key value from a B tree. The following sub-section illustrates the deletion operation in various cases. The formal algorithm is also followed by.

**Deletion**

Deletion of a key value from a B tree may take place in several situations. Each situation has a unique rule for managing the deletion.

Whatever be the situation, deletion of node must ensure the properties of a B tree. The main two properties are to be taken care of during deletion are stated as below:

- The root node must have at least one key value.
- All other nodes (except the root node) must have at least  $\lceil m/2 \rceil - 1$  key values.

Let us first illustrate the various cases of deletion. We will assume a B tree of order 5 as shown in Figure 7.92.

**Case 1: Deletion of a key value from a leaf node.** In a B tree, a *leaf* node is the node which does not have any children. As we have already pointed out that all the nodes other than the root node have at least  $\lceil m/2 \rceil - 1$  key values. After deletion, depending on whether a node (from which a key value has to be deleted) contains minimum number of nodes or not, there are two sub-cases:

**Case 1.1:** Removal of a key value leads to the number of keys  $\geq \lceil m/2 \rceil - 1$ . It is the simplest case of deletion. Removal of a key value from the leaf node which does not disturb the requirement of minimum number of key values in that node.

As an illustration, let us consider the case of deletion of  $l$ ,  $t$ , and  $y$  from the B tree as in Figure 7.92. These deletions are illustrated as in Figure 7.93.

In this case, as we have seen, deletion takes place by removing the key value from the node and then shifting the key values towards the left within the node (to fill the gap), if required.



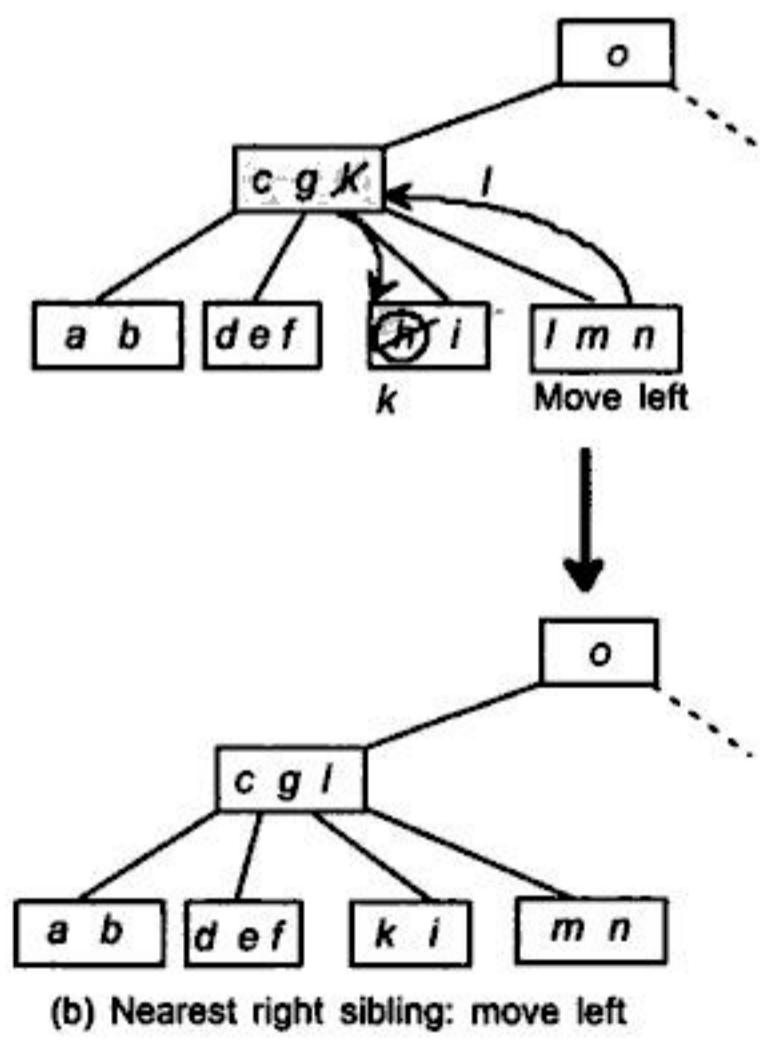
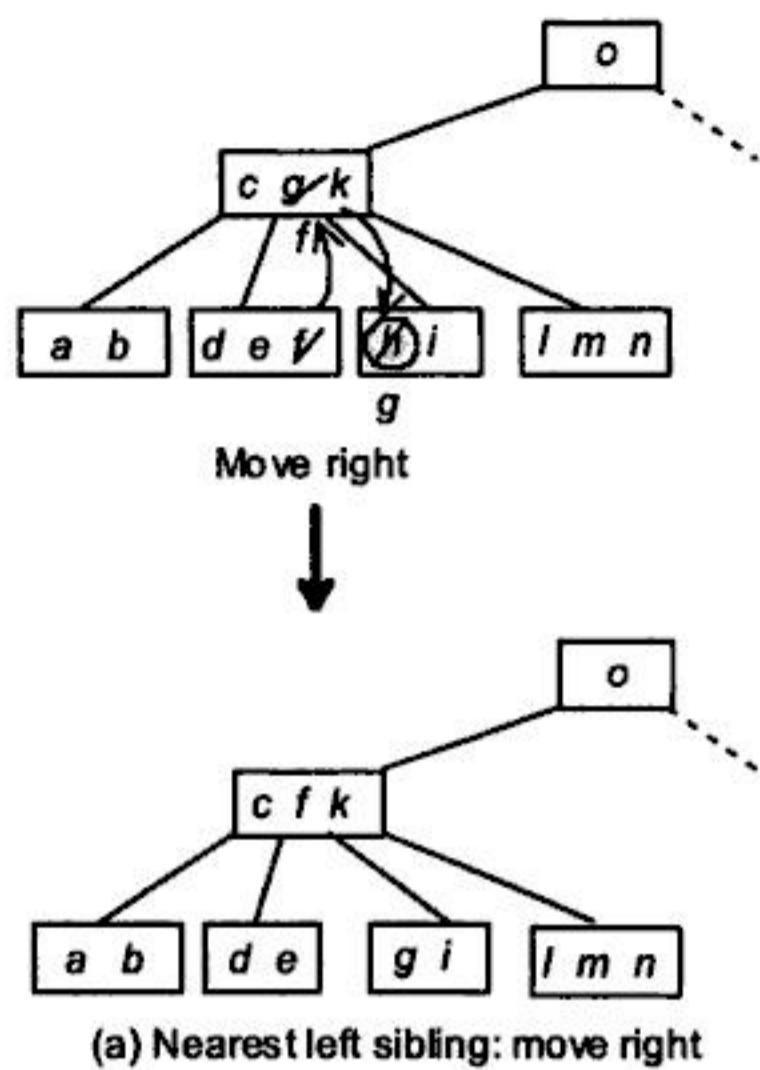
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



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.



**Fig. 7.96** Deletion of same key leads to two different B trees.



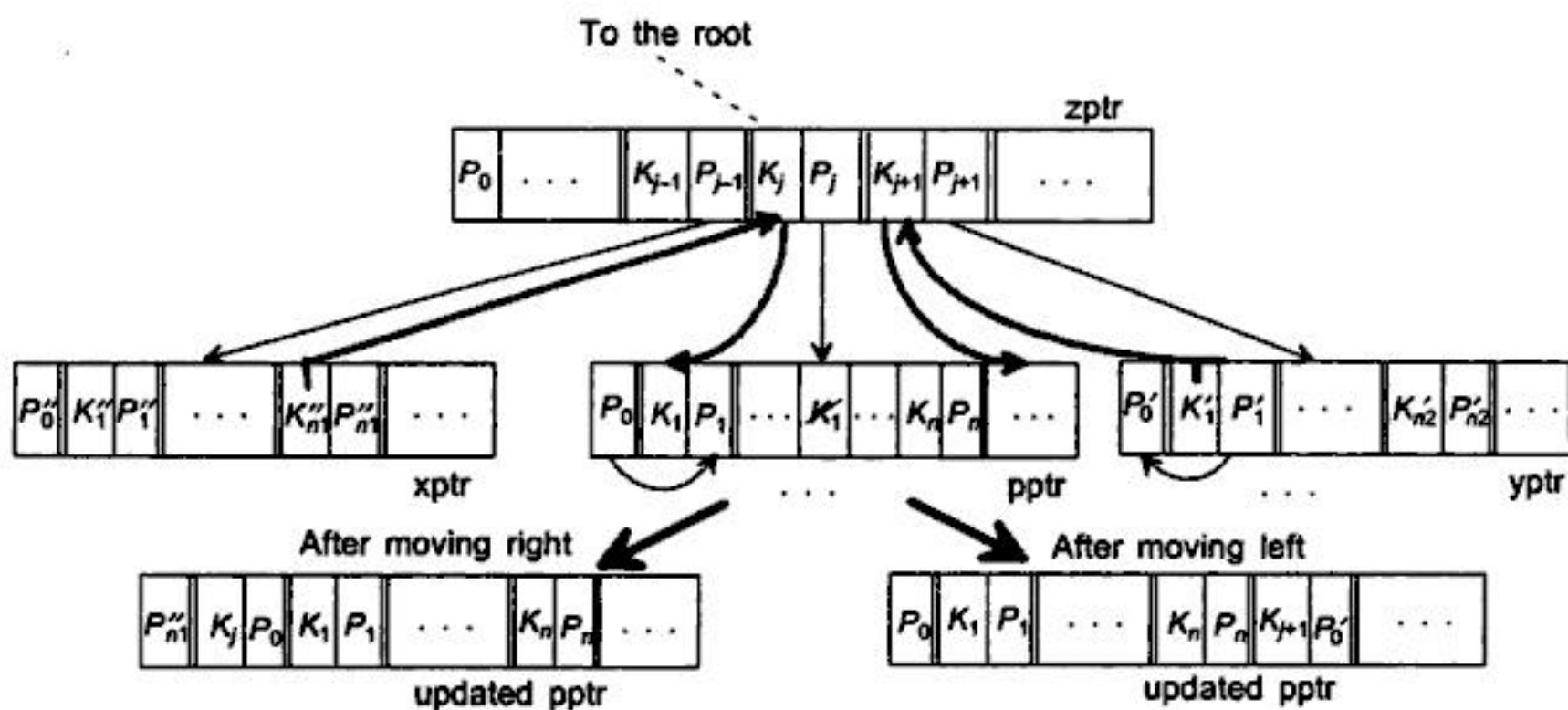
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



