

# DATA STRUCTURES AND ALGORITHMS

---

Concepts, Techniques and Applications



G A V PAI

---



**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

Second reprint 2008  
**RCLCRDRXRCBLX**

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



# CONTENTS

<u>Advance Praise</u>	v
<u>More from the Reviewers</u>	vi
<u>Preface</u>	ix
<b>1. Introduction</b>	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
Summary	7
<b>2. Analysis of Algorithms</b>	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
Summary	19
Illustrative Problems	20
Review Questions	25
Part I	
<b>3. Arrays</b>	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
Summary	34
Illustrative Problems	35
Review Questions	37
Programming Assignments	37

<b>4. Stacks</b>	<b>39</b>
<b>4.1 Introduction</b>	39
<b>4.2 Stack Operations</b>	40
<b>4.3 Applications</b>	43
<i>Summary</i>	48
<i>Illustrative Problems</i>	49
<i>Review Questions</i>	54
<i>Programming Assignments</i>	55
<b>5. Queues</b>	<b>56</b>
<b>5.1 Introduction</b>	56
<b>5.2 Operations on Queues</b>	57
<b>5.3 Circular Queues</b>	62
<b>5.4 Other Types of Queues</b>	66
<b>5.5 Applications</b>	71
<i>Summary</i>	75
<i>Illustrative Problems</i>	76
<i>Review Questions</i>	81
<i>Programming Assignments</i>	82
<b>Part II</b>	
<b>6. Linked Lists</b>	<b>84</b>
<b>6.1 Introduction</b>	84
<b>6.2 Singly Linked Lists</b>	87
<b>6.3 Circularly Linked Lists</b>	93
<b>6.4 Doubly Linked Lists</b>	98
<b>6.5 Multiply Linked Lists</b>	103
<b>6.6 Applications</b>	105
<i>Summary</i>	112
<i>Illustrative Problems</i>	113
<i>Review Questions</i>	119
<i>Programming Assignments</i>	121
<b>7. Linked Stacks and Linked Queues</b>	<b>123</b>
<b>7.1 Introduction</b>	123
<b>7.2 Operations on Linked Stacks and Linked Queues</b>	124
<b>7.3 Dynamic Memory Management and Linked Stacks</b>	130
<b>7.4 Implementation of Linked Representations</b>	132
<b>7.5 Applications</b>	133
<i>Summary</i>	137
<i>Illustrative Problems</i>	137
<i>Review Questions</i>	148
<i>Programming Assignments</i>	149
<b>Part III</b>	
<b>8. Trees and Binary Trees</b>	<b>151</b>
<b>8.1 Introduction</b>	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
<b>9.</b>	<b>Graphs</b>	<b>186</b>
9.1	Introduction	186
9.2	<u>Definitions and Basic Terminologies</u>	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
<b>Part IV</b>		
<b>10.</b>	<b>Binary Search Trees and AVL Trees</b>	<b>218</b>
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
<b>11.</b>	<b>B Trees and Tries</b>	<b>262</b>
11.1	Introduction	262
11.2	<i>m-way search trees: Definition and Operations</i>	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
<b>12.</b>	<b>Red-Black Trees and Splay Trees</b>	<b>293</b>
12.1	Red-Black Trees	293
12.2	Splay Trees	311

12.3 Applications	318
<i>Summary</i>	<u>319</u>
<i>Illustrative Problems</i>	<u>319</u>
<i>Review Questions</i>	329
<i>Programming Assignments</i>	330
<b>13. Hash Tables</b>	<b>331</b>
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>	<u>346</u>
<i>Illustrative Problems</i>	<u>347</u>
<i>Review Questions</i>	351
<i>Programming Assignments</i>	352
<b>14. File Organizations</b>	<b>353</b>
14.1 Introduction	353
<u>14.2 Files</u>	<u>354</u>
<u>14.3 Keys</u>	<u>355</u>
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>	<u>369</u>
<i>Review Questions</i>	370
<u>Programming Assignments</u>	<u>371</u>
<b>Part V</b>	
<b>15. Searching</b>	<b>373</b>
15.1 Introduction	373
15.2 Linear Search	373
15.3 Transpose Sequential Search	375
<u>15.4 Interpolation Search</u>	<u>376</u>
15.5 Binary Search	378
15.6 Fibonacci Search	381
15.7 Other Search Techniques	384
<i>Summary</i>	<u>385</u>
<i>Illustrative Problems</i>	<u>386</u>
<i>Review Questions</i>	391
<i>Programming Assignments</i>	393
<b>16. Internal Sorting</b>	<b>394</b>
<u>16.1 Introduction</u>	<u>394</u>

<b>16.2 Bubble Sort</b>	395
<b>16.3 Insertion Sort</b>	396
<b>16.4 Selection Sort</b>	399
<b>16.5 Merge Sort</b>	401
<b>16.6 Shell Sort</b>	405
<b>16.7 Quick Sort</b>	410
<b>16.8 Heap Sort</b>	414
<b>16.9 Radix Sort</b>	422
<i>Summary</i>	426
<i>Illustrative Problems</i>	426
<i>Review Questions</i>	433
<i>Programming Assignments</i>	434
<b>17. External Sorting</b>	<b>435</b>
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
<b>Index</b>	<b>457</b>

# Visual Walkthrough

Contents	
4.2 Stack Operations 40	
4.3 Applications 43	
Summary 49	
Illustrative Problems 49	
Review Questions 54	
Programming Assignments 55	
<b>5. Queues</b>	
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	
Summary 75	
Illustrative Problems 76	
Review Questions 82	
Programming Assignments 82	
<b>Part II</b>	
<b>6. Linked Lists</b>	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	
Summary 112	
Illustrative Problems 113	
Review Questions 119	
Programming Assignments 127	
<b>7. Linked Stacks and Linked Queues</b>	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	
Summary 137	
Illustrative Problems 137	
Review Questions 148	
Programming Assignments 149	
<b>Part III</b>	
<b>8. Trees and Binary Trees</b>	151
8.1 Introduction 151	
8.2 Trees: Definition and Basic Terminologies 151	
8.3 Representation of Trees 153	

Each chapter lists the topics covered.

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

**CHAPTER**

**LINKED LISTS**

**6**

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.

**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 108 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  $A$  with 108 inserted at the appropriate position or making use of  $B$  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)

## 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 modular 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 rebushing, quadratic probing and random probing.

Chapter-end summary for use as quick reference.

386 Data Structure and Algorithms

### Illustrative Problems

**Problem 15.1** For the list (NAMELE, I, KARN, 220E, ..., 2PN, 111B, 100G, 000S, 000A, 000T, 000N, 000R, 000K, 000C, 000L, 000M, 000V) trace through a sequential search for the element 000L.

**External Sorting**

### 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 technique to generate runs. These runs are then merged in passes so 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  $(k+1)$  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 100,000 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)	(8)	(20)	(20)	(8)

(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 terms 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 = 25,000 sectors

Extensive Illustrative Problems throughout.

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

### Review Questions

- A minimal superkey is in fact a \_\_\_\_\_.
  - (a) secondary key
  - (b) primary key
  - (c) non key
  - (d) none of these
- State whether true or false.
  - (i) A cluster index is a sparse index.
  - (ii) A secondary key field with distinct values yields a dense index.
- An index consisting of variable length entries where each index entry would be of the form  $(K, B_1T, B_2T, B_3T, \dots, B_nT)$  where  $B_iT$ '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 initialize three user variables of the queue.  
`CREATE (QUEUE, FRONT, REAR)`
- Check if queue `QUEUE` is empty.  
`Q_ISQUEUE_EMPTY(QUEUE) : Boolean Function`
- Check if queue `QUEUE` is full.  
`Q_ISQUEUE_FULL(QUEUE) Boolean Function`
- Insert item `ITEM` into queue `QUEUE`.  
`Q_ENQUEUE(QUEUE, ITEM)`
- Delete element from queue `QUEUE` and output the element deleted.  
`ITEM Q_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

- 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.
- Implement Algorithm 8.4 to convert an infix expression into its postfix form.
- Write a recursive procedure to count the number of nodes in a binary tree.
- 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.

```

Algorithm 16.7 Procedure for Partition
procedure PARTITION(L, first, last, loc)
    /* L[first:last] is the list to be partitioned. Loc is the
       position where the pivot element finally settles down*/
    left = first;
    right = last-1;
    pivot_elt = L[first]; /* set the pivot element to the first
                           element in list L*/
    while left < right do
        repeat
            left = left+1; /* pivot element moves left to right*/
        until (left) > pivot_elt;
        repeat
            right = right-1; /* pivot element moves right to left*/
        until (right) <= pivot_elt;
        if (left < right) then swap(L[left], L[right]); /*arrows face each
                                                       other*/
    end;
    loc = right;
    swap(L[first], L[right]); /* arrows have crossed each other - exchange
                               pivot element L[first] with L[right]*/
end PARTITION.

```

**Example 16.13** Let us quick sort the list  $L = [5, 1, 26, 15, 76, 34, 15]$ . 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(L, first, last)
    /* L[first:last] is the unordered list of elements to be
       quick sorted. The call to the procedure to sort the
       list L[1:n] would be QUICK_SORT(L, 1, n)*/
    if (first < last) then
        PARTITION(L, first, last, loc); /* partition the list into two
                                         sublists at loc*/
        QUICK_SORT(L, first, loc-1); /* quick sort the sublist
                                       L[first, loc-1]*/
        QUICK_SORT(L, loc+1, last); /* quick sort the sublist
                                       L[loc+1, 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.

Pseudo-code algorithms are given for better comprehension

Extensive examples are given to illustrate theoretical concepts.



**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:  $J_1(1)$   $J_2(1)$   $J_3(0)$

Operations on the JOB queue in the chronological order:

1.  $J_4(2)$  arrives
2.  $J_1(2)$  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
		Opening JOB queue
		Insert $J_4(2)$

(Contd.)

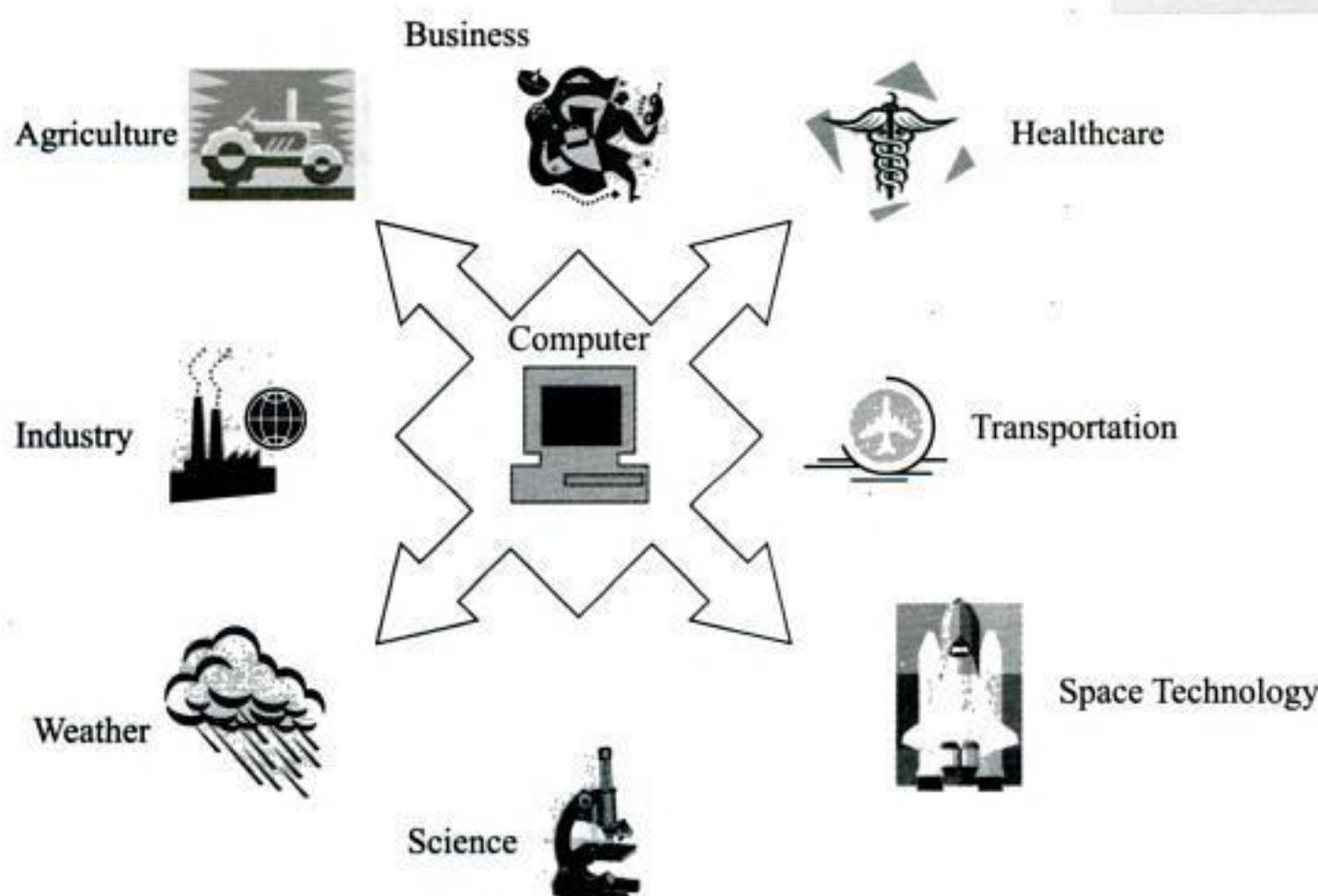
The Online Learning Centre at [www.mhhe.com/pai/dsa](http://www.mhhe.com/pai/dsa) contains C programs for algorithms present in the text, Sample Questions with Solutions and Web Links.



# 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:

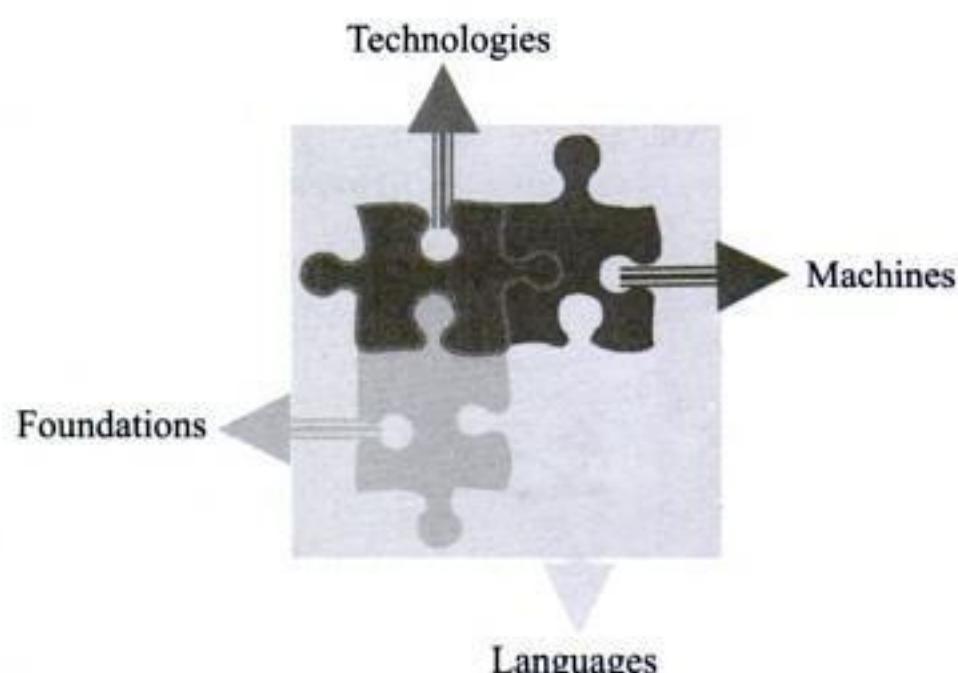
- *Machines* What machines are appropriate or available for the solution of a problem? What is the machine configuration – its processing power, memory capacity etc., that would be required for the efficient execution of the solution?
- *Languages* What is the language or software with which the solution 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:

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

## 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

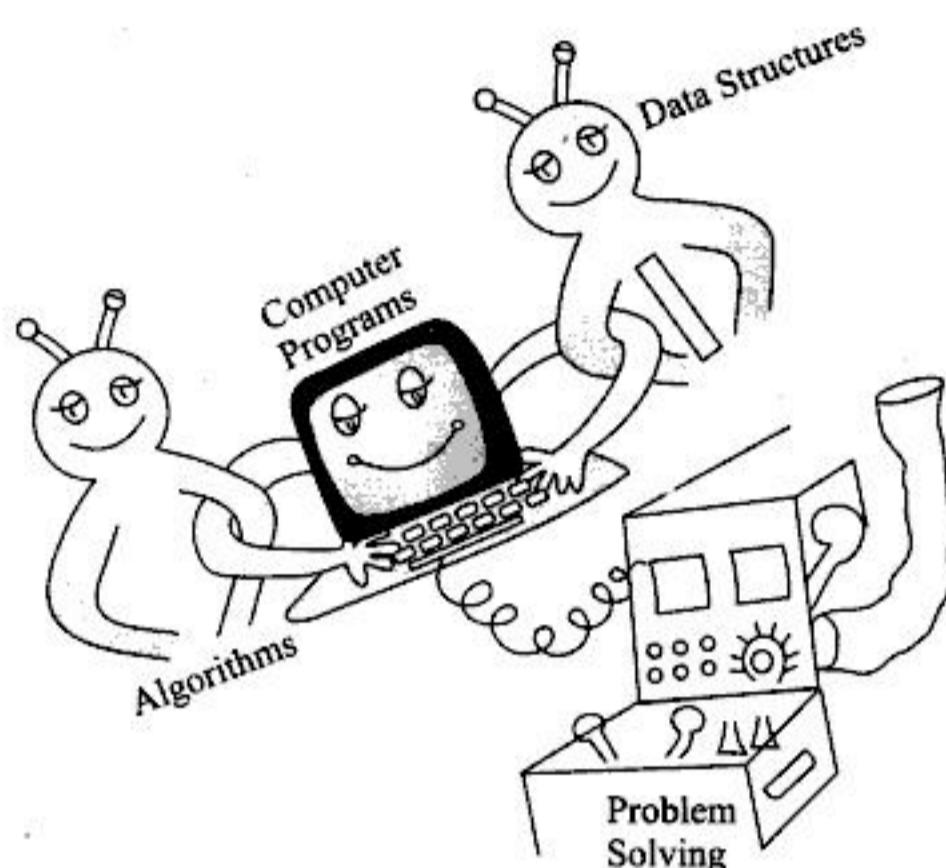
As was detailed in the previous section, the design of an *efficient* algorithm for the solution of the problem calls for the *inclusion of appropriate data structures*. A clear, unambiguous set of instructions following the properties of the algorithm alone does not contribute to the efficiency of the 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)

