



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.

DATA STRUCTURES AND ALGORITHMS

Concepts, Techniques and Applications



GAVPAI

Copyrighted material



Tata McGraw-Hill

Published by the Tata McGraw-Hill Publishing Company Limited,
7 West Patel Nagar, New Delhi 110 008.

Copyright © 2008, by Tata McGraw-Hill Publishing Company Limited.

No part of this publication may be reproduced or distributed in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise or stored in a database or retrieval system without the prior written permission of the publishers. The program listings (if any) may be entered, stored and executed in a computer system, but they may not be reproduced for publication.

This edition can be exported from India only by the publishers,
Tata McGraw-Hill Publishing Company Limited

ISBN (13): 978-0-07-066726-6

ISBN (10): 0-07-066726-8

Managing Director: *Ajay Shukla*

General Manager: Publishing—SEM & Tech Ed: *Vibha Mahajan*

Asst. Sponsoring Editor: *Shalini Jha*

Editorial Executive: *Nilanjan Chakravarty*

Executive—Editorial Services: *Sohini Mukherjee*

Senior Proof Reader: *Suneeta S Bohra*

General Manager: Marketing—Higher Education & School: *Michael J Cruz*

Product Manager: SEM & Tech Ed: *Biju Ganesan*

Controller—Production: *Rajender P Ghansela*

Asst. General Manager—Production: *B L Dogra*

Information contained in this work has been obtained by Tata McGraw-Hill, from sources believed to be reliable. However, neither Tata McGraw-Hill nor its authors guarantee the accuracy or completeness of any information published herein, and neither Tata McGraw-Hill nor its authors shall be responsible for any errors, omissions, or damages arising out of use of this information. This work is published with the understanding that Tata McGraw-Hill and its authors are supplying information but are not attempting to render engineering or other professional services. If such services are required, the assistance of an appropriate professional should be sought.

Typeset at The Composers, 260, C.A. Apt., Paschim Vihar, New Delhi 110 063 and printed at
Pashupati Printers, 429/16, Gali No. 1, Friends Colony, Industrial Area, GT Road, Shahdara, Delhi 110 095

Cover Printer: Rashtriya Printers

RQLCRLXRAQQX

The McGraw-Hill Companies



CONTENTS

<i>Advance Praise</i>	v
<i>More from the Reviewers</i>	vi
<i>Preface</i>	ix

1. Introduction	1
1.1 History of Algorithms	2
1.2 Definition, Structure and Properties of Algorithms	3
1.3 Development of an Algorithm	4
1.4 Data Structures and Algorithms	4
1.5 Data Structure—Definition and Classification	5
<i>Summary</i>	7
2. Analysis of Algorithms	8
2.1 Efficiency of Algorithms	8
2.2 Apriori Analysis	9
2.3 Asymptotic Notations	11
2.4 Time Complexity of an Algorithm Using O Notation	12
2.5 Polynomial Vs Exponential Algorithms	12
2.6 Average, Best and Worst Case Complexities	13
2.7 Analyzing Recursive Programs	15
<i>Summary</i>	19
<i>Illustrative Problems</i>	20
<i>Review Questions</i>	25
Part I	
3. Arrays	26
3.1 Introduction	26
3.2 Array Operations	27
3.3 Number of Elements in an Array	27
3.4 Representation of Arrays in Memory	28
3.5 Applications	32
<i>Summary</i>	34
<i>Illustrative Problems</i>	35
<i>Review Questions</i>	37
<i>Programming Assignments</i>	37

4. Stacks	39
4.1 Introduction	39
4.2 Stack Operations	40
4.3 Applications	43
<i>Summary</i>	48
<i>Illustrative Problems</i>	49
<i>Review Questions</i>	54
<i>Programming Assignments</i>	55
5. Queues	56
5.1 Introduction	56
5.2 Operations on Queues	57
5.3 Circular Queues	62
5.4 Other Types of Queues	66
5.5 Applications	71
<i>Summary</i>	75
<i>Illustrative Problems</i>	76
<i>Review Questions</i>	81
<i>Programming Assignments</i>	82
Part II	
6. Linked Lists	84
6.1 Introduction	84
6.2 Singly Linked Lists	87
6.3 Circularly Linked Lists	93
6.4 Doubly Linked Lists	98
6.5 Multiply Linked Lists	103
6.6 Applications	105
<i>Summary</i>	112
<i>Illustrative Problems</i>	113
<i>Review Questions</i>	119
<i>Programming Assignments</i>	121
7. Linked Stacks and Linked Queues	123
7.1 Introduction	123
7.2 Operations on Linked Stacks and Linked Queues	124
7.3 Dynamic Memory Management and Linked Stacks	130
7.4 Implementation of Linked Representations	132
7.5 Applications	133
<i>Summary</i>	137
<i>Illustrative Problems</i>	137
<i>Review Questions</i>	148
<i>Programming Assignments</i>	149
Part III	
8. Trees and Binary Trees	151
8.1 Introduction	151

8.2	Trees: Definition and Basic Terminologies	151
8.3	Representation of Trees	153
8.4	Binary Trees: Basic Terminologies and Types	155
8.5	Representation of Binary Trees	156
8.6	Binary Tree Traversals	158
8.7	Threaded Binary Trees	167
8.8	Application	169
	<i>Summary</i>	175
	<i>Illustrative Problems</i>	175
	<i>Review Questions</i>	184
	<i>Programming Assignments</i>	185
9.	Graphs	186
9.1	Introduction	186
9.2	Definitions and Basic Terminologies	187
9.3	Representations of Graphs	195
9.4	Graph Traversals	199
9.5	Applications	203
	<i>Summary</i>	209
	<i>Illustrative Problems</i>	209
	<i>Review Questions</i>	214
	<i>Programming Assignments</i>	216
Part IV		
10.	Binary Search Trees and AVL Trees	218
10.1	Introduction	218
10.2	Binary Search Trees: Definition and Operations	218
10.3	AVL Trees: Definition and Operations	228
10.4	Applications	243
	<i>Summary</i>	246
	<i>Illustrative Problems</i>	247
	<i>Review Questions</i>	259
	<i>Programming Assignments</i>	260
11.	B Trees and Tries	262
11.1	Introduction	262
11.2	<i>m</i> -way search trees: Definition and Operations	262
11.3	B Trees: Definition and Operations	269
11.4	Tries: Definition and Operations	277
11.5	Applications	281
	<i>Summary</i>	284
	<i>Illustrative Problems</i>	285
	<i>Review Questions</i>	290
	<i>Programming Assignments</i>	292
12.	Red-Black Trees and Splay Trees	293
12.1	Red-Black Trees	293
12.2	Splay Trees	311

12.3 Applications	318
<i>Summary</i>	319
<i>Illustrative Problems</i>	319
<i>Review Questions</i>	329
<i>Programming Assignments</i>	330
13. Hash Tables	331
13.1 Introduction	331
13.2 Hash Table Structure	332
13.3 Hash Functions	333
13.4 Linear Open Addressing	334
13.5 Chaining	339
13.6 Applications	342
<i>Summary</i>	346
<i>Illustrative Problems</i>	347
<i>Review Questions</i>	351
<i>Programming Assignments</i>	352
14. File Organizations	353
14.1 Introduction	353
14.2 Files	354
14.3 Keys	355
14.4 Basic File Operations	356
14.5 Heap or Pile Organization	356
14.6 Sequential File Organisation	357
14.7 Indexed Sequential File Organization	358
14.8 Direct File Organization	363
<i>Illustrative Problems</i>	365
<i>Summary</i>	369
<i>Review Questions</i>	370
<i>Programming Assignments</i>	371
Part V	
15. Searching	373
15.1 Introduction	373
15.2 Linear Search	373
15.3 Transpose Sequential Search	375
15.4 Interpolation Search	376
15.5 Binary Search	378
15.6 Fibonacci Search	381
15.7 Other Search Techniques	384
<i>Summary</i>	385
<i>Illustrative Problems</i>	386
<i>Review Questions</i>	391
<i>Programming Assignments</i>	393
16. Internal Sorting	394
16.1 Introduction	394

16.2	Bubble Sort	395
16.3	Insertion Sort	396
16.4	Selection Sort	399
16.5	Merge Sort	401
16.6	Shell Sort	405
16.7	Quick Sort	410
16.8	Heap Sort	414
16.9	Radix Sort	422
	<i>Summary</i>	426
	<i>Illustrative Problems</i>	426
	<i>Review Questions</i>	433
	<i>Programming Assignments</i>	434
17.	External Sorting	435
17.1	Introduction	435
17.2	External Storage Devices	436
17.3	Sorting with Tapes: Balanced Merge	438
17.4	Sorting with Disks: Balanced Merge	441
17.5	Polyphase Merge Sort	445
17.6	Cascade Merge Sort	447
	<i>Summary</i>	449
	<i>Illustrative Problems</i>	449
	<i>Review Questions</i>	455
	<i>Programming Assignments</i>	456
	Index	457

	Contents
Part I	
4. Stacks and Queues	50
4.1 Stack Operations	40
4.2 Applications	43
Summary	43
Illustrative Problems	49
Review Questions	54
Programming Assignments	55
5. Queues	56
5.1 Introduction	56
5.2 Operations on Queues	57
5.3 Circular Queues	57
5.4 Other Types of Queues	58
5.5 Applications	71
Summary	75
Illustrative Problems	76
Review Questions	81
Programming Assignments	82
Part II	
6. Linked Lists	84
6.1 Introduction	84
6.2 Singly Linked Lists	87
6.3 Circularly Linked Lists	93
6.4 Doubly Linked Lists	98
6.5 Multiply Linked Lists	105
6.6 Applications	203
Summary	212
Illustrative Problems	213
Review Questions	219
Programming Assignments	221
7. Linked Stacks and Linked Queues	228
7.1 Introduction	228
7.2 Operations on Linked Stacks and Linked Queues	228
7.3 Dynamic Memory Management and Linked Stacks	230
7.4 Implementation of Linked Representations	232
7.5 Applications	233
Summary	237
Illustrative Problems	237
Review Questions	243
Programming Assignments	249
Part III	
8. Trees and Binary Trees	251
8.1 Introduction	251
8.2 Tree Definition and Basic Terminologies	251
8.3 Representation of Trees	251

The book is conveniently organized into five parts to favor selection of topics suiting the level of the course offered.

Each chapter lists the topics covered.

CHAPTER 6 **LINKED LISTS**



In Part I of the book we dealt with arrays, stacks and queues which are linear sequential data structures (of these, stacks and queues have a linked representation as well, which will be discussed in Chapter 7). In this chapter we detail linear data structures having a linked representation. We first list the elements of the sequential data structures before introducing the need for a linked representation. Next, the linked data structures of singly linked list, circularly linked list, doubly linked list and multiply linked list are elaborately presented. Finally, two problems, viz., Polynomial addition and Sparse matrix representation, demonstrating the application of linked lists are discussed.

6.1 Introduction
6.2 Singly Linked Lists
6.3 Circularly Linked Lists
6.4 Doubly Linked Lists
6.5 Multiply Linked Lists
6.6 Applications

Introduction

6.1

Drawbacks of sequential data structures

Arrays are fundamental sequential data structures. Even stacks and queues rely on arrays for their representation and implementation. However, arrays or sequential data structures in general, suffer from the following drawbacks:

- (i) inefficient implementation of insertion and deletion operations and
- (ii) inefficient use of storage memory.

Let us consider an array $A[1 : 20]$. This means a contiguous set of twenty memory locations have been made available to accommodate the data elements of A . As shown in Fig. 6.1(a), let us suppose the array is partially full. Now, to insert a new element 109 in the position indicated, it is not possible to do so without affecting the neighbouring data elements from their positions. Methods such as making use of a temporary array (B) to hold the data elements of A which follow 108, before copying B into A , call for extensive data movement which is computationally expensive. Again, attempting to delete 217 from A calls for the use of a temporary array B to hold the elements with 217 excluded, before copying B to A . (Fig. 6.1)

Chapter-end summary for use as quick reference.

Summary

- Hash tables are ideal data structures for dictionaries. They favor efficient storage and retrieval of data lists which are linear in nature.
- A hash function is a mathematical function which maps keys to positions in the hash tables known as buckets. The process of mapping is called hashing. Keys which map to the same bucket are called as synonyms. In such a case a collision is said to have occurred. A bucket may be divided into slots to accommodate synonyms. When a bucket is full and a synonym is unable to find space in the bucket then an overflow is said to have occurred.
- The characteristics of a hash function are that it must be easy to compute and at the same time minimize collisions. Folding, truncation and modulus arithmetic are some of the commonly used hash functions.
- A hash table could be implemented using a sequential data structure such as arrays. In such a case, the method of handling overflows where the closest slot that is vacant is utilized to accommodate the synonym key is called linear open addressing or linear probing. However, in course of time, linear probing can lead to the problem of clustering thereby deteriorating the performance of the hash table to a mere sequential search!
- The other alternative methods of handling overflows are rehashing, quadratic probing and random probing.

Illustrative Problems

Problem 15.1 For the list `CDNSHLD=1 ABBB, BBBB, CCDC, CCDD, DDDC, DDDD, EEEC, EEEE, FFFF, GGGG, HHHH, IIII`, trace through a sequential search for the value `CCDD`.

External Sorting

448

Summary

- External sorting deals with sorting of files or lists that are too large to be accommodated in the internal memory of the computer and hence need to be stored in external storage devices such as disks or drums.
- The principle behind external sorting is to first make use of any efficient internal sorting techniques to generate runs. These runs are then merged in passes to obtain a single run at which stage the file is deemed sorted. The merge patterns called for by the strategies, are influenced by external storage medium on which the runs reside, viz., disks or tapes.
- Magnetic tapes are sequential devices built on the principle of audio tape devices. Data is stored in blocks occurring sequentially. Magnetic disks are random access storage devices. Data stored in a disk is addressed by its cylinder, track and sector numbers.
- Balanced merge sort is a technique that can be adopted on files residing on both disks and tapes. In its general form, a k-way merging could be undertaken during the runs. For the efficient management of merging runs, buffer handling and selection tree mechanisms are employed.
- Balanced k-way merge sort on tapes calls for the use of 2k tapes for an efficient management of runs. Polyphase merge sort is a clever strategy that makes use of only $\lceil \frac{k+1}{2} \rceil$ tapes to perform the k-way merge. The distribution of runs on the tapes follows a Fibonacci number sequence.
- Cascade merge sort is yet another smart strategy which unlike polyphase merge sort does not employ a uniform merge pattern. Each pass makes use of a 'cascading' sequence of merge patterns.

Illustrative Problems

Problem 17.1 The specification for a typical disk storage system is shown in Table I 17.1. An employee file consisting of 100000 records is stored on the disk. The employee record structure and the size of the fields in bytes (shown in brackets) are given below:

Employee number	Employee name	Designation	Address	Basic pay	Allowances	Deductions	Total salary
(6)	(20)	(10)	(30)	(6)	(20)	(20)	(6)

- (a) What is the storage space (in terms of bytes) needed to store the employee file in the disk?
 (b) What is the storage space (in term of cylinders) needed to store the employee file in the disk?

Solution:

- (a) The size of the employee record = 118 bytes
 Number of employee records that can be held in a sector = $512/118 = 4$ records
 Number of sectors needed to hold the whole employee file = $100000/4 = 25000$ sectors

Extensive Illustrative Problems throughout.

Review Questions include objective-type, short-answer and long-answer type questions.



Review Questions

1. A minimal superkey is in fact a _____
 - (a) secondary key
 - (b) primary key
 - (c) non-key
 - (d) none of these
2. State whether true or false:
 - (a) A cluster index is a sparse index
 - (b) A secondary key field with distinct values yields a dense index
 - (c) (i) true (ii) false
 - (d) (i) false (ii) true
 - (e) (i) false (ii) false
 - (f) (i) false (ii) true
3. An index consisting of variable length entries where each index entry would be of the form $(k, R_1, R_2, \dots, R_l)$ where R_i 's are block addresses of the various records holding the same value for the secondary key K can occur only in
 - (a) primary indexing
 - (b) secondary indexing
 - (c) cluster indexing
 - (d) multilevel indexing

ADT for Queues

```

Data objects
A finite set of elements of the same type
Operations
• Create an empty queue and initialise front and rear variables of the queue
  CREATE ( queue, front, rear)
• Check if queue queue is empty
  IS_QUEUE_EMPTY (queue) : Boolean Function
• Check if queue queue is full
  IS_QUEUE_FULL (queue) : Boolean Function
• Insert item into queue queue
  ENQUEUE (queue, item)
• Delete element from queue queue and output the element deleted to item
  DEQUEUE (queue, item)
  
```

The ADTs for selective data structures are separately presented for convenience of reference.

Programming Assignments are given at the end of each chapter.



Programming Assignment

1. Write a program to input a binary tree implemented in a linked representation. Execute Algorithms 8.1-8.3 to perform inorder, postorder and preorder traversals of the binary tree.
2. Implement Algorithm 8.4 to convert an infix expression into its postfix form.
3. Write a recursive procedure to count the number of nodes in a binary tree.
4. Implement a threaded binary tree. Write procedures to insert a node NEW to the left of node NODE when
 - (a) the left subtree of NODE is empty, and
 - (b) the left subtree of NODE is non-empty.

Interval Scheduling

413

```

Algorithm 16.7: Procedure for Partition.
Procedure PARTITION(first, last, loc)
    /* left is the list to be partitioned, loc is the
       position where the pivot element finally occurs down*/
left = first
right = last-1
pivot_val = list[first] /* set the pivot element to the first
                           element in list left */
while left < right do
    DEPART:
        left = left+1 /* point element above left to right*/
        SELL (left) < pivot_val
    REPART:
        right = right-1 /* pivot element moves right to left*/
        SELL (right) > pivot_val
    IF left < right THEN REPART(left, right) /*remove from each
       other*/
    end
loc = right
SELL (loc) < pivot_val /* move last element back above + exchange
                           pivot element (different with left)*/
end PARTITION.

```

Example 16.13 Let us quick sort the list $L = [9, 1, 26, 15, 76, 34, 15]$. The various phases of the sorting process are shown in Fig. 16.8. When the partitioned sublists contain only one element then no sorting is done. Also in phase 4 of Fig. 16.8 observe how the pivot element 34 exchanges with itself. The final sorted list is $[1, 5, 15, 15, 26, 34, 76]$.

```

Algorithm 16.8: Procedure for Quick Sort.
Procedure QUICK_SORT(first, last)
    /* left is the successive list of elements to be
       quick sorted. The call to the procedure to sort the
       list list would be QUICK_SORT(1, n) */
IF first < last then
    1. PARTITION(first, last, loc) /* partitions the list list into two
       sublists at loc */
    2. QUICK_SORT(first, loc-1) /* quick sort the smaller
       sublist list[1..loc-1] */
    3. QUICK_SORT(loc+1, last) /* quick sort the larger
       sublist list[loc..last] */
end QUICK_SORT.

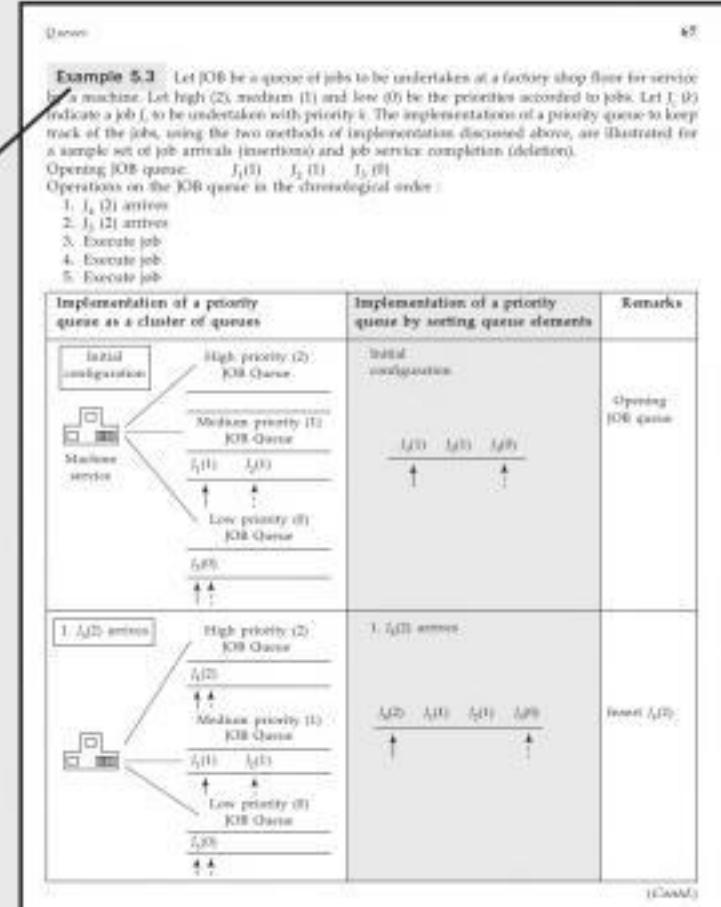
```

Stability and performance analysis:

Quick sort is not a stable sort. During the partitioning process keys which are equal are subject to exchange and hence undergo changes in their relative orders of occurrence in the sorted list.

Extensive examples are given to illustrate theoretical concepts.

Pseudo-code algorithms are given for better comprehension



The Online Learning Centre at www.mhhe.com/pai/dsa contains C programs for algorithms present in the text, Sample Questions with Solutions and Web Links.



INTRODUCTION

While looking around and marveling at the technological advancements of this world—both within and without, one cannot but perceive the intense and intrinsic association of the disciplines of Science and Engineering and their allied and hybrid counterparts, with the ubiquitous machines called *computers*. In fact it is difficult to spot a discipline that has distanced itself from the discipline of computer science. To quote a few, be it a medical surgery or diagnosis performed by robots or doctors on patients half way across the globe, or the launching of space crafts and satellites into outer space, or forecasting tornadoes and cyclones, or the more mundane needs of online reservations of tickets or billing at the food store, or control of washing machines etc. one cannot but deem computers to be *omnipresent, omnipotent, why even omniscient!* (Refer Fig. 1.1.)

- 1.1 History of Algorithms
- 1.2 Definition, Structure and Properties of Algorithms
- 1.3 Development of an Algorithm
- 1.4 Data structures and Algorithms
- 1.5 Data structure—Definition and Classification
- 1.6 Organization of the Book

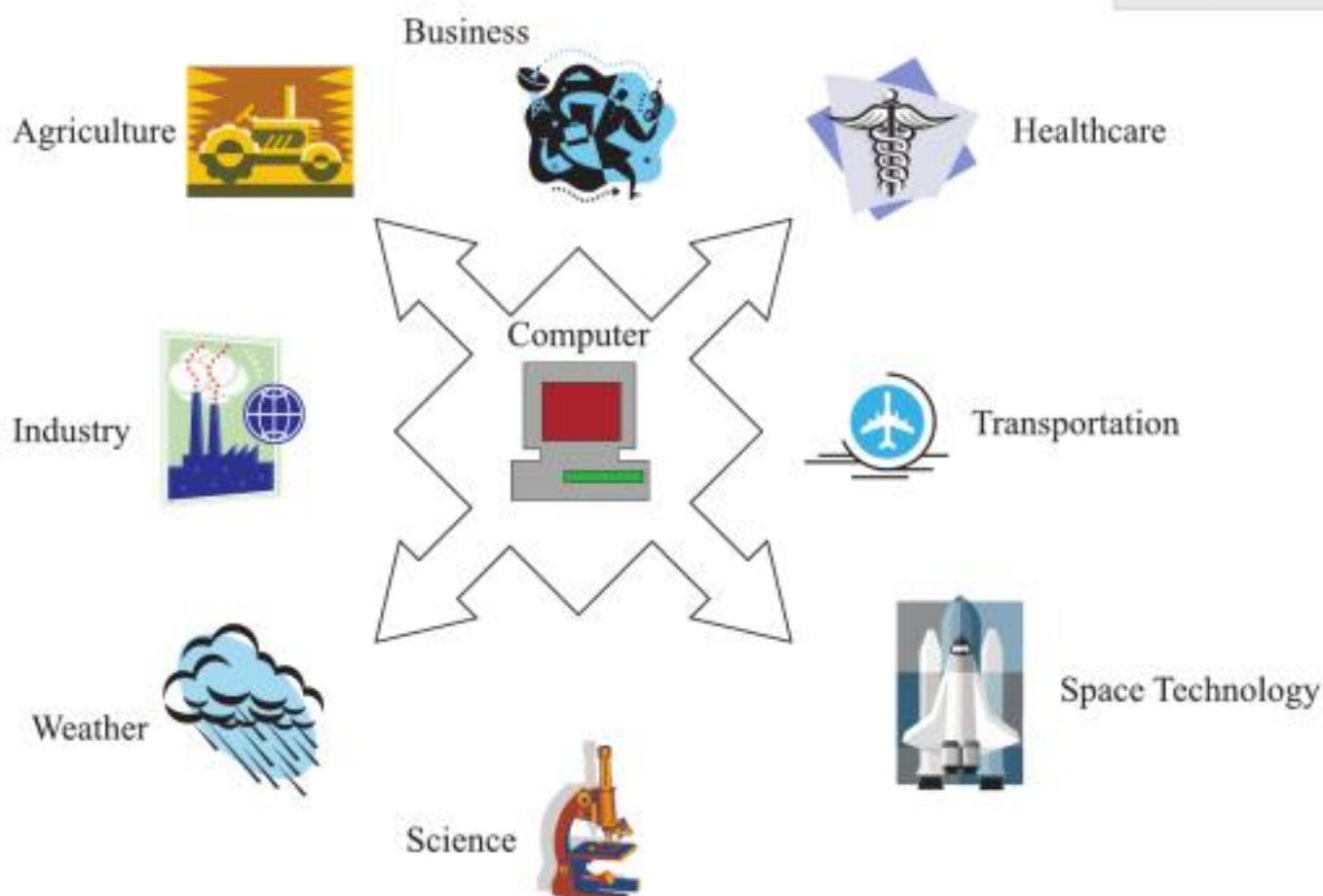


Fig. 1.1 Omnipresence of computers

In short, any discipline that calls for **problem-solving** using computers, looks up to the discipline of computer science for efficient and effective methods and techniques of solutions to the problems in their respective fields. From the point of view of problem solving, the discipline of computer science could be naively categorized into the following four sub areas notwithstanding the overlaps and grey areas amongst themselves:

- *Machines* What machines are appropriate or available for the solution of a problem? What is the machine configuration – its processing power, memory capacity etc., that would be required for the efficient execution of the solution?
- *Languages* What is the language or software with which the solution of the problem needs to be coded? What are the software constraints that would hamper the efficient implementation of the solution?
- *Foundations* What is the model of a problem and its solution? What methods need to be employed for the efficient design and implementation of the solution? What is its performance measure?
- *Technologies* What are the technologies that need to be incorporated for the solution of the problem? For example, does the solution call for a web based implementation or needs activation from mobile devices or calls for hand shaking broadcasting devices or merely needs to interact with high end or low end peripheral devices?

Figure 1.2 illustrates the categorization of the discipline of computer science from the point of view of problem solving.

One of the core fields that belongs to the foundations of computer science deals with the design, analysis and implementation of **algorithms** for the efficient solution of the problems concerned. An algorithm may be loosely defined as a **process**, or **procedure** or **method** or **recipe**. It is a specific set of rules to obtain a definite output from specific inputs provided to the problem.

The subject of **data structures** is intrinsically connected with the design and implementation of efficient algorithms. **Data structures deals with the study of methods, techniques and tools to organize or structure data.**

Next, the history, definition, classification, structure and properties of algorithms are discussed.

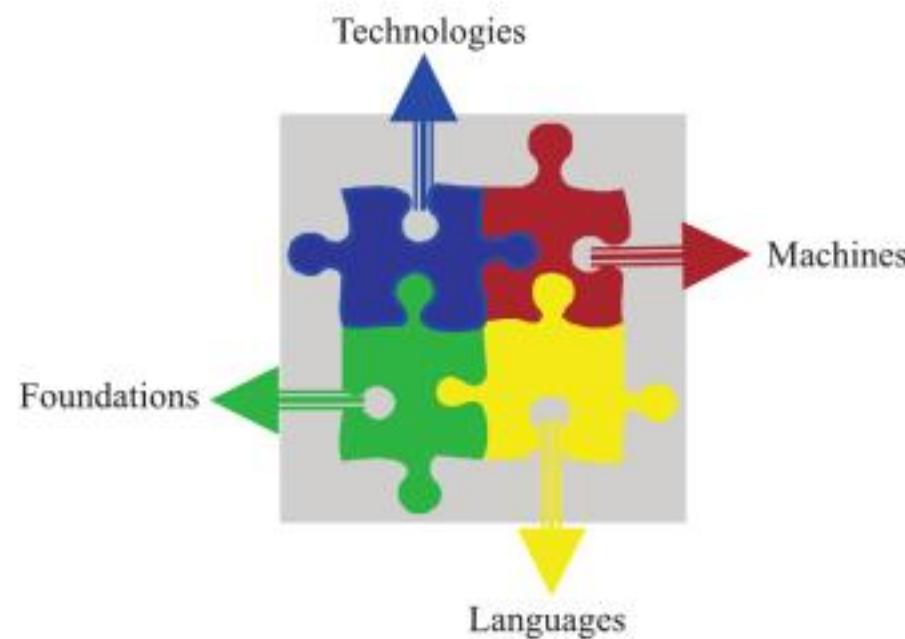


Fig. 1.2 Discipline of computer science from the point of view of problem solving

History of Algorithms

1.1

The word **algorithm** originates from the Arabic word **algorism** which is linked to the name of the Arabic mathematician Abu Jafar Mohammed Ibn Musa Al Khwarizmi (825 A.D.). Al Khwarizmi

is considered to be the first algorithm designer for adding numbers represented in the Hindu numeral system. The algorithm designed by him and followed till today, calls for summing up the digits occurring at specific positions and the previous carry digit, repetitively moving from the least significant digit to the most significant digit until the digits have been exhausted.

Example 1.1 Demonstration of Al Khwarizmi's algorithm for the addition of 987 and 76:

$$\begin{array}{r} 987 + \\ 76 \\ \hline \text{(Carry 1) } 3 \end{array}
 \quad \Rightarrow \quad
 \begin{array}{r} 987 + \\ 76 + \\ \text{Carry 1} \\ \hline \text{(Carry 1) } 63 \end{array}
 \quad \Rightarrow \quad
 \begin{array}{r} 987 + \\ 76 + \\ \text{Carry 1} \\ \hline 1063 \end{array}$$

Definition, Structure and Properties of Algorithms

1.2

Definition An algorithm may be defined as a finite sequence of instructions each of which has a clear meaning and can be performed with a finite amount of effort in a finite length of time.

Structure and properties

An algorithm has the following structure:

- | | |
|----------------------|----------------------|
| (i) Input step | (iv) Repetitive step |
| (ii) Assignment step | (v) Output step |
| (iii) Decision step | |

Example 1.2 Consider the demonstration of Al Khwarizmi's algorithm shown on the addition of the numbers 987 and 76 in Example 1.1. In this, the input step considers the two operands 987 and 76 for addition. The assignment step sets the pair of digits from the two numbers and the previous carry digit if it exists, for addition. The decision step decides at each step whether the added digits yield a value that is greater than 10 and if so, to generate the appropriate carry digit. The repetitive step repeats the process for every pair of digits beginning from the least significant digit onwards. The output step releases the output which is 1063.

An algorithm is endowed with the following properties:

Finiteness	an algorithm must terminate after a finite number of steps.
Definiteness	the steps of the algorithm must be precisely defined or unambiguously specified.
Generality	an algorithm must be generic enough to solve all problems of a particular class.
Effectiveness	the operations of the algorithm must be basic enough to be put down on pencil and paper. They should not be too complex to warrant writing another algorithm for the operation!
Input-Output	the algorithm must have certain initial and precise inputs, and outputs that may be generated both at its intermediate and final steps.

An algorithm does not enforce a language or mode for its expression but only demands adherence to its properties. Thus one could even write an algorithm in one's own expressive way to make a cup of hot coffee! However, there is this observation that a cooking recipe that calls for

instructions such as "add a pinch of salt and pepper", 'fry until it turns golden brown' are "anti-algorithmic" for the reason that terms such as 'a pinch', 'golden brown' are subject to ambiguity and hence violate the property of definiteness!

An algorithm may be represented using pictorial representations such as flow charts. An algorithm encoded in a programming language for implementation on a computer is called a *program*. However, there exists a school of thought which distinguishes between a program and an algorithm. The claim put forward by them is that programs need not exhibit the property of finiteness which algorithms insist upon and quote an operating systems program as a counter example. An operating system is supposed to be an 'infinite' program which terminates only when the system crashes! At all other times other than its execution, it is said to be in the 'wait' mode!

Development of an Algorithm

1.3

The steps involved in the development of an algorithm are as follows:

- | | |
|----------------------------|-------------------------|
| (i) Problem statement | (v) Implementation |
| (ii) Model formulation | (vi) Algorithm analysis |
| (iii) Algorithm design | (vii) Program testing |
| (iv) Algorithm correctness | (viii) Documentation |

Once a clear statement of the problem is done, the model for the solution of the problem is to be formulated. The next step is to design the algorithm based on the solution model that is formulated. It is here that one sees the role of data structures. The right choice of the data structure needs to be made at the design stage itself since data structures influence the efficiency of the algorithm. Once the correctness of the algorithm is checked and the algorithm implemented, the most important step of measuring the performance of the algorithm is done. This is what is termed as *algorithm analysis*. It can be seen how the use of appropriate data structures results in a better performance of the algorithm. Finally the program is tested and the development ends with proper documentation.

Data Structures and Algorithms

1.4

As was detailed in the previous section, the design of an *efficient* algorithm for the solution of the problem calls for the *inclusion of appropriate data structures*. A clear, unambiguous set of instructions following the properties of the algorithm alone does not contribute to the efficiency of the solution. It is essential that the data on which the problems need to work on are appropriately *structured* to suit the needs of the problem, thereby contributing to the efficiency of the solution.

For example, let us consider the problem of searching for a telephone number of a person, in the telephone directory. It is well known that searching for the telephone number in the directory is an easy task since the data is sorted according to the alphabetical order of the subscribers' names. All that the search calls for, is to turn over the pages until one reaches the page that is approximately closest to the subscriber's name and undertake a sequential search in the relevant page. Now, what if the telephone directory were to have its data arranged according to the order in which the subscriptions for telephones were received. What a mess would it be! One may need

to go through the entire directory—name after name, page after page in a sequential fashion until the name and the corresponding telephone number are retrieved!

This is a classic example to illustrate the significant role played by data structures in the efficiency of algorithms. The problem was retrieval of a telephone number. The algorithm was a simple search for the name in the directory and thereby retrieve the corresponding telephone number. In the first case since the data was appropriately structured (sorted according to alphabetical order), the search algorithm undertaken turned out to be efficient. On the other hand, in the second case, when the data was unstructured, the search algorithm turned out to be crude and hence inefficient.

For the design of efficient programs and for the solution of problems, it is essential that algorithm design goes hand in hand with appropriate data structures. (Refer Fig. 1.3.)

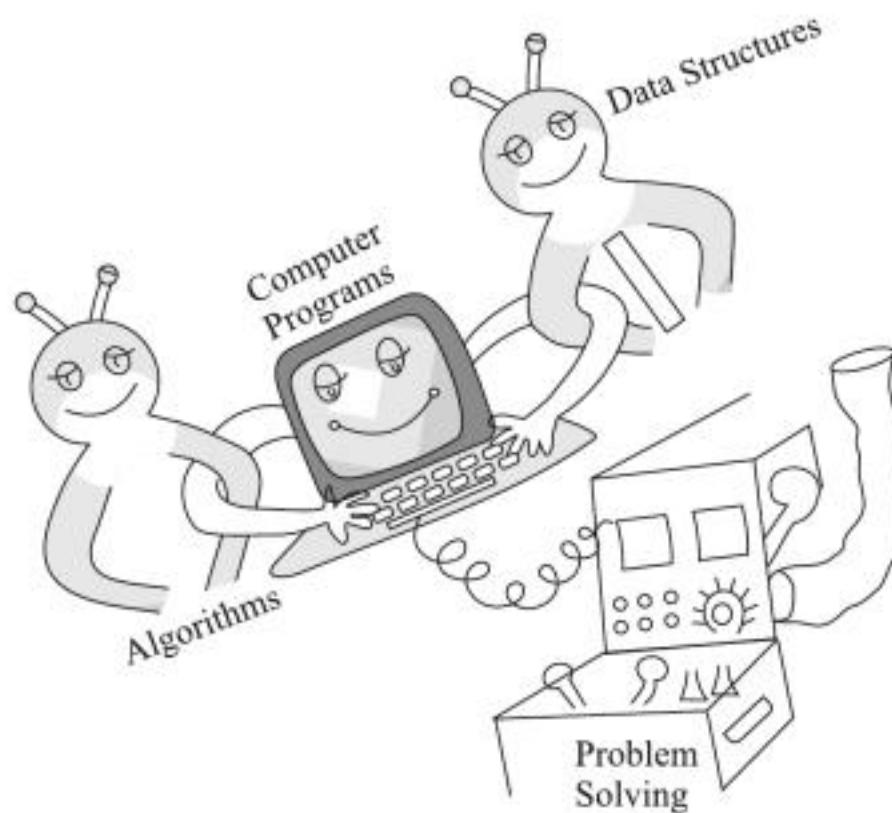


Fig. 1.3 Algorithms and Data structures for efficient problem solving using computers

Data Structure—Definition and Classification

1.5

Abstract data types

A **data type** refers to the type of values that variables in a programming language hold. Thus the data types of integer, real, character, Boolean which are inherently provided in programming languages are referred to as **primitive data types**.

A list of elements is called as a **data object**. For example, we could have a list of integers or list of alphabetical strings as data objects.

The data objects which comprise the data structure, and their fundamental operations are known as **Abstract Data Type (ADT)**. In other words, an ADT is defined as a set of data objects D defined over a domain L and supporting a list of operations O .

Example 1.3 Consider an ADT for the data structure of positive integers called POSITIVE_INTEGER defined over a domain of integers Z^+ , supporting the operations of addition (ADD), subtraction(MINUS) and check if positive (CHECK_POSITIVE). The ADT is defined as follows:

$$L = Z^+, D = \{x \mid x \in L\}, Q = \{\text{ADD}, \text{MINUS}, \text{CHECK_POSITIVE}\}$$

A descriptive and clear presentation of the ADT is as follows:

Data objects

Set of all positive integers D

$$D = \{x \mid x \in L\}, L = Z^+$$

Operations

- Addition of positive integers INT1 and INT2 into RESULT
ADD (INT1, INT2, RESULT)
- Subtraction of positive integers INT1 and INT2 into RESULT
SUBTRACT (INT1, INT2, RESULT)
- Check if a number INT1 is a positive integer
CHECK_POSITIVE(INT1) (Boolean function)

An ADT promotes ***data abstraction*** and focuses on *what* a data structure does rather than *how* it does. It is easier to comprehend a data structure by means of its ADT since it helps a designer to plan on the implementation of the data objects and its supportive operations in any programming language belonging to any paradigm such as procedural or object oriented or functional etc. Quite often it may be essential that one data structure calls for other data structures for its implementation. For example, the implementation of stack and queue data structures calls for their implementation using either arrays or lists.

While deciding on the ADT of a data structure, a designer may decide on the set of operations O that are to be provided, based on the application and accessibility options provided to various users making use of the ADT implementation.

The ADTs for various data structures discussed in the book are presented as box items in the respective chapters.

Classification

Figure 1.4 illustrates the classification of data structures. The data structures are broadly classified as ***linear data structures*** and ***non-linear data structures***. Linear data structures are uni-dimensional in structure and represent linear lists. These are further classified as ***sequential*** and ***linked representations***. On the other hand, non-linear data structures are two-dimensional representations of data lists. The individual data structures listed under each class have been shown in Fig. 1.4.

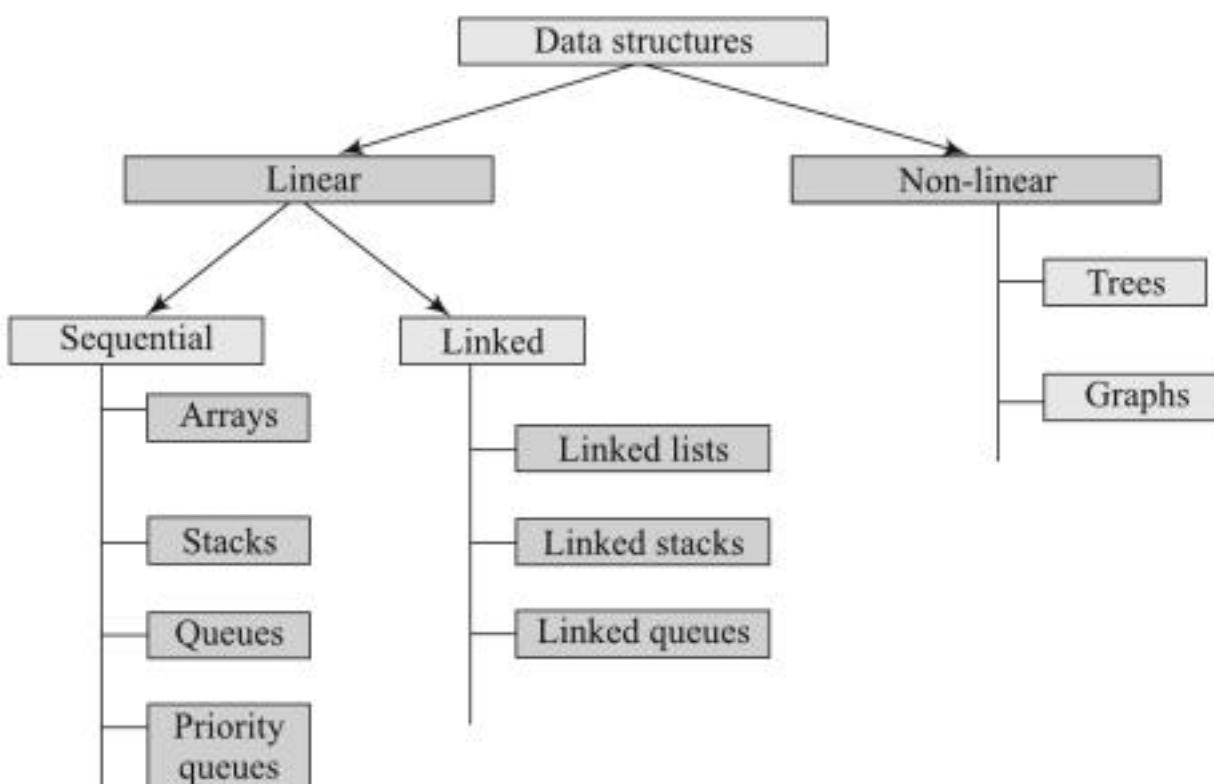


Fig. 1.4 Classification of data structures

Organization of the book

The book is divided into five parts. Chapter 1 deals with an introduction to the subject of data structures and algorithms. Chapter 2 introduces analysis of algorithms.

Part I discusses linear data structures and includes three chapters pertaining to *sequential data structures*. Chapters 3, 4 and 5 discuss the data structures of arrays, stacks and queues.

Part II also discusses linear data structures and incorporates two chapters on *linked data structures*. Chapter 6 discusses linked lists in its entirety and Chapter 7 details linked stacks and queues.

Part III discusses the *non-linear data structures* of trees and graphs. Chapter 8 discusses trees and binary trees and Chapter 9 details on graphs.

Part IV discusses some of the *advanced data structures*. Chapter 10 discusses binary search trees and AVL trees. Chapter 11 details B trees and tries. Chapter 12 deals with red-black trees and splay trees. Chapter 13 discusses hash tables and Chapter 14 describes methods of file organizations.

The ADTs for some of the fundamental data structures discussed in PARTS I, II, III and IV have been provided towards the end of the appropriate chapters.

Part V deals with *searching and sorting techniques*. Chapter 15 discusses searching techniques, Chapter 16 details internal sorting methods and Chapter 17 describes external sorting methods.



Summary

- Any discipline in Science and Engineering that calls for solving problems using computers, looks up to the discipline of Computer Science for its efficient solution.
- From the point of view of solving problems, computer science can be naively categorized into the four areas of machines, languages, foundations and technologies.
- The subjects of Algorithms and Data structures fall under the category of foundations. The design formulation of algorithms for the solution of problems and the inclusion of appropriate data structures for their efficient implementation must progress hand in hand.
- An Abstract Data Type (ADT) describes the data objects which constitute the data structure and the fundamental operations supported on them.
- Data structures are classified as linear and non linear data structures. Linear data structures are further classified as sequential and linked data structures. While arrays, stacks and queues are examples of sequential data structures, linked lists, linked stacks and queues are examples of linked data structures.
- The non-linear data structures include trees and graphs
- The tree data structure includes variants such as binary search trees, AVL trees, B trees, Tries, Red Black trees and Splay trees.



ANALYSIS OF ALGORITHMS

2

In the previous chapter we introduced the discipline of computer science from the perspective of problem solving. It was detailed how problem solving using computers calls not just for good algorithm design but also for the appropriate use of data structures to render them efficient. This chapter discusses methods and techniques to analyze the efficiency of algorithms.

Efficiency of Algorithms

2.1

When there is a problem to be solved it is probable that several algorithms crop up for its solution and therefore one is at a loss to know which one is the best. This raises the question of how one could decide on which among the algorithms is preferable and which among them is the best.

The performance of algorithms can be measured on the scales of *time* and *space*. The former would mean looking for the fastest algorithm for the problem or that which performs its task in the minimum possible time. In this case the performance measure is termed *time complexity*. The time complexity of an algorithm or a program is a function of the running time of the algorithm or program.

In the case of the latter, it would mean looking for an algorithm that consumes or needs limited memory space for its execution. The performance measure in such a case is termed *space complexity*. The space complexity of an algorithm or a program is a function of the space needed by the algorithm or program to run to completion. However, in this book our discussions would emphasize mostly on time complexities of the algorithms presented.

The time complexity of an algorithm can be computed either by an empirical or theoretical approach.

The *empirical* or *posteriori testing* approach calls for implementing the complete algorithms and executing them on a computer for various instances of the problem. The time taken by the execution of the programs for various instances of the problem are noted and compared. That algorithm whose implementation yields the least time, is considered as the best among the candidate algorithmic solutions.

2.1 *Efficiency of Algorithms*

2.2 *Apriori Analysis*

2.3 *Asymptotic Notations*

2.4 *Time Complexity of an Algorithm using O Notation*

2.5 *Polynomial vs Exponential Algorithms*

2.6 *Average, Best and Worst Case Complexities*

2.7 *Analyzing Recursive Programs*

The *theoretical* or *apriori* approach calls for mathematically determining the resources such as time and space needed by the algorithm, as a function of a parameter related to the instances of the problem considered. A parameter that is often used is the size of the input instances. For example, for the problem of searching for a name in the telephone directory, an apriori approach could determine the efficiency of the algorithm used, in terms of the size of the telephone directory (i.e.) the number of subscribers listed in the directory. There exist algorithms for various classes of problems which make use of the number of basic operations such as additions or multiplications or element comparisons, as a parameter to determine their efficiency.

The disadvantage of posteriori testing is that it is dependent on various other factors such as the machine on which the program is executed, the programming language with which it is implemented and why, even on the skills of the programmer who writes the program code! On the other hand, the advantage of apriori analysis is that it is entirely machine, language and program independent.

The efficiency of a newly discovered algorithm over that of its predecessors can be better assessed only when they are tested over large input instance sizes. For smaller to moderate input instance sizes it is highly likely that their performances may break even. In the case of posteriori testing, practical considerations may permit testing the efficiency of the algorithm only on input instances of moderate sizes. On the other hand, apriori analysis permits study of the efficiency of algorithms on any input instance of any size.

Apriori Analysis

2.2

Let us consider a program statement, for example, $x = x + 2$ in a sequential programming environment. We do not consider any parallelism in the environment. Apriori estimation is interested in the following for the computation of efficiency:

- (i) the number of times the statement is executed in the program, known as the *frequency count* of the statement, and
- (ii) the time taken for a single execution of the statement.

To consider the second factor would render the estimation machine dependent since the time taken for the execution of the statement is determined by the machine instruction set, the machine configuration, and so on. Hence apriori analysis considers only the first factor and computes the efficiency of the program as a function of the *total frequency count* of the statements comprising the program. The estimation of efficiency is restricted to the computation of the total frequency count of the program.

Let us estimate the frequency count of the statement $x = x + 2$ occurring in the following three program segments (A , B , C):

Program segment A

```
...  
x = x + 2;  
...
```

Program segment B

```
...  
for k = 1 to n do  
x = x + 2;  
end  
...
```

Program segment C

```
...  
for j = 1 to n do  
for x = 1 to n do  
x = x + 2;  
end  
end
```

The frequency count of the statement in the program segment A is 1. In the program segment B , the frequency count of the statement is n , since the **for** loop in which the statement is embedded executes n ($n \geq 1$) times. In the program segment C , the statement is executed n^2 ($n \geq 1$) times since the statement is embedded in a nested **for** loop, executing n times each.

In apriori analysis, the frequency count f_i of each statement i of the program is computed and summed up to obtain the total frequency count $T = \sum_i f_i$.

The computation of the total frequency count of the program segments A, B, and C are shown in Tables 2.1, 2.2 and 2.3. It is well known that the opening statement of a **for** loop such as **for** $i = \text{low_index}$ **to** up_index executes $((\text{up_index} - \text{low_index} + 1) + 1$ times and the statements within the loop are executed $(\text{up_index} - \text{low_index}) + 1$ times. In the

Table 2.1 Total frequency count of program segment A

Program statements	Frequency count
...	
$x = x + 2;$	1
...	
Total frequency count	1

Table 2.2 Total frequency count of program segment B

Program statements	Frequency count
...	
for $k = 1$ to n do	$(n + 1)$
$x = x + 2;$	n
end	n
...	
Total frequency count	$3n + 1$

Table 2.3 Total frequency count of program segment C

Program statements	Frequency count
...	
for $j = 1$ to n do	$(n + 1)$
for $k = 1$ to n do	$\sum_{j=1}^n (n + 1) = (n + 1)n$
$x = x + 2;$	n^2
end	$\sum_{j=1}^n n = n^2$
end	n
...	
Total frequency count	$3n^2 + 3n + 1$

case of nested **for** loops, it is easier to compute the frequency counts of the embedded statements making judicious use of the following fundamental mathematical formulae:

$$\sum_{i=1}^n 1 = n \quad \sum_{i=1}^n i = \frac{n(n+1)}{2} \quad \sum_{i=1}^n i^2 = \frac{n(n+1)(2n+1)}{6}$$

Observe in Table 2.3 how the frequency count of the statement **for** $k = 1$ **to** n **do** is computed as

$$\sum_{j=1}^n (n-1+1) + 1 = \sum_{j=1}^n (n+1) = (n+1)n$$

The total frequency counts of the program segments A , B and C given by 1, $(3n + 1)$ and $3n^2 + 3n + 1$ respectively, are expressed as $O(1)$, $O(n)$ and $O(n^2)$ respectively. These notations mean that the orders of the magnitude of the total frequency counts are proportional to 1, n and n^2 respectively. The notation O has a mathematical definition as discussed in Sec. 2.3. These are referred to as the time complexities of the program segments since they are indicative of the running times of the program segments. In a similar manner, one could also discuss about the space complexities of a program which is the amount of memory they require for their execution and its completion. The space complexities can also be expressed in terms of mathematical notations.

Asymptotic Notations

2.3

Apriori analysis employs the following notations to express the time complexity of algorithms. These are termed *asymptotic notations* since they are meaningful approximations of functions that represent the time or space complexity of a program.

Definition 2.1: $f(n) = O(g(n))$ (read as f of n is “big oh” of g of n), if there exists a positive integer n_0 and a positive number C such that $|f(n)| \leq C|g(n)|$, for all $n \geq n_0$.

Example

$f(n)$	$g(n)$	
$16n^3 + 78n^2 + 12n$	n^3	$f(n) = O(n^3)$
$34n - 90$	n	$f(n) = O(n)$
56	1	$f(n) = O(1)$

Here $g(n)$ is the upper bound of the function $f(n)$.

Definition 2.2: $f(n) = \Omega(g(n))$ (read as f of n is omega of g of n), if there exists a positive integer n_0 and a positive number C such that $|f(n)| \geq C|g(n)|$, for all $n \geq n_0$.

Example

$f(n)$	$g(n)$	
$16n^3 + 8n^2 + 2$	n^3	$f(n) = \Omega(n^3)$
$24n + 9$	n	$f(n) = \Omega(n)$

Here $g(n)$ is the lower bound of the function $f(n)$.

Definition 2.3: $f(n) = \Theta(g(n))$ (read as f on n is theta of g of n) if there exist two positive constants c_1 and c_2 , and a positive integer n_0 such that $c_1|g(n)| \leq |f(n)| \leq c_2|g(n)|$ for all $n \geq n_0$.

Example

$f(n)$	$g(n)$	
$28n + 9$	n	$f(n) = \Theta(n)$ since $f(n) > 28n$
$16n^2 + 30n - 90$	n^2	$f(n) = \Theta(n^2)$
$7.2^n + 30n$	2^n	$f(n) = \Theta(2^n)$

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

Definition 2.4: $f(n) = o(g(n))$ (read as f of n is “little oh” of g of n) if $f(n) = O(g(n))$ and $f(n) \neq \Omega(g(n))$.

Example

$f(n)$	$g(n)$	
$18n + 9$	n^2	$f(n) = o(n^2)$ since $f(n) = O(n^2)$ and $f(n) \neq \Omega(n^2)$ however, $f(n) \neq O(n)$.

Time Complexity of an Algorithm Using O Notation**2.4**

O notation is widely used to compute the time complexity of algorithms. It can be gathered from its definition (Definition 2.1) that if $f(n) = O(g(n))$ then $g(n)$ acts as an upper bound for the function $f(n)$. $f(n)$ represents the computing time of the algorithm. When we say the time complexity of the algorithm is $O(g(n))$, we mean that its execution takes a time that is no more than constant times $g(n)$. Here n is a parameter that characterizes the input and/or output instances of the algorithm.

Algorithms reporting $O(1)$ time complexity indicate *constant running time*. The time complexities of $O(n)$, $O(n^2)$ and $O(n^3)$ are called *linear*, *quadratic* and *cubic* time complexities respectively. $O(\log n)$ time complexity is referred to as *logarithmic*. In general, time complexities of the type $O(n^k)$ are called *polynomial time complexities*. In fact it can be shown that a polynomial $A(n) = a_m n^m + a_{m-1} n^{m-1} + \dots + a_1 n + a_0 = O(n^m)$ (see Illustrative Problem 2.2). Time complexities such as $O(2^n)$, $O(3^n)$, in general $O(k^n)$ are called as *exponential time complexities*.

Algorithms which report $O(\log n)$ time complexity are faster for sufficiently large n , than if they had reported $O(n)$. Similarly $O(n \log n)$ is better than $O(n^2)$, but not as good as $O(n)$. Some of the commonly occurring time complexities in their ascending orders of magnitude are listed below:

$$O(1) \leq O(\log n) \leq O(n) \leq O(n \log n) \leq O(n^2) \leq O(n^3) \leq O(2^n)$$

Polynomial Vs Exponential Algorithms**2.5**

If n is the size of the input instance, then the number of operations for polynomial algorithms are of the form $P(n)$ where P is a polynomial. In terms of O notation, polynomial algorithms have time complexities of the form $O(n^k)$, where k is a constant.

In contrast, in the exponential algorithms the number of operations are of the form k^n . In terms of O notation, exponential algorithms have time complexities of the form $O(k^n)$, where k is a constant.

It is clear from the inequalities listed above that polynomial algorithms are a lot more efficient than exponential algorithms. From Table 2.4 it is seen that exponential algorithms can quickly get beyond the capacity of any sophisticated computer due to their rapid growth rate (Refer Fig. 2.1). Here, it is assumed that the computer takes 1 microsecond per operation. While the time complexity functions of n^2 , n^3 can be executed in a reasonable time, one can never hope to finish the execution of exponential algorithms even if the fastest computer were to be employed. Thus if one were to find an algorithm for a problem that reduces from exponential to polynomial time then it is indeed a great accomplishment!

Table 2.4 Comparison of polynomial and exponential algorithms

Size	10	20	50
Time complexity function			
n^2	10^{-4} sec	4×10^{-4} sec	25×10^{-4} sec
n^3	10^{-3} sec	8×10^{-3} sec	125×10^{-3} sec
2^n	10^{-3} sec	1 sec	35 years
$3n$	6×10^{-2} sec	58 mins	2×10^3 centuries

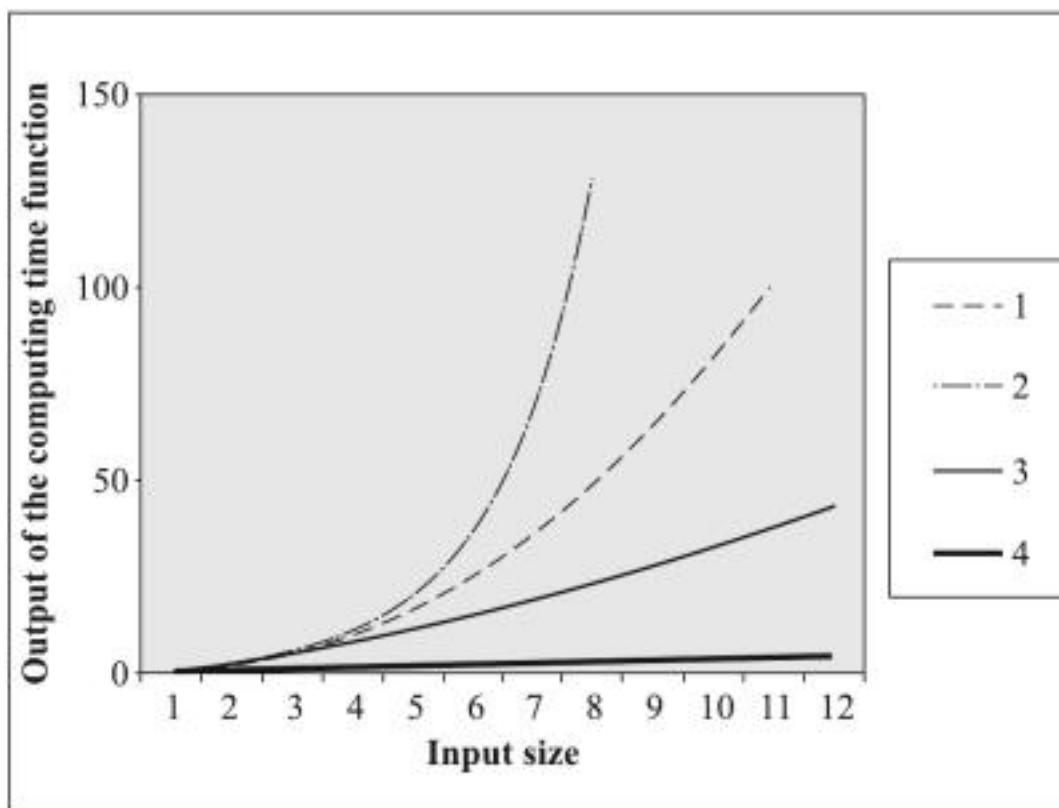


Fig. 2.1 Growth rate of some computing time functions

Average, Best and Worst Case Complexities

2.6

The time complexity of an algorithm is dependent on parameters associated with the input/output instances of the problem. Very often the running time of the algorithm is expressed as a

function of the input size. In such a case it is fair enough to presume that larger the input size of the problem instances the larger is its running time. But such is not the case always. There are problems whose time complexity is dependent not just on the size of the input but on the nature of the input as well. Example 2.1 illustrates this point.

Example 2.1 Algorithm: To sequentially search for the first-occurring even number in the list of numbers given.

Input 1: -1, 3, 5, 7, -5, 7, 11, -13, 17, 71, 21, 9, 3, 1, 5, -23, -29, 33, 35, 37, 40

Input 2: 6, 17, 71, 21, 9, 3, 1, 5, -23, 3, 64, 7, -5, 7, 11, 33, 35, 37, -3, -7, 11

Input 3: 71, 21, 9, 3, 1, 5, -23, 3, 11, 33, 36, 37, -3, -7, 11, -5, 7, 11, -13, 17, 22

Let us determine the efficiency of the algorithm for the input instances presented in terms of the number of comparisons done before the first occurring even number is retrieved. Observe that all three input instances are of the same size.

In the case of Input 1, the first occurring even number occurs as the last element in the list. The algorithm would require 21 comparisons, equivalent to the size of the list, before it retrieves the element. On the other hand, in the case of Input 2 the first occurring even number shows up as the very first element of the list thereby calling for only one comparison before it is retrieved! If Input 2 is the *best* possible case that can happen for the quickest execution of the algorithm, then Input 1 is the *worst* possible case that can happen when the algorithm takes the longest possible time to complete. Generalizing, the time complexity of the algorithm in the best possible case would be expressed as $O(1)$ and in the worst possible case would be expressed as $O(n)$ where n is the size of the input.

This justifies the statement that the running time of algorithms are not just dependent on the size of the input but also on its nature. That input instances (or instances) for which the algorithm takes the maximum possible time is called the *worst case* and the time complexity in such a case is referred to as the *worst case time complexity*. That input instances for which the algorithm takes the minimum possible time is called the *best case* and the time complexity in such a case is referred to as the *best case time complexity*. All other input instances which are neither of the two are categorized as the *average cases* and the time complexity of the algorithm in such cases is referred to as the *average case complexity*. Input 3 is an example of an average case since it is neither the best case nor the worst case. By and large, analyzing the average case behaviour of algorithms is harder and mathematically involved when compared to their worst case and best case counterparts. Also such an analysis can be misleading if the input instances are not chosen at random or appropriately to cover all possible cases that may arise when the algorithm is put to practice.

Worst case analysis is appropriate when the response time of the algorithm is critical. For example, in the case of a nuclear power plant controller, it is critical to know of the maximum limit of the system response time regardless of the input instance that is to be handled by the system. The algorithms designed cannot have a running time that exceeds this response time limit.

On the other hand in the case of applications where the input instances may be wide and varied and there is no knowing beforehand of the kind of input instance that has to be worked on, it is prudent to choose algorithms with good average case behaviour.

Analyzing Recursive Programs

2.7

Recursion is an important concept in computer science. Many algorithms can best be described in terms of recursion.

Recursive procedures

If P is a procedure containing a call statement to itself (Fig. 2.2(a)) or to another procedure that results in a call to itself (Fig. 2.2(b)), then the procedure P is said to be a *recursive procedure*. In the former case it is termed *direct recursion* and in the latter case it is termed *indirect recursion*.

Extending the concept to programming can yield program functions or programs themselves that are recursively defined. In such cases they are referred to as *recursive functions* and *recursive programs* respectively.

Extending the concept to mathematics would yield what are called *recurrence relations*.

In order that the recursively defined function may not run into an infinite loop it is essential that the following properties are satisfied by any recursive procedure:

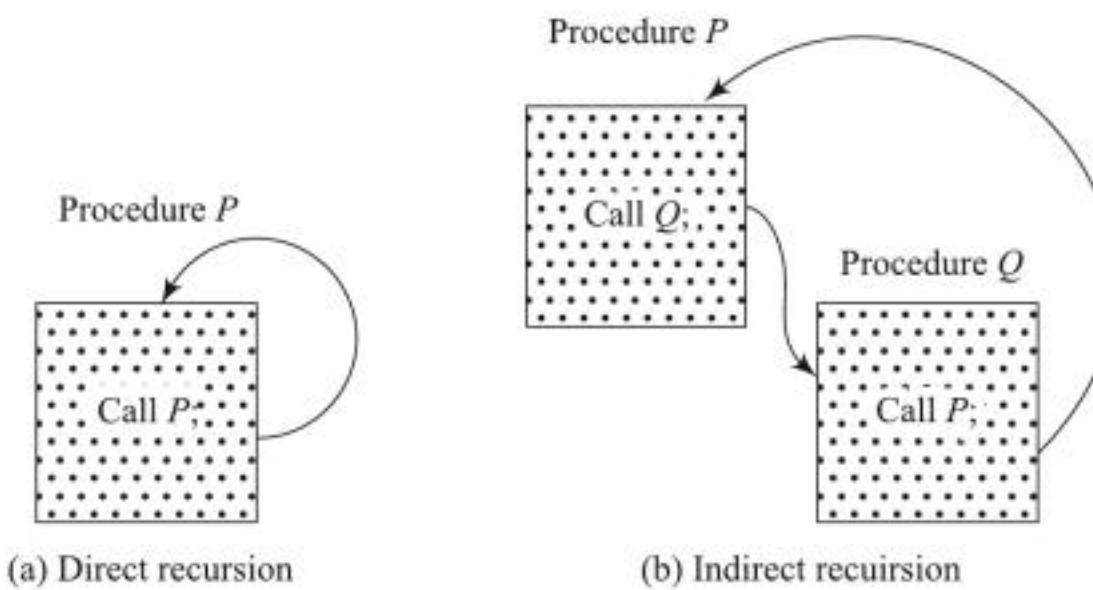


Fig. 2.2 *Skeletal recursive procedures*

- (i) There must be criteria, one or more, called the *base criteria* or simply *base case(s)*, where the procedure does not call itself either directly or indirectly.
- (ii) Each time the procedure calls itself directly or indirectly, it must be closer to the base criteria.

Example 2.2 illustrates a recursive procedure and Example 2.3 a recurrence relation. Example 2.4 describes the Tower of Hanoi puzzle which is a classic example for the application of recursion and recurrence relation.

Example 2.2 A recursive procedure to compute factorial of a number n is shown below:

$$\begin{aligned} n! &= 1, && \text{if } n = 1 \text{ (base criterion)} \\ n! &= n \cdot (n-1)! && \text{if } n > 1 \end{aligned}$$

Note the recursion in the definition of factorial function($!$). $n!$ calls $(n-1)!$ for its definition. A pseudo-code recursive function for computation of $n!$ is shown below:

```

function factorial(n)
1-2. if (n = 1) then factorial = 1;
or else
3. factorial = n* factorial(n-1);
and end factorial.

```

Example 2.3

A recurrence relation $S(n)$ is defined as below:

$$S(n) = \begin{cases} 0, & \text{if } n = 1 \text{ (base criterion)} \\ S(n/2) + 1, & \text{if } n > 1 \end{cases}$$

Example 2.4 *The Tower of Hanoi puzzle*

The Tower of Hanoi puzzle was invented by the French mathematician Edouard Lucas in 1883. There are three Pegs, Source (*S*), Intermediary (*I*) and Destination (*D*). Peg *S* contains a set of disks stacked to resemble a tower, with the largest disk at the bottom and the smallest at the top. Figure 2.3 illustrates the initial configuration of the Pegs for 6 disks. The objective is to transfer the entire tower of disks in Peg *S*, to Peg *D*, maintaining the same order of the disks. Also only one disk can be moved at a time and never can a larger disk be placed on a smaller disk during the transfer. The *I* Peg is for intermediate use during the transfer.

A simple solution to the problem, for $N = 3$ disk is given by the following transfers of disks:

1. Transfer disk from Peg *S* to Peg *D*
2. Transfer disk from Peg *S* to Peg *I*
3. Transfer disk from Peg *D* to Peg *I*
4. Transfer disk from Peg *S* to Peg *D*
5. Transfer disk from Peg *I* to Peg *S*
6. Transfer disk from Peg *I* to Peg *D*
7. Transfer disk from Peg *S* to Peg *D*

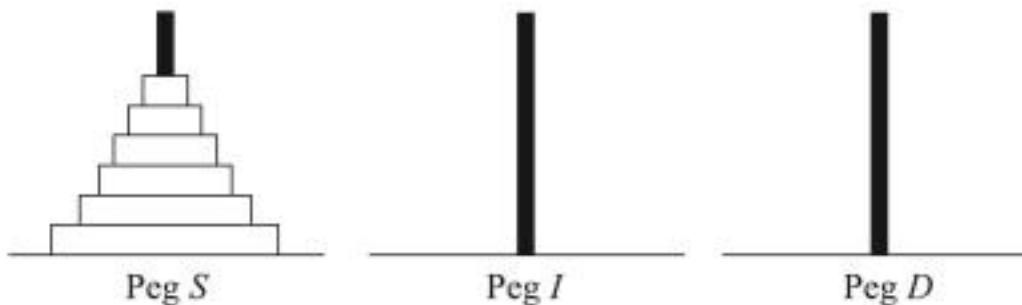


Fig. 2.3 Tower of Hanoi puzzle (initial configuration)

The solution to the puzzle calls for an application of recursive functions and recurrence relations. A skeletal recursive procedure for the solution of the problem for N number of disks, is as follows:

1. Move the top $N-1$ disks from Peg *S* to Peg *I* (using *D* as an intermediary Peg)
2. Move the bottom disk from Peg *S* to Peg *D*
3. Move $N-1$ disks from Peg *I* to Peg *D* (using Peg *S* as an intermediary Peg)

A pictorial representation of the skeletal recursive procedure for $N = 6$ disks is shown in Fig. 2.4. Function TRANSFER illustrates the recursive function for the solution of the problem.

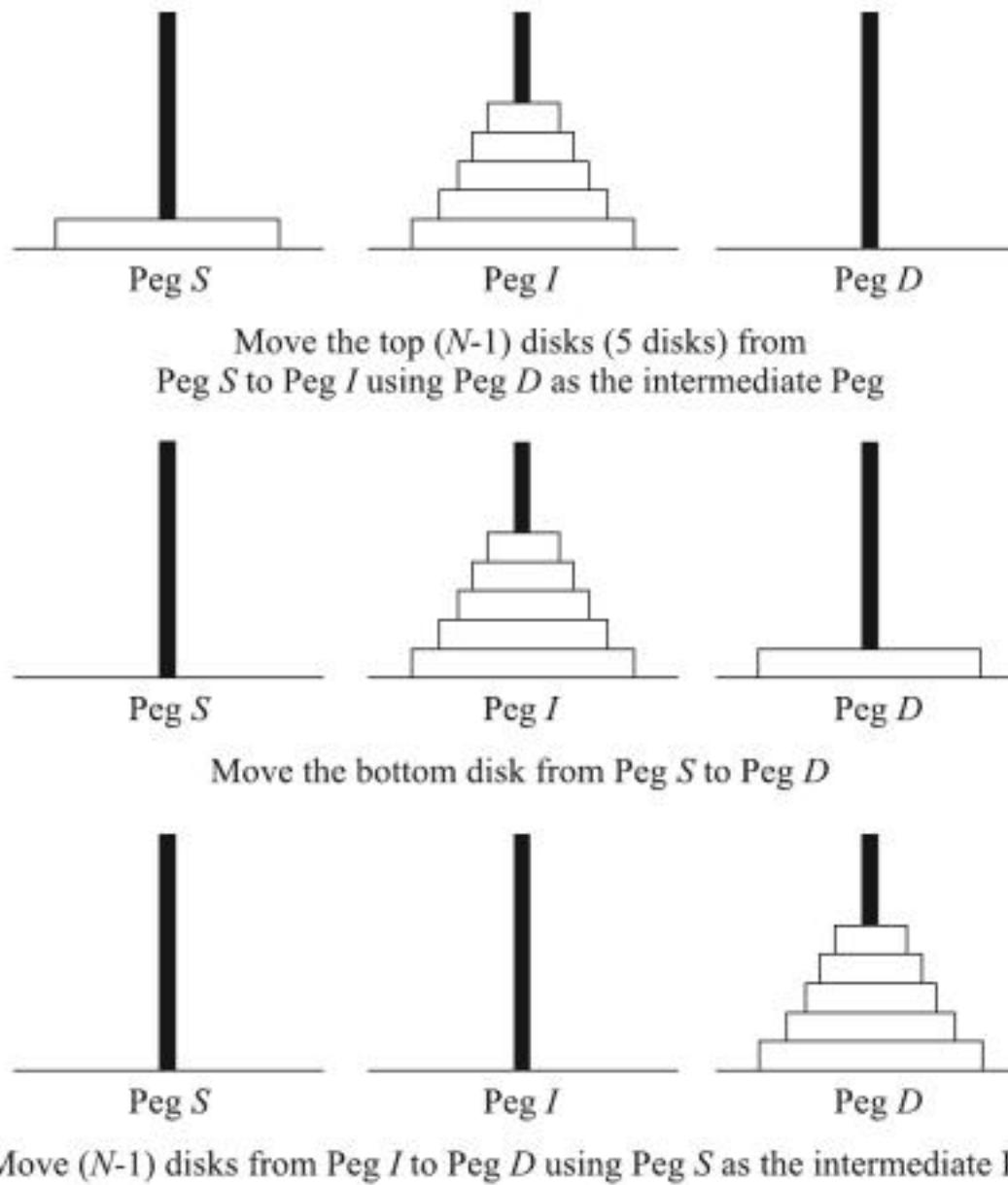


Fig. 2.4 Pictorial representation of the skeletal recursive procedure for Tower of Hanoi puzzle

```

function TRANSFER(N, S, I, D)
/* N disks are to be transferred from peg S to peg D with
peg I as the intermediate peg*/
if N is 0 then exit();
else
  {TRANSFER(N-1, S, D, I); /* transfer N-1 disks from peg S to
  peg I with peg D as the intermediate peg*/
  Transfer disk from S to D; /* move the disk which is the last
  and the largest disk, from peg S to peg D*/
  TRANSFER(N-1, I, S, D); /* transfer N-1 disks from peg I to
  peg D with peg S as the intermediate peg*/}
end TRANSFER.

```

Apriori analysis of recursive functions

The apriori analysis of recursive functions is different from that of iterative functions. In the latter case as was seen in Sec. 2.2, the total frequency count of the programs were computed before approximating them using mathematical functions such as O . In the case of recursive functions we first formulate recurrence relations that define the behaviour of the function. The solution of the recurrence relation and its approximation using the conventional O or any other notation yields the resulting time complexity of the program.

To frame the recurrence relation, we associate an unknown time function $T(n)$ where n measures the size of the arguments to the procedure. We then get a recurrence relation for $T(n)$ in terms of $T(k)$ for various values of k .

Example 2.5 illustrates obtaining the recurrence relation for the recursive factorial function FACTORIAL(n) shown in Example 2.2.

Example 2.5 Let $T(n)$ be the running time of the recursive function FACTORIAL(n). The running times of lines 1 and 2 is $O(1)$. The running time for line 3 is given by $O(1) + T(n - 1)$. Here $T(n - 1)$ is the time complexity of the call to the recursive function FACTORIAL($n-1$). Thus for some constants c, d ,

$$\begin{aligned} T(n) &= c + T(n - 1), && \text{if } n > 1 \\ &= d, && \text{if } n \leq 1 \end{aligned}$$

Example 2.6 derives the recurrence relation for the Tower of Hanoi puzzle.

Example 2.6 The recurrence relation for the Tower of Hanoi puzzle is derived as follows: Let $T(N)$ be the minimum number of transfers that are needed to solve the puzzle with N disks. From the function TRANSFER it is evident that for $N = 0$, no disks are transferred. Again for $N > 0$, two recursive calls each enabling the transfer of $(N - 1)$ disks, and a single transfer of the last (largest) disk from peg S to peg D are done. Thus the recurrence relation is given by,

$$\begin{aligned} T(N) &= 0, && \text{if } N = 0 \\ &= 2 \cdot T(N - 1) + 1, && \text{if } N > 0 \end{aligned}$$

Now what remains to be done is to solve the recurrence relation, in other words to solve for $T(n)$. Such a solution where $T(n)$ expresses itself in a form where no T occurs on the right side is termed as a *closed form solution*, in conventional mathematics.

The general method of solution is to repeatedly replace terms $T(k)$ occurring on the right side of the recurrence relation, by the relation itself with appropriate change of parameters. The substitutions continue until one reaches a formula in which T does not appear on the right side. Quite often at this stage, it may be essential to sum a series which could be either an arithmetic progression or a geometric progression or some such series. Even if we cannot obtain a sum exactly, we could work to obtain at least a close upper bound on the sum, which could serve to act as an upper bound for $T(n)$.

Example 2.7 illustrates the solution of the recurrence relation for the function FACTORIAL(n), discussed in Example 2.5 and Example 2.8 illustrates the solution of the recurrence relation for the Tower of Hanoi puzzle, discussed in Example 2.6.

Example 2.7 Solution of the recurrence relation

$$\begin{aligned} T(n) &= c + T(n - 1), && \text{if } n > 1 \\ &= d, && \text{if } n \leq 1 \end{aligned}$$

yields the following steps.

$$\begin{aligned} T(n) &= c + T(n - 1) && \dots(\text{step 1}) \\ &= c + (c + T(n - 2)) \\ &= 2c + T(n - 2) && \dots(\text{step 2}) \\ &= 2c + (c + T(n - 3)) \\ &= 3c + T(n - 3) && \dots(\text{step 3}) \end{aligned}$$

In the k th step the recurrence relation is transformed as

$$T(n) = k \cdot c + T(n - k), \quad \text{if } n > k, \quad \dots(\text{step } k)$$

Finally when ($k = n - 1$), we obtain

$$\begin{aligned} T(n) &= (n - 1) \cdot c + T(1), \\ &= (n - 1)c + d \\ &= O(n) \end{aligned} \quad \dots(\text{step } n - 1)$$

Observe how the recursive terms in the recurrence relation are replaced so as to move the relation closer to the base criterion viz., $T(n) = 1$, $n \leq 1$. The approximation of the closed form solution obtained viz., $T(n) = (n - 1)c + d$ yields $O(n)$.

Example 2.8 Solution of the recurrence relation for the Tower of Hanoi puzzle,

$$\begin{aligned} T(N) &= 0, & \text{if } N = 0 \\ &= 2 \cdot T(N - 1) + 1, & \text{if } N > 0 \end{aligned}$$

yields the following steps.

$$\begin{aligned} T(N) &= 2 \cdot T(N - 1) & \dots(\text{step 1}) \\ &= 2 \cdot (2 \cdot T(N - 2) + 1) + 1 \\ &= 2^2 T(N - 2) + 2 + 1 & \dots(\text{step 2}) \\ &= 2^2(2 \cdot T(N - 3) + 1) + 2 + 1 \\ &= 2^3 \cdot T(N - 3) + 2^2 + 2 + 1 & \dots(\text{step 3}) \end{aligned}$$

In the k th step the recurrence relation is transformed as

$$T(N) = 2^k T(N - k) + 2^{(k-1)} + 2^{(k-2)} + \dots + 2^3 + 2^2 + 2 + 1, \quad \dots(\text{step } k)$$

Finally when ($k = N$), we obtain

$$\begin{aligned} T(N) &= 2^N T(0) + 2^{(N-1)} + 2^{(N-2)} + \dots + 2^3 + 2^2 + 2 + 1 & \dots(\text{step } N) \\ &= 2^N \cdot 0 + (2^N - 1) \\ &= 2^N - 1 \\ &= O(2^N) \end{aligned}$$



Summary

- When several algorithms can be designed for the solution of a problem, there arises the need to determine which among them is the best. The efficiency of a program or an algorithm is measured by computing its time and/or space complexities. The time complexity of an algorithm is a function of the running time of the algorithm and the space complexity is a function of the space required by it to run to completion.
- The time complexity of an algorithm can be measured using Apriori analysis or Posteriori testing. While the former is a theoretical approach that is general and machine independent, the latter is completely machine dependent.

- The apriori analysis computes the time complexity as a function of the total frequency count of the algorithm. Frequency count is the number of times a statement is executed in a program.
- O , Ω , Θ , and o are asymptotic notations that are used to express the time complexity of algorithms. While O serves as the upper bound of the performance measure, Ω serves as the lower bound.
- The efficiency of algorithms is not just dependent on the input size but is also dependent on the nature of the input. This results in the categorization of worst, best and average case complexities. Worst case time complexity is that input instance(s) for which the algorithm reports the maximum possible time and best case time complexity is that for which it reports the minimum possible time.
- Polynomial algorithms are highly efficient when compared to exponential algorithms. The latter due to their rapid growth rate can quickly get beyond the computational capacity of any sophisticated computer.
- Apriori analysis of recursive algorithms calls for the formulation of recurrence relations and obtaining their closed form solutions, before expressing them using appropriate asymptotic notations.



Illustrative Problems

Problem 2.1 If $T_1(n)$ and $T_2(n)$ are the time complexities of two program fragments P_1 and P_2 where $T_1(n) = O(f(n))$ and $T_2(n) = O(g(n))$, find $T_1(n) + T_2(n)$, and $T_1(n) \cdot T_2(n)$.

Solution: Since $T_1(n) \leq c \cdot f(n)$ for some positive number c and positive integer n_1 such that $n \geq n_1$ and $T_2(n) \leq d \cdot g(n)$ for some positive number d and positive integer n_2 such that $n \geq n_2$, we obtain $T_1(n) + T_2(n)$ as follows:

$$\begin{aligned} T_1(n) + T_2(n) &\leq c \cdot f(n) + d \cdot g(n), \text{ for } n > n_0 \text{ where } n_0 = \max(n_1, n_2) \\ (\text{i.e.}) \quad T_1(n) + T_2(n) &\leq (c + d) \max(f(n), g(n)) \text{ for } n > n_0 \\ \text{Hence} \quad T_1(n) + T_2(n) &= O(\max(f(n), g(n))). \end{aligned}$$

(This result is referred to as *Rule of Sums of O notation*)

To obtain $T_1(n) \cdot T_2(n)$, we proceed as follows:

$$\begin{aligned} T_1(n) \cdot T_2(n) &\leq c \cdot f(n) \cdot d \cdot g(n) \\ &\leq k \cdot f(n) \cdot g(n) \end{aligned}$$

Therefore, $T_1(n) \cdot T_2(n) = O(f(n) \cdot g(n))$

(This result is referred to as *Rule of Products of O notation*)

Problem 2.2 If $A(n) = a_m n^m + a_{m-1} n^{m-1} + \dots + a_1 n + a_0$ then $A(n) = O(n^m)$ for $n \geq 1$.

Solution: Let us consider $|A(n)|$. We have,

$$|A(n)| = |a_m n^m + a_{m-1} n^{m-1} + \dots + a_1 n + a_0|$$

$$\begin{aligned}
 &\leq |a_m n^m| + |a_{m-1} n^{m-1}| + \dots |a_1 n| + |a_0| \\
 &\leq (|a_m| + |a_{m-1}| + \dots |a_1| + |a_0|) \cdot n^m \\
 &\leq c \cdot n^m \text{ where } c = |a_m| + |a_{m-1}| + \dots |a_1| + |a_0| \\
 \text{Hence } A(n) = O(n^m).
 \end{aligned}$$

Problem 2.3 Two algorithms A and B report time complexities expressed by the functions n^2 and 2^n respectively. They are to be executed on a machine M which consumes 10^{-6} seconds to execute an instruction. What is the time taken by the algorithms to complete their execution on machine A for an input size of 50? If another machine N , which is 10 times faster than machine M is offered for the execution, what is the largest input size that can be handled by the two algorithms on machine N ? What are your observations?

Solution: Algorithms A and B report a time complexity of n^2 and 2^n respectively. In other words each of the algorithms execute approximately n^2 and 2^n instructions respectively. For an input size of $n = 50$ and with a speed of 10^{-6} seconds per instruction, the time taken by the algorithms on machine M are as follows:

$$\text{Algorithm A: } 50^2 \times 10^{-6} = 0.0025 \text{ sec}$$

$$\text{Algorithm B: } 2^{50} \times 10^{-6} \cong 35 \text{ years}$$

If another machine N which is 10 times faster than machine M is offered, then the number of instructions that algorithms A and B can execute on machine M would also be 10 times more than that on M . Let x^2 and 2^y be the number of instructions that algorithms A and B execute on the machine N . Then the new input size that each of these algorithms can handle is given by

$$\text{Algorithm A: } x^2 = 10 \times n^2$$

$$\therefore x = \sqrt{10} \times n \cong 3n$$

That is, algorithm A can handle 3 times the original input size that it could handle on machine M .

$$\text{Algorithm B: } 2^y = 10 \times 2^n$$

$$\therefore y = \log_{10} 2 + n \cong 3 + n$$

That is, algorithm B can handle just 3 units more than the original input size that it could handle on machine M .

Observations: Since algorithm A is a polynomial algorithm, it displays a superior performance of executing the specified input on machine M in 0.0025 secs. Also when offered a faster machine N , it is able to handle 3 times the original input size that it could handle on machine M .

In contrast, algorithm B is an exponential algorithm. While it takes 35 years to process the specified input on machine M , despite the faster machine offered, it is able to process just 3 more over the input data size that it could handle on machine M .

Problem 2.4 Analyze the behaviour of the following program which computes the n^{th} Fibonacci number, for appropriate values of n . Obtain the frequency count of the statements (that are given line numbers) for various cases of n .

```

Procedure Fibonacci (n)
1.   read (n);
2-4.  if (n < 0) then Print ("error"); exit ( );
5-7.  if (n = 0) then Print (" Fibonacci number is 0");
     exit ( );
8-10. if (n = 1) then Print (" Fibonacci number is 1");
      exit ( );

11-12. f1 = 0;
        f2=1;
13.   for i = 2 to n do
14-16.   f = f1 + f2;
          f1 = f2;
          f2 = f;
17.   end
18.   Print (" Fibonacci number is", f);
end Fibonacci

```



Solution: The behaviour of the program can be analyzed for the cases as shown in Table I 2.4.

Table I 2.4

Line number	Frequency count of the statements			
	<i>n < 0</i>	<i>n = 0</i>	<i>n = 1</i>	<i>n > 1</i>
1	1	1	1	1
2	1	1	1	1
3, 4	1, 1	0	0	0
5	0	1	1	1
6, 7	0	1, 1	0	0
8	0	0	1	1
9, 10	0	0	1, 1	0
11, 12	0	0	0	1, 1
13	0	0	0	$(n - 2 + 1) + 1$
14, 15, 16	0	0	0	$(n - 1), (n - 1), (n - 1)$
17	0	0	0	$(n - 1)$
18	0	0	0	1
Total frequency count	4	5	6	$5n + 3$

Problem 2.5 Obtain the time complexity of the following program:

```

Procedure whirlpool(m)
begin

```

```

if (m ≤ 0) then print("eddy!"); exit();
else {
    swirl = whirlpool(m - 1) + whirlpool(m - 1);
    Print("whirl");
    end whirlpool
}

```

Solution: We first obtain the recurrence relation for the time complexity of the procedure `whirlpool`. Let $T(m)$ be the time complexity of the procedure. The recurrence relation is formulated as given below:

$$T(m) = \begin{cases} a, & \text{if } m \leq 0 \\ 2T(m-1) + b, & \text{if } m > 0. \end{cases}$$

Here $2T(m-1)$ expresses the total time complexity of the two calls to `whirlpool(m - 1)`. a, b indicate the constant time complexities to execute the rest of the statements when $m \leq 0$ and $m > 0$ respectively.

Solving for the recurrence relation yields the following steps:

$$\begin{aligned} T(m) &= 2 \cdot T(m-1) + b && \dots(\text{step 1}) \\ &= 2(2T(m-2) + b) + b \\ &= 2^2T(m-2) + b(1+2) && \dots(\text{step 2}) \\ &= 2^2(2T(m-3) + b) + 3b \\ &= 2^3(T(m-3) + b(1+2+2^2)) && \dots(\text{step 3}) \end{aligned}$$

Generalizing, in the i^{th} step

$$T(m) = 2^i T(m-i) + b(1+2+2^2+\dots+2^i) \quad \dots(\text{step } i)$$

When $i = m$,

$$\begin{aligned} T(m) &= 2^m T(0) + b(1+2+2^2+\dots+2^m) \\ &= a \cdot 2^m + b(2^{m+1}-1) \\ &= k \cdot 2^m + l \text{ where } k, l \text{ are positive constants} \\ &= O(2^m) \end{aligned}$$

The time complexity of procedure `whirlpool` is therefore $O(2^m)$.

Problem 2.6 The frequency count of line 3 in the following program fragment is _____.

$$(a) \frac{4n^2 - 2n}{2} \quad (b) \frac{i^2 - i}{2} \quad (c) \frac{(i^2 - 3i)}{2} \quad (d) \frac{(4n^2 - 6n)}{2}$$

1. $i = 2n$
2. **for** $j = 1$ **to** i
3. **for** $k = 3$ **to** j
4. $m = m + 1;$
5. **end**
6. **end**

Solution: The frequency count of line 3 is given by $\sum_{j=1}^i (j-3+1) + 1 = \sum_{j=1}^{2n} (j-1) = \frac{4n^2 - 2n}{2}$.

Hence the correct option is *a*.

Problem 2.7 Find the frequency count and the time complexity of the following program fragment:

1. **for** $i = 20$ **to** 30
2. **for** $j = 1$ **to** n

3. $am = am + 1;$
 4. **end**
 5. **end**

Solution: The frequency count of the program fragment is shown in Table I 2.7

Table I 2.7

Line number	Frequency count
1	12
2	$\sum_{i=20}^{30} (n+1) = 11(n+1)$
3	$\sum_{i=20}^{30} \sum_{j=1}^n 11n$
4	$11n$
5	11

The total frequency count is $33n + 34$ and time complexity is therefore $O(n)$.