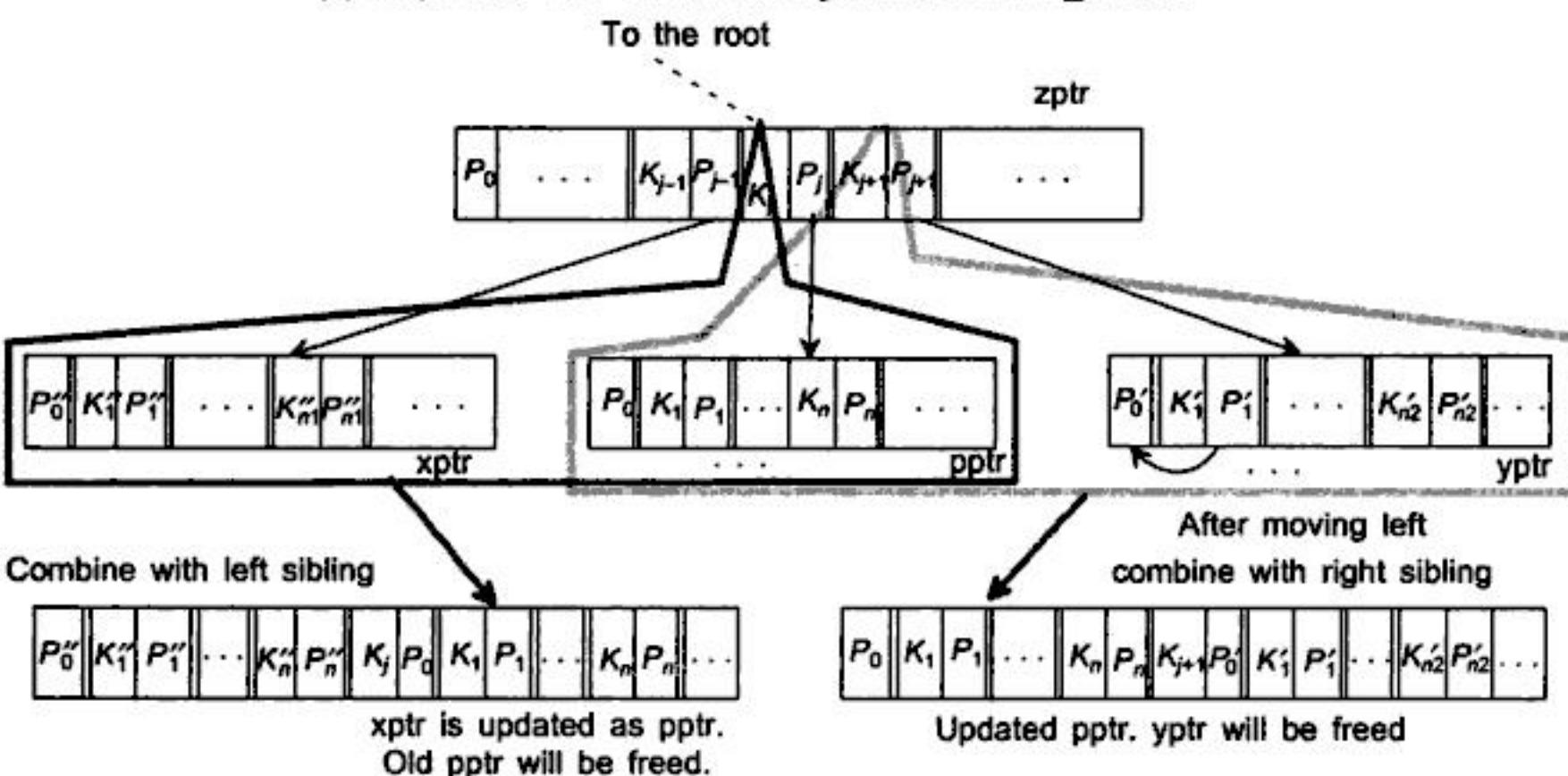
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



(a) Steps 12.7 and 12.9 in the algorithm DELETE\_BTREE



(b) Steps 12.11 and 12.13 in the algorithm DELETE\_BTREE

**Fig. 7.99** Case 2 deletions in B tree according to the algorithm DELETE\_BTREE.

3. EndFor
3. EndIf
4. Stop

One can easily obtain the difference of this inorder traversal of BTREE with the inorder traversal of binary trees (as discussed in Section 7.3). It can also be revealed that the DISPLAY will work in the same manner as that of the inorder traversal of binary tree if we take  $m = 2$ . (This is left as an assignment.)



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

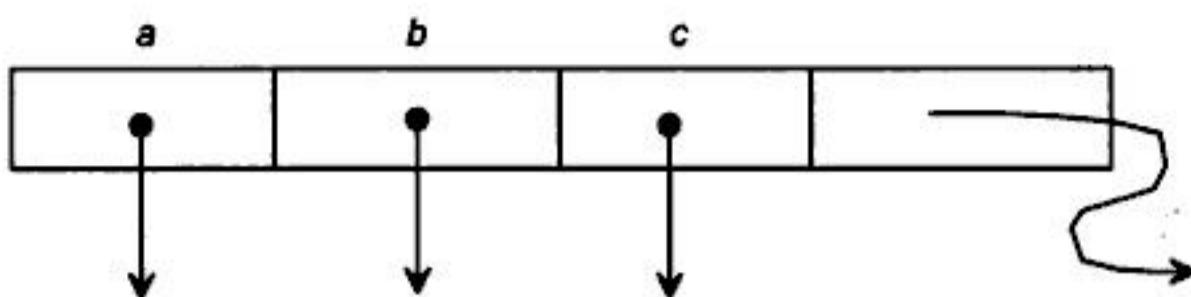


You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

In this figure, each node has the following structure (Figure 7.104):



**Fig. 7.104** Node structure of a trie of order 3.

Here, 3 link fields point to three nodes in the next level, and the last field is called the *information field*. This information field *plays* an important role, its value is either TRUE or FALSE. If the value in this field is TRUE then traversing from the root node to this node yields some information. For example, let us assume the key value '*babc*'. Starting from the root node, and branching based on each letter, our traversal will be 0–2–7–14–19 (as shown by thick links) and in node 19, the information field TRUE implies that '*babc*' is a word (that is, a key value). As an another example, let us see whether '*cab*' is a word in this trie or not. If we search for this, our branching will proceed as 0–3–9; now at node 9, there is no link for '*b*' so '*cab*' is not a word. however, '*ca*' is a word. Likewise, one can easily verify that '*abc*' is not a word. The above trie structure as shown in Figure 7.103 stores 22 different words.

Several important points that can be noted from the above illustration are listed below:

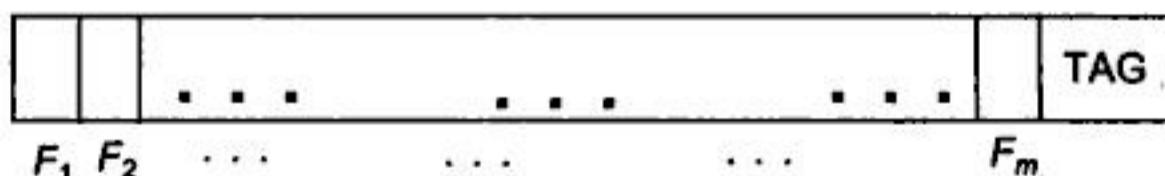
- Trie indexing is suitable for maintaining variable sized key values.
- Actual key value is never stored but key values are implied through links.
- If English alphabets are used, then a trie of order 26 can maintain whole English dictionary. (This is specially termed as *lexicographic trie*).
- It allows us multi-way branching based on the part of key value, not the entire key value. The branching on the *i*-th level is determined by the *i*-th component of the key value.

### 7.9.2 Operations on Trie

Let us consider the searching, insertion and deletion operations on a trie structure of order *m*.

#### Searching

In fact, all the operations discussed in Section 7.9.1 are straightforward and do not require much elaboration. We will assume the following node structure in a trie.



Here, *m* fields are there. These *m* fields ( $m \geq 2$ ) hold *m* different components that are possible to constitute a key value. TAG is a boolean field containing either TRUE or FALSE (1 or 0). Also, we will assume a key value as  $C_1 C_2 \dots C_n$ , which is constituted with *n* key components ( $n > 0$ ). Consider the following algorithm:



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.

**Assignment 7.42**

1. Modify all the algorithms on B tree so that they will be fit for maintaining any file organization. (You have to modify the algorithms in such a way that these can take care of the pointers to the records as well.)
2. Assume a suitable file organization and a key field in it. Then build a B tree index for all the records in the file. Use this tree to search for any record in the file.
3. The tree indexing with B tree can be extended for more than one level as well. This is highly advisable when the file structure as well as the number of records in the file is very large. As an example, for the file structure as shown in Figure 7.105, in the first level, the indexing can be done on the COUNTRY field and in the second level it is on the RANKING field.  
Suggest a way, how this can be done. What special record organization you should adopt for the purpose.
4. For B tree indexing, what factor(s) will influence choosing the order of the tree? Give reasons with example(s).