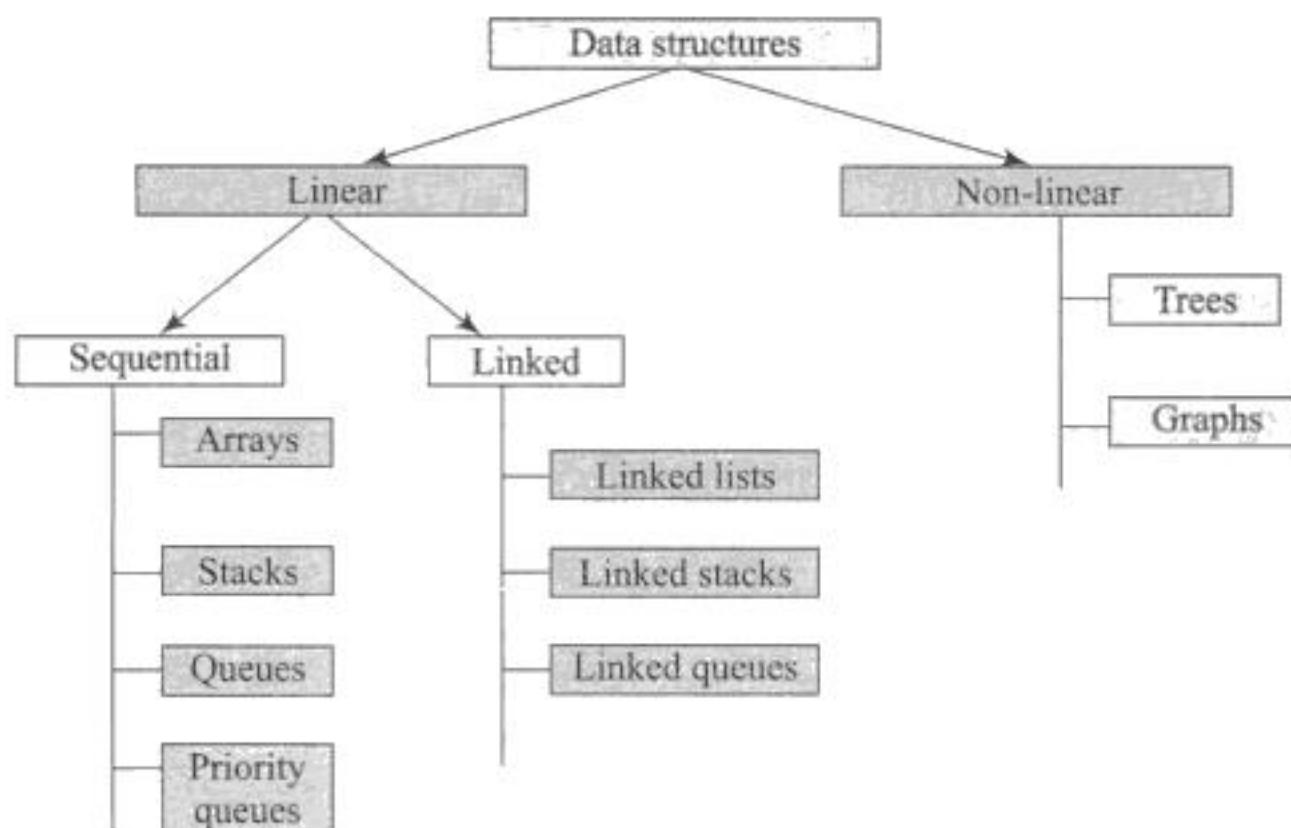
An ADT promotes ***data abstraction*** and focuses on *what* a data structure does rather than *how* it does. It is easier to comprehend a data structure by means of its ADT since it helps a designer to plan on the implementation of the data objects and its supportive operations in any programming language 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 unidimensional 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

Program segment B

Program segment C

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

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

```
...  
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**  $\text{up\_index}$  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
...	
<b>for</b> $k = 1$ <b>to</b> $n$ <b>do</b>	$(n + 1)$
$x = x + 2;$	$n$
<b>end</b>	$n$
...	
Total frequency count	$3n + 1$

**Table 2.3 Total frequency count of program segment C**

Program statements	Frequency count
...	
<b>for</b> $j = 1$ <b>to</b> $n$ <b>do</b>	$(n + 1)$
<b>for</b> $k = 1$ <b>to</b> $n$ <b>do</b>	$\sum_{j=1}^n (n + 1) = (n + 1)n$
$x = x + 2;$	$n^2$
<b>end</b>	$\sum_{j=1}^n n = n^2$
<b>end</b>	$n$
...	
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 $f(n) > 28n$ and $f(n) \leq 37n$ for $n \geq 1$
$16n^2 + 30n - 90$	$n^2$	$f(n) = \Theta(n^2)$
$7.2^n + 30n$	$2^n$	$f(n) = \Theta(2^n)$

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

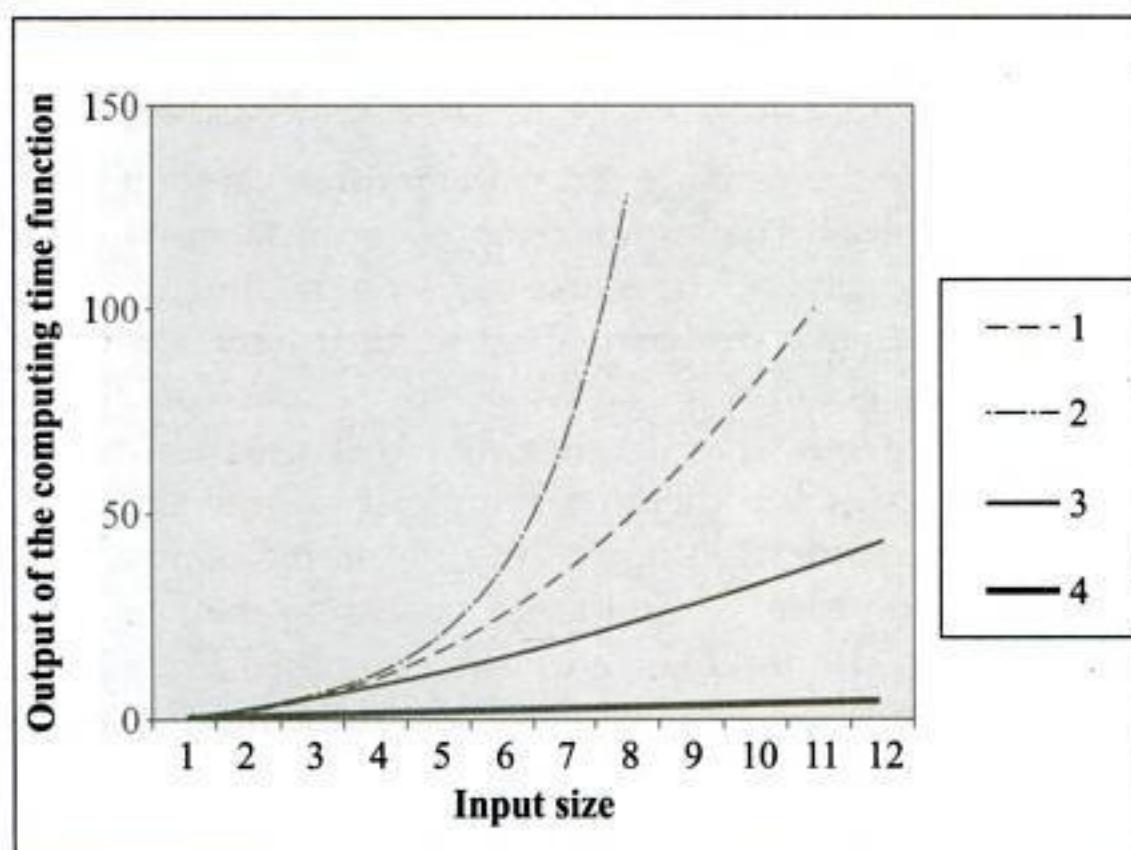
If  $n$  is the size of the input instance, then the number of operations for polynomial algorithms are of the form  $P(n)$  where  $P$  is a polynomial. In terms of  $O$  notation, polynomial algorithms have time complexities of the form  $O(n^k)$ , where  $k$  is a constant.

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	$4 \times 10^{-4}$ sec	$25 \times 10^{-4}$ sec
$n^3$	$10^{-3}$ sec	$8 \times 10^{-3}$ sec	$125 \times 10^{-3}$ sec
$2^n$	$10^{-3}$ sec	1 sec	35 years
$3n$	$6 \times 10^{-2}$ sec	58 mins	$2 \times 10^3$ 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.

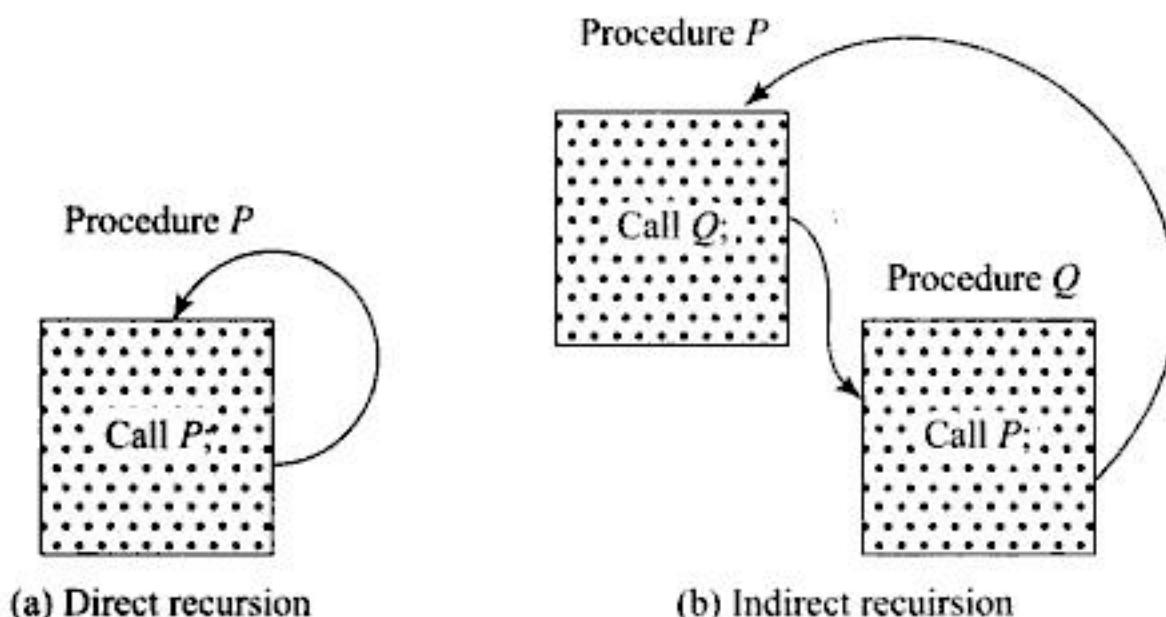
### Recursive procedures

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:

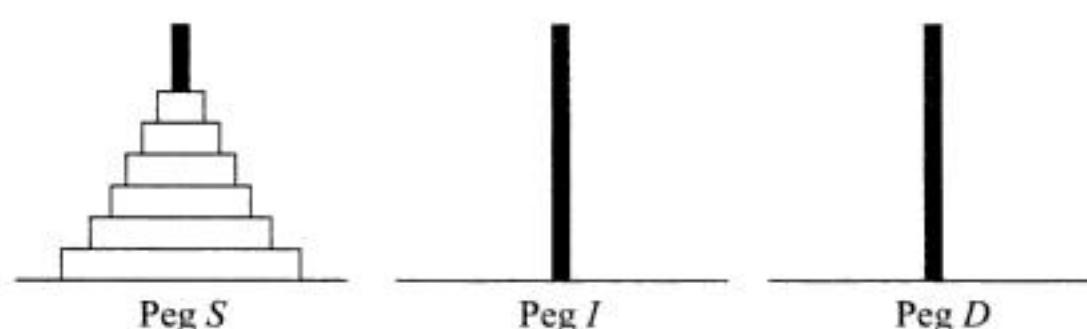
$$S(n) = \begin{cases} 0, & \text{if } n = 1 \text{ (base criterion)} \\ S(n/2) + 1, & \text{if } n > 1 \end{cases}$$

**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  $D$

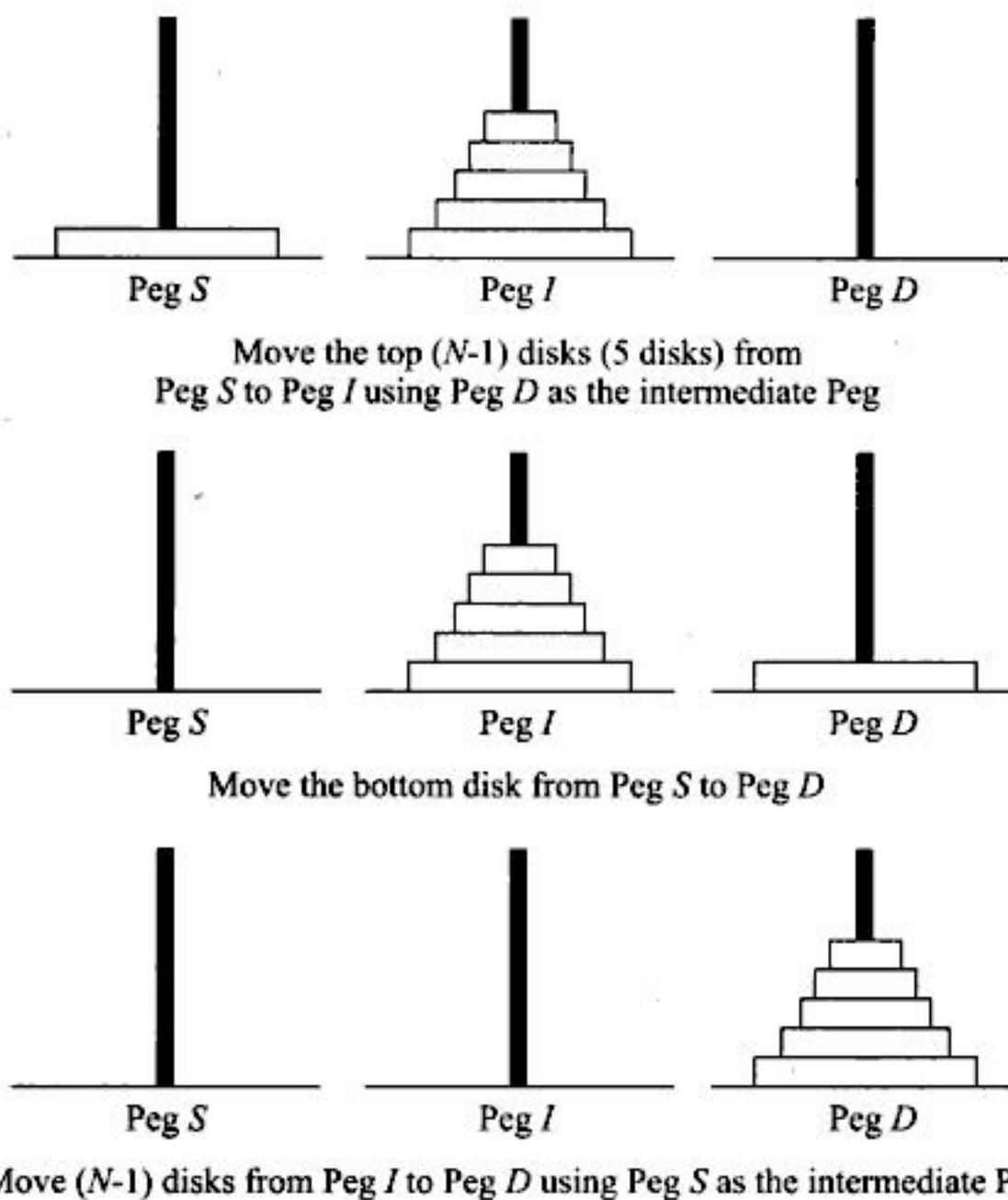


**Fig. 2.3 Tower of Hanoi puzzle (initial configuration)**

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.
- $O$ ,  $\Omega$ ,  $\Theta$ , 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,  $\Omega$  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,

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

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

Hence  $A(n) = O(n^m)$ .

**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:

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

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

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

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

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

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

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

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

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  $n^{\text{th}}$  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			
	$n < 0$	$n = 0$	$n = 1$	$n > 1$
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:

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

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:

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

Generalizing, in the  $i^{\text{th}}$  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

$$\text{(i.e.) } \left| \frac{30n^2 2^n + 6n2^n + 8n^2}{2^n} \right| \leq c$$

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  $n = 2k$ :

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

**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  $i^{\text{th}}$  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

- The frequency count of the statement "for  $k = 3$  to  $(m + 2)$  do" is  
 (a)  $(m + 2)$       (b)  $(m - 1)$       (c)  $(m + 1)$       (d)  $(m + 5)$
- 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)  $f(n) = o(g(n))$
- 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)  $T(n) = O(n^2 + n)$
- 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
- 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)  $O(3^n)$
- How does one measure the efficiency of algorithms?
- Distinguish between best, worst and average case complexities of an algorithm.
- Define  $O$  and  $\Omega$  notations of time complexity.
- Compare and contrast exponential time complexity with polynomial time complexity.
- How are recursive programs analyzed?
- 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

```

- 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

## 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.

$A[1 : 5]$ <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> </table>	6	-4	3	2	11	$B[1 : 3, 1 : 2]$ $\begin{bmatrix} 1 & 2 \\ -6 & 4 \\ 2 & 3 & 2 \\ 3 & 7 & -5 \end{bmatrix}$
6	-4	3	2	11		
(a) One-dimension	(b) Two-dimension					

**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 ( $A, 3, 17$ )	$A$ <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>	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]																			
		RETRIEVE ( $A, 2$ )	-4																				