Problem 2.8 State which of the following are true or false:

- (i) $f(n) = 30n^2 2^n + 6n2^n + 8n^2 = O(2^n)$
- (ii) $g(n) = 9.2^n + n^2 = \Omega(2^n)$
- (iii) $h(n) = 9.2^n + n^2 = \Theta(2^n)$

Solution:

- (i) False.
For $f(n) = O(2^n)$, it is essential that

$$\text{(i.e.) } \left| f(n) \right| \leq c \cdot |2^n|$$

$$\left| \frac{30n^2 2^n + 6n2^n + 8n^2}{2^n} \right| \leq c$$

This is not possible since the left-hand side is an increasing function.

- (ii) True.
- (iii) True.

Problem 2.9 Solve the following recurrence relation assuming $n = 2k$:

$$\begin{aligned} C(n) &= 2, n = 2 \\ &= 2 \cdot C(n/2) + 3, n > 2 \end{aligned}$$

Solution: The solution of the recurrence relation proceeds as given below:

$$\begin{aligned} C(n) &= 2 \cdot C(n/2) + 3 && \dots(\text{step 1}) \\ &= 2^2 C(n/4) + 3 + 3 \\ &= 2^2 C(n/2^2) + 3 \cdot (1 + 2) && \dots(\text{step 2}) \\ &= 2^2 (2 \cdot C(n/2^3) + 3) + 3 \cdot (1 + 2) \\ &= 2^3 C(n/2^3) + 3(1 + 2 + 2^2) && \dots(\text{step 3}) \end{aligned}$$

In the i^{th} step,

$$C(n) = 2^i C(n/2^i) + 3(1 + 2 + 2^2 + \dots + 2^{i-1}) \quad \dots(\text{step } i)$$

Since $n = 2^k$, in the step when $i = (k - 1)$,

$$\begin{aligned} C(n) &= 2^{k-1} C(n/2^{k-1}) + 3(1 + 2 + 2^2 + \dots + 2^{k-2}) \quad \dots(\text{step } k-1) \\ &= \frac{n}{2} C(2) + 3(2^{k-1} - 1) \\ &= \frac{n}{2} \cdot 2 + 3\left(\frac{n}{2} - 1\right) \\ &= 5 \cdot \frac{n}{2} - 3 \end{aligned}$$

Hence $C(n) = 5 \cdot n/2 - 3$.



Review Questions

1. The frequency count of the statement “**for** $k = 3$ **to** $(m + 2)$ **do**” is
 (a) $(m + 2)$ (b) $(m - 1)$ (c) $(m + 1)$ (d) $(m + 5)$
2. If functions $f(n)$ and $g(n)$, for a positive integer n_0 and a positive number C , are such that $|f(n)| \geq C|g(n)|$, for all $n \geq n_0$, then
 (a) $f(n) = \Omega(g(n))$ (b) $f(n) = O(g(n))$ (c) $f(n) = \Theta(g(n))$ (d) $f(n) = o(g(n))$
3. For $T(n) = 167n^5 + 12n^4 + 89n^3 + 9n^2 + n + 1$,
 (a) $T(n) = O(n)$ (b) $T(n) = O(n^5)$ (c) $T(n) = O(1)$ (d) $T(n) = O(n^2 + n)$
4. State whether true or false:
 (i) Exponential functions have rapid growth rates when compared to polynomial functions
 (ii) Therefore, exponential time algorithms run faster than polynomial time algorithms
 (a) (i) true (ii) true (b) (i) true (ii) false (c) (i) false (ii) false (d) (i) false (ii) true
5. Find the odd one out: $O(n)$, $O(n^2)$, $O(n^3)$, $O(3^n)$
 (a) $O(n)$ (b) $O(n^2)$ (c) $O(n^3)$ (d) $O(3^n)$
6. How does one measure the efficiency of algorithms?
7. Distinguish between best, worst and average case complexities of an algorithm.
8. Define O and Ω notations of time complexity.
9. Compare and contrast exponential time complexity with polynomial time complexity.
10. How are recursive programs analyzed?
11. Analyze the time complexity of the following program:

```

for send = 1 to n do
    for receive = 1 to send do
        for ack = 2 to receive do
            message = send - (receive + ack);
        end
    end
end

```

12. Solve the recurrence relation:

$$\begin{aligned} S(n) &= 2 \cdot S(n - 1) + b \cdot n, && \text{if } n > 1 \\ &= a, && \text{if } n = 1 \end{aligned}$$



ARRAYS

3

Introduction

3.1

In Chapter 1, an Abstract Data Type (ADT) was defined to be a set of data objects and the fundamental operations that can be performed on this set. In this regard, an *array* is an ADT whose objects are sequence of elements of the same type and the two operations performed on it are *store* and *retrieve*. Thus if a is an array the operations can be represented as STORE (a, i, e) and RETRIEVE (a, i) where i is termed as the index and e is the element that is to be stored in the array. These functions are equivalent to the programming language statements $a[i] := e$ and $a[i]$ where i is termed *subscript* and a the *array variable name* in programming language parlance.

Arrays could be of one-dimension, two dimension, three-dimension or in general multi-dimension. Figure 3.1 illustrates a one and two dimensional array. It may be observed that while one-dimensional arrays are mathematically likened to *vectors*, two-dimensional arrays are likened to *matrices*. In this regard, two-dimensional arrays also have the terminologies of *rows* and *columns* associated with them.

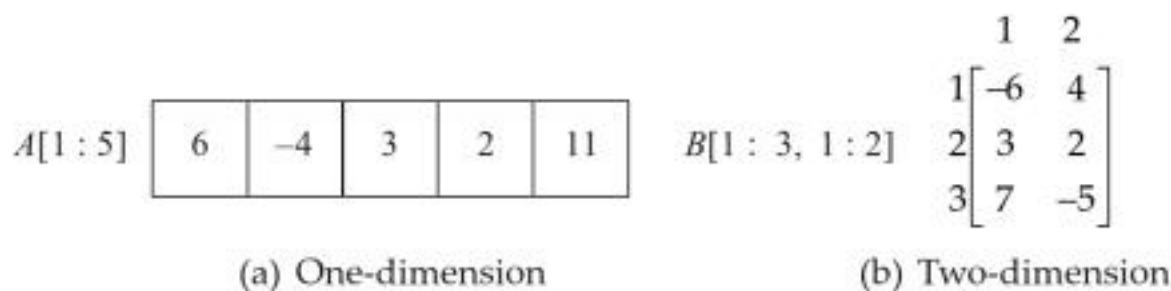


Fig. 3.1 Examples of arrays

In Fig. 3.1, $A[1:5]$ refers to a one-dimensional array where 1, 5 are referred to as the *lower* and *upper indexes* or the *lower* and *upper bounds* of the index range respectively. Similarly, $B[1:3, 1:2]$ refers to a two-dimensional array with 1, 3 and 1, 2 being the lower and upper indexes of the rows and columns respectively.

- 3.1 Introduction
- 3.2 Array Operations
- 3.3 Number of Elements in an Array
- 3.4 Representation of Arrays in Memory
- 3.5 Applications

Also, each element of the array viz., $A[i]$ or $B[i, j]$ resides in a memory location also called a *cell*. Here cell refers to a unit of memory and is machine dependent.

Array Operations

3.2

An array when viewed as a data structure supports only two operations viz.,

- (i) storage of values (i.e.) writing into an array (STORE (a, i, e)) and,
- (ii) retrieval of values (i.e.) reading from an array (RETRIEVE (a, i))

For example, if A is an array of 5 elements then Fig. 3.2 illustrates the operations performed on A .

OBJECT	REPRESENTATION IN MEMORY	OPERATIONS	RESULT OF THE OPERATIONS																				
$A[1 : 5]$	A <table border="1" style="display: inline-table; vertical-align: middle;"> <tr> <td>6</td><td>-4</td><td>3</td><td>2</td><td>11</td> </tr> <tr> <td>[1]</td><td>[2]</td><td>[3]</td><td>[4]</td><td>[5]</td> </tr> </table>	6	-4	3	2	11	[1]	[2]	[3]	[4]	[5]	STORE ($A, 3, 17$) RETRIEVE ($A, 2$)	A <table border="1" style="display: inline-table; vertical-align: middle;"> <tr> <td>6</td><td>-4</td><td>17</td><td>2</td><td>11</td> </tr> <tr> <td>[1]</td><td>[2]</td><td>[3]</td><td>[4]</td><td>[5]</td> </tr> </table> -4	6	-4	17	2	11	[1]	[2]	[3]	[4]	[5]
6	-4	3	2	11																			
[1]	[2]	[3]	[4]	[5]																			
6	-4	17	2	11																			
[1]	[2]	[3]	[4]	[5]																			

Fig. 3.2 Array operations: Store and Retrieve

Number of Elements in an Array

3.3

In this section, the computation of size of the array by way of number of elements is discussed. This is important because, when arrays are declared in a program, it is essential that the number of memory locations needed by the array are 'booked' before hand.

One-dimensional array

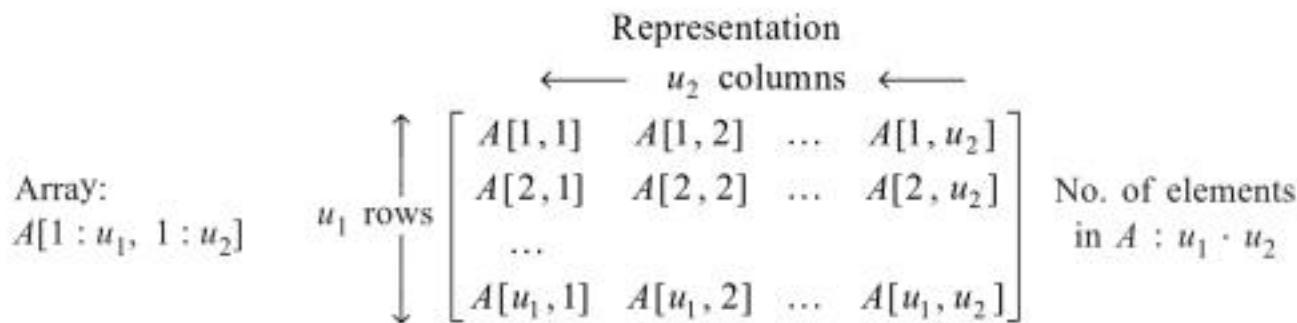
Let $A[1:u]$ be a one-dimensional array. The size of the array, as is evident is u and the elements are $A[1], A[2], \dots, A[u - 1], A[u]$. In the case of the array $A[l : u]$ where l is the lower bound and u is the upper bound of the index range, the number of elements is given by $(u - l + 1)$.

Example 3.1 The number of elements in

- (i) $A[1:26] = 26$
- (ii) $A[5:53] = 49$ ($\because 53 - 5 + 1$)
- (iii) $A[-1:26] = 28$

Two-dimensional array

Let $A[1 : u_1, 1 : u_2]$ be a two-dimensional array where u_1 indicates the number of rows and u_2 the number of columns in the array. Then the number of elements in A is $u_1 \cdot u_2$. Generalizing, $A[l_1 : u_1, l_2 : u_2]$ has a size of $(u_1 - l_1 + 1)(u_2 - l_2 + 1)$ elements. Figure 3.3 illustrates a two dimensional array and its size.

**Fig. 3.3** Size of a two-dimensional array

Example 3.2 The number of elements in

- $A[1:10, 1:5] = 10 \times 5 = 50$
- $A[-1:2, 2:6] = 4 \times 5 = 20$
- $A[0:5, -1:6] = 6 \times 8 = 48$

Multi-dimensional array

A multi-dimensional array $A[1 : u_1, 1 : u_2, \dots, 1 : u_n]$ has a size of $u_1 \cdot u_2 \cdots u_n$ elements, (i.e.) $\prod_{i=1}^n u_i$.

Figure 3.4 illustrates a three-dimensional array and its size. Generalizing, the array $A[l_1 : u_1, l_2 : u_2, l_3 : u_3, \dots, l_n : u_n]$ has a size of $\prod_{i=1}^n (u_i - l_i + 1)$ elements.

Array:	Elements	Number of elements
$A[1 : 2, 1 : 2, 1 : 3]$	$A[1, 1, 1], A[1, 1, 2], A[1, 1, 3]$ $A[1, 2, 1], A[1, 2, 2], A[1, 2, 3]$ $A[2, 1, 1], A[2, 1, 2], A[2, 1, 3]$ $A[2, 2, 1], A[2, 2, 2], A[2, 2, 3]$	$2 \times 2 \times 3 = 12$

Fig. 3.4 Size of a three-dimensional array

Example 3.3 The number of elements in

- $A[-1 : 3, 3 : 4, 2 : 6] = (3 - (-1) + 1)(4 - 3 + 1)(6 - 2 + 1) = 50$
- $A[0 : 2, 1 : 2, 3 : 4, -1 : 2] = 3 \times 2 \times 2 \times 4 = 48$

Representation of Arrays in Memory

3.4

How are arrays represented in memory? This is an important question at least from the compiler's point of view. In many programming languages the name of the array is associated with the address of the starting memory location so as to facilitate efficient storage and retrieval. Also it is to be remembered that while the computer memory is considered one-dimensional (linear) it has to accommodate arrays which are multi-dimensional. Hence address calculation to determine the appropriate locations in the memory becomes important.

In this respect, it is convenient to imagine a two-dimensional array $A[1 : u_1, 1 : u_2]$ as u_1 number of one-dimensional arrays whose dimension is u_2 . Again, in the case of three-dimensional arrays $A[1 : u_1, 1 : u_2, 1 : u_3]$ it can be viewed as u_1 number of two-dimensional arrays of size $u_2 \times u_3$. Figure 3.5 illustrates this idea. Generalizing, a multi-dimensional array $A[1 : u_1, 1 : u_2, \dots, 1 : u_n]$ is a colony of u_1 number of arrays each of dimension $A[1 : u_2, 1 : u_3, \dots, 1 : u_n]$.

The arrays are stored in the memory in one of the two ways, viz., **row major order** or **lexicographic order** or **column major order**. In the ensuing discussion we assume a row major order representation. Figure 3.6 distinguishes between the two methods of representation.

One-dimensional array

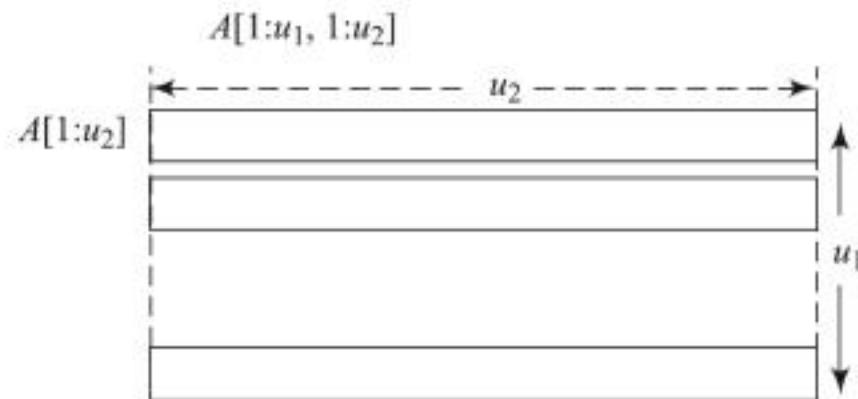
Consider the array $A(1 : u_1)$ and let α be the address of the starting memory location referred to as the **base address** of the array. Here as is evident, $A[1]$ occupies the memory location whose address is α , $A(2)$ occupies $\alpha + 1$ and so on. In general, the address of $A[i]$ is given by $\alpha + (i - 1)$. Figure 3.7 illustrates the representation of a one-dimensional array in memory. In general, for a one-dimensional array $A(l_1 : u_1)$ the address of $A[i]$ is given by $\alpha + (i - l_1)$, where α is the base address.

Example 3.4 For the array given below with base address $\alpha = 100$, the addresses of the array elements specified are computed as given below:

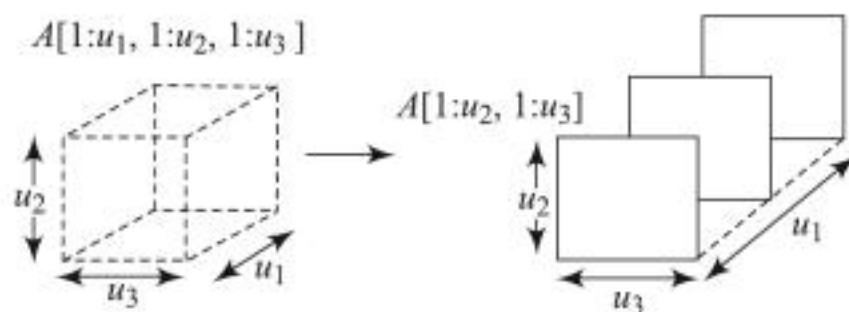
Array	Element	Address
(i) $A[1:17]$	$A[7]$	$\alpha + (7 - 1) = 100 + 6 = 106$
(ii) $A[-2:23]$	$A[16]$	$\alpha + (16 - (-2)) = 100 + 18 = 118$

Two-dimensional array

Consider the array $A[1 : u_1, 1 : u_2]$ which is to be stored in the memory. It is helpful to imagine this array as u_1 number of one-dimensional arrays of length u_2 . Thus if $A[1, 1]$ is stored in address α , the base address, then $A[i, 1]$ has address $\alpha + (i - 1)u_2$, and $A[i, j]$ has address $\alpha + (i - 1)u_2 + (j - 1)$. To understand this let us imagine the two-dimensional array $A[i, j]$ to be a building with i floors each accommodating j rooms. To access room $A[i, 1]$, the first room in the i^{th} floor, one has to traverse $(i - 1)$ floors each having u_2 rooms. In other words, $(i - 1) \cdot u_2$ rooms have to be



(a) Two-dimensional array viewed in terms of one-dimensional arrays



(b) Three-dimensional array viewed in terms of two-dimensional arrays

Fig. 3.5 Viewing higher-dimensional arrays in terms of their lower-dimensional counter parts

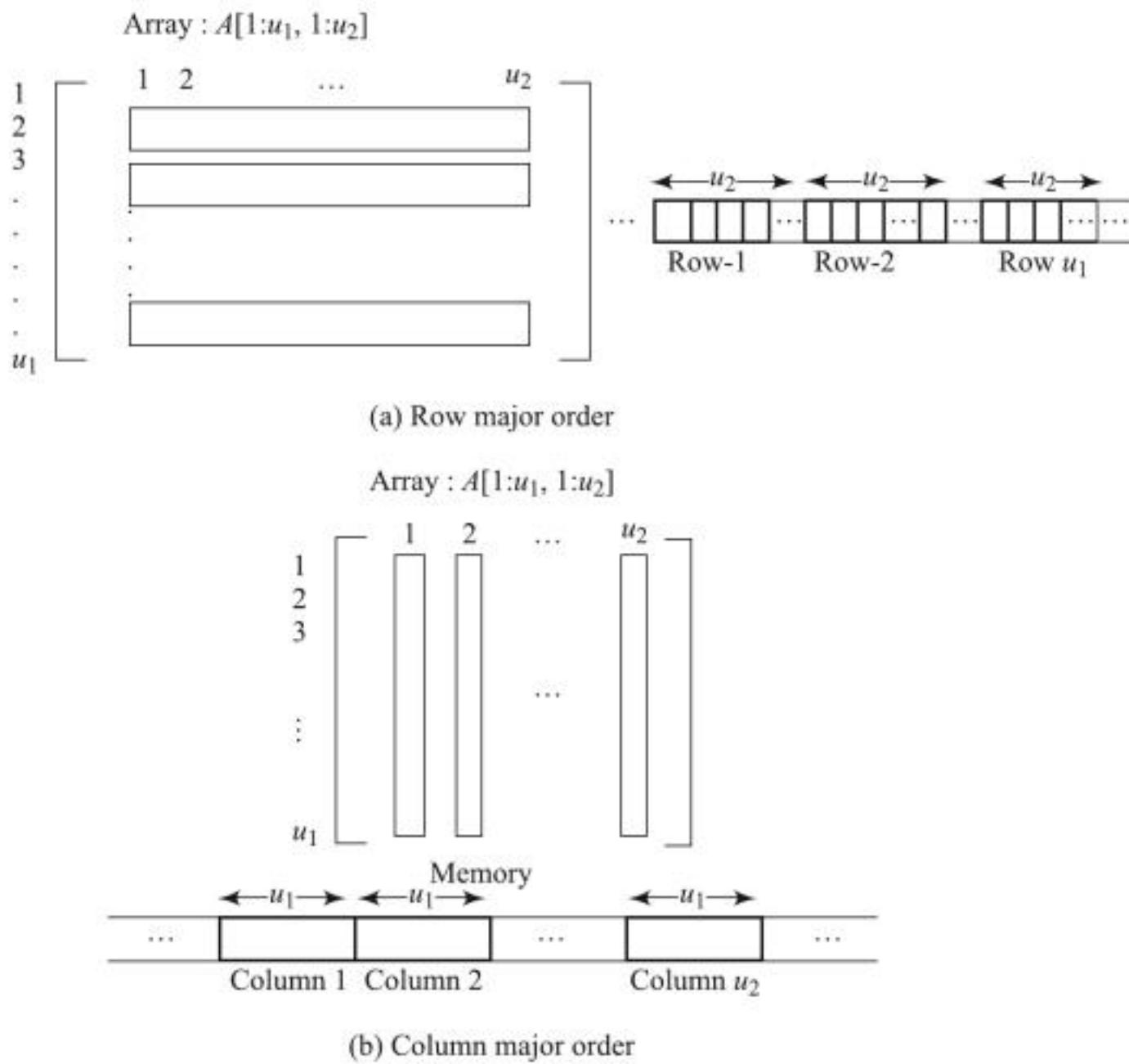


Fig. 3.6 Row major order and column major order of a two-dimensional array

Array :	Memory:					
$A[1:u_1]$	α	$\alpha + 1$	$\alpha + 2$	$\alpha + (u_1 - 1)$		
...	$A[1]$	$A[2]$	$A[3]$...	$A[u_1]$...

Fig. 3.7 Representation of one-dimensional arrays in memory

left behind before one knocks at the first room in the i^{th} floor. Since α is the base address, the address of $A[i, 1]$ would be $\alpha + (i - 1)u_2$. Again, extending a similar argument to access $A[i, j]$, the j^{th} room on the i^{th} floor, one has to leave behind $(i - 1)u_2$ rooms and reach the j^{th} room of the i^{th} floor. This again as before, would compute the address of $A[i, j]$ as $\alpha + (i - 1)u_2 + (j - 1)$. Figure 3.8 illustrates the representation of two-dimensional arrays in the memory.

Observe that the addresses of array elements are expressed in terms of the cells, which hold the array.

In general, for a two-dimensional array $A[l_1 : u_1, l_2 : u_2]$ the address of $A[i, j]$ is given by

$$\alpha + (i - l_1)(u_2 - l_2 + 1) + (j - l_2)$$

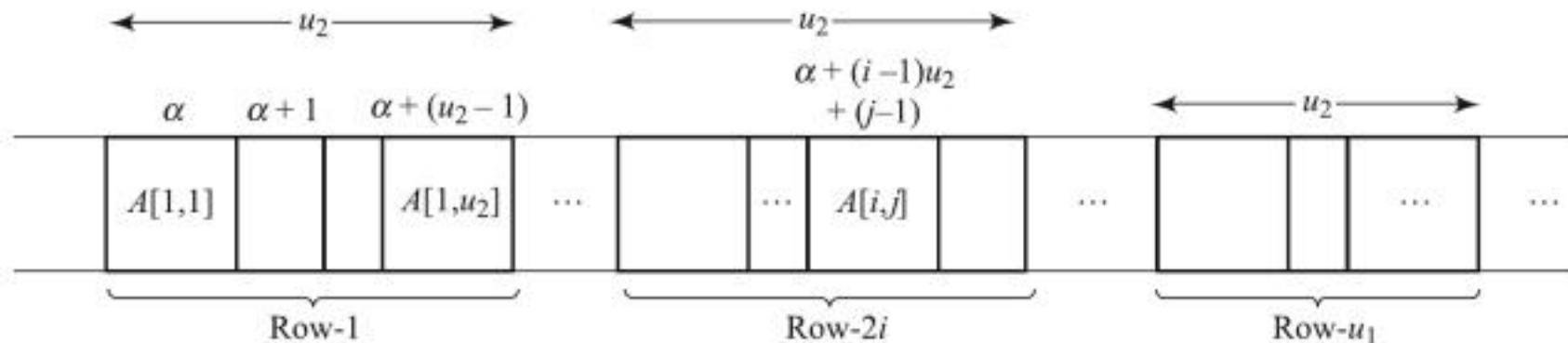


Fig. 3.8 Representation of a two-dimensional array in memory

Example 3.5 For the arrays given below with $\alpha = 220$ as the base address, the addresses of the elements specified, are computed as given below:

Array	Element	Address
$A[1 : 10, 1 : 5]$	$A[8, 3]$	$220 + (8 - 1)5 + (3 - 1) = 257$
$A[-2 : 4, -6 : 10]$	$A[3, -5]$	$220 + (3 - (-2))(10 - (-6) + 1) + (-5 - (-6)) = 306$

Three-dimensional array

Consider the three-dimensional array $A[1 : u_1, 1 : u_2, 1 : u_3]$. As discussed before, we shall imagine it to be u_1 number of two-dimensional arrays of dimension $u_2 \cdot u_3$. Reverting to the analogy of building-floor-rooms, the three dimensional array $A[i, j, k]$ could be viewed as a colony of i buildings each having j floors with each floor accommodating k rooms.

To access $A[i, 1, 1]$, (i.e.) the first room in the first floor of the i^{th} building, one has to walk past $(i - 1)$ buildings each comprising $u_2 u_3$ rooms, before climbing on to the first floor of the i^{th} building to reach the first room! This means the address of $A[i, 1, 1]$ would be $\alpha + (i - 1)u_2 u_3$. Similarly the address of $A[i, j, 1]$ requires accessing the first room on the j^{th} floor of the i^{th} building which works out to $\alpha + (i - 1)u_2 u_3 + (j - 1)u_3$. Proceeding on similar lines, the address of $A[i, j, k]$ is given by $\alpha + (i - 1)u_2 u_3 + (j - 1)u_3 + (k - 1)$.

Figure 3.9 illustrates the representation of three-dimensional arrays in the memory.

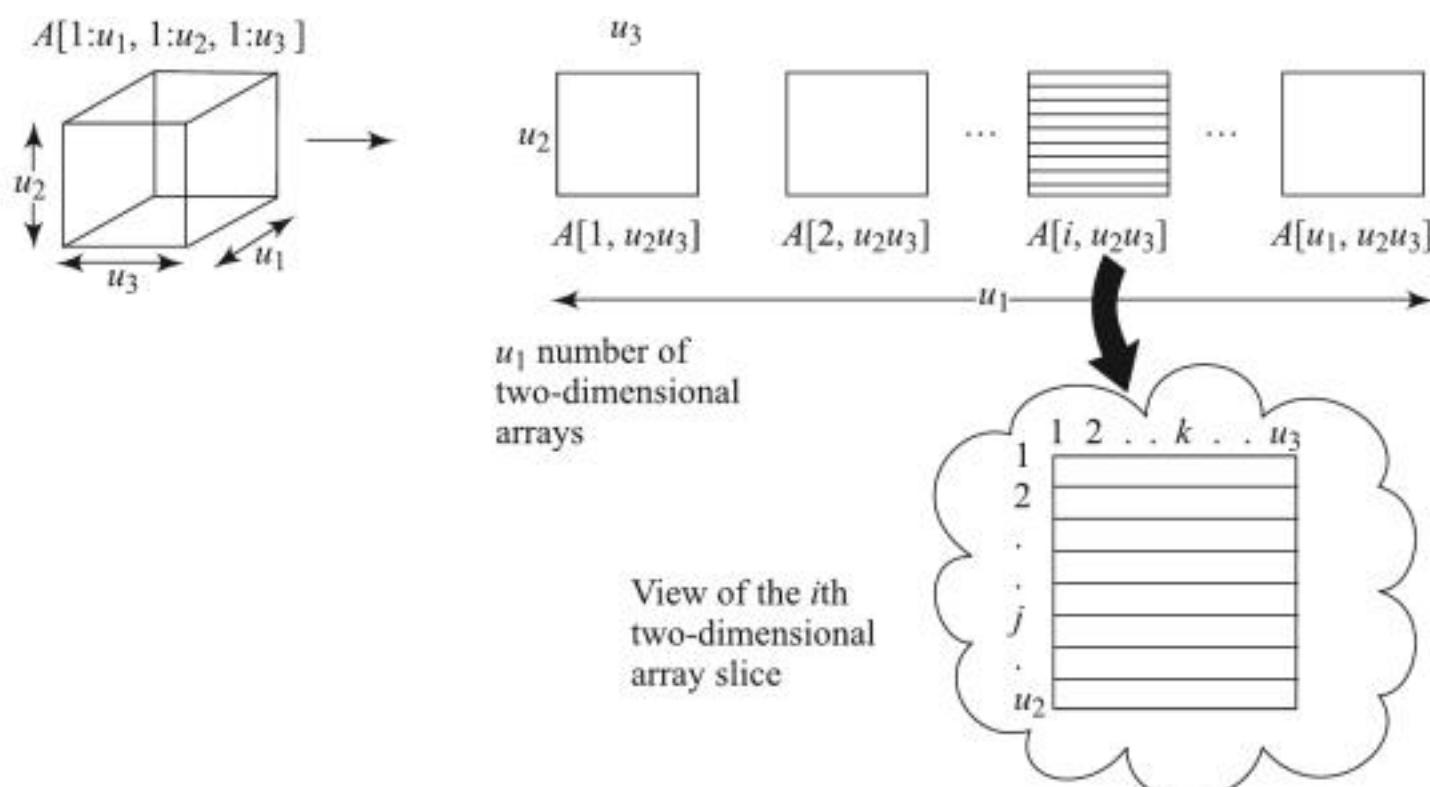


Fig. 3.9 Representation of three-dimensional arrays in the memory

In general for a three-dimensional array $A[l_1 : u_1, l_2 : u_2, l_3 : u_3]$ the address of $A[i, j, k]$ is given by

$$\alpha + (i - l_1)(u_2 - l_2 + 1)(u_3 - l_3 + 1) + (j - l_2)(u_3 - l_3 + 1) + (k - l_3)$$

Example 3.6 For the arrays given below with base address $\alpha = 110$ the addresses of the elements specified are as given below:

Array	Element	Address
$A[1 : 5, 1 : 2, 1 : 3]$	$A[2, 1, 3]$	$110 + (2 - 1)6 + (1 - 1)3 + (3 - 1) = 118$
$A[-2 : 4, -6 : 10, 1 : 3]$	$A[-1, -4, 2]$	$110 + (-1 - (-2))17.3 + (-4 - (-6))3 + (2 - 1) = 168$

N-dimensional array

Let $A[1 : u_1, 1 : u_2, \dots, 1 : u_N]$ be an N -dimensional array. The address calculation for the retrieval of various elements are as given below:

Element	Address
$A[i_1, 1, 1, \dots, 1]$	$\alpha + (i_1 - 1)u_2u_3 \cdot u_N$
$A[i_1, i_2, 1, 1, \dots, 1]$	$\alpha + (i_1 - 1)u_2u_3 \cdot u_N + (i_2 - 1)u_3 \cdot u_4 \dots u_N$
$A[i_1, i_2, i_3, 1, 1, \dots, 1]$	$\alpha + (i_1 - 1)u_2u_3 \cdot u_N + (i_2 - 1)u_3u_4 \cdot u_N + (i_3 - 1)u_4u_5 \dots u_N$
$A[i_1, i_2, i_3, \dots, i_N]$	$\alpha + (i_1 - 1)u_2u_3 \cdot u_N + (i_2 - 1)u_3u_4 \cdot u_N + \dots + (i_N - 1)$ $= \alpha + \sum_{j=1}^N (i_j - 1)a_j$ where $a_j = \prod_{k=j+1}^N u_k, 1 \leq j < N$

Applications

3.5

In this section, we introduce two concepts that are useful to computer science and also serve as applications of arrays viz., Sparse matrices and ordered lists.

Sparse matrix

A matrix is a mathematical object which finds its applications in various scientific problems. A matrix is an arrangement of $m.n$ elements arranged as m rows and n columns. The **Sparse matrix** is a matrix with **zeros as the dominating elements**. There is no precise definition for a sparse matrix. In other words, the “sparseness” is relatively defined. Figure 3.10 illustrates a matrix and a sparse matrix.

$$\begin{bmatrix} 2 & 4 & 6 & 8 \\ 1 & 2 & 0 & 2 \\ 0 & 1 & 1 & 6 \\ 2 & 0 & 1 & 4 \end{bmatrix}$$

(a) Matrix

$$\begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 2 & 0 & 0 \end{bmatrix}$$

(b) Sparse Matrix

Fig. 3.10 Matrix and a sparse matrix

A matrix consumes a lot of space in memory. Thus, a 1000×1000 matrix needs 1 million storage locations in memory. Imagine the situation when the matrix is sparse! To store a handful of non-zero elements, voluminous memory is allotted and thereby wasted!

In such a case to save valuable storage space, we resort to a triple representation viz., (i, j, value) to represent each non-zero element of the sparse matrix. In other words, a sparse matrix A is represented by another matrix $B[0 : t, 1 : 3]$ with $t + 1$ rows and 3 columns. Here t refers to the number of non-zero elements in the sparse matrix. While rows 1 to t record the details pertaining to the non-zero elements as triple (that is 3 columns), the zeroth row viz. $B[0, 1]$, $B[0, 2]$ and $B[0, 3]$ record the number of non-zero elements of the original sparse matrix A . Figure 3.11 illustrates a sparse matrix representation

$A[1 : 7, 1 : 6]$	$B[0 : 5, 1 : 3]$
0 1 0 0 0 0	7 6 5
0 0 0 0 0 0	1 2 1
-2 0 0 1 0 0	3 1 -2
0 0 0 0 0 0	3 4 1
0 0 0 0 0 0	6 2 -3
0 -3 0 0 0 0	7 6 1
0 0 0 0 0 1	

Fig. 3.11 Sparse matrix representation

A simple example of a sparse matrix arises in the arrangement of choice of say 5 elective courses from the specified list of 100 elective courses, by 20000 students of a university. The arrangement of choice would turn out to be a matrix with 20000 rows and 100 columns with just 5 non-zero entries per row, indicative of the choice made. Such a matrix could definitely be classified as sparse!

Ordered lists

One of the simplest and useful data objects in computer science is an *ordered list* or *linear list*. An ordered list can be either empty or non empty. In the latter case, the elements of the list are known as *atoms*, chosen from a set D . The ordered lists provide a variety of operations such as retrieval, insertion, deletion, update etc. The most common way to represent an ordered list is by using a one-dimensional array. Such a representation is termed *sequential mapping* though better forms of representation have been presented in the literature.

Example 3.7 The following are ordered lists

- (i) (sun, mon, tue, wed, thu, fri, sat)
 - (ii) ($a_1, a_2, a_3, a_4, \dots, a_n$)
 - (iii) (Unix, CP/M, Windows, Linux)
- The ordered lists represented as one-dimensional arrays are given as follows:

WEEK [1 : 7]

...	sun	mon	tue	wed	thu	fri	sat	...
	[1]	[2]	[3]	[4]	[5]	[6]	[7]	

VARIABLE [1 : N]

...	a_1	a_2	a_3	...	a_N	
	[1]	[2]	[3]		[N]	

OS [1 : 4]

...	Unix	CP/M	Windows	Linux	...
	[1]	[2]	[3]	[4]	

We illustrate below some of the operations performed on ordered lists, with examples.

Operation	Original ordered list	Resultant ordered list after the operation
Insertion (Insert a_6)	(a_1, a_2, a_7, a_9)	$(a_1, a_2, a_6, a_7, a_9)$
Deletion (Delete a_9)	(a_1, a_2, a_7, a_9)	(a_1, a_2, a_7)
Update (update a_2 to a_5)	(a_1, a_2, a_7, a_9)	(a_1, a_5, a_7, a_9)

ADT for Arrays**Data objects:**

A set of elements of the same type stored in a sequence

Operations:

- Store value VAL in the i^{th} element of the array ARRAY
ARRAY[i] = VAL
- Retrieve the value in the i^{th} element of array ARRAY as VAL
VAL = ARRAY[i]

**Summary**

- Array as an ADT supports only two operations STORE and RETRIEVE.
- Arrays may be one, two or multi dimensioned and stored in memory either in row major order or column major order, in consecutive memory locations
- Since memory is considered one dimensional and arrays may be multi-dimensional it becomes essential to know the representations of arrays in memory, especially from the

compiler's point of view. The address calculation of array elements has been elaborately discussed.

- Two concepts viz., sparse matrices and ordered lists, of use to computer science have been briefly described as applications of arrays.

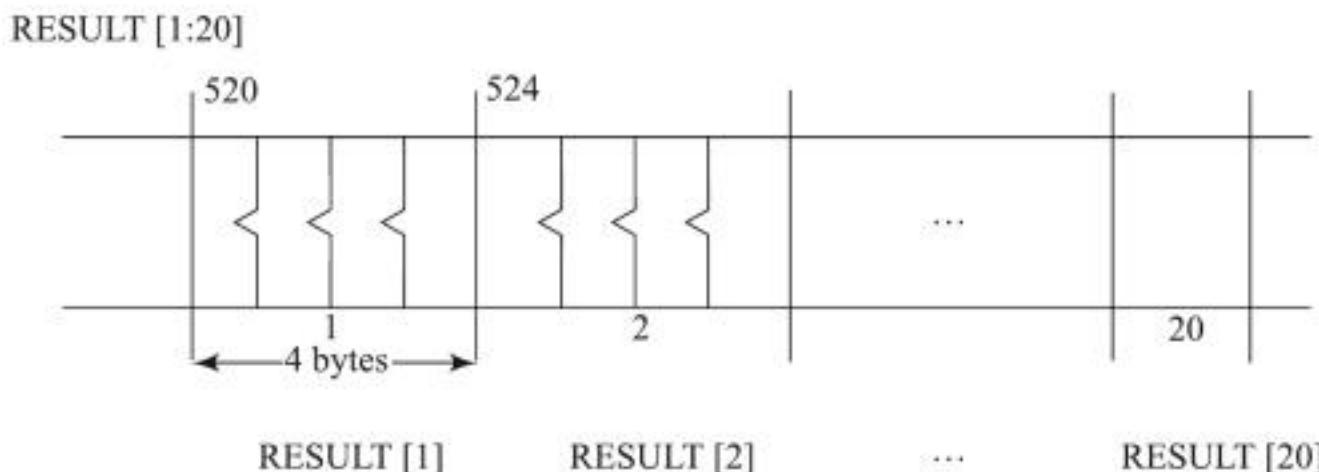


Illustrative Problems

Problem 3.1 The following details are available about an array RESULT. Find the address of RESULT[17].

Base address	: 520
Index range	: 1:20
Array type	: Real
Size of the memory location	: 4 bytes

Solution: Since RESULT[1:20] is a one-dimensioned array, the address for RESULT[17] is given by base address + (17 – lower index). However, the cell is made of 4 bytes, hence the address



is given by base address + (17 – lower index) · 4 = 520 + (17 – 1) · 4 = 584
The array RESULT may be visualized as shown.

Problem 3.2 For the following array B , compute

- (i) the dimension of B
- (ii) the space occupied by B in the memory
- (iii) the address of $B[7, 2]$

Array : B	Column index: 0:5
Base address : 1003	Size of the memory location : 4 bytes
Row index : 0:15	

Solution:

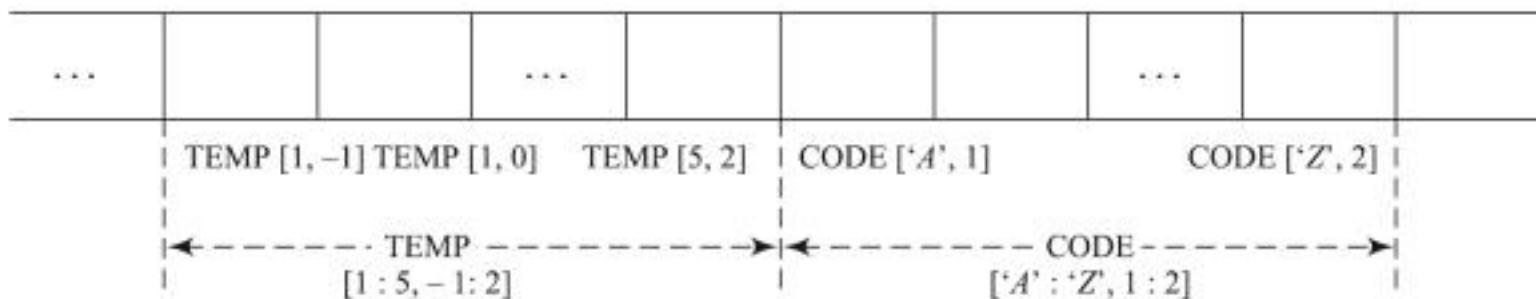
- (i) The number of elements in B is $16 \times 6 = 96$
- (ii) The space occupied by B is $96 \times 4 = 384$ bytes
- (iii) The address of $B[7, 2]$ is given by

$$\begin{aligned} 1003 + (7 - 0) \cdot 6 + (2 - 0) &= 1003 + 42 + 2 \\ &= 1047 \end{aligned}$$

Problem 3.3 A programming language permits indexing of arrays with character subscripts; for example, CHR_ARRAY['A':'D']. In such a case the elements of the array are CHR_ARRAY['A'], CHR_ARRAY['B'] etc. and the ordinal number (ORD) of the characters viz., ORD('A') = 1, ORD('B') = 2, ORD('Z') = 26 and so on are used to denote the index.

Now two arrays TEMP[1 : 5, -1 : 2] and CODE['A' : 'Z', 1 : 2] are stored in the memory beginning from address 500. Also CODE succeeds TEMP in storage. Calculate the addresses of (i) TEMP [5, -1] (ii) CODE['N',2] and (iii) CODE['Z',1].

Solution: From the details given, the representation of TEMP and CODE arrays in memory is as given below:



- (i) The address of TEMP[5, -1] is given by

$$\begin{aligned} \text{base-address} + (5 - 1)(2 - (-1) + 1) + (-1 - (-1)) \\ = 500 + 16 \\ = 516 \end{aligned}$$

- (ii) To obtain the addresses of CODE elements it is necessary to obtain its base address which is the immediate location after TEMP[5, 2], the last element of array TEMP.
Hence the address of TEMP[5, 2] is computed as

$$\begin{aligned} 500 + (5 - 1)(2 - (-1) + 1) + (2 - (-1)) \\ = 500 + 16 + 3 \\ = 519 \end{aligned}$$

Therefore the base address of CODE is given by 520.

Now the address of CODE ['N', 2] is given by

$$\begin{aligned} \text{base address of CODE} + (\text{ORD}('N') - \text{ORD}('A'))(2 - 1 + 1) + (2 - 1) \\ = 520 + (14 - 1)2 + 1 \\ = 547 \end{aligned}$$

- (iii) The address of CODE['Z',1] is computed as

$$\begin{aligned} \text{Base-address of CODE} + ((\text{ORD}('Z') - \text{ORD}('A'))(2 - 1 + 1)) + (1 - 1) \\ \text{Of CODE} \end{aligned}$$

$$\begin{aligned} &= 520 + (26 - 1) \cdot (2) + 0 \\ &= 570 \end{aligned}$$

Note: The base address of CODE may also be computed as

$$\begin{aligned} \text{Base-address of TEMP} + (\text{number of elements in TEMP} - 1) + 1 \\ = 500 + (5.4 - 1) + 1 \\ = 520 \end{aligned}$$



Review Questions

$$\begin{bmatrix} 0 & 0 & 0 & -7 & 0 \\ 0 & -5 & 0 & 0 & 0 \\ 3 & 0 & 6 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 9 & 0 & 0 & 4 & 0 \end{bmatrix}$$



Programming Assignments

1. Declare a one, two and a three-dimensional array in a programming language(such as C) which has the capability to display the addresses of array elements. Verify the various address calculation formulae that you have learnt in this chapter against the arrays that you have declared in the program.
 2. For the matrix A given below obtain a sparse matrix representation B . Write a program to
 - (i) Obtain B given matrix A as input, and
 - (ii) Obtain the transpose of A using matrix B .

	1	2	3	4	5	6	7	8	9	10	11	12
1	0	0	0	0	0	0	0	0	0	0	0	0
2	0	-1	0	0	0	2	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0	0
A: 10×12	5	4	0	0	-3	0	0	0	0	1	0	0
6	0	0	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0	0	0
8	-1	0	0	0	5	0	0	0	0	0	0	0
9	0	0	0	0	0	0	2	0	0	4	0	0
10	0	0	0	0	0	0	0	1	1	0	0	0

3. Open an ordered list $L[d_1, d_2, \dots, d_n]$ where each d_i is the name of a peripheral device, which is maintained in the alphabetical order.

Write a program to

- (i) Insert a device d_k onto the list L
- (ii) Delete an existing device d_i from L . In this case the new ordered list should be $L^{new} = (d_1, d_2, \dots, d_{i-1}, d_{i+1}, \dots, d_n)$ with $(n - 1)$ elements
- (iii) Find the length of L
- (iv) Update device d_j to d_l and print the new list.



STACKS

4

- 4.1 Introduction
- 4.2 Stack Operations
- 4.3 Applications

In this chapter we introduce the stack data structure, the operations supported by it and their implementation. Also, we illustrate two of its useful applications in computer science among the innumerable available.

Introduction

4.1

A *stack* is an ordered list with the restriction that elements are added or deleted from only one end of the list termed *top of stack*. The other end of the list which lies 'inactive' is termed *bottom of stack*.

Thus if S is a stack with three elements a, b, c where c occupies the top of stack position, and if d were to be added, the resultant stack contents would be a, b, c, d . Note that d occupies the top of stack position. Again, initiating a delete or remove operation would automatically throw out the element occupying the top of stack, viz., d . Figure 4.1 illustrates this functionality of the stack data structure.

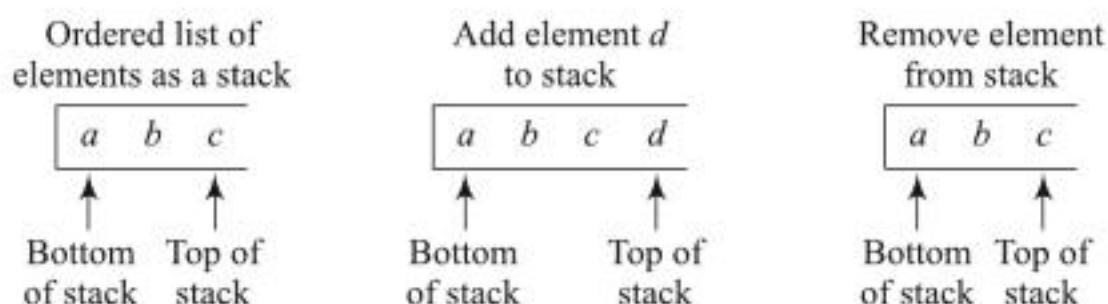


Fig. 4.1 Stack and its functionality

It needs to be observed that during insertion of elements into the stack it is essential that their identities are specified, whereas for removal no identity need be specified since by virtue of its functionality, the element which occupies the top of stack position is automatically removed.

The stack data structure therefore obeys the principle of Last In First Out (LIFO). In other words, elements inserted or added into the stack join last and those that joined last are the first to be removed.

Some common examples of a stack occur during the serving of slices of bread arranged as a pile on a platter or during the usage of an elevator (Fig. 4.2). It is obvious that when one adds a slice to a pile or removes one for serving, it is the top of the pile that is affected. Similarly, in

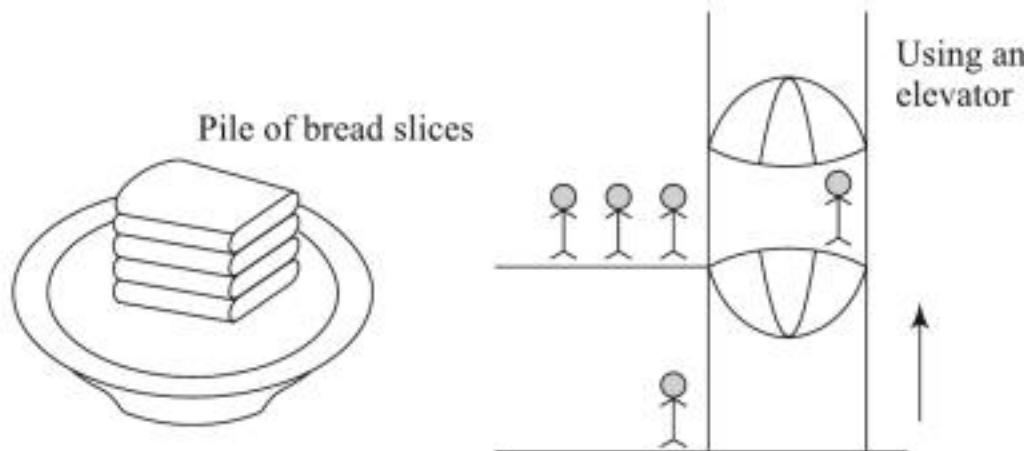


Fig. 4.2 Common examples of a stack

the case of an elevator, the last person to board the cabin has to be the first person to alight from it (at least to make room for the others to alight!)

Stack Operations

4.2

The two operations which stack data structure supports are

- (i) Insertion or addition of elements known as *Push*
- (ii) Deletion or removal of elements known as *Pop*

Before we discuss the operations supported by stack in detail, it is essential to know how stacks are implemented.

Stack implementation

A common and a basic method of implementing stacks is to make use of another fundamental data structure viz., arrays. While arrays are sequential data structures the other alternative of employing linked data structures have been successfully attempted and applied. We discuss this elaborately in Chapter 7. In this chapter we confine our discussion to the implementation of stacks using arrays.

Figure 4.3 illustrates an array based implementation of stacks. This is fairly convenient considering the fact that stacks are uni-dimensional ordered lists and so are arrays which despite their multi-dimensional structure are inherently associated with a one-dimensional consecutive set of memory locations. (Refer Chapter 3).

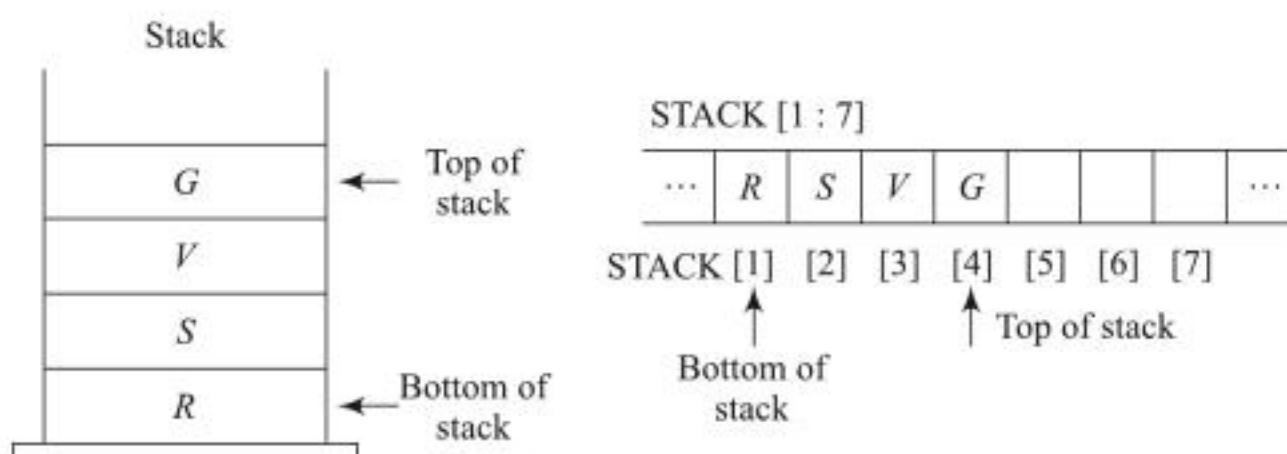


Fig. 4.3 Array implementation of stacks

Figure 4.3 shows a stack of four elements R, S, V, G represented by an array $\text{STACK}[1:7]$. In general, if a stack is represented as an array $\text{STACK}[1 : n]$ then n elements and not one more can be stored in the stack. It therefore becomes essential to issue a signal or warning termed STACK_FULL when elements whose number is over and above n are attempted to be pushed into the stack.

Again, during a pop operation, it is essential to ensure that one does not delete an empty stack! Hence the necessity for a signal or a warning termed STACK_EMPTY during the implementation of the pop operation. While implementation of stacks using arrays necessitates checking for $\text{STACK_FULL}/\text{STACK_EMPTY}$ conditions during push/pop operations respectively, the implementation of stacks with linked data structures dispenses with these testing conditions.

Implementation of push and pop operations

Let $\text{STACK } [1:n]$ be an array implementation of a stack and top be a variable recording the current top of stack position. top is initialized to 0. item is the element to be pushed into the stack. n is the maximum capacity of the stack.

Algorithm 4.1: Implementation of push operation on a stack

```
Procedure PUSH(STACK, n, top, item)
    if (top = n) then STACK_FULL;
    else
        {top = top + 1;
        STACK[top] = item; /* store item as top element
        of STACK */ }
    end PUSH
```

In the case of pop operation, as said earlier, no element identity need be specified since by default the element occupying the top of stack position is deleted. However, in Algorithm 4.2, item is used as an output variable which stores a copy of the element removed.

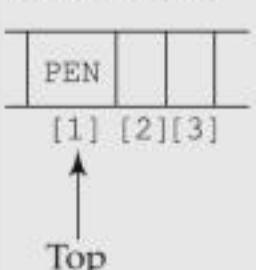
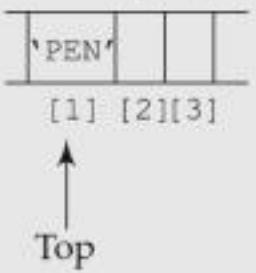
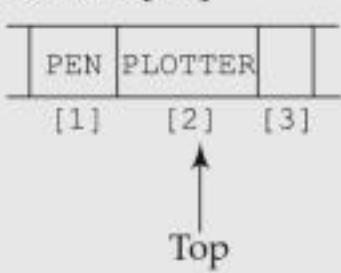
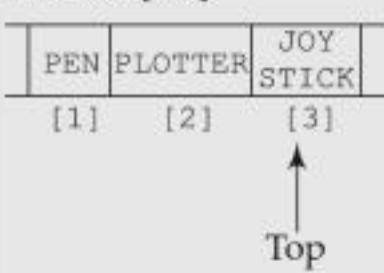
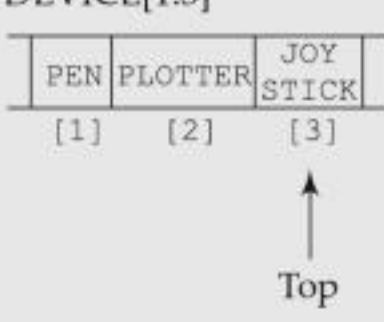
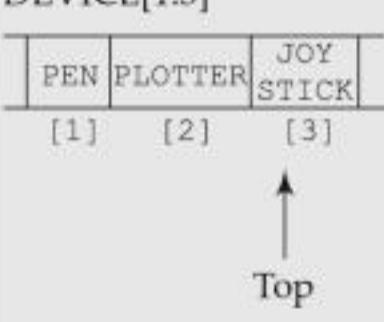
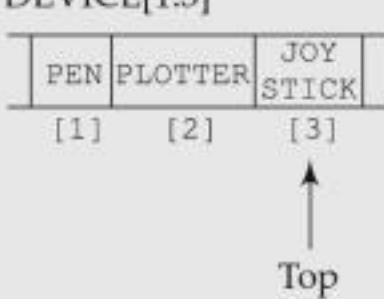
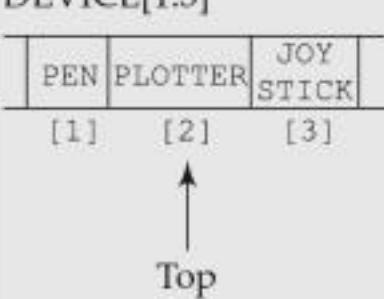
Algorithm 4.2: Implementation of pop operation on a stack

```
Procedure POP(STACK, top, item)
    if (top = 0) then STACK_EMPTY;
    else { item = STACK[top];
            top = top - 1;
    }
    end POP
```

It is evident from the algorithms that to perform a single push/pop operation the time complexity is $O(1)$.

Example 4.1 Consider a stack $\text{DEVICE}[1:3]$ of peripheral devices. The insertion of the four items PEN, PLOTTER, JOY STICK and PRINTER into DEVICE and a deletion are illustrated in Table 4.1

Table 4.1 Push/pop operations on stack DEVICE[1:3]

Stack operation	Stack before operation	Algorithm invocation	Stack after operation	Remarks
1. Push 'PEN' into DEVICE[1:3]	DEVICE[1:3] 	PUSH(DEVICE, 3, 0, 'PEN')	DEVICE[1:3] 	Push 'PEN' Successful
2. Push 'PLOTTER' into DEVICE[1:3]	DEVICE[1:3] 	PUSH(DEVICE, 3, 1, 'PLOTTER')	DEVICE[1:3] 	Push 'PLOTTER' successful
3. Push 'JOY STICK' into DEVICE[1:3]	DEVICE[1:3] 	PUSH(DEVICE, 3, 2, 'JOY STICK')	DEVICE[1:3] 	Push 'JOY STICK' successful
4. Push 'PRINTER' into DEVICE[1:3]	DEVICE[1:3] 	PUSH(DEVICE, 3, 3, 'PRINTER')	DEVICE[1:3] 	Push 'PRINTER' failure! STACK- FULL condition invoked
5. Pop from DEVICE[1:3]	DEVICE[1:3] 	POP(DEVICE, 3, ITEM)	DEVICE[1:3] 	ITEM = 'JOY STICK' Pop operation successful

Note that in operation 5 which is a pop operation, the top pointer is merely decremented as a mark of deletion. No physical erasure of data is carried out.

Applications

4.3

Stacks have found innumerable applications in computer science and other allied areas. In this section we introduce two applications of stacks which are useful in computer science, viz.,

- (i) Recursive programming, and (ii) Evaluation of expressions

Recursive programming

The concept of recursion and recursive programming had been introduced in Chapter 2. In this section we demonstrate through a sample recursive program how stacks are helpful in handling recursion. Consider the recursive pseudo-code for factorial computation shown in Fig. 4.4. Observe the recursive call in Step 3. It is essential that during the computation of $n!$, the procedure does not lead to an endless series of calls to itself! Hence the need for a base case $0! = 1$ which is in Step 1. The spate of calls made by procedure FACTORIAL() to itself based on the value of n , can be viewed as FACTORIAL() replicating itself as many times as it calls itself with varying values of n . Also, all these procedures await normal termination before the final output of $n!$ is completed and displayed by the very first call made to FACTORIAL(). A procedural call would have a normal termination only when either the base case is executed (Step 1) or the recursive case has successfully ended, (i.e.) Steps 2-5 have completed their execution.

During the execution, to keep track of the calls made to itself and to record the status of the parameters at the time of the call, a stack data structure is used. Figure 4.5 illustrates the various snap shots of the stack during the execution of FACTORIAL(5). Note that the values of the three parameters of the procedure FACTORIAL() viz., n , x , y are kept track of in the stack data structure.

```
Procedure FACTORIAL(n)
Step 1: if (n = 0) then FACTORIAL = 1;
Step 2: else {x = n - 1;
Step 3:           y = FACTORIAL(x);
Step 4:           FACTORIAL = n * y;
Step 5: end FACTORIAL
```

Fig. 4.4 Recursive procedure to compute $n!$

When the procedure FACTORIAL(5) is initiated (Fig. 4.5(a)) and executed (Fig. 4.5(b)) x obtains the value 4 and the control flow moves to Step 3 in the procedure FACTORIAL(5). This initiates the next call to the procedure as FACTORIAL(4). Observe that the first call (FACTORIAL(5)) has not yet finished its execution when the next call (FACTORIAL(4)) to the procedure has been issued. Therefore there is this need to preserve the values of the variables used viz., n , x , y , in the preceding calls. Hence the need for a stack data structure.

Every new procedure call pushes the current values of the parameters involved into the stack, thereby preserving the values used by the earlier calls. Figures 4.5(c-d) illustrate the contents of the stack during the execution of FACTORIAL(4) and subsequent procedure calls. During the execution of FACTORIAL(0) (Fig. 4.5(e)) Step 1 of the procedure is satisfied and this terminates the procedure call yielding the value FACTORIAL = 1. Since the call for FACTORIAL(0) was initiated in Step 3 of the previous call (FACTORIAL(1)), y acquires the value of FACTORIAL(0) (i.e.)

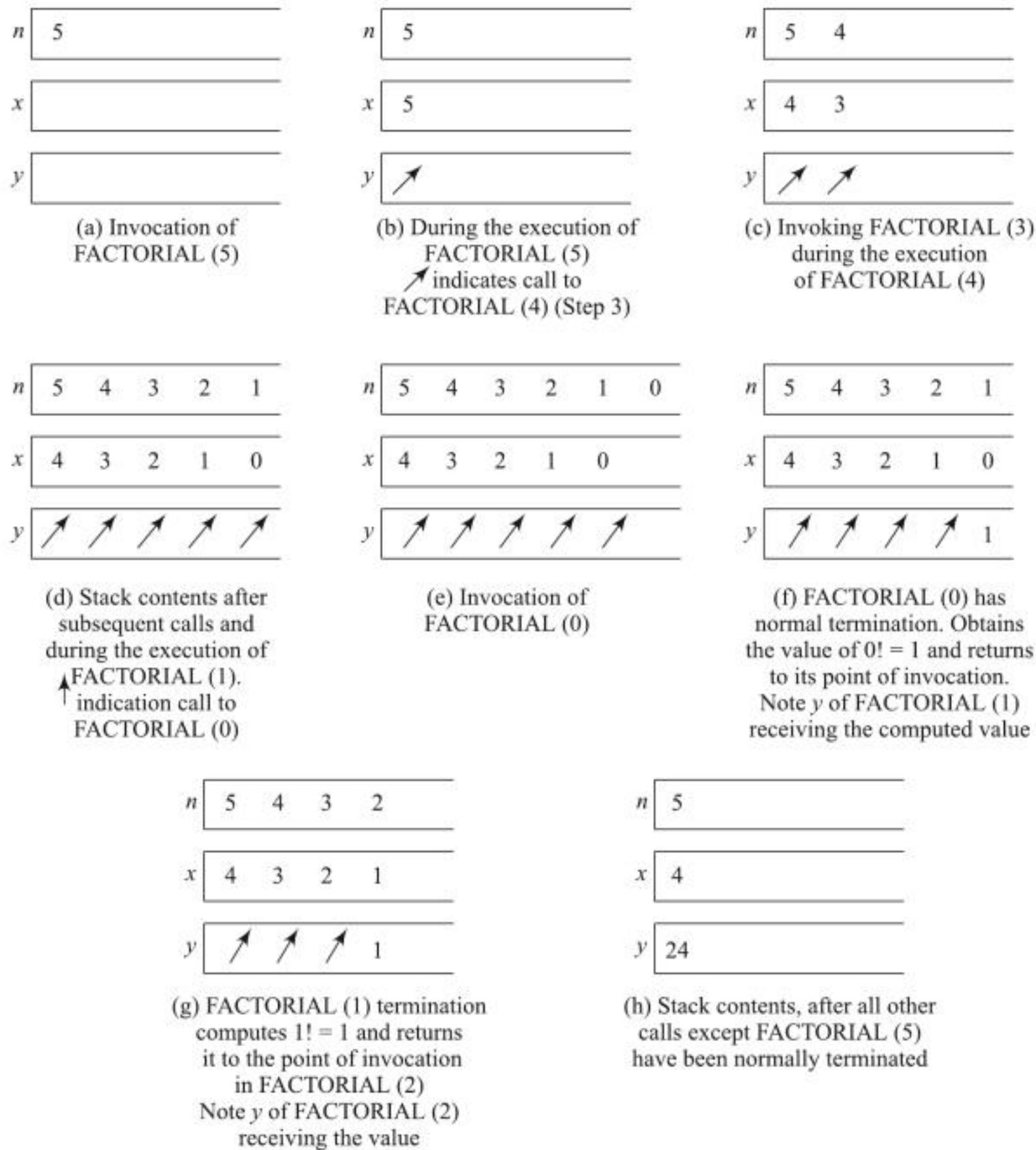


Fig. 4.5 Snapshots of the stack data structure during the execution of the procedural call FACTORIAL(5)

1 and the execution control moves to Step 4 to compute $\text{FACTORIZATION} = n * y$ (i.e.) $\text{FACTORIZATION} = 1 * 1 = 1$. With this computation, FACTORIAL(1) terminates its execution. As said earlier, FACTORIAL(1) returns the computed value of 1 to Step 3 of the previous call FACTORIAL(2). Once again it yields the result $\text{FACTORIZATION} = n * y = 2 * 1 = 2$, which terminates the procedure call to FACTORIAL(2) and returns the result to Step 3 of the previous call FACTORIAL(3) and so on.

Observe that the stack data structure grows due to a series of push operations during the procedure calls and unwinds itself by a series of pop operations until it reaches the step associated with the first procedure call, to complete its execution and display the result.

During the execution of FACTORIAL(5), the first and the oldest call to be made, y in Step 3 computes $y = \text{FACTORIZATION}(4) = 24$ and proceeds to obtain $\text{FACTORIZATION} = n * y = 5 * 24 = 120$ which is the desired result.

Tail recursion *Tail recursion* or *Tail-end recursion* is a special case of recursion where a recursive call to the function turns out to be the last action in the calling function. Note that the recursive call needs to be the *last executed statement* in the function and not necessarily the last statement in the function.

Generally, in a stack implementation of a recursive call, all the local variables of the function that are to be “remembered”, are pushed into the stack when the call is made. Upon termination of the recursive call, the local variables are popped out and restored to their previous values. Now for tail recursion, since the recursive call turns out to be the last executed statement, there is no need that the local variables must be pushed into a stack for them to be “remembered” and “restored” on termination of the recursive call. This is because when the recursive call ends, the calling function itself terminates at which all local variables are automatically discarded.

Tail recursion is considered important in many high level languages, especially functional programming languages. These languages rely on tail recursion to implement iteration. It is known that compared to iterations, recursions need more stack space and tail recursions are ideal candidates for transformation into iterations.

Evaluation of expressions

Infix, Prefix and Postfix Expressions The evaluation of expressions is an important feature of compiler design. When we write or understand an arithmetic expression for example, $-(A + B) \uparrow C * D + E$, we do so by following the scheme of *<operator> <operand> <operator>* (i.e.) an *<operator>* is preceded and succeeded by an *<operand>*. Such an expression is termed *infix expression*. It is already known how infix expressions used in programming languages have been accorded rules of hierarchy, precedence and associativity to ensure that the computer does not misinterpret the expression but computes its value in a unique way.

In reality the compiler re-works on the infix expression to produce an equivalent expression which follows the scheme of *<operand> <operand> <operator>* and is known as *postfix expression*. For example, the infix expression $a + b$ would have the equivalent postfix expression $a\ b\ +$. A third category of expression is the one which follows the scheme of *<operator> <operand> <operand>* and is known as *prefix expression*. For example, the equivalent prefix expression corresponding to $a + b$ is $+a\ b$. Examples 4.2, 4.3 illustrate the hand computation of prefix and postfix expressions from a given infix expression.

Example 4.2 Consider an infix expression $a + b * c - d$. The equivalent postfix expression can be hand computed by decomposing the original expression into sub expressions based on the usual rules of hierarchy, precedence and associativity.

Expression	Sub expression chosen based on rules of hierarchy, precedence and associativity	Postfix expression
(i) $a + \underline{b * c} - d$ ①	$b * c$	①: $bc *$
(ii) $a + \underline{\textcircled{1}} - d$ ②	$a + \textcircled{1}$	$a \textcircled{1} +$ (i.e) ②: $abc * +$
(iii) $\underline{\textcircled{2}} - d$ ③	$\textcircled{2} - d$	$\textcircled{2} d -$ (i.e) ③: $abc * + d -$

Hence $abc * + d -$ is the equivalent postfix expression of $a + b * c - d$.

Example 4.3 Consider the infix expression $(a * b - f * h) \uparrow d$. The equivalent prefix expression is hand computed as given below:

Expression	Sub expression chosen based on rules of hierarchy, precedence and associativity	Prefix expression
(i) $\underline{a * b} - f * h) \uparrow d$ ①	$a * b$	①: $* ab$
(ii) $(\textcircled{1}) - \underline{f * h) \uparrow d}$ ②	$f * h$	②: $* fh$
(iii) $(\underline{\textcircled{1}} - \textcircled{2}) \uparrow d$ ③	$(\textcircled{1} - \textcircled{2})$	③: $- \textcircled{1} \textcircled{2}$ (i.e) $- * ab * fh$
(iv) $\underline{\textcircled{3} \uparrow d}$ ④	$\textcircled{3} \uparrow d$	④: $\uparrow \textcircled{3} d$ (i.e) $\uparrow - * ab * fh d$

Hence the equivalent prefix expression of $(a * b - f * h) \uparrow d$ is $\uparrow - * ab * fh d$.

Evaluation of postfix expressions As discussed earlier, the compiler finds it convenient to evaluate an expression in its postfix form. The virtues of postfix form include elimination of parentheses which signify priority of evaluation and the elimination of the need to observe rules of hierarchy, precedence and associativity during evaluation of the expression. This implies that the evaluation of a postfix expression is done by merely undertaking a left to right scan of the expression, pushing operands into a stack and evaluating the operator with the appropriate number of operands popped out from the stack and finally placing the output of the evaluated expression into the stack.

Algorithm 4.3 illustrates the evaluation of a postfix expression. Here the postfix expression is terminated with \$ to signal end of input.

Algorithm 4.3: Procedure to evaluate a postfix expression E

```

Procedure EVAL_POSTFIX( $E$ )
     $X = \text{get\_next\_character } (E);$ 
        /* get the next character of expression  $E$  */
    case  $x$  of
        : $x$  is an operand: Push  $x$  into stack  $S$ ;
        : $x$  is an operator: Pop out required number of operands
                            from the stack  $S$ , evaluate the
                            operator and push the result into
                            the stack  $S$ ;
        : $x = \$$ :           Pop out the result from stack  $S$ ;
    end case
end EVAL-POSTFIX.

```

The evaluation of a postfix expression using Algorithm EVAL_POSTFIX is illustrated in Example 4.4.

Example 4.4 To evaluate the postfix expression of $A + B * C \uparrow D$ for $A = 2$, $B = -1$, $C = 2$ and $D = 3$, using Algorithm EVAL_POSTFIX.

The equivalent postfix expression can be computed to be $ABCD \uparrow * +$.

The evaluation of the postfix expression using the algorithm is illustrated below: The values of the operands pushed into stack S are given within parentheses e.g. $A(2)$, $B(-1)$ etc.

X	Stack S	Action
A	$A(2)$	Push A into S
B	$A(2) B(-1)$	Push B into S
C	$A(2) B(-1) C(2)$	Push C into S
D	$A(2) B(-1) C(2) D(3)$	Push D into S

(Contd.)

(Contd.)

\uparrow	$A(2) \ B(-1) \ 8$	Pop out two operands from stack S viz. $C(2)$, $D(3)$. Compute $C \uparrow D$ and push the result $C \uparrow D = 2 \uparrow 3 = 8$ into stack S.
*	$A(2) - 8$	Pop out $B(-1)$ and 8 from stack S. Compute $B * 8 = -1 * 8 = -8$ and push the result into stack S.
+	-6	Pop out $A(2)$, -8 from stack S. Compute $A - 8 = 2 - 8 = -6$ and push the result into stack S
\$		Pop out -6 from stack S and output the same as the result.

ADT for Stacks

Data objects:

A finite set of elements of the same type

Operations:

- Create an empty stack and initialize top of stack
CREATE(STACK)
- Check if stack is empty
CHK_STACK_EMPTY(STACK) (Boolean function)
- Check if stack is full
CHK_STACK_FULL(STACK) (Boolean function)
- Push ITEM into stack STACK
PUSH(STACK, ITEM)
- Pop element from stack STACK and output the element popped in ITEM
POP(STACK, ITEM)



Summary

- A stack data structure is an ordered list with insertions and deletions done at one end of the list known as top of stack.
- An insert operation is called as a push operation and delete operation is called as pop operation.
- A stack can be commonly implemented using the array data structure. However, in such a case it is essential to take note of stack full / stack empty conditions during the implementation of push and pop operations respectively.
- Two applications of the stack data structure, viz.,
 - (i) Handling recursive programming, and
 - (ii) Evaluation of postfix expressions
 have been detailed.



Illustrative Problems

Problem 4.1 Following is a pseudo code of a series of operations on a stack S . $\text{PUSH}(S, X)$ pushes an element X into S , $\text{POP}(S, X)$ pops out an element from stack S as X , $\text{PRINT}(X)$ displays the variable X and $\text{EMPTYSTACK}(S)$ is a Boolean function which returns true if S is empty and false otherwise. What is the output of the code?

- | | |
|--------------------------|---|
| 1. $X := 30;$ | 9. $\text{PUSH}(S, Z);$ |
| 2. $Y := 15;$ | 10. $\text{POP}(S, X);$ |
| 3. $Z := 20;$ | 11. $\text{PUSH}(S, 20);$ |
| 4. $\text{PUSH}(S, X);$ | 12. $\text{PUSH}(S, X);$ |
| 5. $\text{PUSH}(S, 40);$ | 13. while not $\text{EMPTYSTACK}(S)$ do |
| 6. $\text{POP}(S, Z);$ | 14. $\text{POP}(S, X);$ |
| 7. $\text{PUSH}(S, Y);$ | 15. $\text{PRINT}(X);$ |
| 8. $\text{PUSH}(S, 30);$ | 16. end |

Solution: We track the contents of the stack S and the values of the variables X, Y, Z as below:

Steps	Stack S	Variables		
		X	Y	Z
1-3		30	15	20
4	30	30	15	20
5	30 40	30	15	20
6	30	30	15	40
7	30 15	30	15	40
8	30 15 30	30	15	40
9	30 15 30 40	30	15	40
10	30 15 30	40	15	40
11	30 15 30 20	40	15	40
12	30 15 30 20 40	40	15	40

The execution of Steps 13-16 repeatedly pops out the elements from S displaying each element. The output therefore would be,

40 20 30 15 30

with the stack S empty.

Problem 4.2 Use procedure $\text{PUSH}(S, X)$, $\text{POP}(S, X)$, $\text{PRINT}(X)$ and $\text{EMPTY_STACK}(S)$ (as described in Illustrative Problem 4.1) and $\text{TOP_OF_STACK}(S)$ which returns the top element of stack S to write pseudo code for

- Assign X to the bottom element of the stack S leaving the stack empty.
- Assign X to the bottom element of the stack leaving the stack unchanged.
- Assign X to the n^{th} element in the stack (from the top) leaving the stack unchanged.

Solution:

```
(i) while not EMPTYSTACK(S) do
    POP(S, X)
end
PRINT(X);
```

X holds the element at the bottom of the stack.

- Since the stack S has to be left unchanged we make use of another stack T to temporarily hold the contents of S .

```
while not EMPTYSTACK(S) do
    POP(S, X)
    PUSH(T, X)
end                                /* empty contents of S into T */
PRINT(X);                            /* output X */
while not EMPTYSTACK(T) do
    POP(T, Y)
    PUSH(S, Y)
end                                /* empty contents of T back into S */
```

- We make use of a stack T to remember the top n elements of stack S before replacing it back into S .

```
for i := 1 to n do
    POP(S, X)
    PUSH(T, X)
end                                /* Push top n elements of S into T */
PRINT(X);                            /* display X */
for i = 1 to n do
    POP(T, Y);
    PUSH(S, Y);
end                                /* Replace back the top n elements available in T into S */
```

Problem 4.3 What is the output produced by the following segment of code where for a stack S , $\text{PUSH}(S, X)$, $\text{POP}(S, X)$, $\text{PRINT}(X)$, $\text{EMPTY_STACK}(S)$ are procedures as described in Illustrative Problem 4.1 and $\text{CLEAR}(S)$ is a procedure which empties the contents of the stack S ?

- | | |
|-----------------------|------------------------------|
| 1. TERM = 3; | 6. else |
| 2. CLEAR(STACK); | 7. POP(STACK, TERM); |
| 3. repeat | 8. PRINT(TERM); |
| 4. if TERM <= 12 then | 9. TERM = 3 * TERM + 2; |
| PUSH(STACK, TERM); | 10. until EMPTY_STACK(STACK) |
| 5. TERM = 2 * TERM; | and TERM > 15. |

Solution: Let us keep track of the stack contents and the variable TERM as shown below:

Steps	stack STACK	TERM	Output displayed
1-2		3	
3, 4, 5, 10	3	6	
3, 4, 5, 10	3 6	12	
3, 4, 5, 10	3 6 12	24	
3, 6, 7	3 6	12	
8	3 6	12	12
9, 10	3 6	38	
3, 6, 7	3	6	
8	3	6	6
9, 10	3	20	
3, 6, 7		3	
8		3	3
9, 10		11	
3, 4, 5, 10	11	22	
3, 6, 7		11	
8		11	11
9, 10		35	

The output is 12, 6, 3, 11.

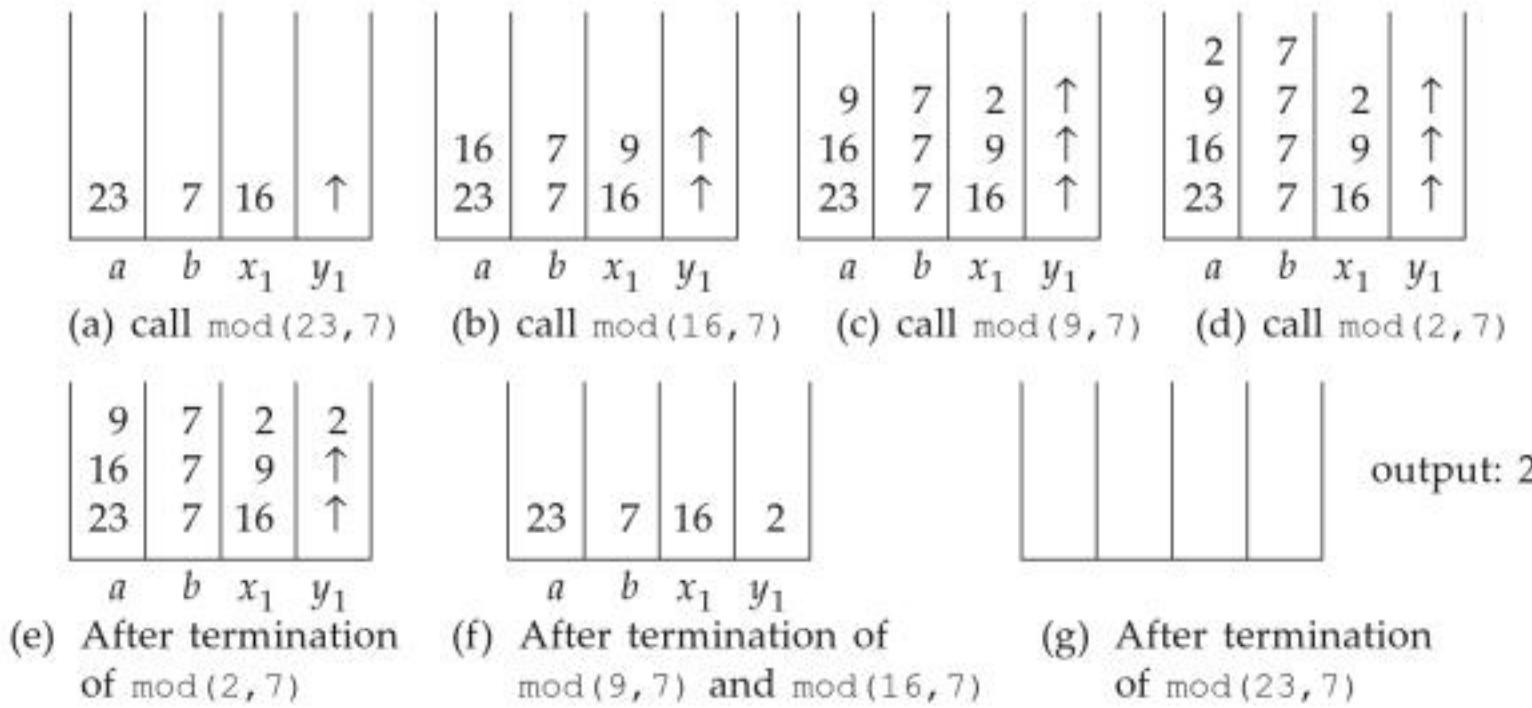
Problem 4.4 For the following pseudo code of a recursive program mod which computes $a \bmod b$ given a, b as inputs, trace the stack contents during the execution of the call mod(23, 7).

```

Procedure mod (a, b)
  if (a < b) then mod : = a
  else
    { x1 : = a - b
      y1 : = mod (x1, b)
      mod : = y1
    }
  end mod

```

Solution: We open a stack structure to track the variables a, b, x_1, y_1 as shown below. The snapshots of the stack during recursion are shown.



Problem 4.5 For the infix expression given below, obtain (i) the equivalent postfix expression, (ii) the equivalent prefix expression, and (iii) evaluate the equivalent postfix expression obtained in (i) using the algorithm EVAL_POSTFIX() (Algorithm 4.3), with $A = 1, B = 10, C = 1, D = 2, G = -1$ and $H = 6$.

Solution: (i), (ii): We demonstrate the steps to compute the prefix expression and postfix expression in parallel in the following table:

Expression	Sub-expression chosen based on rules of hierarchy, precedence and associativity	Equivalent Postfix expression	Equivalent Prefix expression
$-(\underbrace{A + B + C}_{\textcircled{1}})^{\uparrow} D)^{*} (G + H)$	$(A + B + C)$ [Note: $(A + B + C)$ is equivalent to the two subexpressions $\begin{array}{c} (A + B + C) \\ \xrightarrow{\textcircled{1}} \\ ((\textcircled{1}) + C) \end{array}$ $\textcircled{1}$	①: $AB + C+$	①: $++ ABC$

(Contd.)

(Contd.)

$(-\underbrace{\textcircled{1}}_{\textcircled{2}} \uparrow D)^* (G + H)$	$- \textcircled{1}$	$\textcircled{2}: AB + C + -$	$\textcircled{2}: -++ABC$
$(\underbrace{\textcircled{2}}_{\textcircled{3}} \uparrow D)^* (G + H)$	$(\textcircled{2} \uparrow D)$	$\textcircled{3}: AB + C + -D \uparrow$	$\textcircled{3}: \uparrow -++ABCD$
$\textcircled{3} * (\underbrace{G + H}_{\textcircled{4}})$	$(G + H)$	$\textcircled{4}: GH +$	$\textcircled{4}: +GH$
$\underbrace{\textcircled{3} * \textcircled{4}}_{\textcircled{5}}$	$\textcircled{3} * \textcircled{4}$	$\textcircled{5}: AB + C +$ $-D \uparrow GH + *$	$* \uparrow -++ABCD$ $+ GH$

The equivalent postfix and prefix expressions are $AB + C + -D \uparrow GH + *$ and $* \uparrow -++ABCD + GH$ respectively.

- (iii) To evaluate $AB + C + -D \uparrow GH + *$ \$ for $A = 1$, $B = 10$, $C = 1$, $D = 2$, $G = -1$ and $H = 6$, using Algorithm EVAL_POSTFIX(), the steps are listed in the following table:

x	Stack S	Action
A	$A(1)$	Push A into S
B	$A(1) B(10)$	Push B into S
$+$	11	Evaluate $A + B$ and push result into S
C	$11 C(1)$	Push C into S
$+$	12	Evaluate $11 + C$ and push result into S
$-^*$	-12	Evaluate (unary minus) -12 and push result into S
D	$-12 D(2)$	Push D into S
\uparrow	144	Evaluate $(-12) \uparrow D$ and push result into S
G	$144 G(-1)$	Push G into S
H	$144 G(-1) H(6)$	Push H into S

[#]: A compiler basically distinguishes between a unary “-” and a binary “-” by generating different tokens. Hence there is no ambiguity regarding the number of operands to be popped out from the stack when the operator is “-”. In the case of a unary “-” a single operand is popped out and in the case of binary “-”, two operands are popped out from the stack.

(Contd.)

+	144 5	Evaluate $G+H$ and push result into S
*	720	Evaluate $144 * 5$ and push result into S
\$		Output 720



Review Questions

1. Which among the following properties does not hold good in a stack?
 - (i) A stack supports the principle of Last In First Out
 - (ii) A push operation decrements the top pointer
 - (iii) A pop operation deletes an item from the stack
 - (iv) A linear stack has limited capacity
 - (a) (i)
 - (b) (ii)
 - (c) (iii)
 - (d) (iv)
 2. A linear stack S is implemented using an array as shown below. The TOP pointer which points to the top most element of the stack is set as shown.

X	Y	A	Z	F
[1] of ↑ stack	[2]	[3]	[4] ↑ TOP	[5]

Execution of the operation $\text{PUSH}(S, 'W')$ would result in



Programming Assignments

1. Implement a stack S of n elements using arrays. Write functions to perform PUSH and POP operations. Implement queries using the push and pop functions to

- (i) Retrieve the m^{th} element of the stack S from the top ($m < n$), leaving the stack without its top $m - 1$ elements
- (ii) Retain only the elements in the odd position of the stack and pop out all even positioned elements.

(e.g.)

Stack S Output stack S Elements:

a	b	c	d
-----	-----	-----	-----

a	c	
-----	-----	--

Position: 1 2 3 4

1 2

2. Write a recursive program to obtain the n^{th} order Fibonacci sequence number. Include appropriate input / output statements to track the variables participating in recursion. Do you observe the 'invisible' stack at work? Record your observations.

3. Implement a program to evaluate any given postfix expression. Test your program for the evaluation of the equivalent postfix form of the expression $\frac{-(A * B)}{D} \uparrow C + E - F * H * I$ for $A = 1, B = 2, D = 3, C = 14, E = 110, F = 220, H = 16.78, I = 364.621$.



QUEUES

5

In this chapter, we discuss the queue data structure, its operations and its variants viz, circular queues, priority queues and deques. The application of the data structure is demonstrated on the problem of job scheduling in a time sharing system environment.

Introduction

5.1

- 5.1 Introduction
- 5.2 Operations on Queues
- 5.3 Circular Queues
- 5.4 Other types of Queues
- 5.5 Applications

A **Queue** is a linear list in which all insertions are made at one end of the list known as *rear* or *tail* of the queue and all deletions are made at the other end known as *front* or *head* of the queue. An insertion operation is also referred to as *enqueueing a queue* and a deletion operation is referred to as *dequeuing a queue*.

Figure 5.1 illustrates a queue and its functionality. Here, Q is a queue of three elements a, b, c (Fig. 5.1(a)). When an element d is to join the queue, it is inserted at the rear end of the queue (Fig. 5.1(b)) and when an element is to be deleted, the one at the front end of the queue, viz, a , is deleted automatically (Fig. 5.1(c)). Thus a queue data structure obeys the principle of *first in first out* (FIFO) or *first come first served* (FCFS).

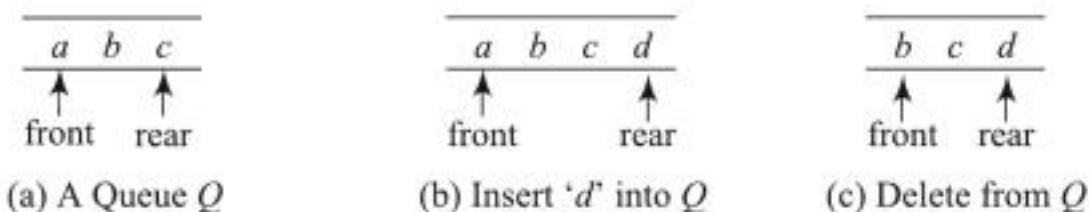


Fig. 5.1 A queue and its functionality

Many examples of queues occur in everyday life. Figure 5.2(a) illustrates a queue of clients awaiting to be served by a clerk in a booking counter and Fig. 5.2(b) illustrates a trail of components moving down an assembly line to be processed by a robot at the end of the line. The FIFO principle of insertion at the rear end of the queue when a new client arrives or when a new component is added, and deletion at the front end of the queue when the service of the client or processing of the component is complete is evident.