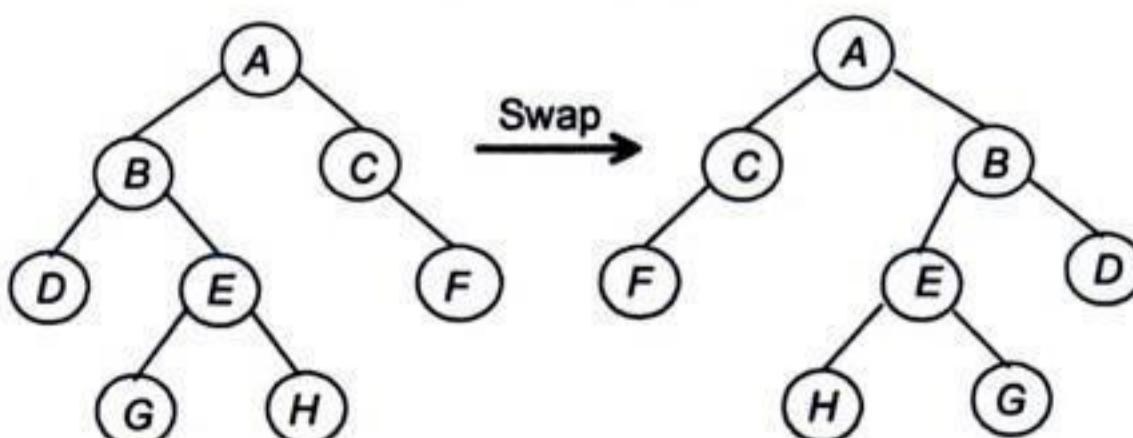
**PROBLEMS TO PONDER**

- 7.1** In a binary tree, a *full* node is defined as a node with two children, and *empty* node is defined as a node with no children. Show that  $n_1 + 1 = n_0$ , where  $n_0$  = number of empty nodes and  $n_1$  = number of full nodes.
- 7.2** Write an algorithm to count the
- (a) Number of nodes with degree 2.
  - (b) Number of nodes with degree 1.
  - (c) Number of nodes with degree 0.
  - (d) Number of edges in a binary tree.

Assume that a binary tree is represented with:

1. Sequential storage representation.
2. Linked storage representation.

- 7.3** The swap operation on a binary tree can be done by interchanging the left and right children of every node. For example, Figure 7.107 illustrates the swap operation on a binary tree. Obtain an algorithm SWAPTREE ( $T$ ) to obtain the swap of a binary tree  $T$ .



**Fig. 7.107** Swap operation on a binary tree.



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.



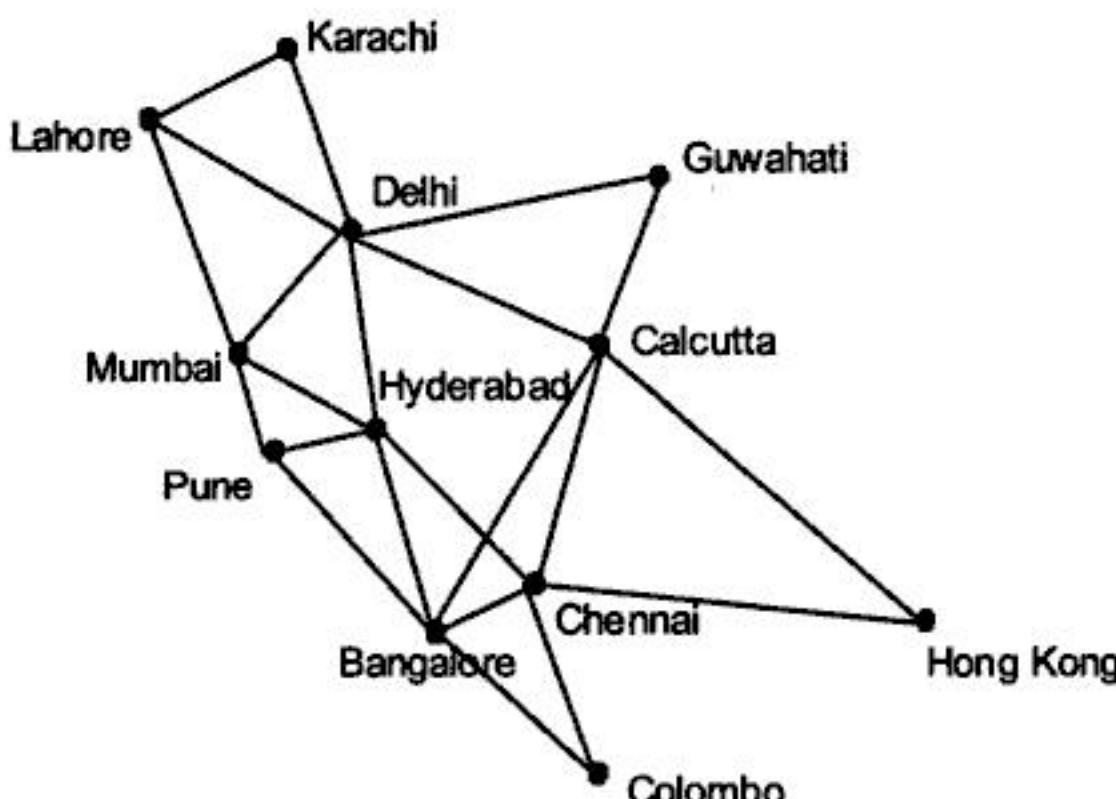
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

whether it is possible to cross all the bridges exactly once and return to the starting land area. This problem is known as the *Konigsberg's bridge problem*.

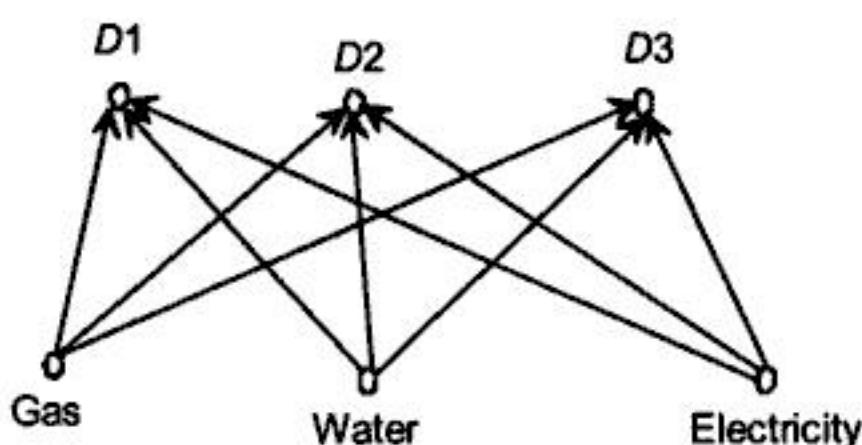
### Flowchart of a program

Flowchart of a program is in fact the graphical representation of an algorithm of a problem. Figure 8.2(d) represents a flowchart.

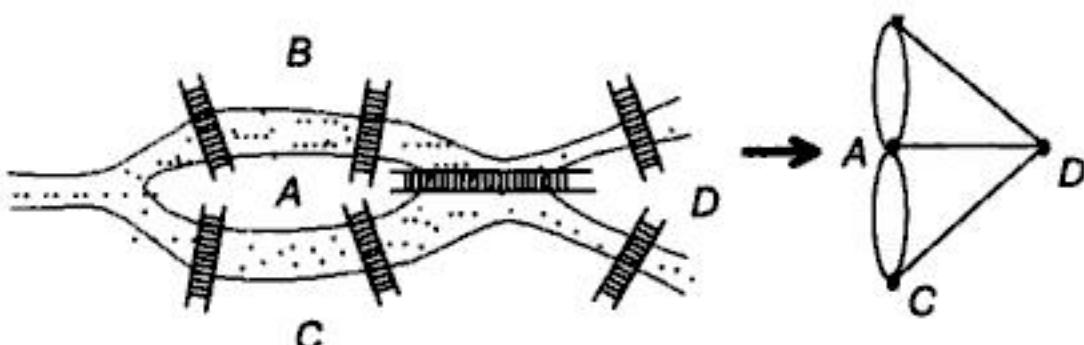
Similarly, almost in all application areas, graph structures exist.



(a) Graphical representation of airlines



(b) Graphical representation of three commodities in three destinations



(c) Konigsberg's bridges and their graphical representation

**Fig. 8.2** Continued.



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.



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.



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.



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.



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.

**Assignment 8.17**

1. Modify the algorithm KRUSKAL to take into account the following considerations:
  - (a) Calculate the sum of the weights of the resultant spanning tree.
  - (b) Obtain the minimum spanning tree of the directed graph.
2. We have considered the spanning tree of minimum length. In some application, it desires to obtain a longest spanning tree, that is, a spanning tree where the sum of the weights of the edges in the tree is maximum. How can the algorithm KRUSKAL be modified to accomplish this? Verify your suggestion with a suitable example.
3. The diameter of a tree is defined as the maximum distance between any two vertices. For given a connected and undirected graph, devise an algorithm for finding a spanning tree of minimum diameter. Verify the correctness of your algorithm with an example.

**Prim's algorithm**

This approach does not require listing of all edges in increasing order of weights or checking at each step that whether a newly selected edge forms a circuit or not. According to *Prim's* algorithm, minimum spanning tree grows in successive stages: at any stage in the algorithm, we can see that we have a set of vertices that have already been included in the tree, the rest of the vertices have not. The prim's algorithm then finds a new vertex to add it to the tree by choosing the edge  $\langle V_i, V_j \rangle$ , the smallest among all edges, where  $V_i$  is in the tree and  $V_j$  is yet to be included in the tree. The algorithm starts by selecting a vertex arbitrarily, and then in each stage, we add an edge (by adding an associated vertex) to the tree.