**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

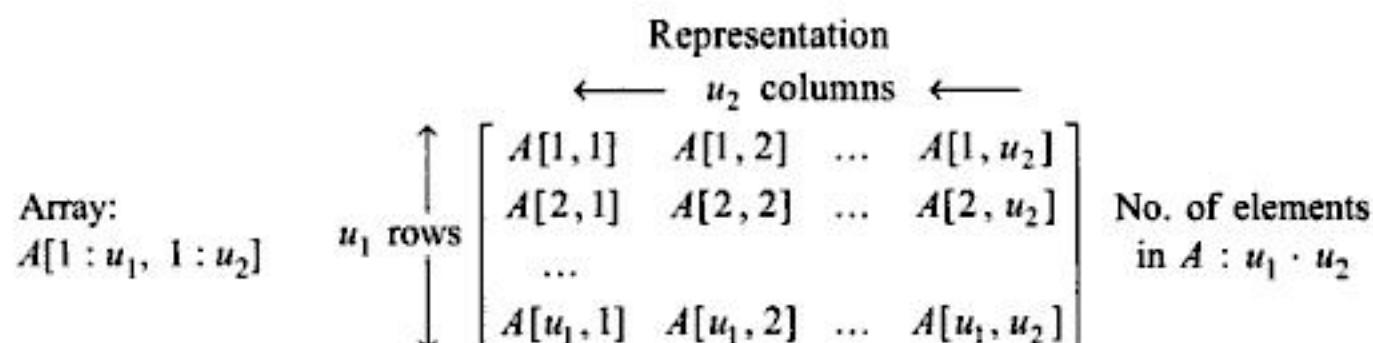
Let  $A[1 : u]$  be a one-dimensional array. The size of the array, as is evident is  $u$  and the elements are  $A[1], A[2], \dots, A[u - 1], A[u]$ . In the case of the array  $A[l : u]$  where  $l$  is the lower bound and  $u$  is the upper bound of the index range, the number of elements is given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

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

### 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

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

### 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  $u_2 \cdot u_3$ . 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

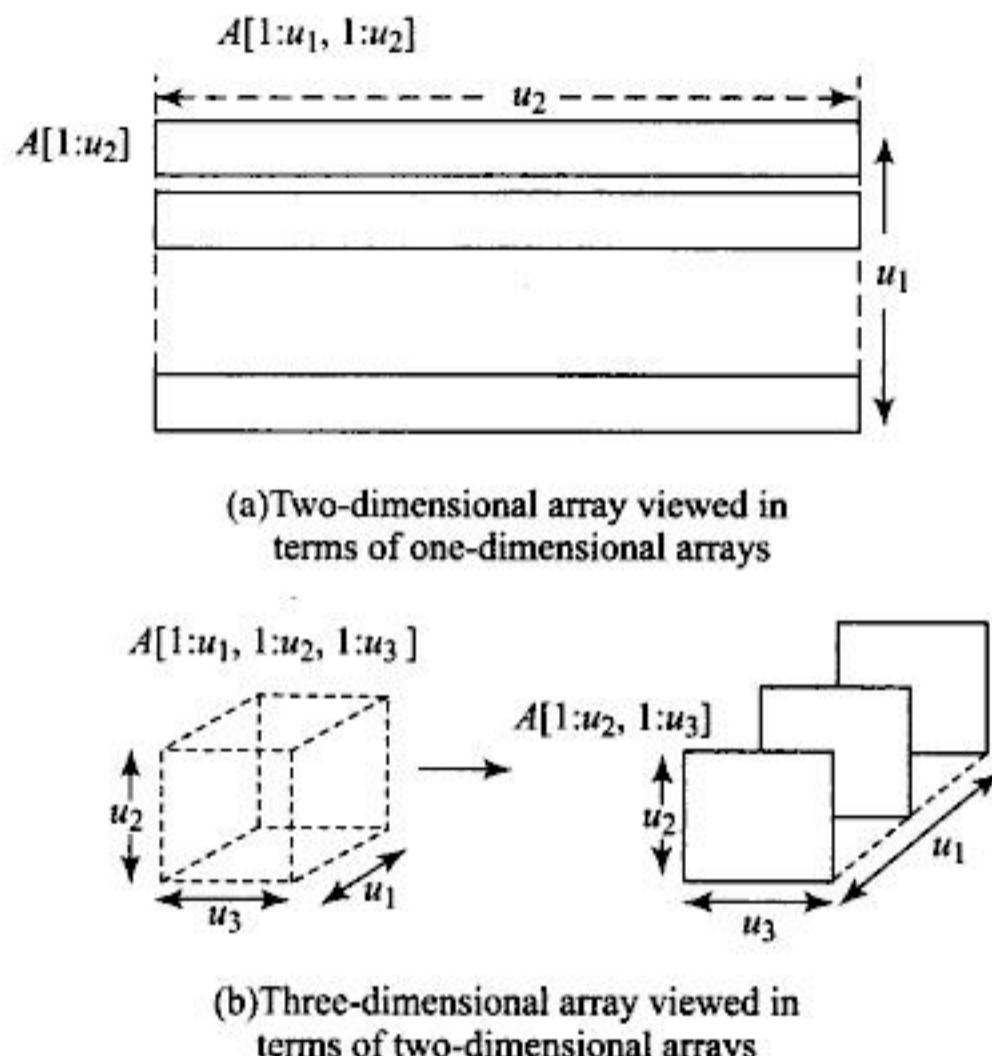
Consider the array  $A(1 : u_1)$  and let  $\alpha$  be the address of the starting memory location referred to as the *base address* of the array. Here as is evident,  $A[1]$  occupies the memory location whose address is  $\alpha$ ,  $A(2)$  occupies  $\alpha + 1$  and so on. In general, the address 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  $\alpha$  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:

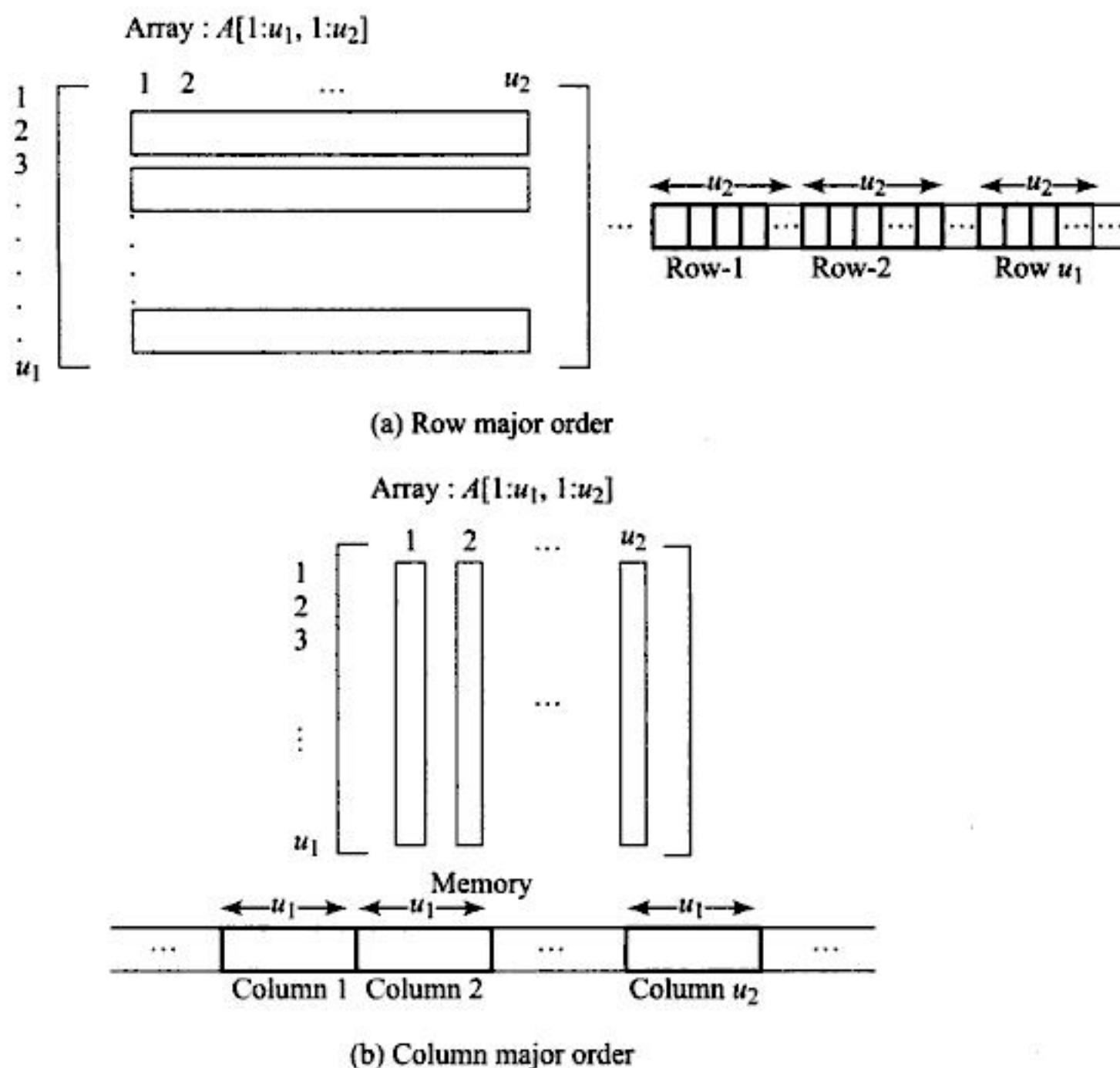
Array	Element	Address
(i) $A[1:17]$	$A[7]$	$\alpha + (7 - 1) = 100 + 6 = 106$
(ii) $A[-2:23]$	$A[16]$	$\alpha + (16 - (-2)) = 100 + 18 = 118$

### 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  $\alpha$ , 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  $i^{\text{th}}$  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



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

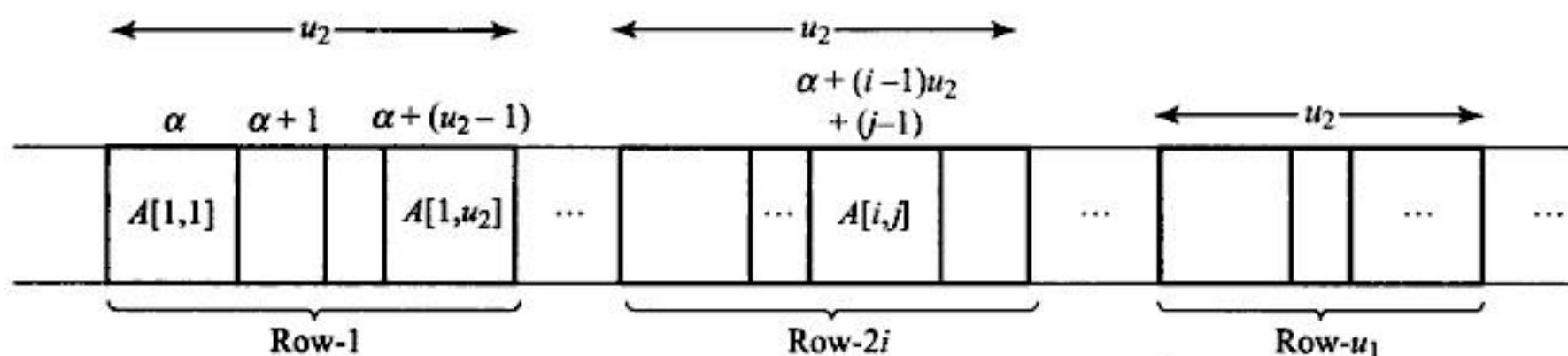
**Fig. 3.7 Representation of one-dimensional arrays in memory**

left behind before one knocks at the first room in the  $i^{\text{th}}$  floor. Since  $\alpha$  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  $j^{\text{th}}$  room on the  $i^{\text{th}}$  floor, one has to leave behind  $(i - 1)u_2$  rooms and reach the  $j^{\text{th}}$  room of the  $i^{\text{th}}$  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:

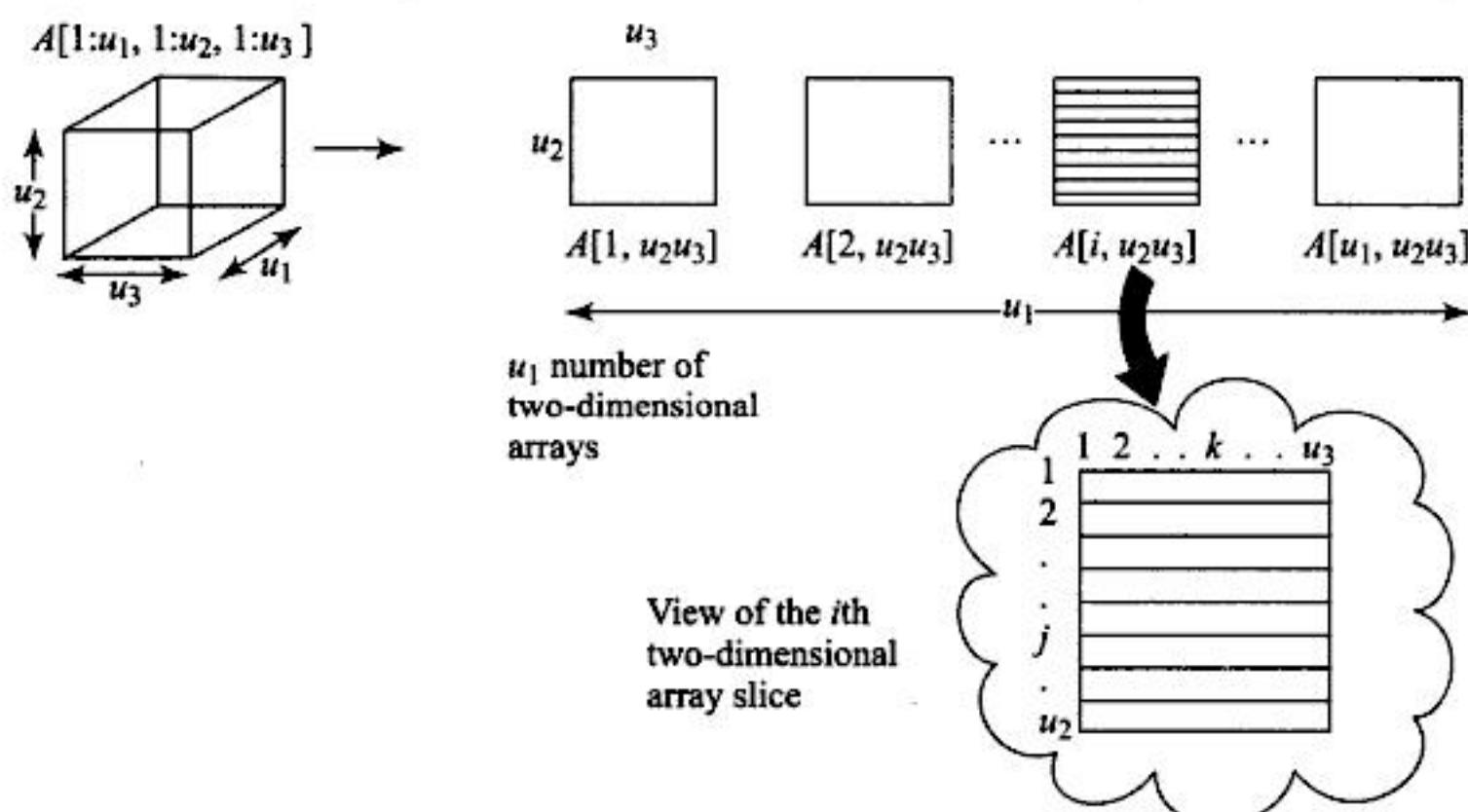
Array	Element	Address
$A[1 : 10, 1 : 5]$	$A[8, 3]$	$220 + (8 - 1)5 + (3 - 1) = 257$
$A[-2 : 4, -6 : 10]$	$A[3, -5]$	$220 + (3 - (-2))(10 - (-6) + 1) + (-5 - (-6)) = 306$

### 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  $i^{\text{th}}$  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  $i^{\text{th}}$  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  $j^{\text{th}}$  floor of the  $i^{\text{th}}$  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  $m \times n$  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  $1000 \times 1000$  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.,  $(i, j, \text{value})$  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  $i^{\text{th}}$  element of the array ARRAY  
 $\text{ARRAY}[i] = \text{VAL}$
- Retrieve the value in the  $i^{\text{th}}$  element of array ARRAY as VAL  
 $\text{VAL} = \text{ARRAY}[i]$



## 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.

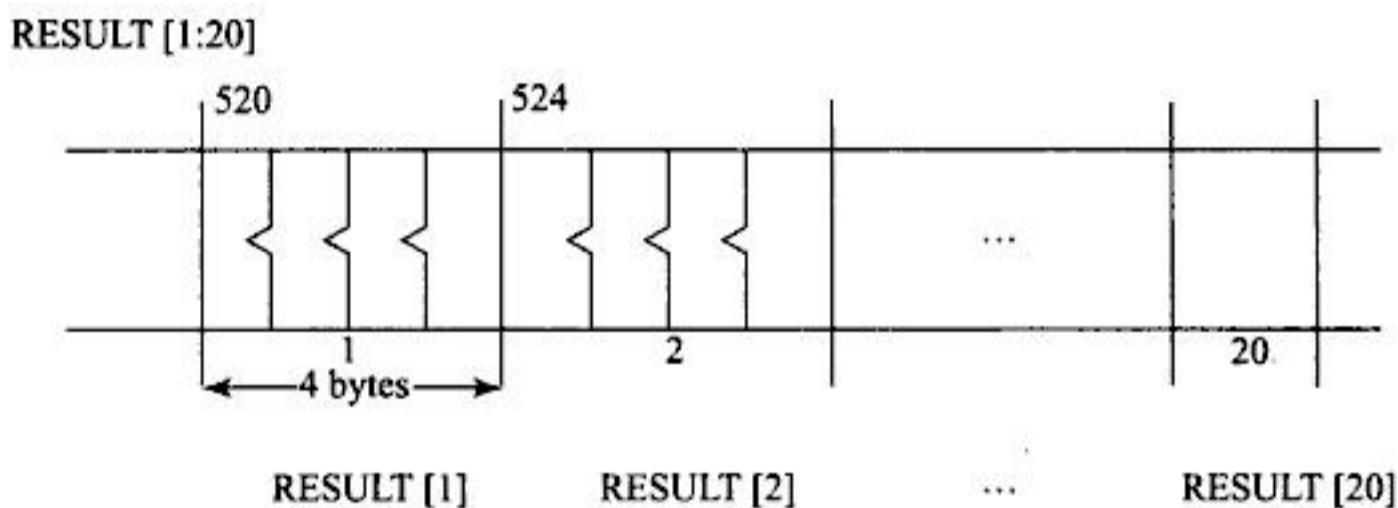
- Two concepts viz., sparse matrices and ordered lists, of use to computer science have been briefly described as applications of arrays.