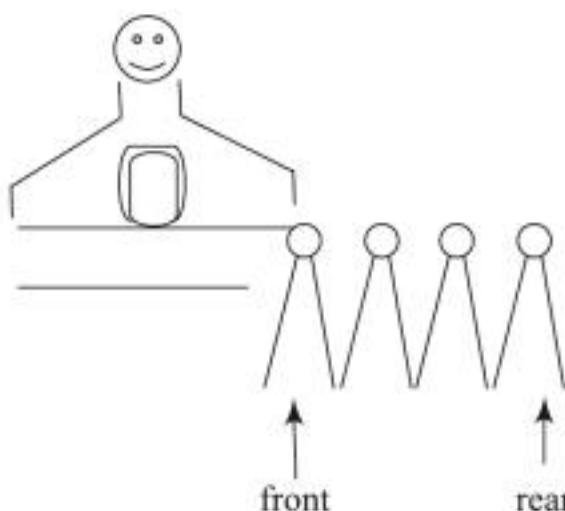
Operations on Queues

5.2

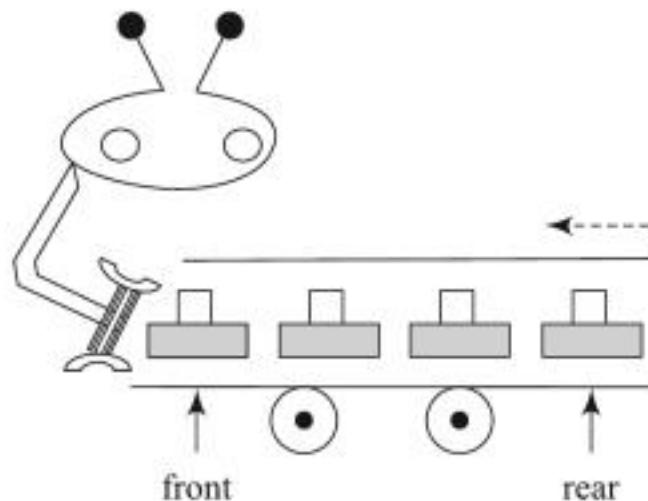
The queue data structure supports two operations, viz.,

- Insertion or addition of elements to a queue
- Deletion or removal of elements from a queue

Before we proceed to discuss these operations, it is essential to know how queues are implemented.



(a) Queue before a booking counter



(b) Queue of components in an assembly line

Fig. 5.2 Common examples of queues

Queue Implementation

As discussed for stacks, a common method of implementing a queue data structure is to use another sequential data structure, viz, arrays. However, queues have also been implemented using a linked data structure (Refer Chapter 7). In this chapter, we confine our discussion to the implementation of queues using arrays.

Figure 5.3 illustrates an array based implementation of a queue. A queue Q of four elements R, S, V, G is represented using an array $Q[1:7]$. Note how the variables FRONT and REAR keep track of the front and rear ends of the queue to facilitate execution of insertion and deletion operations respectively.

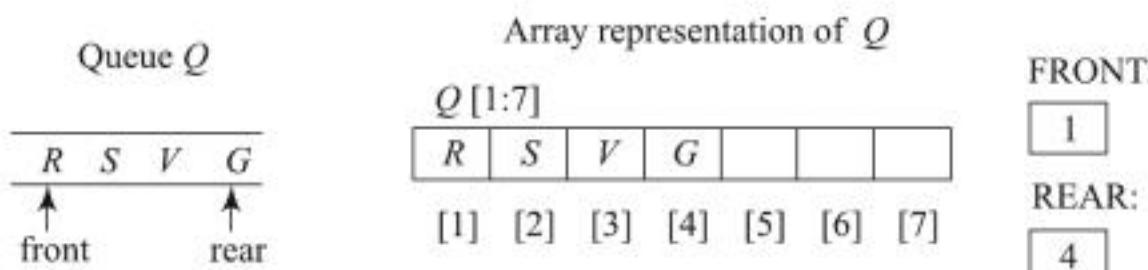


Fig. 5.3 Array implementation of a queue

However, just as in the stack data structure, the array implementation puts a limitation on the capacity of the queue. In other words, the number of elements in the queue cannot exceed the maximum dimension of the one dimensional array. Thus a queue that is accommodated in an array $Q[1 : n]$, cannot hold more than n elements. Hence every insertion of an element into the queue has to necessarily test for a QUEUE-FULL condition before executing the insertion

operation. Again, each deletion has to ensure that it is not attempted on a queue which is already empty calling for the need to test for a QUEUE-EMPTY condition before executing the deletion operation. But as said earlier with regard to stacks, the linked representation of queues dispenses with the need for these QUEUE-FULL and QUEUE-EMPTY testing conditions and hence prove to be elegant and efficient.

Implementation of insert and delete operations on a queue

Let $Q[1 : n]$ be an array implementation of a queue. Let `FRONT` and `REAR` be variables recording the front and rear positions of the queue. The `FRONT` variable points to a position which is physically one less than the actual front of the queue. `ITEM` is the element to be inserted into the queue. n is the maximum capacity of the queue. Both `FRONT` and `REAR` are initialized to 0.

Algorithm 5.1 illustrates the insert operation on a queue.

Algorithm 5.1: Implementation of an insert operation on a queue

```
Procedure INSERTO ( $O$ ,  $n$ , ITEM, REAR)
    /* insert item ITEM into  $O$  with capacity  $n$  */
    if (REAR =  $n$ ) then QUEUE_FULL;
    REAR = REAR + 1; /* Increment REAR */
     $O[\text{REAR}]$  = ITEM; /* Insert ITEM as the rear element*/
end INSERTO
```

It can be observed in Algorithm 5.1 that addition of every new element into the queue increments the `REAR` variable. However, before insertion, the condition whether the queue is full (QUEUE_FULL) is checked. This ensures that there is no overflow of elements in a queue.

The delete operation is illustrated in Algorithm 5.2. Though a deletion operation automatically deletes the front element of the queue, the variable `ITEM` is used as an output variable to store and perhaps display the value of the element removed.

Algorithm 5.2: Implementation of a delete operation on a queue

```
Procedure DELETEO ( $O$ , FRONT, REAR, ITEM)
    if (FRONT = REAR) then QUEUE_EMPTY;
    FRONT = FRONT + 1;
    ITEM =  $O[\text{FRONT}]$ ;
end DELETEO.
```

In Algorithm 5.2, observe that to perform a delete operation, the participation of both the variables `FRONT` and `REAR` is essential. Before deletion, the condition (`FRONT` = `REAR`) checks for the emptiness of the queue. If the queue is not empty, `FRONT` is incremented by 1 to point to the element to be deleted and subsequently the element is removed through `ITEM`. Note how this leaves the `FRONT` variable remembering the position which is one less than the actual front of the queue. This helps in the usage of (`FRONT` = `REAR`) as a common condition for testing whether a queue is empty, which occurs either after its initialization or after a sequence of insert and delete operations, when the queue has just emptied itself.

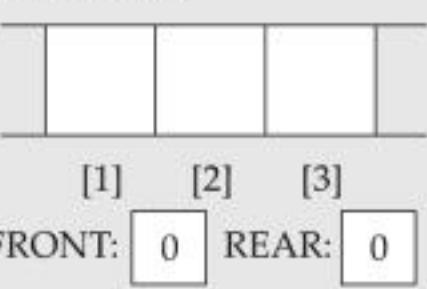
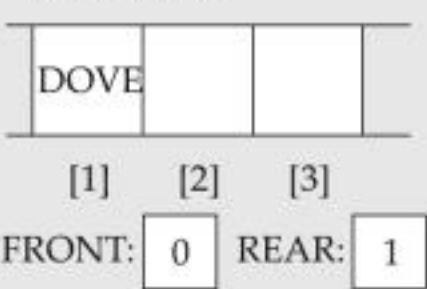
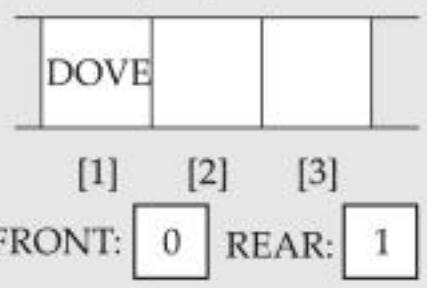
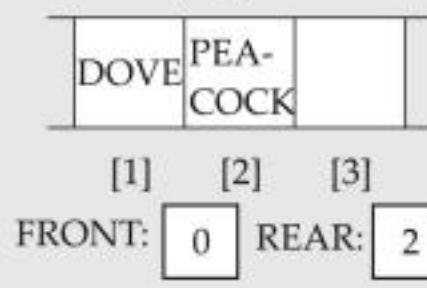
Soon after the queue Q has been initialized, $\text{FRONT} = \text{REAR} = 0$. Hence the condition ($\text{FRONT} = \text{REAR}$) ensures that the queue is empty. Again after a sequence of operations when Q has become partially or completely full and delete operations are repeatedly invoked to empty the queue, it may be observed how FRONT increments itself in steps of one with every deletion and begins moving towards REAR . During the final deletion which renders the queue empty, FRONT coincides with REAR satisfying the condition ($\text{FRONT} = \text{REAR} = k$), $k \neq 0$. Here k is the position of the last element to be deleted.

Hence, we observe that in an array implementation of queues, with every insertion, REAR moves away from FRONT and with every deletion FRONT moves towards REAR . When the queue is empty, $\text{FRONT} = \text{REAR}$ is satisfied and when full, $\text{REAR} = n$ (the maximum capacity of the queue) is satisfied.

Queues whose insert/delete operations follow the procedures implemented in Algorithms 5.1 and 5.2, are known as **linear queues** to distinguish them from **circular queues** which will be discussed in Sec. 5.3. Example 5.1 demonstrates the working of a linear queue. The time complexity to perform a single insert/delete operation in a linear queue is $O(1)$.

Example 5.1 Let BIRDS [1:3] be a linear queue data structure. The working of Algorithms 5.1 and 5.2 demonstrated on the insertions and deletions performed on BIRDS is illustrated in Table 5.1.

Table 5.1 Insert/delete operations on the queue BIRDS [1:3]

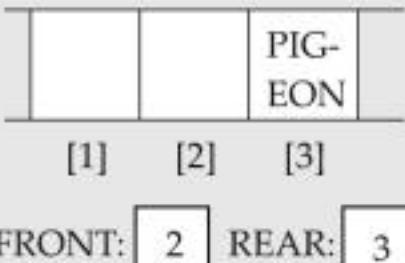
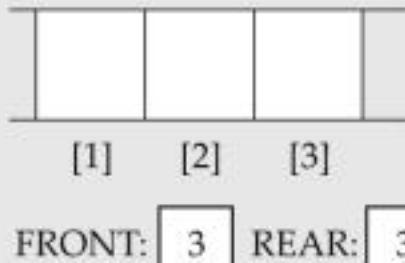
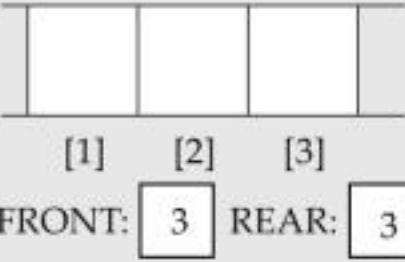
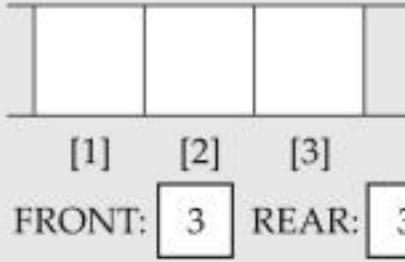
Operation	Queue before operation	Algorithm	Queue after operation	Remarks
1. Insert 'DOVE' into BIRDS [1:3]	BIRDS [1:3]  FRONT: 0 REAR: 0	INSERTQ (BIRDS 3, 'DOVE', 0)	BIRDS [1:3]  FRONT: 0 REAR: 1	Insert 'DOVE' successful
2. Insert 'PEACOCK' into BIRDS [1:3]	BIRDS [1:3]  FRONT: 0 REAR: 1	INSERTQ (BIRDS, 3, 'PEACOCK', 1)	BIRDS [1:3]  FRONT: 0 REAR: 2	Insert 'PEACOCK' successful

(Contd.)

(Contd.)

3. Insert 'PIGEON' in to BIRDS [1:3]	<p>BIRDS [1:3]</p> <table border="1" data-bbox="443 264 892 422"> <tr><td>DOVE</td><td>PEA-COCK</td><td></td></tr> <tr><td>[1]</td><td>[2]</td><td>[3]</td></tr> </table> <p>FRONT: 0 REAR: 2</p>	DOVE	PEA-COCK		[1]	[2]	[3]	INSERTQ (BIRDS, 3, 'PIGEON', 2)	<p>BIRDS [1:3]</p> <table border="1" data-bbox="1212 264 1660 422"> <tr><td>DOVE</td><td>PEA-COCK</td><td>PIG-EON</td></tr> <tr><td>[1]</td><td>[2]</td><td>[3]</td></tr> </table> <p>FRONT: 0 REAR: 3</p>	DOVE	PEA-COCK	PIG-EON	[1]	[2]	[3]	Insert 'PIGEON' successful
DOVE	PEA-COCK															
[1]	[2]	[3]														
DOVE	PEA-COCK	PIG-EON														
[1]	[2]	[3]														
4. Insert 'SWAN' in to BIRDS [1:3]	<p>BIRDS [1:3]</p> <table border="1" data-bbox="443 673 892 831"> <tr><td>DOVE</td><td>PEA-COCK</td><td>PIG-EON</td></tr> <tr><td>[1]</td><td>[2]</td><td>[3]</td></tr> </table> <p>FRONT: 0 REAR: 3</p>	DOVE	PEA-COCK	PIG-EON	[1]	[2]	[3]	INSERTQ (BIRDS, 3, 'SWAN', 3)	<p>BIRDS [1:3]</p> <table border="1" data-bbox="1212 673 1660 831"> <tr><td>DOVE</td><td>PEA-COCK</td><td>PIG-EON</td></tr> <tr><td>[1]</td><td>[2]</td><td>[3]</td></tr> </table> <p>FRONT: 0 REAR: 3</p>	DOVE	PEA-COCK	PIG-EON	[1]	[2]	[3]	Insert 'SWAN' failure! QUEUE_FULL condition invoked.
DOVE	PEA-COCK	PIG-EON														
[1]	[2]	[3]														
DOVE	PEA-COCK	PIG-EON														
[1]	[2]	[3]														
5. Delete	<p>BIRDS [1:3]</p> <table border="1" data-bbox="443 1075 892 1232"> <tr><td>DOVE</td><td>PEA-COCK</td><td>PIG-EON</td></tr> <tr><td>[1]</td><td>[2]</td><td>[3]</td></tr> </table> <p>FRONT: 0 REAR: 3</p>	DOVE	PEA-COCK	PIG-EON	[1]	[2]	[3]	DELETEQ (BIRDS, 0, 3, ITEM)	<p>BIRDS [1:3]</p> <table border="1" data-bbox="1212 1075 1660 1232"> <tr><td></td><td>PEA-COCK</td><td>PIG-EON</td></tr> <tr><td>[1]</td><td>[2]</td><td>[3]</td></tr> </table> <p>FRONT: 1 REAR: 3</p>		PEA-COCK	PIG-EON	[1]	[2]	[3]	Delete successful. <i>ITEM</i> =DOVE
DOVE	PEA-COCK	PIG-EON														
[1]	[2]	[3]														
	PEA-COCK	PIG-EON														
[1]	[2]	[3]														
6. Delete	<p>BIRDS [1:3]</p> <table border="1" data-bbox="443 1510 892 1667"> <tr><td></td><td>PEA-COCK</td><td>PIG-EON</td></tr> <tr><td>[1]</td><td>[2]</td><td>[3]</td></tr> </table> <p>FRONT: 1 REAR: 3</p>		PEA-COCK	PIG-EON	[1]	[2]	[3]	DELETEQ (BIRDS, 0, 3, <i>ITEM</i>)	<p>BIRDS [1:3]</p> <table border="1" data-bbox="1212 1510 1660 1667"> <tr><td></td><td></td><td>PIG-EON</td></tr> <tr><td>[1]</td><td>[2]</td><td>[3]</td></tr> </table> <p>FRONT: 2 REAR: 3</p>			PIG-EON	[1]	[2]	[3]	Delete successful. <i>ITEM</i> =PEACOCK
	PEA-COCK	PIG-EON														
[1]	[2]	[3]														
		PIG-EON														
[1]	[2]	[3]														
7. Insert 'SWAN' in to BIRDS [1:3]	<p>BIRDS [1:3]</p> <table border="1" data-bbox="443 1903 892 2060"> <tr><td></td><td></td><td>PIG-EON</td></tr> <tr><td>[1]</td><td>[2]</td><td>[3]</td></tr> </table> <p>FRONT: 2 REAR: 3</p>			PIG-EON	[1]	[2]	[3]	INSERTQ(BIRDS, 3, 'SWAN', 3)	<p>BIRDS [1:3]</p> <table border="1" data-bbox="1212 1903 1660 2060"> <tr><td></td><td></td><td>PIG-EON</td></tr> <tr><td>[1]</td><td>[2]</td><td>[3]</td></tr> </table> <p>FRONT: 2 REAR: 3</p>			PIG-EON	[1]	[2]	[3]	Insert 'SWAN' failure! QUEUE_FULL condition invoked.
		PIG-EON														
[1]	[2]	[3]														
		PIG-EON														
[1]	[2]	[3]														

(Contd.)

8. Delete	BIRDS [1:3]  FRONT: 2 REAR: 3	DELETEQ (BIRDS, 2, 3, <i>ITEM</i>)	BIRDS [1:3]  FRONT: 3 REAR: 3	Delete successful. <i>ITEM</i> = PIGEON
9. Delete	BIRDS [1:3]  FRONT: 3 REAR: 3	DELETEQ (BIRDS, 3, 3, <i>ITEM</i>)	BIRDS [1:3]  FRONT: 3 REAR: 3	QUEUE_EMPTY condition invoked.

invocation

Limitations of linear queues

Example 5.1 illustrated the implementation of insert and delete operations on a linear queue. In operation 4 when 'SWAN' was inserted into BIRDS [1:3], the insertion operation was unsuccessful since the QUEUE_FULL condition was invoked. Also, one observes the queue BIRDS to be physically full justifying the condition. But after operations 5 and 6 were performed and when two elements viz., DOVE and PEACOCK were deleted, despite the space it had created to accommodate two more insertions, the insertion of 'SWAN' attempted in operation 7 was rejected once again due to the invocation of the QUEUE_FULL condition. This is a gross limitation of a linear queue since QUEUE_FULL condition does not check whether Q is 'physically' full. It merely relies on the condition (REAR = *n*) which may turn out to be true even for a queue that is only partially full as shown in operation 7 of Example 5.1.

When one contrasts this implementation with the working of a queue that one sees around in every day life, it is easy to see that with every deletion (after completion of service at one end of the queue) the remaining elements move forward towards the head of the queue leaving no gaps in-between. This obviously makes room for that many insertions to be accommodated at the tail end of the queue depending on the space available.

However, to attempt implementing this strategy during every deletion of an element is worthless since data movement is always computationally expensive and may render the process of queue maintenance highly inefficient.

In short, when a QUEUE_FULL condition is invoked it does not necessarily imply that the queue is 'physically' full. This leads to the limitation of rejecting insertions despite the space available to accommodate them. The rectification of this limitation leads to what are known as circular queues.

Circular Queues

5.3

In this section we discuss the implementation and operations on circular queues which serve to rectify the limitation of linear queues.

As the name indicates a circular queue is not linear in structure but instead it is circular. In other words, the FRONT and REAR variables which displayed a linear (left to right) movement over a queue, display a circular movement (clock wise) over the queue data structure.

Operations on a circular queue

Let CIRC_Q be a circular queue with a capacity of three elements as shown in Fig. 5.4(a). The queue is obviously full with FRONT pointing to the element at the head of the queue and REAR pointing to the element at the tail end of the queue. Let us now perform two deletions and then attempt insertions of 'd' and 'e' into the queue.

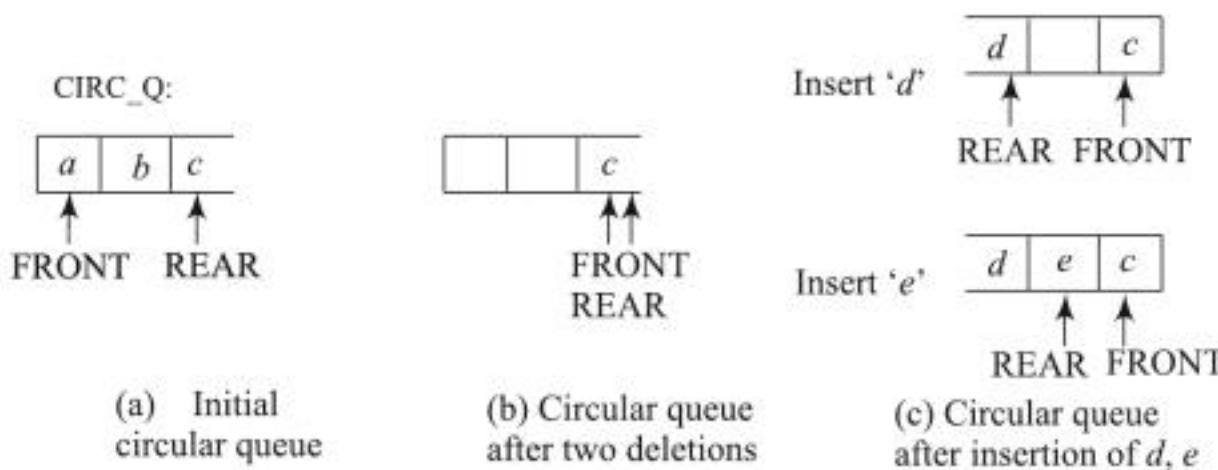


Fig. 5.4 Working of a circular queue

Observe the circular movement of the FRONT and REAR variables. After two deletions, FRONT moves towards REAR and points to 'c' as the current front element of CIRC_Q (Fig. 5.4(b)). When 'd' is inserted, unlike linear queues, REAR curls back in a clock wise fashion to accommodate 'd' in the vacant space available. A similar procedure follows for the insertion of 'e' as well (Fig. 5.4(c)).

Figure 5.5 emphasizes this circular movement of FRONT and REAR variables over a general circular queue during a sequence of insertions/deletions.

A circular queue when implemented using arrays is not different from linear queues in their physical storage. In other words, a linear queue is conceptually viewed to have a circular form to understand the clockwise movement of FRONT and REAR variables as shown in Fig. 5.6.

Implementation of insertion and deletion operations in a circular queue

Algorithms 5.3 and 5.4 illustrate the implementation of insert and delete operations in a circular queue respectively. The circular movement of FRONT and REAR variables is implemented using the **mod** function which is cyclical in nature. Also the array data structure CIRC_Q to implement the queue is declared to be CIRC_Q [0: n - 1] to facilitate the circular operation of FRONT and REAR variables. As in linear queues, FRONT points to a position which is one less than the actual front of the circular queue. Both FRONT and REAR are initialized to 0. Note that (n - 1) is the actual physical capacity of the queue in spite of the array declaration as [0 : n - 1]

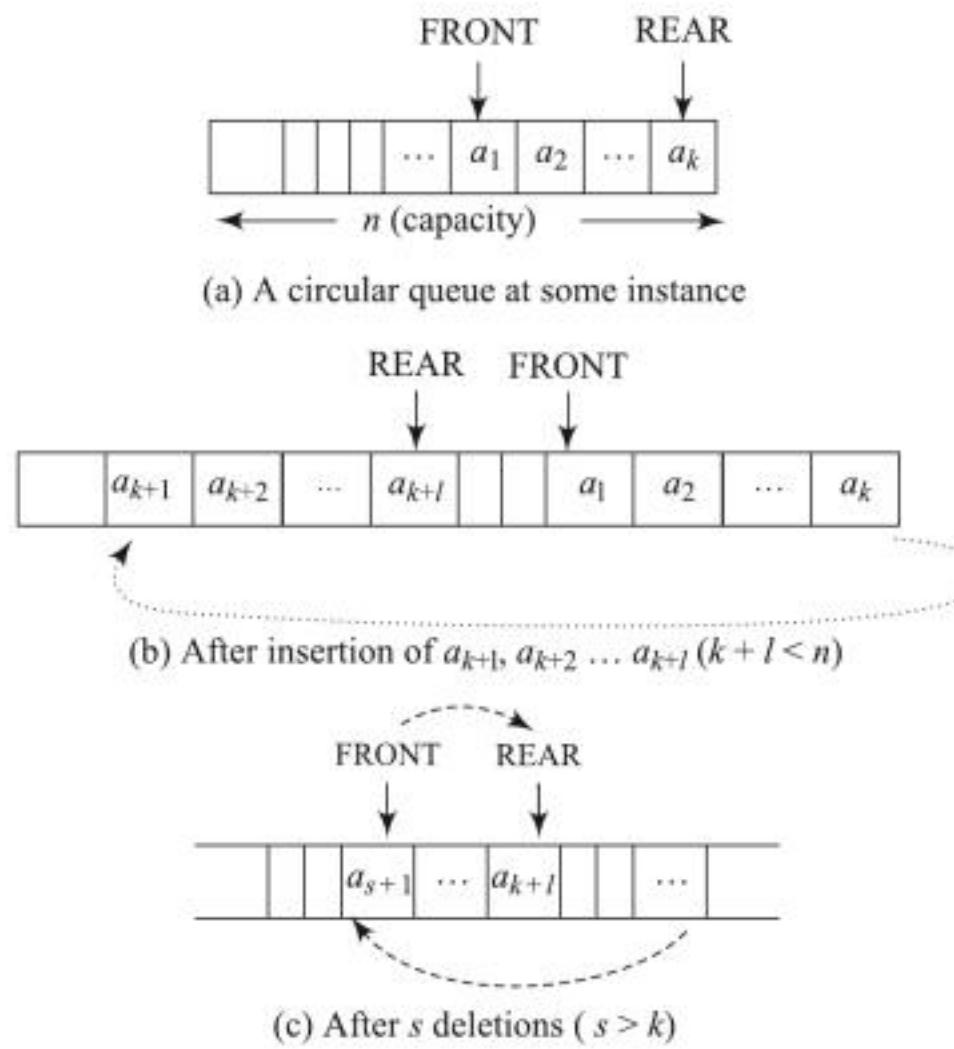


Fig. 5.5 Circular movement of FRONT and REAR variables in a circular queue

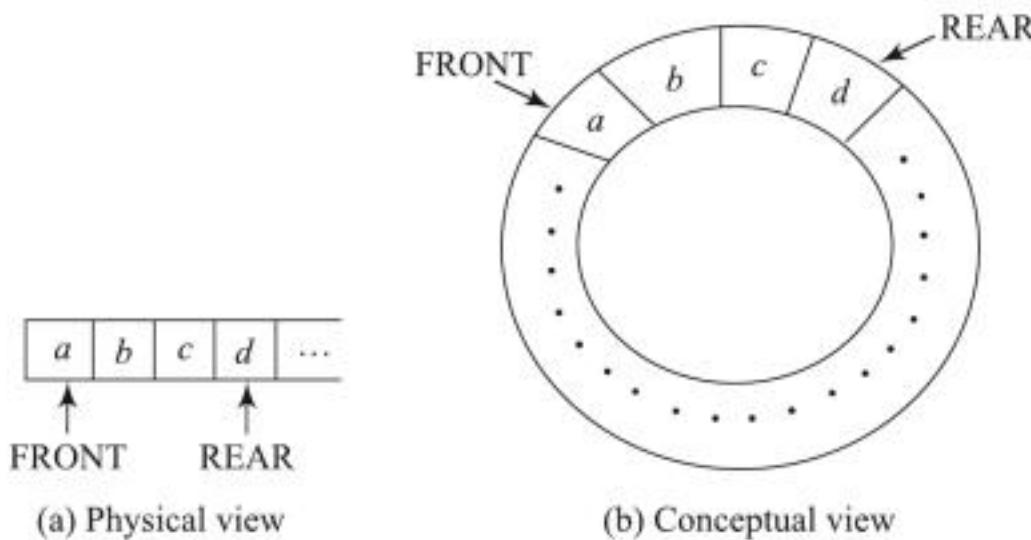


Fig. 5.6 Physical and conceptual view of a circular queue

Algorithm 5.3: Implementation of insert operation on a circular queue

```

Procedure INSERT_CIRCO(CIRC_O, FRONT, REAR, n, ITEM)
  REAR = (REAR + 1) mod n;
  If (FRONT = REAR) then CIRCO_FULL; /* Here CIRCO_FULL tests for the
                                             queue full condition and if so,
                                             retracts REAR to its
                                             previous value*/
  CIRC_O [REAR] = ITEM;
end INSERT_CIRCO.
```



Algorithm 5.4: Implementation of a delete operation on a circular queue

```

procedure DELETE_CIRCO(CIRC_O, FRONT, REAR, n, ITEM)
If (FRONT = REAR) then CIRCO_EMPTY; /* CIRC_O is physically empty*/
FRONT = (FRONT+1) mod n;
ITEM = CIRC_O [FRONT];
end DELETE_CIRCO

```



The time complexities of Algorithms 5.3 and 5.4 is $O(1)$. The working of the algorithms is demonstrated on an illustration given in Example 5.2.

Example 5.2 Let COLOURS [0:3] be a circular queue data structure. Note the actual physical capacity of the queue is only 3 elements despite the declaration of the array as [0:3]. The operations illustrated below (Table 5.2) demonstrate the working of Algorithms 5.3 and 5.4.

Table 5.2 Insert and delete operations on the circular queue COLOURS [0:3]

Circular Queue operation	Circular queue before operation	Algorithm Invocation	Circular queue after operation	Remarks
1. Insert 'ORANGE' into COLOURS [0:3]	COLOURS [0: 3] FRONT: <input type="text" value="0"/> REAR: <input type="text" value="0"/>	INSERT_CIRCQ(COLOURS, 0, 0, 4, 'ORANGE')	COLOURS [0:3] FRONT: <input type="text" value="0"/> REAR: <input type="text" value="1"/>	Insert 'ORANGE' successful
2. Insert 'BLUE' into COLOURS [0:3]	COLOURS [0:3] FRONT: <input type="text" value="0"/> REAR: <input type="text" value="1"/>	INSERT_CIRCQ(COLOURS, 0, 1, 4, 'BLUE')	COLOURS [0:3] FRONT: <input type="text" value="0"/> REAR: <input type="text" value="2"/>	Insert 'BLUE' successful
3. Insert 'WHITE' into COLOURS [0:3]	COLOURS [0:3] FRONT: <input type="text" value="0"/> REAR: <input type="text" value="2"/>	INSERT_CIRCQ(COLOURS, 0, 2, 4, 'WHITE')	COLOURS [0:3] FRONT: <input type="text" value="0"/> REAR: <input type="text" value="3"/>	Insert 'WHITE' successful

(Contd.)

(Contd.)

4. Insert 'RED' into COLOURS [0:3]	<p>COLOURS [0:3]</p> <table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td></td><td>ORA NGE</td><td>BLUE</td><td>WHI TE</td></tr> <tr> <td>[0]</td><td>[1]</td><td>[2]</td><td>[3]</td></tr> </table> <p>FRONT: 0 REAR: 3</p>		ORA NGE	BLUE	WHI TE	[0]	[1]	[2]	[3]	INSERT_CIRCQ(COLOURS, 0, 3, 4, 'RED')	<p>COLOURS [0:3]</p> <table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td></td><td>ORA NGE</td><td>BLUE</td><td>WHI TE</td></tr> <tr> <td>[0]</td><td>[1]</td><td>[2]</td><td>[3]</td></tr> </table> <p>FRONT: 0 REAR: 3</p>		ORA NGE	BLUE	WHI TE	[0]	[1]	[2]	[3]	CIRCQ_FULL condition is invoked. Insert 'RED' failure! Note: REAR retracts to its previous value of 3.
	ORA NGE	BLUE	WHI TE																	
[0]	[1]	[2]	[3]																	
	ORA NGE	BLUE	WHI TE																	
[0]	[1]	[2]	[3]																	
5, 6. Delete twice from COLOURS [0:3]	<p>COLOURS [0:3]</p> <table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td></td><td>ORA NGE</td><td>BLUE</td><td>WHI TE</td></tr> <tr> <td>[0]</td><td>[1]</td><td>[2]</td><td>[3]</td></tr> </table> <p>FRONT: 0 REAR: 3</p>		ORA NGE	BLUE	WHI TE	[0]	[1]	[2]	[3]	DELETE_CIRCQ(COLOURS, 0, 3, 4, 'ITEM') DELETE_CIRCQ(COLOURS, 1, 3, 4, ITEMS)	<p>COLOURS [0:3]</p> <table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td></td><td></td><td></td><td>WHI TE</td></tr> <tr> <td>[0]</td><td>[1]</td><td>[2]</td><td>[3]</td></tr> </table> <p>FRONT: 2 REAR: 3</p>				WHI TE	[0]	[1]	[2]	[3]	DELETE operation successful ITEM = ORANGE ITEM = BLUE
	ORA NGE	BLUE	WHI TE																	
[0]	[1]	[2]	[3]																	
			WHI TE																	
[0]	[1]	[2]	[3]																	
7. Insert 'YELLOW' into COLOURS [0:3]	<p>COLOURS [0:3]</p> <table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td></td><td></td><td></td><td>WHI TE</td></tr> <tr> <td>[0]</td><td>[1]</td><td>[2]</td><td>[3]</td></tr> </table> <p>FRONT: 2 REAR: 3</p>				WHI TE	[0]	[1]	[2]	[3]	INSERT_CIRCQ(COLOURS, 2, 3, 4, 'YELLOW')	<p>COLOURS [0:3]</p> <table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>YEL LOW</td><td></td><td></td><td>WHI TE</td></tr> <tr> <td>[0]</td><td>[1]</td><td>[2]</td><td>[3]</td></tr> </table> <p>FRONT: 2 REAR: 0</p>	YEL LOW			WHI TE	[0]	[1]	[2]	[3]	Insert 'YELLOW' successful
			WHI TE																	
[0]	[1]	[2]	[3]																	
YEL LOW			WHI TE																	
[0]	[1]	[2]	[3]																	
8. Insert 'VIOLET' into COLOURS [0:3]	<p>COLOURS [0:3]</p> <table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>YEL LOW</td><td></td><td></td><td>WHI TE</td></tr> <tr> <td>[0]</td><td>[1]</td><td>[2]</td><td>[3]</td></tr> </table> <p>FRONT: 2 REAR: 0</p>	YEL LOW			WHI TE	[0]	[1]	[2]	[3]	INSERT_CIRCQ(COLOURS, 2, 0, 4, 'VIOLET')	<p>COLOURS [0:3]</p> <table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>YEL LOW</td><td>VIO LET</td><td></td><td>WHI TE</td></tr> <tr> <td>[0]</td><td>[1]</td><td>[2]</td><td>[3]</td></tr> </table> <p>FRONT: 2 REAR: 1</p>	YEL LOW	VIO LET		WHI TE	[0]	[1]	[2]	[3]	Insert 'VIOLET' successful
YEL LOW			WHI TE																	
[0]	[1]	[2]	[3]																	
YEL LOW	VIO LET		WHI TE																	
[0]	[1]	[2]	[3]																	

Other Types of Queues

5.4

Priority queues

A **priority queue** is a queue in which insertion or deletion of items from any position in the queue are done based on some property (such as **priority** of task)

For example, let P be a priority queue with three elements a, b, c whose priority factors are 2, 1, 1 respectively. Here, larger the number, higher is the priority accorded to that element (Fig. 5.7 (a)). When a new element d with higher priority viz., 4 is inserted, d joins at the head of the queue superceding the remaining elements (Fig. 5.7(b)). When elements in the queue have the same priority, then the priority queue behaves as an ordinary queue following the principle of FIFO amongst such elements.

The working of a priority queue may be likened to a situation when a file of patients wait for their turn in a queue to have an appointment with a doctor. All patients are accorded equal priority and follow an FCFS scheme by appointments. However, when a patient with bleeding injuries is brought in, he/ she is accorded high priority and is immediately moved to the head of the queue for immediate attention by the doctor. This is priority queue at work.

A common method of implementation of a priority queue is to open as many queues as there are priority factors. A low priority queue will be operated for deletion only when all its high priority predecessors are empty. In other words, deletion of an element in a priority queue q_1 with priority p_i is possible only when those queues q_j with priorities p_j ($p_j > p_i$) are empty. However, with regard to insertions, an element e_k with priority p_l joins the respective queue obeying the scheme of FIFO with regard to the queue q_l alone.

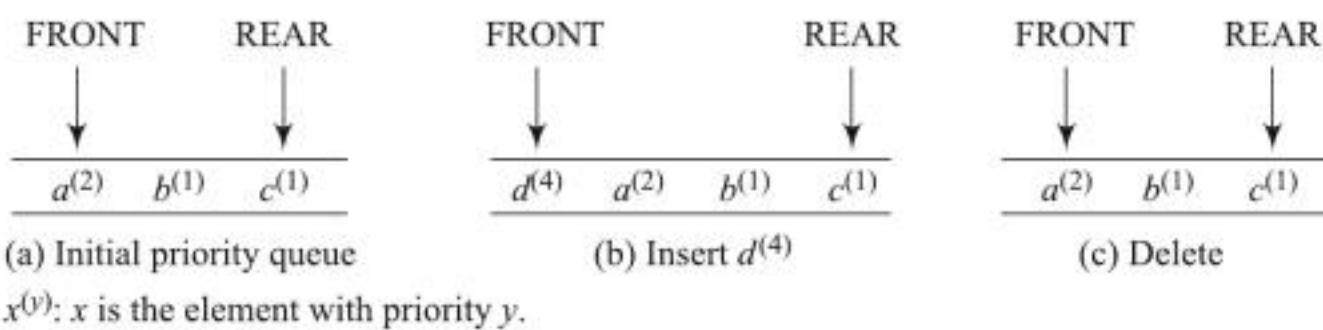


Fig. 5.7 A priority queue

Another method of implementation could be to sortout the elements in the queue according to the descending order of priorities every time an insertion takes place. The top priority element at the head of the queue is the one to be deleted.

The choice of implementation depends on a time-space trade off based decision made by the user. While the first method of implementation of a priority queue using a cluster of queue consumes space, the time complexity of an insertion is only $O(1)$. In the case of deletion of an element in a specific queue with a specific priority, it calls for the checking of all other queues preceding it in priority, to be empty.

On the other hand, the second method consumes less space since it handles just a single queue. However, insertion of every element calls for sorting all the queue elements in the descending order, the most efficient of which reports a time complexity of $O(n \log n)$. With regard to deletion, the element at the head of the queue is automatically deleted with a time complexity of $O(1)$.

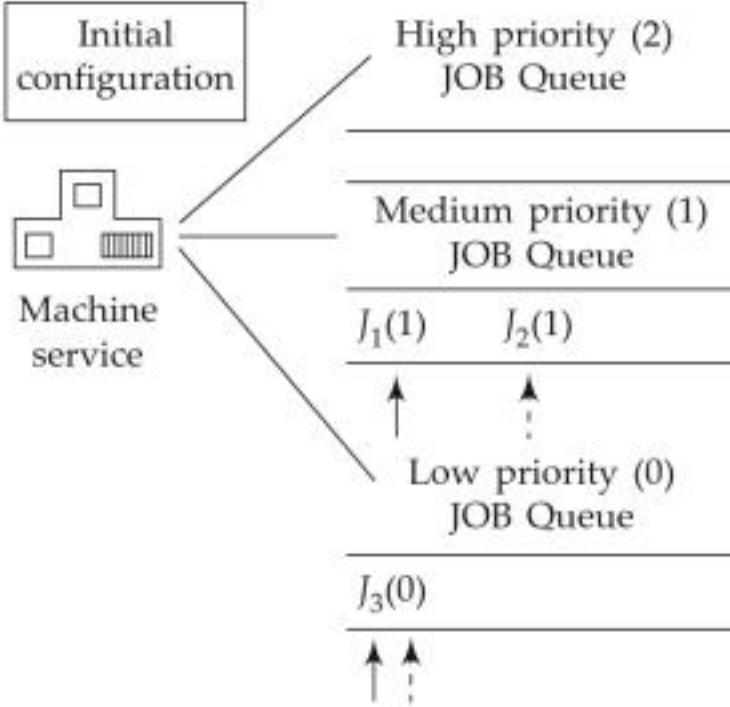
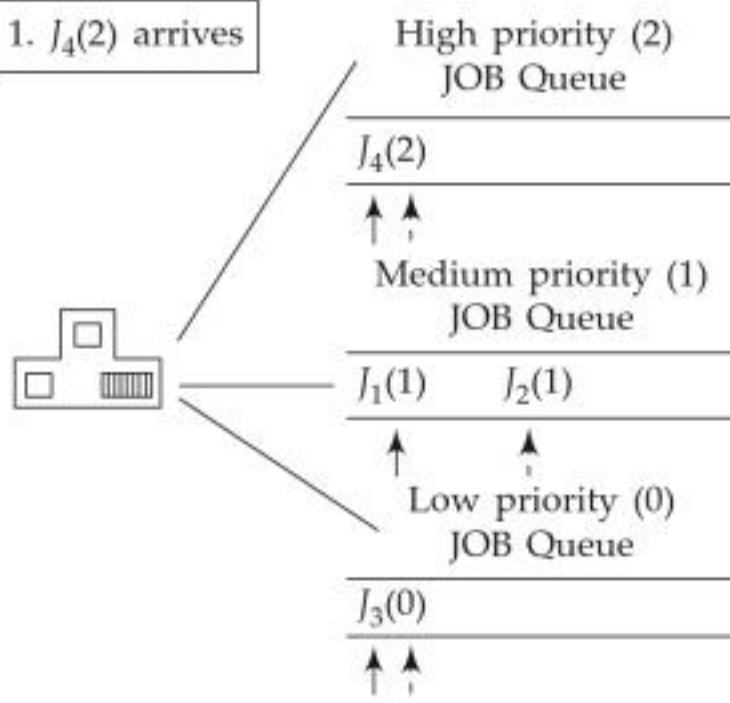
The two methods of implementation of a priority queue are illustrated in Example 5.3.

Example 5.3 Let JOB be a queue of jobs to be undertaken at a factory shop floor for service by a machine. Let high (2), medium (1) and low (0) be the priorities accorded to jobs. Let $J_i(k)$ indicate a job J_i to be undertaken with priority k . The implementations of a priority queue to keep track of the jobs, using the two methods of implementation discussed above, are illustrated for a sample set of job arrivals (insertions) and job service completion (deletion).

Opening JOB queue: $J_1(1) \quad J_2(1) \quad J_3(0)$

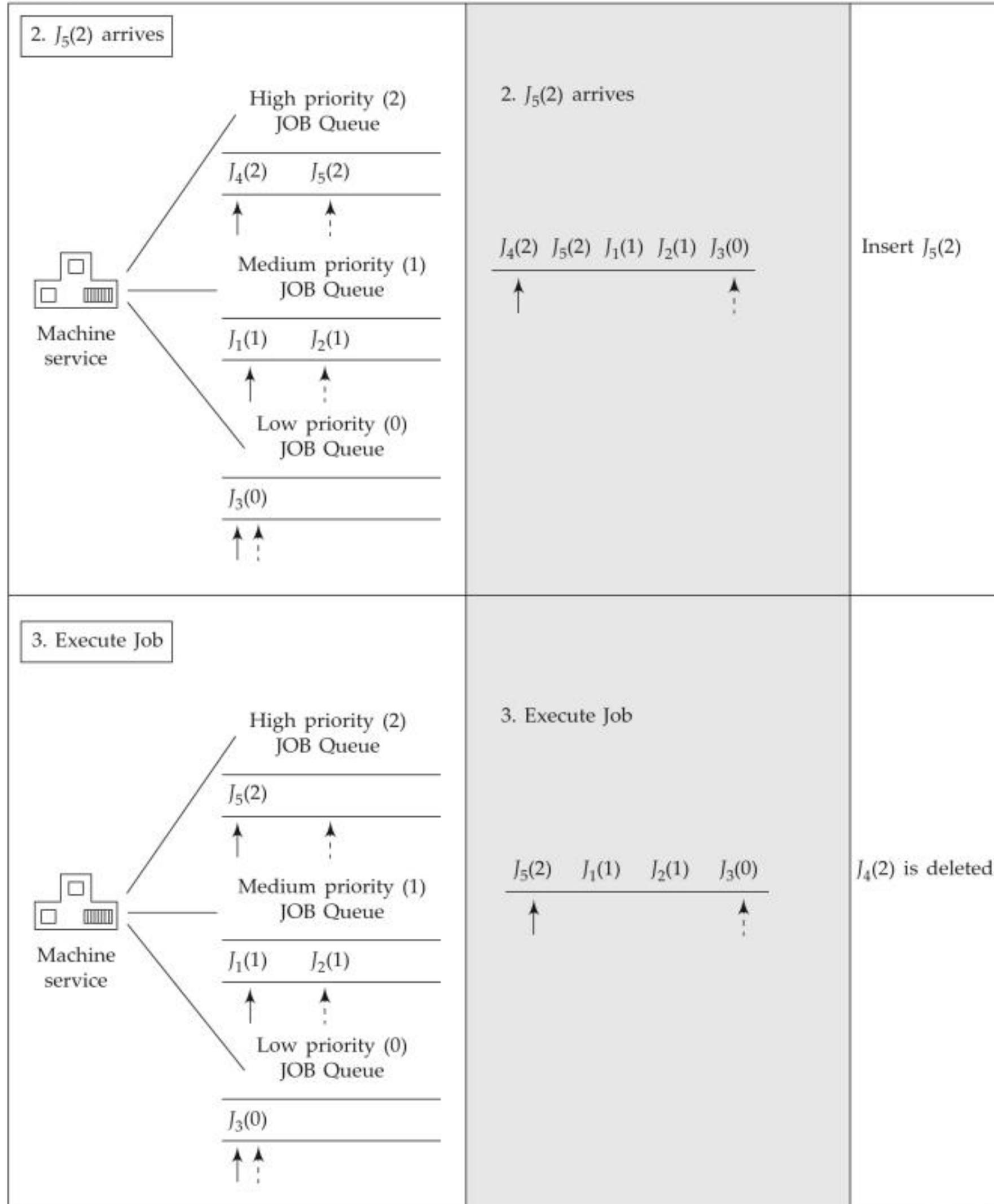
Operations on the JOB queue in the chronological order :

1. $J_4(2)$ arrives
2. $J_5(2)$ arrives
3. Execute job
4. Execute job
5. Execute job

Implementation of a priority queue as a cluster of queues	Implementation of a priority queue by sorting queue elements	Remarks
 <p>Initial configuration</p> <p>Machine service</p>	<p>Initial configuration</p> <p>$J_1(1) \quad J_2(1) \quad J_3(0)$</p>	<p>Opening JOB queue</p>
 <p>1. $J_4(2)$ arrives</p> <p>Machine service</p>	<p>1. $J_4(2)$ arrives</p> <p>$J_4(2) \quad J_1(1) \quad J_2(1) \quad J_3(0)$</p>	<p>Insert $J_4(2)$</p>

(Contd.)

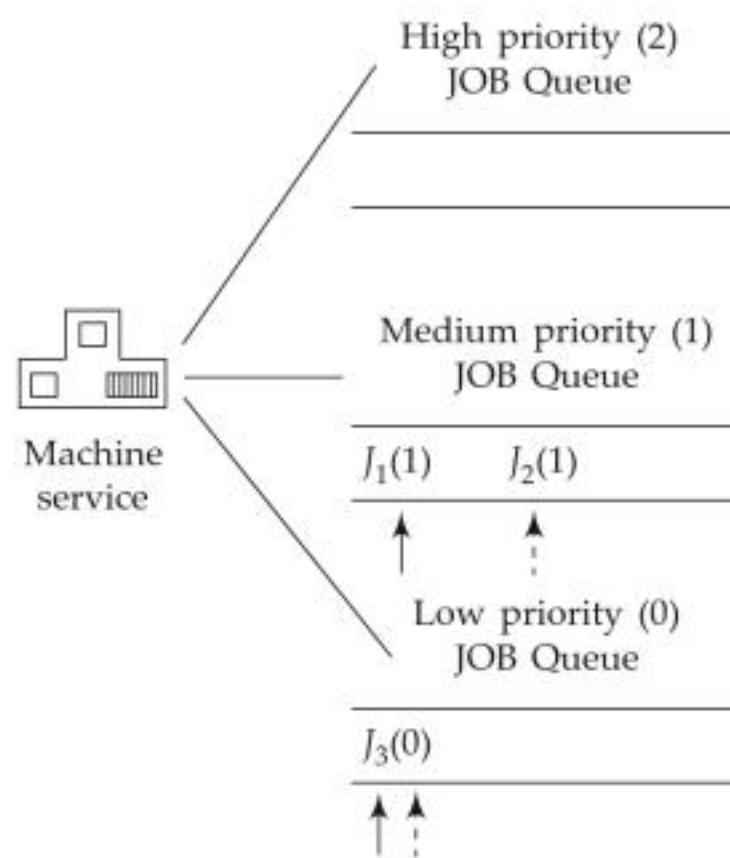
(Contd.)



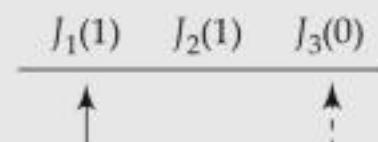
(Contd.)

(Contd.)

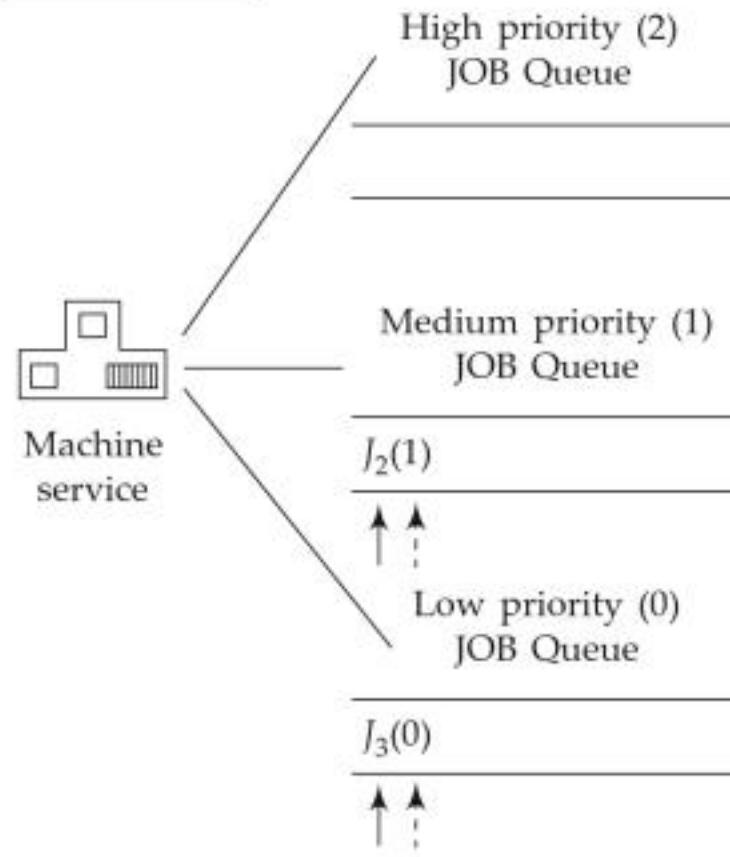
4. Execute Job



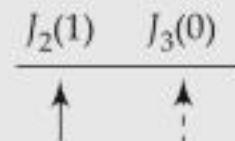
4. Execute Job

 $J_5(2)$ is deleted

5. Execute Job



5. Execute Job

 $J_1(1)$ is deleted

↑ Front

↑ Rear

↑ : Front

↑ : Rear



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

DEQ:		LEFT: 3	RIGHT: 5
[1] [2] [3] [4] [5] [6]	R T S		

The following operations demonstrate the working of the deque *DEQ* which supports insertions and deletions at both ends.

- (i) Insert X at the left end and Y at the right end

DEQ:		LEFT: 2	RIGHT: 6
[1] [2] [3] [4] [5] [6]	X R T S Y		

- (ii) Delete twice from the right end

DEQ:		LEFT: 2	RIGHT: 4
[1] [2] [3] [4] [5] [6]	X R T		

- (iii) Insert G, Q and M at the left end

DEQ:		LEFT: 5	RIGHT: 4
[1] [2] [3] [4] [5] [6]	G X R T M Q		

- (iv) Insert J at the right end

Here no insertion is possible since the deque is full. Observe the condition LEFT=RIGHT+1 when the deque is full.

- (v) Delete twice from the left end

DEQ:		LEFT: 1	RIGHT: 4
[1] [2] [3] [4] [5] [6]	G X R T		

It is easy to observe that for insertions at the left end, LEFT is decremented by 1 ($\text{mod } n$) and for insertions at the right end RIGHT is incremented by 1 ($\text{mod } n$). For deletions at the left end, LEFT is incremented by 1 ($\text{mod } n$) and for deletions at the right end, RIGHT is decremented by 1 ($\text{mod } n$) where n is the capacity of the deque. Again, before performing a deletion if LEFT=RIGHT, then it implies that there is only one element and in such a case after deletion set LEFT =RIGHT=NIL to indicate that the deque is empty.

Applications

5.5

In this section we discuss the application of a linear queue and a priority queue in the scheduling of jobs by a processor in a time sharing system.

Application of a linear queue

Figure 5.9 shows a basic diagram of a time-sharing system. A CPU (processor) endowed with memory resources, is to be shared by n number of computer users. The sharing of the processor

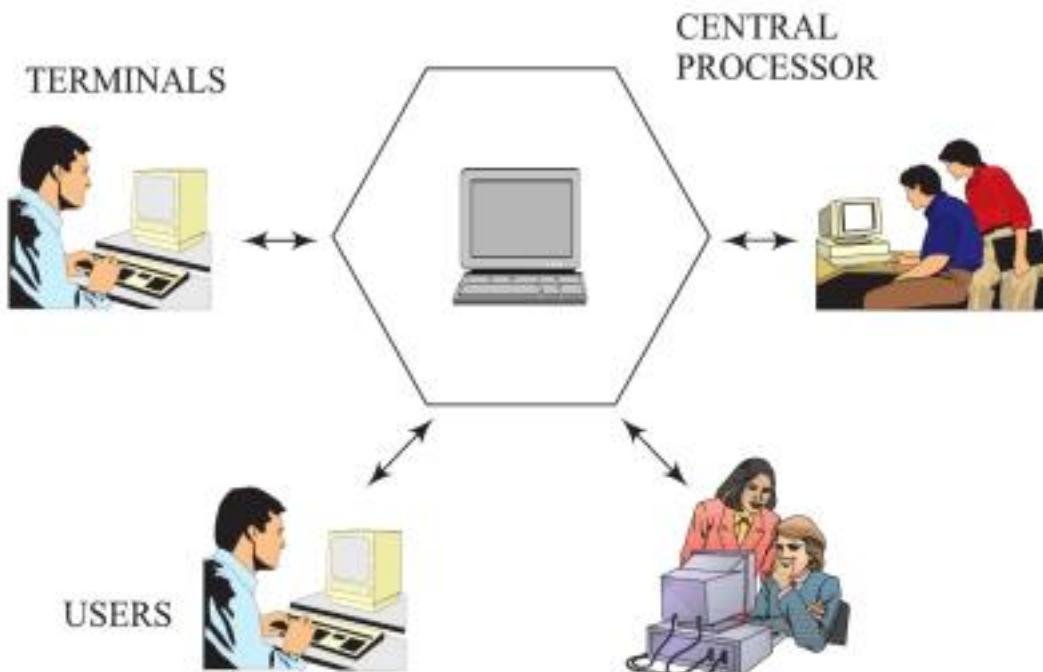


Fig. 5.9 A basic diagram of a time-sharing system

and memory resources is done by allotting a definite time slice of the processor's attention on the users and in a round-robin fashion. In a system such as this, the users are unaware of the presence of other users and are led to believe that their job receives the undivided attention of the CPU. However, to keep track of the jobs initiated by the users, the processor relies on a queue data structure recording the active user ids. Example 5.5 demonstrates the application of a queue data structure for this job-scheduling problem.

Example 5.5 The following is a table of three users A, B, C with their job requests $J_i(k)$ where i is the job number and k is the time required to execute the job.

User	Job requests and the execution time in μ secs
A	$J_1(4), J_2(3)$
B	$J_3(2), J_4(1), J_5(1)$
C	$J_6(6)$

Thus $J_1(4)$, a job request initiated by A needs 4μ secs for its execution before the user initiates the next request of $J_2(3)$. Throughout the simulation, we assume a uniform user delay period of 5μ secs between any two sequential job requests initiated by a user. Thus B initiates $J_4(1)$, 5μ secs after the completion of $J_3(2)$ and so on. Also to simplify simulation, we assume that the CPU gives whole attention to the completion of a job request before moving to the next job request. In other words, all the job requests complete their execution well within the time slice allotted to them. To initiate the simulation, we assume that A logged in at time 0, B at time 1 and C at time 2. Figure 5.10 shows a graphical illustration of the simulation. Note that at time 2 while A 's $J_1(4)$ is being executed, B is in the wait mode with $J_3(2)$ and C has just logged in. The objective is to ensure the CPU's attention to all the jobs logged in according to the principle of FIFO.

To tackle such a complex scenario, a queue data structure comes in handy. As soon as a job request is made by a user, the user id is inserted into a queue. A job that is to be processed next

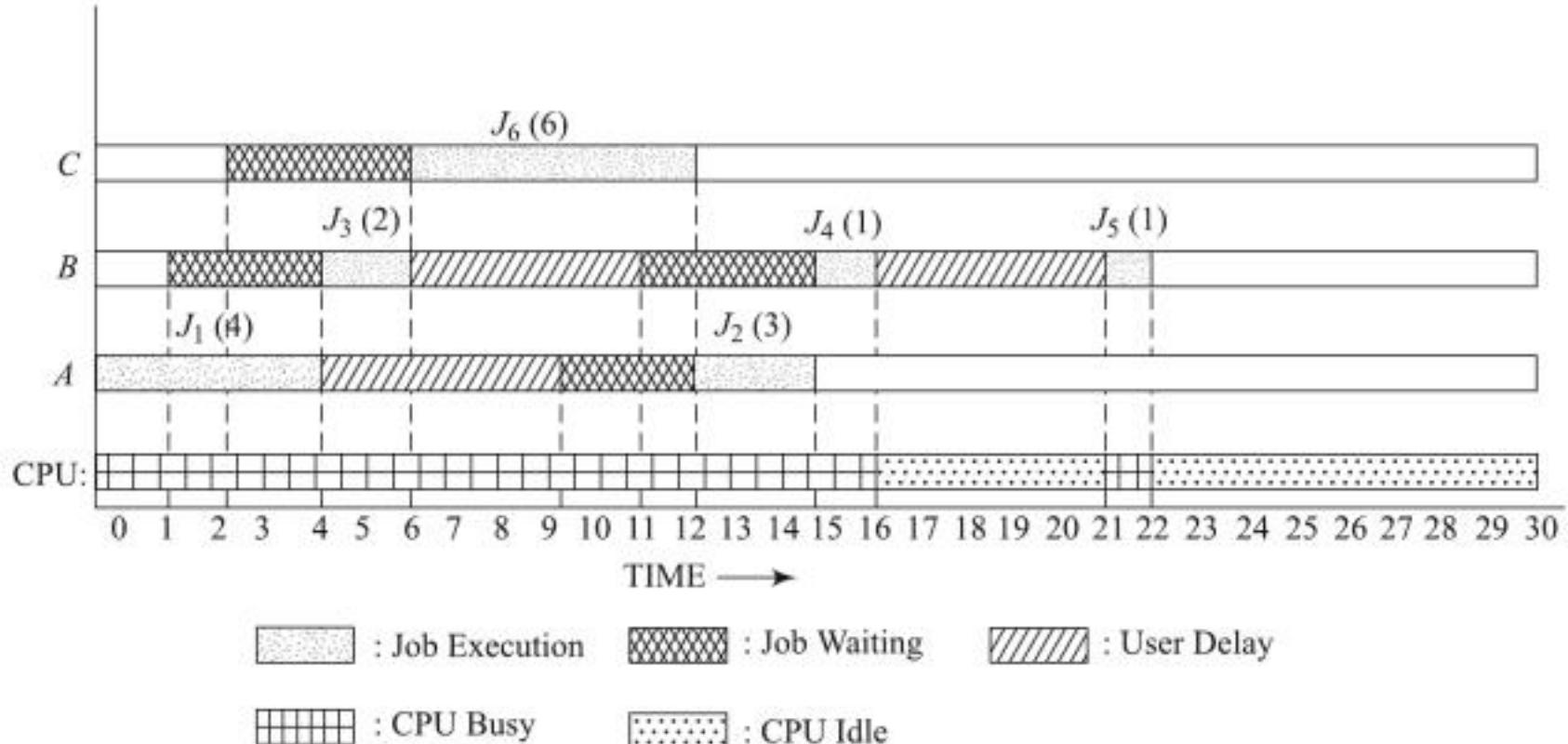


Fig. 5.10 Time sharing system simulation non-priority based job requests

would be the one at the head of the queue. A job until its execution is complete remains at the head of the queue. Once the request has been processed and execution is complete, the user id is deleted from the queue.

A snap shot of the queue data structure at times 5, 10 and 14 is shown in Fig. 5.11. Observe that during the time period 16-21 the CPU is left idle.

	Job Queue	
Time 5	B (J ₃ (2))	C (J ₆ (6))
Time 10	C (J ₆ (6))	A (J ₂ (3))
Time 14	A (J ₂ (3))	C (J ₄ (1))

Fig. 5.11 Snapshot of the queue at times 5, 10 and 14

Application of priority queues

Assume a time-sharing system in which job requests by users are of different categories. For example, some requests may be real time, the others online and the last may be batch processing requests. It is known that real time job requests carry the highest priority, followed by online processing and batch processing in that order. In such a situation the job scheduler needs to maintain a priority queue to execute the job requests based on their priorities. If the priority queue were to be implemented using a cluster of queues of varying priorities, the scheduler has to maintain one queue for real time jobs (*R*), one for online processing jobs (*O*) and the third for batch processing jobs (*B*). The CPU proceeds to execute a job request in *O* only when *R* is empty. In other words all real time jobs awaiting execution in *R* have to be completed and cleared before

execution of a job request from O . In the case of queue B , before executing a job in queue B , the queues R and O should be empty. Example 5.6 illustrates the application of a priority queue in a time-sharing system with priority-based job requests.

Example 5.6 The following is a table of three users A, B, C with their job requests. $R_i(k)$ indicates a real time job R_i whose execution time is $k \mu$ secs. Similarly $B_i(k)$ and $O_i(k)$ indicate batch processing and online processing jobs respectively.

User	Job requests and their execution time in μ secs		
A	$R_1(4)$ $B_1(1)$		
B	$O_1(2)$	$O_2(3)$	$B_2(3)$
C	$R_2(1)$	$B_3(2)$	$O_3(3)$

As before we assume a user delay of 5μ secs between any two sequential job requests by the user and assume that the CPU gives undivided attention to a job request until its completion. Also, A, B and C login at times 0, 1 and 2 respectively.

Figure 5.12. illustrates the simulation of the job scheduler for the priority based job requests. Figure 5.13 shows the snap shot of the priority queue at times 4, 8 and 12. Observe that the processor while scheduling jobs and executing them falls into idle modes during time periods 7-9 and 15-17.

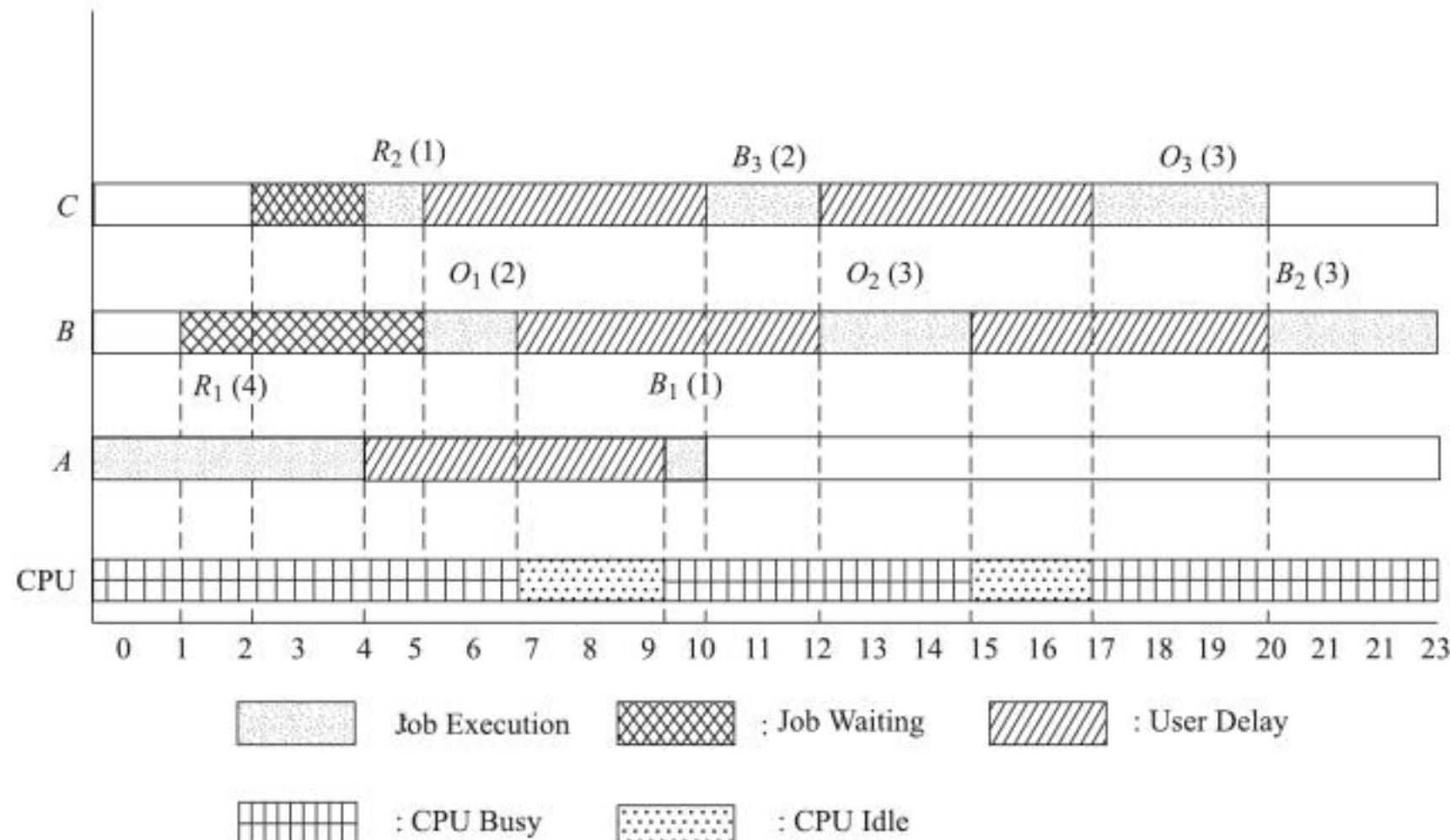


Fig. 5.12 Simulation of the time sharing system for priority based jobs



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.

(Contd.)

11	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td>56</td><td></td><td>50</td><td>77</td><td>22</td></tr> <tr><td>[0]</td><td>[1]</td><td>[2]</td><td>[3]</td><td>[4]</td></tr> </table>	56		50	77	22	[0]	[1]	[2]	[3]	[4]	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td>56</td><td>56</td><td>10</td></tr> </table>	56	56	10									
56		50	77	22																				
[0]	[1]	[2]	[3]	[4]																				
56	56	10																						
12-16	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td></td><td></td><td></td><td></td><td></td></tr> <tr><td>[0]</td><td>[1]</td><td>[2]</td><td>[3]</td><td>[4]</td></tr> </table>						[0]	[1]	[2]	[3]	[4]	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td>50</td><td>56</td><td>0</td></tr> <tr><td>77</td><td>56</td><td>0</td></tr> <tr><td>22</td><td>56</td><td>0</td></tr> <tr><td>56</td><td>56</td><td>0</td></tr> </table>	50	56	0	77	56	0	22	56	0	56	56	0
[0]	[1]	[2]	[3]	[4]																				
50	56	0																						
77	56	0																						
22	56	0																						
56	56	0																						

Output of the program code: 50 77 22 56

Problem 5.3 S and Q are a stack and a priority queue of integers respectively. The priority of an element C joining the priority queue Q is computed as $C \bmod 3$. In other words the priority numbers of the elements are either 0 or 1 or 2. Given A, B, C to be integer variables, what is the output of the following code? The procedures are similar to those used in Illustrative Problems 5.1 and 5.2, $I = 5.1$ and $I = 5.2$. However, the queue procedures are modified to appropriately work on a priority queue.

```

1. A = 10
2. B = 11
3. C = A+B
4. while (C < 110) do
5.   if (C mod 3) = 0 then PUSH (S, C)
6.   else ENQUEUE (Q, C)
7.   A = B
8.   B = C
9.   C = A + B
10. end
11. while not EMPTY_STACK (S) do
12.   POP (S, C)
13.   PRINT (C)
14. end
15. while not EMPTY_QUEUE (Q) do
16.   DEQUEUE (Q, C)
17.   PRINT (C)
18. end

```

Solution:

Steps	Stack S	Queue Q	A	B	C
1-3	21		10	11	21
4-6	21		10	11	21

(Contd.)

(Contd.)

7-10	21		11	21	32
4-6	21	32 ⁽²⁾	11	21	32
7-10	21	32 ⁽²⁾	21	32	53
4-6	21	32 ⁽²⁾ 53 ⁽²⁾	21	32	53
7-10	21	32 ⁽²⁾ 53 ⁽²⁾	32	53	85
4-6	21	32 ⁽²⁾ 53 ⁽²⁾ 85 ⁽¹⁾	32	53	85
7-10	21	32 ⁽²⁾ 53 ⁽²⁾ 85 ⁽¹⁾	53	85	138
11-14		32 ⁽²⁾ 53 ⁽²⁾ 85 ⁽¹⁾	53	85	21
			Output: 21		
15-18			53	85	32
			53	85	53
			53	85	85
			Output: 32 53 85		

The final output is: 21 32 53 85

Problem 5.4 TOKEN is a priority queue for organizing n data items with m priority numbers. TOKEN is implemented as a two dimensional array TOKEN[1 : m , 1 : p] where p is the maximum number of elements with a given priority. Execute the following operations on TOKEN [1 : 3, 1 : 2]. Here INSERT ('xxx', m) indicates the insertion of item 'xxx' with priority number m and DELETE() indicates the deletion of the first among the high priority items.

- (i) INSERT('not', 1)
- (ii) INSERT('and', 2)
- (iii) INSERT('or', 2)
- (iv) DELETE()
- (v) INSERT('equ', 3);

Solution: The two dimensional array TOKEN[1:3, 1:2] before the execution of operations is as given below:

TOKEN: [1] [2]

$$\begin{matrix} 1 & \begin{bmatrix} - & - \end{bmatrix} \\ 2 & \begin{bmatrix} - & - \end{bmatrix} \\ 3 & \begin{bmatrix} - & - \end{bmatrix} \end{matrix}$$

After the execution of operations, TOKEN[1:3, 1:2] is as shown below:

(i) INSERT ('not', 1) (ii) INSERT ('and', 2) (iii) INSERT ('or', 2)	$\begin{matrix} [1] & [2] \\ 1 & \begin{bmatrix} \text{'not'} & - \\ - & - \end{bmatrix} \\ 2 & \begin{bmatrix} \text{'and'} & \text{'or'} \\ - & - \end{bmatrix} \\ 3 & \end{matrix}$
(iv) DELETE ()	$\begin{matrix} [1] & [2] \\ 1 & \begin{bmatrix} - & - \\ - & - \end{bmatrix} \\ 2 & \begin{bmatrix} \text{'and'} & \text{'or'} \\ - & - \end{bmatrix} \\ 3 & \end{matrix}$
	Note how 'not' which is the first among the elements with the highest priority, is deleted

Problem 5.5 $DEQ[0:4]$ is an output restricted deque implemented as a circular array and LEFT and RIGHT indicate the ends of the deque as shown below. $INSERT('xx', [LEFT | RIGHT])$ indicates the insertion of the data item at the left or right end as the case may be, and $DELETE()$ deletes the item from the left end only.

DEQ:

LEFT: 2

RIGHT: 5

[1]	[2]	[3]	[4]	[5]	[6]
C1	A4	Y7	N6		

Execute the following insertions and deletions on DEQ:

- (i) INSERT('S5', LEFT)
 - (ii) INSERT('K9', RIGHT)
 - (iii) DELETE()
 - (iv) INSERT('V7', LEFT)
 - (v) INSERT('T5', LEFT)

Solution:

- (i) DEQ after the execution of operations (i) INSERT('S5', LEFT)
(ii) INSERT('K9', RIGHT)

DEO:

LEFT: 1

RIGHT: 6

[1]	[2]	[3]	[4]	[5]	[6]
S5	C1	A4	Y7	N6	K9

(ii) DEQ after the execution of $\text{DELETE}()$

DEQ.

LEFT: 2

RIGHT: 6

[1]	[2]	[3]	[4]	[5]	[6]
C1	A4	Y7	N6	K9	

- (iii) DEQ after the execution of operations (iv) INSERT('V7', LEFT)
(v) INSERT('T5', LEFT)

DEQ:

[1]	[2]	[3]	[4]	[5]	[6]
V7	C1	A4	Y7	N6	K9

LEFT: 1

RIGHT: 6

After the execution of operation INSERT('V7', LEFT), the deque is full. Hence 'T5' is not inserted into the deque.



Review Questions

- Which among the following properties does not hold good in a queue?
 - A queue supports the principle of First come First served.
 - An enqueueing operation shrinks the queue length
 - A dequeuing operation affects the front end of the queue.
 - An enqueueing operation affects the rear end of the queue
 - (a) (i) (b) (ii) (c) (iii) (d) (iv)
- A linear queue Q is implemented using an array as shown below. The FRONT and REAR pointers which point to the physical front and rear of the queue, are also shown.

FRONT: 2 REAR: 3

X	Y	A	Z	S
[1]	[2]	[3]	[4]	[5]

Execution of the operation ENQUEUE(Q, 'W') would yield the FRONT and REAR pointers to respectively carry the values shown in

- 2 and 4
 - 3 and 3
 - 3 and 4
 - 2 and 3
- For the linear queue shown in Review Question 2 of Chapter 5, execution of the operation DEQUEUE(Q, M) where M is an output variable would yield M, FRONT and REAR to respectively carry the values
 - Z, 2, 3
 - A, 2, 2
 - Y, 3, 3
 - A, 2, 3
 - Given the following array implementation of a circular queue, with FRONT and REAR pointing to the physical front and rear of the queue,

FRONT: 3 REAR: 4

X	Y	A	Z	S
[1]	[2]	[3]	[4]	[5]

Execution of the operations ENQUEUE(Q, 'H'), ENQUEUE(Q, 'T') done in a sequence would result in

- Invoking Queue full condition soon after ENQUEUE(Q, 'H') operation
 - Aborting the ENQUEUE(Q, 'T') operation
 - Yielding FRONT = 1 and REAR = 4, after the operations.
 - Yielding FRONT = 3 and REAR =1, after the operations
 - (a) (i) (b) (ii) (c) (iii) (d) (iv)
- State whether true or false:

For the following implementation of a queue, where FRONT and REAR point to the physical front and rear of the queue,



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.

a node with address X , from the reserved area of the pool, to the free area of the pool. In other words, X is an input parameter of the function, the value of which is to be provided by the user.

Irrespective of the number of data item fields, a linked list is categorized as *singly linked list*, *doubly linked list*, *circularly linked list* and *multiply linked list* based on the number of link fields it owns and/or its intrinsic nature. Thus a linked list with a *single link field* is known as *singly linked list* and the same with a *circular connectivity* is known as *circularly linked list*. On the other hand, a linked list with *two links each pointing to the predecessor and successor* of a node is known as a *doubly linked list* and the same with *multiple links* is known as *multiply linked list*. The following sections discuss these categories of linked lists in detail.

Singly Linked Lists

6.2

Representation of a singly linked list

A *singly linked list* is a linear data structure, each node of which has one or more data item fields (DATA) but only a *single link field* (LINK).

Figure 6.4 illustrates an example of a singly linked list and its node structure. Observe that the node in the list carries a single link which points to the node representing its immediate successor in the list of data elements.

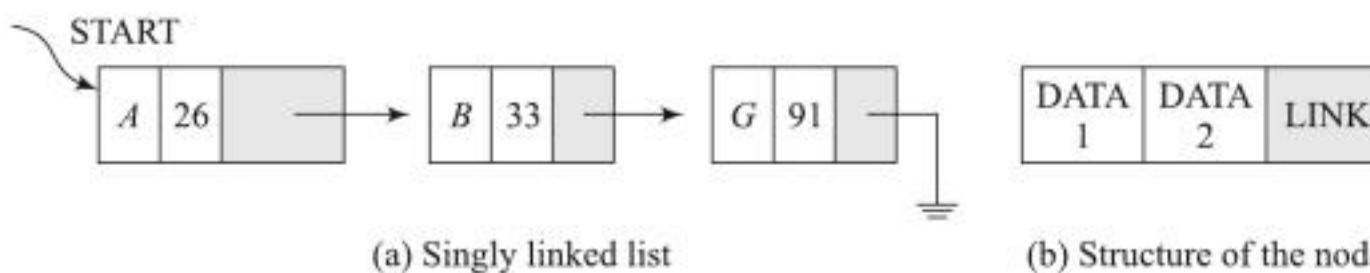
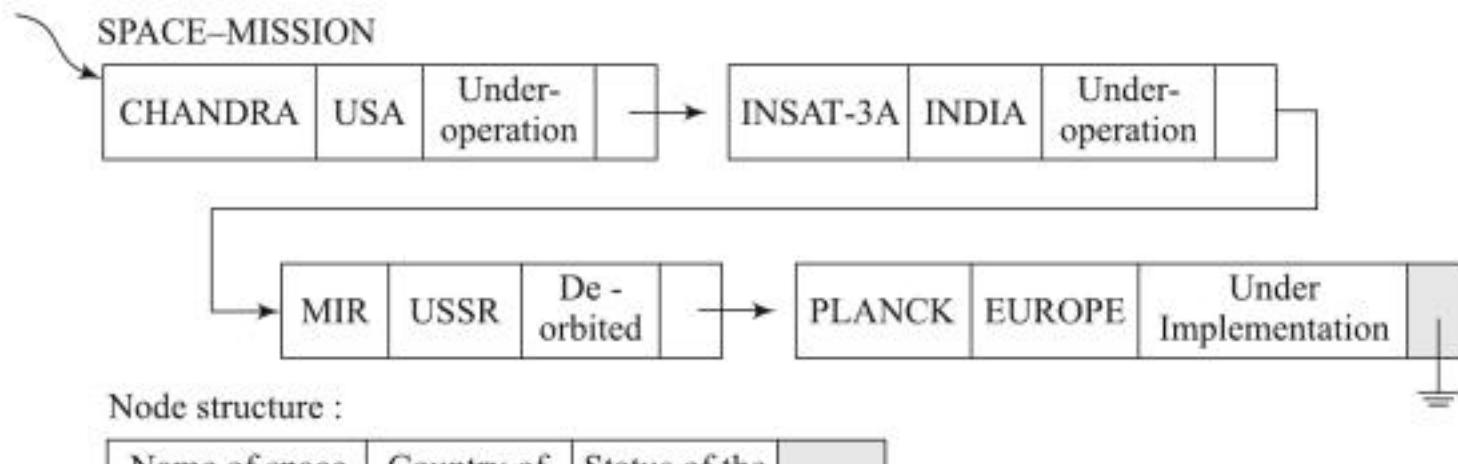


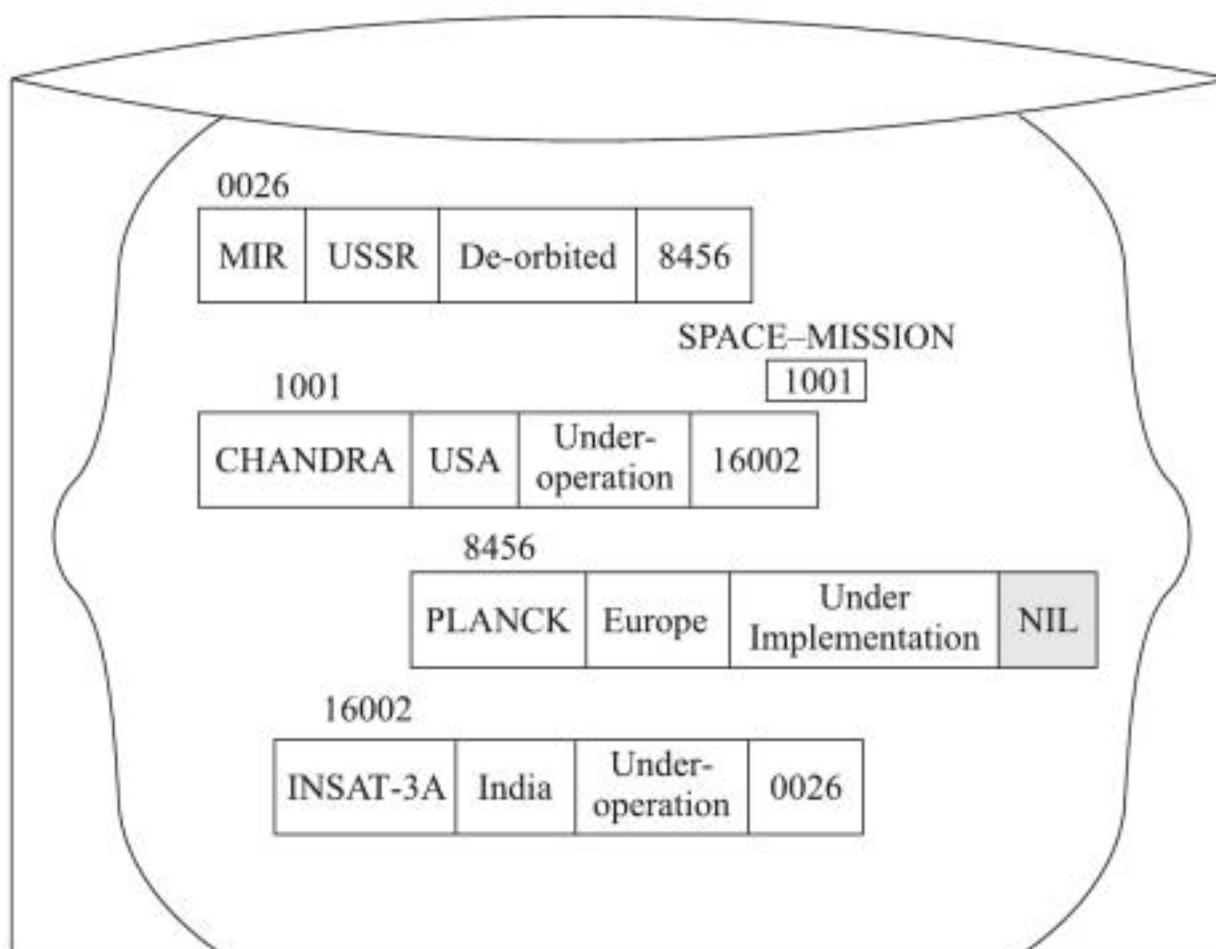
Fig. 6.4 A singly linked list and its node structure

Every node which is basically a chunk of memory, carries an address. When a set of data elements to be used by an application are represented using a linked list, each data element is represented by a node. Depending on the information content of the data element, one or more data items may be opened in the node. However, in a singly linked list only a single link field is used to point to the node which represents its neighbouring element in the list. The last node in the linked lists has its link field empty. The empty link field is also referred to as *null link* or in programming language parlance – *null pointer*. The notations NIL, or a ground symbol ($\underline{\underline{}}^{\top}$) or a zero (0) are commonly used to indicate null links. The entire linked list is kept track of by remembering the address of the *start node*. This is indicated by START in the figure. Obviously it is essential that the START pointer is carefully handled, lest it results in losing the entire list.

Example Consider a list SPACE-MISSION of four data elements as shown in Fig. 6.5(a). This logical representation of the list has each node carrying three DATA fields viz., name of the space mission, country of origin, the current status of the mission, and a single link pointing to the next node. Let us suppose the nodes which house 'Chandra', 'INSAT-3A', 'Mir' and 'Planck' have addresses 1001, 16002, 0026 and 8456 respectively. Figure 6.5(b) shows the physical



(a) Logical representation of SPACE-MISSION



(b) Physical representation of SPACE-MISSION

Fig. 6.5 A singly linked list—its logical and physical representation

representation of the linked list. Note how the nodes are distributed all over the storage memory and not physically contiguous. Also observe how the LINK field of each node remembers the address of the node of its logical neighbour. The LINK field of the last node is NIL. The arrows in the logical representation represent the addresses of the neighbouring nodes in its physical representation.

Insertion and deletion in a singly linked list

To implement insertion and deletion in a singly linked list, one needs the two functions introduced in Sec. 6.1.2, viz., GETNODE(X) and RETURN(X) respectively.

Insert operation Given a singly linked list START, to insert a data element ITEM into the list to the right of node NODE, (ITEM is to be inserted as the successor of the data element represented by node NODE) the steps to be undertaken are given below. Figure 6.6. illustrates the logical representation of the insert operation.

- (i) Call GETNODE(X) to obtain a node to accommodate ITEM. Node has address X.
- (ii) Set DATA field of node X to ITEM (i.e.) DATA(X) = ITEM.
- (iii) Set LINK field of node X to point to the original right neighbour of node NODE (i.e.) LINK(X) = LINK(NODE).
- (iv) Set LINK field of NODE to point to X (i.e.) LINK(NODE) = X.

Algorithm 6.1 illustrates a pseudo code procedure for insertion in a singly linked list which is non empty.

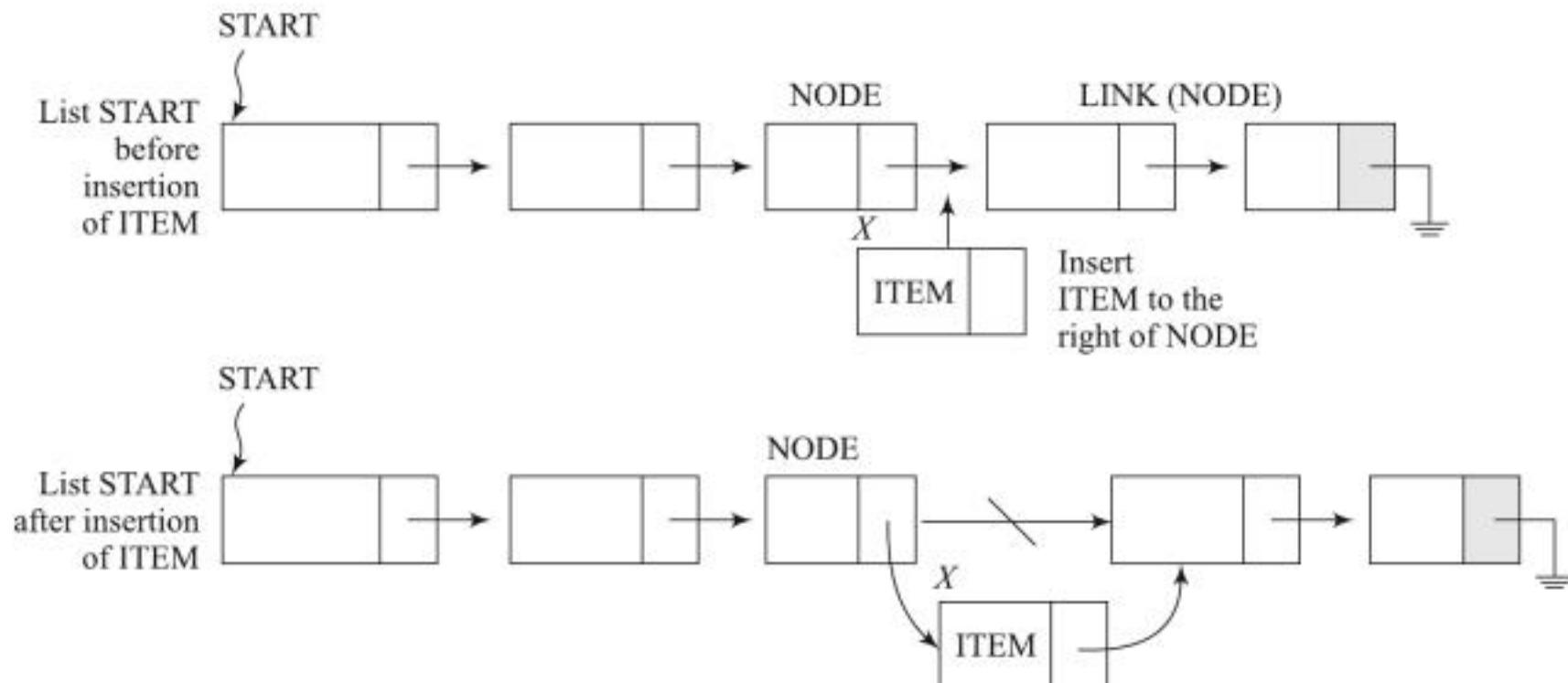


Fig. 6.6 Logical representation of insertion in a singly linked list

Algorithm 6.1: To insert a data element ITEM in a non empty singly liked list START, to the right of node NODE

```

Procedure INSERT_SL(START, ITEM, NODE)
    /* Insert ITEM to the right of node NODE in the list START */
    Call GETNODE(X);
    DATA(X) = ITEM;
    LINK(X) = LINK(NODE); /* Node X points to the original
                           right neighbour of node NODE */
    LINK(NODE) = X;
end INSERT_SL.