The Prim's algorithm can easily be implemented using the adjacency matrix representation of a graph. Suppose, there is an  $N \times N$  adjacency matrix for a given undirected weighted graph, and vertices are labelled as  $v_1, v_2, \dots, v_N$ . Start from vertex  $v_1$ , say. Get its nearest neighbour, that is, the vertex which has the smallest entry in row of  $v_1$  (if more than one smallest entries are there, arbitrarily select any one); let it be  $v_i$ . Now consider  $v_1$  and  $v_i$  as one sub-graph. Next, obtain the closest neighbour to this sub-graph, that is a vertex other than  $v_1$  and  $v_i$  that has the smallest entry among all entries in the rows of  $v_1$  and  $v_i$ . Let this new vertex be  $v_j$ . Therefore, the tree now grows with vertex  $v_1, v_i$  and  $v_j$  as one sub-graph. Continue this process, until all  $N$  vertices have been connected by  $N - 1$  edges.

Let us now illustrate the above method of finding minimum spanning tree. An undirected weighted graph and its weighted adjacency matrix is shown in Figure 8.39(a). We start with  $v_1$  and pick the smallest entry in row 1 which is 2 in column 4; thus  $v_4$  is the nearest neighbour to  $v_1$ . The closest neighbour of sub-graph  $v_1$  and  $v_4$  is  $v_2$ , that can be seen by examining all the entries in row 1 and 4. The four remaining edges selected following the above procedure are  $\langle v_2, v_5 \rangle, \langle v_4, v_6 \rangle, \langle v_6, v_3 \rangle$  and  $\langle v_6, v_7 \rangle$ . See Figure 8.39(b); the total length of the tree can be calculated as 16.

Let us discuss the implementation of the Prim's algorithm in details.

An array **SELECTED[1... N]** of Boolean will be used in our algorithm and **SELECTED[i] = TRUE** means that  $i$ -th vertex is included in the tree. The output of the minimum spanning tree will be stored in an adjacency matrix of order  $N \times N$ . In the adjacency matrix of the given graph, we assume the  $(i, j)$  entry as infinity (a large positive value) if there is no edge between the vertices  $i$  and  $j$ . The algorithm PRIM is described as below.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

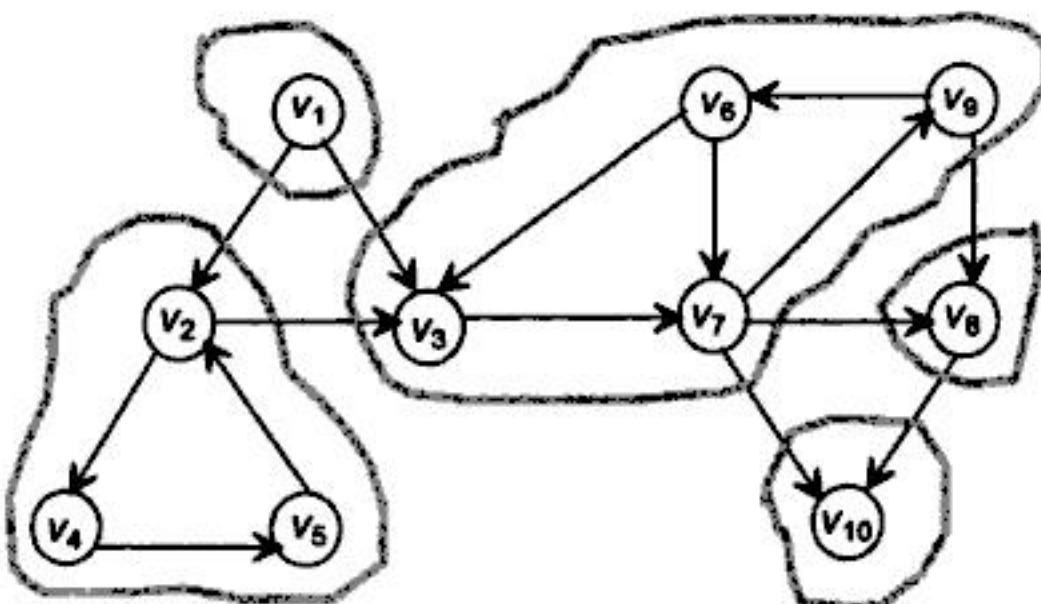


You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

connectivity is specially termed as strong connectivity and it imposes that if there is a path from  $v_i$  to  $v_j$  then there should be another path from  $v_j$  to  $v_i$  too. In a graph, there may be more than one set and within a set all the vertices are connected (or strongly connected). Each set is known as a *component*. Note that, if a graph is connected then number of component is one and all vertices in the component are strongly connected. The problem is more important if the graph is directed and each strongly connected component results a sub-graph of the graph. To illustrate the above, let us consider a digraph as shown in Figure 8.41. This graph consists of five strong components:  $\{v_1\}$ ,  $\{v_2, v_4, v_5\}$ ,  $\{v_3, v_6, v_7, v_9\}$ ,  $\{v_8\}$  and  $\{v_{10}\}$ .



**Fig. 8.41** A digraph and its various strong components.

Let us see the method of determining the strong components in a digraph. The entire method is illustrated in Figure 8.42 and explained as below:

**Step 1:** Perform the DFS traversal(s) on the given graph. To do this, select a vertex randomly as starting vertex and mark the vertex as ‘visited’ after it is visited. DFS traversal should not visit a vertex which is marked as ‘visited’ already. This procedure has to be repeated till all the vertices are not visited. This will result a set of DFS spanning trees, called *DFS spanning forest*.

Consider Figure 8.42(a), the vertex first selected randomly is  $v_6$  and starting with this vertex, DFS spanning tree is shown in Figure 8.42(b)-(i). This traversal visits  $\{v_6, v_3, v_7, v_8, v_9\}$ . Then for the further DFS traversal, another vertex which is not yet ‘visited’ selected as  $v_2$ . Second DFS spanning tree starting with  $v_2$  is shown in Figure 8.42(b)-(ii). This traversal visits  $\{v_2, v_4, v_5\}$ . The remaining vertex which is not yet ‘visited’ is  $v_1$ . Starting with this vertex DFS spanning tree is shown as in Figure 8.42(b)-(iii).

**Step 2:** Obtain the reverse graph  $G'$  of  $G$  by reversing the edges of  $G$  in  $G'$ , that is, if there is an edge directing from  $v_i$  to  $v_j$  then in  $G'$  that edge will appear as  $v_j$  to  $v_i$ .

The reverse graph  $G'$  of  $G$  is shown as in Figure 8.42(c).

**Step 3:** All the vertices in  $G'$  are to be numbered with the help of post-order traversals on DFS spanning forests as obtained in step 1.

Performing the post-order traversals on DFS spanning trees, one-by-one, vertex are numbered according to the order of their visit which is shown in Figure 8.42(d). These numbers are attached to the corresponding vertices in  $G'$ . See Figure 8.42(c).



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.

DFS traversal at 5 results  $\text{PEC}(5) = [5-7-6-5]$ . Therefore, we obtain,  $\text{PEC}(1) = [1-2-5-7-5-4-2-3-1]$ . The vertex 6 is then found as first vertex in  $\text{PEC}(1)$  which has still not traversed edge, and  $\text{PEC}(6) = [6-4-3-6]$ . Splicing this with  $\text{PEC}(1)$ , we get  $\text{PEC}(1) = [1-2-5-7-6-4-3-6-5-4-2-3-1]$ . There is no more vertex in this latest  $\text{PEC}(1)$  which has not traversed edge. The procedure is terminated successfully giving the Euler's circuit as shown in Figure 8.47(c).

Detail of the algorithm for the above mentioned method is left as an exercise to the reader.

#### Assignment 8.21

1. Trace the Euler's path and Euler's circuit in the graph  $G1$  and  $G2$  respectively as given in Figure 8.46 using the method described.
2. We have discussed the Euler's circuit in an undirected graph, same way, it can be applied for directed graph also. Verify that:
  - (a) A directed graph has Euler's circuit if and only if it is strongly connected
  - (b) Every vertex has equal indegree and outdegree.

#### **Hamiltonian path and circuit in a graph**

In contrast to Euler's paths, where all edges are traversed exactly once, a Hamiltonian path is one in which all vertices are traversed exactly once. Despite the similarity of definitions the theory of Hamiltonian paths is significantly more intricate than the theory of Euler's path.

Similarly, a *Hamiltonian circuit* in a graph  $G$  is a circuit in which each vertex of  $G$  appears exactly once except for the starting vertex (it is also ending vertex), which appears just twice.

A graph is a *Hamiltonian graph* if it has a Hamiltonian circuit.

In Figure 8.48, Hamiltonian path and Hamiltonian circuit in graph  $G1$  and  $G2$  is shown. Obviously, not every connected graph has a Hamiltonian circuit. For example, the graph  $G3$  (as shown in the figure) is not a Hamiltonian graph.

The question, therefore that arises: How to decide whether a given graph has a Hamiltonian circuit? This problem, first posed by the famous Irish mathematician Sir William Rowan Hamilton in 1859 which is still unsolved. However, there are a few observations which gives us idea to decide whether a graph has a Hamiltonian circuit or not.

1. A complete graphs of three or more vertices have Hamiltonian circuit (necessary and sufficient condition).
2. In a connected graph with  $n$  vertices, for any pair of vertices  $v_i, v_j$  that are not connected by an edge, and if  $\text{degree}(v_i) + \text{degree}(v_j) \geq n$ , where  $\text{degree}(v)$  denotes the degree of a vertex  $v$ , then there is a Hamiltonian cycle on the graph (this is known as Ore's theorem and gives a sufficient but not necessary condition).
3. In a simple connected graph  $G$ , if  $v_i$  be any vertex in  $G$  and  $\text{degree}(v_i) \geq n/2$ , where  $\text{degree}(v_i)$  denotes the degree of vertex  $v_i$  and  $n$  is the number of vertices, then the graph  $G$  has a Hamiltonian circuit. This theorem is due to G.A. Dirac, and also gives sufficient but not necessary condition.