## Illustrative Problems

**Problem 3.1** The following details are available about an array RESULT. Find the address of  $\text{RESULT}[17]$ .

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

*Solution:* Since  $\text{RESULT}[1:20]$  is a one-dimensioned array, the address for  $\text{RESULT}[17]$  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:*

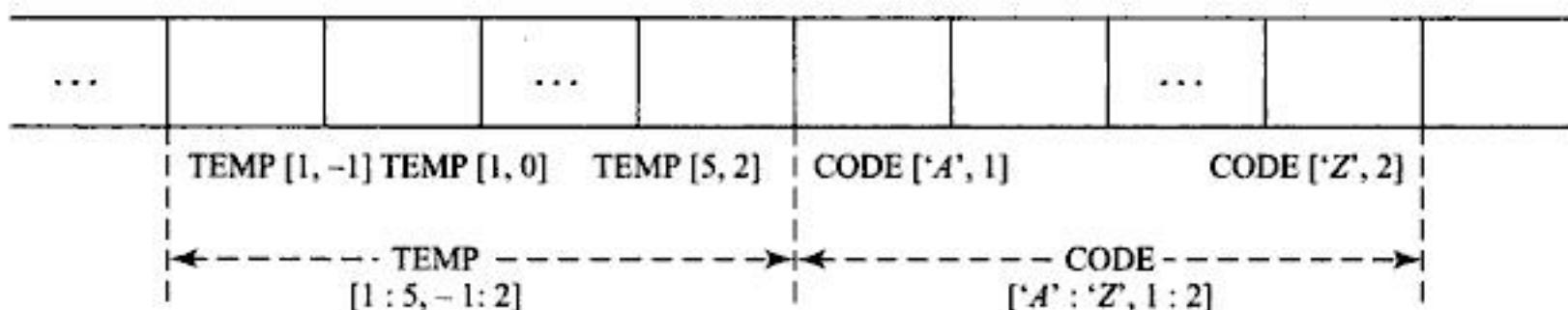
- (i) The number of elements in  $B$  is  $16 \times 6 = 96$
- (ii) The space occupied by  $B$  is  $96 \times 4 = 384$  bytes
- (iii) The address of  $B[7, 2]$  is given by  

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

**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

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

- (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
5	4	0	0	-3	0	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

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.

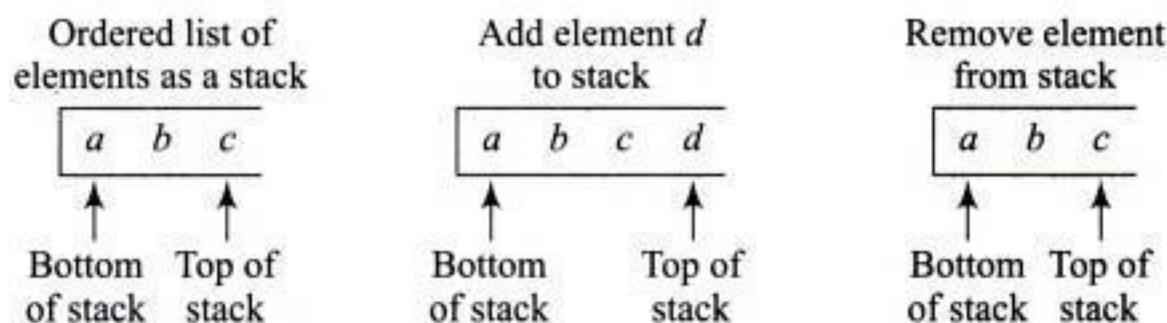
- 4.1 *Introduction*
- 4.2 *Stack Operations*
- 4.3 *Applications*

## 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.

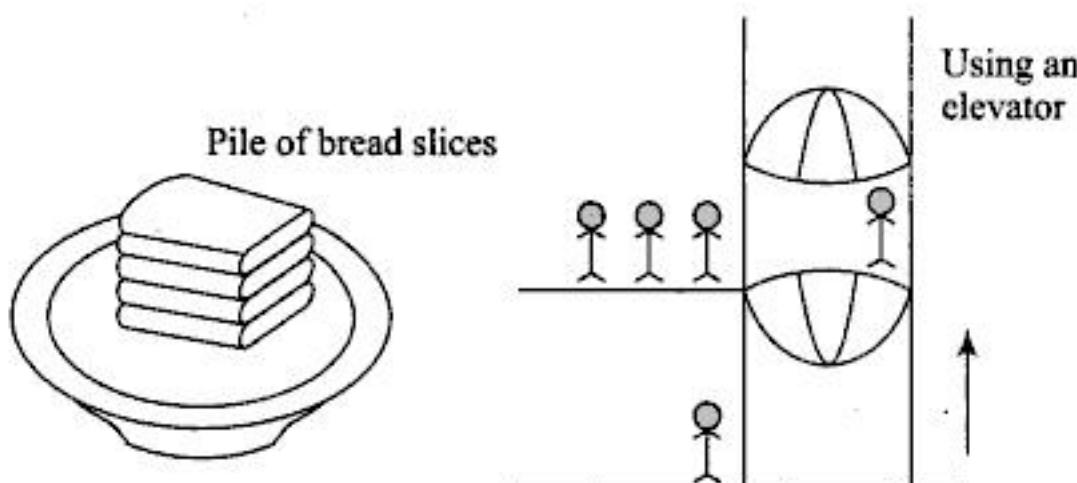


**Fig. 4.1** Stack and its functionality

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

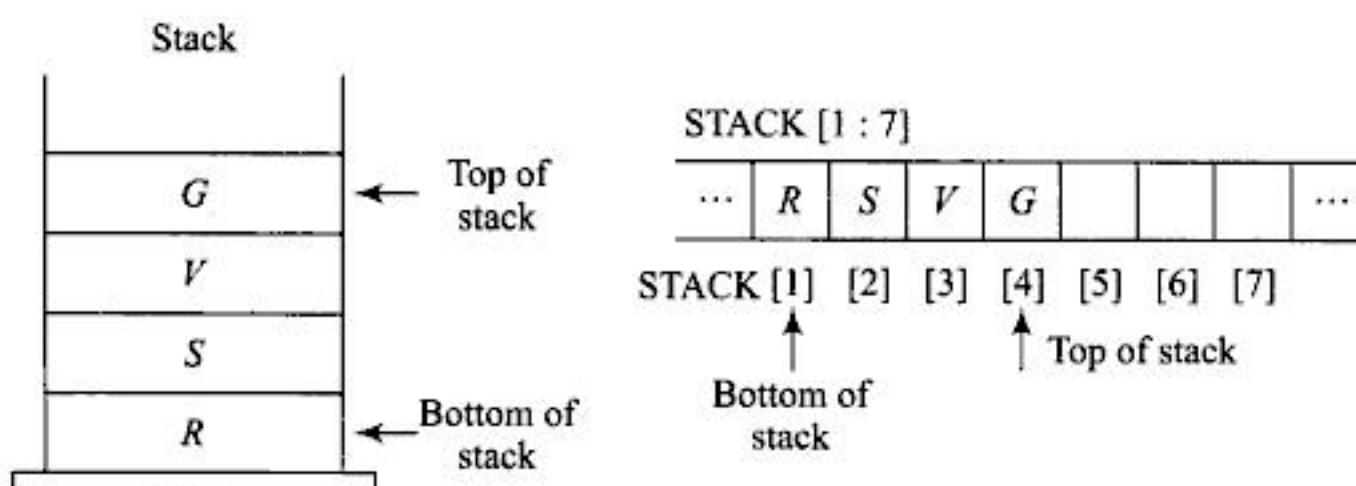
- (i) Insertion or addition of elements known as *Push*
- (ii) Deletion or removal of elements known as *Pop*

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  $\text{STACK\_FULL}$  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  $\text{STACK\_EMPTY}$  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  $\text{top}$  be a variable recording the current top of stack position.  $\text{top}$  is initialized to 0.  $\text{item}$  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,  $\text{item}$  is used as an output variable which stores a copy of the element removed.

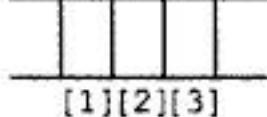
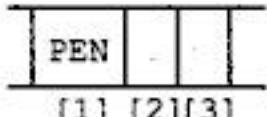
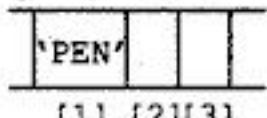
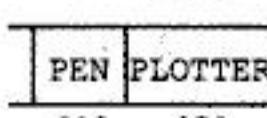
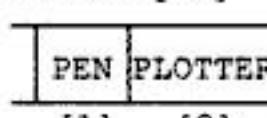
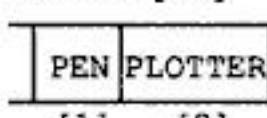
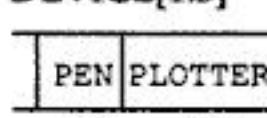
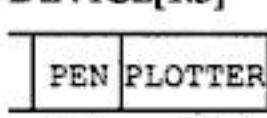
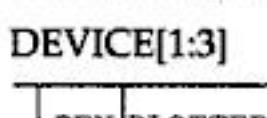
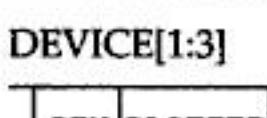
#### Algorithm 4.2: Implementation of pop operation on a stack

```
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  $\text{DEVICE}$  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, '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.)

<i>n</i>	5
<i>x</i>	
<i>y</i>	

(a) Invocation of FACTORIAL (5)

<i>n</i>	5
<i>x</i>	5
<i>y</i>	↗

(b) During the execution of FACTORIAL (5)  
↗ indicates call to FACTORIAL (4) (Step 3)

<i>n</i>	5	4
<i>x</i>	4	3
<i>y</i>	↗	↗

(c) Invoking FACTORIAL (3) during the execution of FACTORIAL (4)

<i>n</i>	5	4	3	2	1
<i>x</i>	4	3	2	1	0
<i>y</i>	↗	↗	↗	↗	↗

(d) Stack contents after subsequent calls and during the execution of FACTORIAL (1).  
↗ indication call to FACTORIAL (0)

<i>n</i>	5	4	3	2	1	0
<i>x</i>	4	3	2	1	0	
<i>y</i>	↗	↗	↗	↗	↗	

(e) Invocation of FACTORIAL (0)

<i>n</i>	5	4	3	2	1
<i>x</i>	4	3	2	1	0
<i>y</i>	↗	↗	↗	↗	1

(f) FACTORIAL (0) has normal termination. Obtains the value of  $0! = 1$  and returns to its point of invocation. Note *y* of FACTORIAL (1) receiving the computed value

<i>n</i>	5	4	3	2
<i>x</i>	4	3	2	1
<i>y</i>	↗	↗	↗	1

(g) FACTORIAL (1) termination computes  $1! = 1$  and returns it to the point of invocation in FACTORIAL (2). Note *y* of FACTORIAL (2) receiving the value

<i>n</i>	5
<i>x</i>	4
<i>y</i>	24

(h) Stack contents, after all other calls except FACTORIAL (5) have been normally terminated

**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  $\text{FACTORYL} = n * y$  (i.e.)  $\text{FACTORYL} = 1 * 1 = 1$ . 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  $\text{FACTORYL} = n * y = 2 * 1 = 2$ , 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  $y = \text{FACTORIAL}(4) = 24$  and proceeds to obtain  $\text{FACTORIAL} = n * y = 5 * 24 = 120$  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> <operator> <operand>* 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> <operator>* 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) $a + b * c - d$ ①	$b * c$	①: $bc *$
(ii) $a + \underline{\text{①}} - d$ ②	$a + \text{①}$	$a \text{ ①} +$ (i.e) ② : $abc * +$
(iii) $\underline{\text{②}} - d$ ③	$\text{②} - d$	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) $(\underline{①} - f * h) \uparrow d$ ②	$f * h$	② : $* fh$
(iii) $(\underline{\text{①} - \text{②}}) \uparrow d$ ③	$(\text{①} - \text{②})$	③ : $- \text{①} \text{②}$ (i.e) $- * ab * fh$
(iv) $\underline{\text{③} \uparrow d}$ ④	$\text{③} \uparrow d$	④ : $\uparrow \text{③} d$ (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. <b>while</b> not $\text{EMPTYSTACK}(S)$ <b>do</b> |
| 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. <b>end</b>  |

**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.



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

**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

```



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



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



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



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



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



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



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

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  $\text{FRONT}$  increments itself in steps of one with every deletion and begins moving towards  $\text{REAR}$ . During the final deletion which renders the queue empty,  $\text{FRONT}$  coincides with  $\text{REAR}$  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,  $\text{REAR}$  moves away from  $\text{FRONT}$  and with every deletion  $\text{FRONT}$  moves towards  $\text{REAR}$ . 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  $\text{BIRDS}[1:3]$  be a linear queue data structure. The working of Algorithms 5.1 and 5.2 demonstrated on the insertions and deletions performed on  $\text{BIRDS}$  is illustrated in Table 5.1.

**Table 5.1** Insert/delete operations on the queue  $\text{BIRDS}[1:3]$

Operation	Queue before operation	Algorithm	Queue after operation	Remarks
1. Insert 'DOVE' into $\text{BIRDS}[1:3]$	<p><math>\text{BIRDS}[1:3]</math></p> <p>[1] [2] [3]</p> <p>FRONT: <input type="button" value="0"/> REAR: <input type="button" value="0"/></p>	INSERTQ ( $\text{BIRDS}, 3$ , 'DOVE', 0)	<p><math>\text{BIRDS}[1:3]</math></p> <p>[1] [2] [3]</p> <p>FRONT: <input type="button" value="0"/> REAR: <input type="button" value="1"/></p>	Insert 'DOVE' successful
2. Insert 'PEACOCK' into $\text{BIRDS}[1:3]$	<p><math>\text{BIRDS}[1:3]</math></p> <p>[1] [2] [3]</p> <p>FRONT: <input type="button" value="0"/> REAR: <input type="button" value="1"/></p>	INSERTQ ( $\text{BIRDS}, 3$ , 'PEACOCK', 1)	<p><math>\text{BIRDS}[1:3]</math></p> <p>[1] [2] [3]</p> <p>FRONT: <input type="button" value="0"/> REAR: <input type="button" value="2"/></p>	Insert 'PEACOCK' successful

(Contd.)



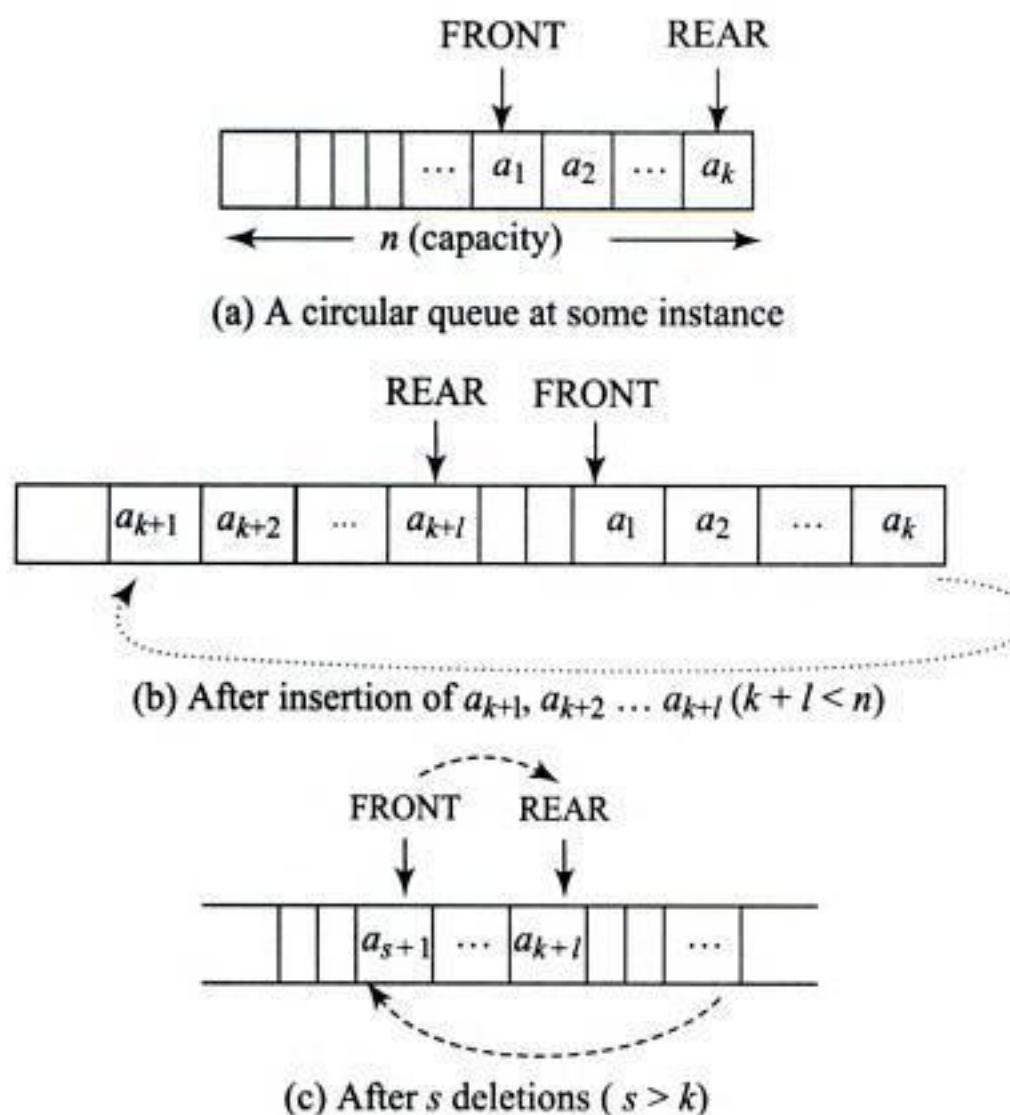
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



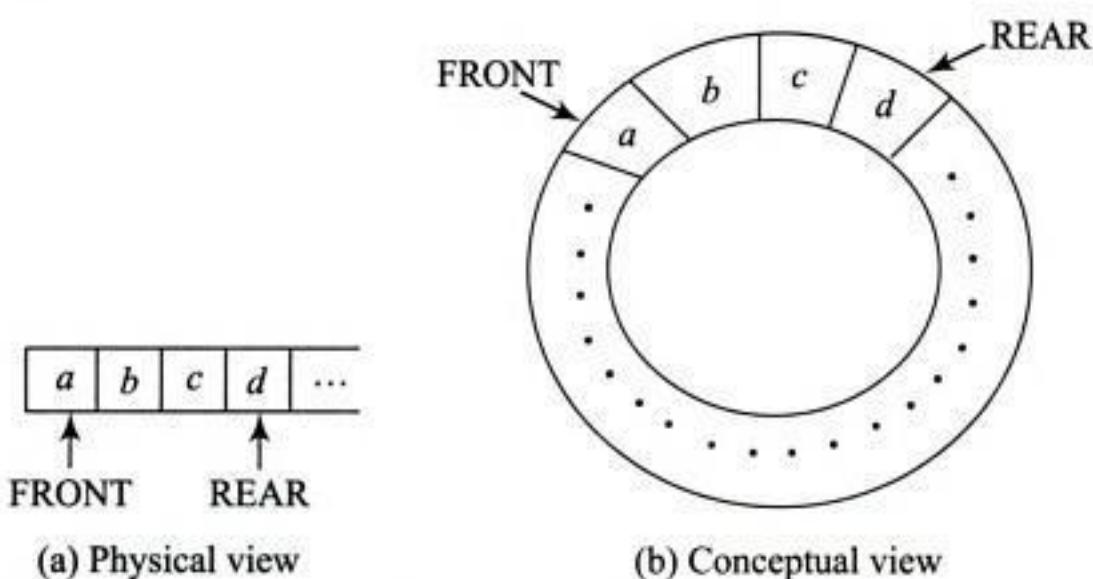
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



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



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



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



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



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

**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><math>J_1(1) \quad J_2(1) \quad J_3(0)</math></p>	<p>Opening JOB queue</p>
<p>1. <math>J_4(2)</math> arrives</p> <p>Machine service</p>	<p>1. <math>J_4(2)</math> arrives</p> <p><math>J_4(2) \quad J_1(1) \quad J_2(1) \quad J_3(0)</math></p>	<p>Insert <math>J_4(2)</math></p>

(Contd.)