```

However, during insert operation in a list, it is advisable to test if START pointer is null or non-null. If START pointer is null (START = NIL) then the singly linked list is empty and hence the insert operation prepares to insert the data as the first node in the list. On the other hand, if START pointer is non-null (START ≠ NIL), then the singly linked list is non empty and hence the insert operation prepares to insert the data at an appropriate position in the list as specified by

the application. Algorithm 6.1 works on a non empty list. To handle empty lists the algorithm has to be appropriately modified as illustrated in Algorithm 6.2.

Algorithm 6.2: To insert *ITEM* after node *NODE* in a singly linked list *START*

```

Procedure INSERT_SL_GEN (START, NODE, ITEM)
    /* Insert ITEM as the first node in the list if START
    is NIL. Otherwise insert ITEM after node NODE */
    Call GETNODE (X);
    DATA (X) = ITEM; /* Create node for ITEM */
    if (START = NIL) then
        {LINK (X) = NIL; /* List is empty*/
        START = X; } /*Insert ITEM as the first node */
    else
        {LINK (X) = LINK(NODE);
        LINK(NODE) = X; } /* List is non empty. Insert ITEM
                           to the right of node NODE */
end INSERT_SL_GEN.

```

In sheer contrast to an insert operation in a sequential data structure, observe the total absence of data movement in the list during insertion of *ITEM*. The insert operation merely calls for the update of two links in the case of a non empty list.

Example 6.1 In the singly linked list SPACE-MISSION illustrated in Fig. 6.5(a-b), insert the following data elements:

- (i)

APPOLLO	USA	Landed
---------	-----	--------
- (ii)

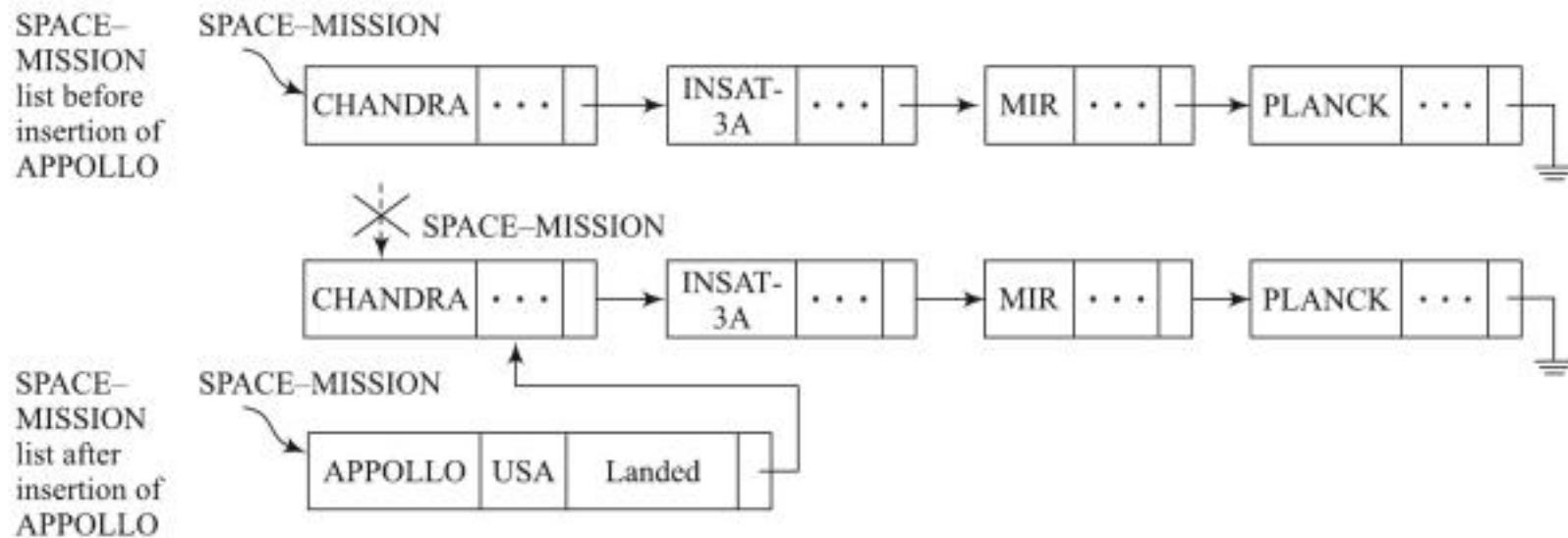
SOYUZ 4	USSR	Landed
---------	------	--------

Let us suppose the *GETNODE(X)* function releases nodes with addresses $X = 646$ and $X = 1187$ to accommodate APOLLO and SOYUZ 4 details respectively. The insertion of APOLLO is illustrated in Fig. 6.7(a-b) and the insertion of SOYUZ 4 is illustrated in Fig. 6.7(c-d).

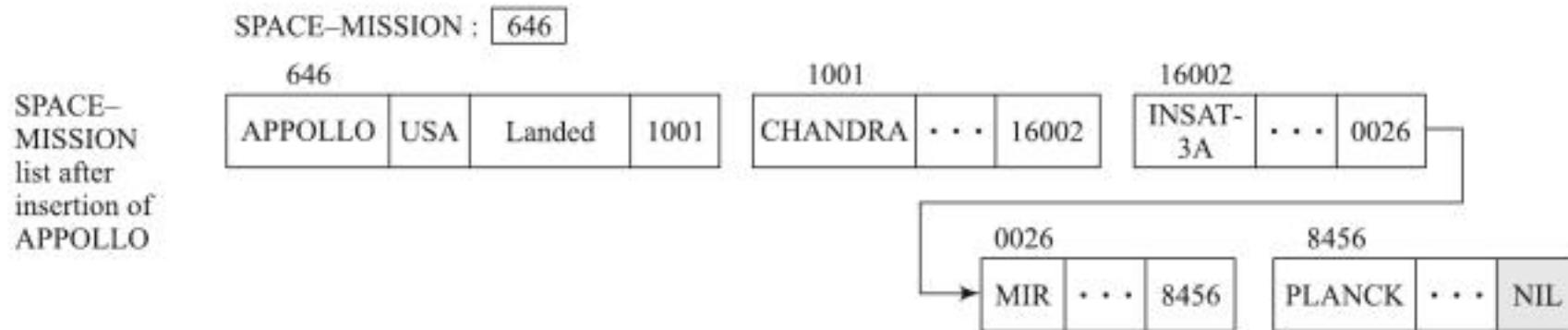
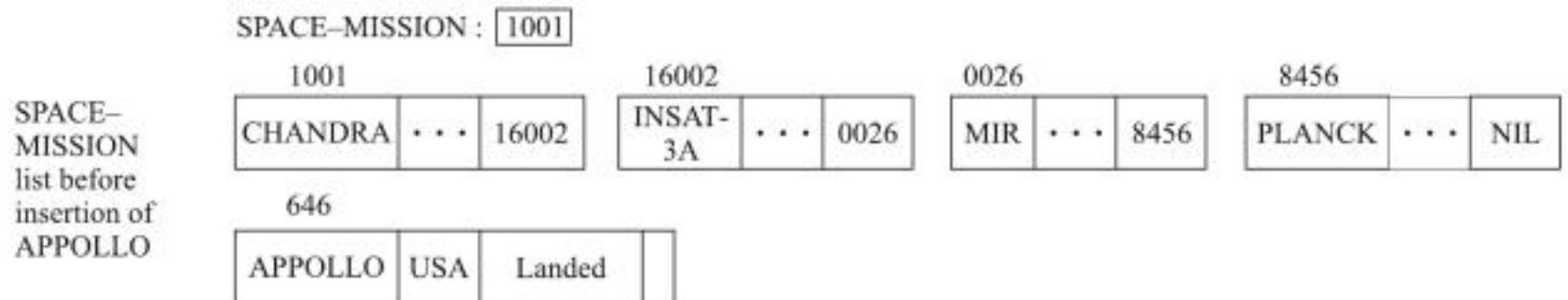
Delete operation Given a singly linked list *START*, the delete operation can acquire various forms such as deletion of a node *NODEY* next to that of a specific node *NODEX*, or more commonly deletion of a particular element in a list and so on. We now illustrate the deletion of a node which is the successor of node *NODEX*.

The steps for the deletion of a node next to that of *NODEX* in a singly linked *START* is given below. Figure 6.8 illustrates the logical representation of the delete operation.

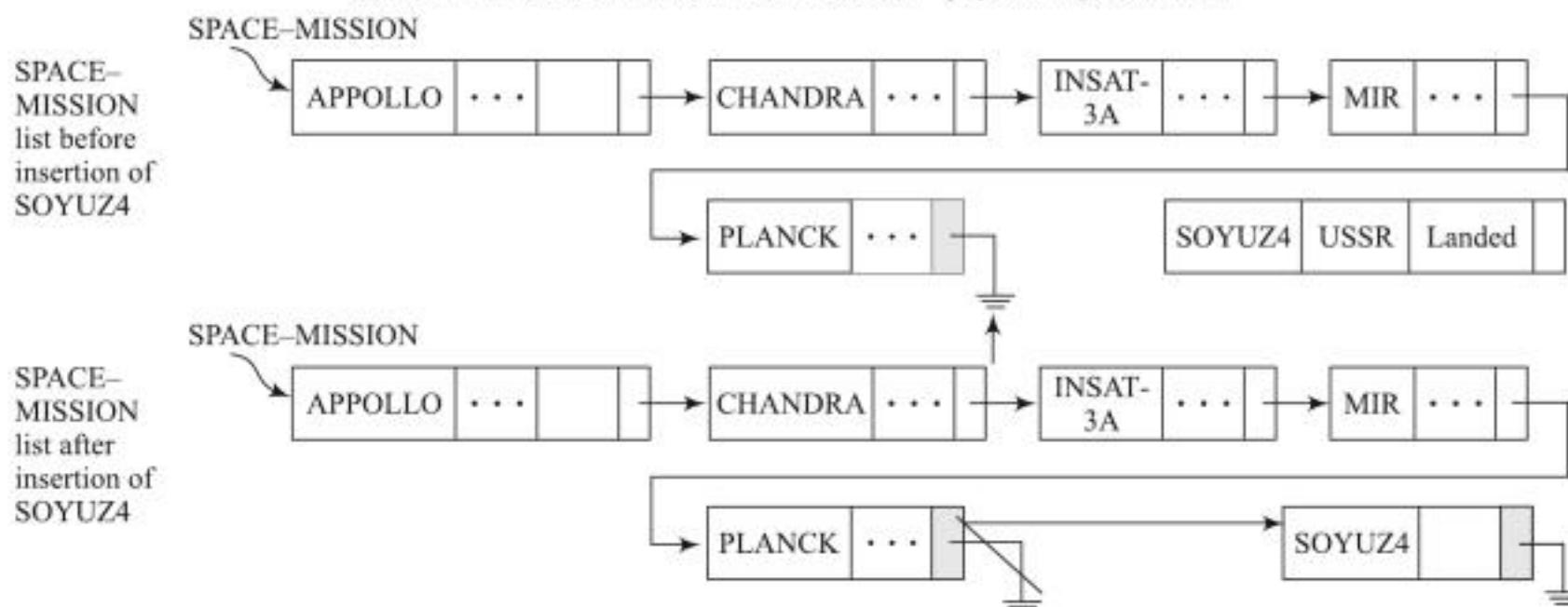
- (i) Set TEMP a temporary variable to point to the right neighbour of *NODEX*
(i.e.) $\text{TEMP} = \text{LINK}(\text{NODEX})$. The node pointed to by TEMP is to be deleted.
- (ii) Set *LINK* field of node *NODEX* to point to the right neighbour of TEMP
(i.e.) $\text{LINK}(\text{NODEX}) = \text{LINK}(\text{TEMP})$.
- (iii) Dispose node TEMP (i.e.) $\text{RETURN} (\text{TEMP})$.



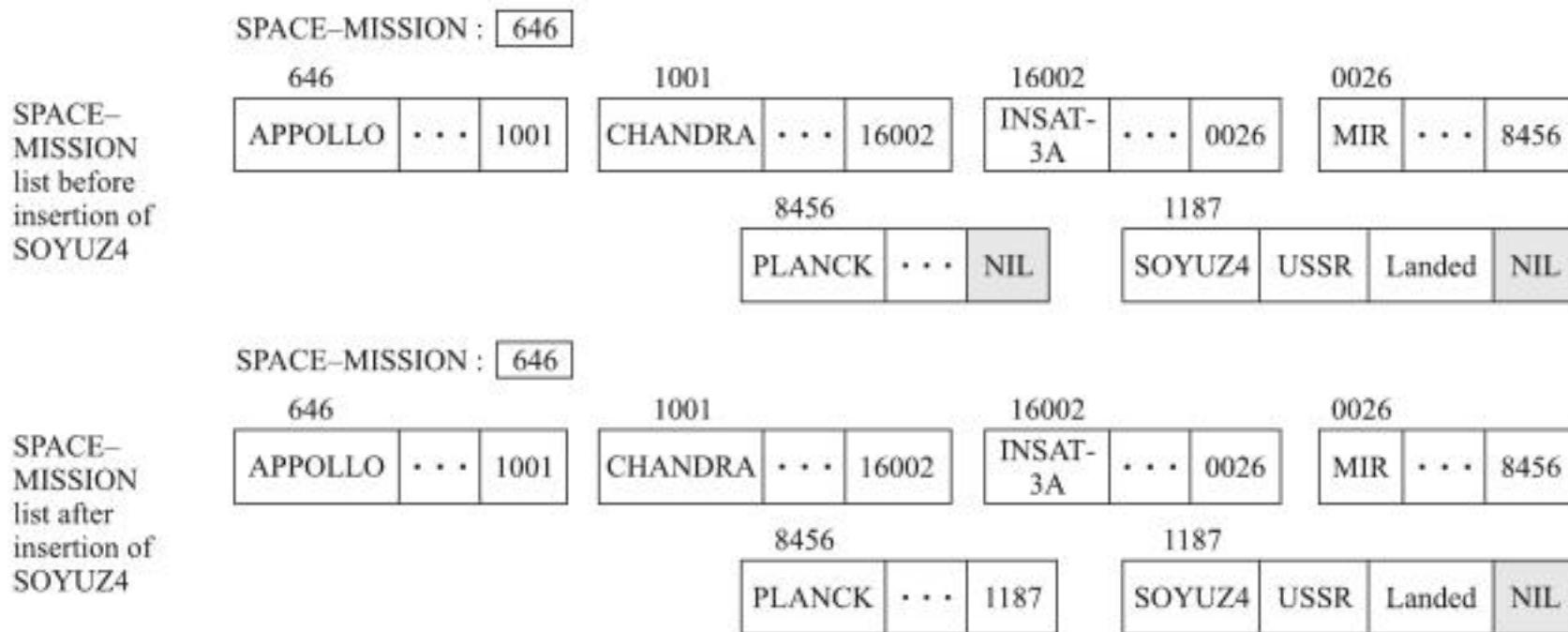
(a) Insert APPOLLO in list SPACE-MISSION—logical representation



(b) Insert APPOLLO in list SPACE-MISSION—physical representation



(c) Insert SOYUZ4 in list SPACE-MISSION—logical representation



(d) Insert SOYUZ4 in list SPACE-MISSION—physical representation

Fig. 6.7 Insertion of APOLLO and SOYUZ4 in the SPACE-MISSION list shown in Fig. 6.5(a-d)

Algorithm 6.3 illustrates a pseudo-code procedure for the deletion of a node which occurs to the right of a node NODEX in a singly linked list START. However, as always, it needs to be ensured that the delete operation is not undertaken over an empty list. Hence it is essential to check if START is empty.

Algorithm 6.3: Deletion of a node which is to the right of node NODEX in a singly linked list START

```

Procedure      DELETE_SL(START, NODEX)
if              (START = NIL) then
                    Call ABANDON_DELETE;
                    /*ABANDON_DELETE terminates the delete operation */
else
                    (TEMP = LINK(NODEX);
                     LINK(NODEX) = LINK(TEMP);
                     Call RETURN(TEMP);
end DELETE_SL.
    
```

Observe how in contrast to deletion in a sequential data structure which involves data movement, the deletion of a node in a linked list merely calls for the update of a single link.

Example 6.2 illustrates deletion of a node in a singly linked list.

Example 6.2 For the SPACE-MISSION list shown in Fig. 6.5(a-b) undertake the following deletions:

- Delete CHANDRA
- Delete PLANCK

The deletion of CHANDRA is illustrated in Fig. 6.9(a-b) and that of PLANCK is illustrated in Fig. 6.9 (c-d).

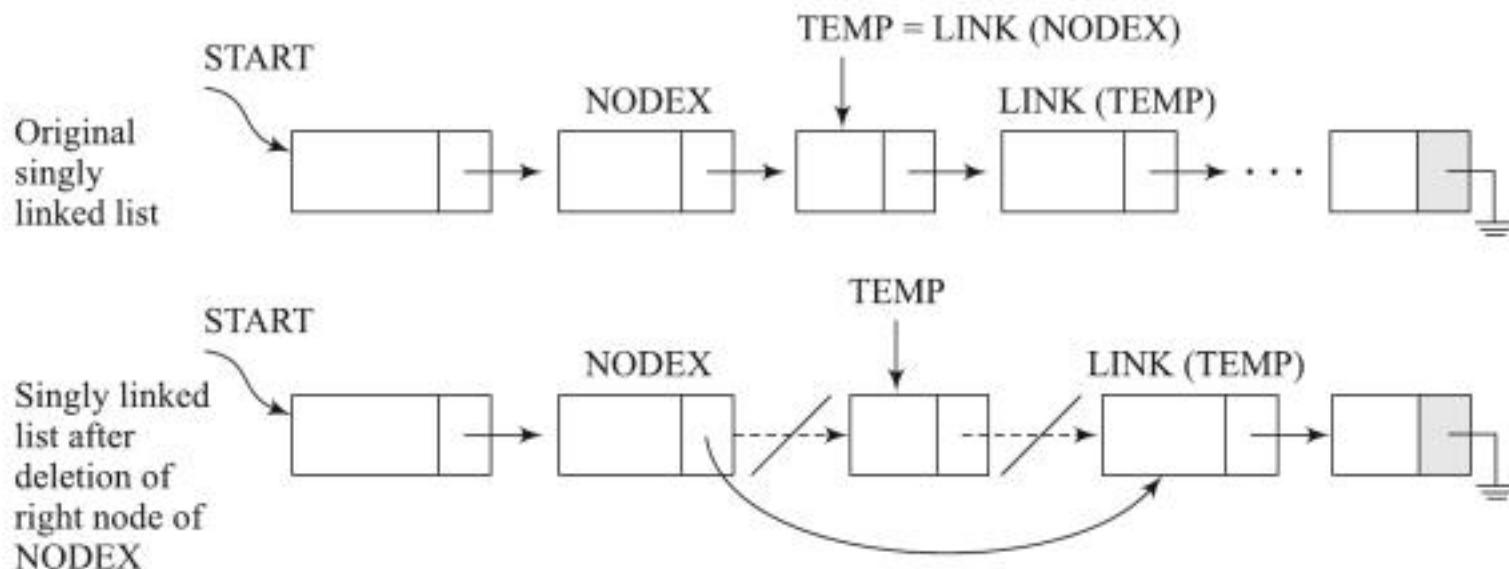


Fig. 6.8 Logical representation of deletion in a singly linked list

Circularly Linked Lists

6.3

Representation

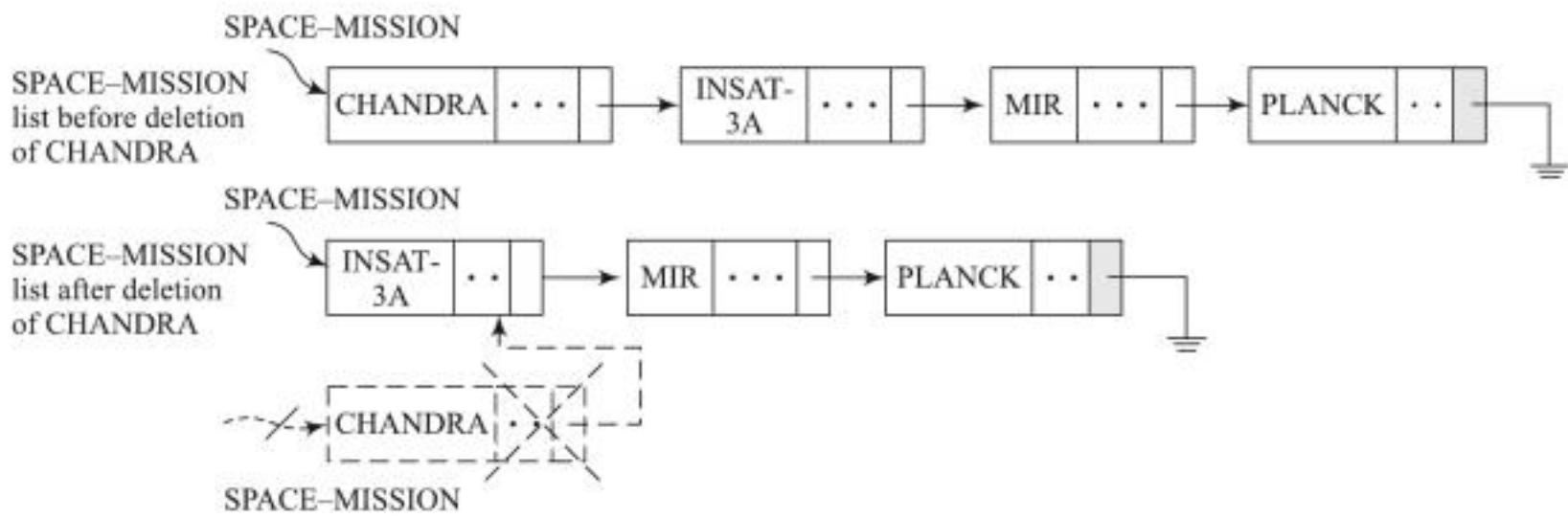
A normal singly linked list has its last node carrying a null pointer. For further improvement in processing one may replace the null pointer in the last node with the address of the first node in the list. Such a list is called as a *circularly linked list* or a *circular linked list* or simply a *circular list*. Figure 6.10 illustrates the representation of a circular list.

Advantages of circularly linked lists over singly linked lists

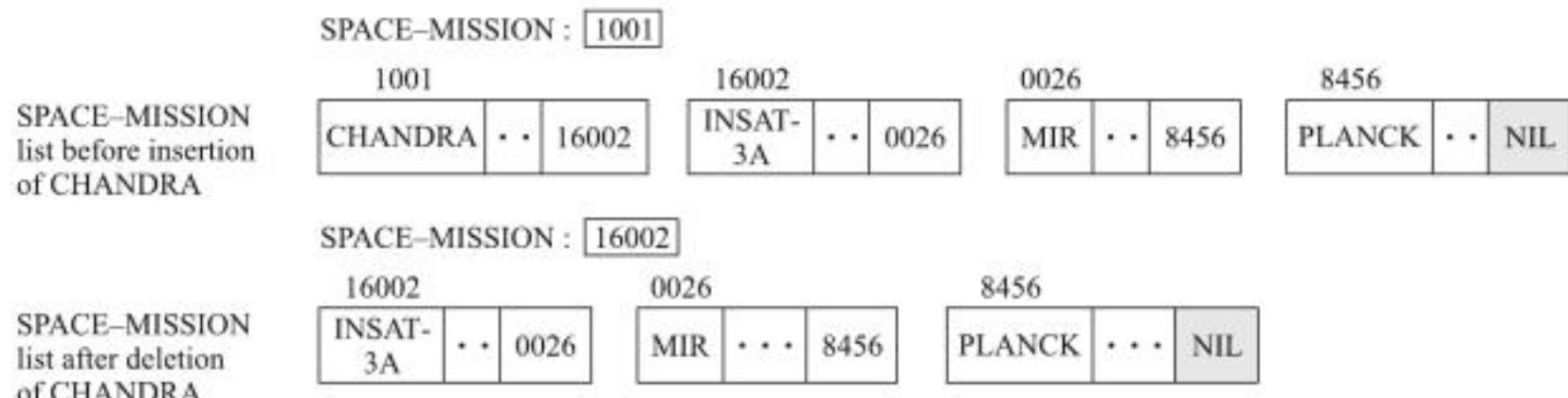
- (i) The most important advantage pertains to the accessibility of a node. One can access any node from a given node due to the circular movement permitted by the links. One has to merely loop through the links to reach a specific node from a given node.
- (ii) The second advantage pertains to delete operations. Recall that for deletion of a node X in a singly linked list, the address of the preceding node (for example node Y) is essential, to enable, update the LINK field of Y to point to the successor of node X. This necessity arises from the fact that in a singly linked list, one cannot access a node's predecessor due to the 'forward' movement of the links. In other words, LINK fields in a singly linked list point to successors and not predecessors. However, in the case of a circular list, to delete node X one need not specify the predecessor. It can be easily determined by a simple 'circular' search through the list before deletion of node X.
- (iii) The third advantage is the relative efficiency in the implementation of list based operations such as concatenation of two lists, erasing a whole list, splitting a list into parts and so on.

Disadvantages of circularly linked lists

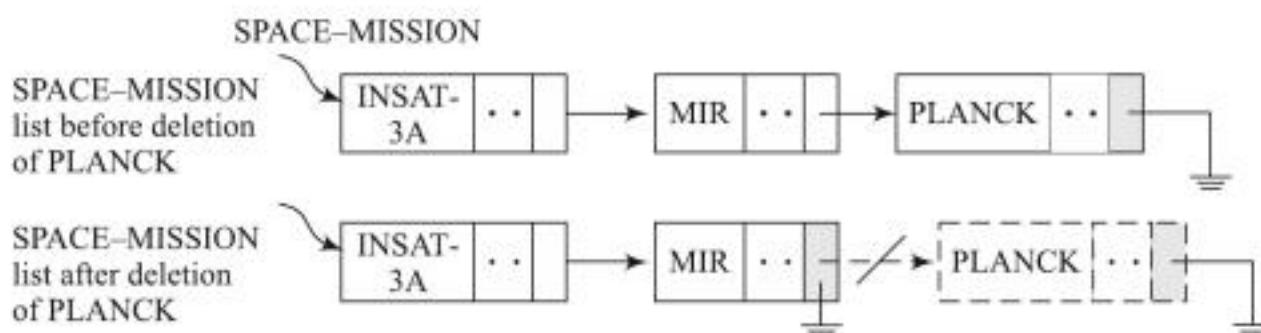
The only disadvantage of circularly linked lists is that during processing one has to make sure that one does not get into an infinite loop owing to the circular nature of pointers in the list. This is liable to occur owing to the absence of a node which will help point out the end of the list and thereby terminate processing.



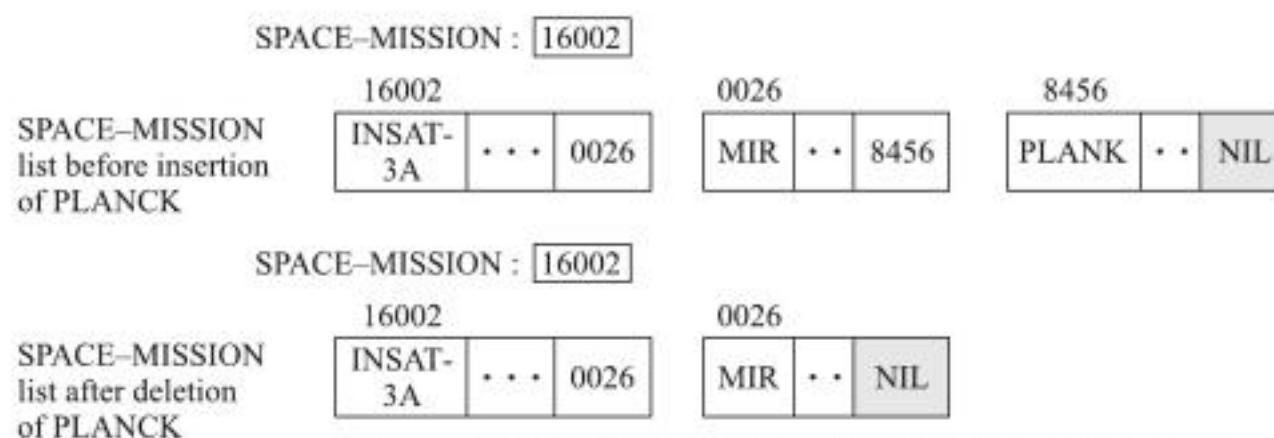
(a) Delete CHANDRA from list SPACE-MISSION—logical representation



(b) Delete CHANDRA from list SPACE-MISSION—physical representation



(c) Delete PLANCK from list SPACE-MISSION—logical representation



(d) Delete PLANCK from list SPACE-MISSION—physical representation

Fig. 6.9 *Deletion of CHANDRA and PLANCK from the SPACE-MISSION list*

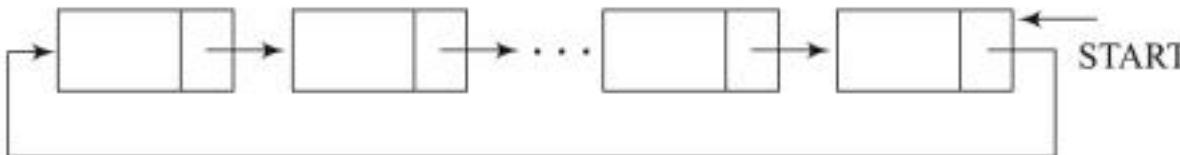


Fig. 6.10 Representation of a circular list

A solution to this problem is to designate a special node to act as the head of the list. This node, known as *list head* or *head node* has its advantages other than pointing to the beginning of a list. The list can never be empty and represented by a 'hanging' pointer ($\text{START} = \text{NIL}$) as was the case with empty singly linked lists. The condition for an empty circular list becomes ($\text{LINK}(\text{HEAD}) = \text{HEAD}$), where HEAD points to the head node of the list. Such a circular list is known as a *headed circularly linked list* or simply *circularly linked list with head node*. Figure 6.11 illustrates the representation of a headed circularly linked list.

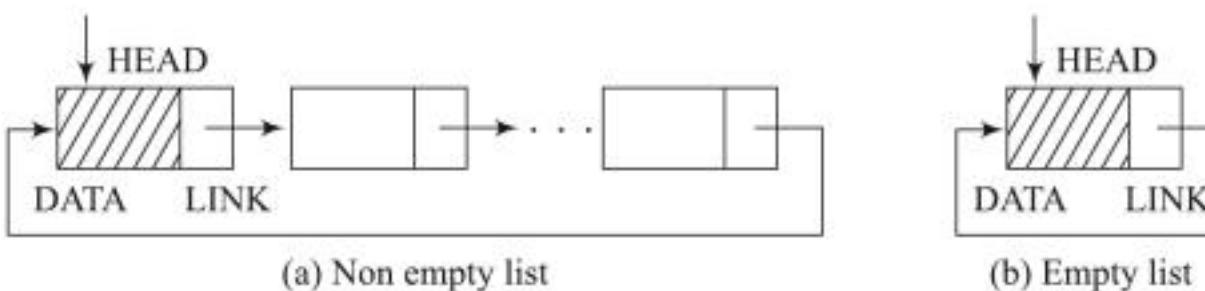


Fig. 6.11 A headed circularly linked list

Though the head node has the same structure as the other nodes in the list, the DATA field of the node is unused and is indicated as a shaded field in the pictorial representation. However, in practical applications these fields may be utilized to represent any useful information about the list relevant to the application, provided they are deftly handled and do not create confusion during the processing of the nodes.

Example 6.3 illustrates the functioning of circularly linked lists.

Example 6.3 Let CARS be a headed circularly linked list of four data elements as shown in Fig. 6.12(a). To insert MARUTI into the list CARS, the sequence of steps to be undertaken are as shown in Fig. 6.12(b-d). To delete FORD from the list CARS shown in Fig. 6.13(a) the sequence of steps to be undertaken are shown in Fig. 6.13(b-d).

Primitive operations on circularly linked lists

Some of the important primitive operations executed on a circularly linked list are detailed below. Here P is a circularly linked list as illustrated in Fig. 6.14(a).

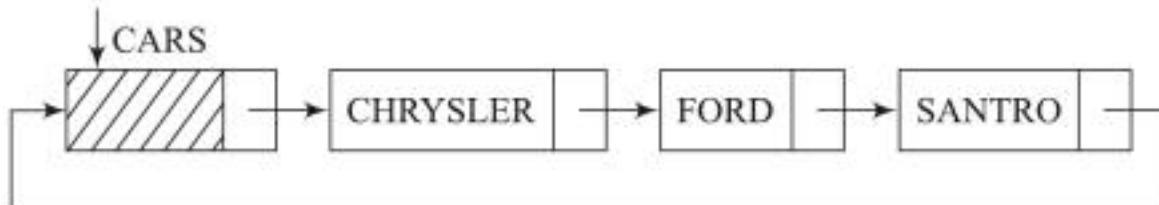
- Insert an element A as the left most element in the list represented by P .
The sequence of operations to execute the insertion is:

```

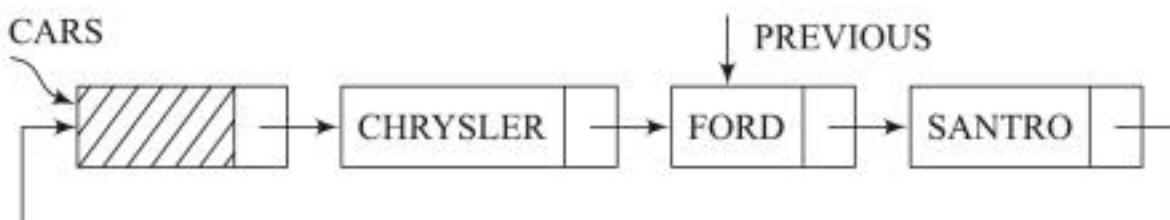
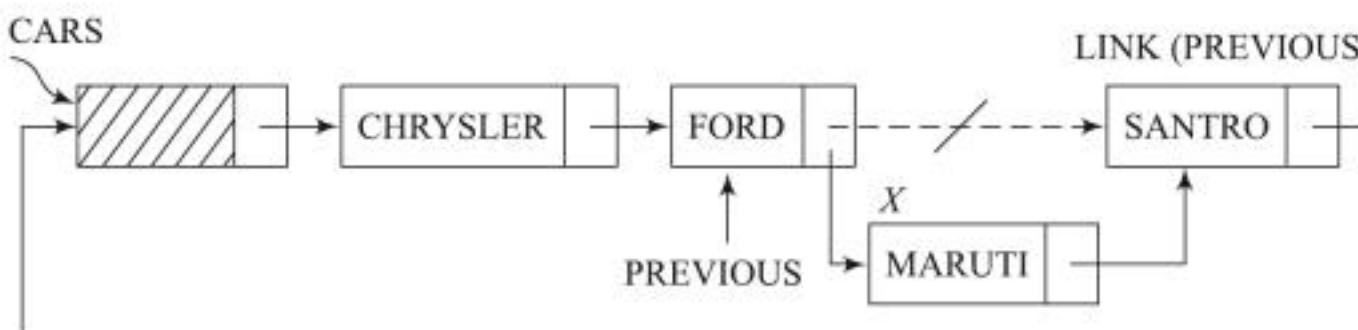
Call GETNODE (X);
      DATA (X) = A;
      LINK (X) = LINK (P);
      LINK (P) = X;
    
```

Figure 6.14(b) illustrates the insertion of A as the left most element in the circular list P .

- Insert an element A as the right most element in the list represented by P .



(a) The headed circularly linked list CARS

(b) Get new node X and store 'MARUTI' into it(c) Obtain the address of the preceding node (PREVIOUS) to insert node X into the list CARS

(d) Set / Reset links to insert MARUTI into the list CARS

$\text{LINK}(X) = \text{LINK}(\text{PREVIOUS})$
 $\text{LINK}(\text{PREVIOUS}) = X$

Fig. 6.12 Insertion of MARUTI into the headed circularly linked list CARS

The sequence of operations to execute the insertion are the same as that of inserting A as the left most element in the list followed by the instruction.

$P = X$

Figure 6.14(c) illustrates the insertion of A as the right most element in list P .

(iii) Set Y to the data of the left most node in the list P and delete the node.

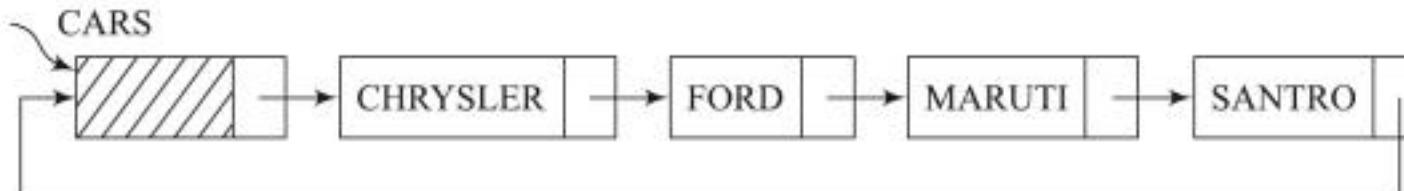
The sequence of operations to execute the deletion are:

```

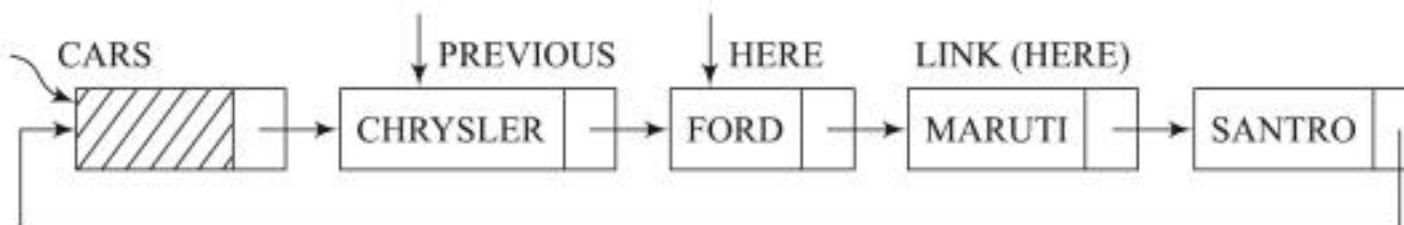
PTR = LINK(P);
Y = DATA(PTR);
LINK(P) = LINK(PTR);
Call RETURN(PTR);
    
```

Here PTR is a temporary pointer variable. Figure 6.14(d) illustrates the deletion of the left most node in the list P , setting Y to its data.

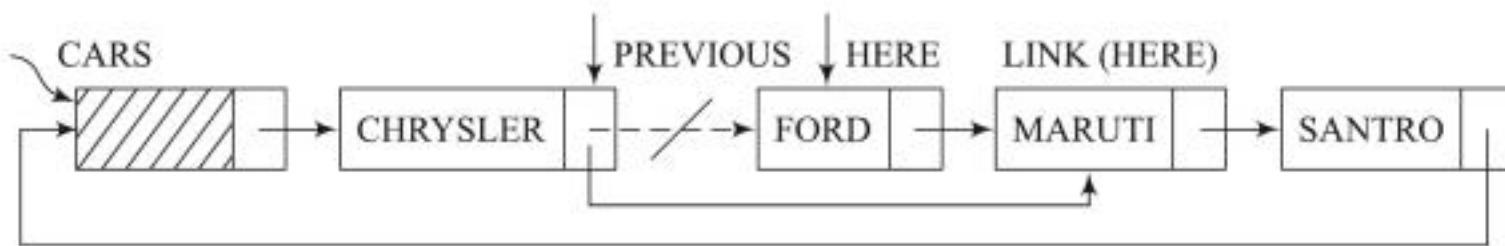
Observe that the primitive operations (i) and (iii) when combined, results in the circularly linked list working as a stack and operations (ii) and (iii) when combined, results in the circularly linked list working as a queue.



(a) The headed circularly linked list CARS

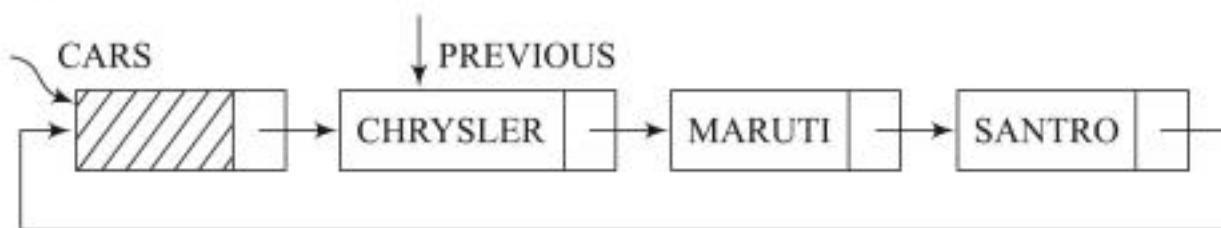


(b) Obtain the address of the node containing FORD (node HERE) by searching for it in the list CARS and its predecessor node (node PREVIOUS)



(c) Reset links to delete FORD

LINK (PREVIOUS) = LINK (HERE)



(d) Dispose node HERE

RETURN (HERE)

Fig. 6.13 Deletion of FORD from the headed circularly linked list CARS

Other operations on circularly linked lists

The concatenation of two circularly linked lists L_1, L_2 as illustrated in Fig. 6.15 has the following sequence of instructions.

```

if  $L_1 \neq \text{NIL}$  then
    { if  $L_2 \neq \text{NIL}$  then
        ( $\text{TEMP} = \text{LINK } (L_1)$ 
          $\text{LINK}(L_1) = \text{LINK}(L_2)$ 
          $\text{LINK}(L_2) = \text{TEMP}$ 
          $L_1 = L_2$ )
    
```

The other operations are splitting a list into two parts (Programming Assignment P6.2) and erasing a list.

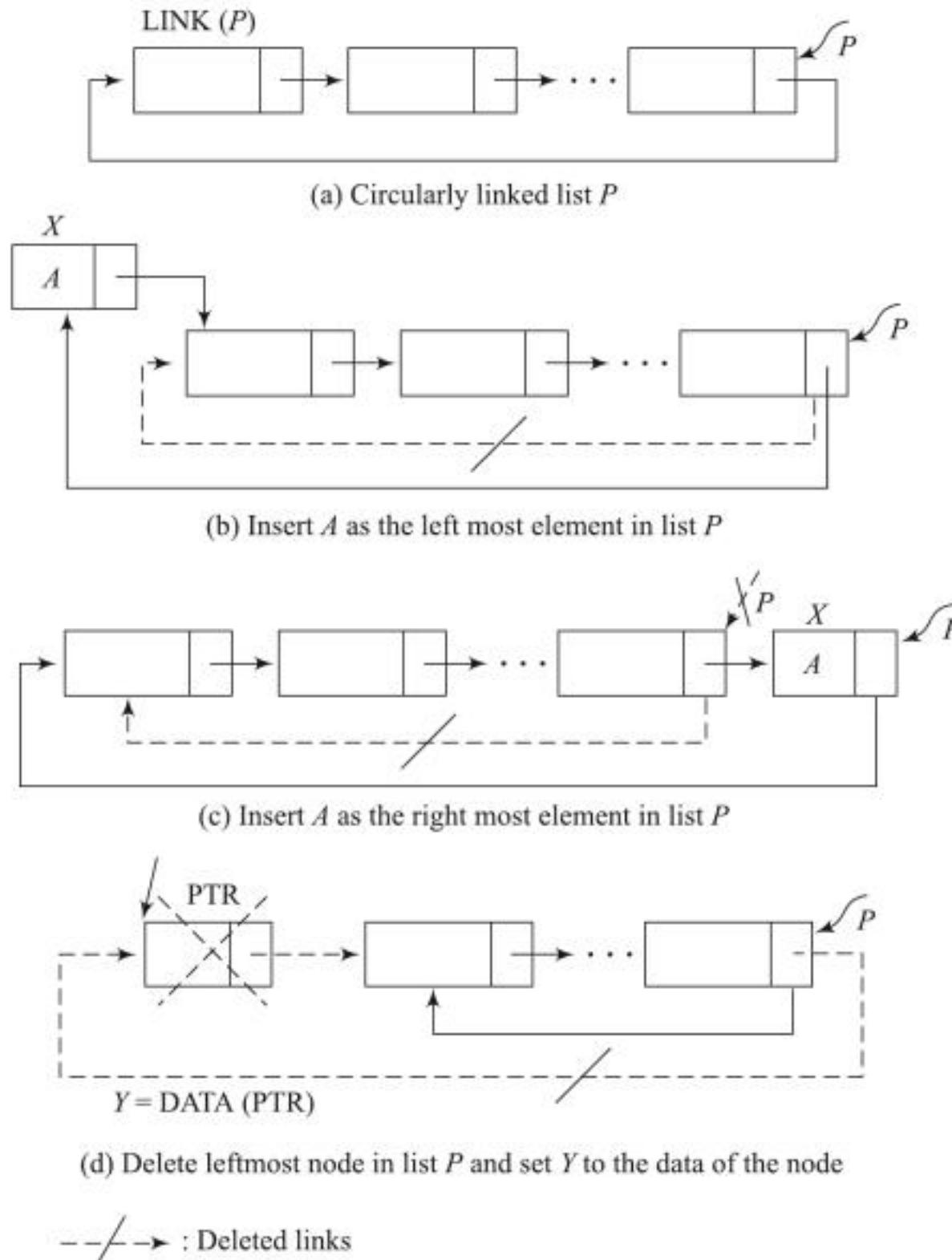


Fig. 6.14 Some primitive operations on a circularly linked list P

Doubly Linked Lists

6.4

In Secs 6.2 and 6.3 we discussed two types of linked representations viz., singly linked list and circularly linked list, both making use of a single link. Also, the circularly linked list served to rectify the drawbacks of the singly linked list. To enhance greater flexibility of movement, the linked representation could include two links in every node, each of which points to the nodes on either side of the given node. Such a linked representation known as *doubly linked list* is discussed in this section.

Representation of a doubly linked list

A *doubly linked list* is a linked linear data structure, each node of which has one or more data

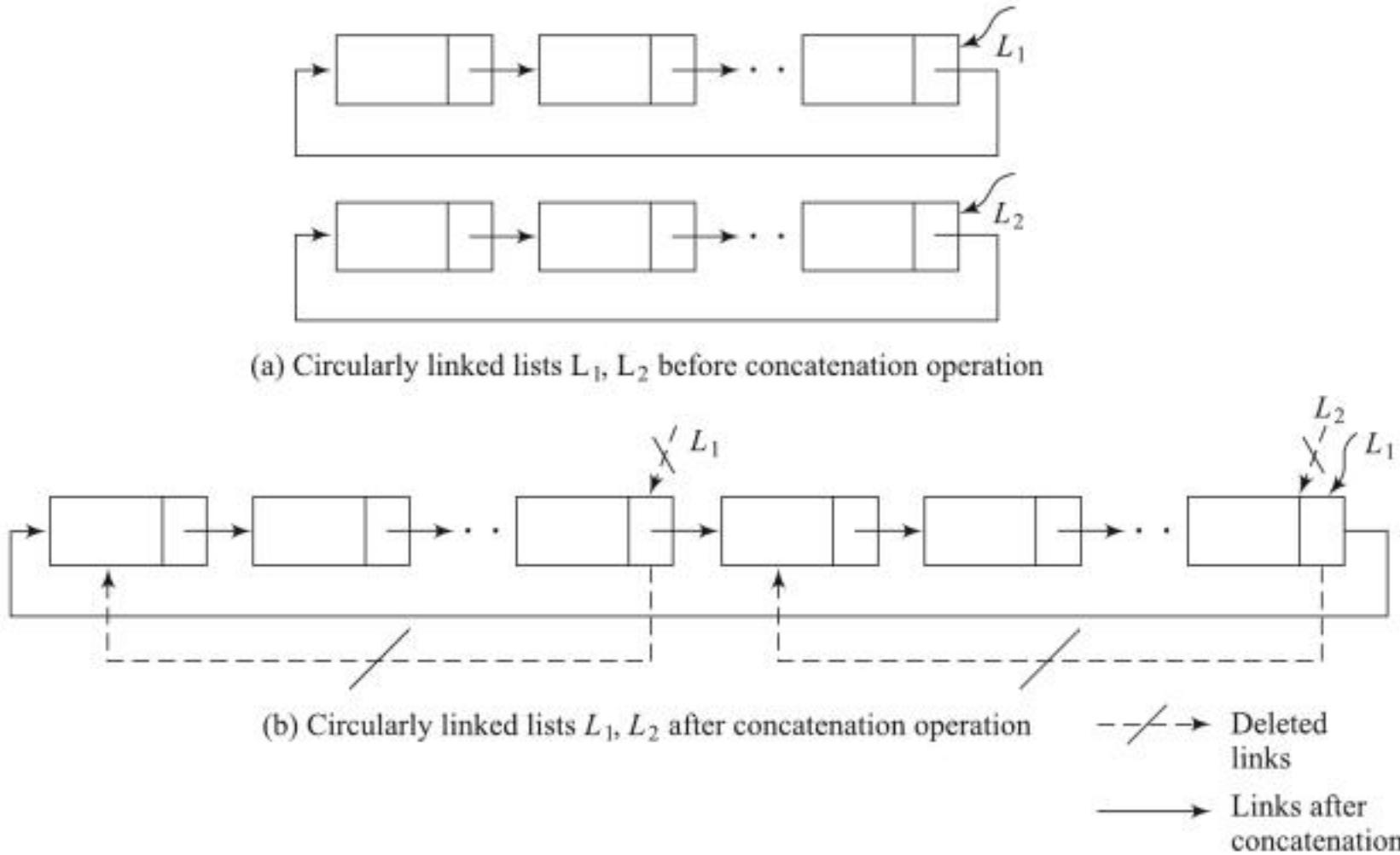


Fig. 6.15 Concatenation of two circularly linked lists

fields but only two link fields termed *left link* (LLINK) and *right link* (RLINK). The LLINK field of a given node points to the node on its left and its RLINK field points to the one on its right. A doubly linked list may or may not have a head node. Again, it may or may not be circular.

Figure 6.16 illustrates the structure of a node in a doubly linked list and the various types of lists.

Example 6.4 illustrates a doubly linked list and its logical and physical representations.

Example 6.4 Consider a list FLOWERS of four data elements LOTUS, CHRYSANTHEMUM, LILY and TULIP stored as a circular doubly linked list with a head node. The logical and physical representation of FLOWERS has been illustrated in Fig. 6.17 (a-b). Observe how the LLINK and RLINK fields store the addresses of the predecessors and successors of the given node respectively. In the case of FLOWERS being an empty list, the representation is as shown in Fig. 6.17 (c-d)

Advantages and disadvantages of a doubly linked list

Doubly linked lists have the following advantages:

- The availability of two links LLINK and RLINK permit forward and backward movement during the processing of the list.
- The deletion of a node X from the list calls only for the value X to be known. Contrast how in the case of a singly linked or circularly linked list, the delete operation necessarily needs to know the predecessor of the node to be deleted. While a singly linked list expects the predecessor of the node to be deleted, to be explicitly known, a circularly linked list is

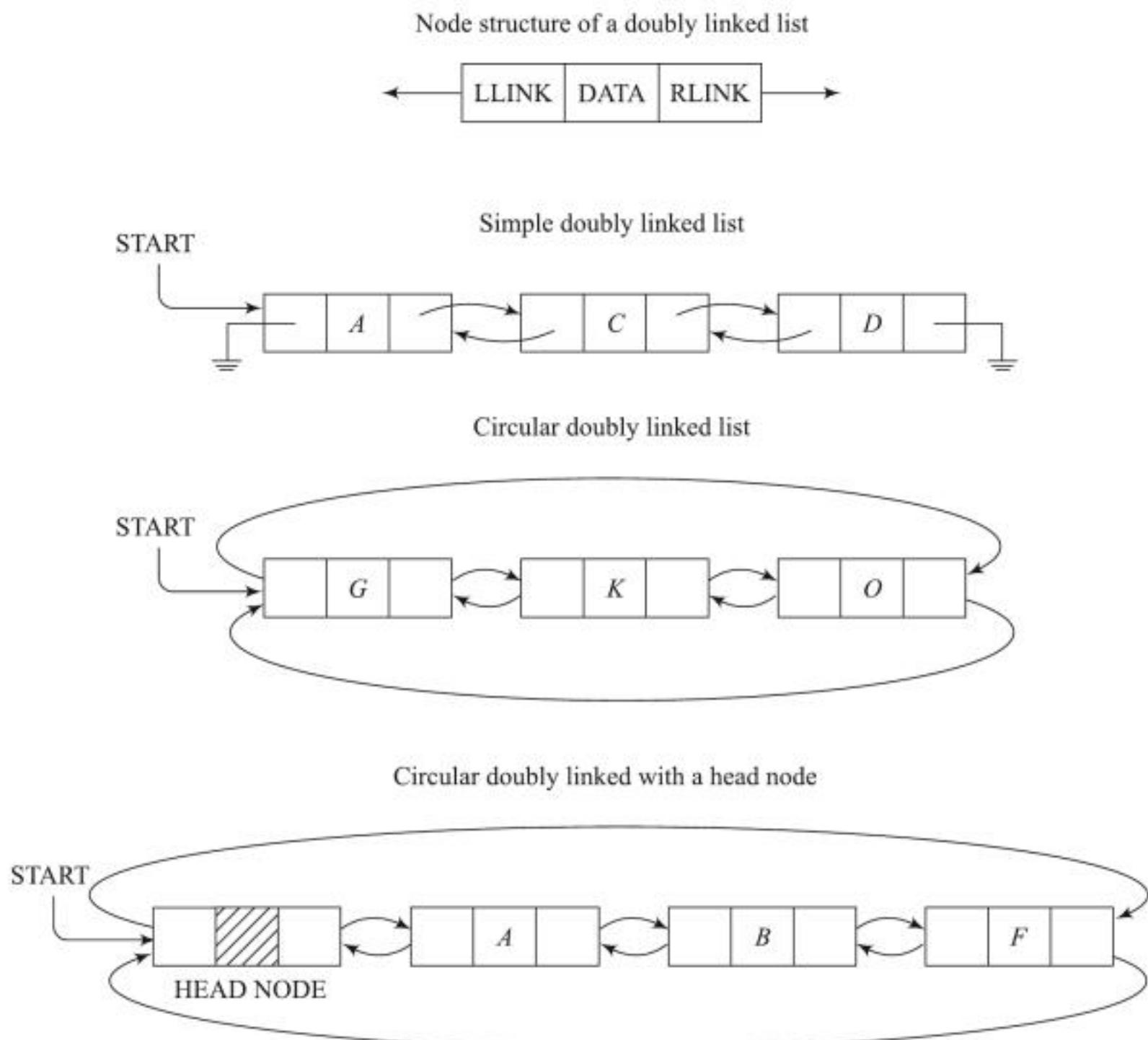


Fig. 6.16 Node structure of a doubly linked list and the various list types

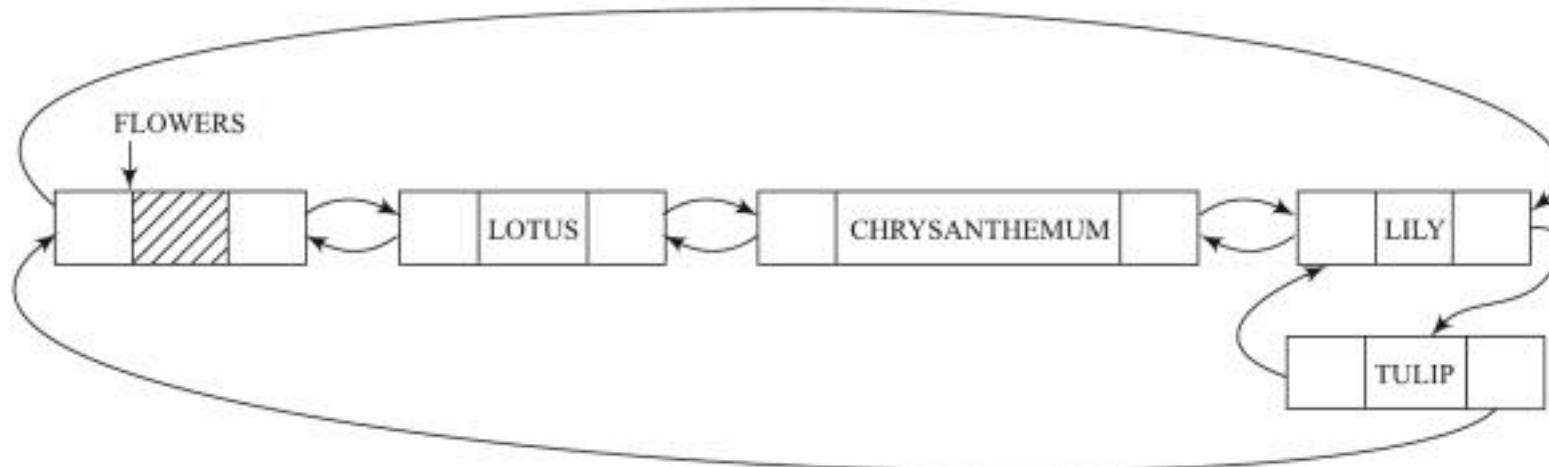
endowed with the capability to move round the list to find the predecessor node. However, in the latter case, if the list is too long it may render the delete operation inefficient.

The only disadvantage of the doubly linked list is its memory requirement. That each node needs two links could be considered expensive storage-wise, when compared to singly linked lists or circular lists. Nevertheless, the efficiency of operations due to the availability of two links more than compensate for the extra space requirement.

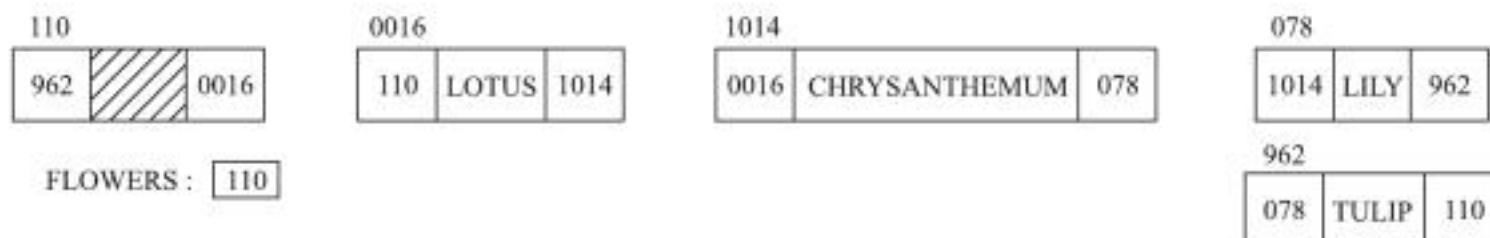
Operations on doubly linked lists

An insert and delete operation on a doubly linked list are detailed here.

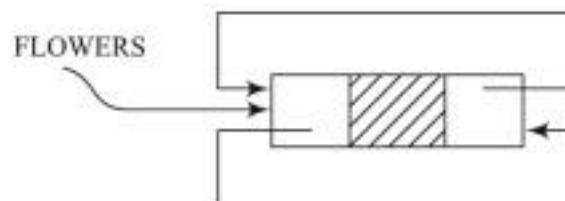
Insert Operation Let P be a headed circular doubly linked list which is non empty. Algorithm 6.4 illustrates the insertion of a node X to the right of node Y . Figure 6.18(a) shows the logical representation of list P before and after insertion.



(a) Logical representation of a circular doubly linked list with a head node FLOWERS



(b) Physical representation of a circular doubly linked list with a head node (FLOWERS)



(c) Logical representation of an empty circular doubly linked list with a head node (FLOWERS)



(d) Physical representation of an empty circular doubly linked list with a head node

Fig. 6.17 The logical and physical representation of a circular doubly linked list with a head node, FLOWERS

Algorithm 6.4: To insert node X to the right of node Y in a headed circular doubly linked list P

```

Procedure INSERT_DL (X, Y)
    LLINK (X) = Y;
    RLINK (X) = RLINK (Y);
    LLINK (RLINK (Y)) = X;
    RLINK (Y) = X;
end INSERT_DL.
```

Note how the four instructions in the Algorithm 6.4 correspond to the setting / resetting of the four link fields, viz., links pertaining to node Y, its original right neighbour ($RLINK(Y)$) and the node X.

Delete operation Let P be a headed, circular doubly linked list. Algorithm 6.5 illustrates the deletion of a node X from P . The condition ($X = P$) that is checked ensures that the head node P is not deleted. Figure 6.18(b) shows the logical representation of list P before and after the deletion of node X from the list P .

Algorithm 6.5: Delete node X from a headed circular doubly linked list P

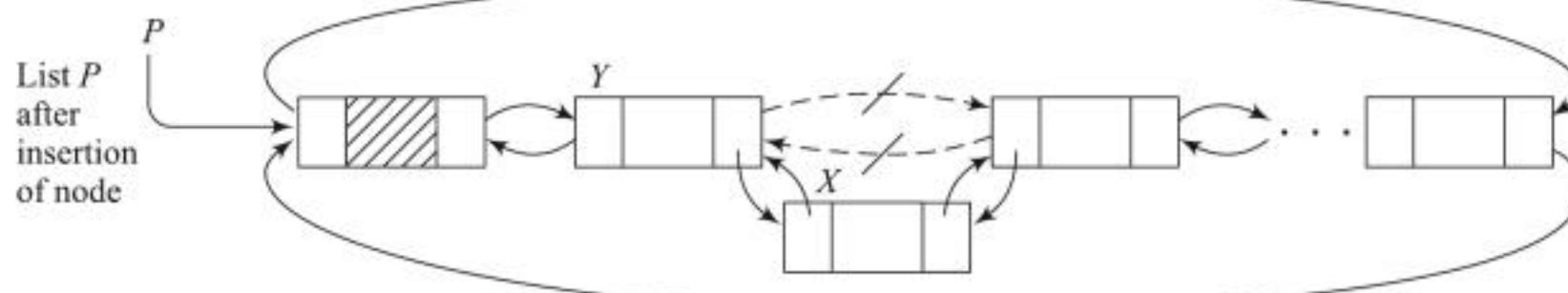
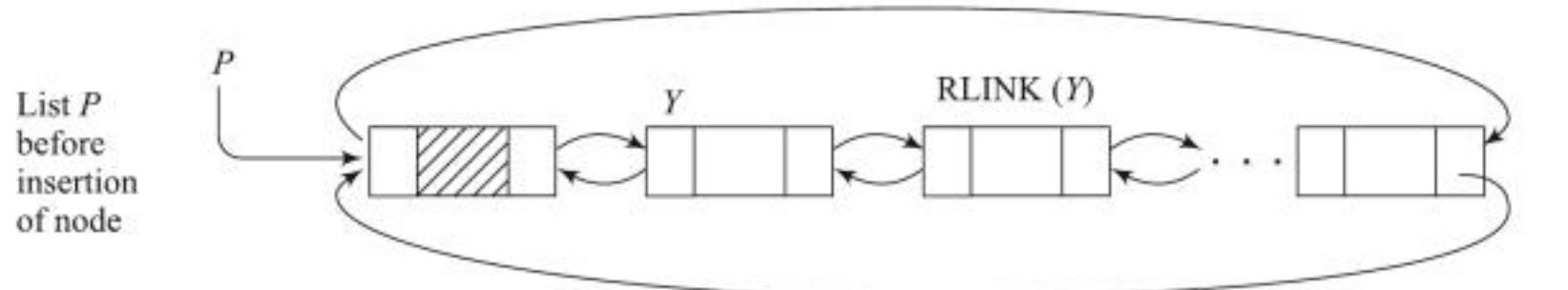
```

procedure DELETE_DL (P, X)
  if (X = P) then ABANDON_DELETE;
  else
    {RLINK(LLINK(X)) = RLINK(X);
     LLINK(RLINK(X)) = LLINK(X);
     Call RETURN (X); }
  end DELETE_DL.

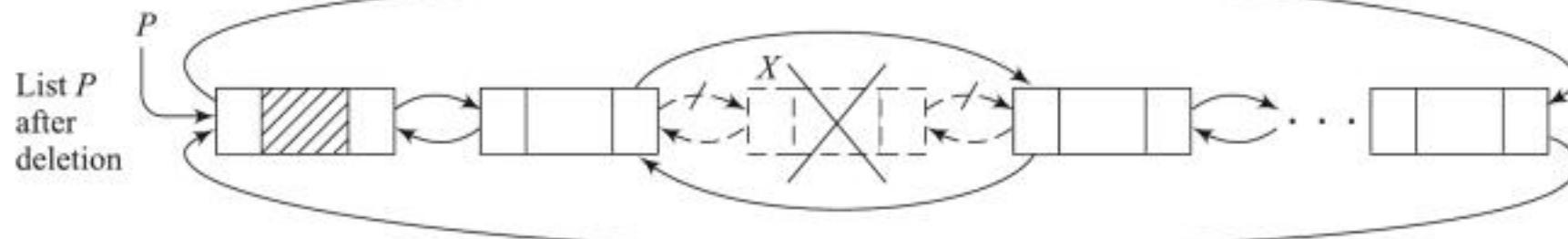
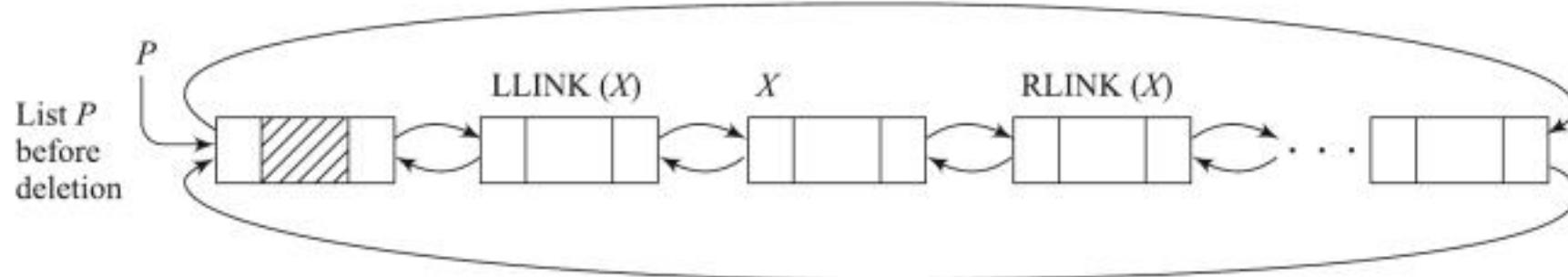
```

Note how the two instructions pertaining to links, in Algorithm 6.5, correspond to the setting / resetting of link fields of the two nodes viz. the predecessor (LLINK (X)) and successor (RLINK (X)) of node X.

Example 6.5 illustrates the insert/delete operation on a doubly linked list PLANET.



(a) Insertion of node X into a headed circular doubly linked list P , after node Y



(b) Deletion of node X from a headed circular doubly linked list P

Fig. 6.18 Insertion/deletion in a headed circular doubly linked list

Example 6.5 Let PLANET be a headed circular doubly linked list with three data elements viz. MARS, PLUTO and URANUS. Figure 6.19 illustrates the logical and physical representation of the list PLANET. Figure 6.20(a) illustrates the logical and physical representation of list PLANET after the deletion of PLUTO and Fig. 6.20(b) the same after insertion of JUPITER.

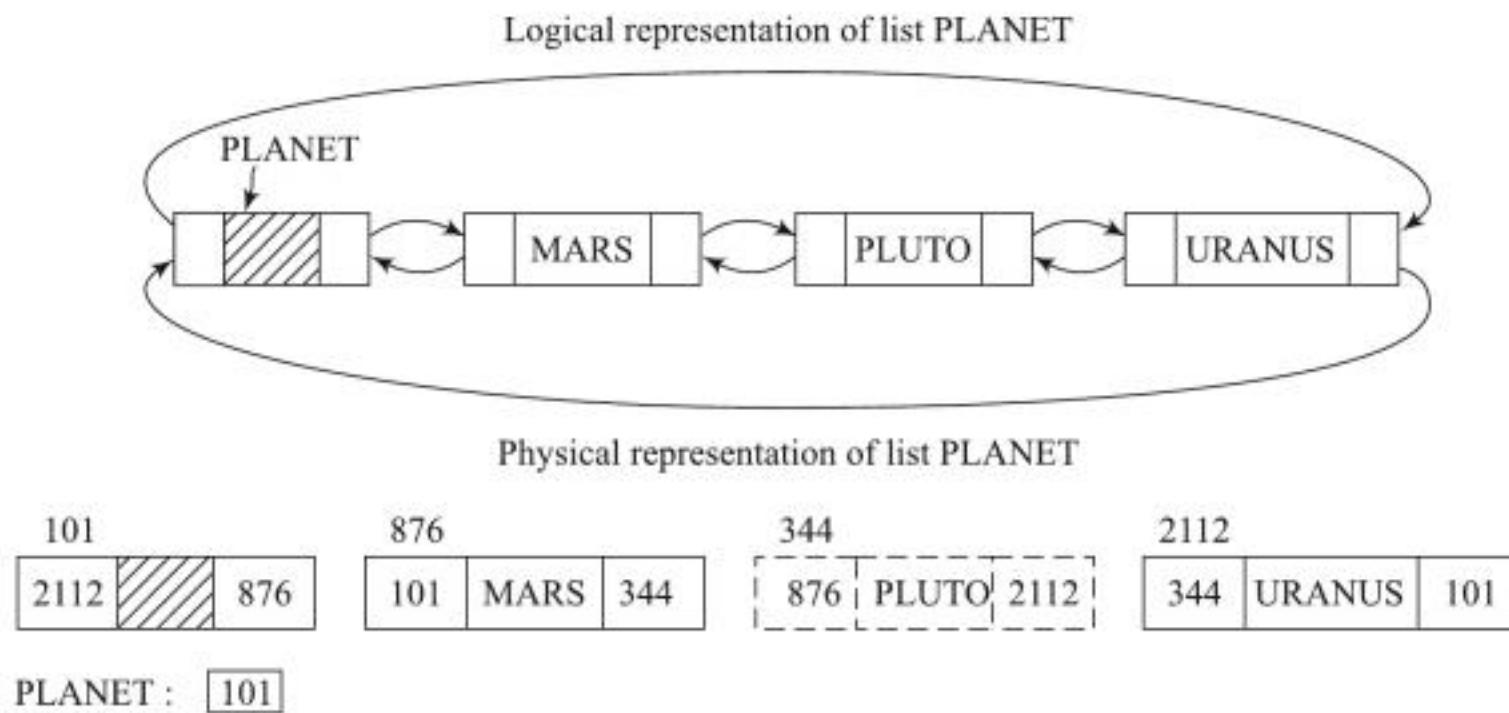


Fig. 6.19 Logical and physical representation of list PLANET

Multiply Linked Lists

6.5

A multiply linked list as its name suggests is a linked representation with multiple data and link fields. A general node structure of a multiply linked list is as shown in Fig. 6.21.

Since each link field connects a group of nodes representing the data elements of a global list L , the multiply linked representation of the list L is a network of nodes which are connected to one another based on some association. The link fields may or may not render their respective lists to be circular or may or may not possess a head node.

Example 6.6 illustrates an example of a multiply linked list.

Example 6.6 Let STUDENT be a multiply linked list representation whose node structure is as shown in Fig. 6.22. Here, SPORTS-CLUB-MEM link field links all student nodes who are members of the sports club. DEPT-ENROLL links all students enrolled with a given department and DAY-STUDENT links all students enrolled as day students.

Consider Table 6.1 illustrating details pertaining to 6 students.

Table 6.1 Student details for representation as a multiply linked list

Name of the Student	Roll #	Number of Credits Registered	Sports Club Membership	Day Student	Department
AKBAR	CS02	200	Yes	Yes	Computer Science

(Contd.)

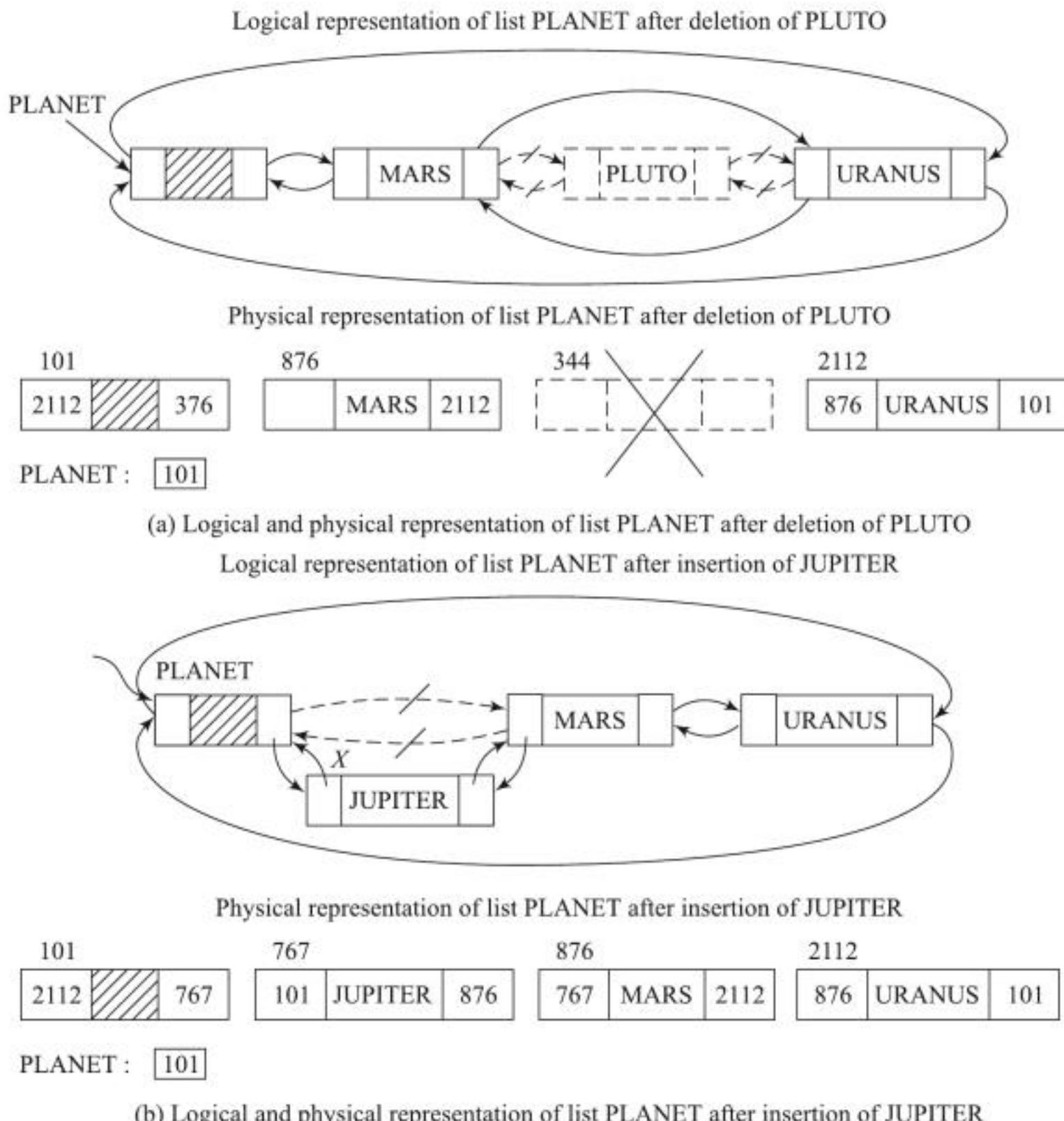


Fig. 6.20 *Deletion of PLUTO and insertion of JUPITER in list PLANET*

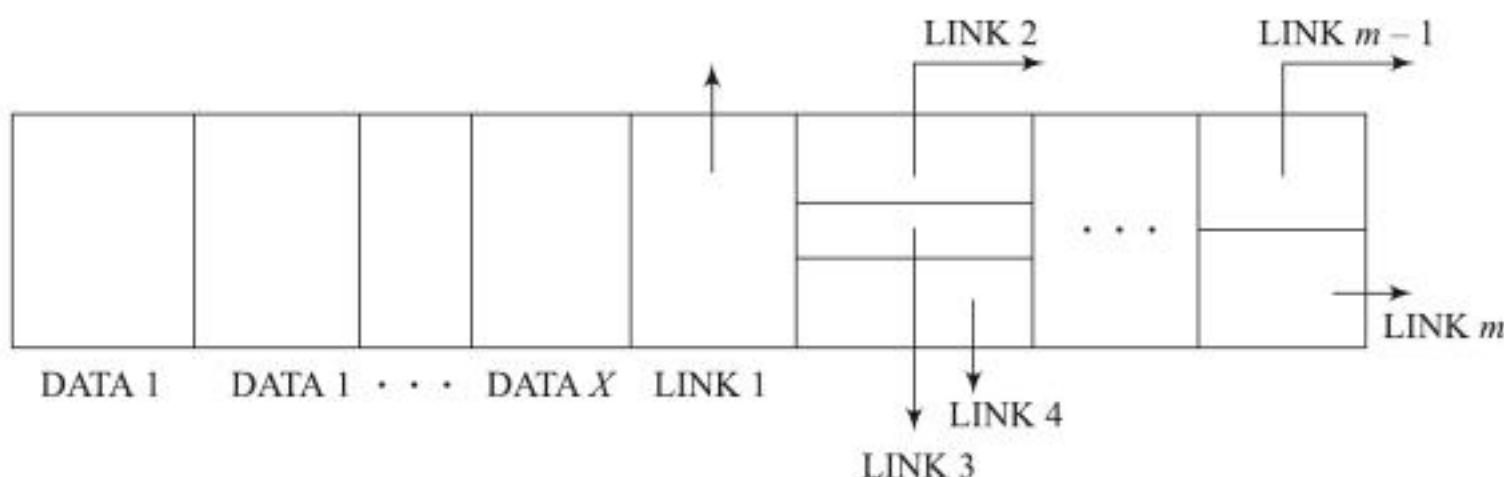


Fig. 6.21 *The node structure of a multiply linked list*

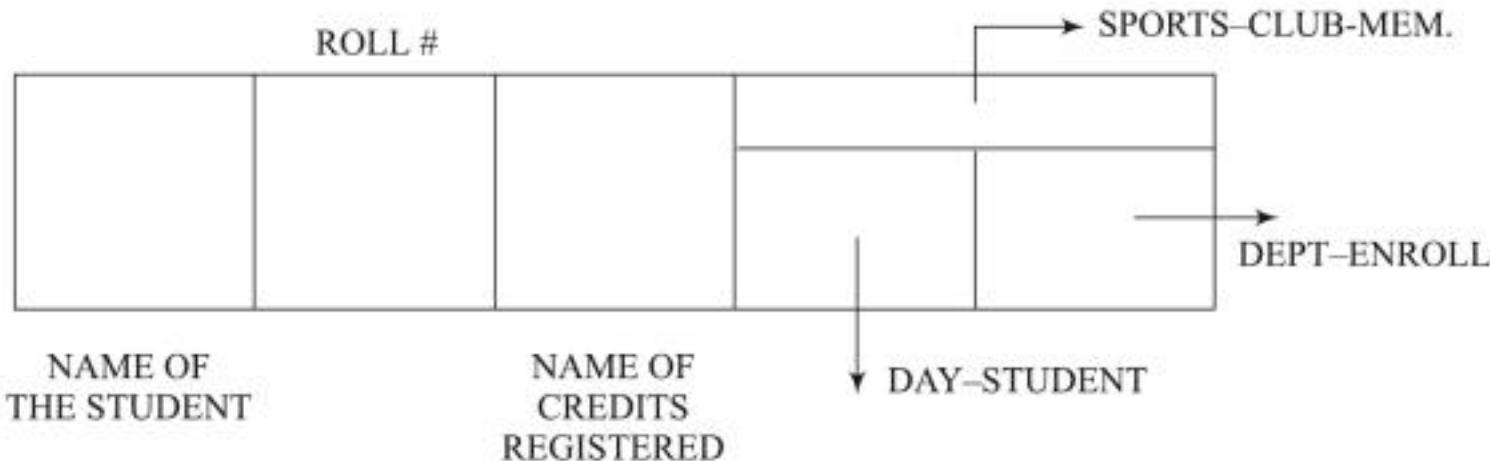


Fig. 6.22 Node structure of the multiply linked list STUDENT

(Contd.)

RAM	ME426	210	No	Yes	Mechanical Science
SINGH	ME927	210	Yes	No	Mechanical Science
YASSER	CE467	190	Yes	No	Civil Engineering
SITA	CE544	190	No	Yes	Civil Engineering
REBECCA	EC424	220	Yes	No	Electronics & Communication Engg.

The multiply linked structure of the data elements in Table 6.1 is shown in Fig. 6.23. Here S is a singly linked list of all sports club members and D the singly linked list of all day students. Note how the DEPT-ENROLL link field maintains individual singly linked lists COMP-SC, MECH-SC, CIVIL-ENGG and ECE to keep track of the students enrolled with the respective departments. To insert a new node with the following details,

ALI	CS108	200	Yes	Yes	Computer Science
-----	-------	-----	-----	-----	------------------

into the list STUDENTS, the procedure is similar to that of insertion in singly linked lists. The point of insertion is to be determined by the user. The resultant list is shown in Fig. 6.24. Here we have inserted ALI in the alphabetical order of students enrolled with the computer science department.

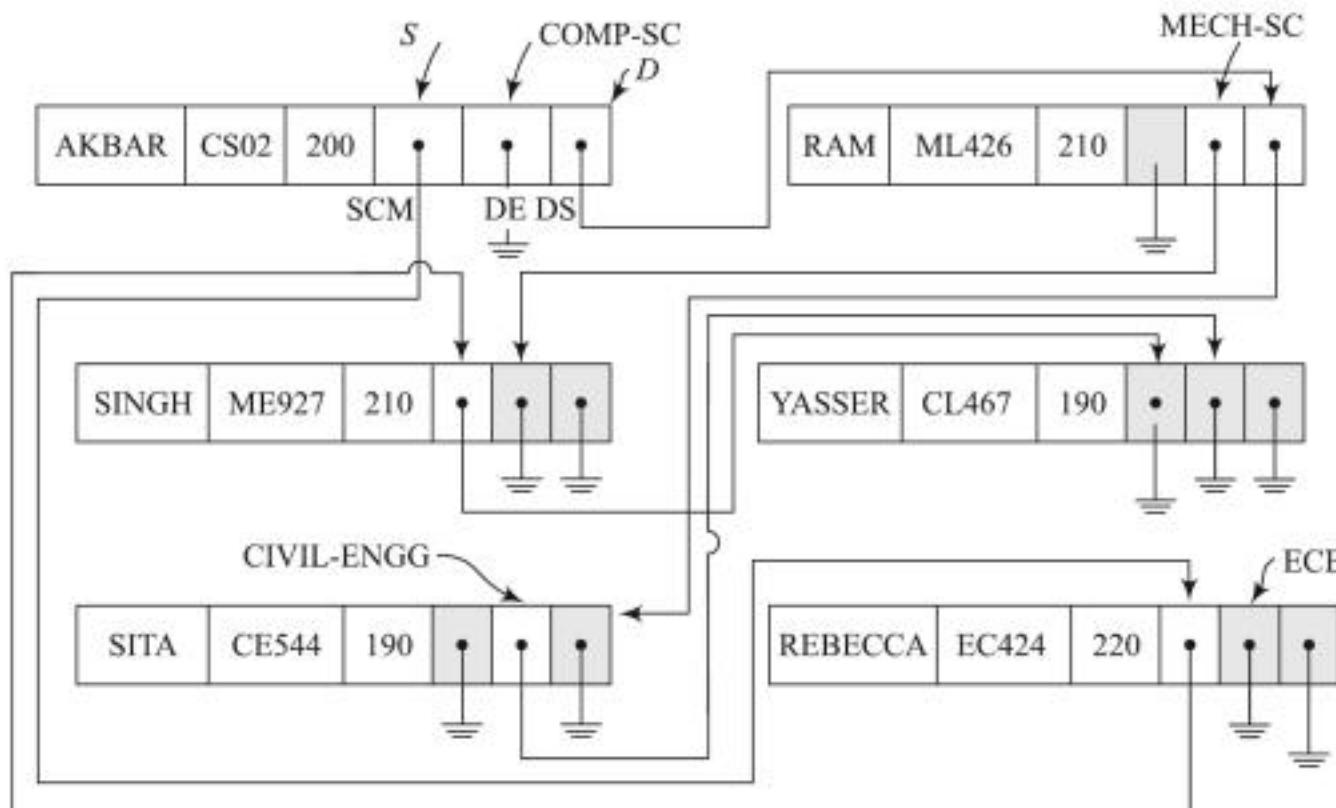
To delete REBECCA from the list of sports club members of the multiply linked list STUDENT, we undertake a sequence of operations as shown in Fig. 6.25. Observe how the node for REBECCA continues to participate in the other lists despite its deletion from the list S .

A multiply linked list can be designed to accommodate a lot of flexibility with respect to its links depending on the needs and suitability of the application.

Applications

6.6

In this section we discuss two applications of linked lists viz.,



SCM: SPORTS-CLUB-MEM link field

DE: DEPT ENROLL link field

DS: DAY-STUDENT link field

S: List of sports club members

D: List of day students

Comp. SC: List of students enrolled with the Dept. of Computer Science

CIVIL-ENGG: List of students enrolled with the Dept. of Civil Engg.

MECH-SC: List of students enrolled with the Dept. of Mech Sc.

ECE: List of students enrolled with the Dept. of E.C.E.

Fig. 6.23 Multiply linked list structure of list STUDENT

- Addition of polynomials and
- Representation of a sparse matrix

Addition of polynomials is illustrative of application of singly linked lists and sparse matrix representation that of multiply linked lists.

Addition of polynomials

The objective of this application is to perform a symbolic addition of two polynomials as illustrated below:

Let $P_1 : 2x^6 + x^3 + 5x + 4$ and

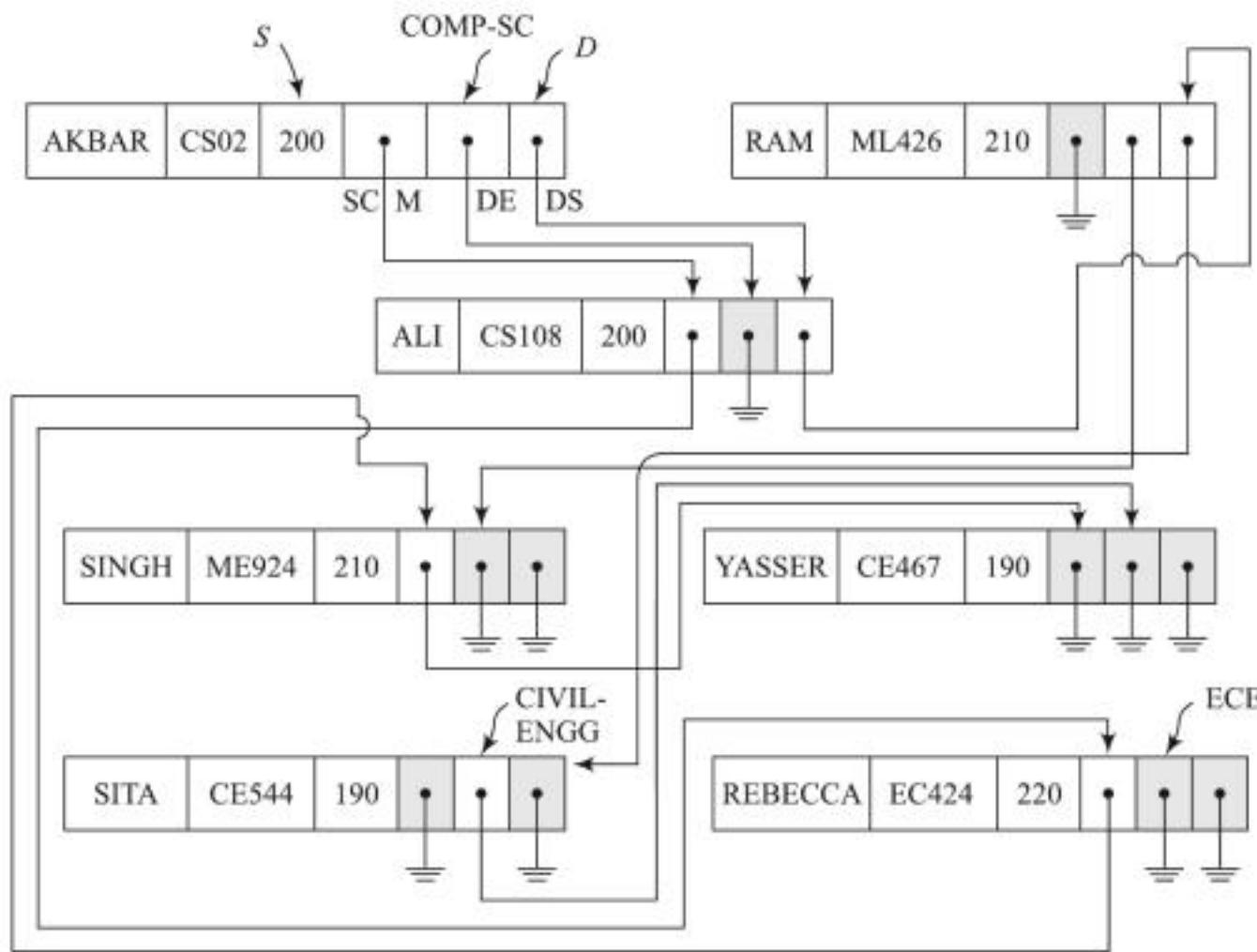
$P_2 : 7x^6 + 8x^5 - 9x^3 + 10x^2 + 14$

be two polynomials over a variable x . The objective is to obtain the algebraic sum of P_1, P_2 (i.e.) $P_1 + P_2$ as,

$$P_1 + P_2 = 9x^6 + 8x^5 - 8x^3 + 10x^2 + 5x + 18$$

To perform this symbolic manipulation of the polynomials, we make use of a singly linked list to represent each polynomial. The node structure and the singly linked list representation for the two polynomials are given in Fig. 6.26. Here each node in the singly linked list represents a term of the polynomial.