Proofs of the above mentioned conditions are beyond the scope of this book.

There is no formal method known till time to trace the Hamiltonian circuit, if it exists, in a given graph. The problem in fact is known as *NP complete* problem. Theoretically, however, the problem can be solved using 'brainless' method: taking all possible permutations of all the vertices in the graph and then trace the graph according to a sequence of vertices in a permutation and test whether it satisfies the criteria of Hamiltonian circuit or not.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



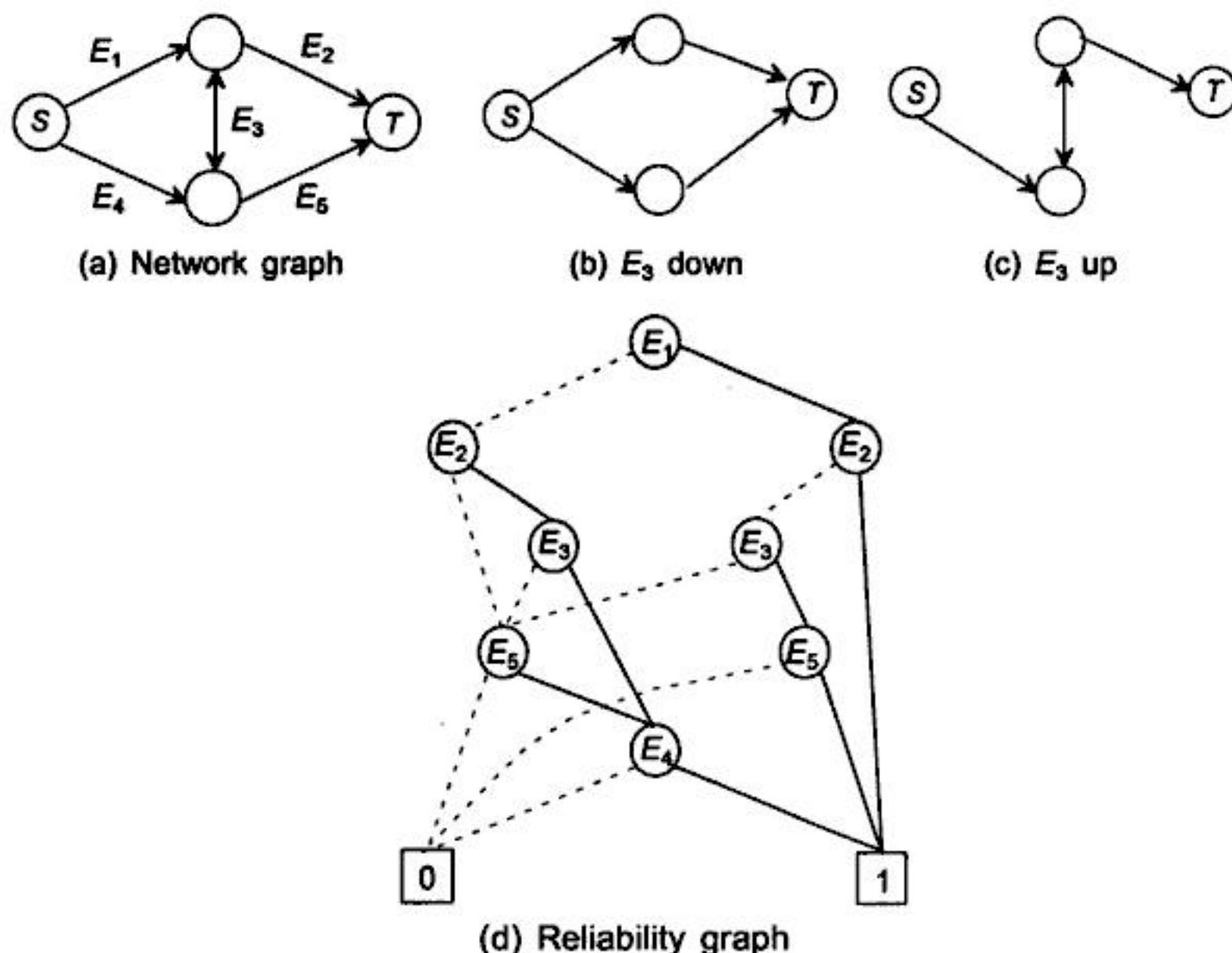
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

### Reliability analysis

Another simple application of the BDD graph is in analysis of reliability. Reliability analysis is an important issue in systems design, manufacturer and maintenance. We will see how BDD graph can be applied to address this kind of problem. To be specific, let us consider the case of *network reliability problem* (this problem is also alternatively termed as *source-termination connectedness problem*). Let us consider the graphs in Figure 8.52(a) which models a network system as a token example.



**Fig. 8.52** A network system and its reliability graph.

In Figure 8.52, two special nodes  $S$  and  $T$  are decided as source node and sink node respectively. These nodes are connected via links  $E_1, E_2, \dots$ , etc., through intermediate nodes. The network system which is represented by such a graph fails when there is no path from the source to the sink. The links connecting the vertices are having finite probability of reliability. Now the problem to decide how the network system is reliable under certain failure conditions of link(s). This problem can be answered very easily, if the network system is very small. But in real applications, even a simple network is very complicated and therefore finding the solution is computationally difficult. However, the problem can be solved very efficiently using BDD graph. Given a network system, a BDD graph can be obtained from it depicting all possible paths. And then this graph can be analyzed under certain failure condition of links.

A suitable method can be formulated to obtain the Boolean expression (where links are to be taken as Boolean variables) for the given network. As an illustration, the Boolean expression of the network in the current example will be obtained as



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.

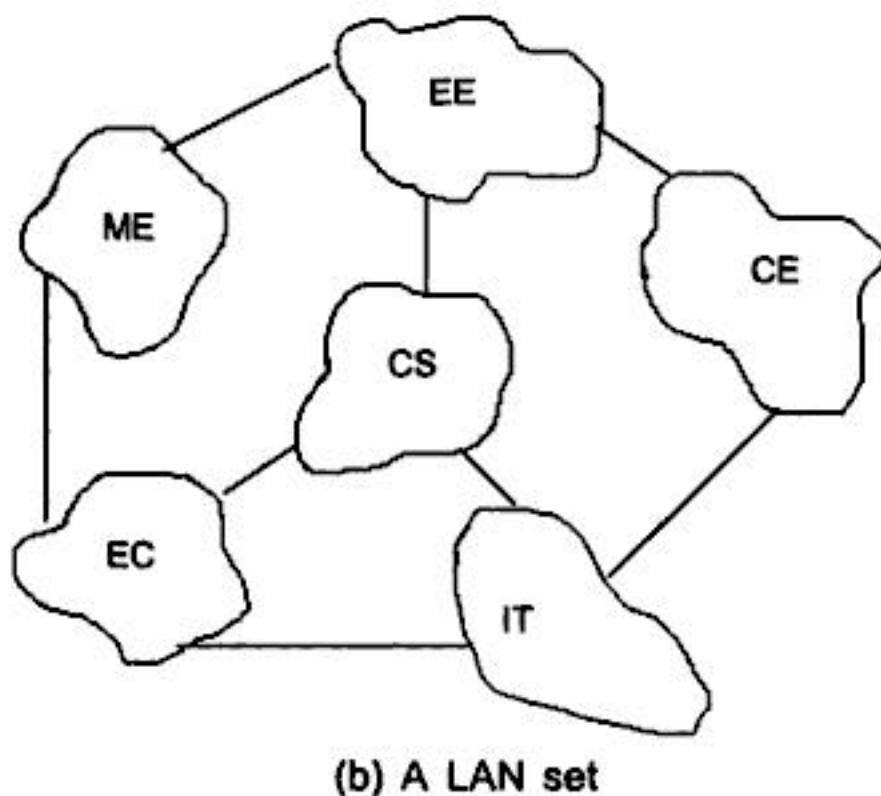
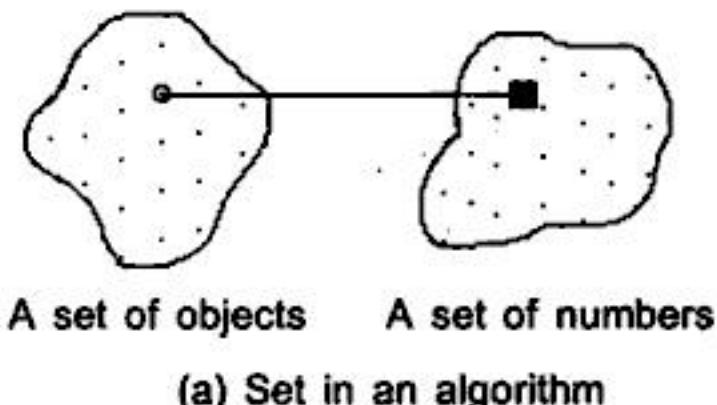
# 9 Sets

Set is such a well-known concept that it is not worthy to devote an introductory section for it. Although, the concept of set is a basic theory in modern mathematics, it also plays an important role in theory of computation; in fact, sets can be seen as integral part in many algorithms. Let us state a few examples, where we can see the presence of sets and how they are assimilated in computing.

**Example 1:** Consider a set of objects, and a unique number is to be assigned to each object. Now, this can be accomplished by generating a set of integers using some random number generation and then each object can be assigned a number from this set of integers.

**Example 2:** A cricket team of a country constitutes a set and elements in that set are the cricket players. Now, if we want to store the information of all the cricketers in the team then a table as shown in Figure 9.1(c) can be obtained. This table, in fact, is a set of records of the cricketers in the cricket team. This kind of set can be observed in any database system.