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



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



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

<i>DEQ:</i>	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

<i>DEQ:</i>	LEFT: 2	RIGHT: 6
[1] [2] [3] [4] [5] [6] _____ X R T S Y		

- (ii) Delete twice from the right end

<i>DEQ:</i>	LEFT: 2	RIGHT: 4
[1] [2] [3] [4] [5] [6] _____ X R T		

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

<i>DEQ:</i>	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

<i>DEQ:</i>	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



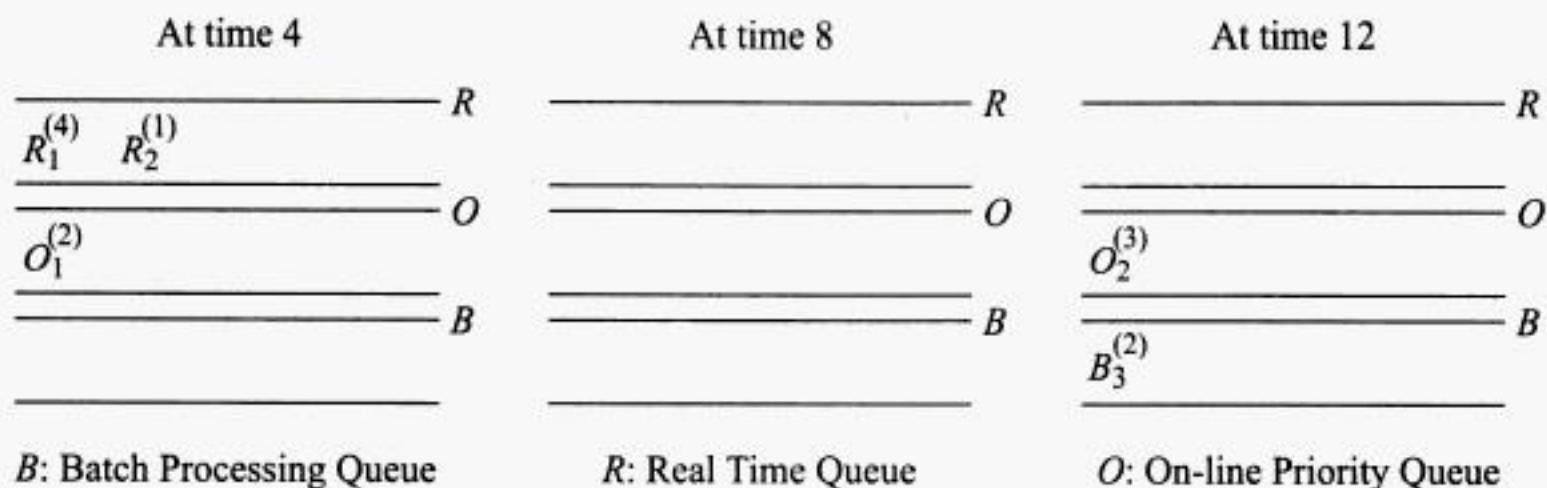
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



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



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



**Fig. 5.13** *Snapshots of the priority queue at time 4, 8 and 12*

### ADT for Queues

#### Data objects

A finite set of elements of the same type

#### Operations

- Create an empty queue and initialize front and rear variables of the queue  
CREATE ( QUEUE, FRONT, REAR)
- Check if queue QUEUE is empty  
CHK\_QUEUE\_EMPTY (QUEUE ) (Boolean function)
- Check if queue QUEUE is full  
CHK\_QUEUE\_FULL (QUEUE) (Boolean function)
- Insert ITEM into queue QUEUE  
ENQUEUE (QUEUE, ITEM)
- Delete element from queue QUEUE and output the element deleted in ITEM  
DEQUEUE (QUEUE , ITEM)



## Summary

- A queue data structure is a linear list in which all insertions are made at the rear end of the list and deletions are made at the front end of the list.
- A queue follows the principle of FIFO or FCFS and is commonly implemented using arrays. It therefore calls for the testing of QUEUE\_ FULL/QUEUE\_ EMPTY conditions during insert/delete operations respectively.
- A linear queue suffers from the draw back of QUEUE\_ FULL condition invocation even when the queue is not physically full to its capacity. This limitation is over come to an extent in a circular queue.
- Priority queue is a queue structure in which elements are inserted or deleted from a queue based on some property known as priority.
- A deque is a double ended queue with insertions and deletions done at either ends or may be appropriately restricted at one of the ends.
- The application of queues and priority queues has been demonstrated on the problem of job scheduling in time-sharing system environments.



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



You have either reached a page that is unavailable for viewing or reached your 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.)

7-10	<u>21</u>		11	21	32
4-6	<u>21</u>	<u>32<sup>(2)</sup></u>	11	21	32
7-10	<u>21</u>	<u>32<sup>(2)</sup></u>	21	32	53
4-6	<u>21</u>	<u>32<sup>(2)</sup> 53<sup>(2)</sup></u>	21	32	53
7-10	<u>21</u>	<u>32<sup>(2)</sup> 53<sup>(2)</sup></u>	32	53	85
4-6	<u>21</u>	<u>32<sup>(2)</sup> 53<sup>(2)</sup> 85<sup>(1)</sup></u>	32	53	85
7-10	<u>21</u>	<u>32<sup>(2)</sup> 53<sup>(2)</sup> 85<sup>(1)</sup></u>	53	85	138
11-14		<u>32<sup>(2)</sup> 53<sup>(2)</sup> 85<sup>(1)</sup></u>	53	85	21
			<b>Output: 21</b>		
15-18			53	85	32
			53	85	53
			53	85	85
			<b>Output: 32 53 85</b>		

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]

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



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



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



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

- (i) Implement the system using an appropriate queue data structure, simulating a random arrival and departure of customers after service completion.
- (ii) If a customer arrives to operate his/her savings account at the post office, then he/she is attended to first by permitting him/her to join a special queue. In such a case the postal worker attends to them immediately before resuming his/her normal service. Modify the system to implement this addition in service.
2. Write a program to maintain a list of items as a circular queue which is implemented using an array. Simulate insertions and deletions to the queue and display a graphical representation of the queue after every operation.
3. Let PQUE be a priority queue data structure and  $a_1^{(p_1)}, a_2^{(p_2)}, a_n^{(p_n)}$  be  $n$  elements with priorities  $p_i$ , ( $0 \leq p_i \leq m - 1$ )
  - (i) Implement PQUE using multiple circular queues one for each priority number.
  - (ii) Implement PQUE as a two dimensional array ARR\_PQUE[1:m, 1:d] where  $m$  is the number of priority values and  $d$  is the maximum number of data items with a given priority.
  - (iii) Execute insertions and deletions presented in a random sequence.
4. A deque DQUE is to be implemented using a circular one dimensional array of size  $N$ . Execute procedures to
  - (i) Insert and delete elements from DQUE at either ends
  - (ii) Implement DQUE as an output restricted deque
  - (iii) Implement DQUE as an input restricted deque
  - (iv) For the procedures, what are the conditions used for testing DQUE\_FULL and DQUE\_EMPTY?
5. Execute a general data structure which is a deque supporting insertions and deletions at both ends but depending on the choice input by the user, functions as a stack or a 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.

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.

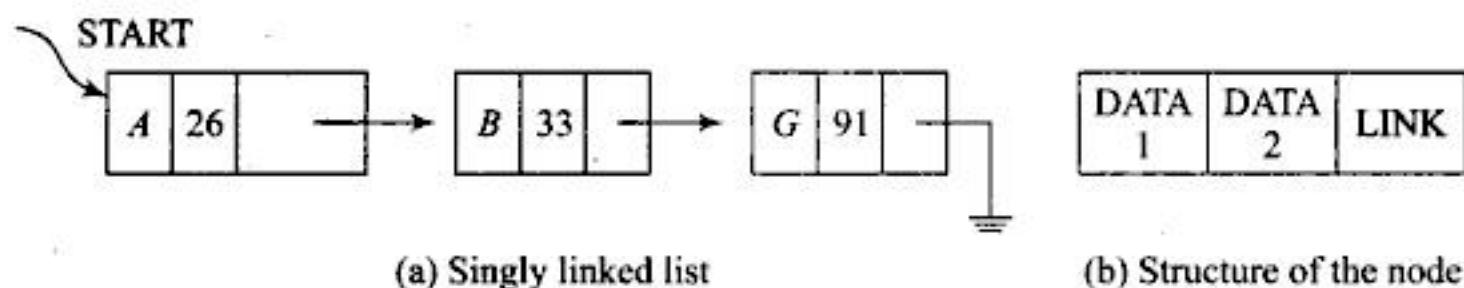
## Singly Linked Lists

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 (      ) 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



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

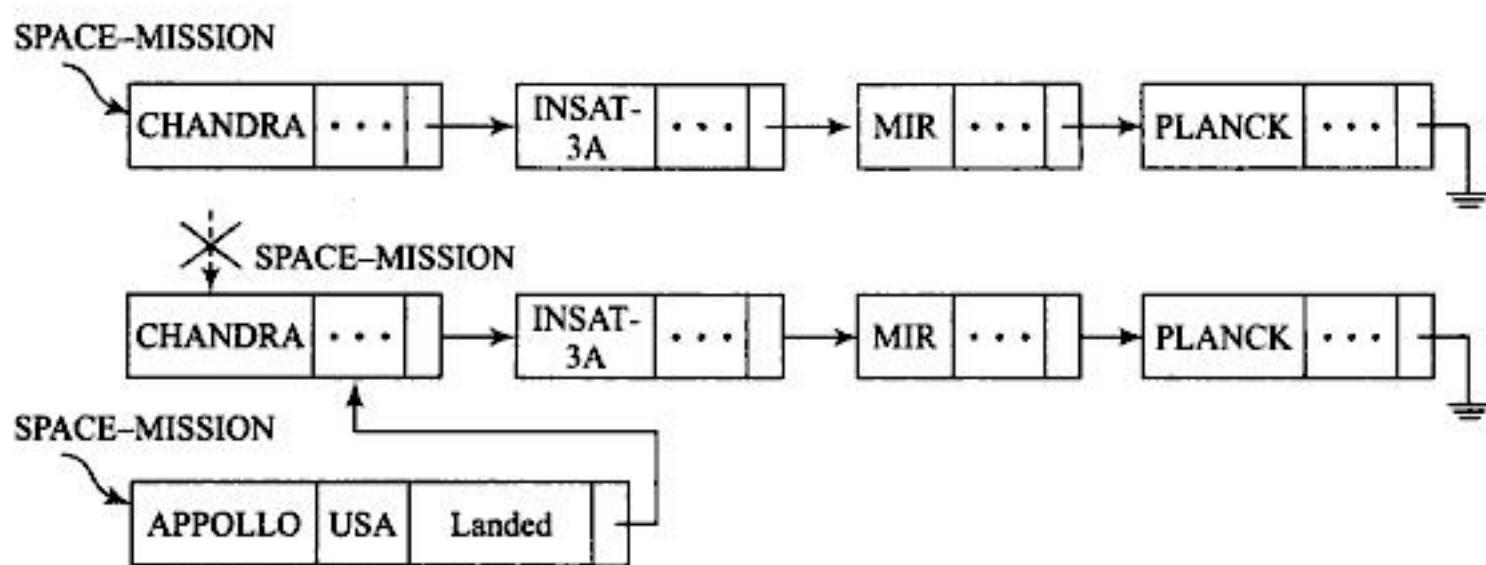


You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



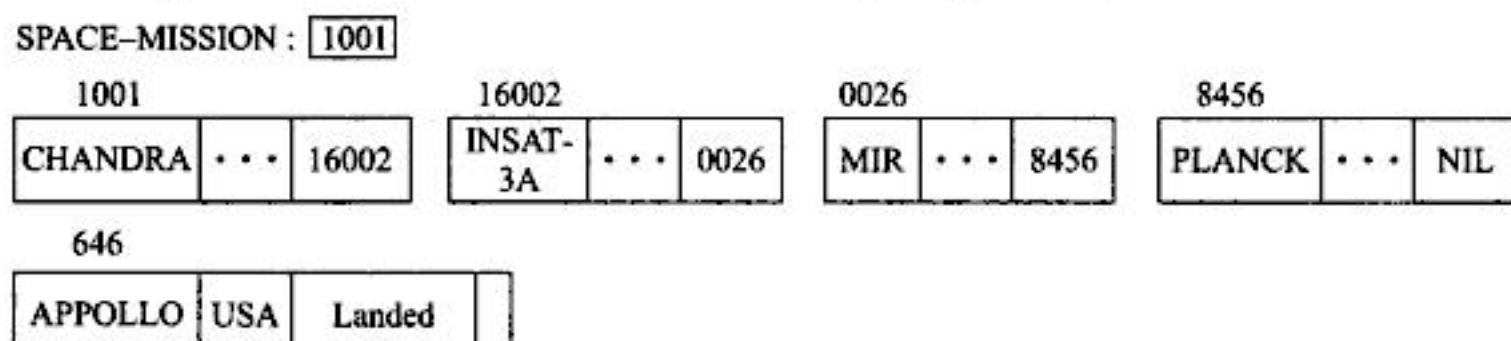
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

SPACE-MISSION  
list before  
insertion of  
APOLLO

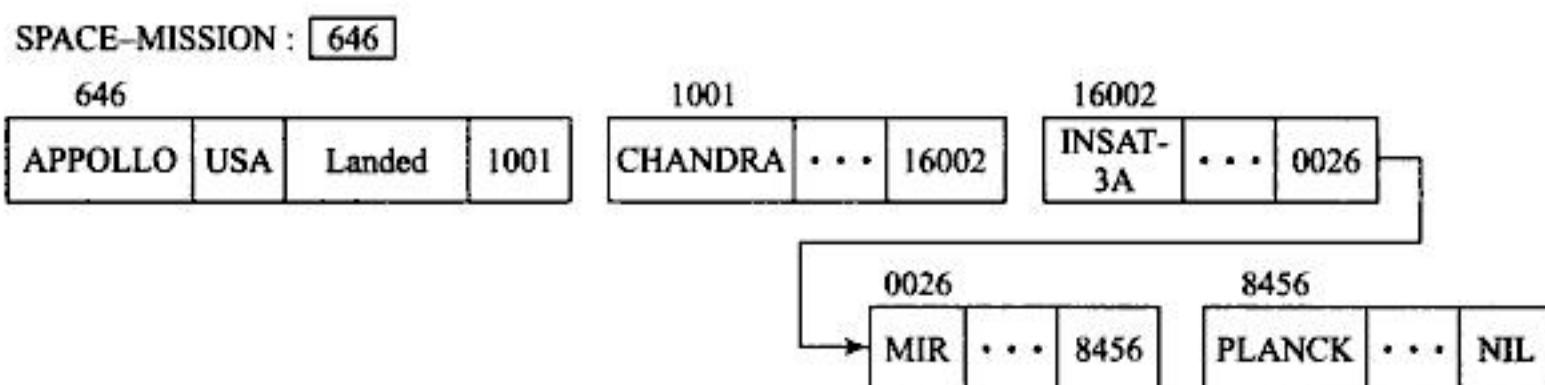


(a) Insert APOLLO in list SPACE-MISSION—logical representation

SPACE-MISSION  
list before  
insertion of  
APOLLO

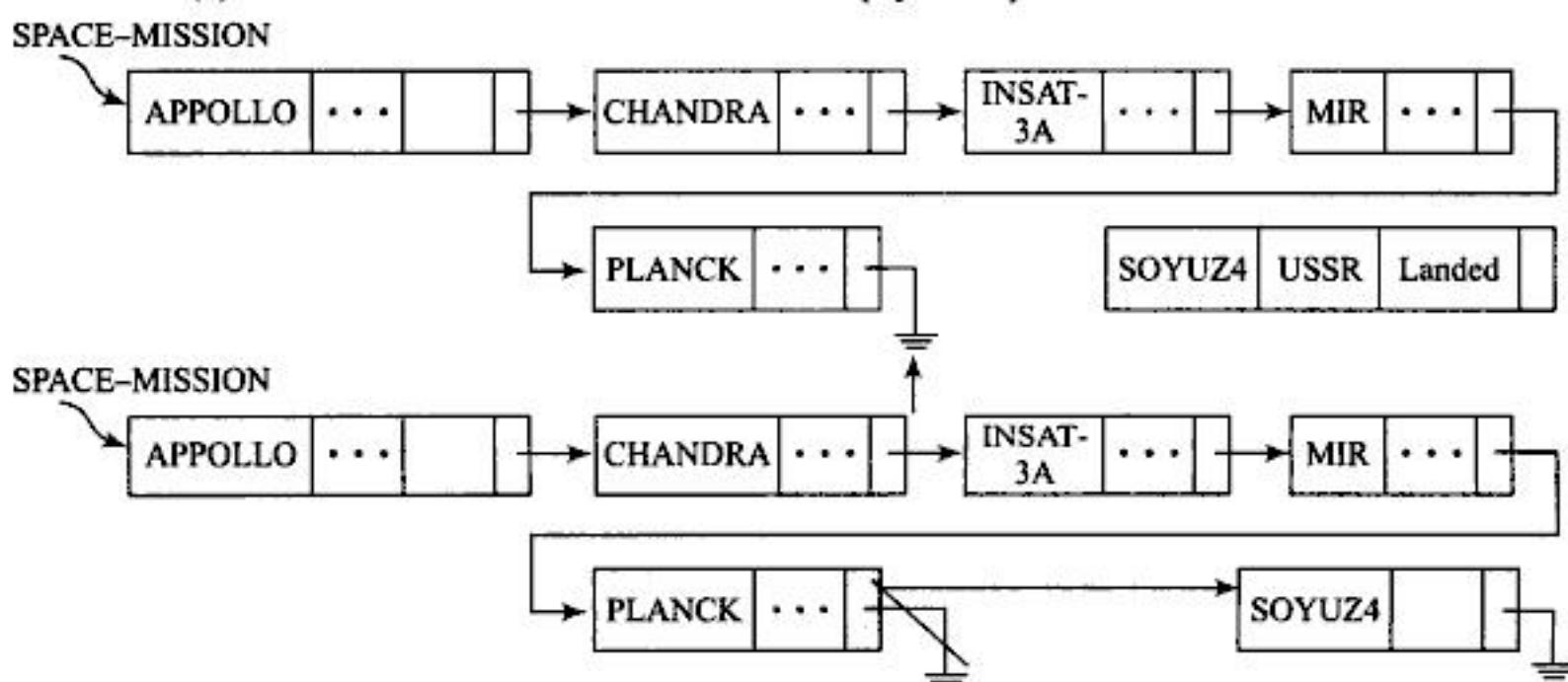


SPACE-MISSION  
list after  
insertion of  
APOLLO



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

SPACE-MISSION  
list before  
insertion of  
SOYUZ4



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



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



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



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

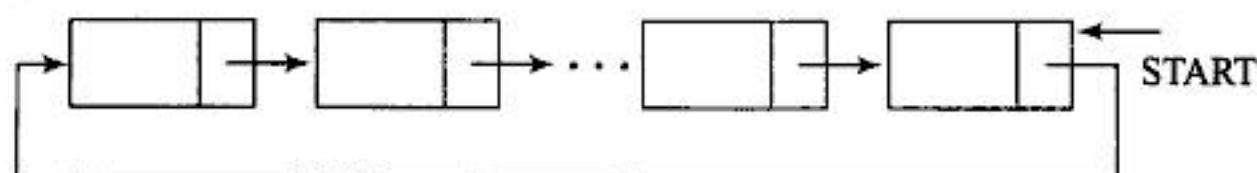


Fig. 6.10 Representation of a circular list

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  $\text{HEAD}$  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.