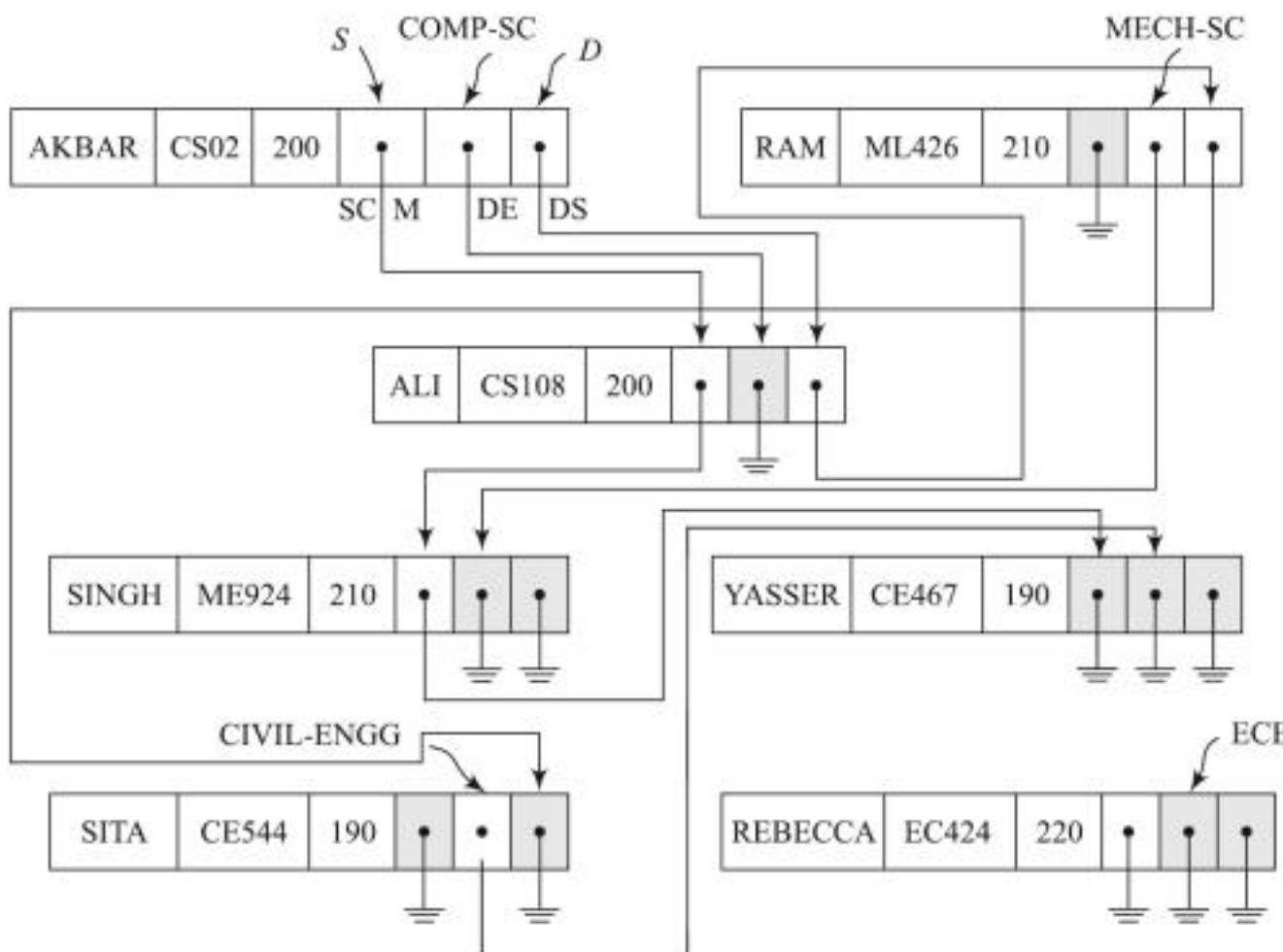
To add the two polynomials, we presume that the singly linked lists have their nodes arranged in the decreasing order of the exponents of the variable x .



SCM: SPORTS-CLUB-MEM link field

DE: DEPT-ENROLL link field

DS: DAY-STUDENT link field

Fig. 6.24 Insert ALI into the multiply linked list STUDENT**Fig. 6.25 Delete REBECCA from the sports club membership list of the multiply linked list STUDENTS**

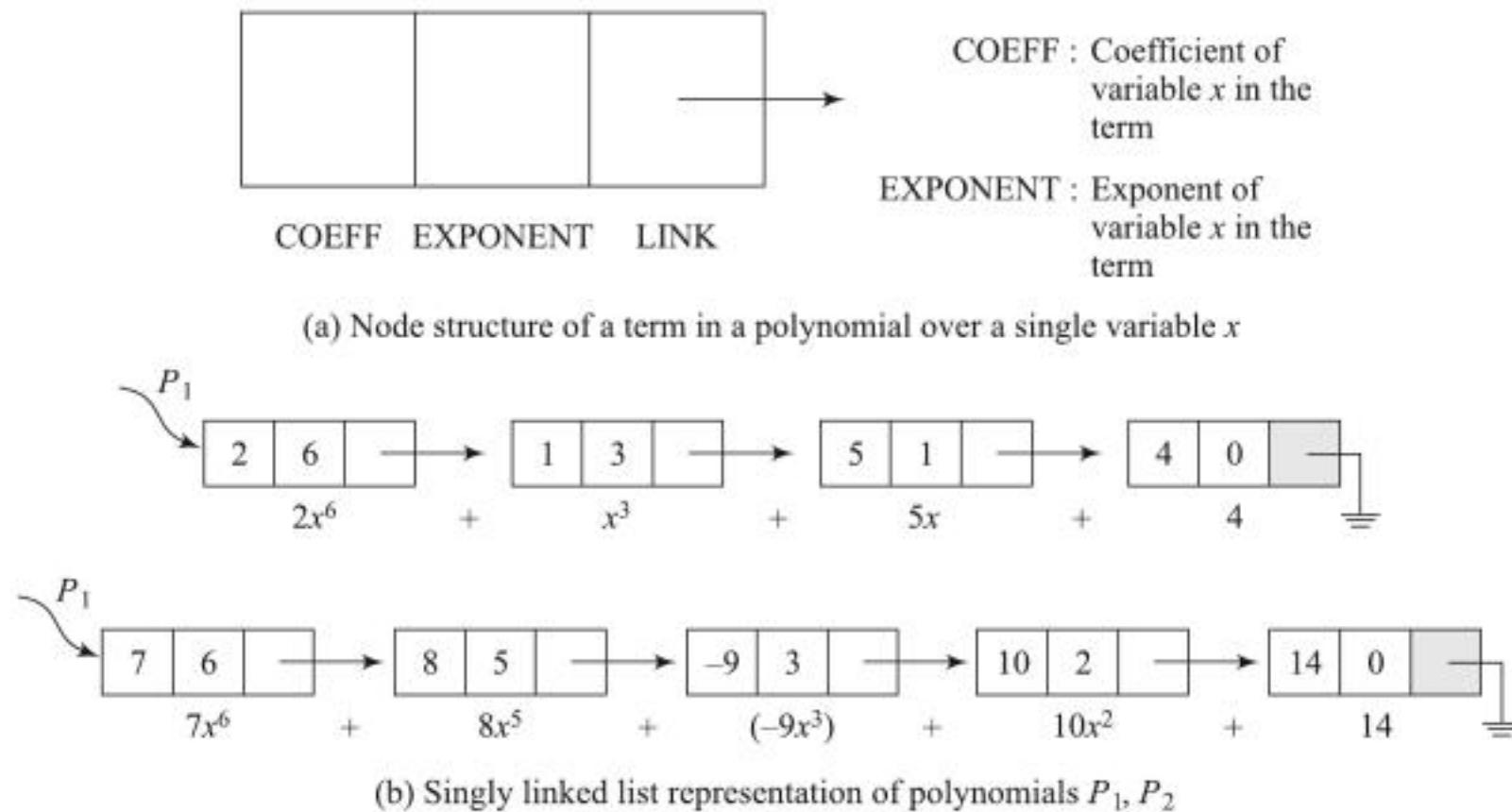


Fig. 6.26 Addition of polynomials—Node structure and singly linked list representation of polynomials

The objective is to create a new list of nodes representing the sum $P_1 + P_2$. This is achieved by adding the COEFF fields of the nodes of like powers of variable x in lists P_1, P_2 and adding a new node reflecting this operation in the resultant list $P_1 + P_2$. We present below, the crux of the procedure:

Here P_1, P_2 are the start pointers of the singly linked lists representing the polynomials P_1, P_2 . Also PTR1 and PTR2 are two temporary pointers initially set to P_1 and P_2 respectively.

```

if (EXPONENT(PTR1) = EXPONENT(PTR2)) then /* PTR1 and PTR2
    point to like terms */
if (COEFF(PTR1) + COEFF(PTR2)) ≠ 0 then
    {Call GETNODE(X); /* Perform the addition of terms and
        include the result node as the last
        node of list  $P_1 + P_2$  */
    COEFF(X) = COEFF(PTR1) + COEFF(PTR2);
    EXPONENT(X) = EXPONENT(PTR1); /*or EXPONENT(PTR2)*/
    LINK(X) = NIL;
    Add node X as the last node of the list  $P_1 + P_2$ ;
}
if (EXPONENT(PTR1) < EXPONENT(PTR2)) then
    /* PTR1 and PTR2 do not point to like terms */
    /* Duplicate the node representing the highest
        power(i.e.) EXPONENT(PTR2) and insert it as
        the last node in  $P_1 + P_2$  */
    {Call GETNODE(X);
    COEFF(X) = COEFF(PTR2);
    EXPONENT(X) = EXPONENT(PTR2);
}

```

```

LINK (X) = NIL;
Add node X as the last node of list  $P_1 + P_2$  ;
}

if (EXPONENT (PTR1) > EXPONENT (PTR2)) then
/* PTR1 and PTR2 do not point to like terms. Hence duplicate
the node representing the highest power (i.e.) EXPONENT
(PTR1) and insert it as the last node of  $P_1 + P_2$  */
( Call GETNODE (X);
COEFF (X) = COEFF (PTR1);
EXPONENT (X) = EXPONENT (PTR1);
LINK(X) = NIL;
Add node X as the last node of list  $P_1 + P_2$ ;
)

```

If any one of the lists during the course of addition of terms has exhausted its nodes earlier than the other list, then the nodes of the other list are simply appended to list $P_1 + P_2$ in the order of their occurrence in their original list.

In case of polynomials of two variables x, y or three variables x, y, z the node structures are as shown in Fig. 6.27.

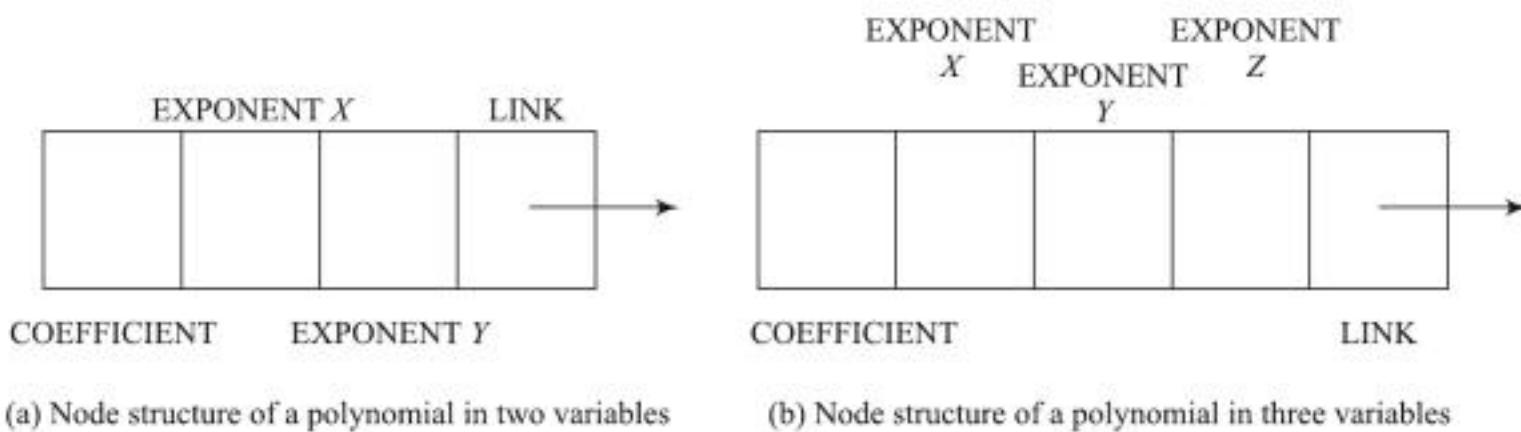


Fig. 6.27 Node structures of polynomials in two/three variables

Here COEFFICIENT refers to the coefficient of the term in the polynomial represented by the node. EXPONENT X, EXPONENT Y and EXPONENT Z are the exponents of the variables x, y and z respectively.

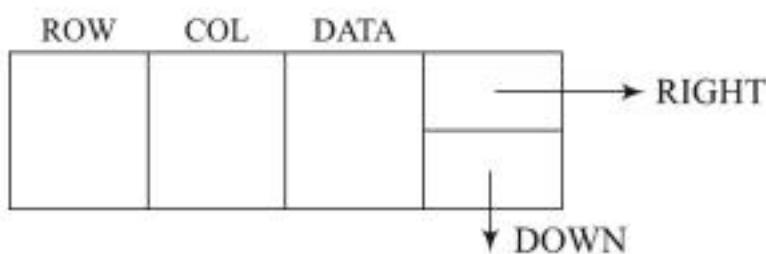
Sparse matrix representation

The concept of sparse matrices was discussed in Chapter 3. An array representation for the efficient representation and manipulation of sparse matrices was suggested in Sec. 3.5. In this section we present a linked representation for the sparse matrix, as an illustration of multiply linked list.

Consider a sparse matrix shown in Fig. 6.28(a). The node structure for the linked representation of the sparse matrix is shown in Fig. 6.28(b). Each non-zero element of the matrix is represented using the node structure. Here ROW, COL and DATA fields record the row, column and value of the non-zero element in the matrix. The RIGHT link points to the node

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ -2 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

(a) Sparse matrix



(b) Node structure of the multiply linked list

Fig. 6.28 A sparse matrix and the node structure for its representation as a multiply linked list

holding the next non-zero value in the same row of the matrix. DOWN link points to the node holding the next non-zero value in the same column of the matrix. Thus, each non-zero value is linked to its row wise and column wise non-zero neighbour. Thus, the linked representation ignores representing the zeros in the matrix. Now each of the fields connect together to form a singly linked list with a head node. Thus all the nodes representing non-zero elements of a row in the matrix link themselves (through RIGHT LINK) to form a singly linked list with a head node. The number of such lists is equal to the number of rows in the matrix which contain at least one non-zero element. Similarly, all the nodes representing the non-zero elements of a column in the matrix link themselves (through DOWN LINK) to form a singly linked list with a head node. The number of such lists is equal to the number of columns in the matrix which contain at least one non-zero element. All the head nodes are also linked together to form a singly linked list. The head nodes of the row lists have their COL fields to be zero and the head nodes of the column lists have their ROW fields to be zero. The head node of all head nodes, indicated by START, stores the dimension of the original matrix in its ROW, COL fields. Figure 6.29 shows the multiply linked list representation of the sparse matrix shown in Fig. 6.28(a).

ADT for Links

Data objects:

Addresses of the nodes holding data and null links

Operations:

- Allocate node (address X) from Available Space to accommodate data
GETNODE (X)
 - Return node (address X) after use to Available Space RETURN(X)
 - Store a value of one link variable LINK1 to another link variable LINK2
STORE_LINK (LINK1, LINK2)
 - Store ITEM into a node whose address is X
STORE_DATA (X, ITEM)
 - Retrieve ITEM from a node whose address is X
RETRIEVE_DATA (X, ITEM)

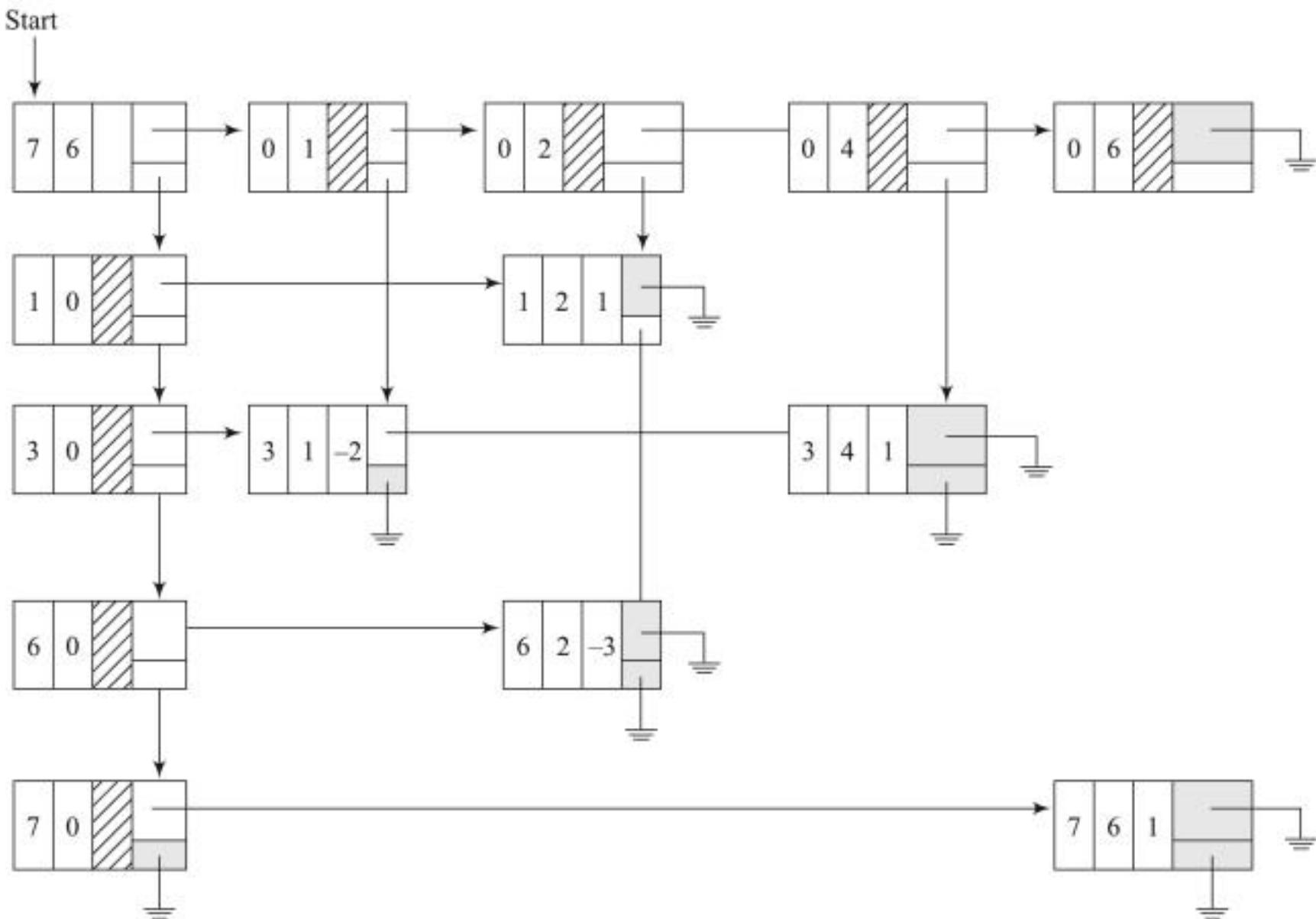


Fig. 6.29 Multiply linked representation of the sparse matrix shown in Fig. 6.28(a)

ADT for Singly Linked Lists

Data objects:

A list of nodes each holding one (or more) data field(s) DATA and a single link field LINK. LIST points to the start node of the list.

Operations:

- Check if list LIST is empty
CHECK_LIST_EMPTY (LIST) (Boolean function)
- Insert ITEM into the list LIST as the first element
INSERT_FIRST (LIST, ITEM)
- Insert ITEM into the list LIST as the last element
INSERT_LAST (LIST, ITEM)
- Insert ITEM into the list LIST in order
INSERT_ORDER (LIST, ITEM)
- Delete the first node from the list LIST
DELETE_FIRST (LIST)
- Delete the last node from the list LIST
DELETE_LAST (LIST)

- Delete ITEM from the list LIST
 DELETE_ELEMENT (LIST, ITEM)
- Advance Link to traverse down the list
 ADVANCE_LINK (LINK)
- Store ITEM into a node whose address is X
 STORE_DATA (X, ITEM)
- Retrieve data of a node whose address is X and return it in ITEM
 RETRIEVE_DATA (X, ITEM)
- Retrieve link of a node whose address is X and return the value in LINK1
 RETRIEVE_LINK (X, LINK1)



Summary

- Sequential data structures suffer from the draw backs of inefficient implementation of insert/delete operations and inefficient use of memory.
- A linked representation serves to rectify these drawbacks. However, it calls for the implementation of mechanisms such as GETNODE(X) and RETURN(X) to reserve a node for use and return the same to the free pool after use, respectively.
- A singly linked list is the simplest of a linked representation with one or more data fields but with a single link field in its node structure that points to its successor. However such a list has lesser flexibility and does not aid in an elegant performance of operation such as deletion.
- A circularly linked list is an enhancement of the singly linked list representation, in that the nodes are circularly linked. This not only provides better flexibility, but also results in a better rendering of the delete operation.
- A doubly linked list has one or more data items fields but two links LLINK and RLINK pointing to the predecessor and successor of the node respectively. Though the list exhibits the advantages of greater flexibility and efficient delete operation, it suffers from the drawback of increased storage requirement for the node structure in comparison to other linked representations.
- A multiply linked list is a linked representation with one or more data item fields and multiple link fields. A multiply linked list in its simplest form may represent a cluster of singly linked lists networked together.
- The application of linked lists has been demonstrated on two problems viz., Addition of Polynomials and linked representation of a sparse matrix.



Illustrative Problems

Problem 6.1 Write a pseudocode procedure to insert NEW_DATA as the first element in a singly linked list T .

Solution: We shall write a general procedure which will take care of the cases,

- (i) T is initially empty
- (ii) T is non empty

The logical representation of the list T before and after insertion of NEW_DATA, for the two cases listed above are shown in Fig. I 6.1.

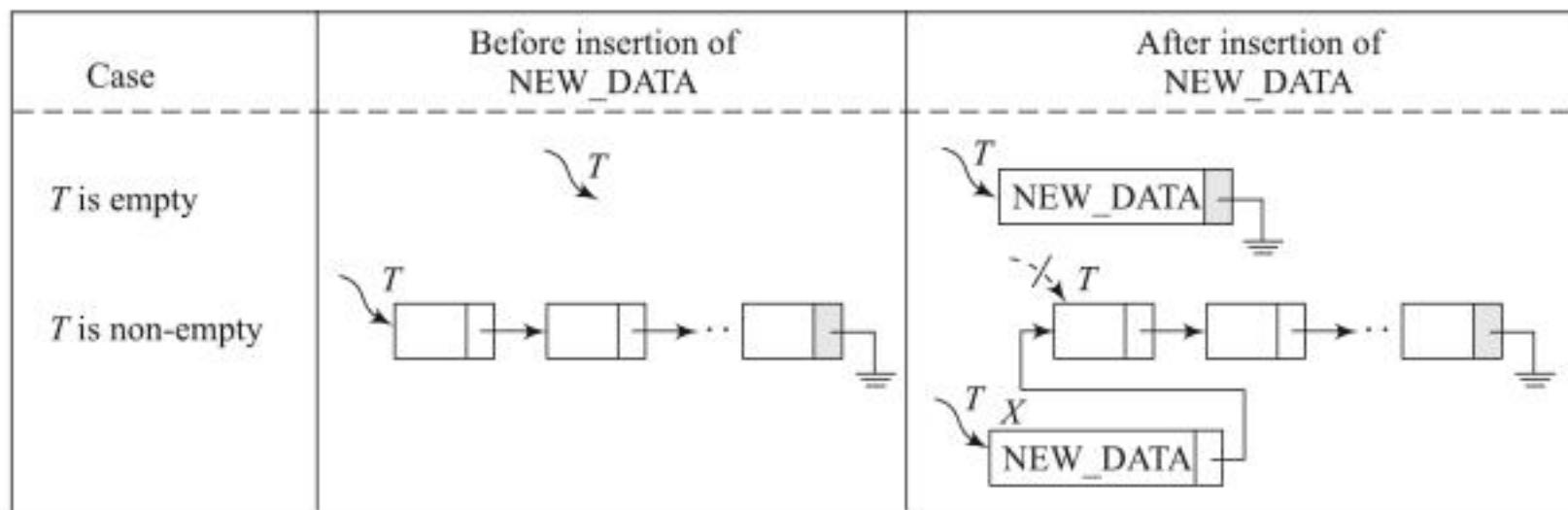


Fig. I 6.1

The general procedure in pseudocode:

```

Procedure INSERT_SL_FIRST (T, NEW_DATA)
    Call GETNODE (X);
    DATA (X) = NEW_DATA;
    if (T = NIL) then { LINK (X) = NIL; }
    else {LINK (X) = T;
           T = X; }
end INSERT_SL_FIRST.

```

Problem 6.2 Write a pseudocode procedure to insert NEW_DATA as the k^{th} element ($k > 1$) in a non empty singly linked list T .

Solution: The logical representation of the list T before and after insertion of NEW_DATA as the k^{th} element in the list is shown in Fig. I 6.2.

The pseudocode procedure is:

```

Procedure INSERT_SL_K (T, k, NEW_DATA)
    Call GETNODE (X);
    DATA (X) = NEW_DATA;
    COUNT = 1;
    TEMP = T;

```

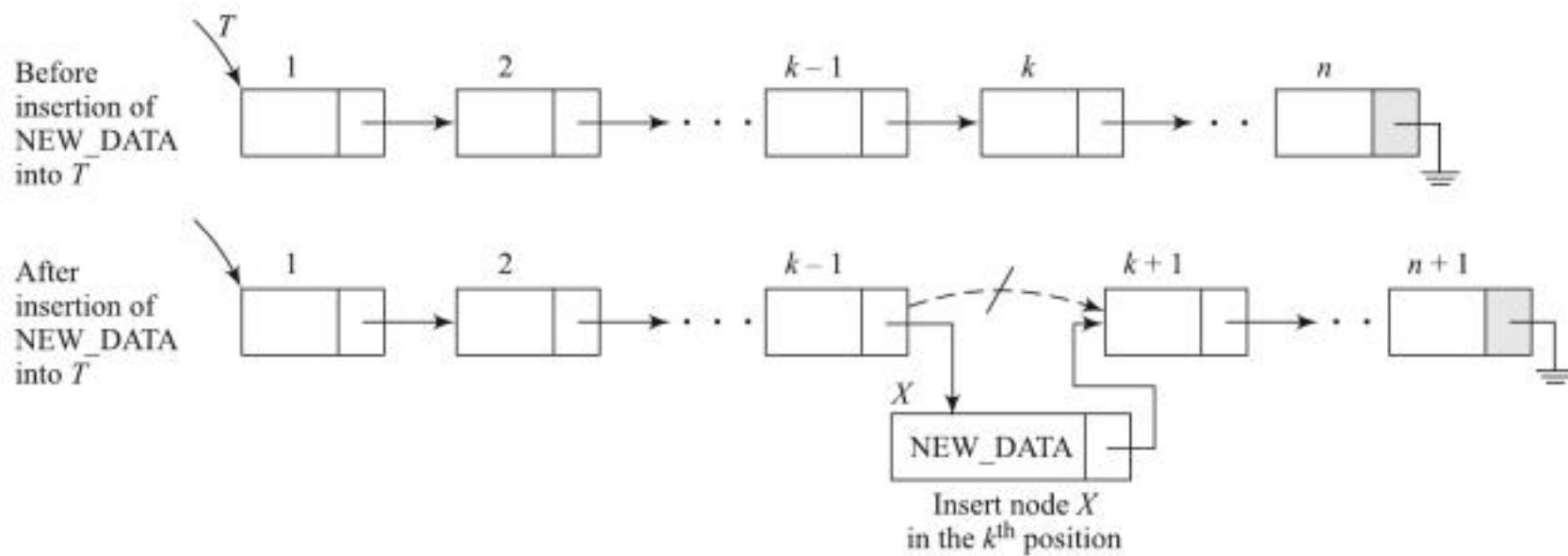


Fig. I 6.2

```

while (COUNT ≠ k) do
    PREVIOUS_PTR = TEMP; /* Remember the address of
                           the predecessor node */
    TEMP = LINK(TEMP); /* TEMP slides down the list */
    COUNT = COUNT + 1;
endwhile
LINK(PREVIOUS_PTR) = X;
LINK(X) = TEMP;
end INSERT_SL_K

```

Problem 6.3 Write a pseudocode procedure to delete the last element of a singly linked list T .

Solution: The logical representation of list T before and after deletion of the last element is shown in Fig. I 6.3.

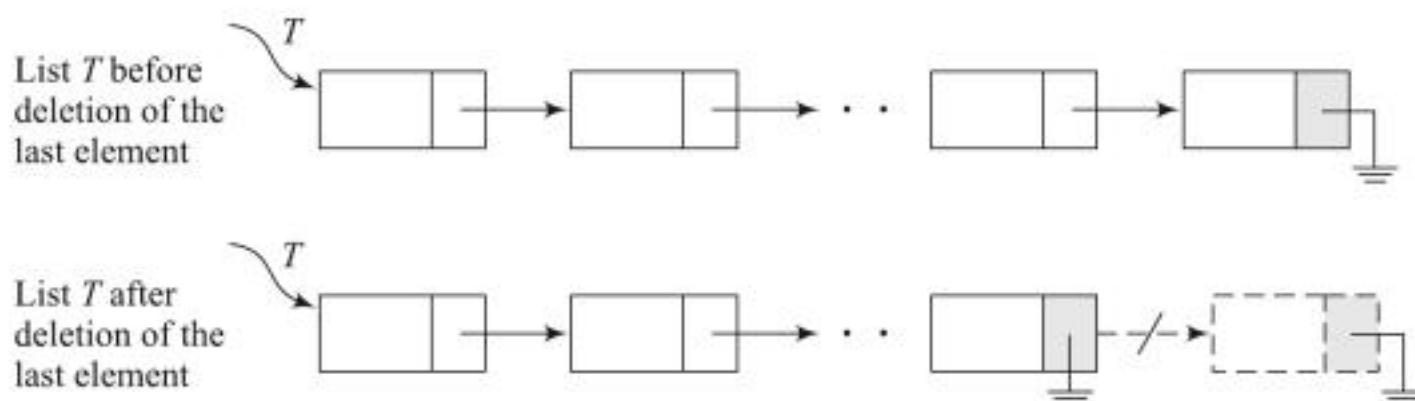


Fig. I 6.3

The pseudocode procedure is

```

Procedure DELETE_LAST( $T$ )
    if ( $T$  = NIL) then (call ABANDON_DELETE;)
    else
        { TEMP =  $T$ 
        while (LINK(TEMP) ≠ NIL)

```

```

PREVIOUS_PTR = TEMP;           /*slide down the list in
search of the last node */
TEMP = LINK(TEMP);
endwhile
LINK(PREVIOUS_PTR) = NIL;
call RETURN(TEMP);
}
end DELETE_LAST.

```

Problem 6.4 Write a pseudocode procedure to count the number of nodes in a circularly linked list with a head node, representing a list of positive integers. Store the count of nodes as a negative number in the head node.

Solution: Let T be a circularly linked list with a head node, representing a list of positive integers. The logical representation of an example list T and the same after execution of the pseudo code procedure is shown in Fig. I 6.4.

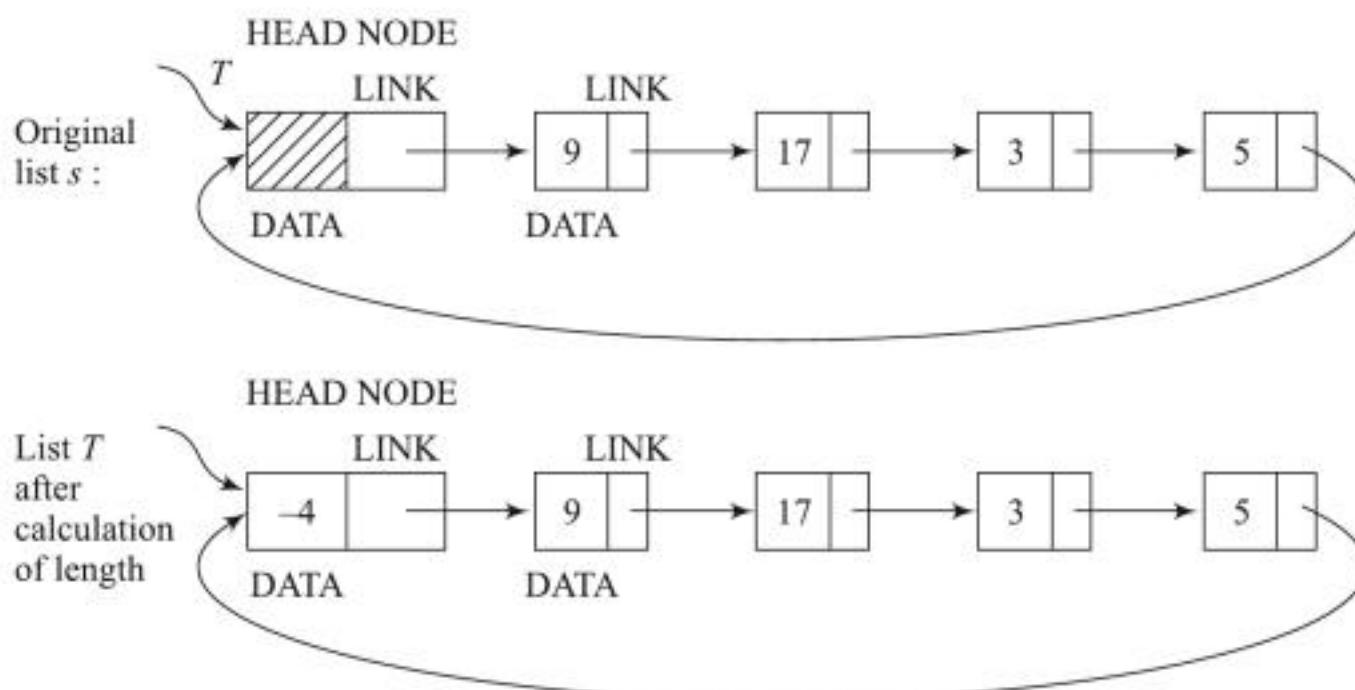


Fig. I 6.4

The pseudocode procedure is:

```

Procedure LENGTH_CLL( $T$ )
    COUNT = 0;
    TEMP =  $T$ ;
while (LINK(TEMP) ≠  $T$ )
    TEMP = LINK(TEMP);
    COUNT = COUNT + 1;
endwhile
    DATA( $T$ ) = -COUNT;
end LENGTH_CLL.

```

Problem 6.5 For the circular doubly linked list T with a head node shown in Fig. I 6.5 with pointers X , Y , Z as illustrated, write a pseudocode instruction to

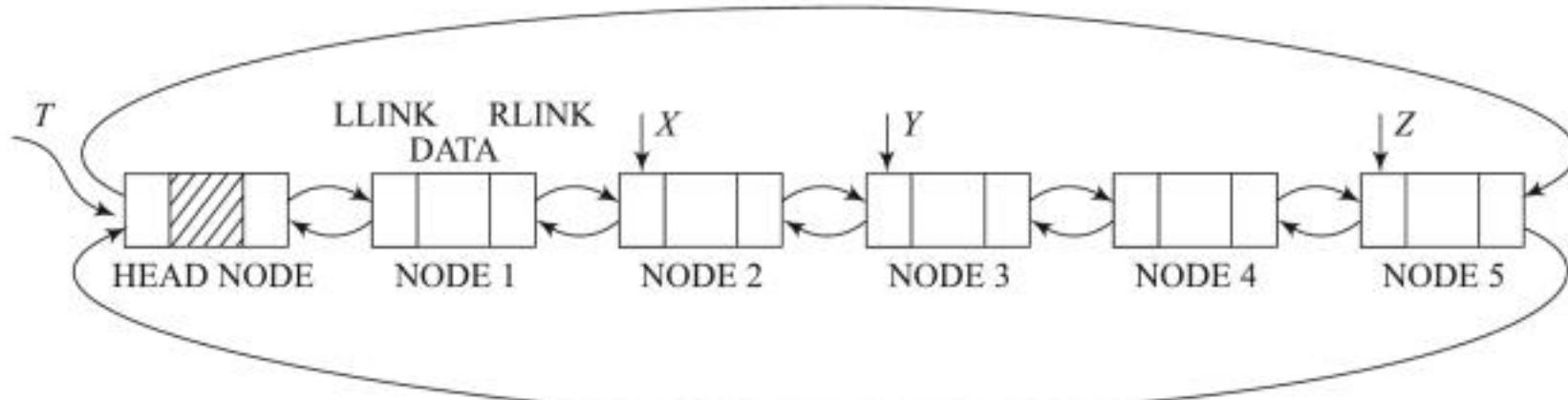


Fig. I 6.5

- Express the DATA field of Node 5
- Express the DATA field of Node 1 as referenced from head node T
- Express the left link of Node 1 as referenced from Node 2
- Express the right link of Node 4 as referenced from Node 5

Solution:

- $\text{DATA}(Z)$
- $\text{DATA}(\text{RLINK}(T))$
- $\text{LLINK}(\text{LLINK}(X))$
- $\text{RLINK}(\text{LLINK}(Z))$

Problem 6.6 Given the following circular doubly linked list Fig. I 6.6(a), fill up the missing values in the DATA fields marked “? ” using the clues given.

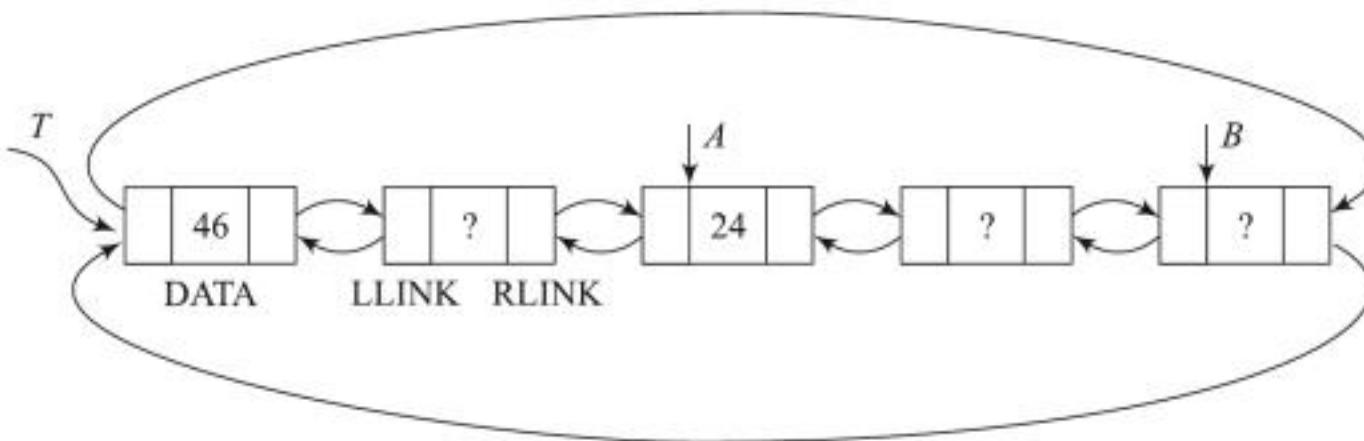


Fig. I 6.6(a)

- $\text{DATA}(B) = \text{DATA}(\text{LLINK}(\text{RLINK}(A))) + \text{DATA}(\text{LLINK}(\text{RLINK}(T)))$
- $\text{DATA}(\text{LLINK}(B)) = \text{DATA}(B) + 10$
- $\text{DATA}(\text{RLINK}(\text{RLINK}(B))) = \text{DATA}(\text{LLINK}(\text{LLINK}(B)))$

Solution:

- $\text{DATA}(B) = \text{DATA}(A) + \text{DATA}(T)$
 $(\because \text{LLINK}(\text{RLINK}(A)) = A \text{ and } \text{LLINK}(\text{RLINK}(T)) = T)$
 $= 24 + 46$
 $= 70$
- $\text{DATA}(\text{LLINK}(B)) = \text{DATA}(B) + 10$
 $= 70 + 10$
 $= 80$

- (iii) DATA (RLINK (RLINK(B))) = DATA(A)
 $= 24$
 $(\because \text{LLINK(LLINK}(B)\text{)} = A)$

The updated list T is shown in Fig. I 6.6(b).

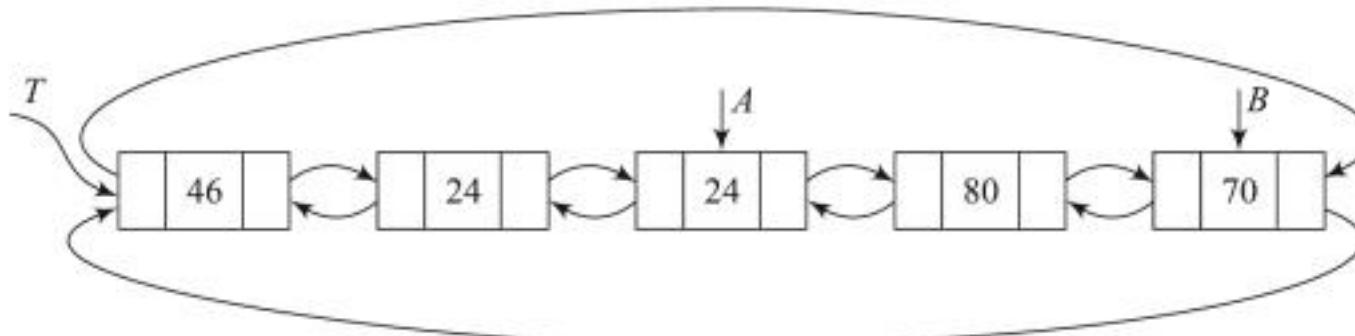


Fig. I 6.6(b)

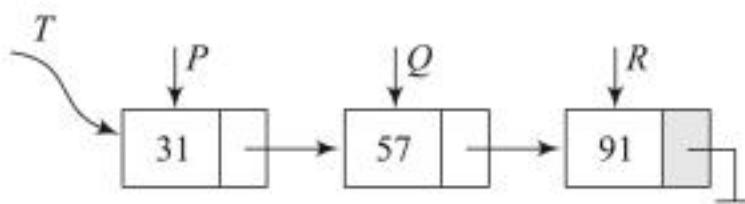
Problem 6.7 In a programming language (Pascal) the declaration of a node in a singly linked list is shown in Fig. I 6.7(a). The list referred to for the problem is shown in Fig. I 6.7(b). Given P to be a pointer to a node, the instructions $\text{DATA}(P)$ and $\text{LINK}(P)$ referring to the DATA and LINK fields respectively of the node P , are equivalently represented by $P \uparrow$. DATA and $P \uparrow.\text{LINK}$ in the programming language.

What do the following commands do to the logical representation of the list T ?

```

TYPE
  POINTER =  $\uparrow$  NODE ;
  NODE = RECORD
    DATA : integer ;
    LINK : POINTER
  END ;
VAR P, Q, R : POINTER
  
```

(a) Declaration of a node in a singly linked list T



(b) A single linked list T

Fig. I 6.7 (a-b) Declaration of a node in a programming language and the logical representation of a singly linked list T

- (i) $P \uparrow.\text{DATA} := Q \uparrow.\text{DATA} + R \uparrow.\text{DATA}$
- (ii) $Q := P$
- (iii) $R \uparrow.\text{LINK} := Q$
- (iv) $R \uparrow.\text{DATA} := Q \uparrow.\text{LINK} \uparrow.\text{DATA} + 10$

Solution: The logical representation of list T after every command is shown in Fig. I 6.7 (c, d, e, f) respectively.

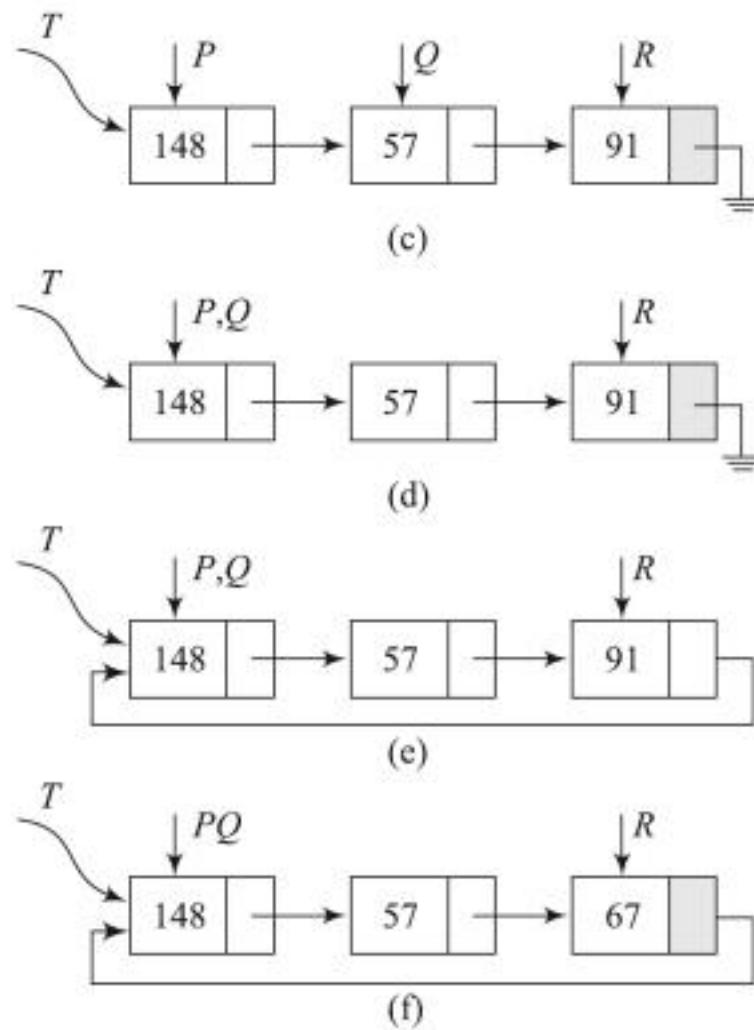


Fig. I 6.7 (c-f) Logical representation of list T after execution of commands (i) – (iv) of Illustrative Problem 6.7

$$(i) P \uparrow .\text{DATA} := Q \uparrow .\text{DATA} + R \uparrow .\text{DATA}$$

$$P \uparrow .\text{DATA} := 57 + 91 = 148$$

$$(ii) Q := P$$

Here Q is reset to point to the node pointed to by P .

$$(iii) R \uparrow .\text{LINK} := Q$$

The link field of node R is reset to point to Q . In other words, the list T turns into a circularly linked list!

$$(iv) R \uparrow .\text{DATA} := Q \uparrow .\text{LINK} \uparrow .\text{DATA} + 10$$

$$:= 57 + 10$$

$$:= 67$$

Problem 6.8 Given the logical representations of a list T and the update in its links, write a one-line instruction which will effect the change indicated. The solid lines indicate the existing pointers and broken lines the updated links.

Solution:

$$(i) \text{RLINK(RLINK}(X)\text{)} = \text{NIL}$$

or

$$\text{RLINK(LLINK}(T)\text{)} = \text{NIL}$$

$$(ii) \text{LINK(LINK}(Y)\text{)} = T$$

$$(iii) \text{RLINK}(T) = \text{RLINK(RLINK}(T)\text{)}$$

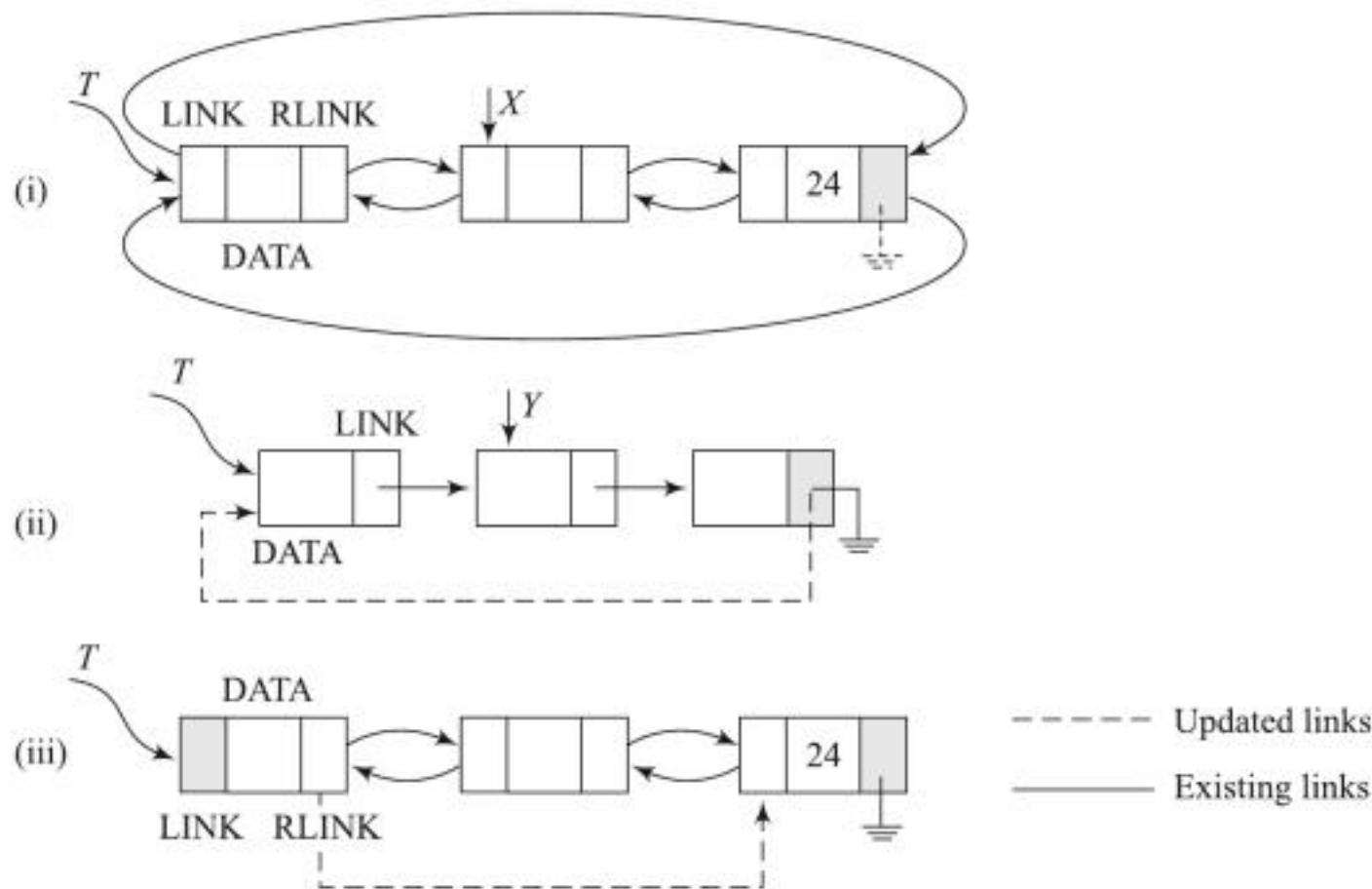


Fig. I 6.8



Review Questions

The following is a snap shot of a memory which stores a circular doubly linked list `TENNIS_STARS`, that is head node free. Answer the questions 1 to 3, with regard to the list.

`TENNIS_STARS:`

2

	LLINK	DATA	RLINK
1	9	sabatini	4
2	6	graf	5
3	2	navarati洛va	8
4	1	mirza	7
5	2	nirupama	6
6	5	chris	2
7	9	myskina	3
8	8	hingis	1
9	1	mandlikova	9

1. The number of data elements in the list `TENNIS_STARS` is
 (a) 3 (b) 2 (c) 5 (d) 7

2. The successor of 'graf' in the list TENNIS_STARS is
 (a) navarati lava (b) sabatini (c) nirupama (d) chris
3. In the list TENNIS_STARS, DATA (RLINK(LLINK(5))) = -----
 (a) mirza (b) graf (c) nirupama (d) chris
4. Given the singly linked list T shown in Fig. R6.4, the following code inserts the node containing the data "where_am_i"

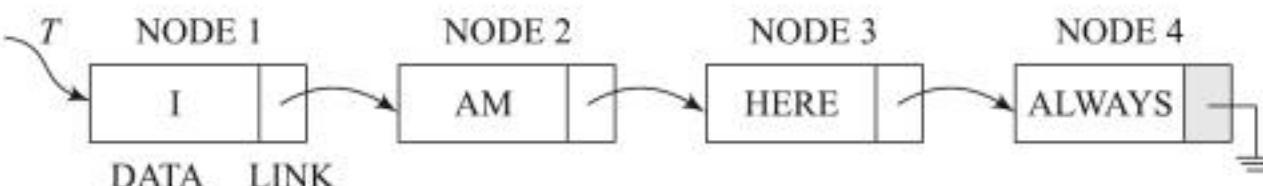


Fig. R6.4

```

 $T = \text{LINK}(T)$ 
 $P = \text{LINK}(\text{LINK}(T))$ 
 $\text{GETNODE}(X)$ 
 $\text{DATA}(X) = \text{"where\_am\_i"}$ 
 $\text{LINK}(X) = P$ 
 $\text{LINK}(\text{LINK}(T)) = X$ 

```

- (a) between nodes 1 and 2 (b) between nodes 2 and 3
- (c) between nodes 3 and 4 (d) after node 4
5. For the singly linked list T shown in Fig. R6.4, after deletion of Node 3, DATA (LINK (LINK (T))) = -----
 (a) I (b) AM (c) HERE (d) ALWAYS
6. What is the need for linked representations of lists?
7. What are the advantages of circular lists over singly linked lists?
8. What are the advantages and disadvantages of doubly linked lists over singly linked lists?
9. What is the use of a head node in a linked list?
10. What are the conditions for testing whether a linked list T is empty, if T is a (i) simple singly linked list (ii) headed singly linked list (iii) simple circularly linked list and (iv) headed circularly linked list?
11. Sketch a multiply linked list representation for the following sparse matrix:

$$\begin{bmatrix} -9 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 5 & 0 & 2 \\ 0 & 7 & 0 & 5 \end{bmatrix}$$

12. Demonstrate the application of singly linked lists for the addition of the polynomials P_1 and P_2 given below:

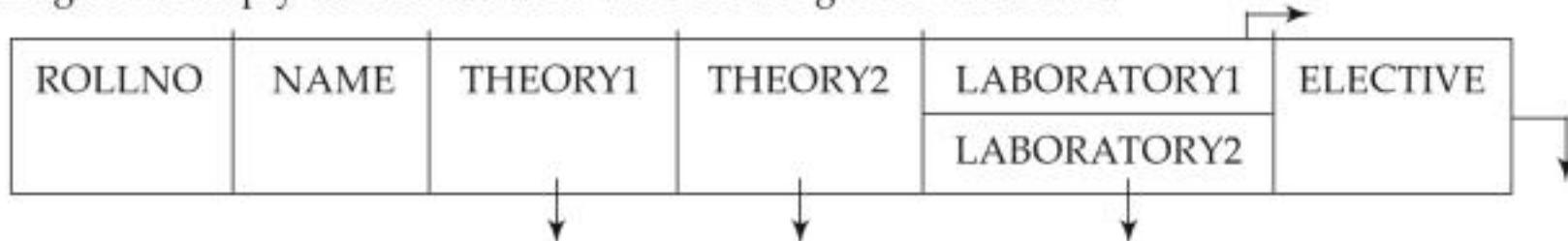
$$P_1 : 19x^6 + 78x^4 + 6x^3 - 23x^2 - 34$$

$$P_2 : 67x^6 + 89x^5 - 23x^3 - 75x^2 - 89x - 21$$



Programming Assignments

- Let $X = (x_1, x_2, \dots, x_n)$, $Y = (y_1, y_2, y_3, \dots, y_m)$ be two lists with a sorted sequence of elements. Execute a program to merge the two lists together as a list Z with $m + n$ elements. Implement the lists using singly linked list representations.
- Execute a program which will split a circularly linked list P with n nodes into two circularly linked lists P_1, P_2 with the first $\lfloor n/2 \rfloor$ and the last $n - \lfloor n/2 \rfloor$ nodes of the list P in them, respectively.
- Write a menu driven program which will maintain a list of car models, their price, name of the manufacturer, engine capacity etc., as a doubly linked list. The menu should make provisions for inserting information pertaining to new car models, delete obsolete models, update data such as price besides answering queries such as listing all car models within a price range specified by the client and listing all details given a car model.
- Students enrolled for a Diploma course in Computer Science opt for two theory courses, an elective course and two laboratory courses from a list of courses offered for the programme. Design a multiply linked list with the following node structure:



A student may change his/her elective course within a week of the enrollment. At the end of the period, the department takes count of the number of students who have enrolled for a specific course in the theory, laboratory and elective options.

Execute a program to implement the multiply linked list with provisions to insert nodes, to update information besides generating reports as needed by the department.

- [Topological Sorting] The problem of *topological sorting* is to arrange a set of objects $\{O_1, O_2, \dots, O_n\}$ obeying rules of precedence, into a linear sequence such that whenever O_i precedes O_j we have $i < j$. The sorting procedure has wide applications in PERT, linguistics, network theory, etc. Thus when a project is made up of a group of activities observing precedence relations amongst themselves, it is convenient to arrange the activities in a linear sequence to effectively execute the project.

Again, as another example, while designing a glossary for a book it is essential that the terms W_i are listed in a linear sequence such that no term is used before it has been defined. Figure P6.5 illustrates topological sorting.

A simple way to do topological sorting is to look for objects which are not preceded by any other objects and release them into the output linear sequence. Remove these objects and continue the same with other objects of the network, until the entire set of objects have been released into the linear sequence. However, topological sort fails when the network has a cycle. In other words if O_i precedes O_j and O_j precedes O_i , the procedure is stalled.

Design and implement an algorithm to perform topological sort of a sequence of objects using a linked list data structure.

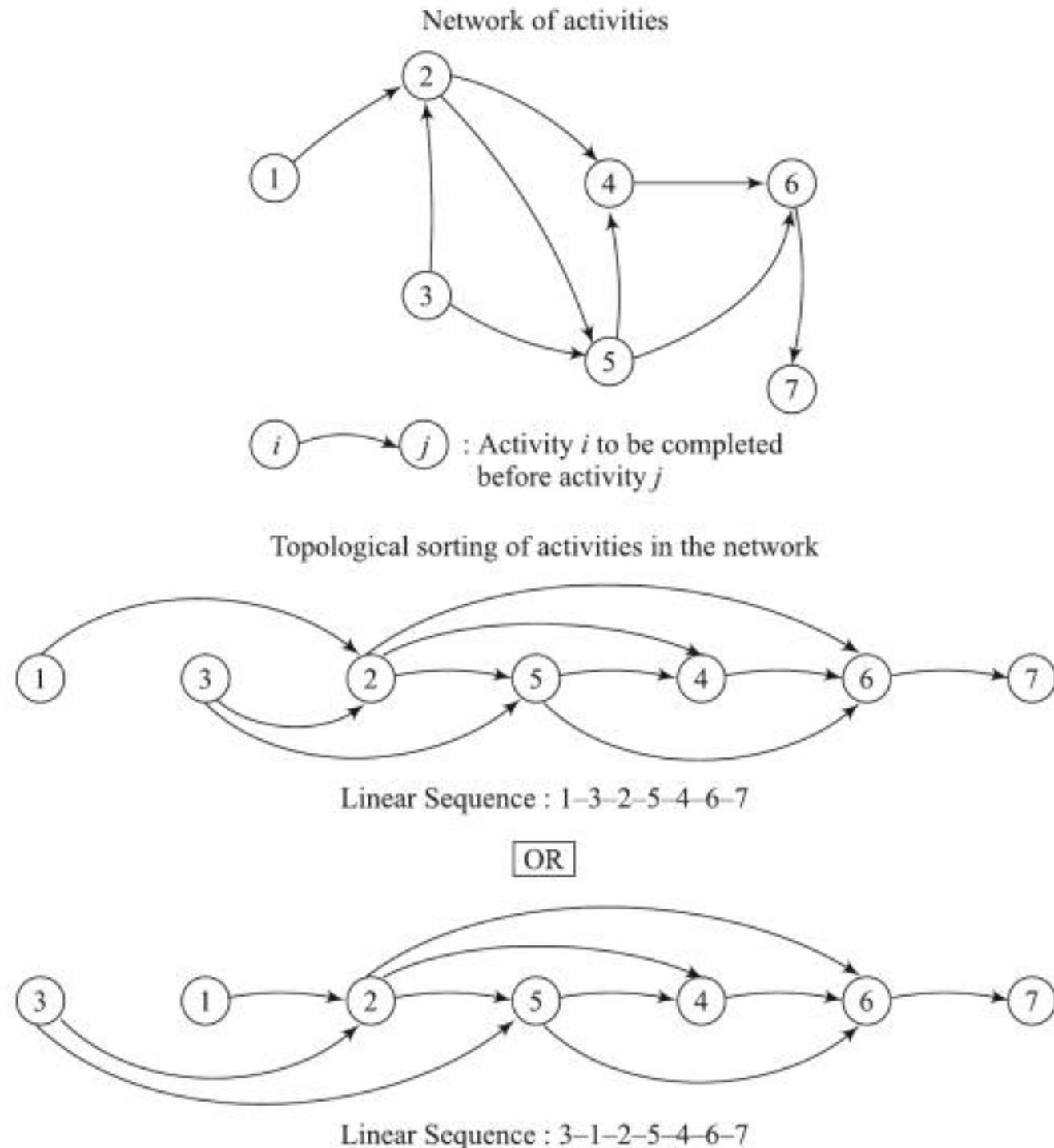


Fig. P6.5 Topological sorting of a network



LINKED STACKS AND LINKED QUEUES

7

In Chapters 4 and 5 we discussed a sequential representation of the stack and queue data structures. Stacks and queues were implemented using arrays and hence inherited all the drawbacks of the sequential data structure.

In this chapter, we discuss the representation of stacks and queues using a linked representation viz., singly linked lists. The inherent merits of the linked representation render an efficient implementation of the linked stack and linked queue.

We first define a linked representation of the two data structures and discuss the insert / delete operations performed on them. The role played by the linked stack in the management of the free storage pool is detailed. The applications of linked stacks and queues in the problems of balancing symbols and polynomial representation respectively, are discussed last.

- | | |
|-----|--|
| 7.1 | <i>Introduction</i> |
| 7.2 | <i>Operations on
Linked Stacks
and Linked
Queues</i> |
| 7.3 | <i>Dynamic
Memory
Management
and Linked
Stacks</i> |
| 7.4 | <i>Implementation
of Linked
Representations</i> |
| 7.5 | <i>Applications</i> |

Introduction

7.1

To review, a stack is an ordered list with the restriction that elements are added or deleted from only one end of the stack termed top of stack with the ‘inactive’ end known as the bottom of stack. A stack observes the last-in-first-out (LIFO) principle and has its insert and delete operations referred to as Push and Pop respectively.

The draw backs of a sequential representation of a stack data structure are

- (i) finite capacity of the stack and
- (ii) checking for STACK_FULL condition every time a Push operation is effected.

A queue on the other hand is a linear list in which all insertions are made at one end of the list known as the rear end and all deletions are made at the opposite end known as the front end. The queue observes a first-in-first-out (FIFO) principle and the insert and delete operations are known as enqueueing and dequeuing respectively.

The drawbacks of sequential representation of a queue are

- (i) finite capacity of the queue, and

- (ii) Checking for the QUEUE_FULL condition before every insert operation is executed, both in the case of a liner queue and a circular queue.

We now discuss linked representations of a stack and a queue.

Linked stack

A *linked stack* is a linear list of elements commonly implemented as a singly linked list whose start pointer performs the role of the top pointer of a stack. Let a, b, c be a list of elements. Figures 7.1 (a-c) shows the conventional, sequential and linked representations of the stack.

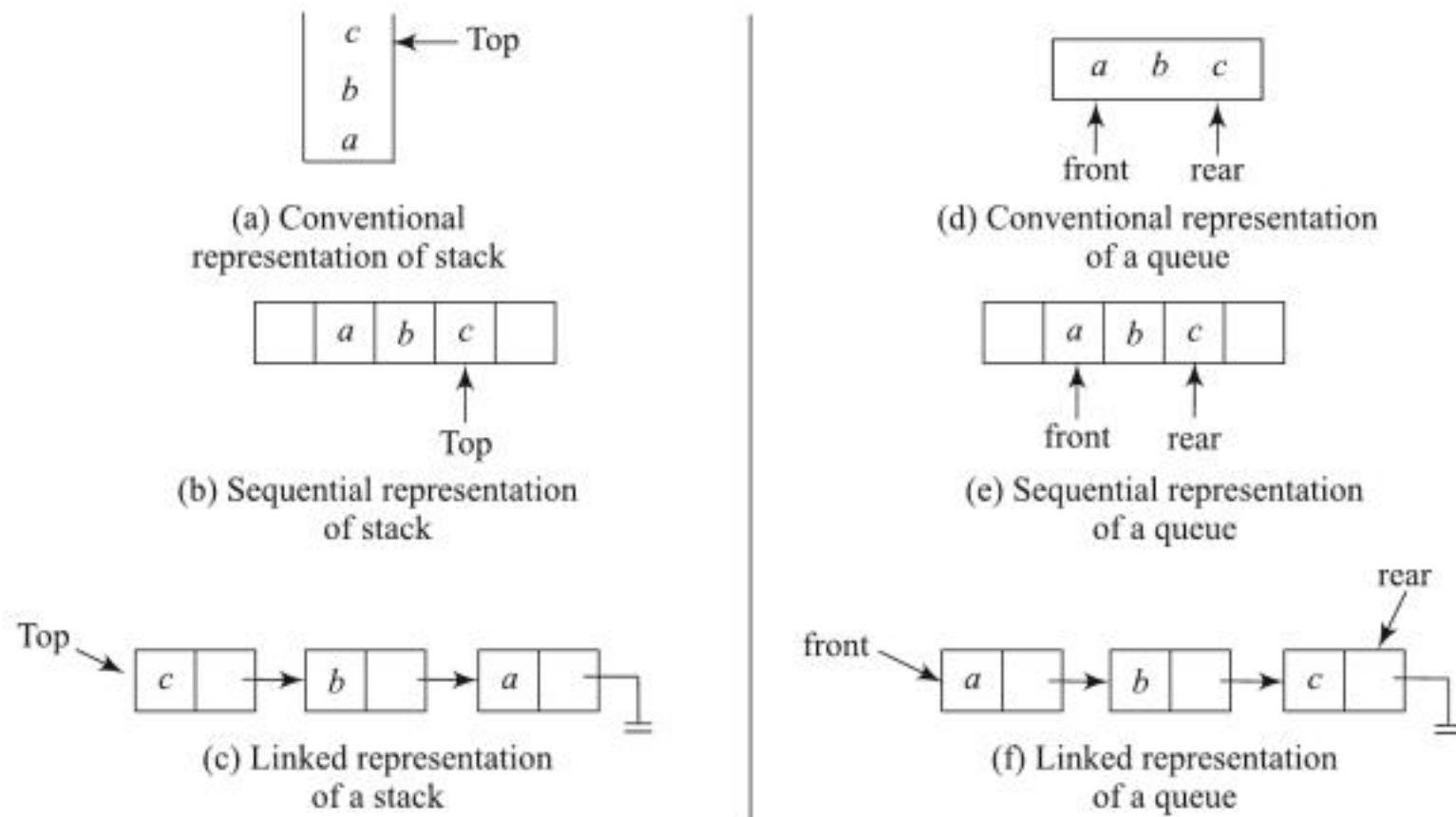


Fig. 7.1 Stack and queue representation (conventional, sequential and linked)

Here, the start pointer of the linked list is appropriately renamed TOP to suit the context.

Linked queues

A *linked queue* is also a linear list of elements commonly implemented as a singly linked list but with two pointers, viz., FRONT and REAR. The start pointer of the singly linked list plays the role of FRONT while the pointer to the last node is set to play the role of REAR. Let a, b, c , be a list of three elements to be represented as a linked queue. Figure 7.1(d-f) shows the conventional, sequential and linked representation of the queue.

Operations on Linked Stacks and Linked Queues

7.2

In this section we discuss the insert and delete operations performed on the linked stack and linked queue data structures and present algorithms for the same.

Linked stack operations

To push an element into the linked stack we insert the node representing the element as the first node in the singly linked list. The top pointer which points to the first element in the singly linked list is automatically updated to point to the new top element. In the case of a pop operation, the node pointed to by the TOP pointer is deleted and TOP is updated to point to the next node as the top element. Figures 7.2(a-b) illustrate the push and pop operation on a linked stack S .

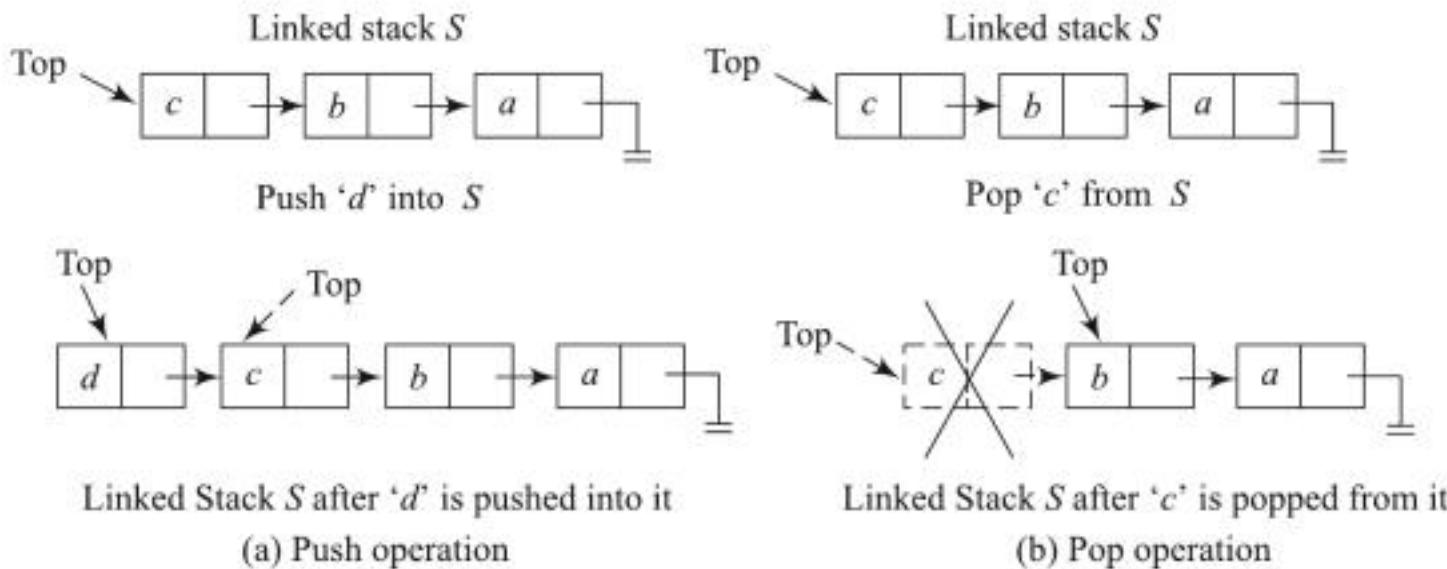


Fig. 7.2 Push and pop operation on a linked stack S

Observe how during the push operation, unlike sequential stack structures, there is no need to check for the STACK-FULL condition due to the unlimited capacity of the data structure.

Linked queue operations

To insert an element into the queue we insert the node representing the element as the last node in the singly linked list for which the REAR pointer is reset to point to the new node as the rear element of the queue. To delete an element from the queue, we remove the first node of the list for which the FRONT pointer is reset to point to the next node as the front element of the queue. Figure 7.3(a-b) illustrates the insert and delete operations on a linked queue Q .

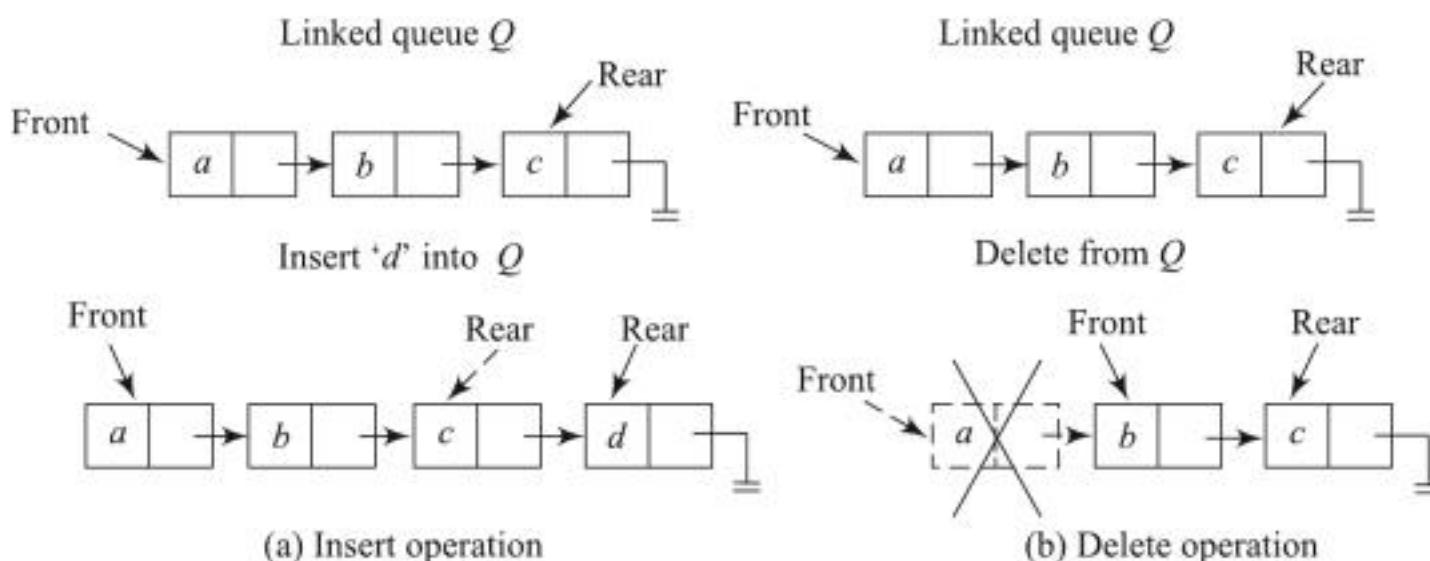


Fig. 7.3 Insert and delete operations on the linked queue Q

The insert operation unlike insertion in sequential queues, does not exhibit the need to check for QUEUE_FULL condition due to the unlimited capacity of the data structure. The introduction of circular queues to annul the drawbacks of the linear queues now appear superfluous, in the light of the linked representation of queues.

Both linked stacks and queues could be represented using a singly linked list with a head node. Also they could be represented as a circularly linked list provided the fundamental principles of LIFO and FIFO are strictly maintained.

We now present the algorithms for the operations discussed in linked stacks and linked queues.

Algorithms for push/pop operations on a linked stack

Let S be a linked stack. Algorithm 7.1 and 7.2 illustrate the push and pop operations to be carried out on the stack S .

Algorithm 7.1: Push item ITEM into a linked stack S with top pointer TOP

```

Procedure PUSH_LINKSTACK (TOP, ITEM)
    /* Insert ITEM into stack */
    Call GETNODE (X)
    DATA(X) = ITEM /*frame node for ITEM */
    LINK(X) = TOP /* insert node X into stack */
    TOP = X /* reset TOP pointer */
end PUSH_LINKSTACK.

```

Note the absence of the STACK_FULL condition. The time complexity of a push operation is $O(1)$.

Algorithm 7.2: Pop from a linked stack S and output the element through ITEM

```

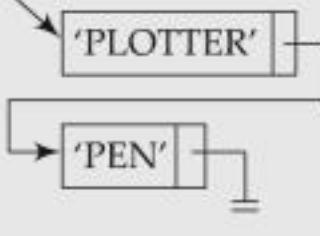
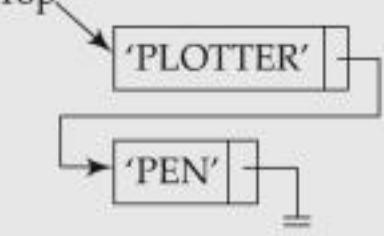
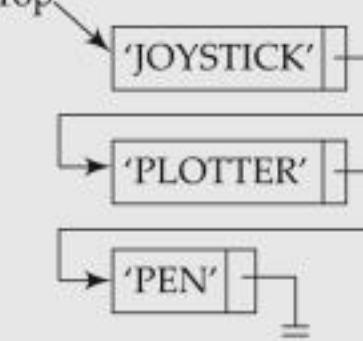
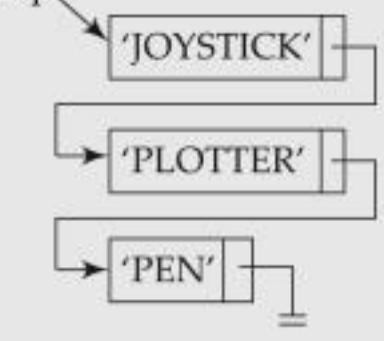
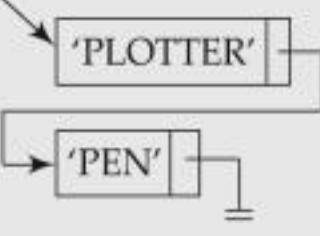
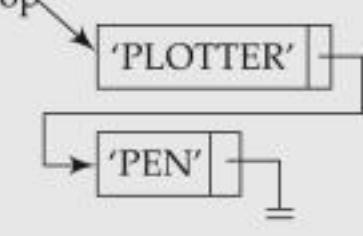
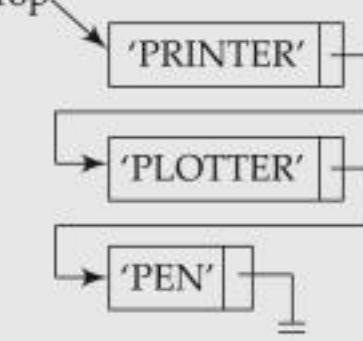
Procedure POP_LINKSTACK(TOP, ITEM)
    /* pop element from stack and set ITEM to the element */
    if (TOP = 0) then call LINKSTACK_EMPTY
        /* check if linked stack is empty */
    else { TEMP = TOP
        ITEM = DATA(TOP)
        TOP = LINK(TOP)
    }
    call RETURN(TEMP) ;
end POP_LINKSTACK.

```

The time complexity of a pop operation is $O(1)$. Example 7.1 illustrates the push and pop operation on a linked stack.

Example 7.1 Consider the stack DEVICE of peripheral devices illustrated in Example 4.1. We implement the same as a linked stack. The insertion of PEN, PLOTTER, JOYSTICK and PRINTER and a deletion operation are illustrated in Table 7.1. We assume the list to be initially empty and TOP to be the top pointer of the stack.

Table 7.1 Insert and delete operations on linked stack DEVICE

Stack Operation	Stack DEVICE before operation	Algorithm invocation	Stack DEVICE after operation	Remarks
1. Push 'PEN' into DEVICE		PUSH_LINKSTACK (Top, 'PEN')		Set Top to point to the first node.
2. Push 'PLOTTER' into DEVICE		PUSH_LINKSTACK (Top, 'PLOTTER')		Insert PLOTTER as the first node and reset Top.
3. Push 'JOYSTICK' into DEVICE		PUSH_LINKSTACK (Top, 'JOYSTICK')		Insert JOYSTICK as the first node and reset Top.
4. Pop from DEVICE		POP_LINKSTACK (Top, 'ITEM')	 ITEM = 'JOYSTICK'	Return the first node and reset Top.
5. Push 'PRINTER' into DEVICE		PUSH_LINKSTACK (Top, 'PRINTER')		Insert PRINTER as the first node and reset Top.

Algorithms for insert and delete operations in a linked queue

Let Q be a linked queue. Algorithms 7.3 and 7.4 illustrate the insert and delete operations on the queue Q .

Algorithm 7.3: Push item *ITEM* into a linear queue *Q* with FRONT and REAR as the front and rear pointer to the queue

```

Procedure INSERT_LINKQUEUE (FRONT, REAR, ITEM)
Call GETNODE (X);
DATA(X) = ITEM;
LINK(X) = NIL; /* Node with ITEM is ready to be inserted into Q */
if (FRONT = 0) then FRONT = REAR = X;
/* If Q is empty then ITEM is the first element in the queue
Q */
else {LINK(REAR) = X;
REAR = X
}
end INSERT_LINKQUEUE.
```

Observe the absence of QUEUE_FULL condition in the insert procedure. The time complexity of an insert operation is $O(1)$.

Algorithm 7.4: Delete element from the linked queue *Q* through ITEM with FRONT and REAR as the front and rear pointers

```

Procedure DELETE_LINKQUEUE (FRONT, ITEM)
if (FRONT = 0) then call LINKQUEUE_EMPTY;
/* Test condition to avoid deletion in an empty queue */
else
    {TEMP = FRONT;
    ITEM = DATA(TEMP);
    FRONT = LINK(TEMP);
    }
call RETURN (TEMP); /* return the node TEMP to the free pool */
end DELETE_LINKQUEUE.
```

The time complexity of a delete operation is $O(1)$. Example 7.2 illustrates the insert and delete operations on a linked queue.

Example 7.2 Consider the queue BIRDS illustrated in Example 5.1. The insertion of DOVE, PEACOCK, PIGEON and SWAN, and two deletions are shown in Table 7.2.

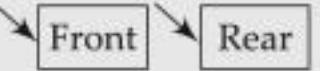
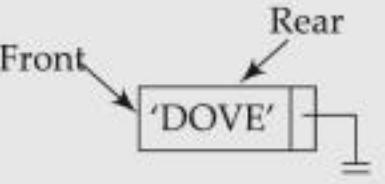
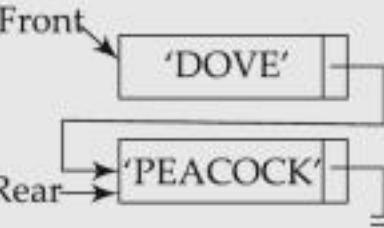
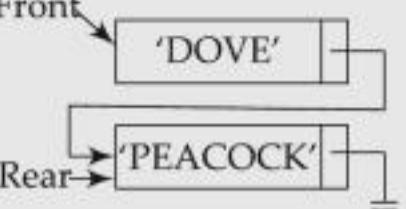
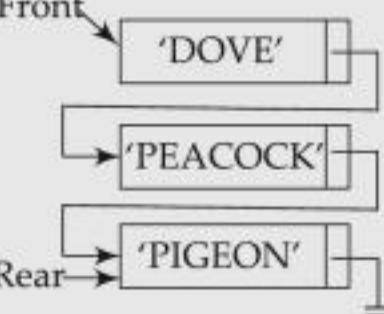
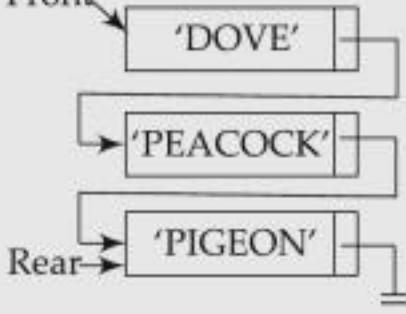
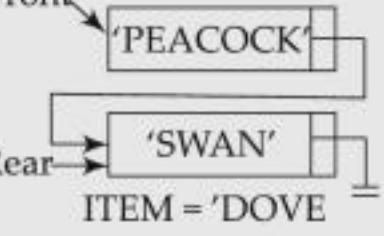
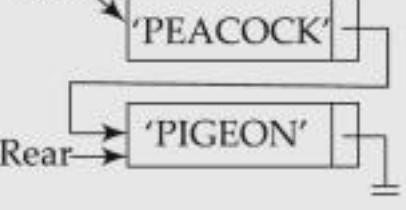
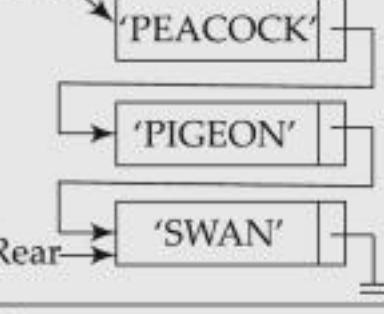
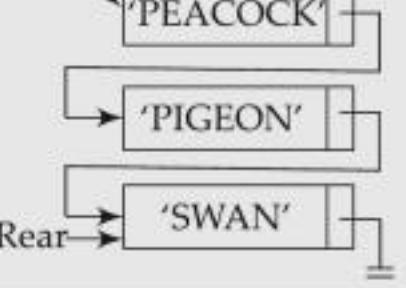
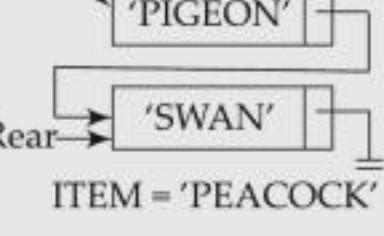
Owing to the linked representation there is no limitation on the capacity of the stack or queue. In fact, the stack or queue can hold as many elements as the storage memory can accommodate! This dispenses with the need to check for STACK_FULL or QUEUE_FULL conditions during push or insert operations respectively.

The merits of linked stacks and linked queues are therefore

- The conceptual and computational simplicity of the operations
- non finite capacity

The only demerit is the requirement of additional space that is needed to accommodate the link fields.

Table 7.2 Insert and delete operations on a linked queue BIRDS

Linked queue	Linked queue before operation	Algorithm Invocation	Linked queue after operation	Remarks
1. Insert 'DOVE' into BIRDS		INSERT_LINKQUEUE (Front, Rear, 'DOVE')		Since the queue BIRDS is empty, insert DOVE as the first node. Front and Rear point to the node.
2. Insert 'PEACOCK' into BIRDS		INSERT_LINKQUEUE (Front, Rear, 'PEACOCK')		Insert PEACOCK as the last node. Reset Rear pointer.
3. Insert 'PIGEON' into BIRDS		INSERT_LINKQUEUE (Front, Rear, 'PIGEON')		Insert PIGEON as the last node. Reset Rear pointer.
4. Delete from BIRDS		DELETE_LINKQUEUE (Front, Rear, ITEM)		Delete node pointed to by Front. Reset Front.
5. Insert SWAN into BIRDS		INSERT_LINKQUEUE (Front, Rear, SWAN)		Insert SWAN as the last node. Reset Rear.
6. Delete from BIRDS		DELETE_LINKQUEUE (Front, Rear, ITEM)		Delete node pointed to by Front. Reset Front.

Dynamic Memory Management and Linked Stacks

7.3

Dynamic memory management deals with methods of allocating storage and recycling unused space for future use. The automatic recycling of dynamically allocated memory is also known as **Garbage collection**.

If the memory storage pool is thought of as a repository of nodes, then dynamic memory management primarily revolves round the two actions of **allocating nodes** (for use by the application) and **liberating nodes** (after their release by the application). Several intelligent strategies for the efficient allocation and liberation of nodes have been discussed in the literature. However, we choose to discuss this topic from the perspective of a linked stack application.