**Example 3:** Suppose, there is a LAN connecting all the departments in an institution as shown in Figure 9.1(b). An IBM-alpha system is there in the computer centre of the institute. At a certain time, the head of the computer centre wants to send some message to all the users of IBM-alpha. To do this, first a set of paths from the head of the computer centre to all the recipients has to be sorted out and then the message can be routed accordingly.



**Fig. 9.1** Continued.



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.

These sets are very useful in several applications. Not only this, implementation of various set operations is very easy and performance of these operations are much better. In Section 9.3.3, set operations using bit arrays will be discussed in details.

#### 9.2.4 Tree Representation of Set

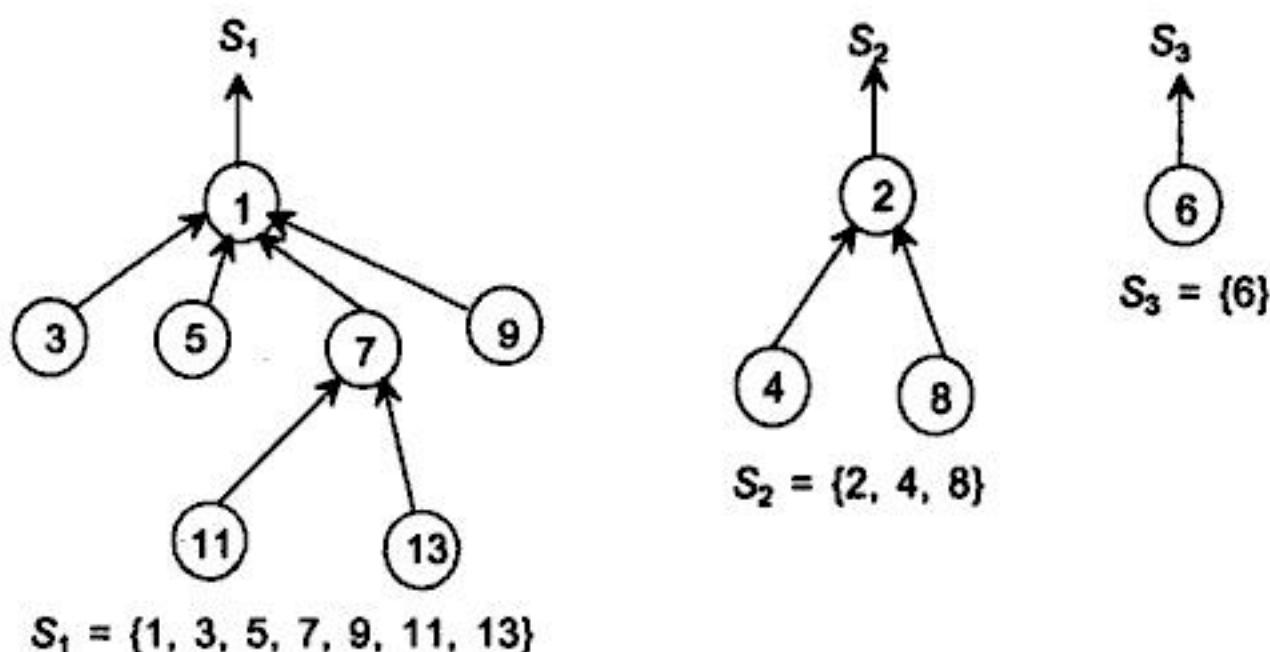
In the tree representation of set, a tree is used to represent one set, and each element in the set has the same root. The trees are not necessarily binary trees. For examples, let us consider three sets:

$$S_1 = \{1, 3, 5, 7, 9, 11, 13\}$$

$$S_2 = \{2, 4, 8\}$$

$$S_3 = \{6\}$$

Tree representation of the above sets may look as shown in Figure 9.7.



**Fig. 9.7** Tree representation of sets.

In this representation, each element in a set has pointer to its parent. For convenience, we draw the root's parent pointer vertically. The name of a set is given by the node at the root, in general.

To store a tree representing a set is very easy. An array is sufficient to store the tree form of a set. We will assume the elements are numbered as integers and indices of array indicates the elements. The array content at the  $i$ -th index is the parent of the  $i$ -th element in the tree. With this, the explicit tree representation using arrays for the set  $S_1$ ,  $S_2$  and  $S_3$  will appear as shown in Figure 9.8.

In this representation, a zero is stored as the pointer to the root. In fact, in actual implementation, a set is identified by its array and pointer of this array is the pointer to the root node which is implicitly denoted as zero.

A collection of disjoint sets can be stored in a single array. For example,  $S_1$ ,  $S_2$  and  $S_3$  as above, all are disjoint. These disjoint sets are stored in the same array as shown in Figure 9.8(d).

Another modification is possible with which we can incorporate more information about a set. As we have learnt, zero indicates the element is in root. Now, this zero can be replaced by the cardinality of the set. Positive entries are pointers to parents of element. The modified



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.

**Algorithm UNION\_LIST\_SET( $S_i, S_j; S$ )**

**Input:**  $S_i$  and  $S_j$  are the headers of two single linked lists representing two distinct sets.

**Output:**  $S$ , the union of two sets  $S_i$  and  $S_j$ .

**Data structure:** Linked list representation of set.

**Steps:**

```

/*To get a header node for S and initialize it*/
1. $S = \text{GETNODE}(\text{NODE})$ //Get a pointer to a node
2. $S.\text{LINK} = \text{NULL}$, $S.\text{DATA} = \text{NULL}$ //Initialization of the new node
 /*To copy the entire list of S_i into S */
3. $\text{ptr}_i = S_i.\text{LINK}$
4. While ($\text{ptr}_i \neq \text{NULL}$) do
 1. $\text{data} = \text{ptr}_i.\text{DATA}$ //Copy from S_i
 2. $\text{INSERT_SL_FRONT}(S, \text{data})$ //Include it into the resultant set S
 3. $\text{ptr}_i = \text{ptr}_i.\text{LINK}$ //Move to the next node in S_i
5. EndWhile
6. $\text{ptr}_j = S_j.\text{LINK}$ //Pointer to the first node in S_j
 /*For each element in S_j , add it to S if it is not in S_i */
7. While ($\text{ptr}_j \neq \text{NULL}$) do
 1. $\text{ptr}_i = S_i.\text{LINK}$ //Search for the present element in S_i
 2. While ($\text{ptr}_i.\text{DATA} \neq \text{ptr}_j.\text{DATA}$) do
 1. $\text{ptr}_i = \text{ptr}_i.\text{LINK}$
 3. EndWhile
 4. If ($\text{ptr}_i = \text{NULL}$) then //When the element is not in S_i
 1. $\text{INSERT_SL_FRONT}(S, \text{ptr}_j.\text{DATA})$
 5. EndIf
 6. $\text{ptr}_j = \text{ptr}_j.\text{LINK}$ //Move to the next element in S_j
8. EndWhile
9. Return(S)
10. Stop

```

In the above mentioned algorithm, we have assumed the method  $\text{INSERT\_SL\_FRONT}(S, \text{KEY})$  to insert a node containing the data value  $\text{KEY}$  at the front of the linked list  $S$ . This procedure can be obtained from Chapter 3.

**Intersection**

Like the union operation, the intersection operation can be defined by searching a list for each element in the other list. Suppose,  $S_i$  and  $S_j$  are the two sets in the form of linked list. We are interested in finding  $S \equiv S_i \cap S_j$ , the intersection of  $S_i$  and  $S_j$ . Initially,  $S$  is empty. For each element in  $S_i$ , we have to search whether that element is in  $S_j$  or not. If that element is in  $S_j$ , we are to insert a node corresponding to that element in  $S$ .

In Figure 9.14, the intersection of two sets is illustrated. The detail of the procedure is described in the algorithm INTERSECTION\_LIST\_SET.



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.

**Difference**

In similar fashion, the *difference* of two sets  $S_i$  and  $S_j$  in hash table representation can be expressed. The algorithm DIFFERENCE\_HASH\_SET to perform the set difference operation is stated as below:

**Algorithm DIFFERENCE\_HASH\_SET ( $S_i, S_j; S$ )**

**Input:**  $S_i$  and  $S_j$  are two sets in the form of hash table, size of each of the hash table is assumed as  $N$ .

**Output:**  $S = S_i - S_j$ , a hash table form of a set giving the intersection of  $S_i$  and  $S_j$ .

**Data structure:** Linked list representation of set.

**Steps:**

1. For  $k = 1$  to  $N$  do
  1.  $S[k] = \text{DIFFERENCE\_LIST\_SET}([S_i[k], S_j[k]])$  //Difference of  $k$ -th buckets in  $//S_i$  and  $S_j$
2. EndFor
3. Return( $S$ )
4. Stop

**Equality**

To test the *equality* of two sets  $S_i$  and  $S_j$  in the form of hash tables, we can employ the operation EQUALITY\_LIST\_SET repeatedly for every pair of buckets in two tables. The algorithm EQUALITY\_HASH\_SET is stated as below to implement this:

**Algorithm EQUALITY\_HASH\_SET ( $S_i, S_j$ )**

**Input:**  $S_i$  and  $S_j$  are the two sets in the form of hash table. Size of each of the hash table is assumed as  $N$ .

**Output:** Returns TRUE if the two sets  $S_i$  and  $S_j$  are equal else it returns FALSE.

**Steps:**

1. flag = TRUE //Initially two sets are assumed to be equal
2.  $k = 1$  //Starting hash location
3. While (flag) and ( $k < N$ ) do
  1. flag = EQUALITY\_LIST\_SET ( $S_i[k], S_j[k]$ )
  2.  $k = k + 1$  //Move for the next pair of buckets
4. EndWhile
5. Return(flag)
6. Stop

It may be noted that, in the above described algorithms, for implementing the operations of sets represented with hash tables, we have assumed that the hash table entries are pointer to the headers of respective buckets.