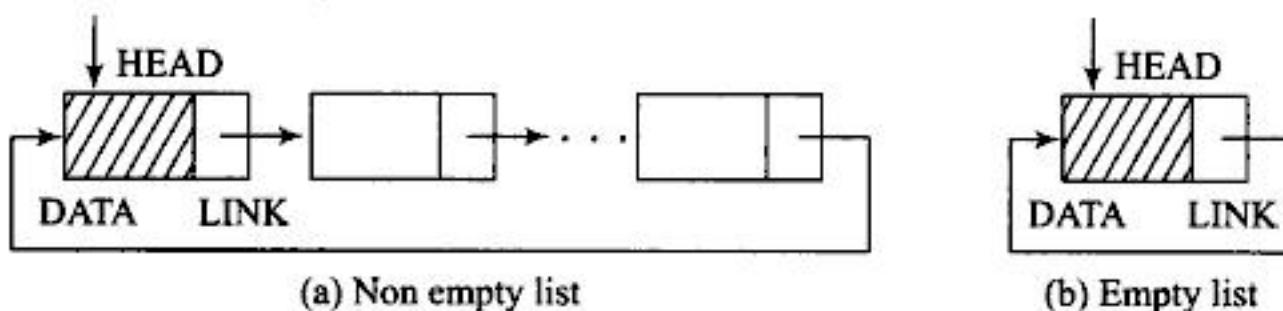


Fig. 6.11 A headed circularly linked list

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  $P$ .



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



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



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

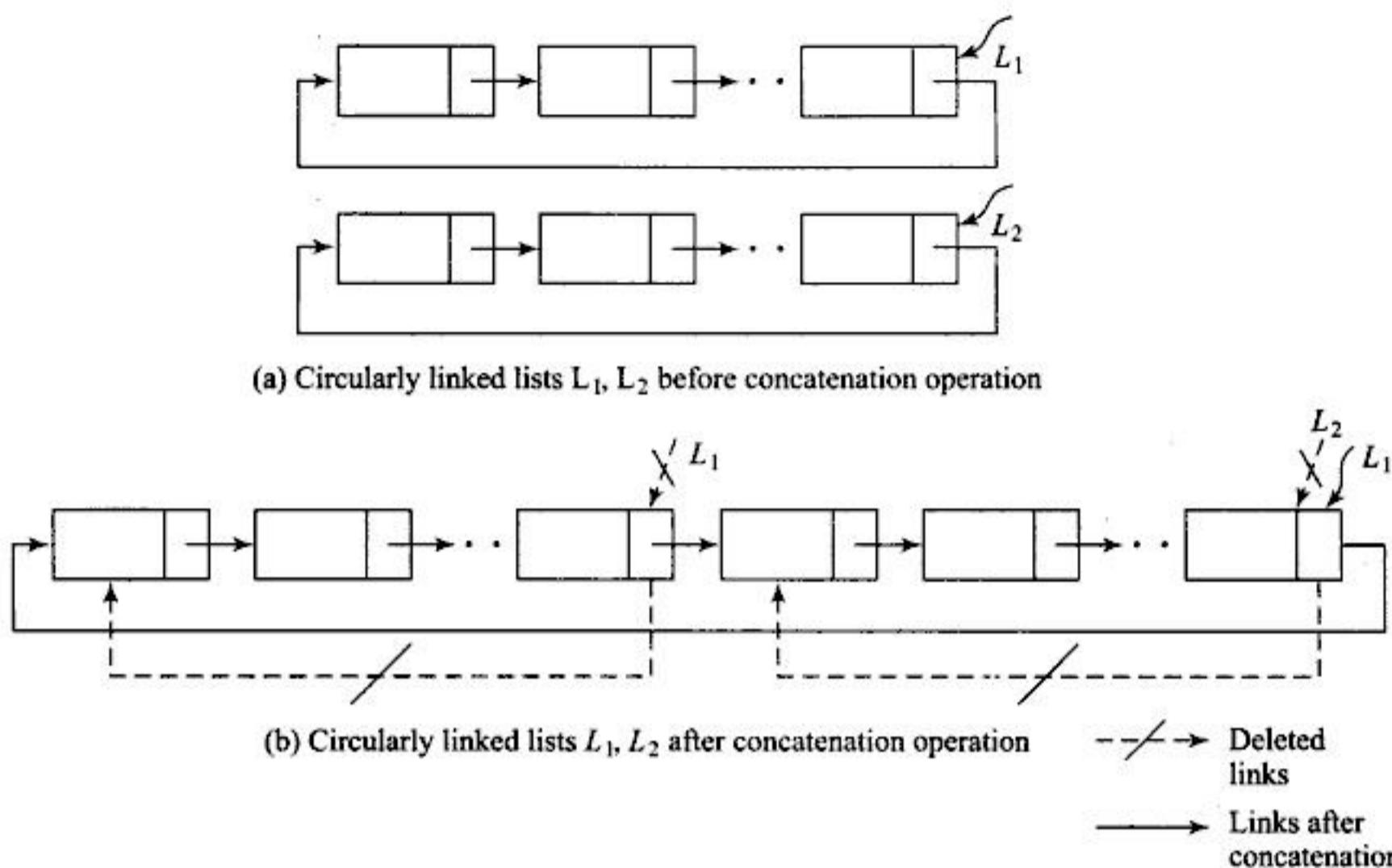


Fig. 6.15 Concatenation of two circularly linked lists

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



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



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



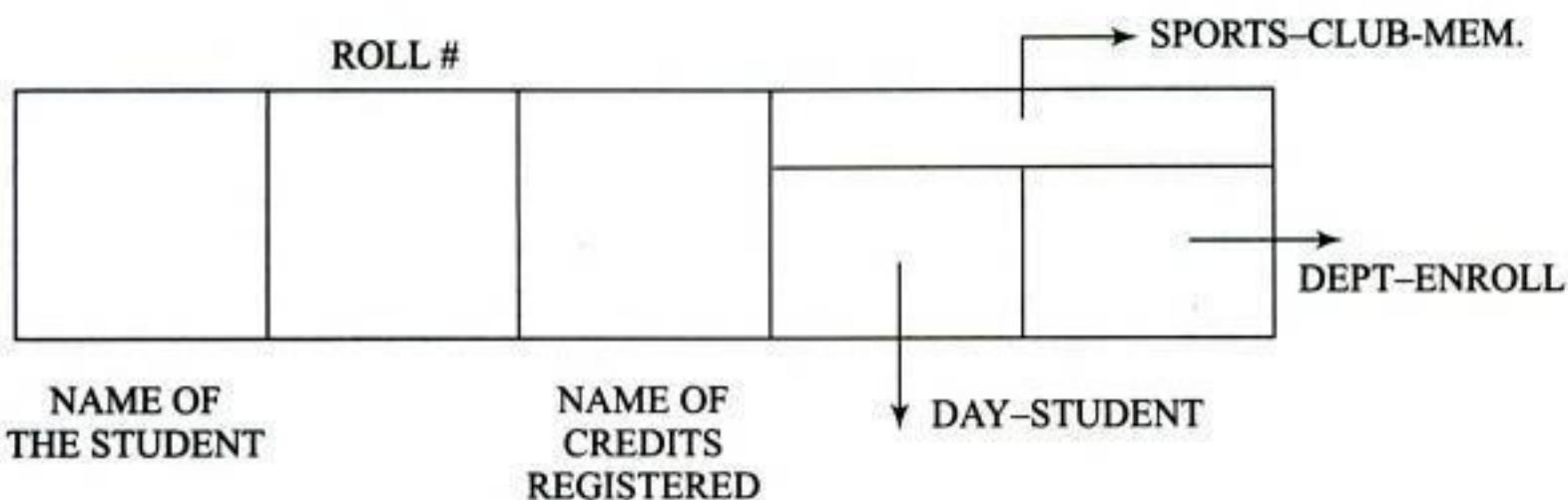
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



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



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



**Fig. 6.22** 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.,



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



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



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

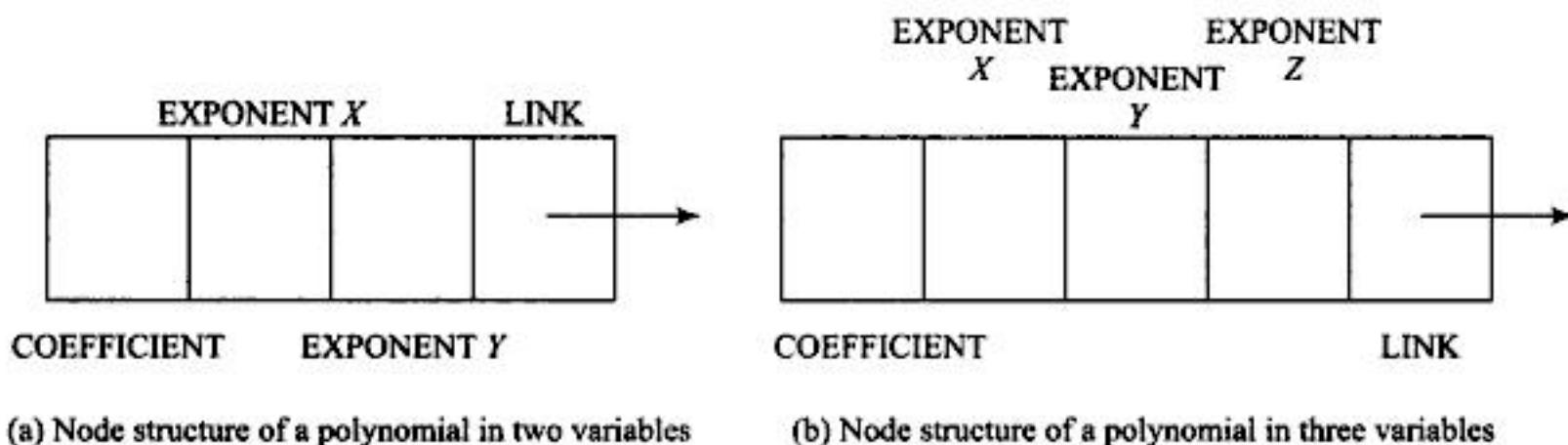
```

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



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



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



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

## 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.

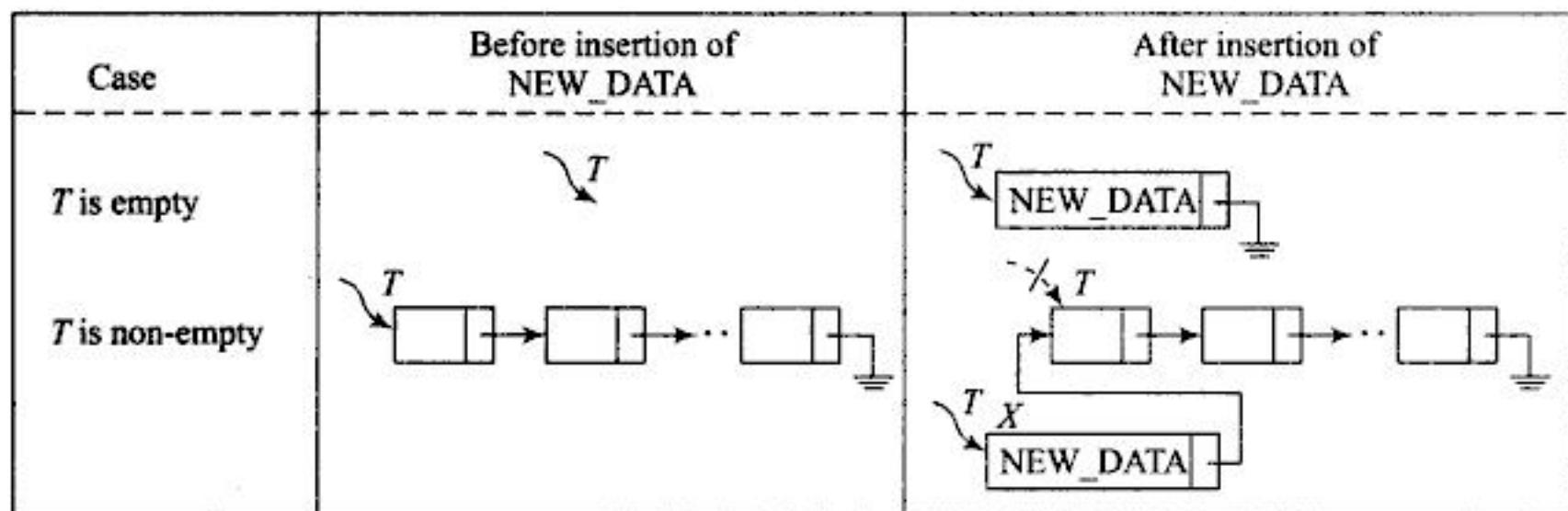


Fig. I 6.1

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  $k^{\text{th}}$  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  $k^{\text{th}}$  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;

```



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



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



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

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

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

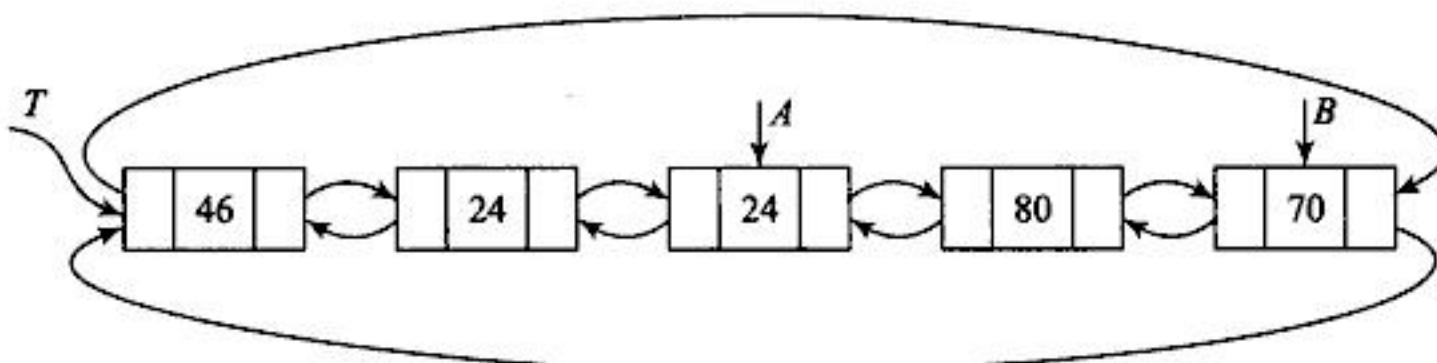


Fig. I 6.6(b)

**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  $P \uparrow$ .  $\text{DATA}$  and  $P \uparrow.\text{LINK}$  in the programming language.