Every linked representation, which makes use of nodes to accommodate data elements, executes procedure `GETNODE ()` to have the desired node allocated to it, from the free storage pool and procedure `RETURN ()` to dispose or liberate the node released by it, into the storage pool. Free storage pool is also referred to as **Available Space** (`AVAIL_SPACE`).

When the application invokes `GETNODE ()`, a node from the available space data structure is deleted, to be handed over for use by the program and when `RETURN ()` is invoked, the node disposed off by the application is inserted into the available space for future use.

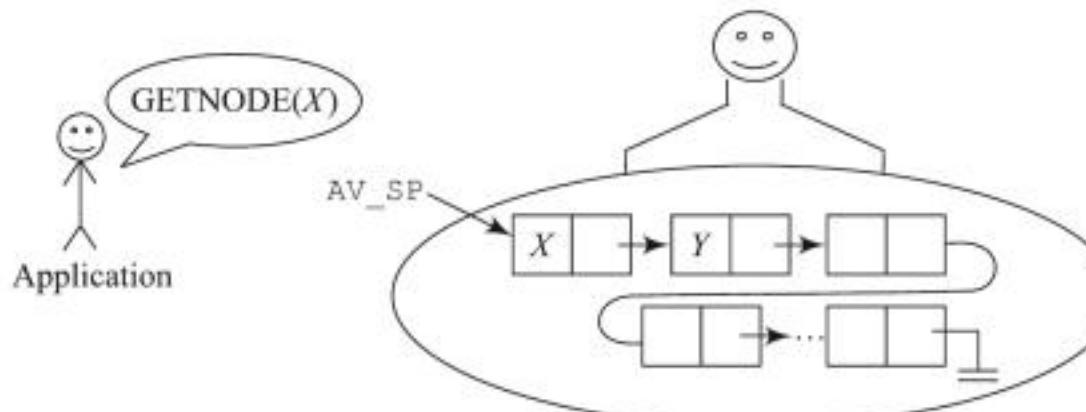
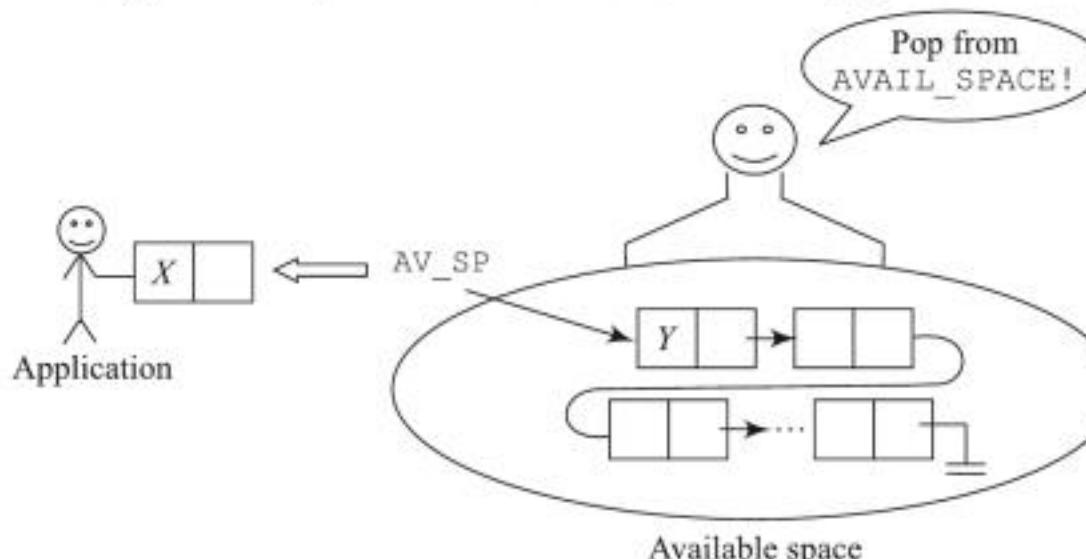
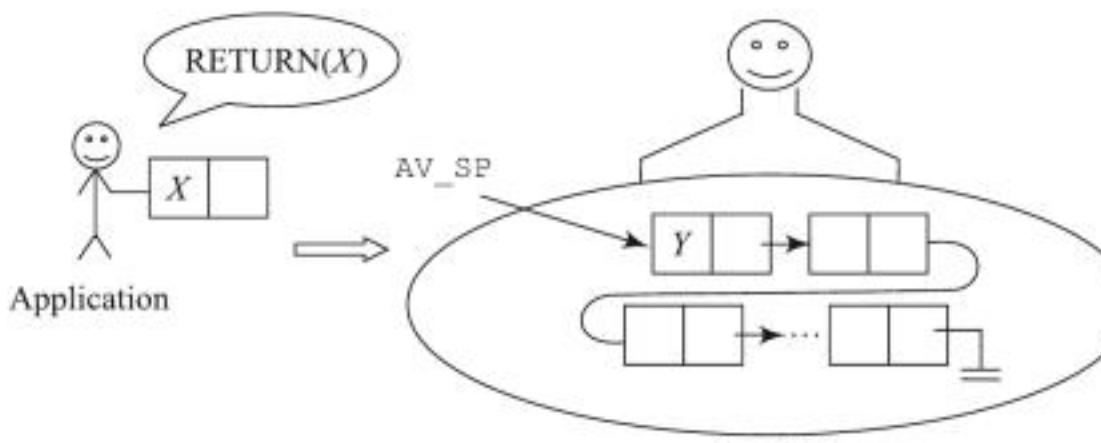
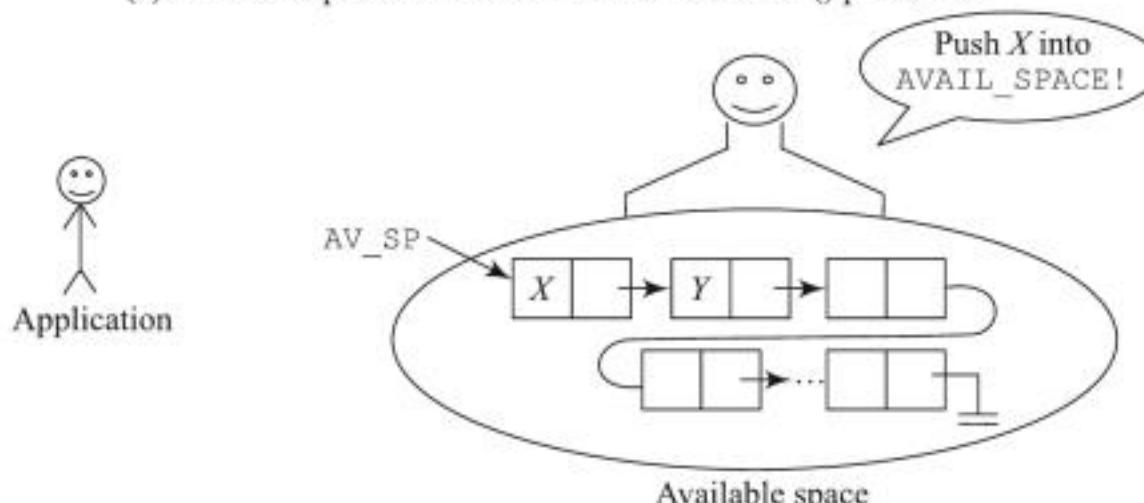
The most commonly used data structure for management of `AVAIL_SPACE` and its insert / delete operation is the linked stack. The list of free nodes in `AVAIL_SPACE` are all linked together and maintained as a linked stack with a top pointer (`AV_SP`). When `GETNODE ()` is invoked, a pop operation of the linked stack is done releasing a node for use by the application and when `RETURN ()` is invoked, a push operation of the linked stack is done. Figure 7.4 illustrates the association between the `GETNODE ()` and `RETURN ()` procedures and `AVAIL_SPACE` maintained as a linked stack.

We now implement `GETNODE ()` and `RETURN ()` procedures which in fact are nothing but POP and PUSH operations on the linked stack `AVAIL_SPACE`. Algorithms 7.5 and 7.6 illustrates the implementation of the procedures.

It is obvious that at a given instance the adjacent or other nodes in the `AVAIL_SPACE` are neighbors that are physically contiguous in the memory but lie scattered in the list. This may eventually lead to holes in the memory leading to inefficient use of memory. When variable size nodes are in use, it is desirable to compact memory so that all free nodes form a contiguous block of memory. Such a thing is termed as **memory compaction**.

It now becomes essential that the storage manager, for efficient management of memory, every time a node is returned to the free pool, ensures that the neighboring blocks of memory that are free are coalesced into a single block of memory, so as to satisfy large requests for memory. This is however easier said than done. To look for neighboring nodes which are free, a 'brute force approach' calls for a complete search through `AVAIL_SPACE` list before collapsing the adjacent free nodes into a single block.

Allocation strategies such as Boundary Tag method and Buddy system method, with efficient reservation and liberation of nodes have been proposed in the literature.

(a) Available space before execution of `GETNODE()` procedure(b) Available space after execution of `GETNODE()` procedure(c) Available space before execution of `RETURN()` procedure(d) Available space after execution of `RETURN()` procedure**Fig. 7.4** Association between `GETNODE()`, `RETURN()` procedures and `AVAIL_SPACE`

Algorithm 7.5: Implementation of procedure GETNODE (X) where AV is the pointer to the linked stack implementation of AVAIL_SPACE

```

Procedure GETNODE (X)
  if (AV = 0) then call NO_FREE_NODES;
    /* AVAIL_SPACE has no free nodes to allocate */
  else { X = AV;
    AV = LINK (AV); /* Return the address X of the top node in
                      AVAIL_SPACE */
  }
end GETNODE.
```

Algorithm 7.6: Implementation of procedure RETURN (X) where AV is the pointer to the linked stack implementation of AVAIL_SPACE

```

Procedure RETURN (X)
  LINK (X) = AV; /* Push node X into AVAIL_SPACE and reset AV */
  AV = X;
end RETURN.
```

Implementation of Linked Representations

7.4

It is emphasized here that nodes belonging to the reserved pool, that is nodes which are currently in use, coexist with the nodes of the free pool in the same storage area. It is therefore not uncommon to have a reserved node having a free node as its physically contiguous neighbor. While the link fields of the free nodes, which in its simplest form is a linked stack, keeps track of the free nodes in the list, the link fields of the reserved pool similarly keep track of the reserved nodes in the list. Figure 7.5 illustrates a simple scheme of reserved pool intertwined with the free pool in the memory storage.

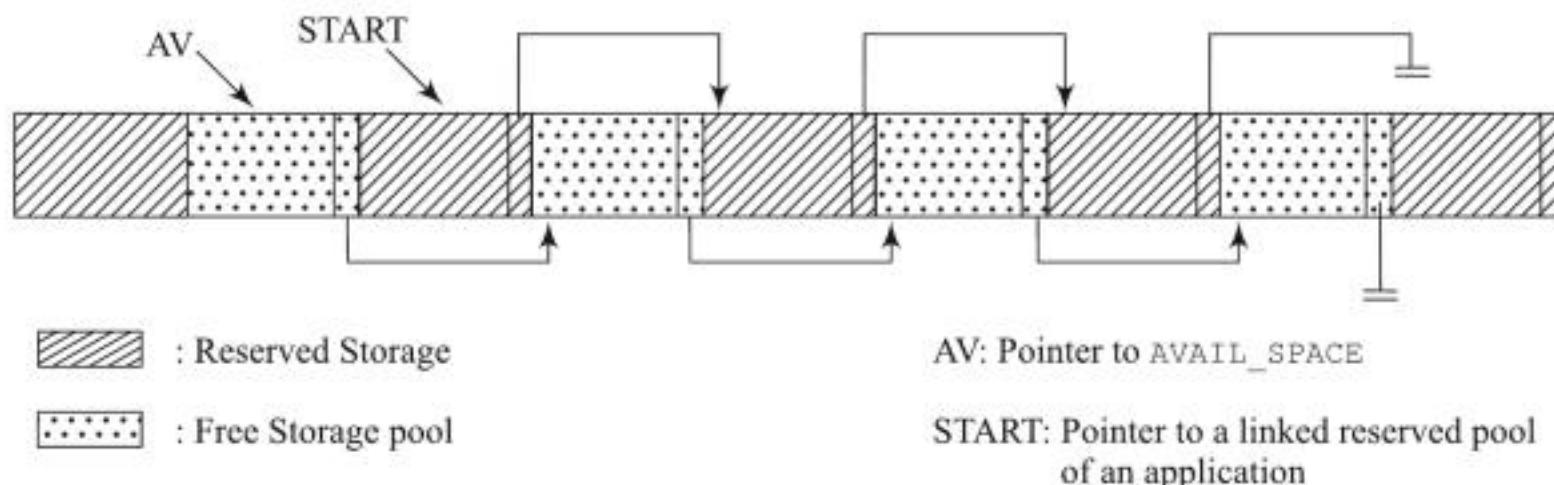


Fig. 7.5 The scheme of reserved storage pool and free storage pool in the memory storage

Example 7.3 illustrates the implementation of a linked representation. For simplicity we consider a singly linked list occupying the reserved pool.

Example 7.3 A snap shot of the memory storage is shown in Fig. 7.6. The reserved pool accommodates a singly linked list (START). The free storage pool of used and disposed nodes is maintained as a linked stack with top pointer AV.

	DATA	Link	
1	22	9	
2	29	8	
3	-14	7	
4	36	1	
5	144	10	
6	-3	2	
7	116	0	
8	43	3	
9	56	5	
10	34	0	

AV: 6

START: 4

Fig. 7.6 A Snapshot of the memory accommodating a singly linked list in its reserved pool and the free storage pool

Note the memory locations AV and START. AV records the address of the first node in the free storage pool and START the same of the singly linked list in the reserved pool. The logical representation of the singly linked list and the available space are illustrated in Fig. 7.7.

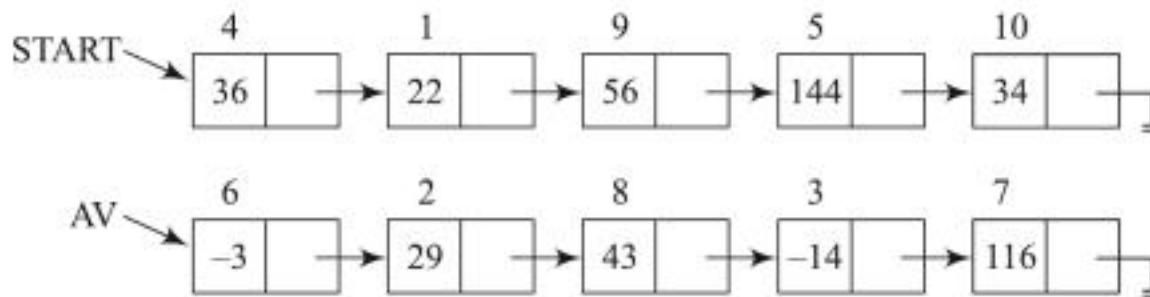


Fig. 7.7 Logical representation of the singly linked list and the AVAIL_SPACE shown in Fig. 7.6

Applications

7.5

All applications of linear queues and linear stacks can be implemented as linked stacks and linked queues. In this section we discuss the following problems,

- (i) Balancing symbols,
 - (ii) Polynomial representation
- as application of linked stacks and linked queues respectively.

Balancing symbols

An important activity performed by compilers is to check for syntax errors in the program code. One such error checking mechanism is the balancing of symbols or specifically, balancing of parentheses in the context of expressions, which is exclusive to this discussion.

For the balancing of parentheses, the left parentheses or braces or brackets as allowed by the language syntax, must have closing or matching right parentheses, braces or brackets respectively. Thus the usage of (), or { } or [] are correct whereas, (, [,] are incorrect, the former indicative of a balanced occurrence and the latter of an imbalanced occurrence in an expression.

The arithmetic expressions shown in Example 7.4 are balanced in parentheses while those listed in Example 7.5 are imbalanced forcing the compiler to report errors.

Example 7.4 Balanced arithmetic expressions

- (i) $((A + B) \uparrow C - D) + E - F$
- (ii) $(- (A + B) * (C - D)) \uparrow F$

Example 7.5 Imbalanced arithmetic expressions

- (i) $(A + B)^* - (C + D + F)$
- (ii) $-((A + B + C)^* - (E + F)))$

The solution to the problem is an easy but elegant use of a stack to check for mismatched parentheses. The general pseudocode procedure for the problem is illustrated in Algorithm 7.5. Appropriate to the discussion, we choose a linked representation for the stack in the algorithm. Examples 7.6 and 7.7 illustrate the working of the algorithm on two expressions with balanced and unbalanced symbols respectively.

Algorithm 7.5: To check for the balancing of parentheses in a string procedure
BALANCE_EXPR(E)

```

/* $E$  is the expression padded with a $ to indicate end of input*/
clear stack;
while not end_of_string( $E$ )
    read character; /* read a character from string  $E$  */
    if the character is an open symbol then push character in to stack;
    if the character is a close symbol then
        if stack is empty then ERROR( )
        else {pop the stack;
            if the character popped is
                not the matching symbol
            then ERROR( );
        }
    endwhile;
if stack not empty then ERROR();
end BALANCE_EXPR.

```



Example 7.6 Consider the arithmetic expression $((A+B)^* C) - D$ which has balanced parentheses. Table 7.3 illustrates the working of the algorithm on the expression.

Example 7.7 Consider the expression $((A+B)^* C + G$ which has unbalanced parentheses. Table 7.4 illustrates the working of the algorithm on the expression.

Polynomial representation

In Chapter 6, Sec. 6 we had discussed the problem of addition of polynomials as an application of linked lists. In this section, we highlight the representation of polynomials as an application of linear queues.

Consider a polynomial $9x^6 - 2x^4 + 3x^2 + 4$. Adopting the node structure shown in Fig. 7.8(a) (reproduction of Fig. 6.26(a)) the linked list for the polynomial is as shown in Fig. 7.8(b).

Table 7.3 Working of Algorithm BALANCE_EXPR () on the expression $((A + B)^* C) - D$

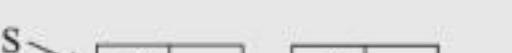
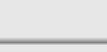
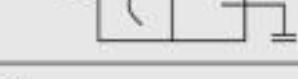
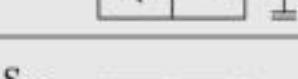
Input string (E)	Stack (S)	Remarks
$((A + B) * C) - D \$$		Initialization. Note E is padded with $\$$ as end of input symbol
$(A + B) * C) - D \$$		Push '(' into S
$A + B) * C) - D \$$		Push '(' into S
$+ B) * C) - D \$$		Ignore character 'A'
$B) * C) - D \$$		Ignore character '+'
$) * C) - D \$$		Ignore character 'B'
$* C) - D \$$		Pop symbol from S . Matching symbol to ")" found. Proceed.
$C) - D \$$		Ignore character '*' Ignore character 'C'
$) - D \$$		Pop symbol from S . Matching symbol to ')' found. Proceed.
$- D \$$		Ignore character '-' Ignore character 'D'
$\$$		End of input encountered. Stack is empty. Success.
$\$$		

Table 7.4 Working of the algorithm BALANCE_EXPR () on the expression $((A+B) * C \uparrow G$

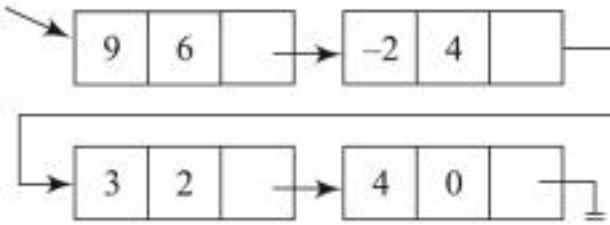
Input string (E)	Stack (S)	Remarks
$((A + B) * C \uparrow G \$$		Initialization. E is padded with \$ as end of input symbol.
$(A + B) * C \uparrow G \$$		Push '(' into S
$A + B) * C \uparrow G \$$		Push '(' into S
$+ B) * C \uparrow G \$$		Ignore character 'A'
$B) * C \uparrow G \$$		Ignore character '+'
$) * C \uparrow G \$$		Ignore character 'B'
$* C \uparrow G \$$		Pop symbol from S. Matching symbol to ")" found. Proceed.
$C \uparrow G \$$		Ignore character '*'
$\uparrow G \$$		Ignore character 'C'
$G \$$		Ignore character '\uparrow '
$\$$		Ignore character 'G'
$\$$		End of input encountered. Stack is not empty. Error.

For easy manipulation of the linked list, we represent the polynomial in its decreasing order of exponents of the variable (in the case of uni-variable polynomials). It would therefore be easy for the function handling the reading of the polynomial to implement the linked list as a linear queue, since this would entail an elegant construction of the list from the symbolic representation of the polynomial, by enqueueing the linear queue with the next highest exponent term. The linear queue representation for the polynomial $9x^6 - 2x^4 + 3x^2 + 4$ is shown in Fig. 7.9.



COEFF: Coefficient of the term
EXP: Exponent of the variable

(a) Node structure

(b) Linked list representation of the polynomial $9x^6 - 2x^4 + 3x^2 + 4$ **Fig. 7.8** Linked list representation of a polynomial

Also, after the manipulation of the polynomials (addition, subtraction etc.) the resulting polynomial could also be elegantly represented as a linear queue. This merely calls for enqueueing the linear queue with the just manipulated term. Recall the problem of addition of polynomials discussed in Sec. 6.6. Maintaining the added polynomial as a linear queue would only call for 'appending' the added terms (coefficients of terms with like exponents) to the rear of the list. However, during the manipulation, the linear queue representation of the polynomials are to be treated as traversable queues. A *traversable queue* while retaining the operations of enqueueing and dequeuing, permits traversal of the list in which nodes may be examined.

**Fig. 7.9** Linear queue representation of the polynomial $9x^6 - 2x^4 + 3x^2 + 4$ 

Summary

- Sequential representation of stacks and queues suffer from the limitation of finite capacity besides checking for the STACK_FULL and QUEUE_FULL conditions, each time a push or insert operation is executed respectively.
- Linked stacks and linked queues are singly linked list implementation of stacks and queues, though a circularly linked list representation can also be attempted without hampering the LIFO or FIFO principle of the respective data structures.
- Linked stacks and linked queues display the merits of conceptual and computational simplicity of insert and delete operations besides absence of limited capacity. However, the requirement of additional space to accommodate the link fields can be viewed as a demerit.
- The maintenance of available space list calls for the application of linked stacks.
- The problems of balancing of symbols and polynomial representation have been discussed to demonstrate the application of linked stack and linked queue respectively.



Illustrative Problems

Problem 7.1 Given the following memory snap shot where START and AV_SP store the start pointers of the linked list and the available space respectively,

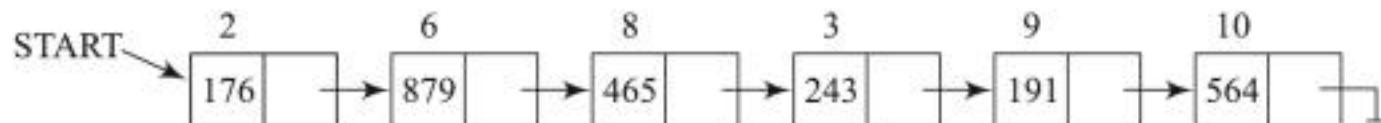
- Identify the linked list
- Show how the linked list and the available space list are affected when the following operations are carried out:

- Insert 116 at the end of the list
 - Delete 243
 - Obtain the memory snap shot after the execution of operations listed in (a) and (b)
- DATA LINK

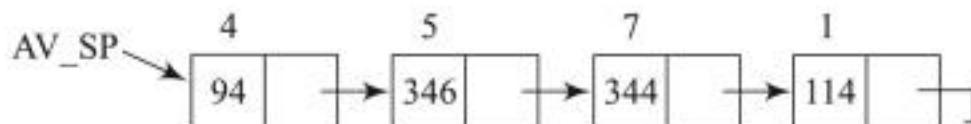
	DATA	LINK
1	114	0
2	176	6
3	243	9
4	94	5
5	346	7
6	879	8
7	344	1
8	465	3
9	191	10
10	564	0

START: 2 AV_SP: 4

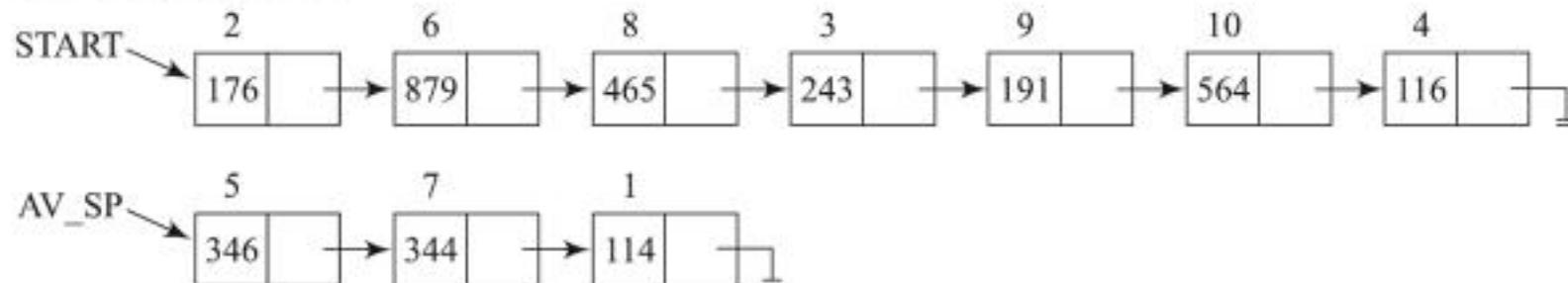
Solution: (i) Since the linked list starts at node whose address is 2, the logical representation of the list is as given below:



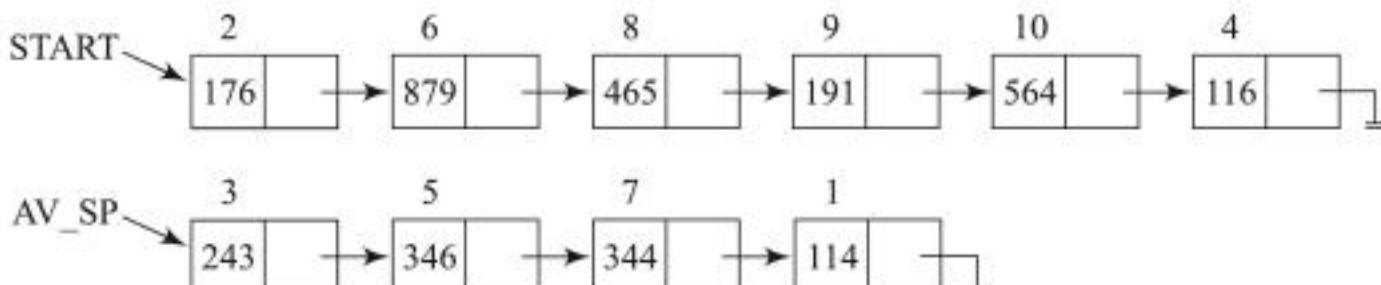
The available space list which functions as a linked stack and starts from node whose address is 4, is given by:



- (a) To insert 116 at the end of the list START, we get a node from the available space list (invoke GETNODE ()). The node released has address 4. The resultant list and the available space list are as follows



- (b) To delete 243, the node holding the element has to be returned to the available space list (invoke (RETURN ())). The resultant list and the available space list are as



(c) The memory snapshot after the execution of (a) and (b) is as given below:

	DATA	LINK		
1	114	0	START: 2	AV_SP: 3
2	176	6		
3	243	5		
4	116	0		
5	346	7		
6	879	8		
7	344	1		
8	465	9		
9	191	10		
10	564	4		

Problem 7.2 Given the following memory snapshot which stores a linked stack L_S and a linked queue L_Q beginning at the respective addresses, obtain the resulting memory snapshot after the following operations are carried out sequentially.

- (i) Enqueue CONCORDE into L_Q.
- (ii) Pop from L_S
- (iii) Dequeue from L_Q
- (iv) Push PALACE_ON_WHEELS into L_S

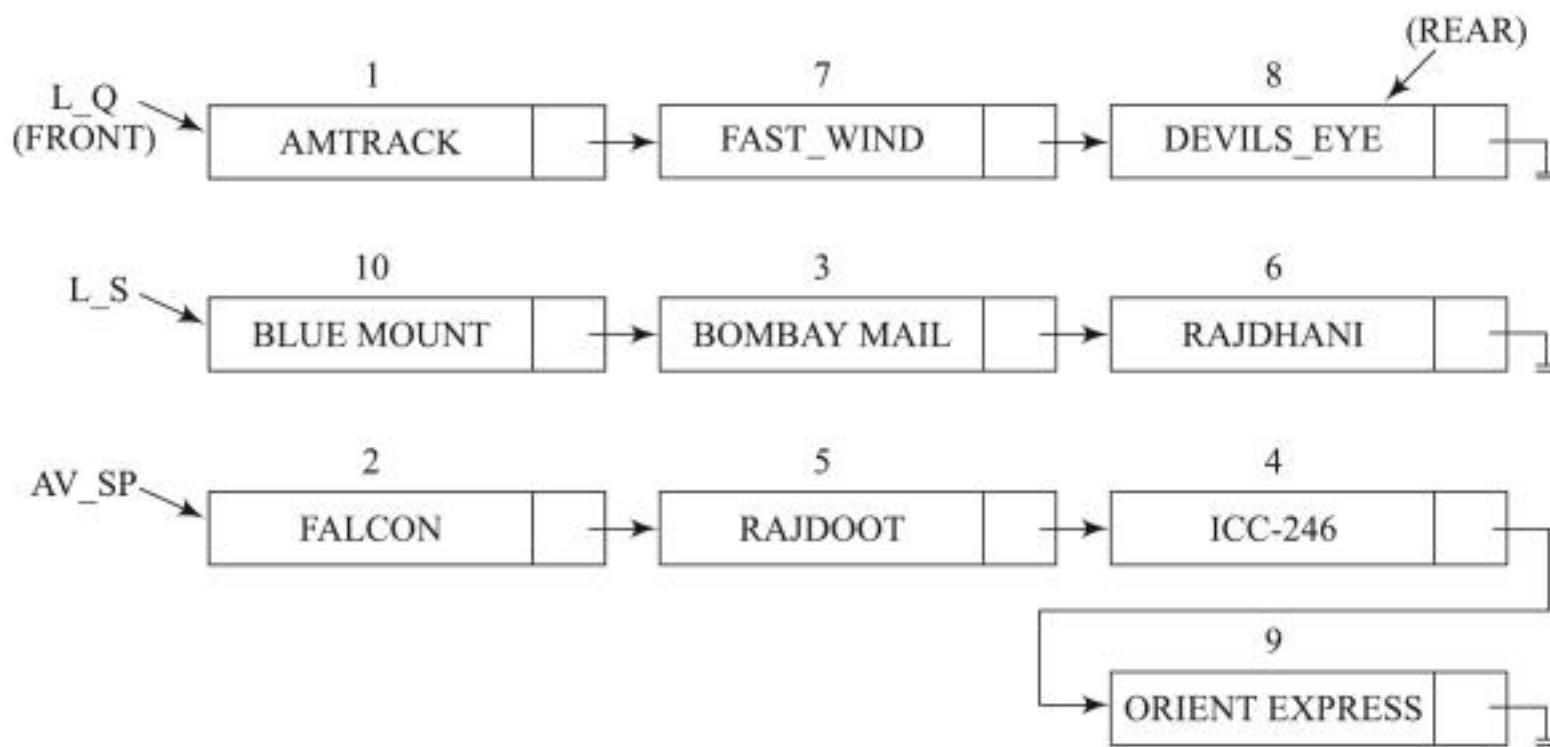
	DATA	LINK	L_Q: 1	L_S: 10	AV_SP: 2
1	AMTRACK	7			
2	FALCON	5			
3	BOMBAY_MAIL	6			
4	ICC 246	9			
5	RAJDOOT	4			
6	RAJDHANI	0			
7	FAST_WIND	8			
8	DEVILS_EYE	0			
9	ORIENT EXPRESS	0			
10	BLUE MOUNT	3			

(FRONT)

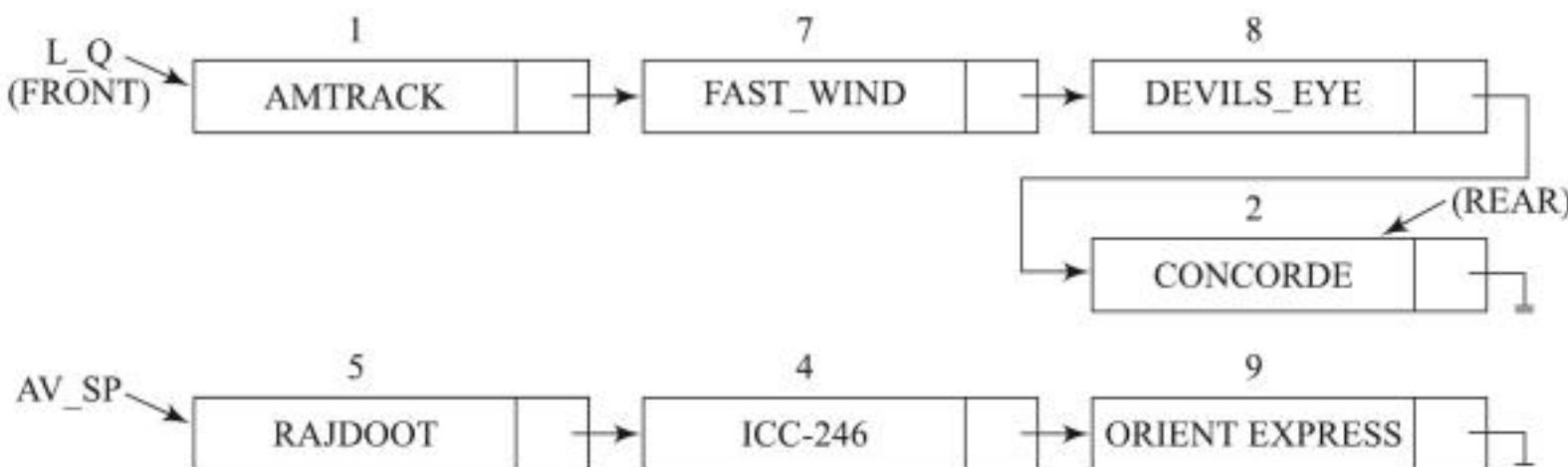
L_Q 8

(REAR)

Solution: It is easier to perform the operations on the logical representations of the lists and available space extracted from the memory, before obtaining the final memory snapshot. The lists are:

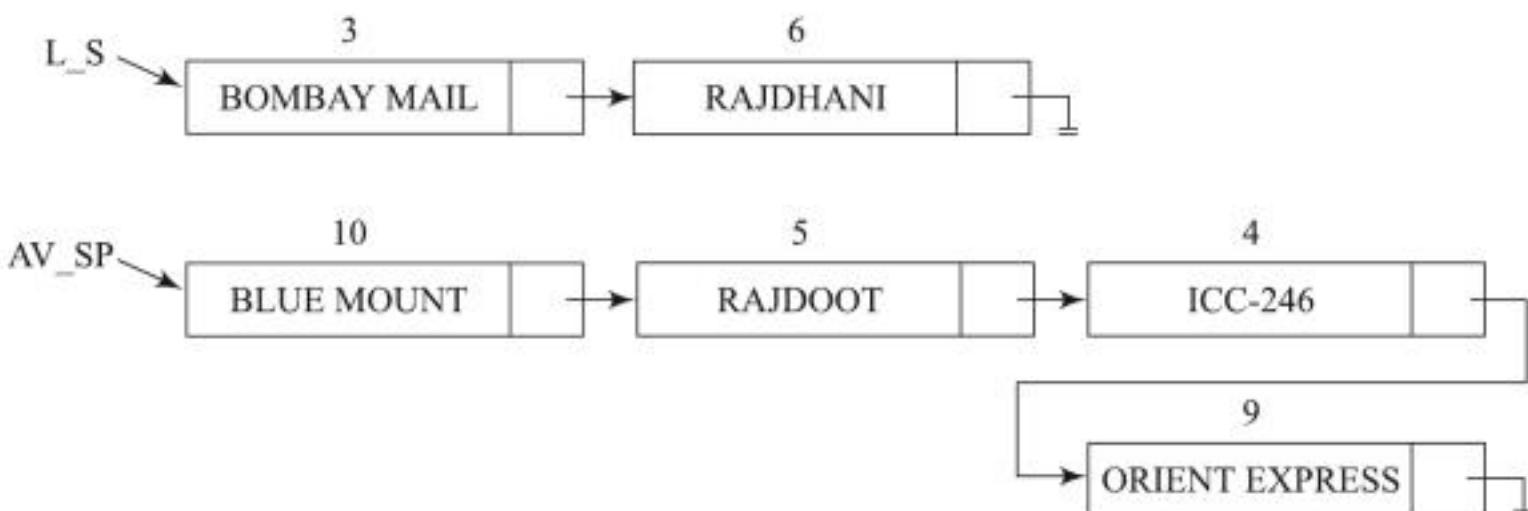


- (i) Enqueue CONCORDE into L_Q yields:



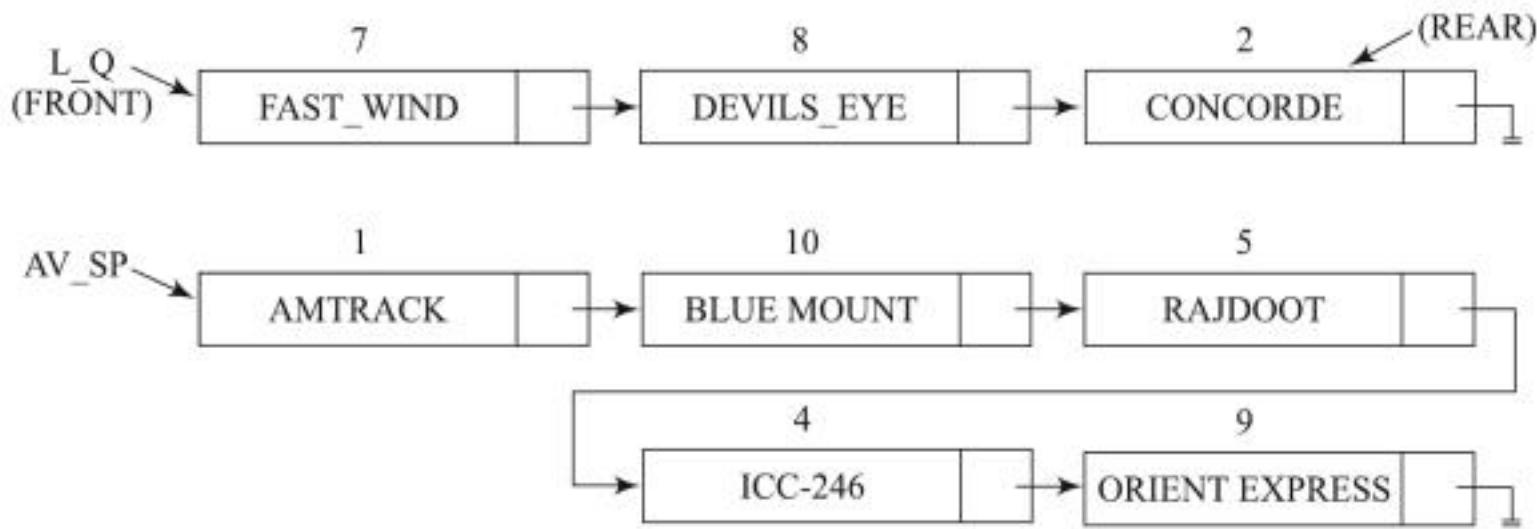
Here node 2 is popped from AV_SP to accommodate CONCORDE which is inserted at the rear of L_Q.

- (ii) Pop from L_S yields:



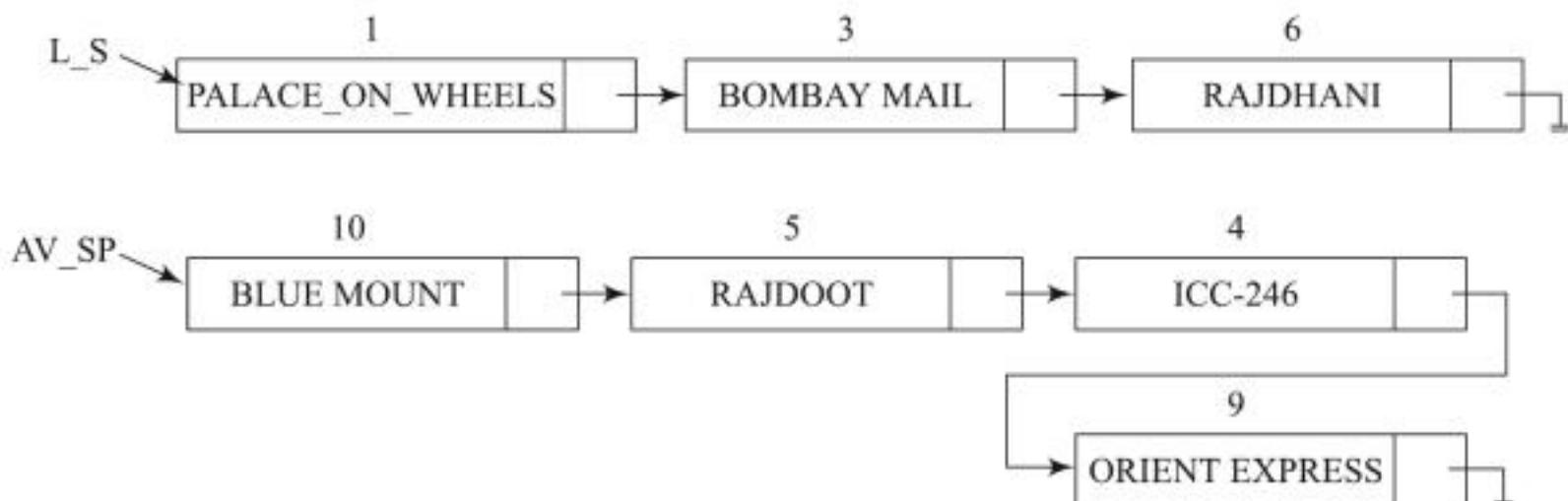
Here node 10 from L_S is deleted and pushed into AV_SP.

(iii) Dequeue from L_Q yields:



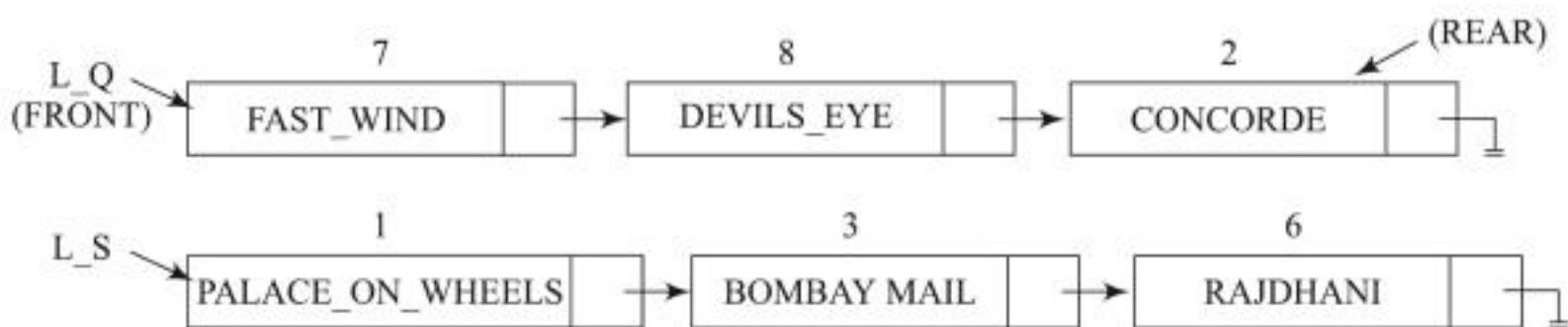
Here, node 1 from L_Q is deleted and pushed into AV_SP .

(iv) Push "PALACE_ON_WHEELS" into L_S yields:



Here, node 1 from AV_SP is popped to accommodate "PALACE_ON_WHEELS" before pushing the node into L_S .

The final lists are



The memory snapshot is given by:

	DATA	LINK		
1	PALACE ON WHEELS	3	$L_Q: \boxed{7}$	$L_S: \boxed{1}$
2	CONCORDE	0	(FRONT)	$AV_SP: \boxed{10}$
3	BOMBAY MAIL	6	$L_Q \boxed{2}$	
4	ICC-246	9	(REAR)	

5	RAJDOOT	4
6	RAJDHANI	0
7	FAST_WIND	8
8	DEVILS_EYE	2
9	ORIENT EXPRESS	0
10	BLUE MOUNT	5

Problem 7.3 Implement an abstract data type STAQUE which is a combination of a linked stack and a linked queue. Develop procedures to perform an Insert and a Delete operation, termed PUSHINS and POPDEL respectively, on a non empty STAQUE. PUSHINS inserts an element at the top or rear of the STAQUE based on an indication given to the procedure and POPDEL deletes elements from the top or front of the list.

Solution: The procedure PUSHINS performs the insertion of an element in the top or rear of the list based on whether the STAQUE is viewed as a stack or queue respectively. On the other hand, the procedure POPDEL which performs a pop or deletion of element, is common to a STAQUE, since in both the cases first element in the list alone is deleted.

```

procedure PUSHINS(WHERE, TOP, REAR, ITEM)
/* WHERE indicates whether the insertion of ITEM
is to be done as on a stack or as on a queue*/
Call GETNODE (X);
DATA (X) = ITEM;
if (WHERE = 'Stack') then {LINK (X) = TOP;
                           TOP = X;
                           }
else
                           {LINK (REAR) = X;
                            LINK (X) = Nil;
                            REAR = X;
                           }
end PUSHINS

```

```

procedure POPDEL (TOP, ITEM)
TEMP = TOP;
ITEM = DATA (TEMP); /* delete top
element of the list through
ITEM*/
TOP=LINK (TEMP);
RETURN (TEMP);
end POPDEL.

```

Problem 7.4 Write a procedure to convert a linked stack into a linked queue.

Solution: An elegant and an easy solution to the problem is to undertake the conversion by returning the addresses of the first and last nodes of the linked stack as FRONT and REAR thereby turning the linked stack into a linked queue.

```

Procedure CONVERT_LINKSTACK(TOP, FRONT, REAR)
/* FRONT and REAR are the variables which return the addresses of
the first and last node of the list converting the linked stack into
a linked queue*/

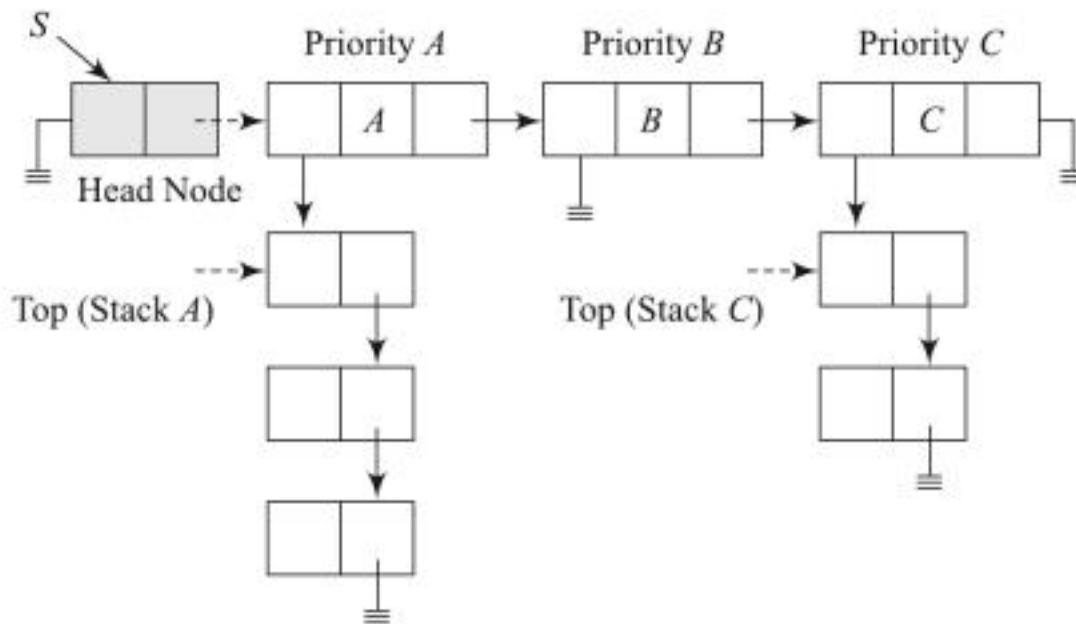
```

```

if (TOP = Nil) then Print ("Conversion not possible");
else (FRONT=TOP;
    TEMP=TOP;
    while (LINK (TEMP) Procedure != Nil)
        TEMP=LINK (TEMP);
        REAR=TEMP;
    endwhile
}
end CONVERT_LINKSTACK.

```

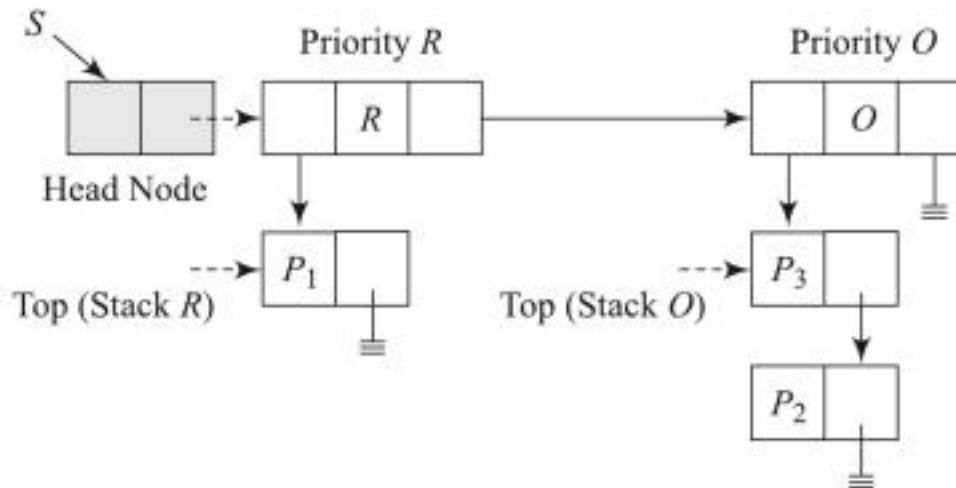
Problem 7.5 An Abstract Data Type STACKLIST is a list of linked stacks stored according to a priority factor viz., A, B, C etc, where A means highest priority, B the next and so on. Elements having the same priority are stored as a linked stack. The following is a structure of the STACKLIST S



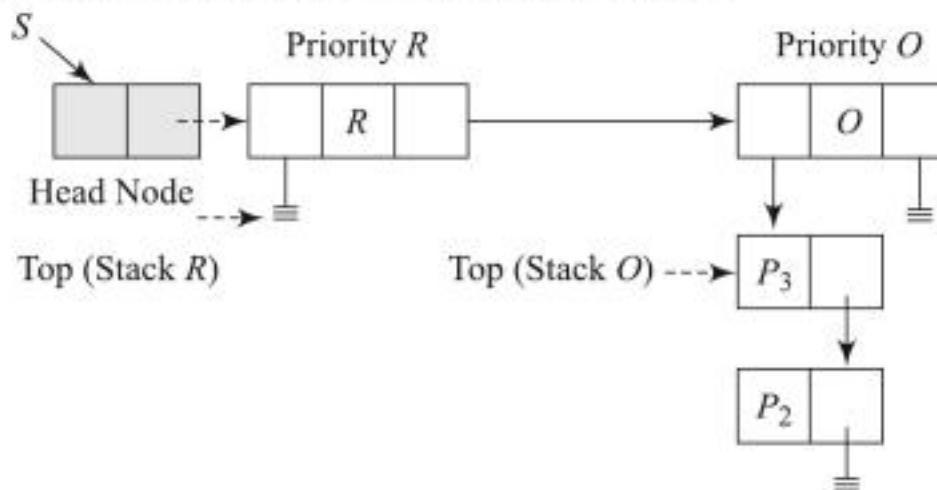
Create a STACKLIST for the following application of Process Scheduling with the processes having two priorities, viz., R (Real time) and O(Online) listed within brackets.

1. Initiate Process P_1 (R)	5. Initiate Process P_5 (O)
2. Initiate Process P_2 (O)	6. Initiate Process P_6 (R)
3. Initiate Process P_3 (O)	7. Terminate Process in Linked Stack O
4. Terminate Process in Linked Stack R	8. Initiate Process P_7 (R)

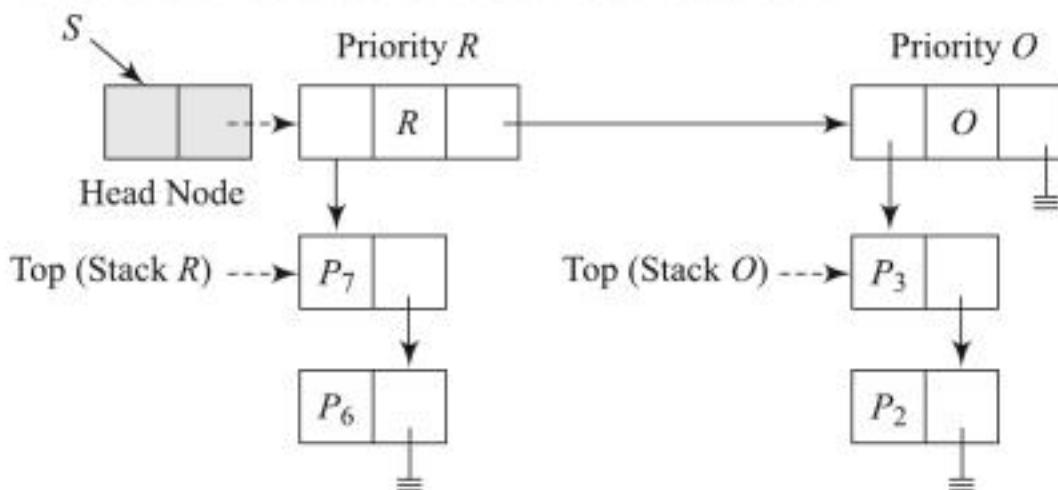
Solution: The STACKLIST at the end of Schedules 1-3 is shown as follows



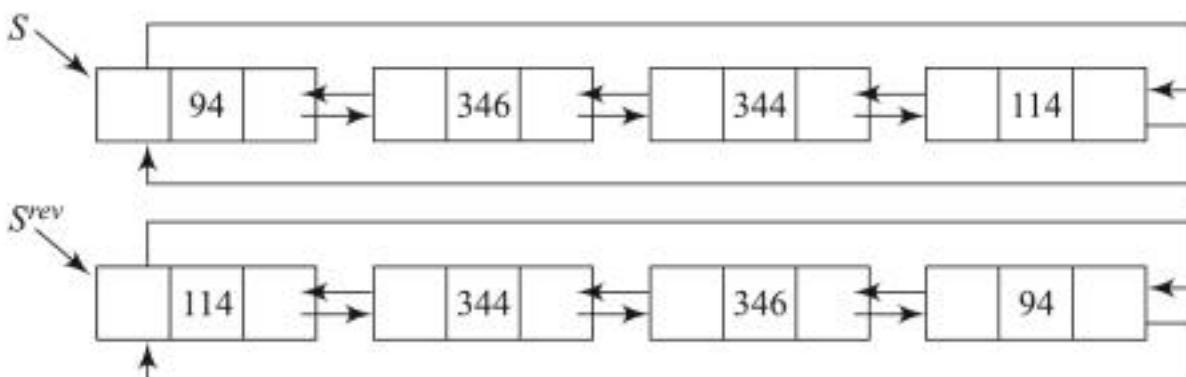
The STACKLIST at the end of Schedule 4 is as given below:



The STACKLIST at the end of Schedule 5-8 is as shown below:



Problem 7.6 Write a procedure to reverse a linked stack implemented as a doubly linked list, with the original top and bottom positions of the stack reversed as bottom and top respectively. For example, a linked stack S and its reversed version S^{rev} are shown below:

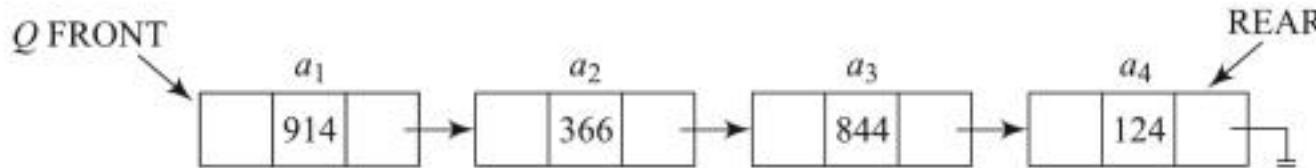


Solution: An elegant solution would be to merely swap the LLINK and RLINK pointers of each of the doubly linked list to reverse the list and remember the address of the last node in the original stack S as the TOP pointer. The procedure is given below:

```

Procedure REVERSE_STACK(TOP)
/* TEMP and HOLD are temporary variables to hold the addresses of nodes*/
TEMP=TOP;
Repeat
HOLD=LLINK(TEMP);
LLINK(TEMP)=RLINK(TEMP);
RLINK(TEMP)=HOLD;           /* Swap left and right links for each node*/
TEMP=LLINK(TEMP);          /* Move to the next node*/
until (TEMP=TOP)
TOP=RLINK(TEMP);
end REVERSE_STACK.
```

Problem 7.7 What does the following pseudocode do to the linked queue Q with the addresses of nodes, as shown below:

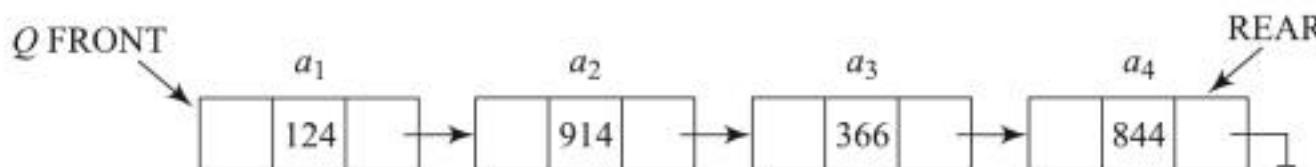


```

Procedure WHAT_DO_I_DO(FRONT, REAR)
/* HAVE, HOLD and HUG are temporary variables to hold the link or data
fields of the nodes as the case may be*/
HAVE=FRONT;
HOLD=DATA(HAVE);

while LINK(HAVE) ≠ Nil
HUG= DATA(LINK(HAVE));
DATA(LINK(HAVE))=HOLD;
HOLD=HUG;
HAVE=LINK(HAVE);
endwhile
DATA(FRONT)=HOLD;
end WHAT_DO_I_DO
```

Solution: The procedure WHAT_DO_I_DO rotates the data items of linked queue Q to obtain the resultant list given below:



Problem 7.8 Write a procedure to remove the n^{th} element (from the top) of a linked stack with the rest of the elements unchanged. Contrast this with a sequential stack implementation for the same problem (Refer Illustrative Problem 4.2 (iii) of Chapter 4).

Solution: To remove the n^{th} element leaving the other elements unchanged, a linked implementation of the stack merely calls for sliding down the list which is easily done, and for a reset of a link to remove the node concerned. The procedure is given below. In contrast, a sequential implementation as illustrated in Illustrative Problem 4.2(iii), calls for the use of another temporary stack to hold the elements popped out from the original stack before pushing them back into it.

```

Procedure REMOVE (TOP, ITEM, n)
/* The  $n^{\text{th}}$  element is removed through ITEM*/
TEMP=TOP;
COUNT=1;
while (COUNT  $\neq$  n) do
PREVIOUS=TEMP;
TEMP = LINK (TEMP);
COUNT=COUNT+1;
endwhile
LINK(PREVIOUS) =LINK(TEMP);
ITEM=DATA(TEMP);
RETURN (TEMP);
end REMOVE

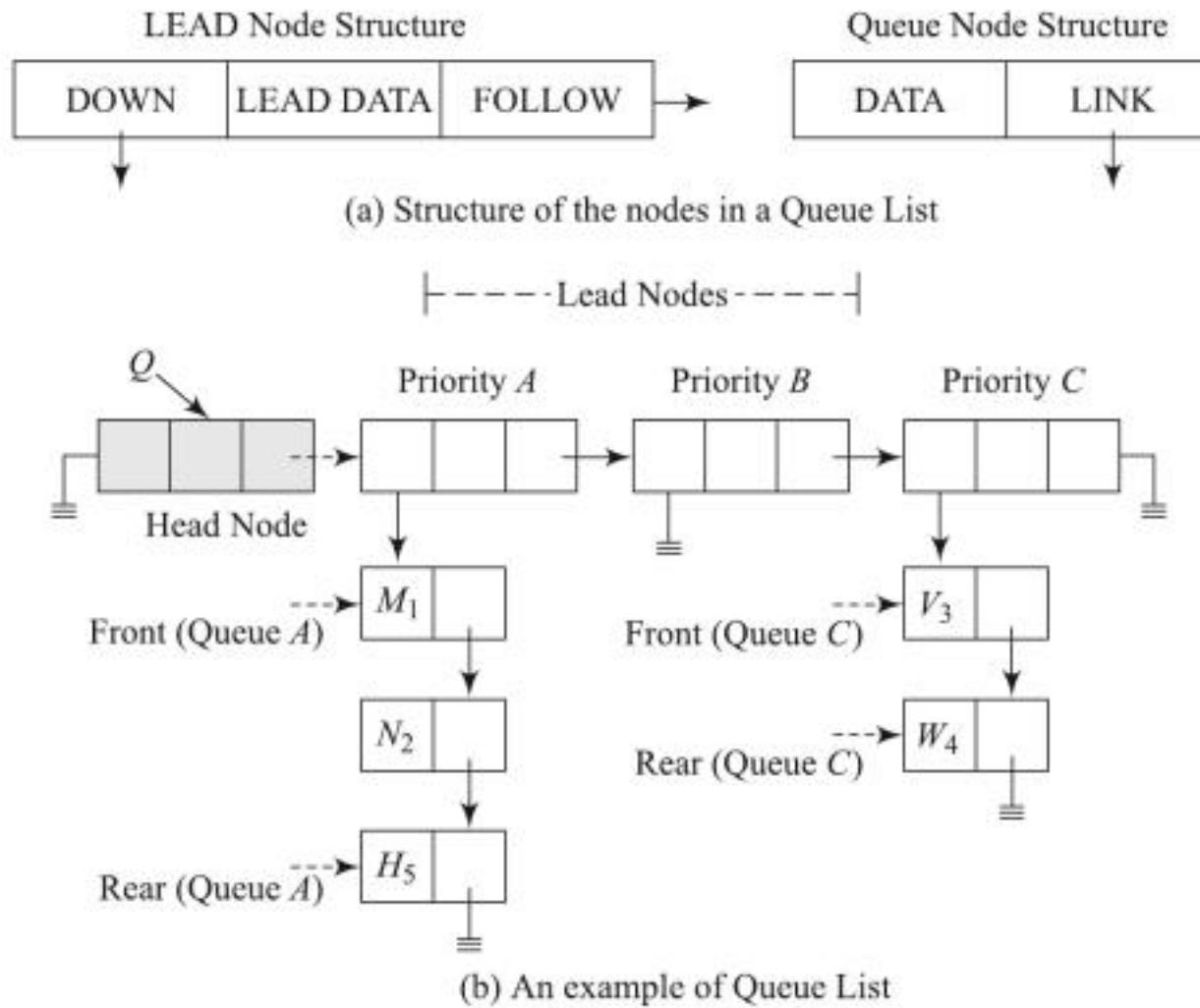
```

Problem 7.9 Given a linked stack L_S and a linked queue L_Q with equal lengths, what do the following procedures to do the lists? Here TOP is the top pointer of L_S and FRONT and REAR are the front and rear of L_Q. What are your observations regarding the functionality of the two procedures?

Procedure WHAT_IS_COOKING1 (TOP, FRONT, REAR) /* TEMP, TEMP1, TEMP2 and TEMP3 are temporary variables*/ TEMP1= FRONT; TEMP2=TOP; while (TEMP1 \neq Nil AND TEMP2 \neq Nil) do TEMP3=DATA(FRONT); DATA(FRONT)=DATA(TOP); DATA(TOP)=TEMP3; TEMP1=LINK(TEMP1); PREVIOUS=TEMP2; TEMP2= LINK(TEMP2); endwhile TEMP=TOP; TOP=FRONT; FRONT=TEMP; REAR=PREVIOUS; end WHAT_IS_COOKING 1	Procedure WHAT_IS_COOKING2 (TOP, FRONT, REAR) /* TEMP, TEMP1, TEMP2 and TEMP3 are temporary variables*/ TEMP= TOP; while (LINK(TEMP) \neq Nil) do TEMP=LINK(TEMP); endwhile TEMP1=TOP; REAR=TEMP; TOP=FRONT; FRONT=TEMP1; end WHAT_IS_COOKING2
---	---

Solution: Both the procedures swap the contents of the Linked stack L_S and linked queue L_Q. While WHAT_IS_COOKING1 does by exchanging the data items of the lists, WHAT_IS_COOKING2 does it by merely manipulating the pointers and hence is an elegant presentation.

Problem 7.10 A Queue List Q is a list of linked queues stored according to orders of priority viz., A, B, C , etc., with A accorded the highest priority and so on. The LEAD nodes serve as head nodes for each of the priority based queues. Elements with the same priority are stored as a normal linked queue. Figure I 7.10 (a-b) illustrate the node structure and an example of Queue List respectively.



	DOWN	LEAD DATA	FOLLOW		DATA	LINK	
10	604	5	7		g	0	START 16
11	26	-4	561		k	571	
12	566	1	13		a	572	
13	0	2	15		l	384	
14	3	4	591		v	0	
15	573	3	0		m	570	
16	0	-3	12		n	0	AVAILABLE SPACE 10

(c) A snap shot of a Queue List

Fig. I 7.10

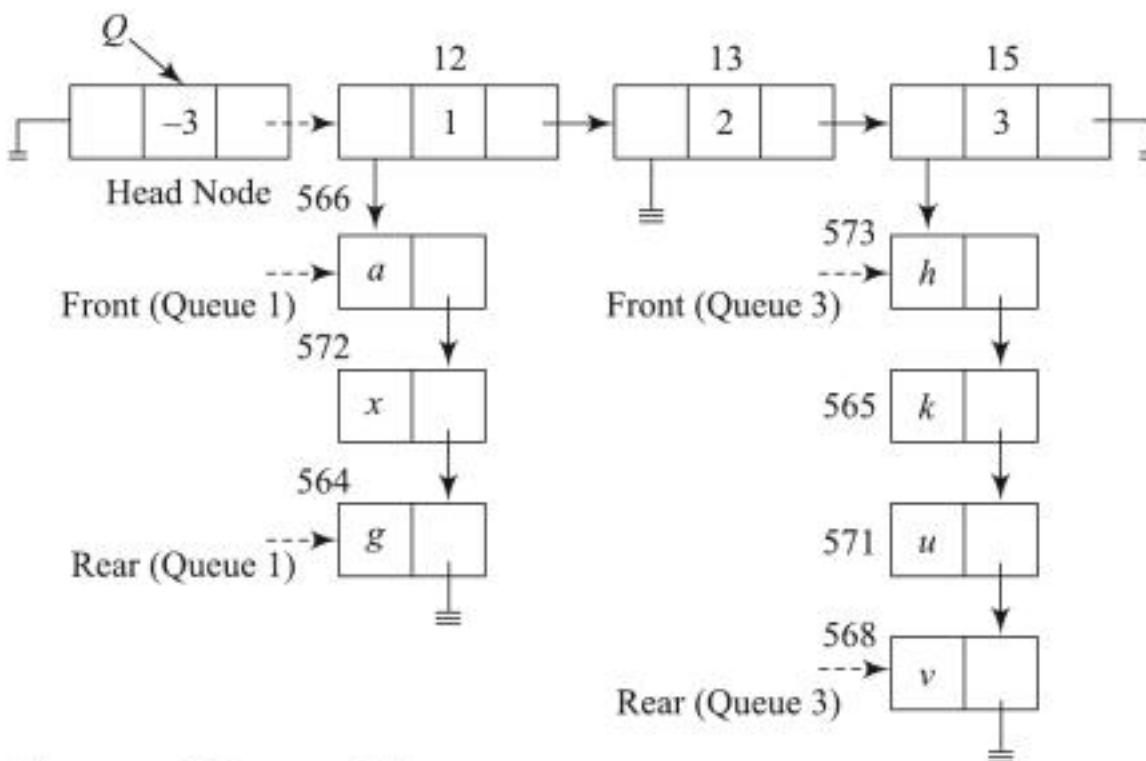
The FOLLOW link links together the head nodes of the queues and DOWN link connects it to the first node in the respective queue. The LEAD DATA field may be used to store the priority factor of the queue.

Here is a QUEUELST Q stored in the memory, a snapshot of which is shown in Fig. I 7.10(c)

There are three queues Q1, Q2, Q3 with priorities 1, 2, and 3. The Head node of QUEUELST stores the number of queues in the list as a negative number. The LEAD DATA field stores the priority factor of each of the three queues. START points to the head node of the QUEUELST and AVAILABLE SPACE the pointer to the free storage pool.

Obtain the QUEUELST by tracing the lead nodes and nodes of the linked queues.

Solution: The structure of the QUEUELST is as shown below:



Review Questions

The following is a snap shot of a memory which stores a linked stack VEGETABLES and a linked queue FRUITS beginning at the respective addresses. Answer the following questions with regard to operations on the linked stack and queue, each of which is assumed to be independently performed on the original linked stack and queue.

	DATA	LINK
1	CABBAGE	7
2	CUCUMBER	5
3	PEAR	6
4	ONION	9
5	ORANGE	4
6	PEACH	0
7	CELERY	8
8	CARROTS	0
9	LEMON	0
10	PLUM	3

FRUITS	VEGETABLES	AV_SP
FRONT	10	TOP
REAR:	6	1

1. Inserting PAPAYA into the linked queue FRUITS results in the following changes to the FRONT, REAR and AV_SP pointers respectively, as given in:
 (a) 10 2 2 (b) 2 6 2 (c) 2 6 5 (d) 10 2 5
2. Undertaking pop operation on VEGETABLES results in the following changes to the TOP and AV_SP pointers respectively, as given in:
 (a) 7 1 (b) 7 2 (c) 8 2 (d) 8 1
3. Undertaking delete operation on FRUITS results in the following changes to the FRONT, REAR and AV_SP pointers respectively, as given in :
 (a) 3 6 2 (b) 10 3 6 (c) 3 6 10 (d) 10 3 2
4. Pushing TURNIPS into VEGETABLES results in the following changes to the TOP and AV_SP pointers respectively, as given in:
 (a) 2 5 (b) 2 9 (c) 1 5 (d) 1 9
5. After the push operation of TURNIPS into VEGETABLES (undertaken in Review Question 7.4), DATA(2) = ----- and DATA (LINK (2)) = -----
 (a) TURNIPS and CABBAGE (b) CUCUMBER and CABBAGE
 (c) TURNIPS and CUCUMBER (d) CUCUMBER and ORANGE
6. What are the merits of linked stacks and queues over their sequential counterparts?
7. How is the memory storage pool associated with a linked stack data structure for its operations?
8. How are push and pop operations implemented on a linked stack?
9. What are traversable queues?
10. Outline the node structure and a linked queue to represent the polynomial:

$$17x^5 + 18x^2 + 9x + 89$$
11. Trace Algorithm 7.5 on the following expression to check whether parentheses are balanced:

$$((X + Y + Z) * H) + (D * T)) - 2$$



Programming Assignments

1. Execute a program to implement a linked stack to check for the balancing of the following pairs of symbols in a Pascal program. The name of the source Pascal program is the sole input to the program.
 Symbols: *begin end* , *()*, *[]*, *{ }*.
 (i) Output errors encountered during mismatch of symbols.
 (ii) Modify the program to set right the errors.
2. Evaluate a postfix expression using a linked stack implementation.
3. Implement the simulation of a time sharing system discussed in Chapter 5, Sec. 5.5, using linked queues.
4. Develop a program to implement a Queue List (Refer Illustrative Problem 7.10) which is a list of linked queues stored according to an order of priority.
 Test for the insertion and deletion of the following jobs with their priorities listed within brackets, on a Queue List JOB_MANAGER with three queues A, B and C listed according to their order of priorities:

1.	Insert Job J_1 (A)	6.	Insert Job J_5 (C)
2.	Insert Job J_2 (B)	7.	Insert Job J_6 (C)
3.	Insert Job J_3 (A)	8.	Insert Job J_7 (A)
4.	Insert Job J_4 (B)	9.	Delete Queue C
5.	Delete Queue B	10.	Insert Job J_8 (A)

5. Develop a program to simulate a calculator which performs the addition, subtraction, multiplication and division of polynomials.



TREES AND BINARY TREES

8

Introduction

8.1

In Chapters 3–5 we discussed the sequential data structures of arrays, stacks and queues. These are termed as *linear data structures* as they are inherently uni-dimensional in structure. In other words, the items form a sequence or a linear list. In contrast, the data structures of trees and graphs are termed *non linear data structures* as they are inherently two dimensional in structure. Trees and their variants, binary trees and graphs, have emerged as truly powerful data structures registering immense contribution to the development of efficient algorithms or efficient solutions to various problems in science and engineering.

In this chapter, we first discuss the tree data structure, the basic terminologies and representation schemes. An important variant of the tree viz., binary tree, its basic concepts, representation schemes and traversals are elaborately discussed next. A useful modification to the binary tree viz., threaded binary tree is introduced. Finally, expression trees and its related concepts are discussed as an application of binary trees.

Trees: Definition and Basic Terminologies

8.2

Definition of trees

A *tree* is defined as a finite set of one or more nodes such that

- there is a specially designated *node* called the *root* and
- the rest of the nodes could be partitioned into t disjoint sets ($t \geq 0$) each set representing a tree T_i , $i = 1, 2, \dots, t$ known as *subtree* of the tree.

A *node* in the definition of the tree represents an item of information, and the links between the nodes termed as *branches*, represent an association between the items of information. Figure 8.1 illustrates a tree.

The definition of the tree emphasizes on the aspect of (i) *connectedness* and (ii) absence of closed loops or what are termed *cycles*. Beginning from the root node, the structure of the tree

- 8.1 Introduction
- 8.2 Trees: Definition and Basic Terminologies
- 8.3 Representation of Trees
- 8.4 Binary Trees: Basic Terminologies and Types
- 8.5 Representation of Binary Trees
- 8.6 Binary Tree Traversals
- 8.7 Threaded Binary Trees
- 8.8 Applications

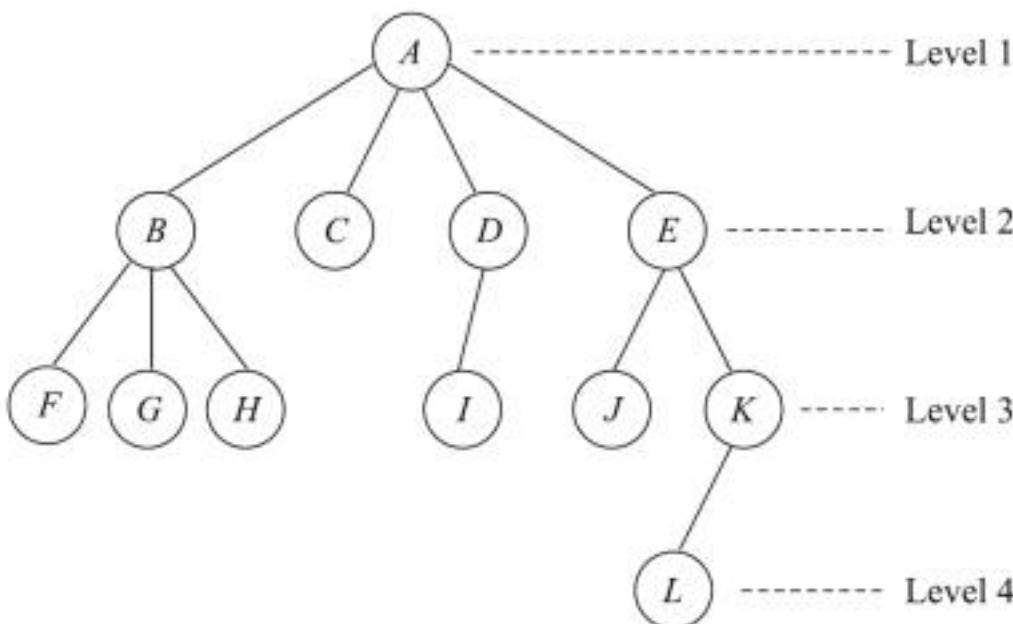


Fig. 8.1 An example tree

permits connectivity of the root to every other node in the tree. In general, any node is reachable from anywhere in the tree. Also, with branches providing the links between the nodes, the structure ensures that no set of nodes link together to form a closed loop or a cycle.

Basic terminologies of trees

There are several basic terminologies associated with the tree. The specially designated node called root has already been introduced in the definition. The number of subtrees of a node is known as the *degree of the node*. Nodes that have zero degree are called *leaf nodes* or *terminal nodes*. The rest of them are called as *non terminal nodes*. These nodes which hang from branches emanating from a node are known as *children* and the node from which the branches emanate is known as the *parent node*. Children of the same parent node are referred to as *siblings*. The *ancestors* of a given node are those nodes that occur on the path from the root to the given node. The *degree of a tree* is the maximum degree of the node in the tree. The level of a node is defined by letting the root to occupy level 1 (some authors let the root occupy level 0). The rest of the nodes occupy various levels depending on their association. Thus if a parent node occupies level i , its children should occupy level $i+1$. This renders the tree to have a *hierarchical structure* with root occupying the top most level of 1. The *height* or *depth* of a tree is defined to be the maximum level of any node in the tree. Some authors define depth of a node to be the length of the longest path from the root node to that node, which yields the relation,

$$\text{depth of the tree} = \text{height of the tree} - 1$$

A *forest* is a set of zero or more disjoint trees. The removal of the root node from a tree results in a forest (of its subtrees!).

In Fig. 8.1, A is the root node. The degree of node E is 2 and L is 0. F, G, H, C, I, J and L are leaf or terminal nodes and all the remaining nodes are non leaf or non terminal nodes. Nodes F, G and H are children of B and B is a parent node. Nodes J, K and nodes F, G, H are sibling nodes with E and B as their respective parents. For the node L, nodes A, E and K are ancestors. The degree of the tree is 4 which is the maximum degree reported by node A. While node A which is the root node occupies level 1, its children B, C, D and E occupy level 2 and so on. The height of the tree is its maximum level which is 4. Removal of A yields a forest of four disjoint (sub) trees viz., {B, F, G, H}, {C}, {D, I} and {E, J, K, L}.

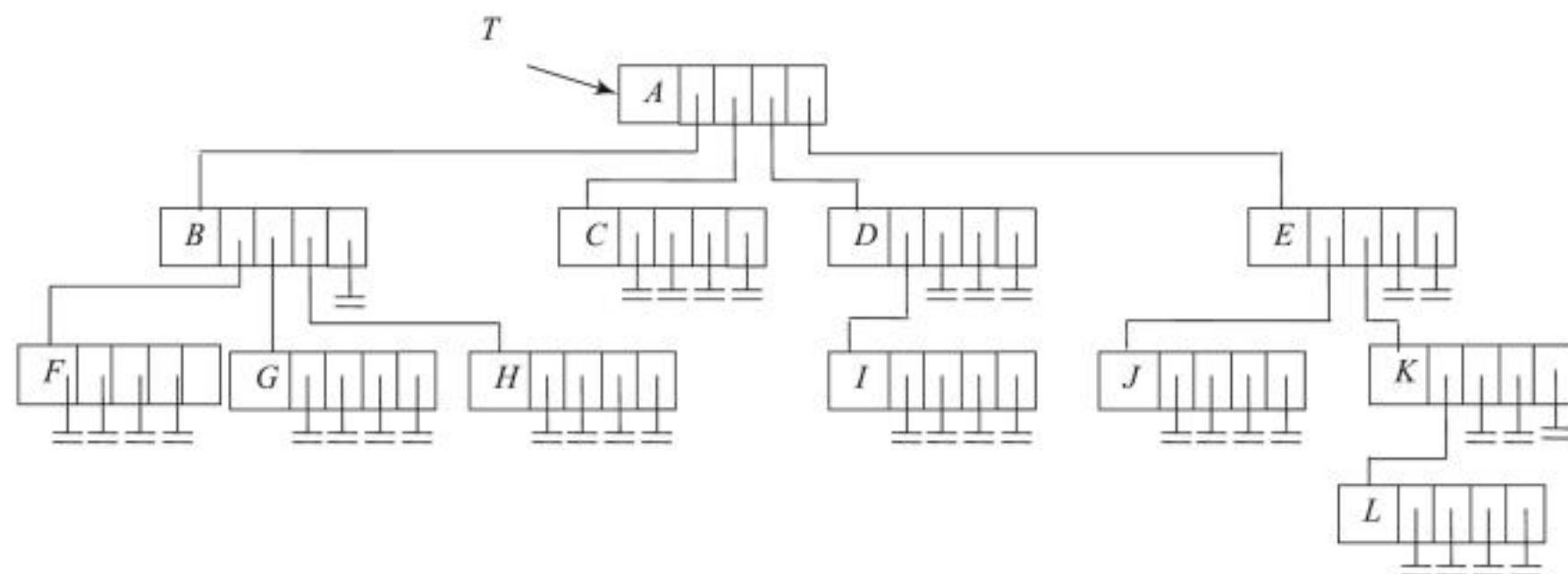
Representation of Trees

8.3

Though trees are better understood in their pictorial forms, a common representation of a tree to suit its storage in the memory of a computer, is a *list*. The tree of Fig. 8.1 could be represented in its list form as $(A (B(F,G,H), C, D(I), E(J,K(L))))$. The root node comes first followed by the list of subtrees of the node. This is repeated for each subtree in the tree. This list form of a tree, paves way for a naïve representation of the tree as a linked list. The node structure of the linked list is shown in Fig. 8.2(a).

DATA	LINK 1	LINK 2	...	LINK n
------	--------	--------	-----	----------

(a) General node structure



(b) Linked list representation of the tree shown in Fig. 8.1

Fig. 8.2 Linked list representation of a tree

The DATA field of the node stores the information content of the tree node. A fixed set of LINK fields accommodate the pointers to the child nodes of the given node. In fact the *maximum number of links the node would require is equal to the degree of the tree*. The linked representation of the tree shown in Fig. 8.1 is illustrated in Fig. 8.2 (b). Observe the colossal wastage of space by way of null pointers!

An alternative representation would be to use a node structure as shown in Fig. 8.3(a). Here TAG = 1 indicates that the next field (DATA / DOWN LINK) is occupied by data (DATA) and TAG = 0 indicates that the same is used to hold a link (DOWN LINK). The node structure of the linked list holds a DOWNLINK whenever it encounters a child node which gives rise to a subtree. Thus the root node A has four child nodes, three of which viz., B, D and E give rise to subtrees. Note the DOWN LINK active fields of the nodes in these cases with TAG set to 0. In contrast, observe the linked list node corresponding to C which has no subtree. The DATA field records C with TAG set to 1.

Example 8.1 We illustrate a tree structure in the organic evolution which deals with the derivation of new species of plants and animals from the first formed life by descent with modification. Figure 8.4 (a) illustrates the tree and Fig. 8.4 (b) shows its linked representation.

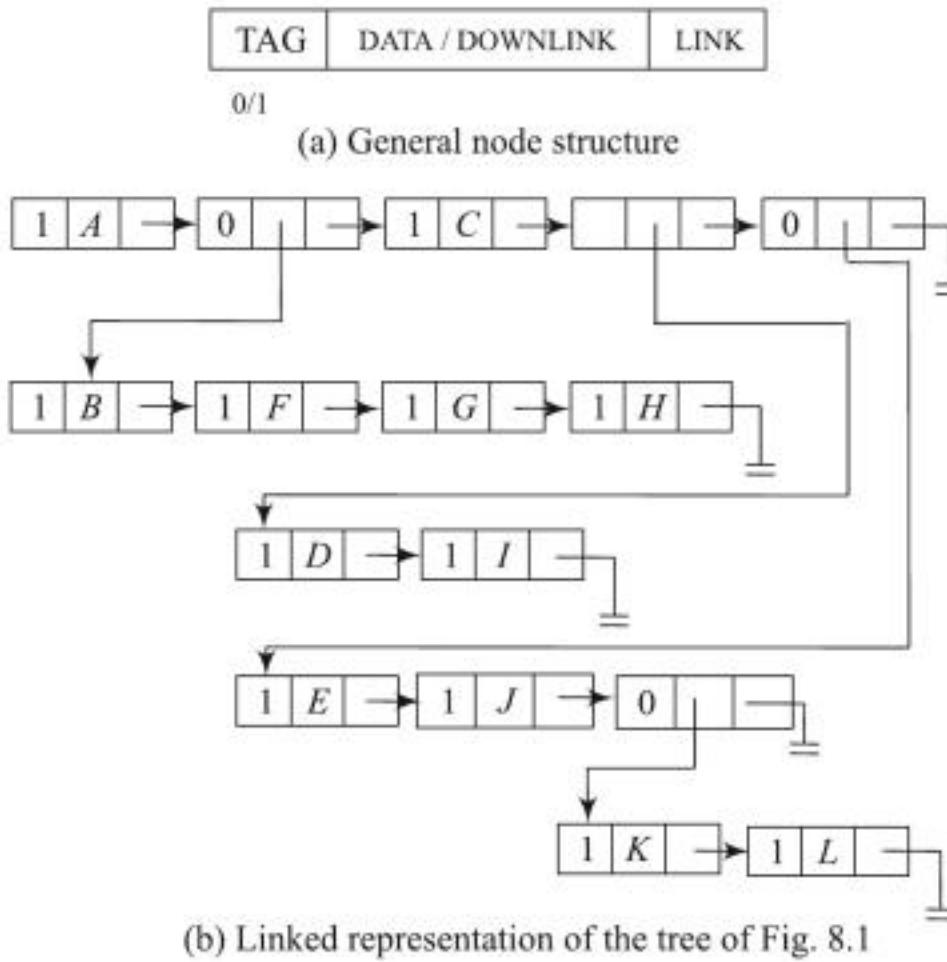


Fig. 8.3 An alternative elegant linked representation of a tree

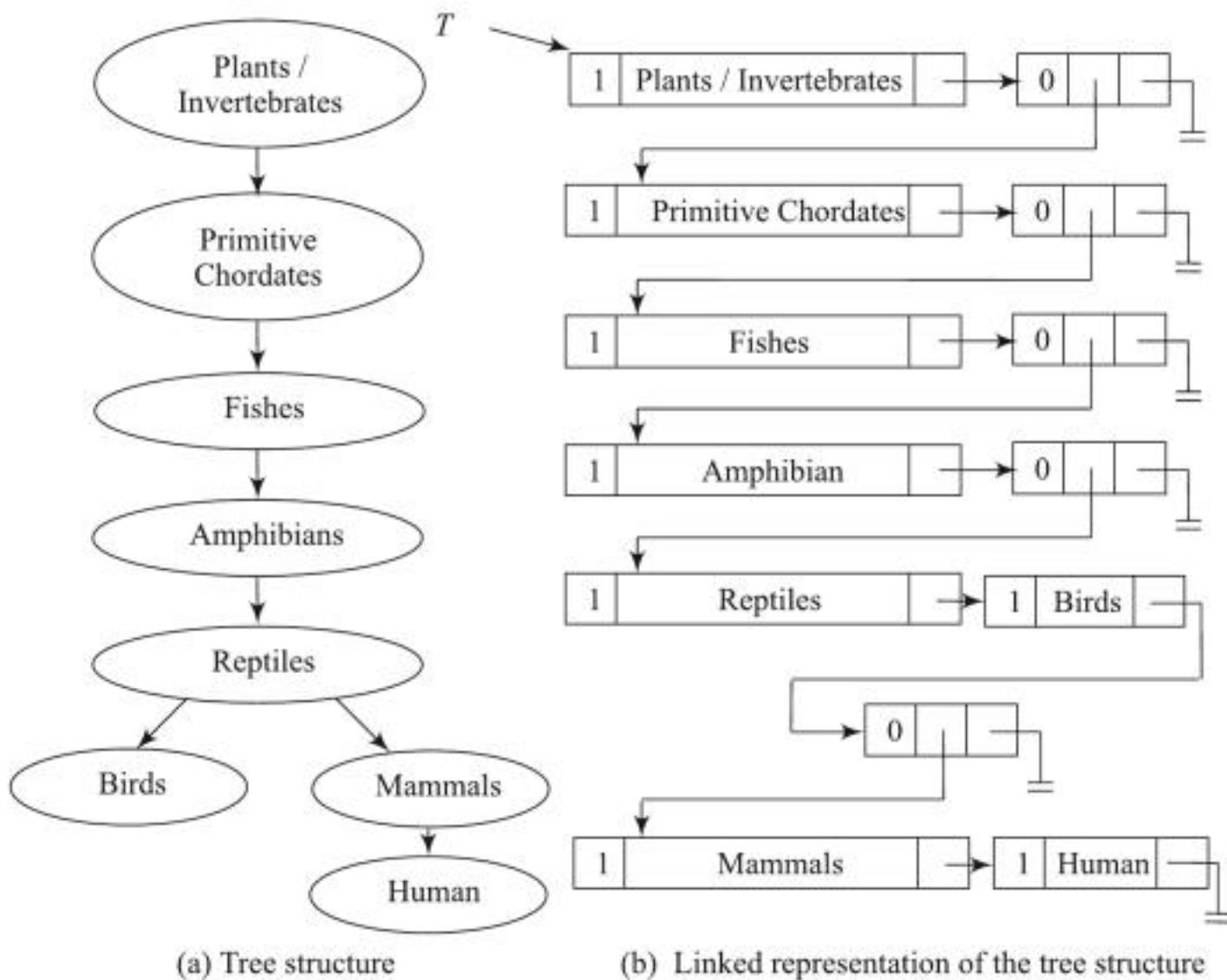


Fig. 8.4 Tree structure of organic evolution

Binary Trees: Basic Terminologies and Types

8.4

Basic terminologies

A **binary tree** has the characteristic of all nodes having at most two branches, that is, all nodes have a *degree of at most 2*. A binary tree can therefore be *empty* or consist of a root node and two disjointed binary trees termed *left subtree* and *right subtree*. Figure 8.5 illustrates a binary tree.