**Assignment 9.4** In Section 9.3.2, we have described the different operations on set represented with hash table. For each of the operation we have assumed that the size of the hash tables are same. But the operations are also permissible when the size of the hash table representing two sets are not necessarily the same. Modify all the algorithms in this section so that the operations can handle two sets whose hash table sizes are unequal.



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.

**Case 2:** Exclusion of an element which is not a leaf node.

- Let the element to be excluded is  $i$ .
- Traverse the PARENT array to select an element  $j$  such that it is a successor of  $i$  and is a leaf node.

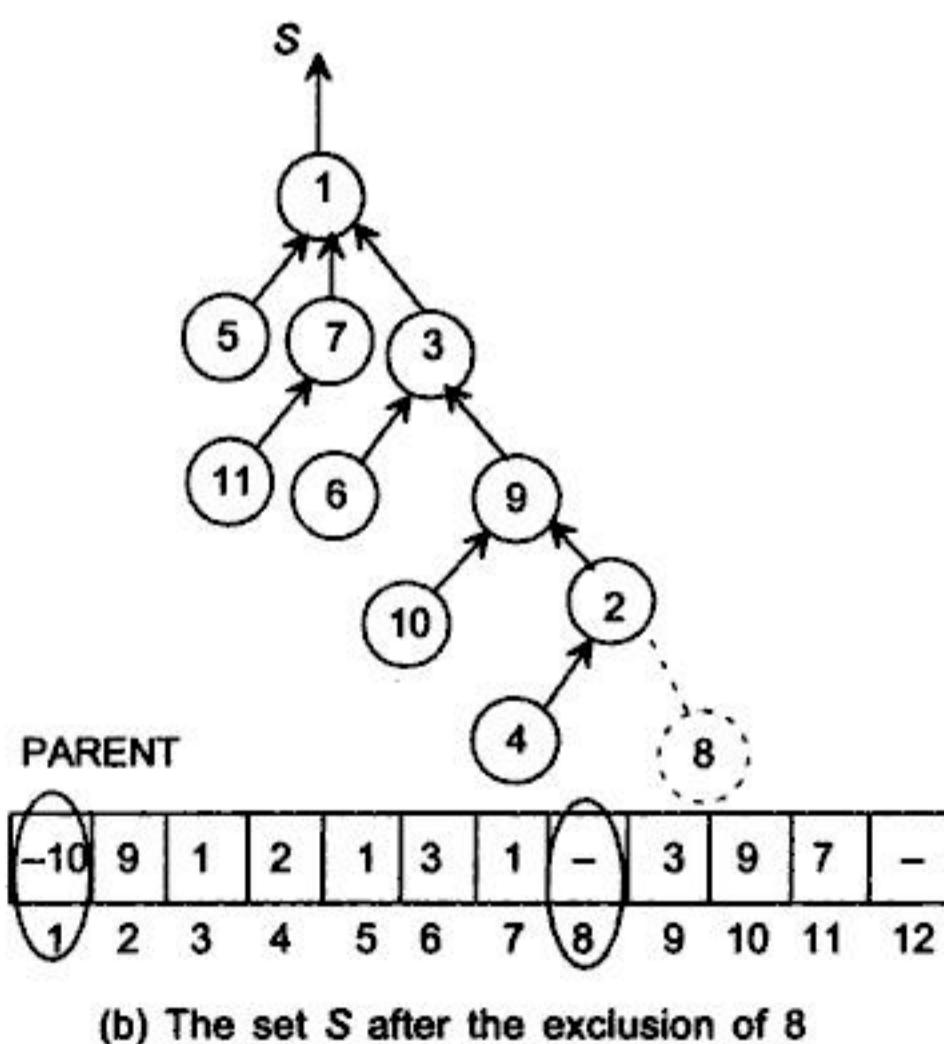
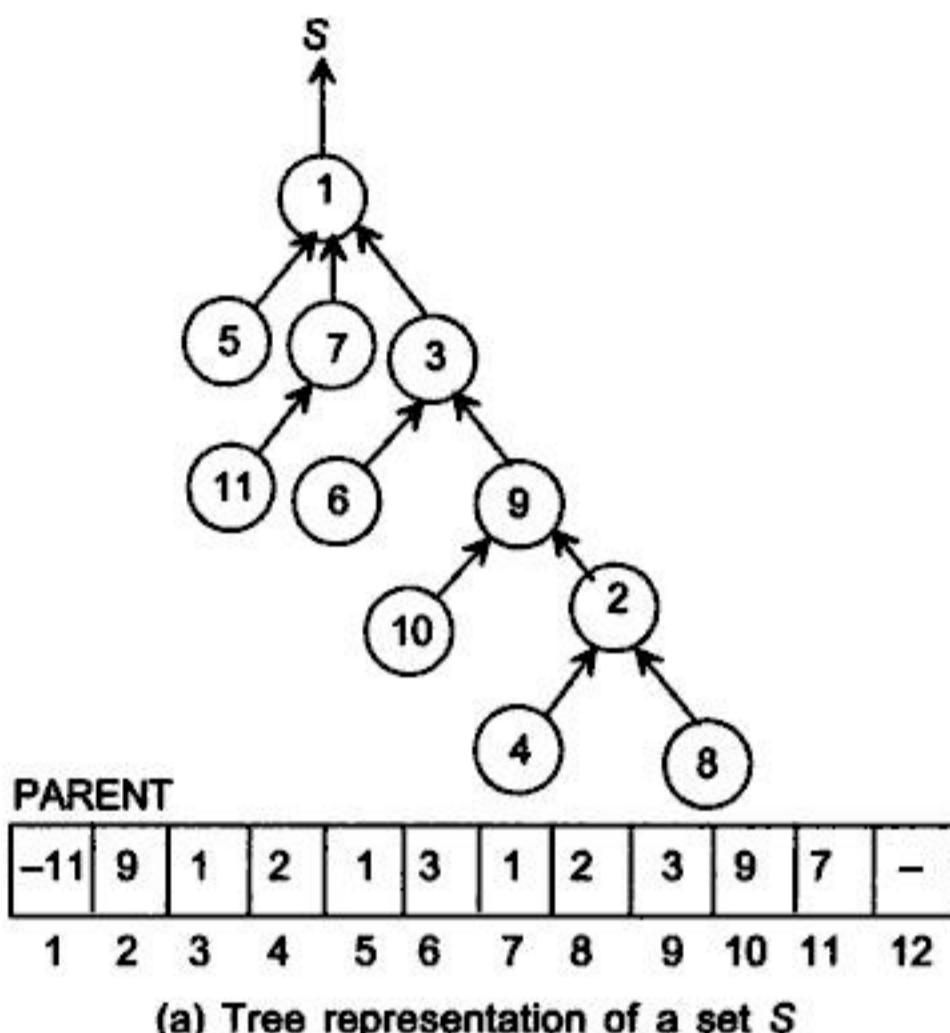


Fig. 9.17 Continued.



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.

# **Index**

---

- Access table, 181  
Activation record, 130–131  
  management, 127–143  
Acyclic graph, 360  
Adjacency matrix, 365–370  
Array, 10–30  
  deletion, 17  
  insertion, 15–16  
  merging, 18  
  n-dimensional, 29–30  
  one-dimensional, 11–20  
  searching, 15  
  sorting, 14  
  three-dimensional, 28–29  
  traversing, 13–14  
  two-dimensional, 20–21  
Articulation point, 419–420  
AVL rotations, 281–293  
  left-to-left, 281–282  
  left-to-right, 284–285  
  right-to-left, 285–286  
  right-to-right, 283
- B tree, 318–350  
  application, 349–350  
  bursting, 326  
  combined, 331–333  
  deletion, 330–340  
  display, 340–341  
  indexing, 319  
  insertion, 322–330  
  lower bound, 342–343  
  searching, 321–322  
  upper bound, 342  
B+ tree, 343–345  
  deletion, 344–345  
  indexing, 343  
  insertion, 344  
  searching, 343–344  
Back edge, 420  
Bag, 435, 436  
Balance factor, 279–280  
BDD, *see* binary decision diagram  
Best fit, 68, 74, 75  
BFS spanning tree, 408  
BFS traversal, 380–381  
Biconnected graph, 419  
Binary decision diagram, 425–430  
Binary search tree, 244–253  
  deletion, 248–253  
  insertion, 247–248  
  searching, 245–246  
  traversal, 253  
Binary sort, 253  
Binary sorted tree, 253  
Binary tree, 207–239  
  deletion, 224–226  
  insertion, 220–223  
  linear representation, 213–216  
  linked representation, 216–218  
  merging, 237–239  
  representation of forest, 316–317  
  traversal, 227–239  
Bit array data structure, 438  
Bit matrix, *see* adjacency matrix  
Bit string, 462–464  
Boolean matrix, *see* adjacency matrix  
Boolean satisfiability, 307–308  
Boolean tree, *see* decision tree  
Boundary tag system, 68, 72–77  
Breadth first search, 377–378  
Bucket hashing, 197–198  
Buddy system, 68, 82–91  
  binary, 82, 91–92  
  Fibonacci, 83, 92
- Chaining, *see* open hashing  
Circular queue, 153–156  
Client server environment, 464–465  
Collision, 187  
Collision resolution techniques, 191–200  
  closed hashing, 191–195  
  open hashing, 191, 196–200  
Column major order, 20, 21, 24, 25  
Compaction, 92–95  
  incremental, 94  
  selective, 94  
Complete binary tree, 209