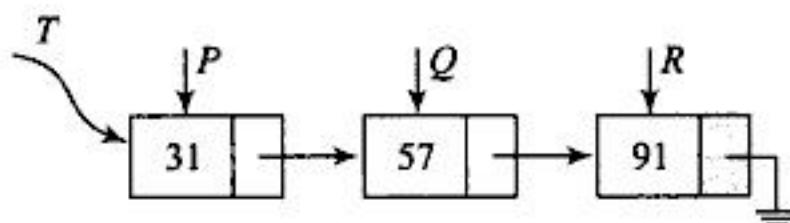
What do the following commands do to the logical representation of the list  $T$ ?

```

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



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



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

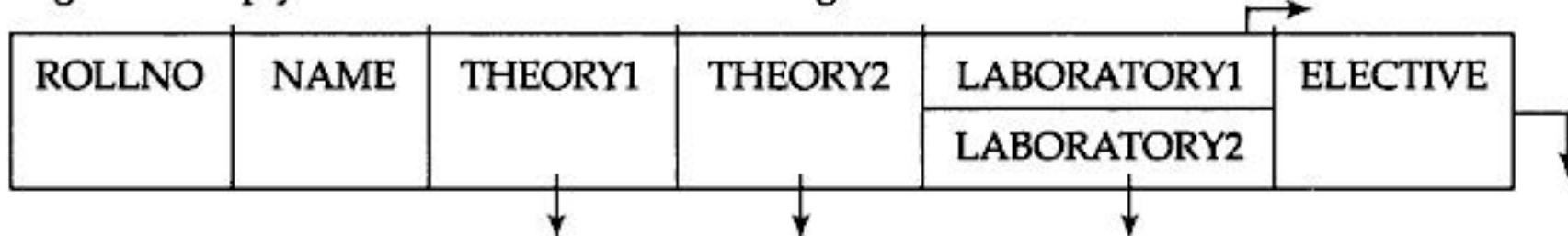


You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



## 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.

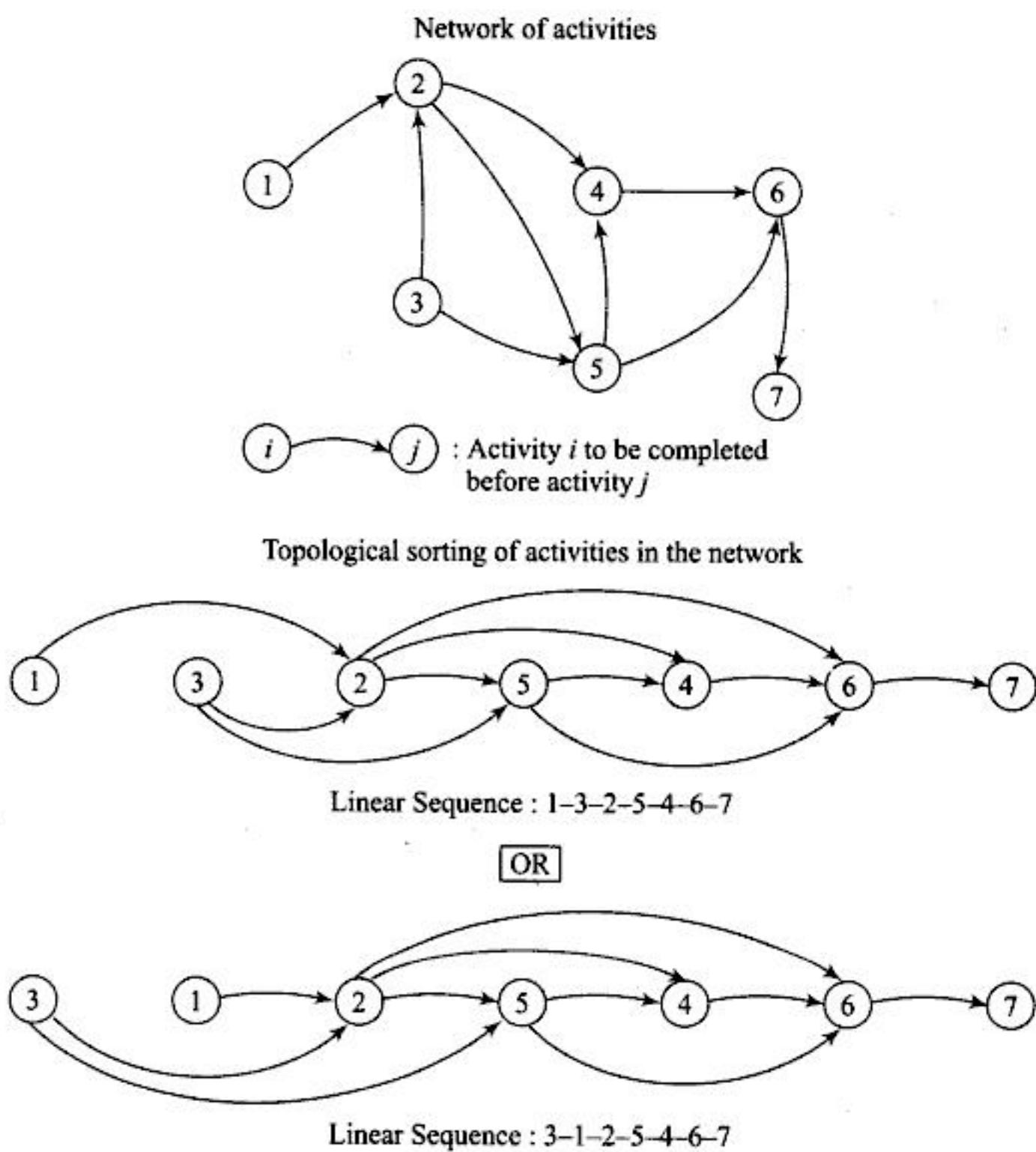
Execute a program to implement the multiply linked list with provisions to insert nodes, to update information besides generating reports as needed by the department.

- [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**



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



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



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



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



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

**Algorithm 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

- (i) The conceptual and computational simplicity of the operations
- (ii) non finite capacity

The only demerit is the requirement of additional space that is needed to accommodate the link fields.



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



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



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

**Algorithm 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

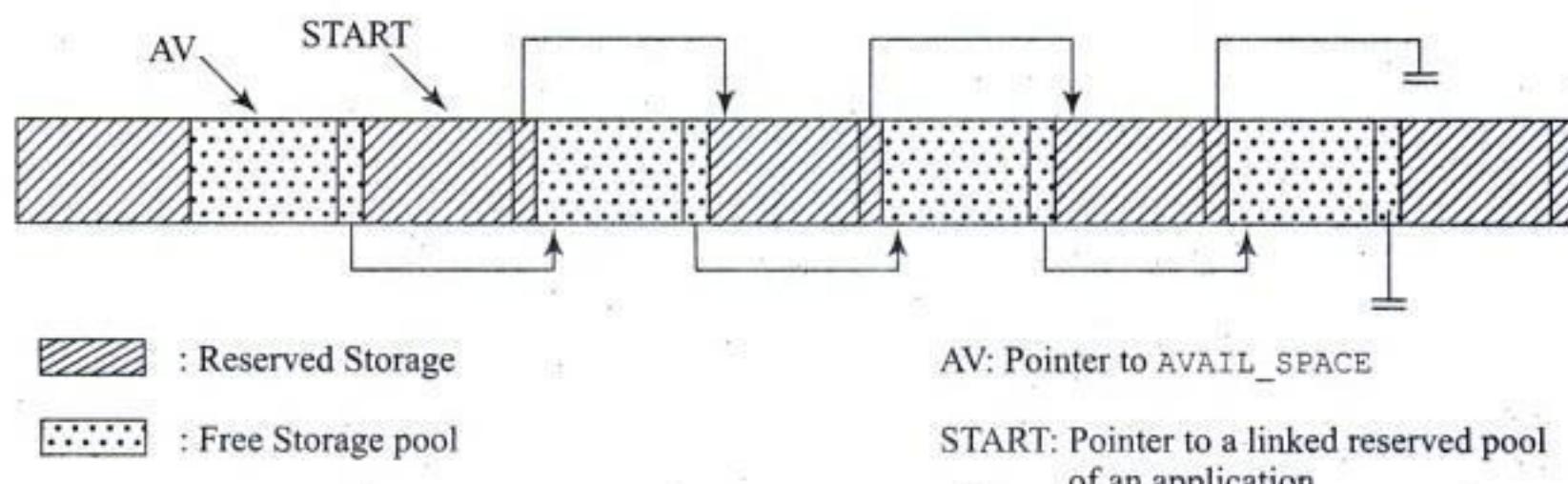
```

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.



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



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



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



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



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



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



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



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



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



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



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



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



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



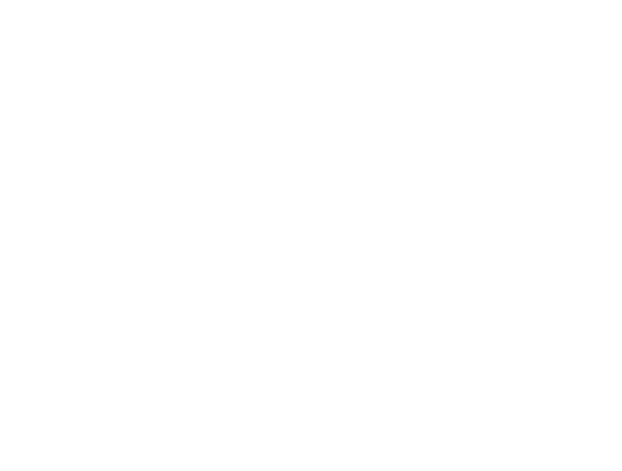
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



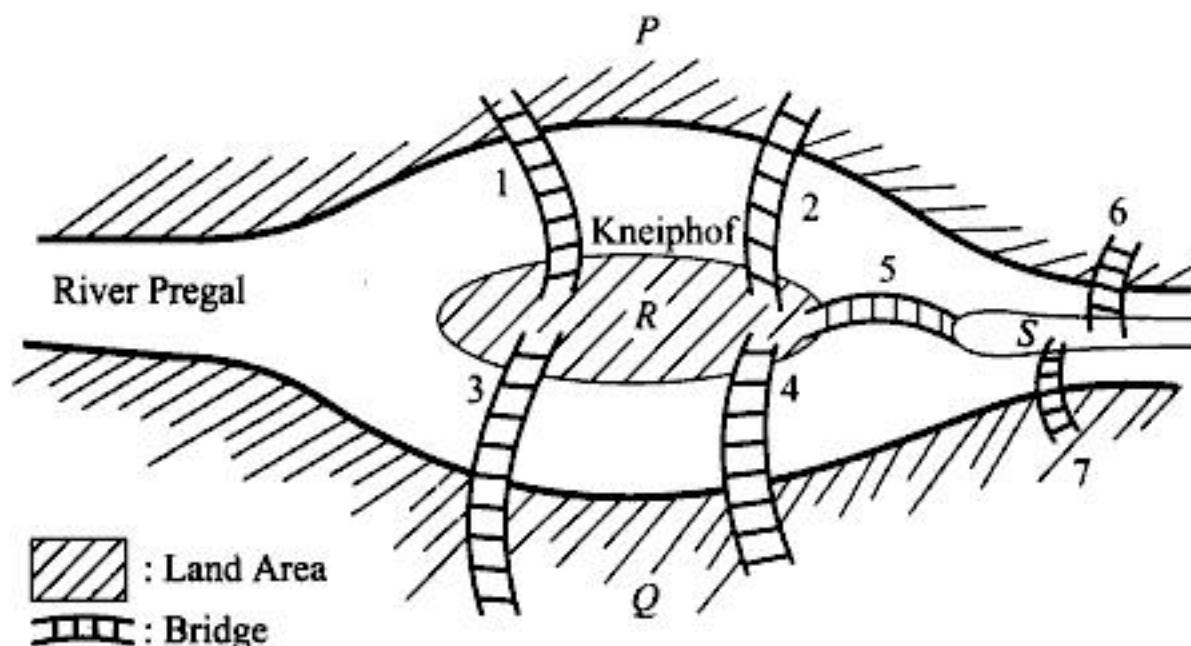
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



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



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

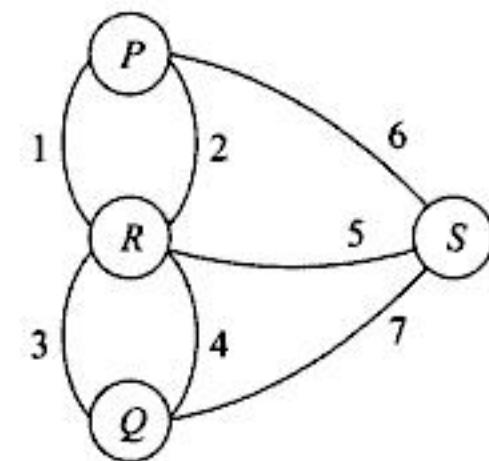


**Fig. 9.1** The Koenigsberg bridge problem

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

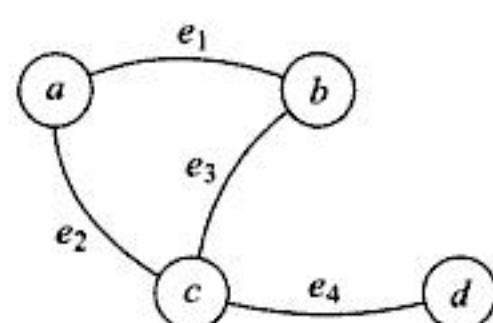
## Definitions and Basic Terminologies

### 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.

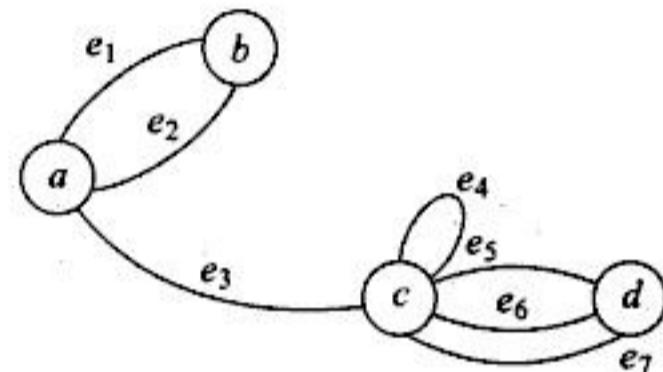


Fig. 9.4 A multigraph

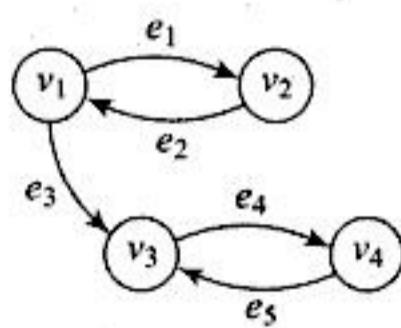
## Directed and undirected graphs

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*.

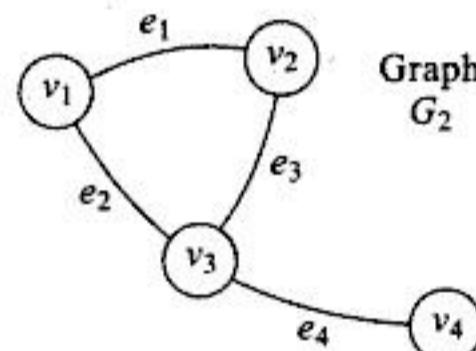
**Example** Figure 9.5(a-b) illustrates a digraph and an undirected graph.

Graph  
 $G_1$



(a) Digraph

Graph  
 $G_2$



(b) 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.)  $e_1 = (v_1, v_2)$ .



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



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



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

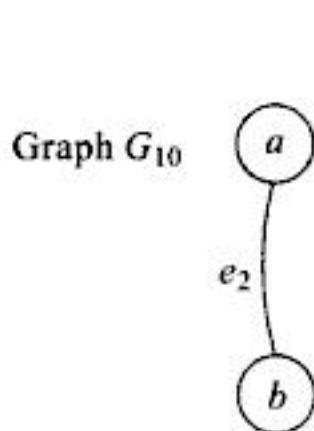


You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

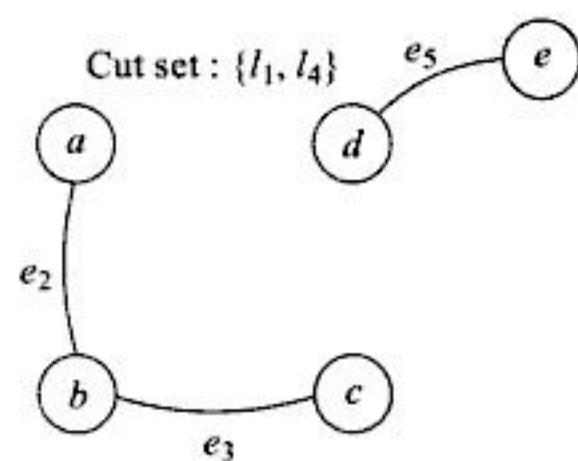


You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

**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



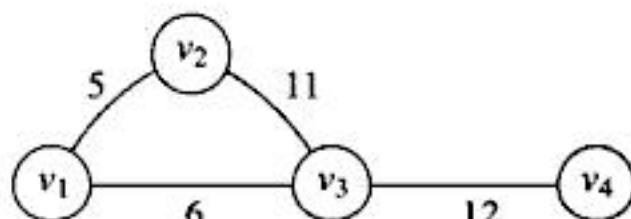
(a) A cut set of the graph

**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*.

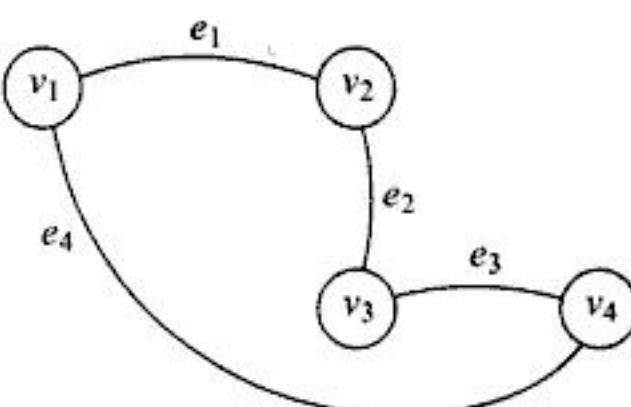
**Fig. 9.16 A labeled graph**

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  $G$ .



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



## Summary

- Red-black trees are derived from B trees of order 4 and are variants of binary search trees. Red-black trees need to satisfy the Red condition which entails no two red nodes can occur consecutively on a path in the tree, and the Black condition which insists that the number of black nodes on all root-to-external node paths must be the same.
- A search operation on a red-black tree is undertaken the same way as that on a binary search tree. The insertion of a key in a red-black tree is similar to the one in a binary search tree. However, the inserted node is set to red initially to avoid violation of the Black condition. If this results in a violation of the Red condition as well, then the tree is said to be unbalanced. The imbalance is classified as  $XYr$  or  $XYb$  where  $X, Y$  may represent an  $L$  or  $R$ . All  $XYr$  imbalances call for a mere colour change to set right the imbalance. On the other hand, all  $XYb$  imbalances call for rotations to set right the imbalance.
- The deletion of a node in a red-black tree proceeds as one would in a binary search tree. In the case of any violation of the Black condition, the imbalances are classified as  $Xb0$ ,  $Xb1$  and  $Xb2$  or  $Xr0$ ,  $Xr1$  and  $Xr2$  where  $X$  may be  $L$  or  $R$  and the appropriate rotations are undertaken to set right the imbalance.
- Splay trees are self-adjusting trees which are variants of binary search trees. The search and insert operations proceed as they would on binary search trees. However after the operation, the inserted key or the searched key is pushed up as the root using splay rotations.
- Splay rotations are classified as zig, zag, zig-zig, zig-zag, zag-zag and zig-zig rotations based on the position of the specific node at which the splaying is initiated.
- Though an insert or search operation on a splay tree may be expensive when undertaken for the first time, the same when considered over a long sequence of operations may prove to be efficient. Such an analysis which spreads over a sequence of operations and in which the expensive operations are averaged over the less expensive ones is what is called as *amortized analysis*. The amortized analysis of an access in a splay tree for a sequence of  $m$  operations is  $O(m \cdot \log n)$ .



## Illustrative Problems

**Problem 12.1** Construct a red-black tree inserting the following keys into an empty tree, in the sequence given:

40, 16, 36, 54, 18, 7, 48, 5

**Solution:** The snap shots of the red-black tree during its construction are shown in Fig. I 12.1. During the insertion of 54 into the tree, an  $RRr$  imbalance is encountered (Fig. I 12.1(d)). Rebalancing the tree calls for a colour change which affects the colour of the root (36) violating the property that the root of a red-black tree should be black. In such a case the colour change is made



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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

In this chapter, we discuss the concept of files and their methods of organization, viz., heap or pile files, sequential files, indexed sequential files and direct files.

## Files

## 14.2

A file is commonly thought of as a folder that holds a sheaf of related documents arranged according to some order. In the context of secondary storage devices, the storage and organization of related data is referred to as a *file*. In fact a file is a *logical organization of data*. A file is technically defined to be a collection of *records*. A record is a logical collection of *fields*. A field is a collection of characters, which can be either numeric or alphabetic or alphanumeric. A file could be a collection of *fixed length records* or *variable length records*, where *length of a record* is indicative of the number of characters that makes up a record.