It is essential that the distinction between trees and binary trees are brought out clearly. While a binary tree can be empty with zero nodes, a tree can never be empty. Again while the ordering of the subtrees in a tree is immaterial, in a binary tree the distinction of left and right subtrees are very clearly maintained. All other terminologies applicable to trees such as levels, degree, height, leaf nodes, parent, child, siblings etc. are also applicable to binary trees. However, there are some important observations regarding binary trees.

- (i) The maximum number of nodes on level i of a binary tree is 2^{i-1} , $i \geq 1$.
- (ii) The maximum number of nodes in a binary tree of height h is $2^h - 1$, $h \geq 1$. (for proof refer Illustrative Problem 6 of Chapter 8)
- (iii) For any non empty binary tree, if t_0 is the number of terminal nodes and t_2 is the number of nodes of degree 2, then $t_0 = t_2 + 1$ (for proof refer Illustrative Problem 8.7)

These observations could be easily verified on the binary tree shown in Fig. 8.5. The maximum number of nodes on level 3 is $2^{3-2} = 2^2 = 4$. Also with the height of the binary tree being 3, the maximum number of nodes = $2^3 - 1 = 7$. Again $t_0 = 4$ and $t_2 = 3$ which yields $t_0 = t_2 + 1$.

Types of binary trees

A binary tree of height h which has all its permissible maximum number of nodes viz., $2^h - 1$ intact is known as a *full binary tree of height h*. Figure 8.6(a) illustrates a full binary tree of height 4. Note the specific method of numbering the nodes.

A binary tree with n' nodes and height h is *complete* if its nodes correspond to the nodes which are numbered 1 to n ($n' \leq n$) in a full binary tree of height h . In other words, a *complete binary tree* is one in which its nodes follow a sequential numbering that increments from a left-to-right and top-to-bottom fashion. A full binary tree is therefore a special case of a complete binary tree. Also, the height of a complete binary tree with n elements has a height h given by $h = \lceil \log_2 (n + 1) \rceil$. A complete binary tree obeys the following properties with regard to its node numbering:

- (i) If a parent node has a number i then its left child has the number $2i$ ($2i \leq n$). If $2i > n$ then i has no left child.
- (ii) If a parent node has a number i , then its right child has the number $2i + 1$ ($2i + 1 \leq n$). If $2i + 1 > n$ then i has no right child.
- (iii) If a child node (left or right) has a number i then the parent node has the number $\lfloor i/2 \rfloor$ if $i \neq 1$. If $i = 1$ then i is the root and hence has no parent.

In the full binary tree of height 4 illustrated in Fig. 8.6(a), observe how the parent-child numbering is satisfied. For example, consider node s (number 4), its left child w has the number $2*4=8$ and its right child has the number $2*4+1=9$. Again the parent of node v (number 7) is the node with number $\lfloor 7/2 \rfloor = 3$ (i.e.) node 3 which is r .

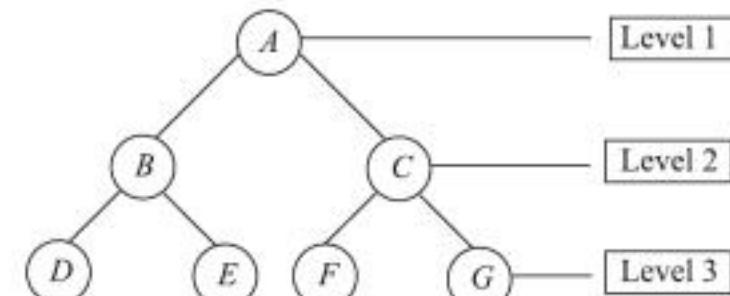
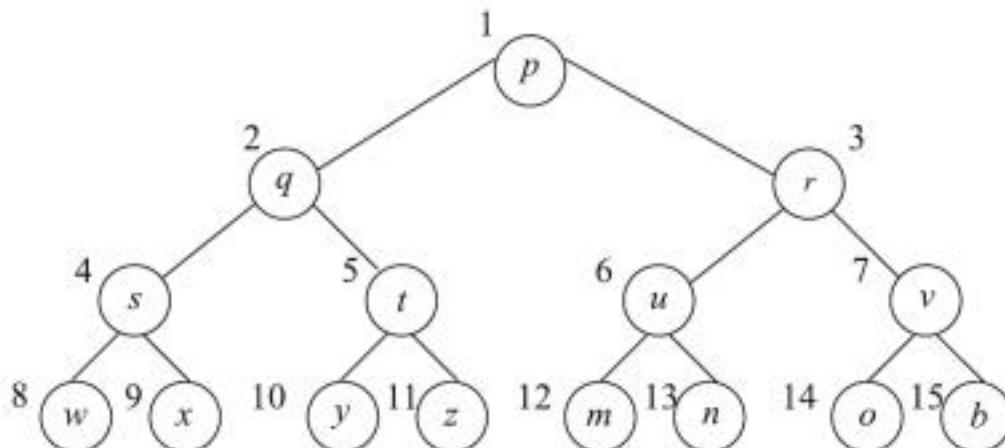
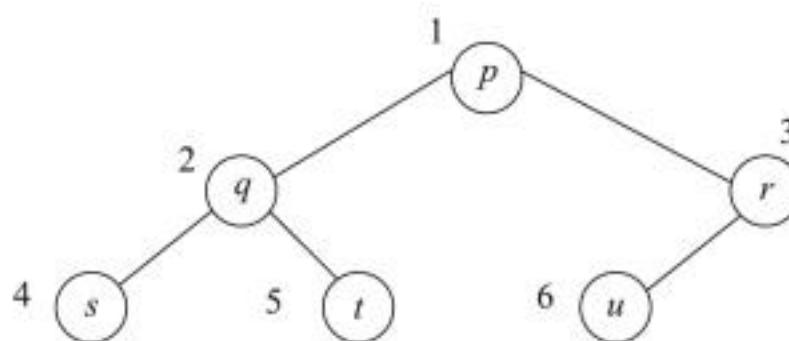


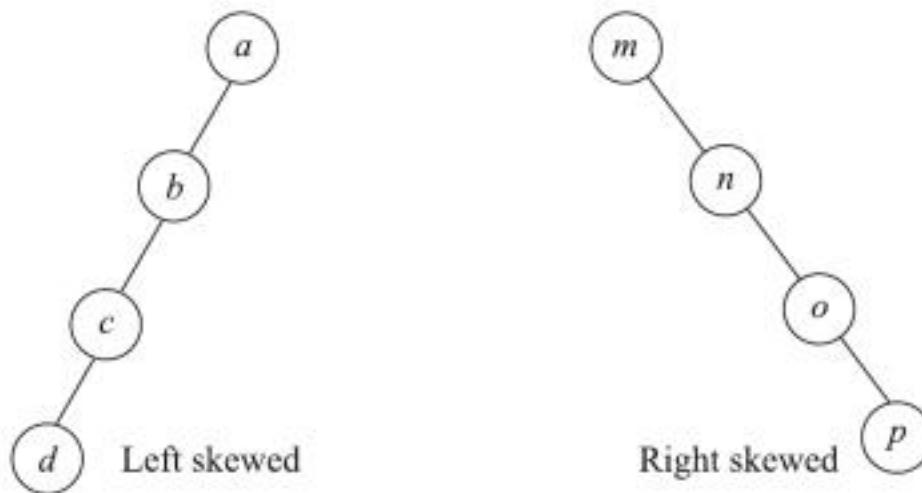
Fig. 8.5 An example of binary tree



(a) Full binary tree of height 4



(b) A complete binary tree of height 3



(c) Skewed binary tree

Fig. 8.6 Examples of full binary tree, complete binary tree and skewed binary trees

Figure 8.6(b) illustrates an example complete binary tree. A binary tree which is dominated solely by left child nodes or right child nodes is called a **skewed binary tree** or more specifically **left skewed binary tree** or **right skewed binary tree** respectively. Figure 8.6(c) illustrates examples of skewed binary trees.

Representation of Binary Trees

8.5

A binary tree could be represented using a sequential data structure (arrays) as well as linked data structure.

Array representation of binary trees

To represent the binary tree as an array, the sequential numbering system emphasized by a complete binary tree comes in handy. Consider the binary tree shown in Fig. 8.7(a). The array representation is as shown in Fig. 8.7(b). The association of numbers pertaining to parent and left/right child nodes makes it convenient to access the appropriate cells of the array. However, the missing nodes in the binary tree and hence the corresponding array locations, are left empty in the array. This obviously leads to a lot of wastage of space. However, the array representation ideally suits a full binary tree due to its non wastage of space.

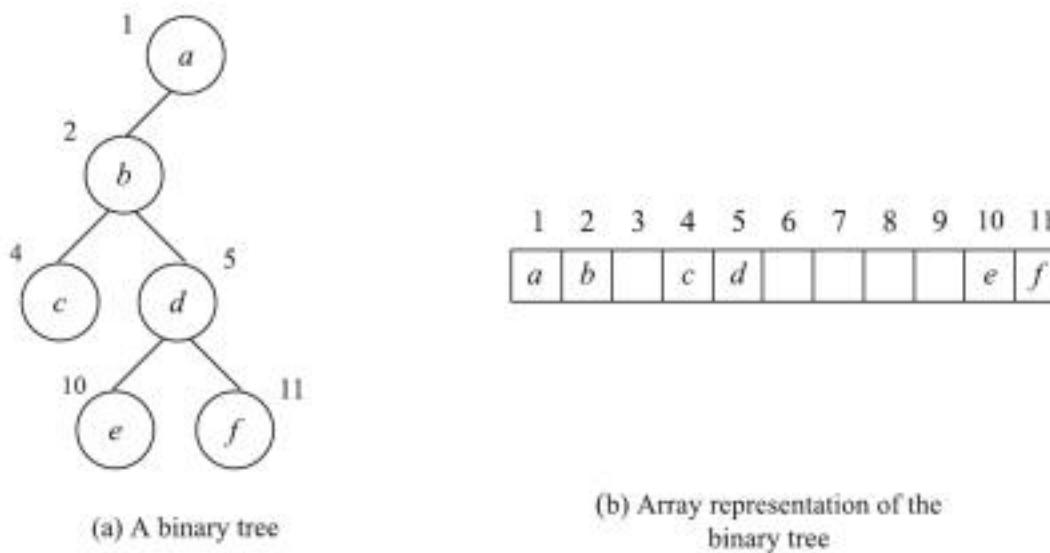


Fig. 8.7 Array representation of a binary tree

Linked representation of binary trees

The linked representation of a binary tree has the node structure shown in Fig. 8.8(a). Here, the node, besides the DATA field, needs two pointers LCHILD and RCHILD to point to the left and right child nodes respectively. The tree is accessed by remembering the pointer to the root node of the tree.

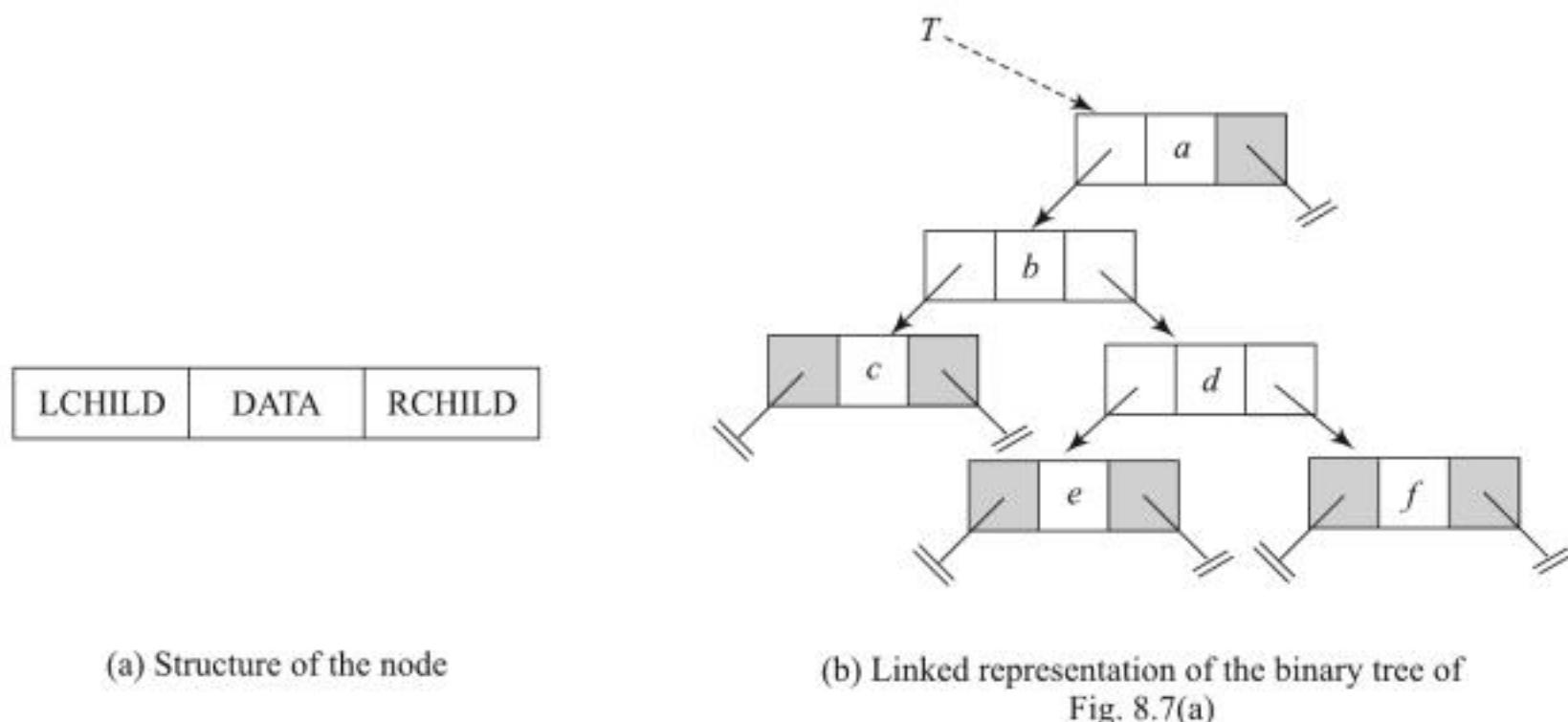


Fig. 8.8 Linked representation of a binary tree

In the binary tree T shown in Fig. 8.8(b), LCHILD (T) refers to the node storing b and RCHILD (LCHILD (T)) refers to the node storing d and so on. The following are some of the important observations regarding the linked representation of a binary tree:

- If a binary tree has n nodes then the number of pointers used in its linked representation is $2 * n$.
- The number of null pointers used in the linked representation of a binary tree with n nodes is $n + 1$.

However, in a linked representation it is difficult to determine a parent given a child node. In any case if an application so requires, a fourth field PARENT may also be included in the structure.

Binary Tree Traversals

8.6

An important operation that is performed on a binary tree is its *traversal*. A traversal of a binary tree is where its nodes are visited in a particular but repetitive order, rendering a linear order of the nodes or information represented by them.

A traversal is governed by three actions, viz. *Move left* (L), *Move Right* (R) and *Process Node* (P). In all, it yields six different combinations of LPR, LRP, PLR, PRL and RLP. Of these, three have emerged significant in computer science. They are,

- LPR — *Inorder traversal*
- LRP — *Postorder traversal*
- PLR — *Preorder traversal*.

The algorithms for each of the traversals are elaborated here.

Inorder Traversal

The traversal keeps moving left in the binary tree until one can move no further, processes the node and moves to the right to continue its traversal again. In the absence of any node to the right, it retracts backwards by a node and continues the traversal.

Algorithm 8.1 illustrates a recursive procedure to perform inorder traversal of a binary tree. For clarity of application, the action *Process Node* (P) is interpreted as *Print node*. Observe the recursive procedure reflect the maxim LPR repetitively. Example 8.2 illustrates the inorder traversal of the binary tree shown in Fig. 8.9.

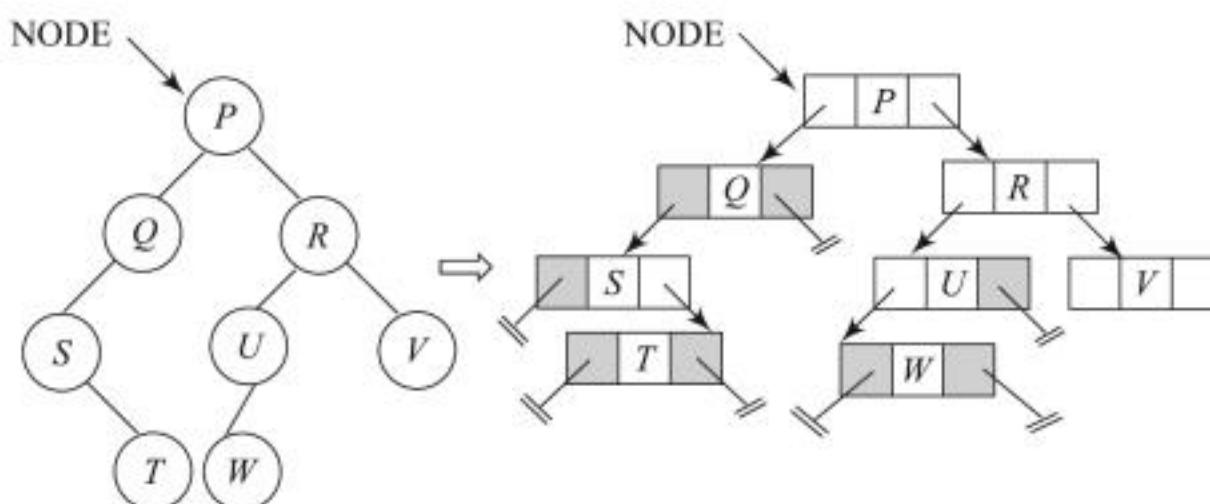


Fig. 8.9 Binary tree to demonstrate Inorder, Postorder and Preorder traversals

Example 8.2 An easy method to obtain the traversal would be to run one's fingers on the binary tree with the maxim: *move left until no more nodes, process node, then move right and continue the traversal.*

An alternative method is to trace the recursive steps of the algorithm using the following scheme:

Algorithm 8.1: Recursive procedure to perform Inorder traversal of a binary tree

```

Procedure INORDER_TRAVERSAL (NODE)
    /* NODE refers to the Root node of the binary tree
       in its first call to the procedure. Root node is the
       starting point of the traversal */
    If NODE ≠ NIL then
        { call INORDER_TRAVERSAL (LCHILD(NODE));
          /* Inorder traverse the left subtree (L) */
          Print (DATA (NODE));
          /* Process node (P) */
          call INORDER_TRAVERSAL (RCHILD(NODE));
          /* Inorder traverse the right subtree (R) */
        }
    end INORDER_TRAVERSAL.

```

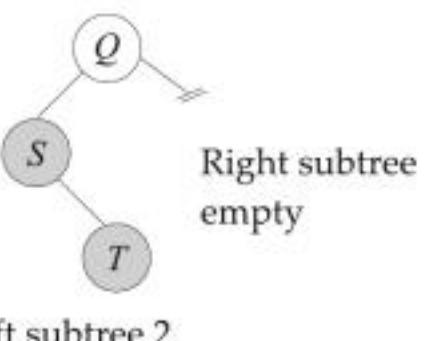
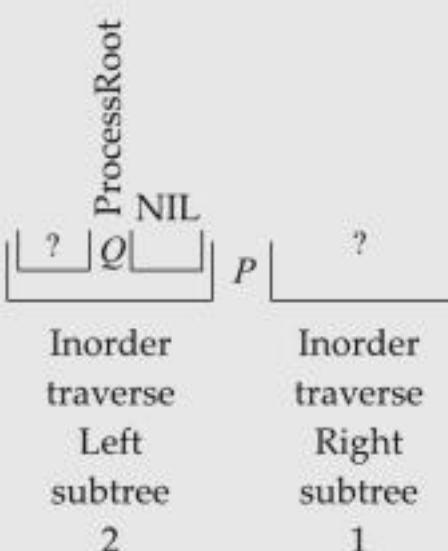
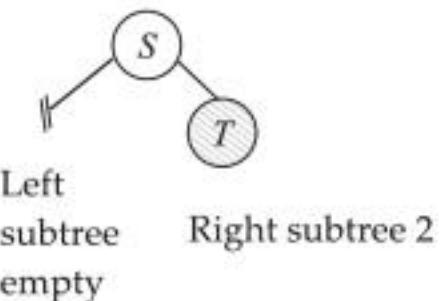
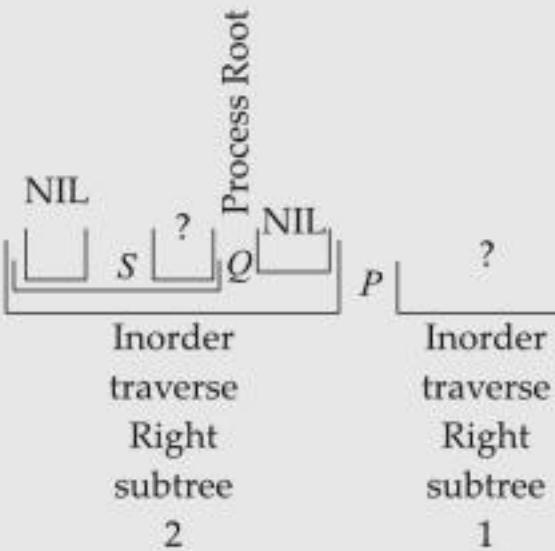
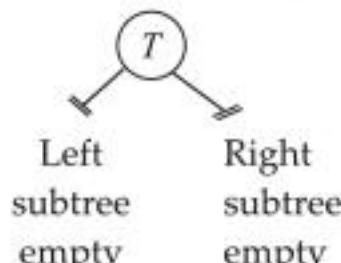
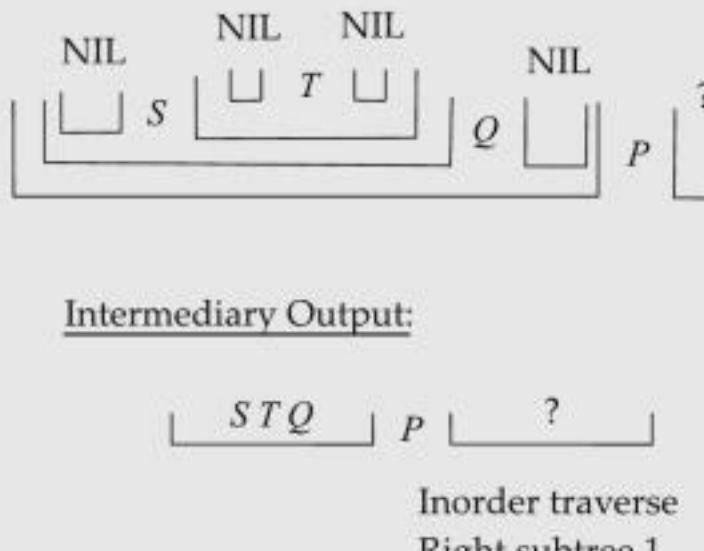
Execute the traversal of the binary tree as *traverse left subtree, process root node and traverse right subtree*. Repeat the same for each of the left and right subtrees encountered. Table 8.1. illustrates the traversal of the binary tree shown in Fig. 8.9, using this scheme. Each open box in the inorder traversal output (Column 2 of Table 8.1) represents the output of the call to the procedure INORDER_TRAVERSAL with the root of the appropriate subtree as its input. The final output of the inorder traversal is *S T Q P W U R V*.

Table 8.1 Inorder traversal of binary tree shown in Fig. 8.9

Binary Tree	Inorder Traversal Output	Remarks
<u>Step 1</u> Inorder Traverse Binary Tree Left subtree 1 Right subtree 1	[?] [P] [?] Inorder traverse Inorder traverse Left subtree 1 Right subtree 1	Inorder traversals of the Left and Right subtrees of the root node are to yield their output.

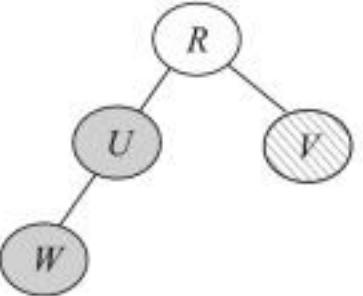
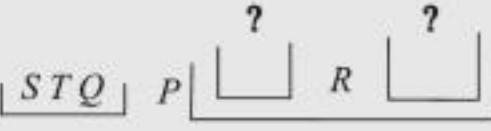
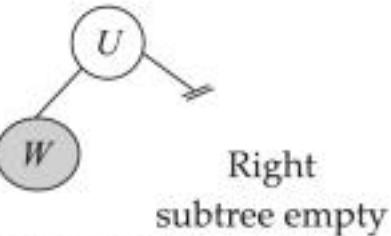
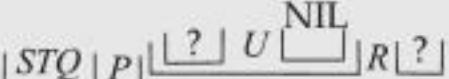
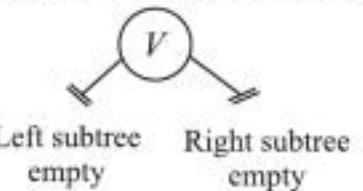
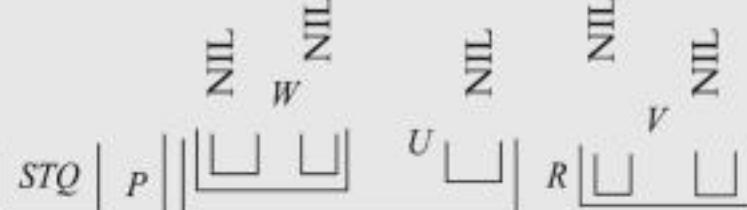
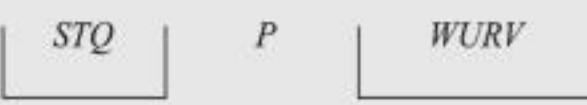
(Contd.)

(Contd.)

<p><u>Step 2</u></p> <p>Inorder Traverse Left subtree 1</p>  <p>Left subtree 2</p>		<p>Inorder traversal of the Left subtree 1 yields Inorder traversal of Left subtree 2, process root Q, and Inorder traverse Right subtree.</p> <p>However, since the Right subtree is empty, its traversal yields NIL output.</p>
<p><u>Step 3</u></p> <p>Inorder Traverse Left subtree 2</p>  <p>Left subtree empty Right subtree 2</p>		
<p><u>Step 4</u></p> <p>Inorder Traverse Right subtree 2</p>  <p>Left subtree empty Right subtree empty</p>	 <p><u>Intermediary Output:</u></p> <p>$[STQ]$ P $[?]$</p> <p>Inorder traverse Right subtree 1</p>	<p>Inorder traversal of Left subtree 1 is done.</p> <p>Gathering the traversal's output for Left subtree 1 yields STQ.</p> <p>The Inorder Traversal of Right subtree1 needs to be performed.</p>

(Contd.)

(Contd.)

<p><u>Step 5</u></p> <p>Inorder Traverse Right subtree 1</p>  <p>Left subtree 3 Right subtree empty</p>	<p>Inorder traverse Left subtree 3</p>  <p>Inorder traverse Right subtree 3</p>	<p>Inorder Traversal of the Left subtree 3 and Right subtree 3 are to yield their output.</p>
<p><u>Step 6</u></p> <p>Inorder Traverse Left subtree 3</p>  <p>Left subtree 4 Right subtree empty</p>	 <p>Inorder traverse Left subtree 4</p> <p>Inorder traverse Right subtree 3</p>	
<p><u>Step 7</u></p> <p>Inorder Traverse Left subtree 4</p>  <p>Left subtree empty Right subtree empty</p>	 <p>Inorder traverse Right subtree 3</p>	
<p><u>Step 8</u></p> <p>Inorder Traverse Right subtree 3</p>  <p>Left subtree empty Right subtree empty</p>	 <p>Inorder traversal of Right subtree 1 is done. Gathering the traversal's output yields WURV.</p>	<p><u>Final Output:</u></p>  <p>Final output: STQPWURV</p>

Postorder Traversal

The traversal proceeds by keeping to the left until it is no further possible, turns right to begin again or if there is no node to the right, processes the node and retraces its direction by one node to continue its traversal.

Algorithm 8.2 illustrates a recursive procedure to perform post order traversal of a binary tree. The recursive procedure reflects the maxim LRP invoked repetitively. Example 8.3 illustrates the postorder traversal of the binary tree shown in Fig. 8.9. The traversal output is *TSQWUVRP*.

Algorithm 8.2: Recursive procedure to perform Postorder traversal of a binary tree

```
Procedure POSTORDER_TRAVERSAL (NODE)
    /* NODE refers to the Root node of the binary tree
       in its first call to the procedure. Root node is the
       starting point of the traversal */
If NODE ≠ NIL then
    { call POSTORDER_TRAVERSAL (LCHILD(NODE));
      /* Postorder traverse the left subtree (L) */
      call POSTORDER_TRAVERSAL (RCHILD(NODE));
      /* Postorder traverse the right subtree (R) */
      Print (DATA (NODE)) ;
      /* Process node (P) */
    }
end POSTORDER_TRAVERSAL.
```

Example 8.3 As pointed out in Example 8.2, an easy method would be to run one's fingers on the binary tree with the maxim: *move left until there are no more nodes and turn right to continue traversal. If there is no right node, process node, retract by one node and continue traversal.*

An alternative method would be to trace the recursive steps of the algorithm using the scheme: *Traverse left subtree, Traverse right subtree and Process root node*. Table 8.2 illustrates the traversal of the binary tree shown in Fig. 8.9 using this scheme.

Preorder Traversal

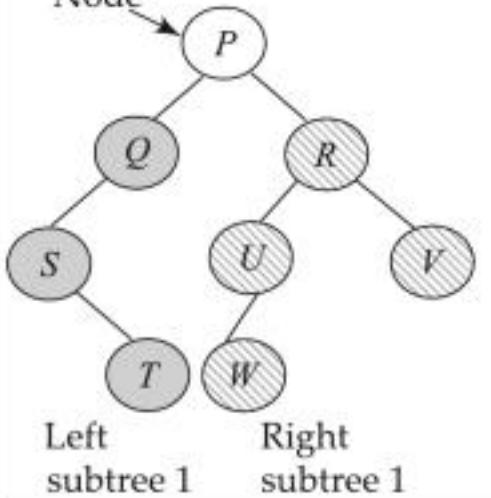
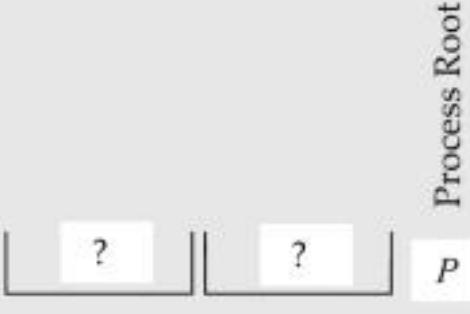
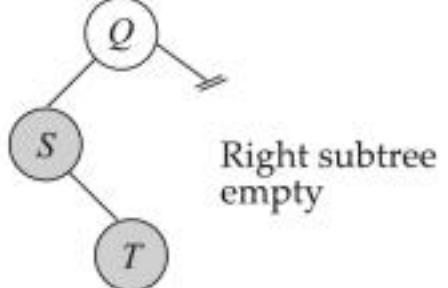
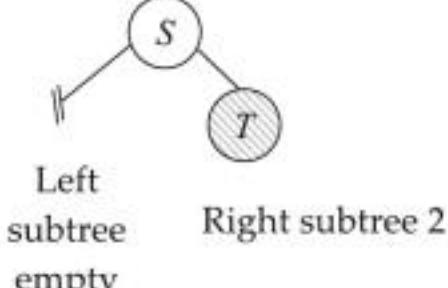
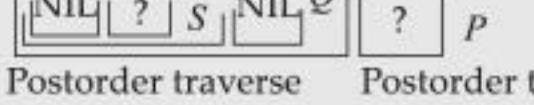
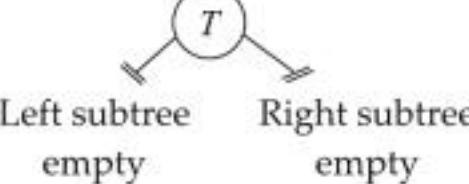
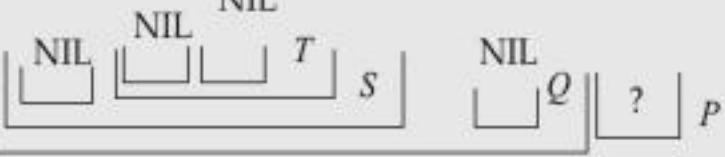
The traversal processes every node as it moves left until it can move no further. Now it turns right to begin again or if there is no node in the right, retracts until it can move right to continue its traversal.

The recursive procedure for the preorder traversal is illustrated in Algorithm 8.3. The recursive procedure reflects the maxim PLR invoked repetitively. Example 8.4 illustrates the preorder traversal of the binary tree shown in Fig. 8.9. The traversal output is *PQSTRUWV*.

Example 8.4 An easy method as discussed before would be to trace the traversal on the binary tree using the maxim: *Process nodes while moving left until no more nodes, turn right, and otherwise retract to continue the traversal.*

An alternative method is to trace the recursive steps of the algorithm using the following scheme:

Table 8.2 Postorder traversal of binary tree shown in Fig. 8.9

Binary Tree	Postorder Traversal Output	Remarks
<u>Step 1</u> Postorder traverse Binary Tree  Left subtree 1 Right subtree 1	 Postorder traverse Left subtree 1 Postorder traverse Right subtree 1	Postorder traversal of the Left and Right subtrees of the root is yet to yield their output.
<u>Step 2</u> Postorder Traverse Left subtree 1  Left subtree 2	Postorder traverse Left subtree 2  Postorder traverse Right subtree 1	
<u>Step 3</u> Postorder Traverse Left subtree 2  Left subtree empty Right subtree 2	 Postorder traverse Right subtree 2 Postorder traverse Right subtree 1	
<u>Step 4</u> Postorder traverse Right subtree 2  Left subtree empty Right subtree empty	 <u>Intermediary Output:</u> <u>TSQ</u> ? P	Postorder traversal of Left subtree 1 is done. Gathering the traversal's output for Left subtree 1 yields TSQ

(Contd.)

(Contd.)

<p><u>Step 5</u></p> <p>Postorder traverse Right subtree 1</p> <p>Left subtree Right subtree 3 3</p>	<p>Postorder traverse Right subtree 3</p> <p>$TSQ ? ? R P$</p> <p>Postorder traverse Left subtree 3</p>	
<p><u>Step 6</u></p> <p>Postorder Traverse Left subtree 3</p> <p>Left subtree 4 Right subtree empty</p>	<p>$TSQ ? \text{NIL} U ? R P$</p> <p>Postorder Traverse Left subtree 4 Postorder Traverse Right subtree 3</p>	
<p><u>Step 7</u></p> <p>Postorder traverse left subtree 4</p> <p>Left subtree empty Right subtree empty</p>	<p>$\text{NIL} \text{NIL} W$</p> <p>$TSQ \text{NIL} \text{NIL} U ? R P$</p> <p>Postorder traverse Right subtree 3</p>	
<p><u>Step 8</u></p> <p>Postorder Traverse Right subtree 3</p> <p>Left subtree empty Right subtree empty</p>	<p>$\text{NIL} \text{NIL} W \quad \text{NIL} \text{NIL} V$</p> <p>$TSQ \text{NIL} \text{NIL} U R P$</p> <hr/> <p><u>Final Output:</u></p> <p>$TSQ \quad WUVR \quad P$</p>	<p>Postorder traversal of Right subtree 1 is done. Gathering the output yields $WUVR$</p> <p>Final output: $TSQWUVRP$</p>

Algorithm 8.3: Recursive procedure to perform Preorder traversal of a binary tree

```

Procedure PREORDER_TRAVERSAL (NODE)
    /* NODE refers to the Root node of the binary tree
       in its first call to the procedure. Root node is the
       starting point of the traversal */

If NODE ≠ NIL then
    { Print (DATA (NODE)) ;
      /* Process node (P) */
      call PREORDER_TRAVERSAL (LCHILD(NODE));
      /* Preorder traverse the left subtree (L) */
      call PREORDER_TRAVERSAL (RCHILD(NODE));
      /* Preorder traverse the right subtree (R) */
    }
end PREORDER_TRAVERSAL.

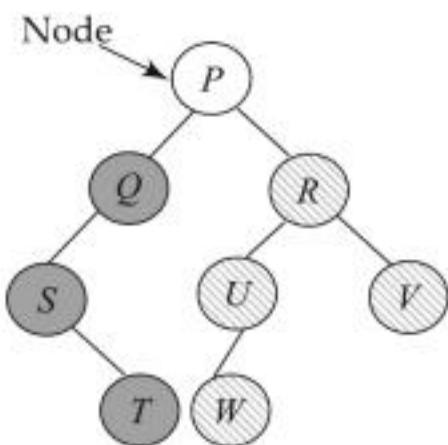
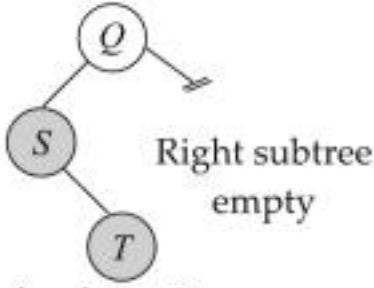
```



Execute the traversal of the binary tree as, *process root node*, *traverse left subtree* and *traverse right subtree*, repeating the same for each of the left and right subtrees encountered.

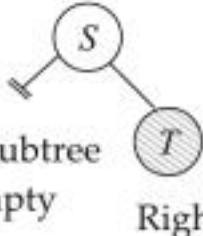
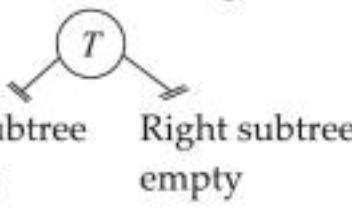
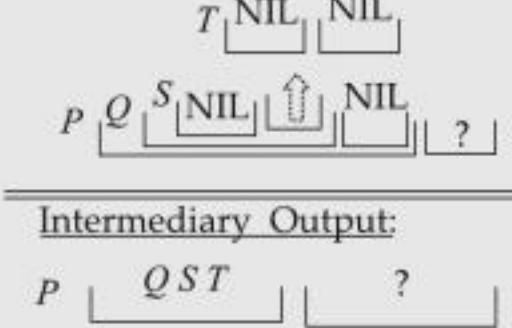
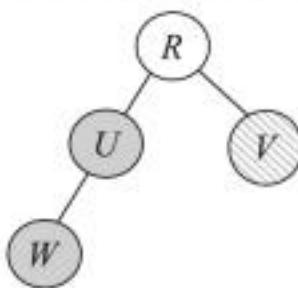
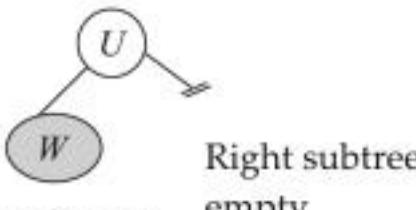
Table 8.3 illustrates the preorder traversal of the binary tree shown in Fig. 8.9 using this scheme. Some significant observations pertaining to the traversals of a binary tree are the following

Table 8.3 Preorder traversal of binary tree shown in Fig. 8.9

Binary Tree	Preorder Traversal Output	Remarks
<u>Step 1</u> Preorder traversal of Binary Tree  Left subtree 1 Right subtree 1	Process Root  Preorder traverse Left subtree 1 Preorder traverse Right subtree 1	Preorder traversals of the Left subtree 1 and Right subtree 1 to yield their output.
<u>Step 2</u> Preorder traverse Left subtree 1  Left subtree 2	Preorder traverse Left subtree 2  Preorder traverse Right subtree 1	

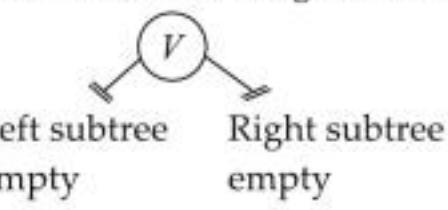
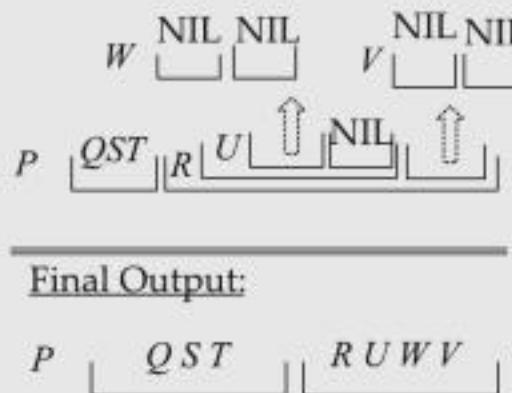
(Contd.)

(Contd.)

<p><u>Step 3</u></p> <p>Preorder traverse Left subtree 2</p>  <p>Left subtree empty Right subtree 2</p>	<p>Preorder traverse Right subtree 2</p> <p>$P Q S \text{NIL} ? \text{NIL} ?$</p> <p>Preorder traverse Right subtree 1</p>	
<p><u>Step 4</u></p> <p>Preorder traverse Right subtree 2</p>  <p>Left subtree empty Right subtree empty</p>	 <p style="text-align: center;"><u>Intermediary Output:</u></p> <p>$P Q S \text{NIL} \text{NIL} ?$</p>	<p>Preorder traversal of Left subtree 1 is done. Gathering the traversal's output yields QST</p>
<p><u>Step 5</u></p> <p>Preorder traverse Right subtree 1</p>  <p>Left subtree 3 Right subtree 3</p>	<p>Preorder traverse Left subtree 3</p> <p>$P Q S T R ? ? ?$</p> <p>Preorder traverse Right subtree 3</p>	
<p><u>Step 6</u></p> <p>Preorder traverse Left subtree 3</p>  <p>Left subtree 4 Right subtree empty</p>	<p>Preorder traverse Left subtree 4</p> <p>$P Q S T R U ? \text{NIL} ?$</p> <p>Preorder traverse Right subtree 3</p>	
<p><u>Step 7</u></p> <p>Preorder traverse left subtree 4</p>  <p>Left subtree empty Right subtree empty</p>	 <p>Preorder traverse Right subtree 3</p>	

(Contd.)

(Contd.)

<p>Step 8 Preorder traverse Right subtree 3</p>  <p>Left subtree empty Right subtree empty</p>	 <p style="text-align: center;">Final Output:</p> <p style="text-align: center;">P <u>Q S T</u> <u>R U W V</u></p>	<p>Preorder traversal of Right subtree 1 is done. Gathering the traversal's output yields RUWV.</p> <p>Final output: PQSTRUWV</p>
---	---	---

- (i) Given a preorder traversal of a binary tree, the root node is the first occurring item in the list.
- (ii) Given a postorder traversal of a binary tree, the root node is the last occurring item in the list.
- (iii) Inorder traversal does not directly reveal the root node of the binary tree.
- (iv) An inorder traversal coupled with any one of preorder or post order traversal helps trace back the structure of the binary tree (Refer Illustrative Problems 8.3, 8.4.).

Threaded Binary Trees

8.7

The linked representation of the binary tree discussed in Section 8.5 showed that for a binary tree with n nodes, $2n$ pointers are required of which $(n+1)$ are null pointers. A.J. Perlis and C.Thornton devised a prudent method to utilize these $(n+1)$ empty pointers, introducing what are called **threads**. Threads are also links or pointers but replace null pointers by pointing to some useful information in the binary tree. Thus, for a node NODE if RCHILD(NODE) is NIL then the null pointer is replaced by a thread which points to the node which would occur after NODE when the binary tree is traversed in inorder. Again if LCHILD (NODE) is NIL then the null pointer is replaced by a thread to the node which would immediately precede NODE when the binary tree is traversed in inorder.

Figure 8.10. illustrates a threaded binary tree. The threads are indicated using broken lines to distinguish them from the normal links indicated with solid lines. The inorder traversal of the binary tree is also shown in the figure.

Note that the left child of G and the right child of E have threads which are left dangling due to the absence of an inorder predecessor and successor respectively.

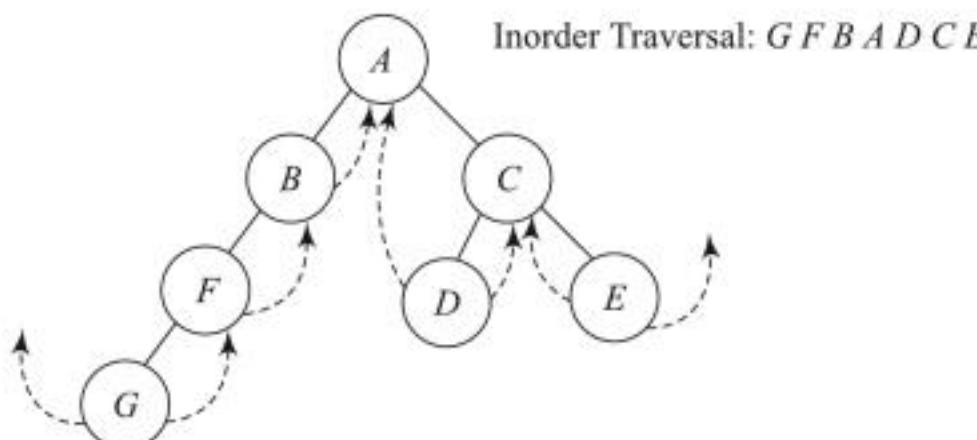


Fig. 8.10 A threaded binary tree

There are many ways to thread a binary tree T , corresponding to the specific traversal chosen. In this work, we have the threading correspond to an Inorder traversal. Also the threading can be of two representations viz., ***one-way threading*** and ***two-way threading***. One-way threading is where a thread appears only on the RCHILD field of a node, when it is null, pointing to the inorder successor of the node (Refer Illustrative Problem I8.10). On the other hand, in two-way threading, which had been introduced above, a thread appears in the LCHILD field also, if it is null, which points to the inorder predecessor of the node. However, the first and the last of the nodes in the inorder traversal will carry dangling threads.

Linked representation of a threaded binary tree

A linked representation of the threaded binary tree (two-way threading) has a node structure as shown in Fig. 8.11.

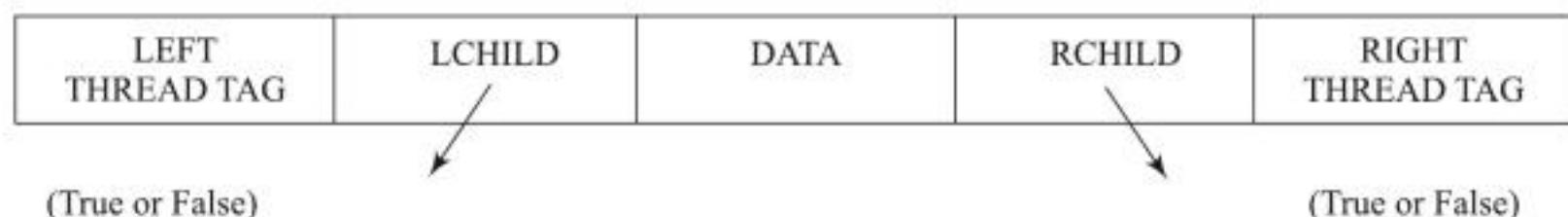


Fig. 8.11 Node Structure of a linked representation of a threaded binary tree

Since the LCHILD and RCHILD fields are utilized to represent both links and threads it becomes essential for the node structure to clearly distinguish between them to avoid confusion while processing the threaded binary tree. Hence it is necessary that the node structure includes two more fields which act as flags to indicate if the LCHILD and RCHILD fields represent a thread or a link.

If the LEFT THREAD TAG or RIGHT THREAD TAG is marked *true* then LCHILD and RCHILD fields represent threads otherwise they represent links. Also, to tuck in the dangling threads which are bound to arise, the linked representation of a threaded binary tree includes a *head node*. The dangling threads point to the head node. The head node by convention has its LCHILD pointing to the root node of the threaded binary tree and therefore has its LEFT THREAD TAG set to *false*. THE RIGHT THREAD TAG field is also set to *false* but the RCHILD link points to the head node itself. Figure 8.12(a) shows the linked representation of an empty threaded binary tree and Fig. 8.12(b) that of a non-empty threaded binary tree.

Growing threaded binary trees

Here we discuss the insertion of nodes contributing to the growth of threaded binary trees. The insertion of a node calls not only for the realignment of links but also of the threads involved.

Consider the case of inserting a node NEW to the right of a node NODE in the threaded binary tree. If the node NODE had no right subtree then the case is trivial. Attach NEW as right child of NODE and appropriately reset the threads (LCHILD and RCHILD) of NEW to point to its inorder predecessor and successor respectively. Figure 8.13(a) illustrates this insertion.

In the next case, if NODE already had a right subtree then attach NEW as the right child of the node NODE and link the previous right subtree of NODE to the right of node NEW. When this is done, the threads of the appropriate nodes are reset as shown in Fig. 8.13(b).

A similar procedure is followed to insert a node in the left subtree of a threaded binary tree.

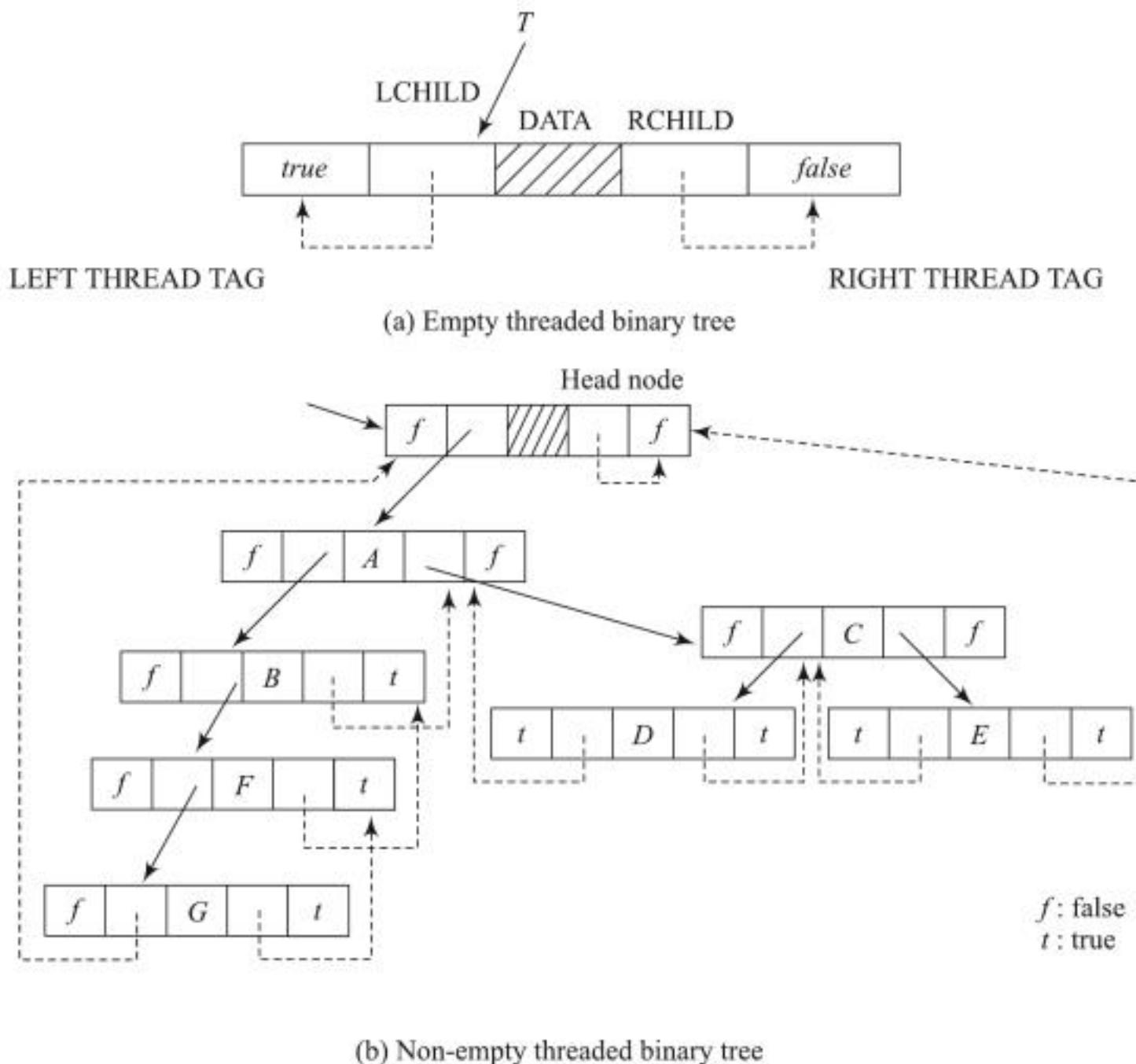


Fig. 8.12 Linked representation of threaded binary trees

Application

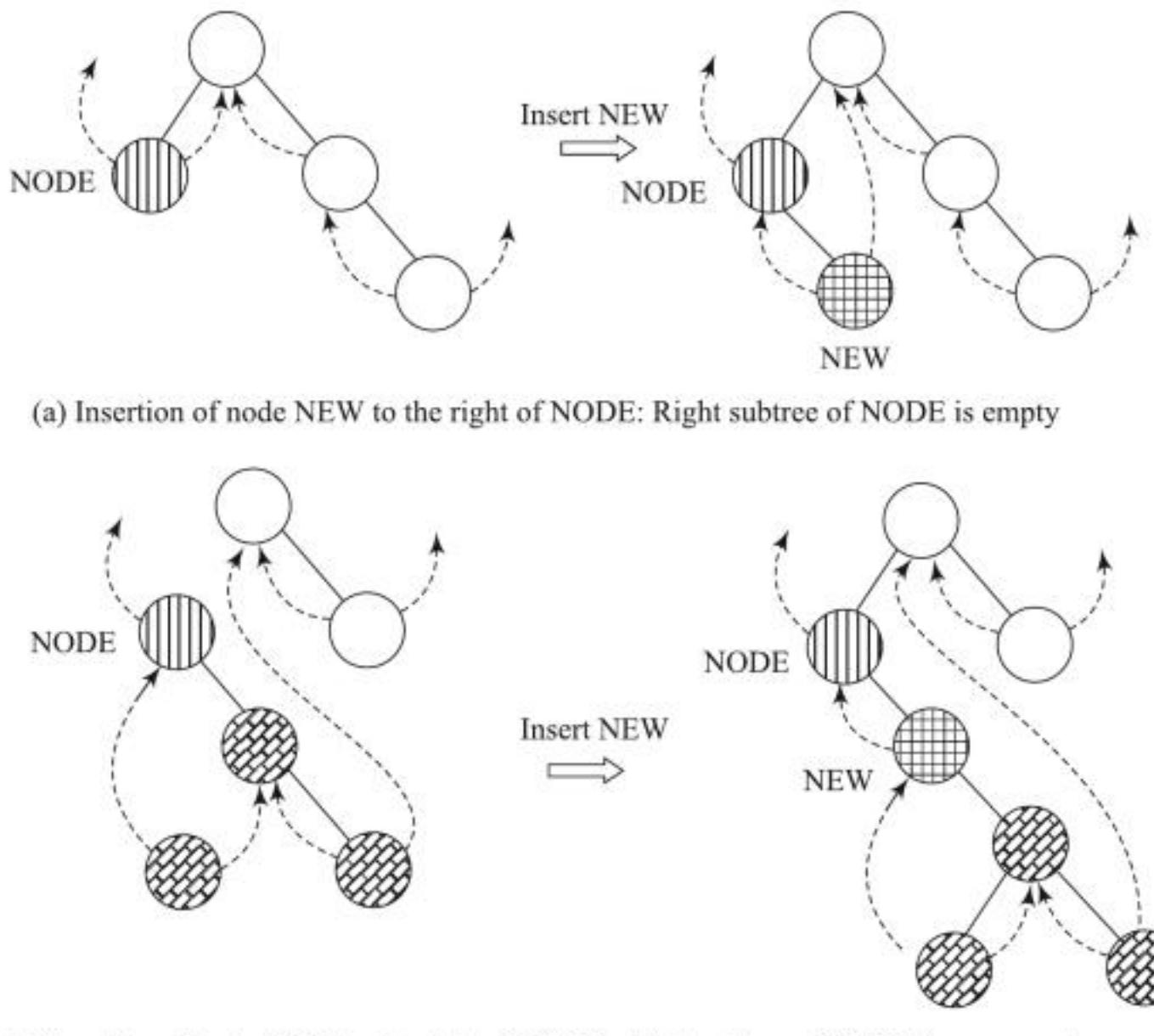
8.8

In this section we discuss an application of binary trees in expression trees which have a significant role to play in the principles of compiler design.

Expression trees

Expressions—arithmetic and logical—are an inherent component of programming languages. The following are examples of arithmetic and logical expressions.

- $((A + B) * C - D) \uparrow G$ (arithmetic expression)
- $(\neg A \wedge B) \vee (B \wedge E) \wedge \neg F$ (logical expression)
- $(T < W) \vee (A \leq B) \wedge (C \neq E)$ (logical expression)



(b) Insertion of node NEW to the right of NODE: Right subtree of NODE is non-empty

Fig. 8.13 Insertion of a node in the right subtree of a threaded binary tree

That expressions are represented in three forms viz., infix, postfix and prefix were detailed in Sec. 4.3. To quickly review, an infix expression which is the commonly used representation of an expression follows the scheme `<operand> <operator> <operand>`.

Examples are $A + B$, $A * B$.

Post fix expressions follow the scheme `< operand > < operand > < operator >`.

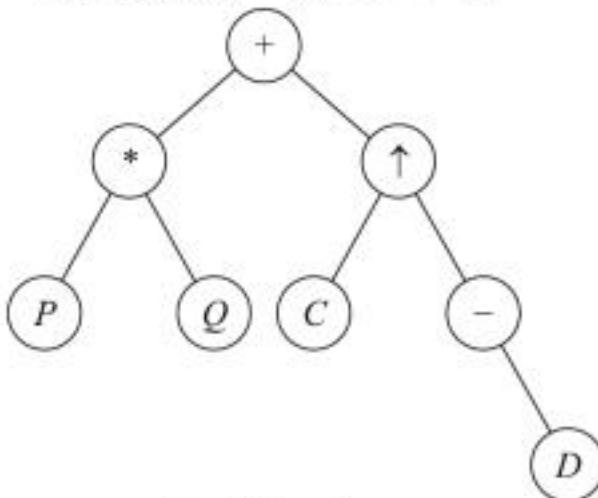
Examples are $AB+$, AB^* .

Prefix expressions follow the scheme `< operator > < operand > < operand >`.

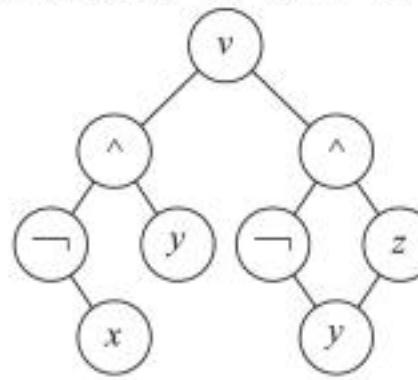
Examples are $+AB$, $*AB$.

Binary trees have found an application in the representation of expressions. An **expression tree** has the operands of the expression as its terminal or leaf nodes and the operators as its non terminal nodes. The arity of the operator is therefore restricted to be 1 or 2 (unary or binary) and this is what is commonly encountered in arithmetic and logical expressions. Figure 8.14 illustrates example expression trees.

The hierarchical precedence and associativity rules of the operators in terms of the expressions are reflected in the orientation of the subtrees or the sibling nodes. Table 8.4 illustrates some examples showing the orientation of the binary tree in accordance with the precedence and associativity rules of the operators in the expression terms.

Expression: $(P * Q) + C \uparrow - D$ 

(a) Arithmetic

Expression: $(\neg x \wedge y) \vee (\neg y \wedge z)$ 

(b) Logical

Fig. 8.14 Example of expression trees**Table 8.4 Orientation of the binary trees with regard to expressions**

Expression	Expression Tree	Remarks
$A + B$	<p>An expression tree for $A + B$. The root node is a circle containing a '+' operator. It has two children: circles containing 'A' and 'B' respectively.</p>	Observe the orientation of the sibling nodes. The left operand A and the right operand B become the left and right child nodes of the operator $+$ respectively.
$B + A$	<p>An expression tree for $B + A$. The root node is a circle containing a '+' operator. It has two children: circles containing 'B' and 'A' respectively.</p>	
$A + B + C$	<p>An expression tree for $A + B + C$. The root node is a circle containing a '+' operator. It has three children: circles containing '+', 'B', and 'C' respectively. The '+' node has two children: circles containing 'A' and 'B' respectively.</p>	The left associativity rule satisfied by $+$ is reflected in the orientation of the subtrees.
$A \uparrow B \uparrow C$	<p>An expression tree for $A \uparrow B \uparrow C$. The root node is a circle containing a '\uparrow' operator. It has two children: circles containing 'A' and 'B' respectively. The 'B' node has one child: a circle containing 'C'.</p>	The right associate rule satisfied by \uparrow is reflected in the orientation of the subtrees.
$A * B - C / D$	<p>An expression tree for $A * B - C / D$. The root node is a circle containing a '-' operator. It has two children: circles containing '*' and '/' respectively. The '*' node has two children: circles containing 'A' and 'B' respectively. The '/' node has two children: circles containing 'C' and 'D' respectively.</p>	The hierarchical precedence relation among the operators decides the orientation of the subtrees.

Traversals of an expression tree

Section 8.6. detailed the traversals of a binary tree. With an expression tree essentially being a binary tree, the traversal of an expression tree also yields significant results. Thus, the inorder traversal of an expression tree yields an infix expression, the postorder traversal, a postfix expression and preorder traversal, a prefix expression. The output of the algorithms `INORDER_TRAVERSAL()`, `PREORDER_TRAVERSAL()` and `POSTORDER_TRAVERSAL()` on any given expression tree can be verified against the hand computed infix, prefix and postfix expressions (discussed in Sec. 4.3).

Conversion of infix expression to postfix expression

We utilize this opportunity to introduce a significant concept of infix to postfix expression conversion which finds a useful place in the theory of compiler design.

Given an infix expression, for example $A+B*C$, the objective is to obtain its postfix equivalent $ABC*+$. In Chapter 4, Sec. 4.3, a hand coded method of conversion was illustrated. In this section, we introduce an algorithm to perform the same.

The algorithm makes use of a stack as its work space and is governed by two priority factors viz., *In Stack priority* (ISP) and *Incoming Priority* (ICP) of the operators participating in the conversion. Thus those operators already pushed into the stack during the process of conversion, command an ISP in relation to those which are just about to be pushed in to the stack (ICP). Table 8.5 illustrates the ISP and ICP of a common list of arithmetic operators.

Table 8.5 *ISP and ICP of a common list of arithmetic operators*

Operator	ISP	ICP
)	-	-
\uparrow	3	4
$*, /$	2	2
$+, -$	1	1
(0	4

The rule which operates during conversion of infix to post fix expression is : *pop operators out of the work stack so long as the ICP of the incoming operator is less than or equal to the ISP of the operators already available in the stack.*

The input infix expression is padded with a "\$" to signal end of input. The bottom of the work stack is also signaled with a "\$" symbol with $ISP(\$) = -1$. Algorithm 8.4 illustrates the pseudo-code procedure to convert an infix expression into postfix expression.

Algorithm 8.4: Procedure to convert infix expression to postfix expression

```

Procedure INFIX_POSTFIX_CONV(E)
    /* to convert an infix expression E padded with a "$"
       as its end-of-input symbol into its equivalent
       postfix expression */
    x := getnextchar (E);
    /* obtain the next character from E */

```

```

while x ≠ “$” do
    case x of
        : x is an operand: Print x;
        : x = ‘)’: while (top element of stack ≠ ‘(’) do
            print top element of stack and pop stack;
            end while;
            Pop ‘(’ from stack;
        : else : while ICP (x) ≤ ISP (top element of stack) do
            print top element of stack and pop stack;
            end while
            push x into stack;
    end case
    x: = getnextchar(E);
end while
while stack is non empty do
    print top element of stack and pop stack;
end while
end INFIX_POSTFIX_CONV.

```

Example 8.5 illustrates the conversion of an infix expression into its equivalent postfix expression using Algorithm INFIX_POSTFIX _CONV ().

Example 8.5 Consider an infix expression $A^*(B+C)-G$. Table 8.6 illustrates the conversion into its postfix equivalent.

Table 8.6 Conversion of $A^*(B+C)-G\$$ into its postfix form

Input character fetched by getnextchar ()	Work stack	Postfix expression	Remarks
<i>A</i>	\$	<i>A</i>	Print <i>A</i>
*	\$*	<i>A</i>	Since ISP(*) > ISP (\$) push * into stack.
(\$*(<i>A</i>	Since ICP (‘(’) > ISP (*) push (into stack.
<i>B</i>	\$*(<i>AB</i>	Print <i>B</i> .
+	\$*(+	<i>AB</i>	Since ICP (+) > ISP(‘(’) push + into stack.
<i>C</i>	\$*(+	<i>ABC</i>	Print <i>C</i>
)	\$*	<i>ABC+</i>	Pop elements from stack until ‘(’ is reached. Also pop ‘(’ from stack.
-	\$-	<i>ABC+*</i>	Since ICP (-) < ISP (*) pop * from the stack. Push ‘-’ into stack
<i>G</i>	\$-	<i>ABC+*G</i>	Print <i>G</i>
\$	\$	<i>ABC+*G-</i>	End of input (\$) reached. Empty contents of stack.

ADT for Binary Trees

Data objects:

A binary tree of nodes each holding one (or more) data field(s) DATA and two link fields, LCHILD and RCHILD. T points to the root node of the binary tree.

Operations:

- Check if binary tree T is empty
 CHECK_BINTREE_EMPTY (T) (Boolean function)
- Make a binary tree T empty by setting T to NIL
 MAKE_BINTREE_EMPTY (T)
- Move to the left subtree of a node X by moving down its LCHILD pointer
 MOVE_LEFT_SUBTREE(X)
- Move to the right subtree of a node X by moving down its RCHILD pointer
 MOVE_RIGHT_SUBTREE(X)
- Insert node containing element $ITEM$ as the root of the binary tree T ; Ensure that T does not point to any node before execution
 INSERT_ROOT ($T, ITEM$)
- Insert node containing $ITEM$ as the left child of node X ; Ensure that X does not have a left child node before execution
 INSERT_LEFT ($X, ITEM$)
- Insert node containing $ITEM$ as the right child of node X ; Ensure that X does not have a right child node before execution
 INSERT_RIGHT ($X, ITEM$)
- Delete root node of binary tree T ; Ensure that the root does not have child nodes
 DELETE_ROOT (T)
- Delete node pointed to by X from the binary tree and set X to point to the left child of the node; Ensure that the node pointed to by X does not have a right child
 DELETE_POINT_LEFTCHILD(X)
- Delete node pointed to by X from the binary tree and set X to point to the right child of the node; Ensure that the node pointed to by X does not have a left child
 DELETE_POINT_RIGHTCHILD(X)
- Store $ITEM$ into a node whose address is X
 STORE_DATA($X, ITEM$)
- Retrieve data of a node whose address is X and return it in $ITEM$
 RETRIEVE_DATA($X, ITEM$)
- Perform Inorder traversal of binary tree T
 INORDER_TRAVERSAL(T)
- Perform Preorder traversal of binary tree T
 PREORDER_TRAVERSAL(T)
- Perform Postorder traversal of binary tree T
 POSTORDER_TRAVERSAL(T)





Summary

- Trees and binary trees are non-linear data structures, which are inherently two dimensional in structure.
- While trees are non empty and may have nodes of any degree, a binary tree may be empty or hold nodes of degree, at most two.
- The terminologies of root node, height, level, parent, children, sibling, ancestors, leaf or terminal nodes and non-terminal nodes are applicable to both trees and binary trees.
- While trees are efficiently represented using linked representations, binary trees are represented using both array and linked representations.
- The traversals of a binary tree are inorder, postorder and preorder.
- A prudent use of null pointers in the linked representation of a binary tree yields a threaded binary tree.
- The application of binary trees has been demonstrated on expression trees and its related concepts.
- The ADT of the binary tree is presented.



Illustrative Problems

Problem 8.1 For the binary tree shown in Fig. I 8.1,

- Identify
 - Root
 - children of G
 - parent of D
 - siblings of Z
 - Level of C
 - Ancestors of Y
 - leaf nodes
 - height of the binary tree
- Obtain the inorder, postorder and preorder traversals of the binary tree.

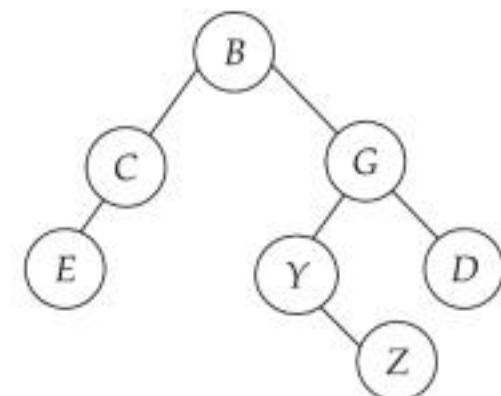
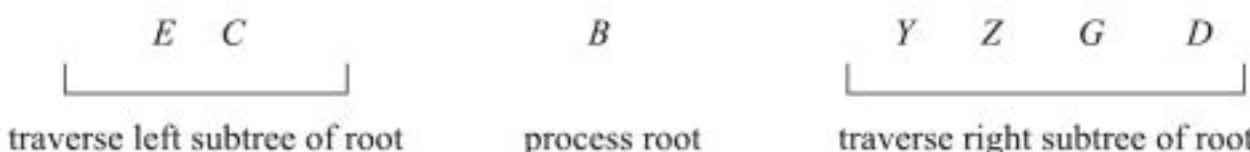


Fig. I 8.1

Solution:

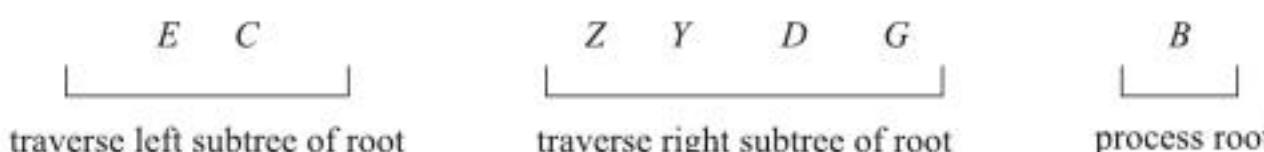
- (i) Root : B (ii) Children of G: Y,D (iii) Parent of D: G (iv) Siblings of Z: None (v) Level of C: 2 (vi) Ancestors of Y : G, B (vii) Leaf nodes : E, Z, D (viii) Height of the binary tree :4.
- Inorder traversal : ECBYZGD

The output of the traversal which follows the scheme of Algorithm 8.1 can be dissected as



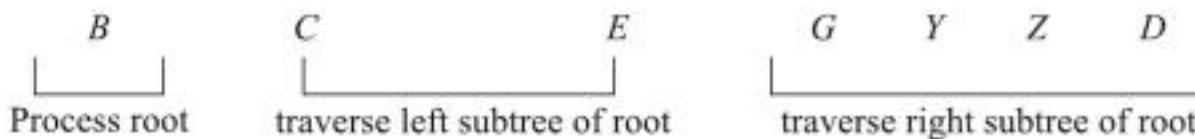
Postorder traversal : ECZYDGB.

The output of the traversal following the scheme of Algorithm 8.2 can be dissected as.



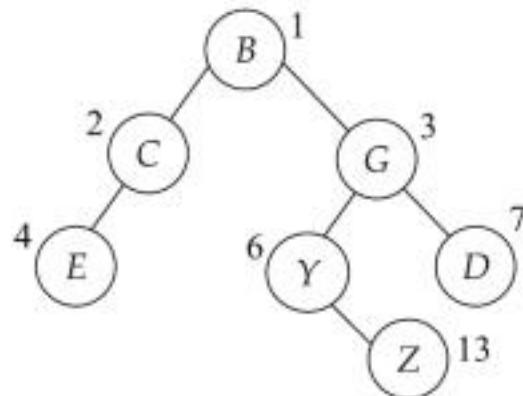
Preorder traversal : BCEGYZD

The output of the traversal following the scheme of Algorithm 8.3 can be dissected as



Problem 8.2 Obtain an array representation and a linked representation of the binary tree shown in Fig. I 8.1.

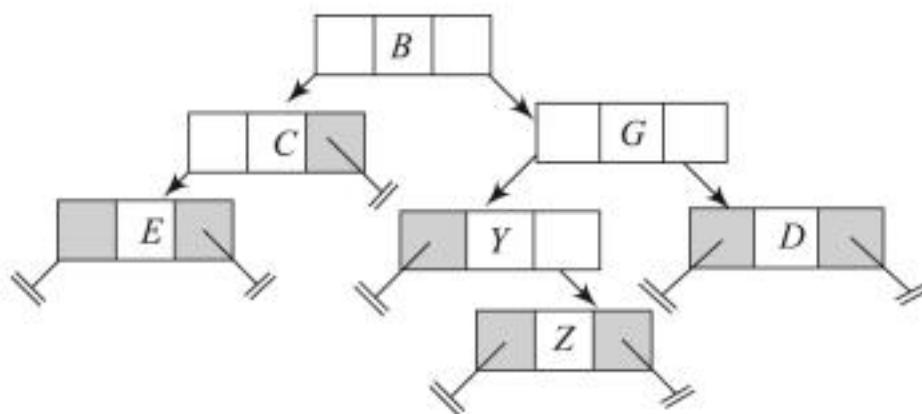
Solution: To obtain the array representation we first number the nodes of the binary tree akin to that of a complete binary tree, as shown below:



The array representation is given as:

B	C	G	E		Y	D						Z
[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]

The linked representation is given as:



Problem 8.3 A binary tree T has 9 nodes. The inorder and preorder traversals of T yield the following:

Inorder traversal (I) : $E \quad A \quad C \quad K \quad F \quad H \quad D \quad B \quad G$

Preorder traversal (P) : $F \quad A \quad E \quad K \quad C \quad D \quad H \quad G \quad B$

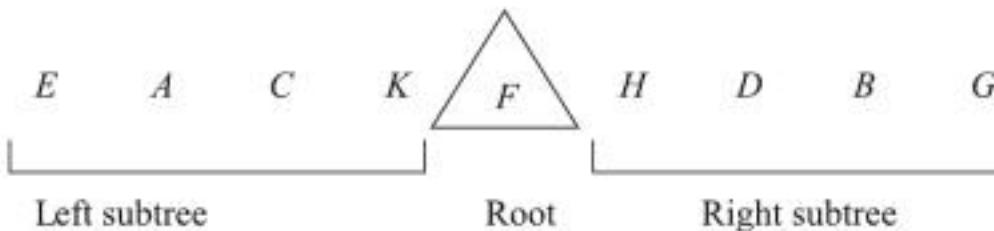
Draw the binary tree T .

Solution: The key to the solution of this problem is the observation that the first occurring node in a preorder traversal is the root of the binary tree and that, once the root is known, the nodes forming the left and the right subtrees can be extracted from the Inorder traversal list. Application

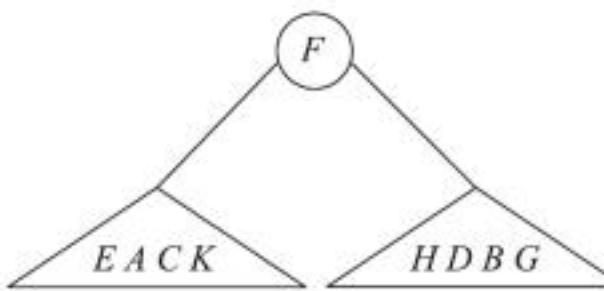
of this key to each of the left and right subtree by obtaining their respective roots from the preorder traversal and moving on to inorder traversal to obtain the nodes forming the sub-subtrees can eventually lead to the tracing of the binary tree.

From P: Root of the binary tree is F .

From I : the nodes forming the left and right subtrees of F are

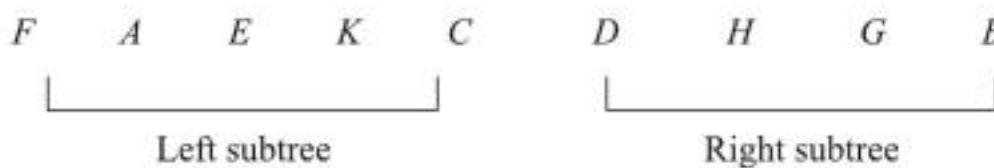


The binary tree can be roughly traced as shown below:

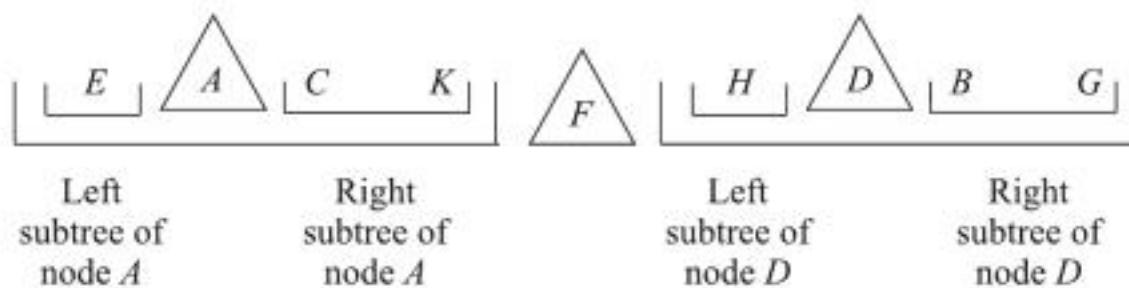


In the next step we proceed to obtain the structure of the left and right subtrees.

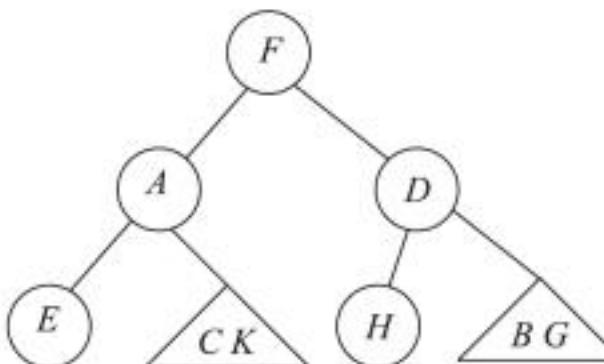
From P : Root of the left subtree is A and root of the right subtree is D



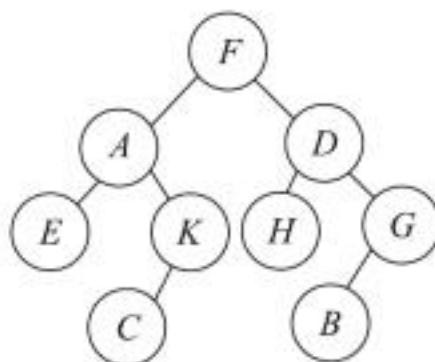
From I : the nodes forming the left and right sub-subtrees are



Tracing the binary tree yields,



Proceeding in a similar fashion, we obtain the roots of the subtrees $\{C, K\}$ and $\{B, G\}$ to be K and G respectively. The final trace yields the binary tree:



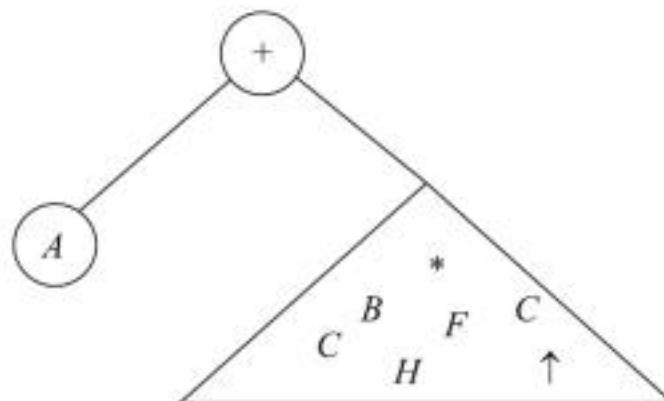
Example 8.4 Make use of the infix and postfix expressions given below to trace the corresponding expression tree

Infix : $A + B * C / F \uparrow H$

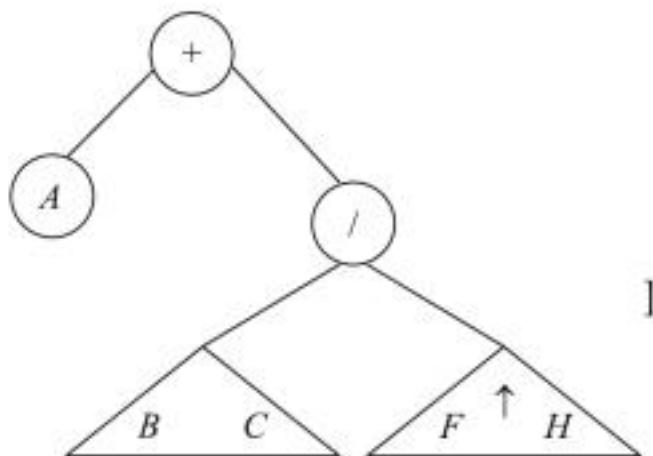
Postfix : $A B C * F H \uparrow / +$

Solution: The key to the problem is similar to the one discussed in Illustrative Problem 8.3 but for the difference that the root node to be picked from the postfix expression is the last occurring node.

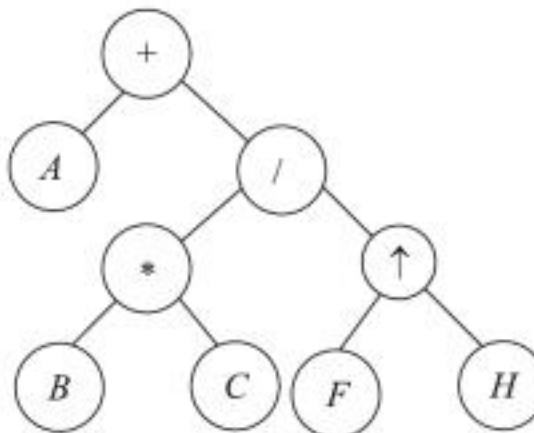
Thus the expression tree traced in the first step is:



In the next step the expression tree traced would be



Progressing in this way, the final expression tree is obtained as shown in the adjacent figure.



Note: Though the expression tree could be easily traced from infix expression alone, the objective of the problem is to emphasize the fact that a binary tree can be traced from its inorder and postorder traversals as well.

Example 8.5 What does the following pseudocode procedure do to the binary tree given in Fig. I 8.5, when invoked as `WHAT_DO_I_DO (THIS)`?

```

Procedure WHAT_DO_I_DO (HERE)
  if HERE ≠ NIL then
    { call WHAT_DO_I_DO(LCHILD (HERE));
      if ( LCHILD (HERE) = NIL) and (RCHILD (HERE) = NIL)
      then Print DATA (HERE) ;
      call WHAT_DO_I_DO (RCHILD(HERE));
    }
  end WHAT_DO_I_DO.
  
```

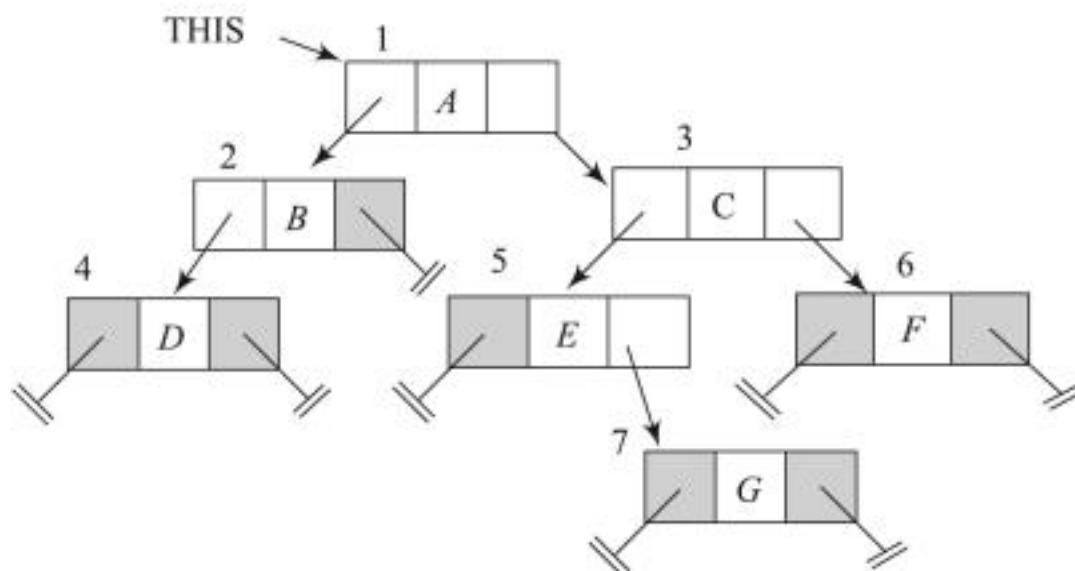
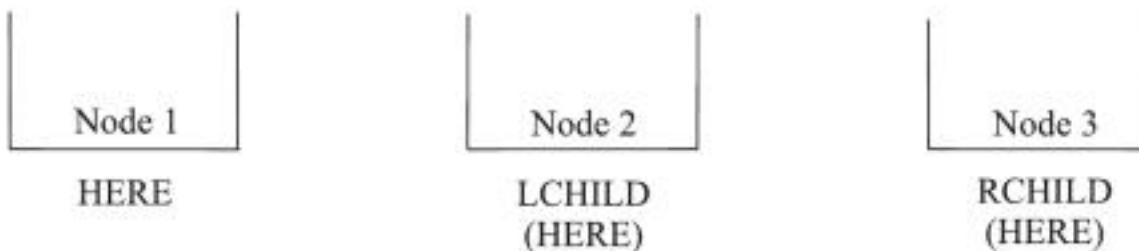


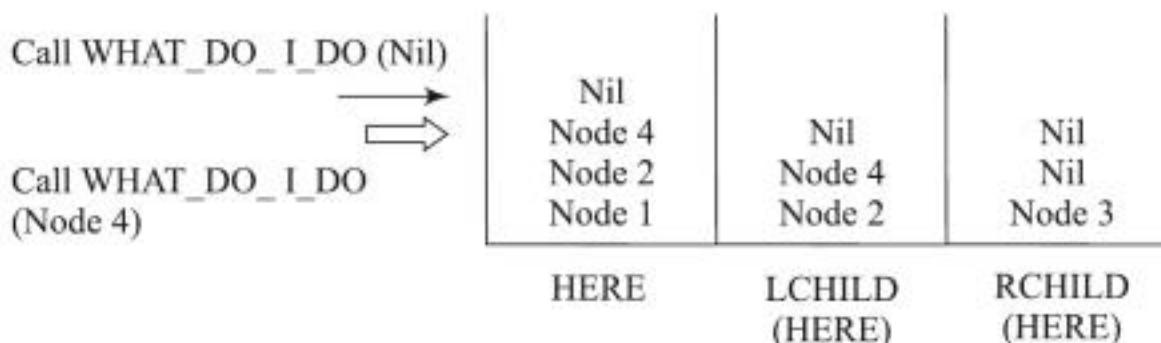
Fig. I 8.5

Solution: We trace the recursive procedure using a stack and for convenience of representing the nodes in the stack, have numbered the nodes from 1 to 7. For every call of `WHAT_DO_I_DO()` we keep track of HERE, LCHILD (HERE) and RCHILD (HERE).