- Complete graph, 360  
 Connected graph, 361, 415–421  
 Cut vertex, *see* articulation point
- Data, 1–2  
 Data structure, 4–7  
     dynamic, 34  
     linear, 6  
     nonlinear, 6  
     static, 34  
 Data type, 3–4  
     abstract, 3  
     built-in, 3  
     user-defined, 3–4  
 Deallocation strategy, 77–84  
     boundary tag system, 77–82  
     buddy system, 84  
 Decision tree, 305–308, 425–426  
 Defragmentation, *see* compaction  
 Depth-first search, 377  
 Deque, 156–158  
     input restricted, 158–159  
     output restricted, 158–159  
 DFS spanning forest, 416–417  
 DFS spanning tree, 408, 416  
 DFS traversal, 379–380  
 Digraph, 358  
 Dijkstra's algorithm, 399–403  
 Directed graph, *see* digraph  
 Double hashing, 194  
 Double linked list, 51–59  
     circular, 56–59  
     deletion, 54–56  
     insertion, 51–54  
 Dynamic memory management, 36, 68  
 Dynamic queue, 152  
 Dynamic scope rule, 104, 127, 138–142  
 Dynamic storage allocation, 130
- Elementary path, 368  
 Empty set, *see* null set  
 Entity, 1  
 Euclid's method, 145  
 Euler's circuit, 421–422, 435  
 Euler's graph, 421  
 Euler's path, 421–423  
 Expression tree, 240–244  
 Extended binary tree, 293–294  
 External node, 293  
 External path length, 294–296  
 External weighted path length, 297–299
- Fibonacci sequence, 127  
 First fit, 68, 74  
 First-in first-out, 148  
 Fixed block  
     allocation, 68  
     storage, 68–71  
 Fixed length coding, 301–302  
 Floyd's algorithm, 395–398  
 Forest, 316  
 Fragmentation, 75  
     external, 75  
     internal, 75  
 Full binary tree, 208
- Garbage collection, 36, 68  
 Genealogical tree, 308–309  
 Generic array, 32  
 Graph, 358–391  
     deletion, 370, 375–377, 386–387  
     insertion, 370, 371–374, 383–386  
     linked representation, 363–364  
     matrix representation, 364–365  
     merging, 371, 381–382, 389–391  
     set representation, 363  
     traversal, 371, 377–381, 387–389
- Hamiltonian circuit, 423  
 Hamiltonian graph, 423, 425  
 Hamiltonian path, 423  
 Hash  
     function, 186–189  
     tables, 185–200  
 Hashing techniques  
     digit analysis method, 189  
     division method, 187  
     fold binary method, 188  
     fold shifting method, 188  
     folding method, 188  
     midsquare method, 187–188  
 Heap sort, 260–262  
 Heap tree, 254–259  
     deletion, 257–259  
     insertion, 255–257  
     merging, 259  
 Height balanced  
     binary tree, 278  
     search tree, 280  
 Horner's rule, 144  
 Huffman  
     algorithm, 298–299  
     coding, 302–304  
     tree, 298–304

- Hyper edge, 432  
 Hyper graph, 432
- Incidence matrix, 430, 431  
 Index variable, 11  
 Indexing formula, 12  
 Infix  
     notation, 105  
     to postfix, 107–109  
 Information, 1–2  
 Inorder  
     threading, 265  
     traversal, 229–230  
 Internal node, 293  
 Internal path length, 294–296  
 Inverted tables, 185
- Jagged tables, 182–184
- Konigsberg's bridges, 356–357, 421
- Last-in first-out, *see* stack  
 Lazy deletion, 96  
 Level order traversal, 237  
 Lexicographic trie, 346  
 Linear probing, *see* closed hashing  
 Link, 34, *see also* node  
 Linked list, 34–56  
     circular, 34, 48–51  
     double, 34, 51–56  
     dynamic representation, 36  
     single, 34  
     static representation, 35
- Magic square, 32  
 Map colouring, 392  
 Max heap, 254  
 Memory bank, 36, 68  
 Min heap, 254–255  
 Minimum spanning tree, 407–415  
     degree constraint, 415  
     Kruskal's algorithm, 408–412  
     Prim's algorithm, 412–415  
 Minimum weighted binary tree, 298  
 Multilevel feedback queue strategy, 175–176  
 Multilevel queue scheduling, 175  
 Multigraph, 360  
 Multiple stack, 103  
 Multiplication table, 32  
 Multiqueue, 161–162
- Multiset, 435  
*m*-way search tree, 345  
     *see also* trie tree
- Network  
     graph, 429  
     reliability problem, 429–430
- Next fit, 68, 75  
 Node, 34  
 Null link problem, 49  
 Null set, 436
- One-way list, 34  
 One-way threading, 277  
 Ordered deallocation, 68  
     *see also* dynamic memory management
- Partial Euler circuit, 422–423  
 Path  
     length, 293  
     matrix, 368–370  
 Pointer, 34 *see also* link  
     arrays, 30–31  
 Polish notation, *see* prefix notation  
 Polynomial-linked representation, 63–67  
 Post-order traversal, 229, 231  
 Post-fix  
     evaluation, 110  
     notation, 106  
 Power matrix, 367–368  
 Prefix notation, 105  
 Preorder traversal, 228, 230  
 Priority queue, 159–164  
     array representation, 160–162  
     linked list representation, 162–164  
     using heap tree, 260, 262–264
- Quadratic probing, 195  
 Queue, 148–152  
     array representation, 148–151  
     linked list representation, 152  
 Quick sort, 118–124
- Random  
     deallocation, 68 *see also* dynamic memory management  
     probing, 193  
 Reachability matrix, *see* path matrix  
 Rectangular tables, 181–182  
 Recursion, 115

- Rehashing, *see* double hashing  
 Reliability graph, 429  
 Reverse polish notation, *see* postfix notation  
 Round robin algorithm, 176–179  
 Row-major order, 20, 21, 24, 25  
 Run time stack, 104, 116, 130–131
- Separable graph, *see* biconnected graph  
 Set, 436–460
  - binary relation, 466
  - bit vector representation, 438
  - cardinality, 436
  - difference, 436, 447–448, 452, 454–455
  - disjoint, 436
  - equality, 436–437, 445, 448–449, 452, 455
  - exclusion, 445, 456–460
  - FIND method, 443–444
  - hash representation, 437–438
  - inclusion, 444
  - intersection, 436, 446–447, 451, 453–454
  - list representation, 437
  - tree representation, 439–443
  - union, 436, 445–446, 450–451, 453
  - union method, 441–443
 Shannon's expansion theorem, 306–307  
 Shortest path (single source), *see* Dijkstra's algorithm  
 problem, 393–403  
 Single linked list, 38–48
  - copy, 46
  - deletion, 42–45
  - insertion, 38–42
  - merging, 46–47
  - searching, 47
  - traversing, 37–38
 Skew binary tree, 210, 216  
 Source termination connectedness problem, 429  
 Sparse matrix, 22–26
  - diagonal, 25–26
  - linked representation, 60–63
  - lower triangular, 22–23
  - upper triangular, 24–25
  - tridiagonal, 26–27
 Spelling checker, 460–462  
 Stack, 99–100
  - array representation, 99–100
  - definition, 99
  - linked list representation, 100
 Stack machine, 104, 113–115  
 Static scope rule, 104, 127–137  
 Static storage allocation, 130  
 Static storage management, 67  
 Strong component, 416, 418  
 Strong connectivity, 415–418  
 Strongly connected graph, 361, *see also* strong connectivity  
 Subscripted variable, 11  
 Subset, 436
- Table, 181  
 Tarjan's method, 419–421  
 Threaded binary tree, 264–277
  - deletion, 274–277
  - inorder predecessor, 268–269
  - inorder successor, 268
  - inorder traversal, 269
  - insertion, 269–273
 Topological sorting, 404–407  
 Tower of Hanoi problem, 117, 124–127  
 Transportation problem, 391–392  
 Travelling salesman problem, 425  
 Tree, 207–315
  - 2-3 tree, 326
  - 2-d, 354
  - array representation, 311–312
  - binary tree representation, 312–313
  - diameter, 354
  - inorder traversal, 314
  - isomorphic, 352
  - linked representation, 310–311
  - ordered, 309
  - post-order traversal, 315
  - preorder traversal, 314–315
  - unordered, 309
 Tree edge, 420  
 Trie tree, 345–350
  - application, 350
  - deletion, 348–349
  - indexing, 345
  - insertion, 347–348
  - searching, 346–347
 Two-way threading, 277
- Variable block  
 allocation, 68  
 storage, 71–72  
 Variable length coding, *see* Huffman coding
- Warshall's algorithm, 393–395  
 Weakly connected graph, 361  
 Weighted binary tree, 293–300  
 Weighted graph, 360  
 Weighted path length, *see* extended binary tree  
 Worst fit, 68, 74

# Classic Data Structures

## D. Samanta

This text is designed for an introductory undergraduate course in data structures for computer science and engineering students. Written in a very accessible style, the book is also appropriate for students of polytechnics who will immensely benefit from its clear and concise analytic explanations presented in simple language. The book describes different types of classic data structures such as arrays, linked lists, stacks, queues, tables, trees, graphs and sets, providing a deep understanding of the essential concepts. To make the book both versatile and complete, the readers are exposed to the full range of design concepts featuring all the important aspects of each type of data structure. Among other distinguishing features, the book:

- ◆ Shows various ways of representing a data structure
- ◆ Examines different operations to manage a data structure
- ◆ Illustrates several applications of a data structure.

In this title, the algorithms are presented in English-like constructs instead of programming codes for ease of comprehension by students, without being bogged down in the details of a particular programming language. The algorithms described are nonetheless suitable for generic implementation, particularly with C++ and Java.

The book includes numerous exercises in the form of section-wise assignments and end-of-chapter problems to enable readers to build understanding of the concepts and acquire problem-solving skills.

**D. SAMANTA**, Ph.D is Assistant Professor, School of Information Technology, Indian Institute of Technology Kharagpur. He is the author of *Object-Oriented Programming with C++ and Java* (Prentice-Hall of India, 2000).

ISBN 81-203-1874-9



9 788120 318748

To learn more about  
**Prentice-Hall of India** products,  
please visit us at : [www.phindia.com](http://www.phindia.com)

**Rs. 225.00**