The first call of `WHAT_DO_I_DO (THIS)` results in the following snap shot of the stack :



In the subsequent calls the snapshot of the stack is given by



when `HERE = NIL` that call of `WHAT_DO_I_DO (HERE)` (marked \rightarrow) terminates and the control returns to the previous call viz., `WHAT_DO_I_DO (Node 4)` (marked \Rightarrow). Here `LCHILD (HERE) = RCHILD (HERE) = NIL`. Hence `DATA(Node 4)` viz., D is printed. Now the control moves further to invoke the call `WHAT_DO_I_DO (RCHILD(Node 4))` that is `WHAT_DO_I_DO (NIL)` which again terminates. Now the control returns to the call `WHAT_DO_I_DO (Node 2)` and so on. It is easy to see that `WHAT_DO_I_DO (THIS)` prints the data fields of all leaf nodes of the binary tree. Hence the output is D, G and F.

Example 8.6 Show that the maximum number of nodes in a binary tree of height h is $2^h - 1$, $h \geq 1$.

Solution: It is known that the maximum number of nodes in level i of a binary tree is 2^{i-1} . Given the height of the binary tree to be h which is the maximum level, the maximum number of nodes is given by

$$\sum_{i=1}^h 2^{i-1} = 1 + 2 + 2^2 + \dots + 2^{h-1} = 2^h - 1$$

Example 8.7 Show that for a non-empty binary tree T if n_o is the number of leaf nodes, n_2 the number of nodes of degree 2 then $n_o = n_2 + 1$.

Solution: Let n be the number of nodes in a non empty binary tree and let n_1 be the number of nodes of degree 1.

Now,

$$n = n_o + n_1 + n_2 \quad \dots(i)$$

Again if b is the number of links or branches in the binary tree, all nodes except the root node hang from a branch yielding the relation

$$b = n - 1 \quad \dots(ii)$$

Also, each branch emanates from a node whose degree is either 1 or 2. Hence,

$$b = n_1 + 2.n_2 \quad \dots \text{(iii)}$$

Subtracting (iii) from (ii) yields.

$$n = n_1 + 2n_2 + 1 \quad \dots \text{(iv)}$$

From (iv) and (i) we obtain

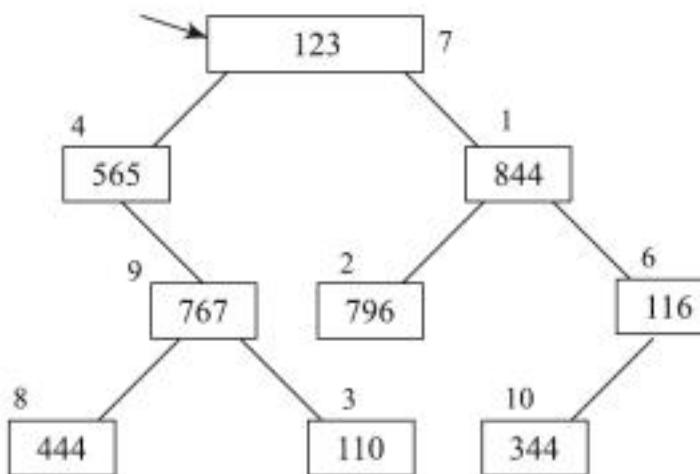
$$n_0 = n_2 + 1$$

Example 8.8 A binary tree is stored in the memory of a computer as shown below. Trace the structure of the binary tree.

	LCHILD	DATA	RCHILD	
1	2	844	6	
2	0	796	0	
3	0	110	0	
4	0	565	9	
5	12	444	0	
6	10	116	0	
7	4	123	1	
8	0	444	0	
9	8	767	3	
10	0	344	0	

Root : 7

Solution: Given the root node's address to be 7 we begin tracing the binary tree from the root onwards. The binary tree is given by:



Example 8.9 Outline a linked representation for the tree and threaded binary tree representation for the binary tree shown in Fig. I 8.9(a) and (b), respectively.

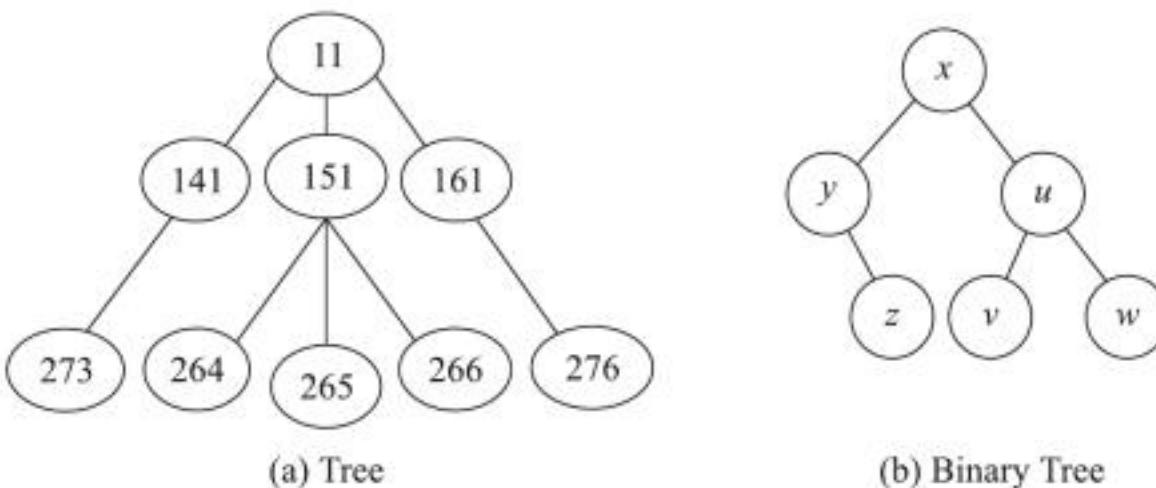
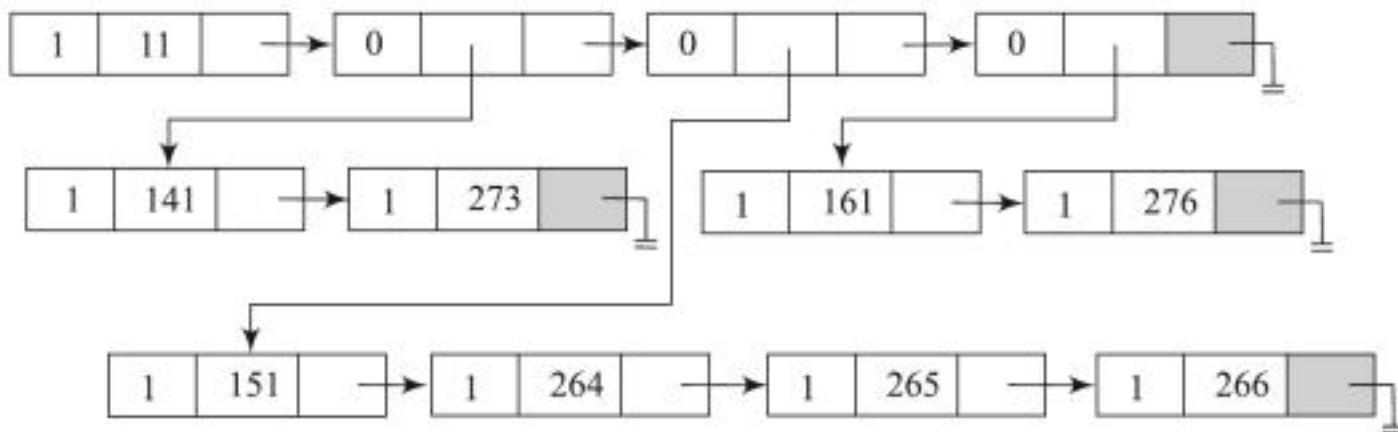
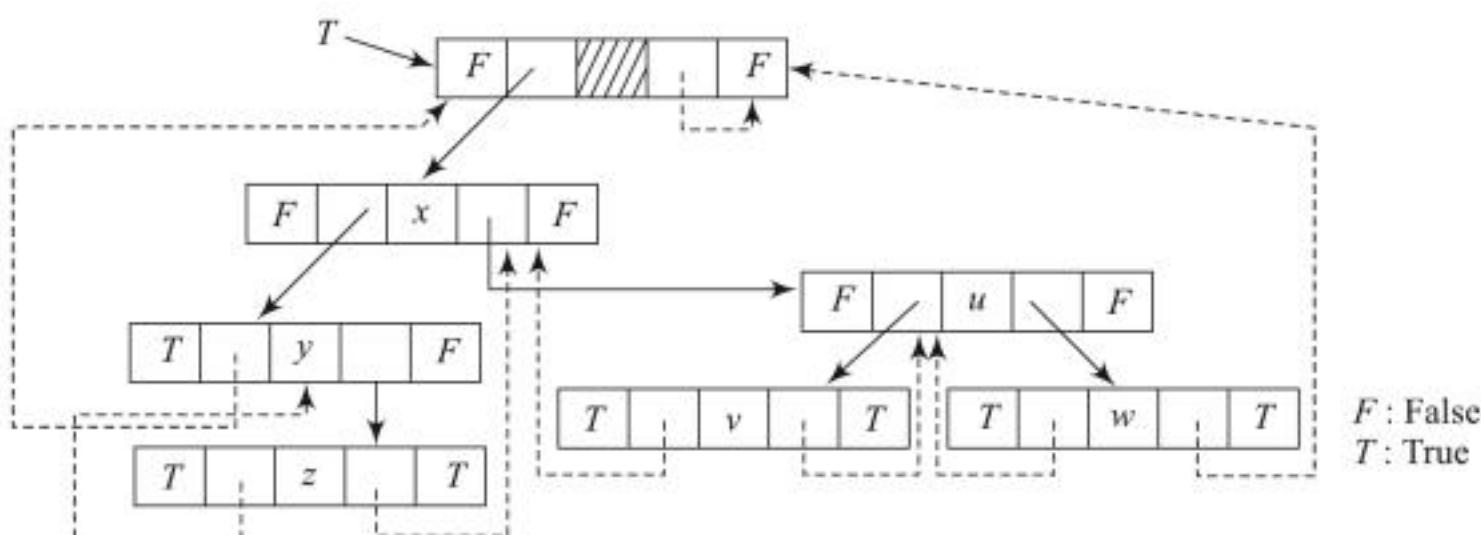


Fig. I 8.9

Solution: Following the node structure of TAG, DATA/DOWNLINK, LINK illustrated in Sec. 8.3 the linked representation of the tree is given by



The threaded binary representation of Fig. I 8.9(b) is illustrated below and is obtained by following the node structure detailed in Sec. 8.7.



The inorder traversal sequence to be tracked by the threads is : $y \ z \ x \ v \ u \ w$. The threads are linked to the appropriate inorder successors and predecessors.

Example 8.10 For the binary tree T given in Fig. I 8.10 obtain (i) a one-way inorder threading of T and (ii) one-way preorder threading of T .

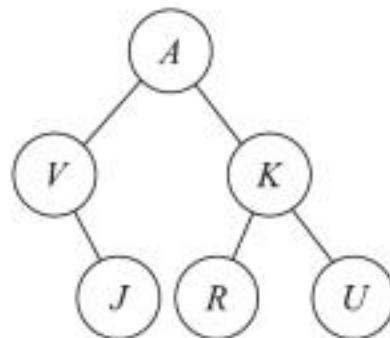
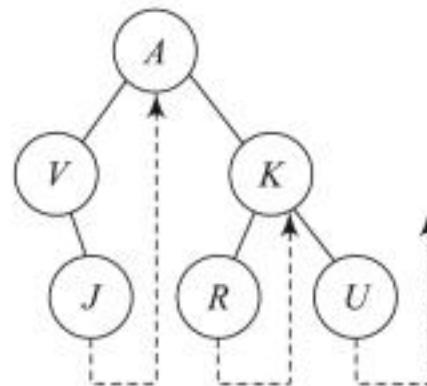


Fig. I 8.10

Solution:

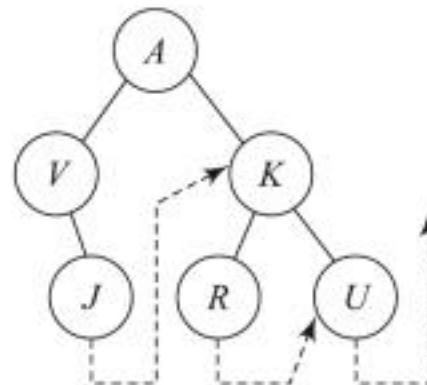
- (i) The inorder traversal of the binary tree T yields: $V \ J \ A \ R \ K \ U$

A one-way threading of T is obtained by replacing the RCHILD links of the nodes, which are null, by threads pointing to the inorder successor of the node. Thus, the RCHILD link of J points to A , that of R points to K and that of U is either kept dangling (or if there is a head node points to the same). The threaded binary tree is shown below:



- (ii) The preorder traversal of binary T yields: $A \ V \ J \ K \ R \ U$.

For the one-way preorder threading, the RCHILD links of the nodes, which are null, are set to point to the preorder successors of the node. Thus, the RCHILD link of J points to K , that of R points to U and the same of U is a dangling thread or may be connected to the head node if available. The threaded tree for the same is shown below:





Review Questions

1. Which among the following is not a property of a tree?

 - There is a specially designated node called the root
 - The rest of the nodes could be partitioned into t disjoint sets ($t \geq 0$) each set representing a tree T_i , $i = 1, 2, \dots, t$ known as subtree of the tree.
 - Any node should be reachable from anywhere in the tree
 - At most one cycle could be present in the tree

(a) (i)	(b) (ii)	(c) (iii)	(d) (iv)
---------	----------	-----------	----------

2. The maximum number of nodes in a binary tree of depth k is

(a) 2^{k-1}	(b) $2^{(k+1)} - 1$	(c) 2^{k-1}	(d) $2^{(k+1)-1}$
---------------	---------------------	---------------	-------------------

3. For a binary tree of $2.k$ nodes, $k > 1$, the number of pointers and the number of null pointers that the tree would use for its representation is respectively given by

(a) k and $k+1$	(b) $2.k$ and $2.k + 1$	(c) $4.k$ and $4.k + 1$	(d) $4.k$ and $2.k + 1$
-------------------	-------------------------	-------------------------	-------------------------

4. An inorder and postorder traversal of a binary tree was 'claimed' to yield the following sequence:

Inorder traversal : HAT	GLOVE	SOCKS	SCARF	GLASSES
Post order traversal : GLOVE	SCARF	HAT	GLASSES	SOCKS

What are your observations?

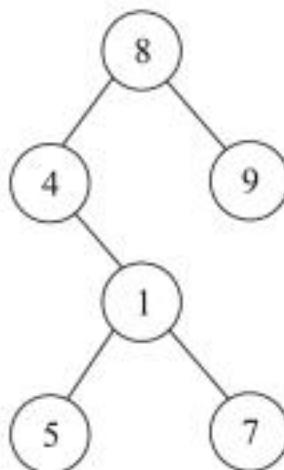
 - HAT is the root of the binary tree
 - SOCKS is the root of the binary tree
 - the binary tree is a skewed binary tree
 - the traversals are incorrect

(a) (i)	(b) (ii)	(c) (iii)	(d) (iv)
---------	----------	-----------	----------

5. Which of the following observations with regard to binary tree traversals is incorrect?

 - Given a preorder traversal of a binary tree, the root node is the first occurring item in the list.
 - Given a postorder traversal of a binary tree, the root node is the last occurring item in the list.
 - Inorder traversal does not directly reveal the root node of the binary tree.
 - To trace back the structure of the binary tree, inorder, postorder and preorder traversal sequences are needed.

6. Sketch (i) an array representation and (ii) a linked list representation for the following binary tree:



7. Sketch a linked representation for a threaded binary tree equivalent of the binary tree shown in Review Questions 6 (Chapter 8).
8. Obtain inorder and post order traversals for the binary tree shown in Review Questions 6 (Chapter 8).
9. Draw an expression tree for the following logical expression:
 $p \text{ and } (q \text{ or not } k) \text{ and } (s \text{ or } b \text{ or } h)$
10. Undertake post order traversal of the expression tree obtained in Review Questions 9 (Chapter 8) and compare it with the hand computed postfix form of the logical expression.
11. Given the following inorder and preorder traversals, trace the binary tree.
 Inorder traversal : $B F G H P R S T W Y Z$
 Preorder traversal : $P F B H G S R Y T W Z$
12. Making use of Algorithm 8.4, convert the following infix expression to its equivalent postfix form and evaluate the postfix expression for the specified values:

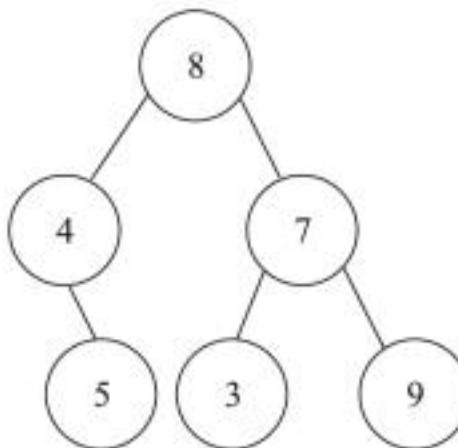
$$(x + y + z) \uparrow (a + b) - g * n * m + r$$

$$x = 1, y = 2, z = -1, a = 1, b = 2, g = 5, n = 2, m = 1, r = 7$$



Programming Assignments

1. Write a program to input a binary tree implemented as a linked representation. Execute Algorithms 8.1–8.3 to perform inorder, postorder and preorder traversals of the binary tree.
2. Implement Algorithm 8.4 to convert an infix expression into its postfix form.
3. Write a recursive procedure to count the number of nodes in a binary tree.
4. Implement a threaded binary tree. Write procedures to insert a node NEW to the left of node NODE when
 - (i) the left subtree of NODE is empty, and
 - (ii) the left subtree of NODE is non-empty.
5. Write non-recursive procedures to perform the inorder, postorder and preorder traversals of a binary tree.
6. **Level order traversal:** It is a kind of binary tree traversal where elements in the binary tree are traversed by levels, top to bottom and within levels, left to right. Write a procedure to execute the level order traversal of a binary tree.(Hint: Use a Queue data structure)
 Example: Level order traversal of the following binary tree is: 8 4 7 5 3 9



7. Implement the ADT of a binary tree in a language of your choice. Include operations to (i) obtain the height of a binary tree and (ii) the list of leaf nodes.



GRAPHS

9

In Chapter 8 we introduced trees and graphs as examples of non linear data structures. To recall, non-linear data structures unlike linear data structures which are uni dimensional in structure (for example arrays), are inherently two dimensional in structure.

Though in the field of computer science, trees have been recognized as efficient non linear data structures with their own set of terminologies and concepts to suit the needs of the digital computer, graph theory which has emerged as an independent field, encompasses studies on trees as well. In other words, in the field of graph theory, a tree is a special kind of graph holding a definition which in principle agrees with that of a tree data structure, but is devoid of most of the terminologies and concepts tagged to it from the view point of data structures. This distinction needs to be borne in mind when one defines a tree- rather 'redefines' tree as a special kind of graph in this chapter.

Though graph theory has turned out to be a vast area with innumerable applications, we restrict the scope of this chapter to introducing graphs as effective data structures only. Hence only those concepts and terminologies needed to promote this aspect of graphs are dealt with.

- 9.1 *Introduction*
- 9.2 *Definitions and Basic Terminologies*
- 9.3 *Representations of Graphs*
- 9.4 *Graph Traversals*
- 9.5 *Applications*

Introduction

9.1

The history of graphs dates back to 1736 in what is now referred to as the classical *Koenigsberg bridge problem*. In the town of Koenigsberg in Eastern Prussia, the island of Kneiphof existed in the middle of the river Pregel. The river bifurcated itself bordering the land areas as shown in Fig. 9.1. There were seven bridges connecting the land areas as shown in the figure. The problem was to find if the people of the town could walk on the seven bridges once only, starting from any land area, and returning to the starting land area after traversing all the bridges.

An example walk is listed below:

Start from the land area *P*-traverse bridge1; land area *R*-traverse bridge 3; land area *Q*-traverse bridge 4; land area *R*-traverse bridge 5; land area *S*-traverse bridge 7; land area *Q*-traverse bridge 7; land area *S*-traverse bridge 6; land area *P*-traverse bridge 2; land area *R*.

This walk, neither does it traverse all bridges once only nor does it reach its starting point which is land area *P*.

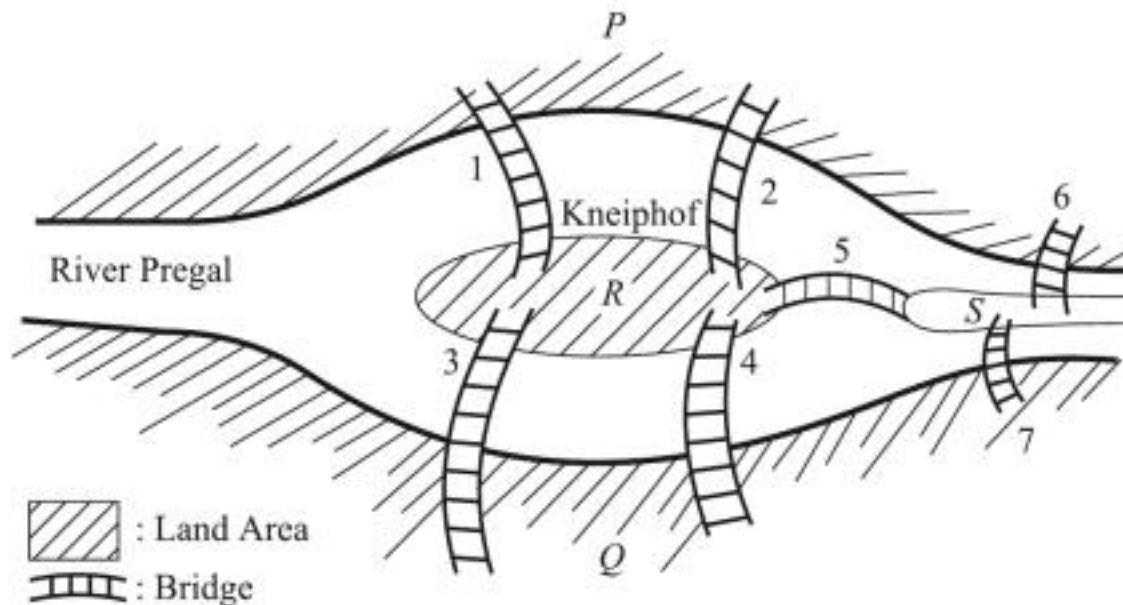


Fig. 9.1 The Koenigsberg bridge problem

It was left to Euler to solve the puzzle of Koenigsberg bridge problem by stating that there is no way that people could walk across the bridges once only and return to the starting point. The solution to the problem was arrived at by representing the land areas as circles called *vertices* and bridges as arcs called *links* or *edges* connecting the circles. Defining the *degree of a vertex* to be the number of arcs converging on it, or in other words, the number of bridges which descend on a land area, Euler showed that *a walk is possible only when all the vertices have even degree*. That is, every land area needs to have only even number of bridges descending on it. In the case of the Koenigsberg bridge problem, all the vertices turned out to have an *odd degree*. Figure 9.2 illustrates the graph representation of the Koenigsberg bridge problem. This vertex-edge representation is what came to be known as a *graph* (here it is a *multigraph*). The walk which beginning from a vertex and returning to it after traversing all edges in the graph came to be known as an *Eulerian walk*.

Since this first application, graph theory has grown *in leaps and bounds* to encompass a wide range of applications in the fields of cybernetics, electrical sciences, genetics and linguistics, to quote a few.

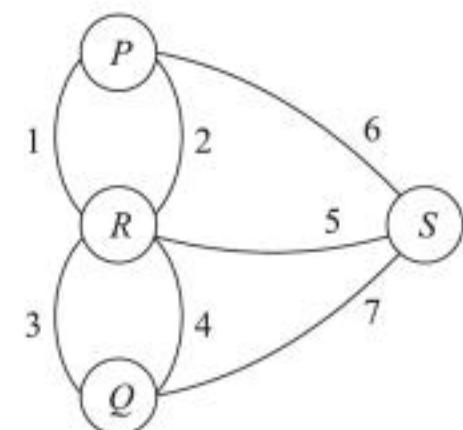


Fig. 9.2 Graph representation of the Koenigsberg bridge problem

Definitions and Basic Terminologies

9.2

Graph

A *graph* $G = (V, E)$ consists of a finite non empty set of *vertices* V also called *points* or *nodes* and a finite set E of unordered pairs of distinct vertices called *edges* or *arcs* or *links*.

Example Figure 9.3 illustrates a graph. Here $V = \{a, b, c, d\}$ and $E = \{(a, b), (a, c), (b, c), (c, d)\}$. However it is convenient to represent edges using labels as shown in the figure.

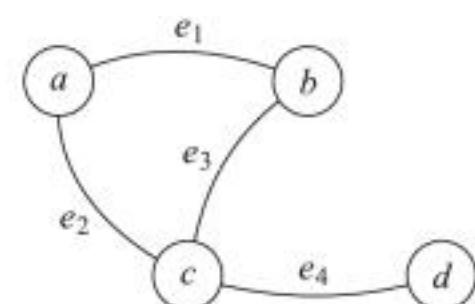


Fig. 9.3 A graph

V : Vertices : $\{a, b, c, d\}$

E : Edges : $\{e_1, e_2, e_3, e_4\}$

A graph $G = (V, E)$ where $E = \emptyset$, is called as a *null* or *empty graph*. A graph with one vertex and no edges is called a *trivial graph*.

Multigraph

A *multigraph* $G = (V, E)$ also consists of a set of vertices and edges except that E may contain *multiple edges* (i.e.) edges connecting the same pair of vertices, or may contain *loops* or *self edges* (i.e.) an edge whose end points are the same vertex.

Example Figure 9.4 illustrates a multigraph

Observe the multiple edges e_1, e_2 connecting vertices a, b and e_5, e_6, e_7 connecting vertices c, d respectively. Also note the self edge e_4 .

However, it has to be made clear that graphs do not contain multiple edges or loops and hence are different from multigraphs. The definitions and terminologies to be discussed in this section are applicable only to graphs.

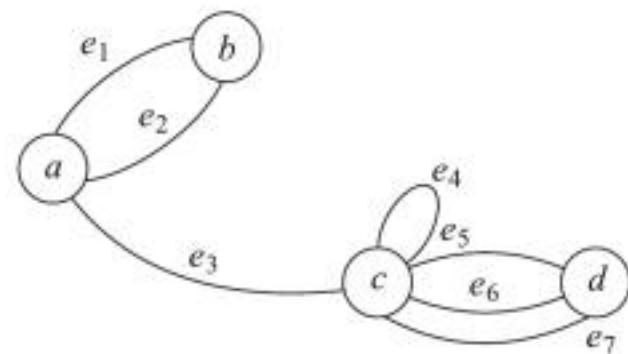


Fig. 9.4 A multigraph

Directed and undirected graphs

A graph whose definition (stated in Sec. 9.2) makes reference to *unordered pairs of vertices* as edges is known as an *undirected graph*. The edge e_{ij} of such an undirected graph is represented as (v_i, v_j) where v_i, v_j are distinct vertices. Thus an undirected edge (v_i, v_j) is equivalent to (v_j, v_i) .

On the other hand, *directed graphs* or *digraphs* make reference to edges which are directed (i.e.) edges which are *ordered pairs of vertices*. The edge e_{ij} is referred to as $\langle v_i, v_j \rangle$ which is distinct from $\langle v_j, v_i \rangle$ where v_i, v_j are distinct vertices. In $\langle v_i, v_j \rangle$, v_i is known as *tail* of the edge and v_j as the *head*.

Example Figure 9.5(a-b) illustrates a digraph and an undirected graph.

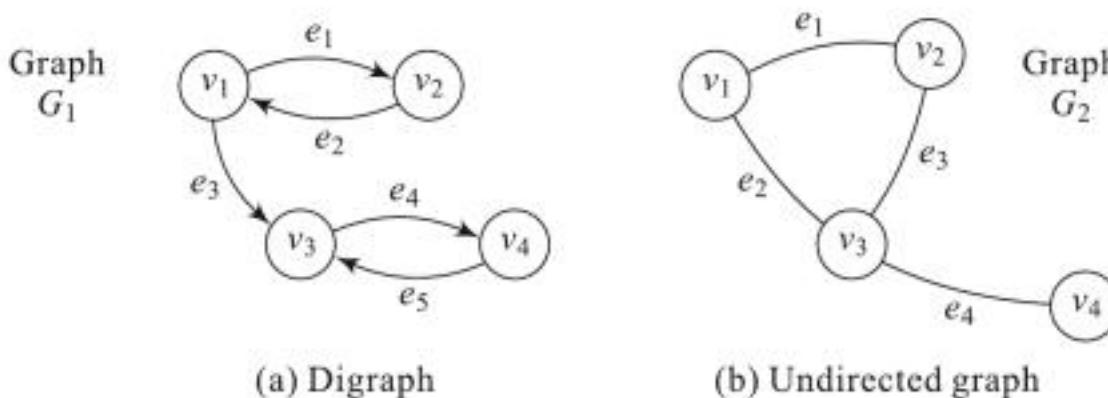


Fig. 9.5 A digraph and an undirected graph

In Fig. 9.5(a), e_1 is a directed edge between v_1 and v_2 , (i.e.) $e_1 = \langle v_1, v_2 \rangle$, whereas in Fig. 9.5(b) e_1 is an undirected edge between v_1 and v_2 , (i.e.) $e_1 = (v_1, v_2)$.

The list of vertices and edges of graphs G_1 and G_2 are:

Vertices (G_1) : $\{v_1, v_2, v_3, v_4\}$

Vertices (G_2) : $\{v_1, v_2, v_3, v_4\}$

Edges (G_1) : $\{\langle v_1, v_2 \rangle, \langle v_2, v_1 \rangle, \langle v_1, v_3 \rangle, \langle v_3, v_1 \rangle, \langle v_3, v_4 \rangle, \langle v_4, v_3 \rangle\}$ or $\{e_1, e_2, e_3, e_4, e_5\}$

Edges (G_2) : $\{(v_1, v_2), (v_1, v_3), (v_2, v_3), (v_3, v_4)\}$ or $\{e_1, e_2, e_3, e_4\}$

In the case of an undirected edge (v_i, v_j) in a graph, the vertices v_i, v_j are said to be *adjacent* or the edge (v_i, v_j) is said to be *incident on vertices* v_i, v_j . Thus in Fig. 9.5(b) vertices v_1, v_3 are adjacent to vertex v_2 and edges $e_1: (v_1, v_2), e_3: (v_2, v_3)$ are incident on vertex v_2 .

On the other hand, if $\langle v_i, v_j \rangle$ is a directed edge, then v_i is said to be *adjacent to* v_j and v_j is said to be *adjacent from* v_i . The edge $\langle v_i, v_j \rangle$ is *incident* to both v_i and v_j . Thus in Fig. 9.5(a) vertices v_2 and v_3 are adjacent from v_1 , and v_1 is adjacent to vertices v_2 and v_3 . The edges incident to vertex v_3 are $\langle v_1, v_3 \rangle, \langle v_3, v_4 \rangle$ and $\langle v_4, v_3 \rangle$.

Complete graphs

The number of distinct unordered pairs $(v_i, v_j), v_i \neq v_j$ in a graph with n vertices is ${}^n C_2 = \frac{n \cdot (n - 1)}{2}$

An n vertex undirected graph with exactly $\frac{n \cdot (n - 1)}{2}$ edges is said to be *complete*.

Example Figure 9.6 illustrates a complete graph. The undirected graph with 4 vertices has all its ${}^4 C_2 = 6$ edges intact.

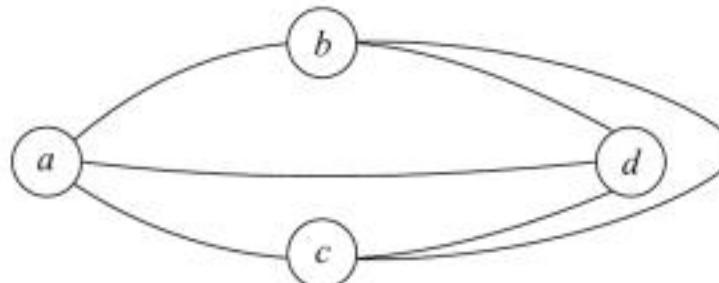


Fig. 9.6 A complete graph

In the case of a digraph with n vertices, the maximum number of edges is given by ${}^n P_2 = n \cdot (n - 1)$. Such a graph with exactly $n \cdot (n - 1)$ edges is said to be a *complete digraph*.

Example Figure 9.7(a) illustrates a digraph which is complete and Fig. 9.7(b) a graph which is not complete.

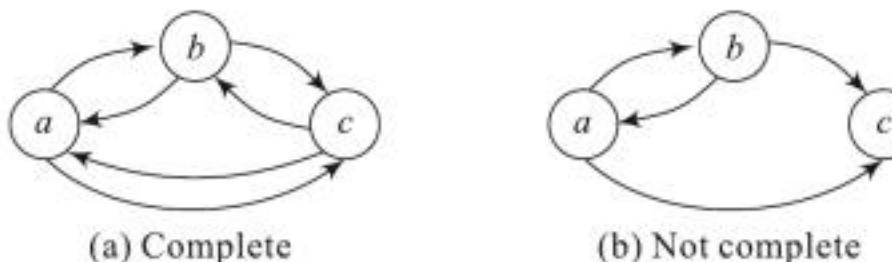


Fig. 9.7 Digraphs which are complete and not complete

Subgraph

A *subgraph* $G' = (V', E')$ of a graph $G = (V, E)$ is such that $V' \subseteq V$ and $E' \subseteq E$.

Example Figure 9.8 illustrates some subgraphs of the directed and undirected graphs shown in Fig. 9.5 (Graphs G_1 and G_2)

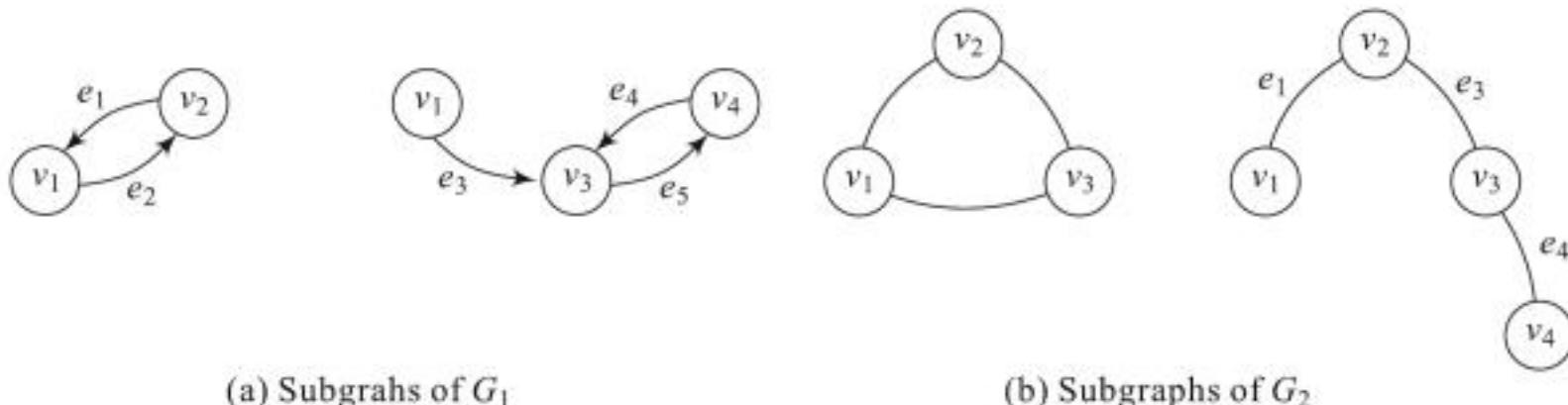


Fig. 9.8 Subgraphs of graphs G_1 and G_2 (Fig. 9.5)

Path

A *path* from a vertex v_i to vertex v_j in an undirected graph G is a sequence of vertices $v_{i'}, v_{l_1}, v_{l_2}, \dots, v_{l_k}, v_j$ such that $(v_{i'}, v_{l_1})$, $(v_{l_1}, v_{l_2}), \dots, (v_{l_k}, v_j)$ are edges in G . If G is directed then the path from v_i to v_j more specially known as a *directed path* consists of edges $\langle v_{i'}, v_{l_1} \rangle$, $\langle v_{l_1}, v_{l_2} \rangle, \dots, \langle v_{l_k}, v_j \rangle$ in G .

Example Figure 9.9(a) illustrates a path P_1 from vertex v_1 to v_4 in graph G_1 of Fig. 9.5(a) and Fig. 9.9(b) illustrates a path P_2 from vertex v_1 to v_4 of graph G_2 of Fig. 9.5(b).

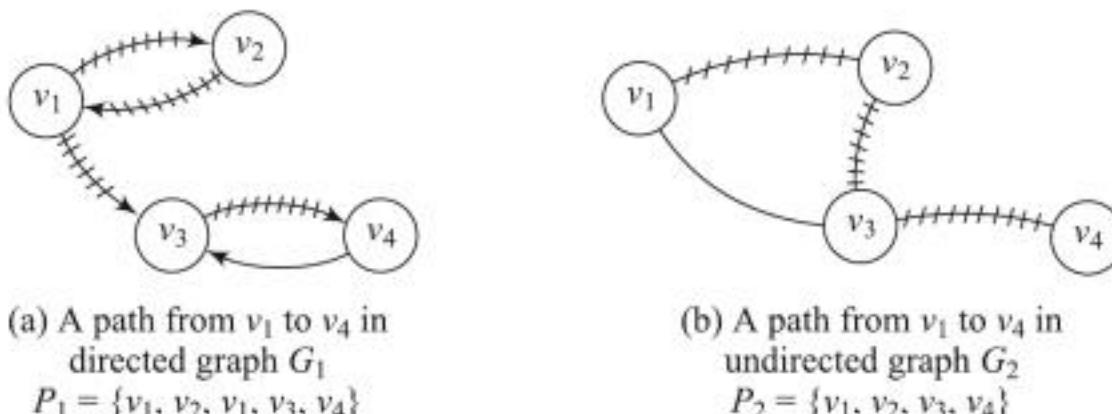


Fig. 9.9 Path between vertices of a graph (Fig. 9.5)

The *length* of a path is the number of edges on it.

Example In Fig. 9.9 the length of path P_1 is 4 and the length of path P_2 is 3. A *simple path* is a path in which all the vertices except possibly the first and last vertices are distinct.

Example In graph G_2 (Fig. 9.5(b)), the path from v_1 to v_4 given by $\{(v_1, v_2), (v_2, v_3), (v_3, v_4)\}$ and written as $\{v_1, v_2, v_3, v_4\}$ is a simple path whereas the path from v_3 to v_4 given by $\{(v_3, v_1), (v_1, v_2), (v_2, v_3), (v_3, v_4)\}$ and written as $\{v_3, v_1, v_2, v_3, v_4\}$ is not a simple path but a path due to the repetition of vertices.

Also in graph G_1 (Fig. 9.5(a)) the path from v_1 to v_3 given by $\{<v_1, v_2>, <v_2, v_1>, <v_1, v_3>\}$ written as $\{v_1, v_2, v_1, v_3\}$ is not a simple path but a mere path due to the repetition of vertices. However, the path from v_2 to v_4 given by $\{<v_2, v_1>, <v_1, v_3>, <v_3, v_4>\}$ written as $\{v_2, v_1, v_3, v_4\}$ is a simple path.

A *cycle* is a simple path in which the first and last vertices are the same. A cycle is also known as a *circuit*, *elementary cycle*, *circular path* or *polygon*.

Example In graph G_2 (Fig. 9.5(b)) the path $\{v_1, v_2, v_3, v_1\}$ is a cycle. Also, in graph G_1 (Fig. 9.5(a)) the path $\{v_1, v_2, v_1\}$ is a cycle or more specifically a *directed cycle*.

Connected graphs

Two vertices v_i, v_j in a graph G are said to be *connected* only if there is a path in G between v_i and v_j . In an undirected graph if v_i and v_j are connected then it automatically holds that v_j and v_i are also connected.

An undirected graph is said to be a *connected graph* if every pair of distinct vertices v_i, v_j are connected.

Example Graph G_2 (Fig. 9.5(b)) is connected whereas graph G_3 shown in Fig. 9.10 is not connected.

In the case of an undirected graph which is not connected, the maximal *connected subgraph* is called as a *connected component* or simply a *component*.

Example Graph G_3 (Fig. 9.10) has two connected components viz., graph G_{31} and G_{32} .

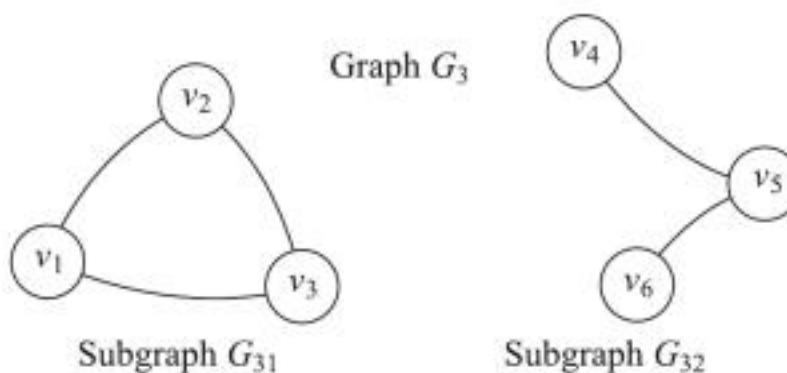


Fig. 9.10 An undirected graph with two connected components

A directed graph is said to be *strongly connected* if every pair of distinct vertices v_i, v_j are connected (by means of a directed path). Thus if there exists a directed path from v_i to v_j then there also exists a directed path from v_j to v_i .

Example Graph G_4 shown in Fig. 9.11 is strongly connected.

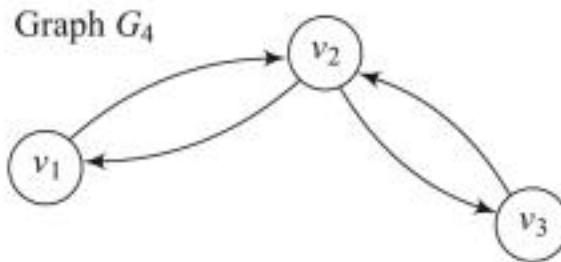


Fig. 9.11 A strongly connected graph

However, the digraph shown in Fig. 9.12 is not strongly connected but is said to possess two *strongly connected components*. A strongly connected component is a maximal subgraph that is strongly connected.

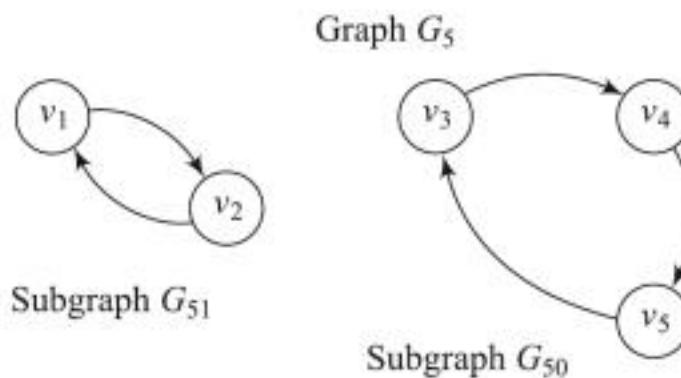


Fig. 9.12 Strongly connected components of a digraph

Trees

A *tree* is defined to be a connected acyclic graph. The following properties are satisfied by a tree:

- (i) There exists a path between any two vertices of the tree, and
- (ii) No cycles must be present in the tree. In other words, trees are acyclic.

Example Figure 9.13(a) illustrates a tree. Figure 9.13(b) illustrates graphs which are not trees due to the violation of the property of acyclicity and connectedness respectively.

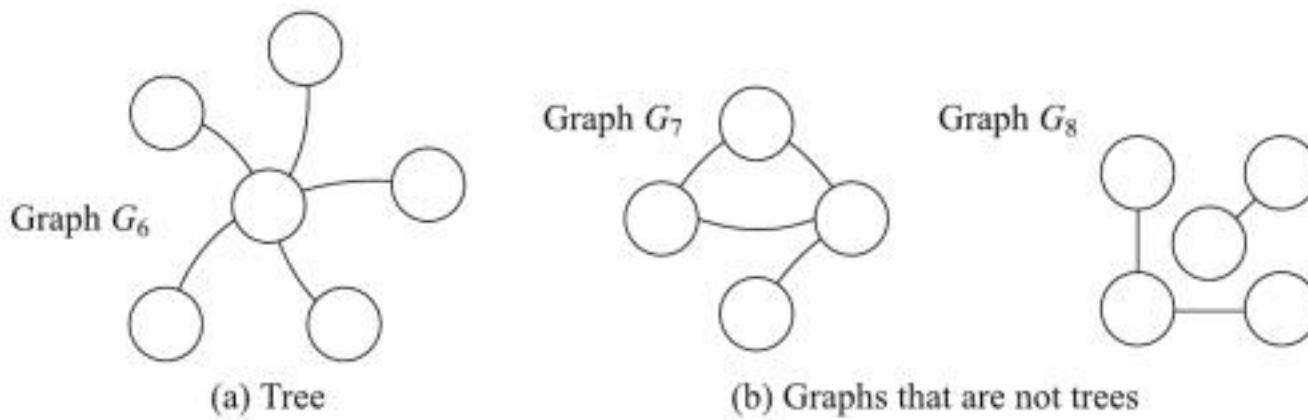


Fig. 9.13 Graphs which are trees and not trees

Note the marked absence of any hierarchical structure and its allied terminologies of parent, child, sibling, ancestor, level etc insisted upon in the tree data structure. However, both the definitions of trees—as a data structure and a type of graph—agree on the principles of connectedness and acyclicity.

Degree

The *degree* of a vertex in an undirected graph is the number of edges incident to that vertex. A vertex with degree one is called as a *pendant vertex* or *end vertex*. A vertex with degree zero and hence has no incident edges is called an *isolated vertex*.

Example In graph G_2 (Fig. 9.5(b)) the degree of vertex v_3 is 3 and that of vertex v_2 is 2. In the case of digraphs, we define the *indegree* of a vertex v to be the number of edges with v as the head and the *outdegree* of a vertex to be number of edges with v as the tail.

Example In graph G_1 (Fig. 9.5(a)) the indegree of vertex v_3 is 2 and the out degree of vertex v_4 is 1.

Isomorphic graphs

Two graphs are said to be *isomorphic* if,

- (i) they have the same number of vertices
- (ii) they have the same number of edges
- (iii) they have an equal number of vertices with a given degree

Example Figure 9.14 illustrates two graphs which are isomorphic.

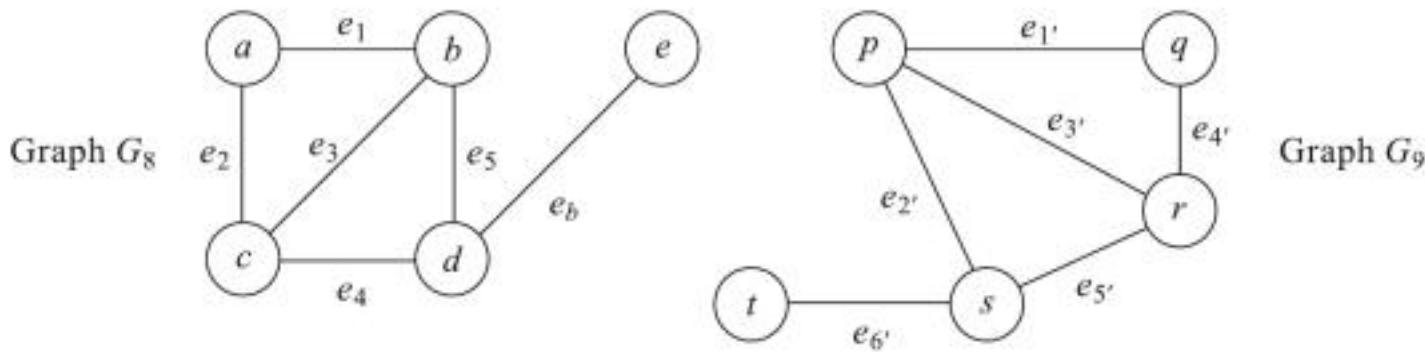


Fig. 9.14 Isomorphic graphs

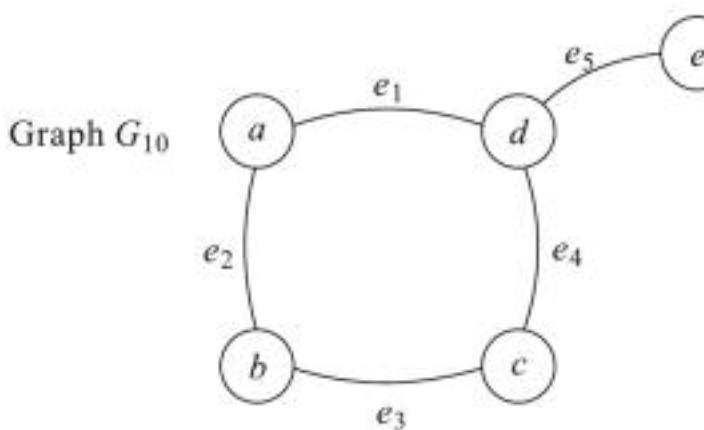
The property of isomorphism can be verified on the lists of vertices and edges of the two graphs G_8 and G_9 when superimposed as shown below:

Vertices (G_8) :	a	b	c	d	e	
	↔	↔	↔	↔	↔	
Vertices (G_9) :	q	p	r	s	t	
Degree of the vertices :	2	3	3	3	1	
Edges (G_8) :	e_1	e_2	e_3	e_4	e_5	e_6
Edges (G_9) :	e'_1	e'_4	e'_3	e'_2	e'_5	e'_6

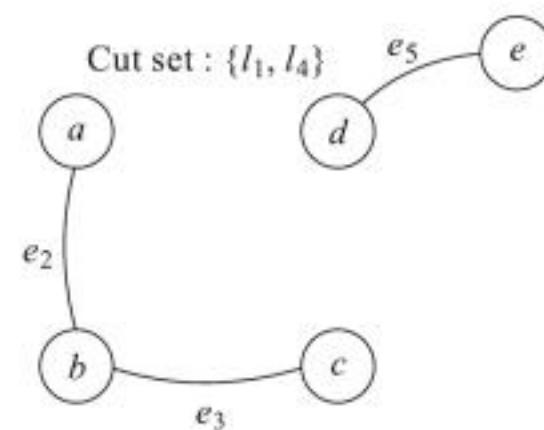
Cut set

A *cut set* in a connected graph G is the set of edges whose removal from G leaves G disconnected, provided the removal of no proper subset of these edges disconnects the graph G . Cut sets are also known as *proper cut set* or *cocycle* or *minimal cut set*.

Example Figure 9.15 illustrates the cut set of the graph G_{10} . The cut set $\{e_1, e_4\}$ disconnects the graph into two components as shown in the figure. $\{e_5\}$ is also another cut set of the graph.



(a) A graph



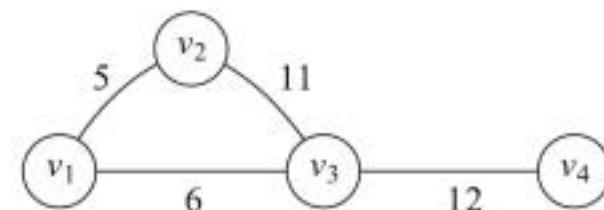
(a) A cut set of the graph

Fig. 9.15 A cut set of a graph

Labeled graphs

A graph G is called a *labeled graph* if its edges and / or vertices are assigned some data. In particular if the edge e is assigned a non negative number $l(e)$ then it is called the *weight* or *length* of the edge e .

Example Figure 9.16 illustrates a labeled graph. A graph with weighted edges is also known as a *network*.

**Fig. 9.16** A labeled graph

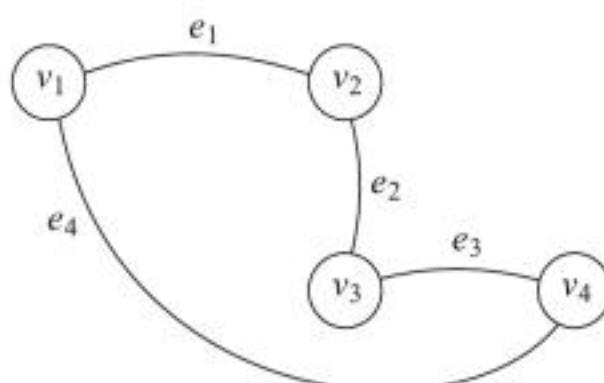
Eulerian graph

A walk starting at any vertex going through each edge exactly once and terminating at the start vertex is called an *Eulerian walk* or *Euler line*.

The Koenigsberg bridge problem was in fact a problem of obtaining an Eulerian walk for the graph concerned. The solution to the problem discussed in Sec. 9.1 can be rephrased as, an Eulerian walk is possible only if the degree of each vertex in the graph is even.

Given a connected graph G , G is an *Euler graph* iff all the vertices are of even degree.

Example Figure 9.17 illustrates an Euler graph. $\{e_1, e_2, e_3, e_4\}$ shows a Eulerian walk. The even degree of the vertices may be noted.

**Fig. 9.17** An Euler graph

Hamiltonian circuit

A *Hamiltonian circuit* in a connected graph is defined as a closed walk that traverses every vertex of G exactly once, except of course the starting vertex at which the walk terminates.

A *circuit* in a connected graph G is said to be *Hamiltonian* if it includes every vertex of G . If any edge is removed from a Hamiltonian circuit then what remains is referred to as a *Hamiltonian path*. Hamiltonian path traverses every vertex of G .

Example

Figure 9.18 illustrates a Hamiltonian circuit.

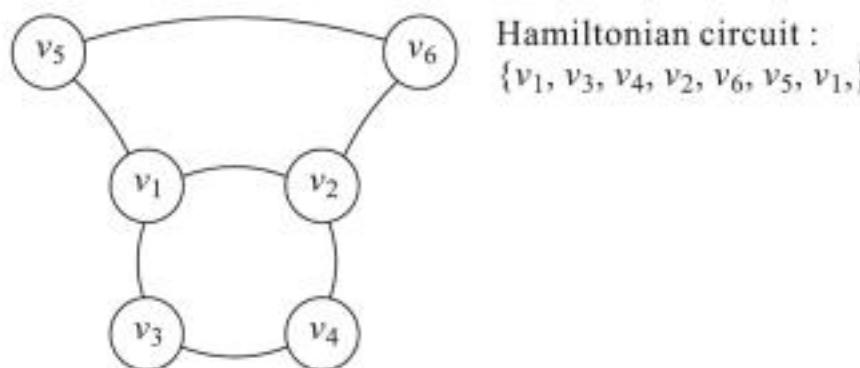


Fig. 9.18 A Hamiltonian circuit

Representations of Graphs

9.3

The representation of graphs in a computer can be categorized as (i) *sequential representation* and (ii) *linked representation*. Of the two, though sequential representation has several methods, all of them follow a matrix representation thereby calling for their implementation using arrays.

The linked representation of a graph makes use of a singly linked list as its fundamental data structure.

Sequential representation of graphs

The sequential or the matrix representation of graphs have the following methods:

- (i) *Adjacency matrix representation*
- (ii) *Incidence matrix representation*
- (iii) *Circuit matrix representation*
- (iv) *Cut set matrix representation*
- (v) *Path matrix representation*

Adjacency matrix representation

The *adjacency matrix* of a graph G with n vertices is an $n \times n$ symmetric binary matrix given by $A = [a_{ij}]$ defined as

$$a_{ij} = \begin{cases} 1 & \text{if the } i^{\text{th}} \text{ and } j^{\text{th}} \text{ vertices are adjacent (i.e.) there is an edge} \\ & \text{connecting the } i^{\text{th}} \text{ and } j^{\text{th}} \text{ vertices} \\ 0 & \text{otherwise, (i.e.) if there is no edge linking the vertices.} \end{cases}$$

Example

Figure 9.19(a) illustrates an undirected graph whose adjacency matrix is shown in Fig. 9.19(b).

It can easily be seen that while adjacency matrices of undirected graphs are symmetric, nothing can be said about the symmetry of the adjacency matrix of digraphs.

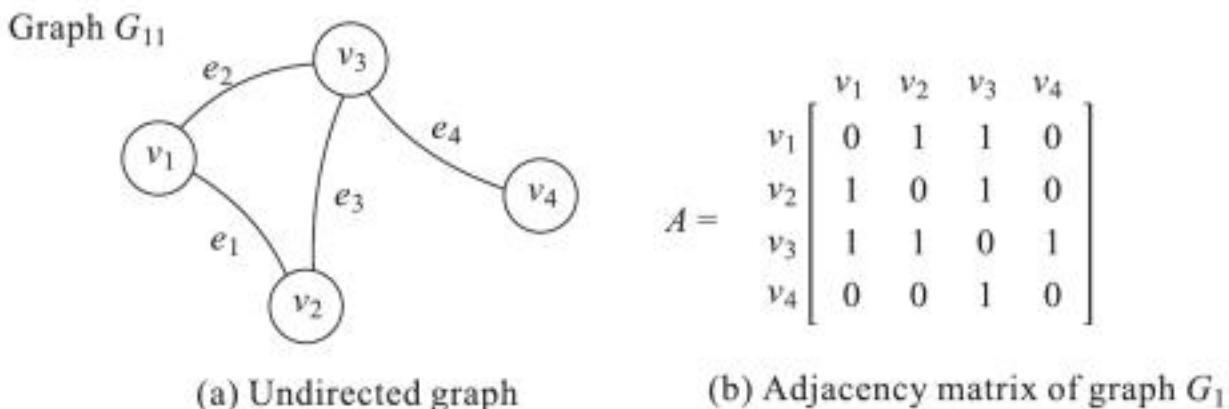
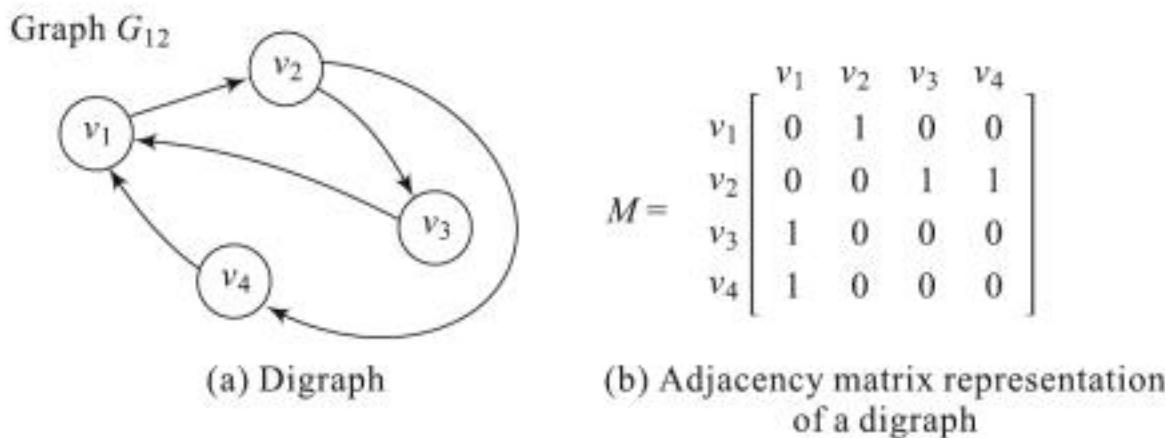
**Fig. 9.19** Adjacency matrix of an undirected graph**Example**

Figure 9.20(a-b) illustrates a digraph and its adjacency matrix representation.

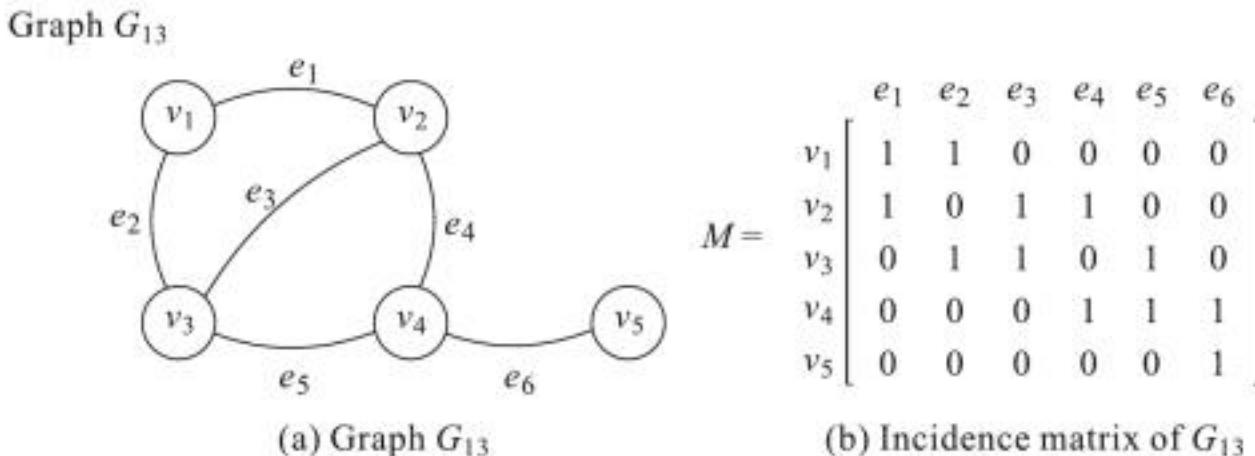
**Fig. 9.20** Adjacency matrix representation of a digraph**Incidence matrix representation**

Let G be a graph with n vertices and e edges. Define an $n \times e$ matrix $M = [m_{ij}]$ whose n rows correspond to n vertices and e columns correspond to e edges, as

$$m_{ij} = \begin{cases} 1 & \text{if the } j^{\text{th}} \text{ edge } e_j \text{ is incident on the } i^{\text{th}} \text{ vertex } v_i, \text{ otherwise} \\ 0 & \end{cases}$$

Matrix M is known as the *incidence matrix* representation of the graph G .

Example Consider the graph G_{13} shown in Fig. 9.21(a), the incidence matrix representation for the graph is given in Fig. 9.21(b).

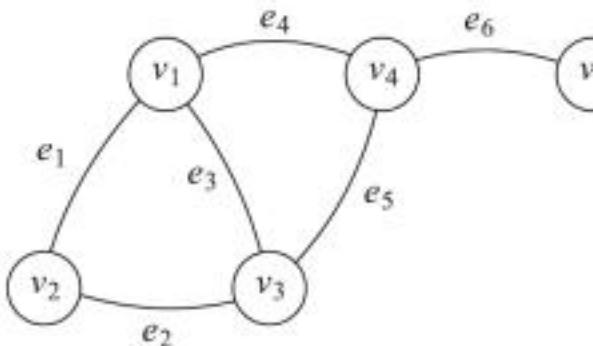
**Fig. 9.21** Incidence matrix representation of a graph

Circuit matrix representation

For a graph G let the number of different circuits be t and the number of edges be e . Then the *circuit matrix* $C = [C_{ij}]$ of G is a $t \times e$ matrix defined as

$$C_{ij} = \begin{cases} 1 & \text{if the } i^{\text{th}} \text{ circuit includes the } j^{\text{th}} \text{ edge, otherwise} \\ 0 & \end{cases}$$

Example Consider the graph G_{14} shown in Fig. 9.22(a). The circuits for this graph expressed in terms of their edges are 1: $\{e_1, e_2, e_3\}$ 2: $\{e_3, e_4, e_5\}$ 3: $\{e_1, e_2, e_5, e_4\}$. The circuit matrix C of order 3×6 is shown in Fig. 9.22(b).

(a) Graph G_{14}

$$C = \begin{bmatrix} e_1 & e_2 & e_3 & e_4 & e_5 & e_6 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 2 & 0 & 0 & 1 & 1 & 1 & 0 \\ 3 & 1 & 1 & 0 & 1 & 1 & 0 \end{bmatrix}$$

Circuit 1 : $\{e_1, e_2, e_3\}$
 Circuit 2 : $\{e_3, e_4, e_5\}$
 Circuit 3 : $\{e_1, e_2, e_5, e_4\}$

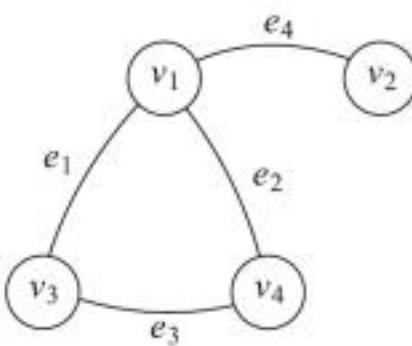
(b) Circuit matrix of G_{14} **Fig. 9.22** Circuit matrix representation of a graph

Cut set matrix representation

For a graph G , a matrix $S = [s_{ij}]$ whose rows correspond to cut sets and columns correspond to edges of the graph is defined to be a *cut set matrix* if

$$s_{ij} = \begin{cases} 1 & \text{if the } i^{\text{th}} \text{ cut set contains the } j^{\text{th}} \text{ edge, otherwise} \\ 0 & \end{cases}$$

Example Consider the graph G_{15} shown in Fig. 9.23(a). The cut sets of the graph are 1: $\{e_4\}$ 2: $\{e_1, e_2\}$ 3: $\{e_2, e_3\}$ and 4: $\{e_1, e_3\}$. The cut set matrix representation is shown in Fig. 9.23(b).

(a) Graph G_{15}

$$S = \begin{bmatrix} e_1 & e_2 & e_3 & e_4 \\ 1 & 0 & 0 & 1 \\ 2 & 1 & 1 & 0 & 0 \\ 3 & 0 & 1 & 1 & 0 \\ 4 & 1 & 0 & 1 & 0 \end{bmatrix} \quad \begin{array}{l} 1 : \{e_4\} \\ 2 : \{e_1, e_2\} \\ 3 : \{e_2, e_3\} \\ 4 : \{e_1, e_3\} \end{array}$$

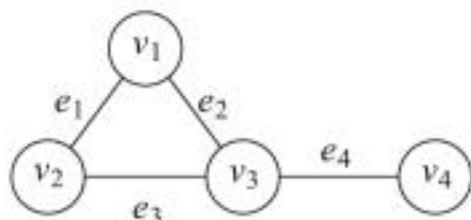
(b) Cut set matrix of G_{15} **Fig. 9.23** Cut set matrix representation of a graph

Path matrix representation

A *path matrix* is generally defined for a specific pair of vertices in a graph. If (u, v) is a pair of vertices then the path matrix denoted as $P(u, v) = [p_{ij}]$ is given by

$$p_{ij} = \begin{cases} 1 & \text{if the } j^{\text{th}} \text{ edge lies in the } i^{\text{th}} \text{ path between vertices } u \text{ and } v, \text{ otherwise} \\ 0 & \end{cases}$$

Example Consider the graph G_{16} shown in Fig. 9.24(a). The paths between vertices v_1 and v_4 are 1:{ e_2, e_4 } and 2:{ e_1, e_3, e_4 }. The path matrix representation is shown in Fig. 9.24(b).

(a) Graph G_{16}

$$P(v_1, v_4) = \begin{matrix} & \begin{matrix} e_1 & e_2 & e_3 & e_4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \end{matrix} & \left[\begin{matrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \end{matrix} \right] \end{matrix}$$

Paths :
1 : { e_2, e_4 }
2 : { e_1, e_3, e_4 }

(b) Path matrix between v_1, v_4 of G_{16} **Fig. 9.24** Path matrix representation

Of all these sequential representations, adjacency matrix representation represents the graph best and is the most widely used representation. The adjacency matrix A of a graph G with n vertices has an order of $n \times n$. As a consequence, graph algorithms which make use of the adjacency matrix representation are bound to report a time complexity of $O(n^2)$ since at least $n^2 - n$ entries (excluding the diagonal elements) are to be examined.

Linked representation of graphs

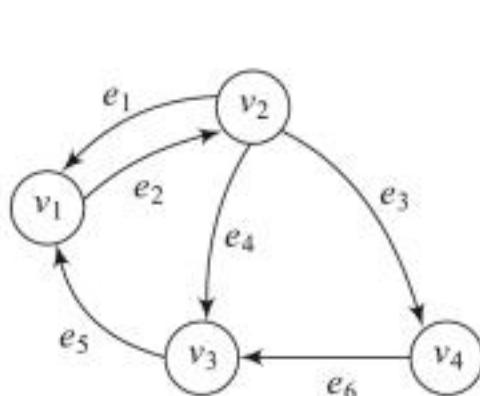
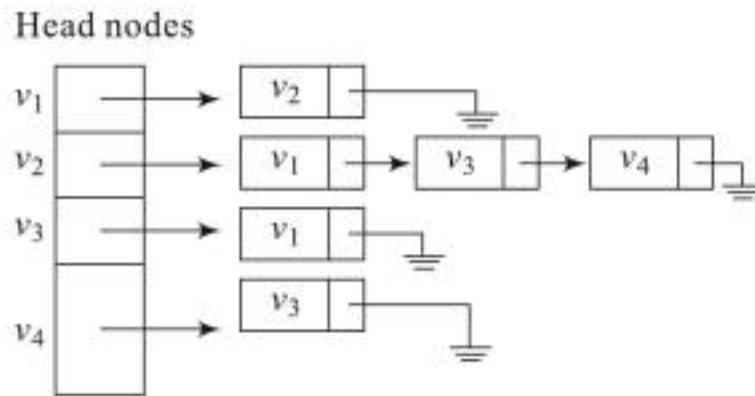
The linked representation of graphs is referred to as *adjacency list representation* and is comparatively efficient with regard to adjacency matrix representation.

Given a graph G with n vertices and e edges, the adjacency list opens n head nodes corresponding to the n vertices of graph G , each of which points to a singly linked list of nodes, which are adjacent to the vertex representing the head node.

Example Figure 9.25 illustrates a graph and its adjacency list representation.

It can easily be seen that if the graph is undirected, then the number of nodes in the singly linked lists put together is $2e$ where as in the case of digraphs the number of nodes is just e , where e is the number of edges in the graph.

In contrast to adjacency matrix representations, graph algorithms which make use of an adjacency list representation would generally report a time complexity of $O(n + e)$ or $O(n + 2e)$ based on whether the graph is directed or undirected respectively, thereby rendering them efficient.

(a) Graph G_{17} (b) Adjacency list representation of G_{17} **Fig. 9.25** Adjacency list representation of a graph



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.

(Contd.)

10	3 2 8	1 5 6 7 4 9 10	1 2 3 4 5 6 7 8 9 10 1 1 1 1 1 1 1 1 1 1
3	2 8	1 5 6 7 4 9 10 3	1 2 3 4 5 6 7 8 9 10 1 1 1 1 1 1 1 1 1 1
2	8	1 5 6 7 4 9 10 3 2	1 2 3 4 5 6 7 8 9 10 1 1 1 1 1 1 1 1 1 1
8		1 5 6 7 4 9 10 3 2 8	1 2 3 4 5 6 7 8 9 10 1 1 1 1 1 1 1 1 1 1
		1 5 6 7 4 9 10 3 2 8	Breadth first traversal ends

The breadth first traversal starts from vertex 1 and visits vertices 5,6,7 which are adjacent to it, while enqueueing them into queue Q. In the next shot, vertex 5 is dequeued and its adjacent, unvisited vertices 4, 9 are visited next and so on. The process continues until the queue Q which keeps track of the adjacent vertices is empty.

If an adjacency matrix representation had been used, the time complexity of the algorithm would have been $O(n^2)$ since to visit each vertex, the while loop incurs a time complexity of $O(n)$. On the other hand, the use of adjacency list only calls for the examination of those nodes which are adjacent to the given node thereby curtailing the time complexity of the loop to $O(e)$.

Depth first traversal

In this section we discuss the depth first traversal of an undirected graph. The traversal starts from a vertex u which is said to be visited. Now, all the nodes v_i adjacent to vertex u are collected and the first occurring vertex v_1 is visited, deferring the visits to other vertices. The nodes adjacent to v_1 viz., w_{1k} are collected and the first occurring adjacent vertex viz., w_{11} is visited deferring the visit to other adjacent nodes and so on. The traversal progresses until there are no more visits possible.

Algorithm 9.2 illustrates a recursive procedure to perform the depth first traversal of graph G.

Algorithm 9.2: Depth first traversal

```

Procedure DFT(s)
    /* s is the start vertex */
    visited(s) = 1;
    Print (s); /* Output visited vertex */
    for each vertex v adjacent to s do
        if visited(v) = 0 then call DFT(v);
    end
end DFT

```



The depth first traversal as its name indicates visits each node, that is, the first occurring among its adjacent nodes and successively repeats the operation, thus moving 'deeper and deeper' into the graph. In contrast, breadth first traversal moves side ways or breadth ways in the graph. Example 9.2 illustrates a depth first traversal of a undirected graph.



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 convenient to represent the weighted digraph using its cost matrix $\text{COST}_{N \times N}$. The cost matrix records the distances between cities connected by an edge.

Dijkstra's algorithm has a complexity of $O(N^2)$ where N is the number of vertices (cities) in the weighted digraph.

Algorithm 9.3: Dijkstra's algorithm for the single source shortest path problem

Procedure DIJKSTRA_SSSP(N , COST)

```

/* $N$  is the number of vertices labeled { 1, 2, 3,... $N$ } of the weighted
digraph. COST[1:N,1:N] is the cost matrix of the graph. If there is
no edge then COST [i, j] =  $\infty$ */

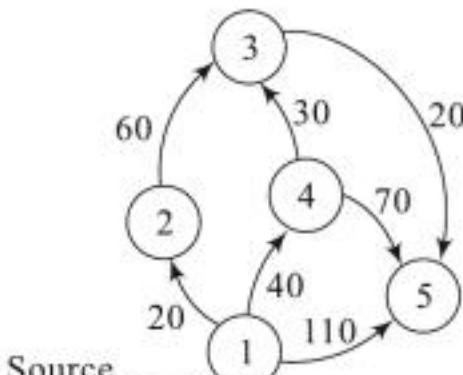
/* The procedure computes the cost of the shortest path from vertex
1 the source, to every other vertex of the weighted digraph */

T = {1}; /* Initialize T to source vertex */
for i = 2 to N do
  DISTANCE[i] = COST[1,i]; /*Initialize DISTANCE vector
  end                                to the cost of the edges connecting
                                         vertex i with the source vertex 1. If
                                         there is no edge then COST [1, i] =  $\infty$ */

for i = 1 to N -1 do
  Choose a vertex u in V - T such that DISTANCE[u]
  is a minimum;
  Add u to T;
  for each vertex w in V-T do
    DISTANCE [w] = minimum (DISTANCE[w],
                           DISTANCE [u] + COST [u, w] );
  end
end
end DIJKSTRA-SSSP

```

Example 9.3 Consider the weighted digraph of cities and its cost matrix shown in Fig. 9.28. Table 9.2 shows the trace of the Dijkstra's algorithm.



(a) Weighted digraph

Cost:	1	2	3	4	5
1	0	20	∞	40	110
2	∞	0	60	∞	0
3	∞	∞	0	∞	20
4	∞	∞	30	0	70
5	∞	∞	∞	∞	0

(b) Cost matrix $C_{5 \times 5}$

Fig. 9.28 A weighted digraph and its cost matrix



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

Minimum cost spanning trees

Consider an application where n stations are to be linked using a communication network. The laying of communication links between any two stations involves a cost. The problem is to obtain a network of communication links which while preserving the connectivity between stations does it with minimum cost. If the problem were to be modeled as a weighted graph, the ideal solution to the problem would be to extract a subgraph termed *minimum cost spanning tree* which while preserving the connectedness of the graph yields minimum cost.

Let $G = (V, E)$ be an undirected connected graph. A subgraph $T = (V, E')$ of G is a *spanning tree* of G iff T is a tree.

Example For the connected graph undirected shown in Fig. 9.29 (a), some of the spanning trees extracted from the graph are shown in Fig. 9.29(b).

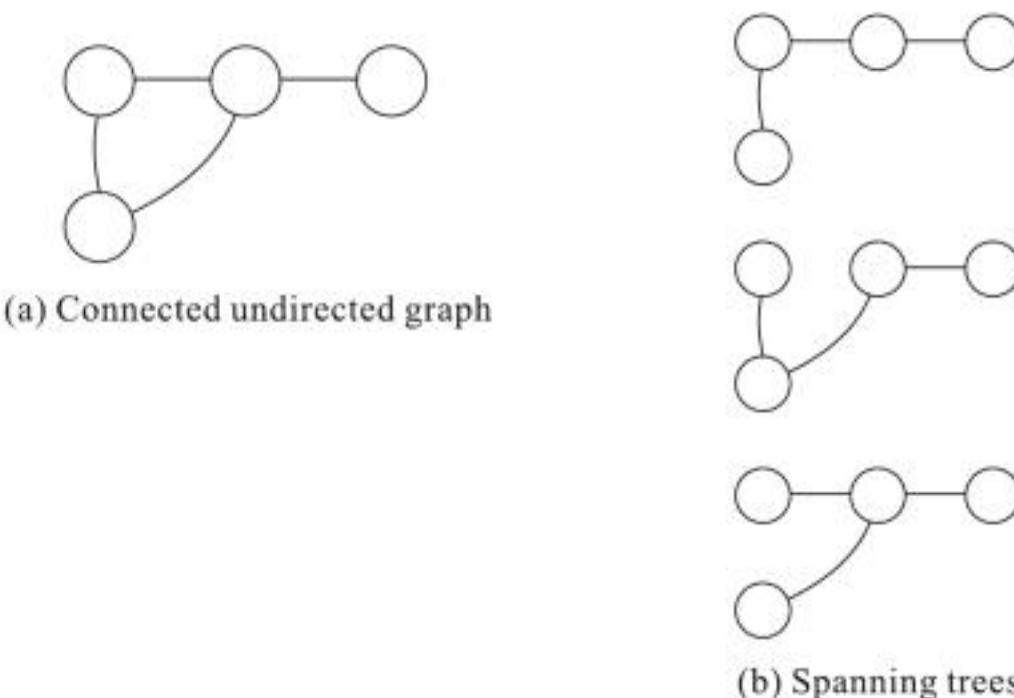


Fig. 9.29 Spanning trees of a graph

Given a connected undirected graph there are several spanning trees that may be extracted from it. Now given $G = (V, E)$ to be a connected, weighted undirected graph where each edge involves a cost, the extraction of a spanning tree extends itself to the extraction of a minimum cost spanning tree. A minimum cost spanning tree is a spanning tree which has a minimum total cost. Algorithm 9.4 illustrates Prims algorithm for the extraction of minimum cost spanning trees.

A spanning tree of a graph G with n vertices will have $(n - 1)$ edges. This is due to the property that a tree with n vertices has always $(n - 1)$ edges (Refer Illustrative Problem 9.9). Also, addition of even one single edge results in the spanning tree losing its property of acyclicity and removal of one single edge results in its losing the property of connectivity.

The time complexity of Prims algorithm is $O(n^2)$.

Example 9.5 Consider the connected, weighted, undirected graph shown in Fig. 9.30. Table 9.3 illustrates the trace of the PRIM's algorithm on the graph.



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.



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.



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.



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.



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.



INDEX

- 2-3 trees 270
- 2-3-4 trees 294
- 2-4 trees 270, 294
- Abstract Data Type 5
- Addition of polynomials 106
- Adjacency list [198](#)
 - matrix [195](#)
 - matrix representation [195](#)
- ADT
 - arrays 34
 - binary trees [174](#)
 - graphs 208
 - links [110](#)
 - queues 75
 - singly linked lists [111](#)
 - stacks 48
- Algorithm 2
 - definition 3
 - development 4
 - properties 3
 - structure 3
- Alternate keys 355
- Amortized analysis of splay trees 317
- Apriori analysis 9
 - recursive functions 17
 - analysis 17
 - approach 9
- Array 27
 - ADT 34
 - multi-dimensional 28
 - number of elements 27
 - one-dimensional 27
 - operations 27
 - representation 28
 - two-dimensional 27
- Asymptotic notations 11
- Available space [130](#)
- Average case complexity 14
- AVL search tree 229
 - deletion 236
 - insertion 230
 - retrieval 230
 - tree 229
- B tree of order m 293
 - definition 269
 - deletion 273
 - height 277
 - inserting 270
 - searching 270
 - trees 269
 - trees of order 4 293
- B+ trees 283
- Balance factor 229
- Balanced k -way merging on disks 442
 - merge sort 438, 441
 - P -way merging on tapes 441
 - trees 228
- Balancing symbols [133](#)
- Base address 29
- Best case time complexity 14
- Bin sort 422
- Binary search 378
 - ADT [174](#)
 - basic terminologies [155](#)
 - definition 218
 - deletion 222
 - drawbacks 227
 - growth [168](#)
 - insertion 222
 - representation [156](#), 219
 - retrieval 220
 - representation [156](#)

- search tree 218
- tree traversals 158
- traversals 158, 172
- trees 155
- types 155
- Bisection 378
- Black condition 295
- Block anchor 358
- Branch node 277
- Breadth first traversal 199
- Bubble sort 395
- Bucket sort 422
- Buffer handling 440
- Candidate keys 355
- Cascade merge sort 447
- Chained hash tables 340
- Chaining 339
- Circuit matrix 195
 - matrix representation 197
- Circular queues 59, 62
 - operations 62
- Circularly linked list 87, 93
 - primitive operations 95
 - representation 93
- Classification 6
- Cluster indexing 360
- Collating 401
- Collision 333
 - resolution 338
- Complexity 8
- Construction of heap 415
- Conversion of infix expression to postfix expression 172
- Cut set matrix 195
 - matrix representation 197
- Cycle 191
- Data abstraction 6
 - classification 5
 - definition 5
 - structure 2, 5
 - structures and algorithms 4
 - algorithms 4
 - type 5
- Decision tree
 - binary search 379
 - Fibonacci search 381
- Deletion from a binary search tree 222
 - from an AVL search tree 236
- Dense index 358
- Depth first traversal 201
- Deque 70
- Dequeueing a queue 56
- Development of an algorithm 4
- Dictionary 331
- Digital sort 422
- Dijkstra's algorithm 203
- Diminishing increment sort 405
- Direct file organization 346, 363
- Doubly linked lists 87, 98
 - advantages and disadvantages 99
 - operations 100
 - representation 98
- Drawbacks of a binary search tree 227
 - of sequential data structures 84
- Dynamic memory management 130
- Enqueuing a queue 56
- Evaluation of expressions 43
- Exponential time complexities 12
- Expression trees 169
- External hashing 363
 - memory 353
 - sorting 394, 435
 - storage devices 353, 436
- Fibonacci merge 447
 - search 381
- File indexing 282
 - operations 356
 - organization 346
- Files 353, 354
- First Come First Served (FCFS) 56
 - In First Out (FIFO) 56
- FLIFLO (First in Last In or First out Last Out) 70
- Folding 334
- Free storage pool 130
- Garbage collection 130
- Graph 187
 - complete graphs 189
 - connected graphs 191
 - cut set 193
 - degree 193
 - directed 188
 - empty graph 188
 - Eulerian graph 194
 - Hamiltonian circuit 194
 - isomorphic graphs 193
 - labeled graphs 194

- multigraph 188
- path 190
- subgraph 190
- trees 192
- undirected 188
- Graph 188
 - search 384
- Growth of threaded binary trees 168
- Hard disks 436
- Hash function H 332
 - functions 333
 - table 332
- Hashing 332
- Head node 95
- Heap 356, 415
 - sort 414
- Height balanced trees 228
- Home bucket 335
- Huffman coding 260
- Incidence matrix 195
 - matrix representation 196
- Index 282
- Indexed sequential file organization 358
 - sequential search 385
- Infix, prefix and postfix expressions 45
- Information node 277
- Inorder traversal 158
- Input buffers 440
 - restricted deque 70
- Insertion and deletion in a singly linked list 88
 - into a binary search tree 222
 - into an AVL search tree 230
 - sort 396
- Internal memory 353
 - sorting 394, 435
- Interpolation search 376
- ISAM files 358
- Join operation 344
- k*-way merging 403
- Keys 355
- Keyword table 342
- Koenigsberg bridge problem 186
- L category rotations 243
- Last In First Out 39
- Lb*0, *Lb*1 and *Lb*2 rotations 322
- Lexicographic search trees 277
- Limitations of linear queues 61
- Linear data structures 6
 - open addressed hash tables 336
 - open addressing 334
 - queues 59
 - search 373
- Linked list 86
- Linked queues 124
 - operations 124, 125
 - representation 6, 168
 - representation of graphs 198
 - stack 124
 - stack operations 125
- LL* rotation 230
- LLb*, *LRb*, *RRb* 297
- LLr*, *LRr*, *RRr* 297
- Loading factor 338
- Logarithmic search 378
- LR* rotation 232
- Lr*0, *Lr*1 and *Lr*2 rotations 323
- m-way search trees 262
 - definition 263
 - deleting 265
 - drawbacks 268
 - inserting 265
 - node structure 263
 - representation 263
 - searching 264
- Magnetic disks 436, 437
 - tapes 436
- Master file 357
- Merge sort 401, 435
- Merging 401
- Merits of linked data structures 85
- Minimum cost spanning trees 206
- Modular arithmetic 334
- MSD first sort 425
- Multi-dimensional array 28
 - way trees 262
- Multilevel indexing 360
- Multiply linked list 87, 103
- N*-dimensional array 32
- Natural join 344
- Non-linear data structures 6
- Number of elements in an array 27
- One-dimensional array 27, 29

- Operations
 circular queue 62
 doubly linked lists 100
 linked stacks and linked queues 124
 queues 57
- Optimal binary search tree 246
- Ordered linear search 373, 374
 lists 33
- Output buffer 440
 restricted deque 70
- Overflow 335
- Partitioning 410
- Path matrix 195
 matrix representation 197
- Pile organization 356
- Pivot element 410
- Polynomial representation 133
 time complexities 12
- Polyphase merge sort 445
- Posteriori testing 8
- Postorder traversal 158, 162
- Preorder traversal 158, 162
- Primary indexing 360
 keys 355
- Primitive operations on circularly linked lists 95
- Prims algorithm 206
- Priority queues 66
- Quadratic probing 339
- Queue 56
 dequeuing 56
 enqueueing 56
 implementation 57
 list 147
 operations 57
- Quick sort 410
- R*-1 rotation 242
- R*0 rotation 240
- R*1 rotation 241
- Radix sort 422
- Random access storage devices 353
 probing 339
- Rb*0, *Rb*1 304
- Rb*2 imbalances 304
- Records 354
- Recurrence relations 15
- Recursion 15
- Recursive merge sort 404
 procedures 15
 programming 43
- Red condition 295
- Red-Black trees 293, 297, 303, 310
 definition 295
 deleting 303
 inserting 297
 introduction 293
 representation 296
 searching 296
 time complexity 310
- Rehashing 338
- Representation of a binary search tree 219
 of a red-black tree 296
 of a singly linked list 87
 of arrays in memory 28
N-dimensional array 32
 one-dimensional array 29
 three-dimensional array 31
 two-dimensional array 29
- Reserved pool 132
- Retrieval from an AVL search tree 230
- RL* rotation 233
- RLb* imbalances 297
- RLr* imbalances 297
- RR* rotation 233
- Rr*0, *Rr*1 304
- Rr*2 imbalances 304
- Runs 435
- Searching a red-black tree 296
- Secondary memory 353
 indexing 361
 keys 355
 storage devices 353
- Selection sort 399
 tree 443
- Self organizing sequential search 375
- Sequential 6
 file organisation 357
 search 373
 storage devices 353
- Shell sort 405
- Sifting 397
- Single-source, shortest-path problem 203
- Singly linked list 87
 ADT 111
 insertion and deletion 88
 representation 87
- Sinking 397
- Skewed binary tree 156
- Sorting by distribution 394
 by exchange 394

- by insertion 394
- by merge 401
- by selection 394
- with disks 441
- with tapes 438
- Space complexity 8
- Sparse index 358
 - matrix 32, [106](#)
 - matrix representation [109](#)
- Spell checker 284
- Splay rotations 311
 - trees 311
 - amortized analysis 317
- Stable 394
- Stack 39, 40
 - ADT 48
 - implementation 40
 - operations 40
- Super key 355
- Symbol tables 243
- Synonyms 333

- Tail recursion 45
- Tertiary storage devices 353
- Threaded binary trees [167](#)
- Three-dimensional array 31
- Time complexity 8
 - sharing system [71](#)
- Topological sorting [121](#)

- Tower of Hanoi 15
- Transaction file 357
- Transpose sequential search 375
- Traversable queue [137](#)
- Traversals of an expression tree [172](#)
- Tree of losers 443
 - of winners 443
 - search 384
- Trees [151](#)
 - basic terminologies [152](#)
 - definition [151](#)
 - representation [153](#)
- Tries 277
 - definition 277
 - deletion 279
 - insertion 279
 - representation 277
 - searching 279
- Truncation 334
- Two-dimensional array 27, 29

- Uniform binary search 379
- Unordered linear search 373, 374

- Worst case time complexity 14

- Zag 311
- Zig 311