Let us consider the example of a student file. The file is a logical collection of student records. A student record is a collection of fields such as roll number, name, city, date of birth, grade etc. Each of these fields could be numeric, alphabetic or alphanumeric. A sample set of student records are shown below:

### Student file

roll number	name	city	date of birth	grade
06MX66	Azad Ali	New Delhi	06121980	S
06MX74	Kamala Devi	Allahabad	11111980	A
06MX88	Andy Jones	Goa	12011980	S
06MX89	Sukh Dev	Bangalore	14041981	B

A file is a logical entity and has to be mapped on to a physical medium for its storage and access. To facilitate storage it is essential to know the *field length* or *field size* (normally specified in bytes). Thus every file has its *physical organization*.

For example, the student file stored on a magnetic tape would have the records listed above occurring sequentially as shown in Fig. 14.1. In such a case the processing of these records would only call for the application of sequential data structures. In fact, in the case of magnetic tapes, the logical organization of the records in the files and their physical organization when stored in the tape, are one and the same.

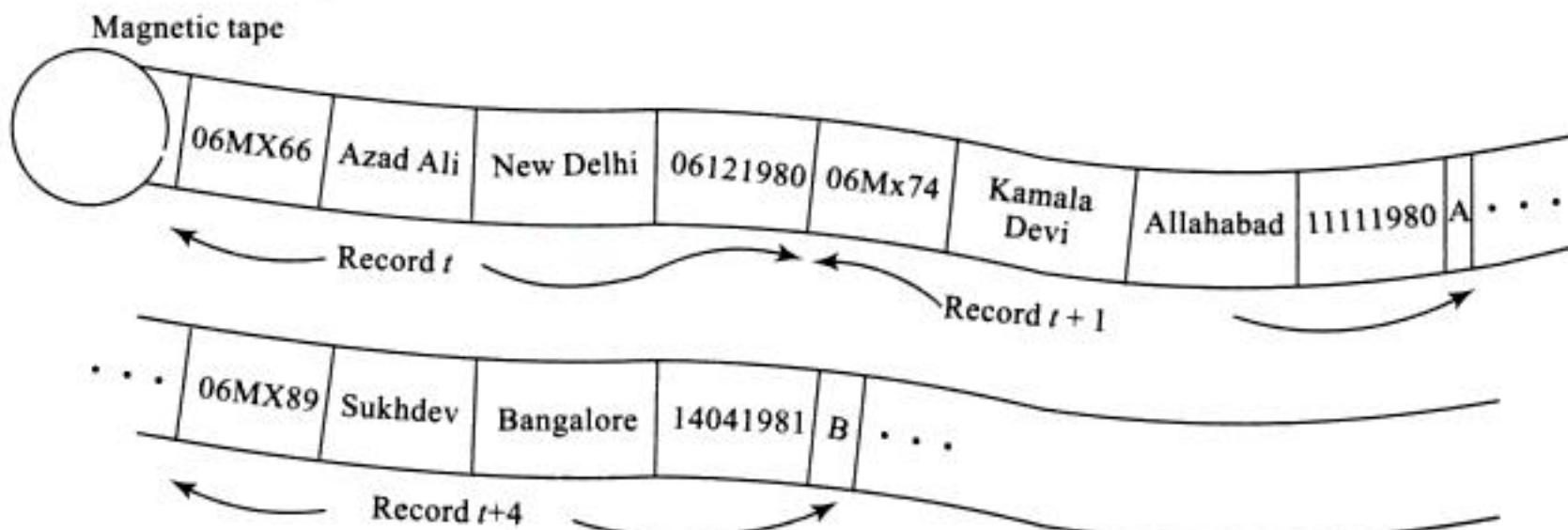


Fig. 14.1 Physical organization of the student file on a magnetic tape

On the other hand, on a magnetic disk the student file could be stored either sequentially or non sequentially (random access) as called for by the applications using the file. In the case of random access the records are physically stored in various portions of the disk where space is available. Figure 14.2 illustrates a snap shot of the student file storage in the disk. The logical organization of the records in the file is kept track of by physically linking the records through pointers. The processing of such files would call for linked data structures. Thus, in the case of magnetic disks, for files that have been stored in a non-sequential manner, the logical and the physical organizations need not coincide.

The physical organization of the files is designed and ordered by the File Manager of the operating system.

## Keys

## 14.3

In a file, one or more fields could serve to *uniquely identify* the records for efficient retrieval and storage. These fields are known as *primary keys* or commonly, *keys*. For example, in the student file discussed above, roll number could be designated as the primary key for it uniquely identifies each student and hence the record too.

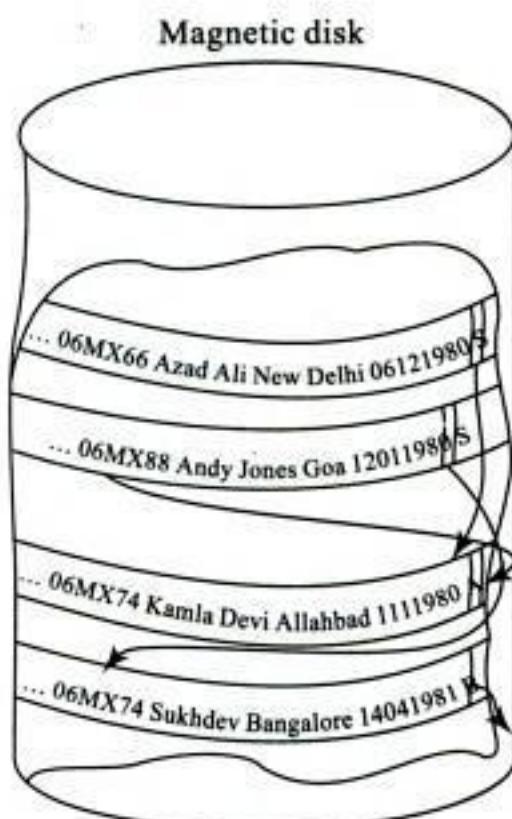
If additional fields were added to the primary key, the combination would still continue to uniquely identify the record. Such a combination of fields is referred to as a *super key*. For example, the combination of roll number and name would still continue to uniquely identify records in the student file. A primary key can therefore be described as a *minimal super key*.

It is possible to have more than one combination of fields that can serve to uniquely identify a record. These combinations are known as *candidate keys*. It now depends on the file administrator to choose any one combination as the primary key. In such a case, the rest of the combinations are called as *alternate keys*. For example, consider an employee file shown below. Here, both the fields, employee number and social security number could act as the primary keys since both would serve to uniquely identify the record. Thus we term them as candidate keys. If we chose to have employee number as the primary key then social security number would be referred to as alternate key.

A field or a combination of fields that may not be a candidate key but can serve to classify records based on a particular characteristic are called *secondary keys*. For example in the employee file, department could be a secondary key to classify employees based on the department.

### Employee file

employee number	name	social security number	department	designation
M345	Abdul	IN-E-765432190	Mining	Engineer
T786	Bhagath	IN-E-678902765	Administration	Officer
M678	Gargi	IN-E-120119809	Mining	Manager



**Fig. 14.2** Physical organization of the student file on a magnetic disk



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



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



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



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



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



You have either reached a page that is unavailable for viewing or reached your 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) (A, (i) ) (B, (iv) ) (C, (iii) ) (D, (ii) )  
 (b) (A, (ii) ) (B, (iv) ) (C, (iii) ) (D, (i) )  
 (c) (A, (ii) ) (B, (i) ) (C, (iv) ) (D, (iii))  
 (d) (A, (iii) ) (B, (i) ) (C, (ii) ) (D, (iv) )

5. Find the odd term out in the context of basic file operations:  
 open close update delete evaluate read  
 (a) close (b) read (c) open (d) evaluate

6. Distinguish between primary memory and secondary memory.  
 7. Give examples for (i) superkey (ii) primary key (iii) secondary key (iv) alternate key  
 8. How are insertions and deletions carried out in a pile?  
 9. Distinguish between logical and physical deletion of records.  
 10. Compare the merits and demerits of a heap file with that of a sequential file organization.  
 11. How do ISAM files ensure random access of data?  
 12. What is the need for multilevel indexing in ISAM files?  
 13. When are cluster indexes used?  
 14. How are secondary indexes maintained?  
 15. What is external hashing?  
 16. A file comprises of the following sample set of primary keys. The block size in the primary storage area is 2. Design an ISAM file organization based on (i) primary indexing and (ii) multilevel indexing (level =3).  
 090 890 678 654 234 123 245 678 900 111 453 231 112 679 238 876 311 433  
 544 655 766 877 988 009 122 233 344 566 677 899 909 512 612 723 823 956  
 17. Making use of the hash function  $h(k) = k \text{ mod } 11$ , where  $k$  is the key, design a direct file organization for the sample file (list of primary keys) shown in Review Questions 16 (Chapter 14). Assume that the bucket size is 3 and the block size is 4.  
 18. Assume that the sample file (list of primary keys) shown in Review Questions 16 (Chapter 14) had a field called category which carries the character 'A' if the primary key is odd and 'B' if the primary key is even. Design a cluster index based file organization built on the field category. Assume a block size of 4.



# Programming Assignments

1. Implement the used car file discussed in Illustrative Problem 14.3 in a programming language of your choice that supports the data structures of files and records. Experiment on the basic operations of a file. What other operations does the language support to enhance the use of the file? Write a menu driven program to implement the operations.
  2. Assume that the used car file was implemented as a sequential file. Simulate the batched mode of updating the sequential file by creating a transaction file of insertions (details of cars that are brought in for sale) and deletions (cars that were sold out), to update the existing master file.
  3. A movie file has the following record structure:

name of the movie	producer	director	type	production cost
-------------------	----------	----------	------	-----------------



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



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



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

```

if ( L[i] = K) then { print (" KEY found"); /* key K found*/
    swap(L[i], L[i-1]); /* swap key with its
                           predecessor in the list*/
}
else
    print ("KEY not found");
end TRANSPOSE_SEQUENTIAL SEARCH.

```

**Example 15.3** Consider an unordered list  $L = \{34, 21, 89, 45, 12, 90, 76, 62\}$  of data elements. Let us search for the following elements in the order of their appearance:

90, 89, 90, 21, 90, 90.

Transpose sequential search proceeds to find each key by the usual process of checking it against each element of  $L$  one at a time. However, once the key is found it is swapped with its predecessor in the list. Table 15.1 illustrates the number of comparisons made for each key during its search. The list  $L$  before and after the search operation are also illustrated in the table. The swapped elements in the list  $L$  after the search key is found is shown in bold. Observe how the number of comparisons made for the retrieval of 90 which is repeatedly looked for in the search list, decreases with each search operation.

**Table 15.1** Transpose sequential search of {90, 89, 90, 21, 90, 90} in the list  $L=\{34, 21, 89, 45, 12, 90, 76, 62\}$

Search key	List $L$ before search	Number of element comparisons made during the search	List $L$ after search
90	{34, 21, 89, 45, 12, 90, 76, 62}	6	{34, 21, 89, 45, <b>90</b> , 12, 76, 62}
89	{34, 21, 89, 45, 90, 12, 76, 62}	3	{34, <b>89</b> , 21, 45, 90, 12, 76, 62}
90	{34, 89, 21, 45, 90, 12, 76, 62}	5	{34, 89, 21, <b>90</b> , 45, 12, 76, 62}
21	{34, 89, 21, 90, 45, 12, 76, 62}	3	{34, <b>21</b> , 89, 90, 45, 12, 76, 62}
90	{34, 21, 89, 90, 45, 12, 76, 62}	4	{34, 21, <b>90</b> , 89, 45, 12, 76, 62}
90	{34, 21, 90, 89, 45, 12, 76, 62}	3	{34, <b>90</b> , 21, 89, 45, 12, 76, 62}

The worst case complexity in terms of comparisons for finding a specific key in the list  $L$  is  $O(n)$ . In the case of repeated searches for the same key the best case would be  $O(1)$ .

## Interpolation Search

## 15.4

Some search methods employed in every day life can be interesting. For example, when one looks for the word "beatitude" in the dictionary, it is quite common for one to turn over pages occurring at the beginning of the dictionary, and when one looks for "tranquility", to turn over pages occurring towards the end of the dictionary. Also, it needs to be observed how during the search we turn sheaves of pages back and forth, if the word that is looked for occurs before or beyond the page that has just been turned. In fact, one may look askance at anybody who 'dares' to undertake sequential search to look for "beatitude" or "tranquility" in a dictionary!



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



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



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



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



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

graph, any of the two traversals may be undertaken on the graph to look for the key. In such a case we term the traversal techniques as Breadth first search (see Illustrative Problem 15.6) and Depth first search (see Illustrative Problem 15.7).

## Indexed sequential search

The Indexed sequential search (see Sec. 15.7) is a successful search technique applicable on files that are too large to be accommodated in the internal memory of the computer. Also known as Indexed Sequential Access Method (ISAM), the search procedure and its variants have been successfully applied to Database systems.

Considering the fact that the search technique is commonly used on data bases or files which span several blocks of storage areas, the technique could be deemed as an external searching technique. To search for a key one needs to look into the index to obtain the storage block where the associated group of records or elements are available. Once the block is retrieved, the retrieval of the record represented by the key merely reduces to a sequential search within the block of records for the key.



## Summary

- The problem of search involves retrieving a key from a list of data elements. In the case of a successful retrieval the search is deemed to be successful otherwise it is unsuccessful.
- The search techniques that work on lists or files that can be accommodated within the internal memory of the computer, are called internal searching methods, otherwise they are called as external searching methods.
- Sequential search involves looking for a key in a list  $L$  which may or may not be ordered. However an ordered sequential search is more efficient than its unordered counterpart.
- A transpose sequential search sequentially searches for a key in a list but swaps it with the predecessor once it is found. This enables efficient search of keys that are repeatedly looked for in a list.
- Interpolation search imitates the kind of search process that one employs while referring to a dictionary. The search key is compared with data elements at "calculated positions" and the process progresses based on whether the key occurs before or after it. However it is essential that the list is ordered.
- Binary search is a successful and efficient search technique that works on ordered lists. The search key is compared with the element at the median of the list. Based on whether the key occurs before or after it the search list is reduced and the search process continues in a similar fashion in the sublist.
- Fibonacci search works on ordered lists and employs the Fibonacci number sequence and its characteristics to search through the list.
- Tree data structures viz., AVL trees, tries,  $m$ -way search trees, B trees etc., and graphs also find applications in search related problems. Indexed sequential search is a popular search technique employed in the management of files and databases.



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



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



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



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

4. Which among the following search techniques does not report a worst case time complexity of  $O(n)$ ?
  - (a) linear search
  - (b) interpolation search
  - (c) transpose sequential search
  - (d) binary search
5. Which among the following search techniques works on unordered lists?
  - (a) Fibonacci search
  - (b) interpolation search
  - (c) transpose sequential search
  - (d) binary search
6. What are the advantages of binary search over sequential search?
7. When is a transpose sequential search said to be most successful?
8. What is the principle behind interpolation search?
9. Distinguish between internal searching and external searching.
10. What are the characteristics of the decision tree of Fibonacci search?
11. How is breadth first search evolved from breadth first traversal of a graph?
16. For the following search list undertake (i) linear ordered search (ii) binary search in the data list given. Tabulate the number of comparisons made for each key in the search list.  
Search list: {766, 009, 999, 238}  
Data list: {111 453 231 112 679 238 876 655 766 877 988 009 122 233 344 566}
17. For the given data list and search list, tabulate the number of comparisons made when (i) a transpose sequential search and (ii) interpolation search is undertaken on the keys belonging to the search list.  
Data list: {pin, ink, pen, clip, ribbon, eraser, duster, chalk, pencil, paper, stapler, pot, scale, calculator}  
Search list: {pen, clip, paper, pen, calculator, pen}
18. Undertake Fibonacci search of the key  $K = 67$  in the list { 11, 89, 34, 15, 90, 67, 88, 01, 36, 98, 76, 50}. Trace the decision tree for the search.
19. Perform (i) Breadth first search and (ii) Depth first search, on the graph given in Fig. R 15.19 for the key  $V$ .

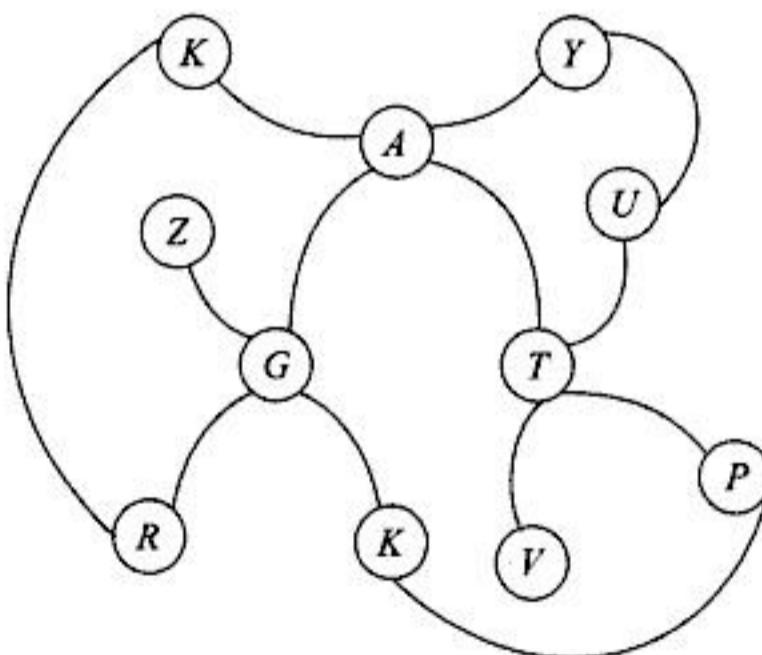


Fig. R 15.19



# INTERNAL SORTING

# 16

## Introduction

## 16.1

Sorting in English language refers to separating or arranging things according to different classes. However, in computer science, *sorting* also referred to as *ordering* deals with arranging elements of a list or a set or records of a file in the ascending or descending order.

In the case of sorting a list of alphabetical or numerical or alphanumerical elements, the elements are arranged in their ascending or descending order based on their alphabetical or numerical sequence number. The sequence is also referred to as a *collating sequence*. In the case of sorting a file of records, one or more fields of the records are chosen as the key based on which the records are arranged in the ascending or descending order.

Examples of lists before and after sorting are shown below:

### Unsorted lists

{ 34, 12, 78, 65, 90, 11, 45}

{ tea, coffee, cocoa, milk, malt, chocolate}

{n12x, m34b, n24x, a78h, g56v, m12k, k34d}

{11, 12, 34, 45, 65, 78, 90}

{ chocolate, cocoa, coffee, malt, milk, tea}

{a78h, g56v, k34d, m12k, m34b, n12x, n24x}

### Sorted lists

Sorting has acquired immense significance in the discipline of computer science. Several data structures and algorithms display efficient performance when presented with sorted data sets.

Many different sorting algorithms have been invented each having its own advantages and disadvantages. These algorithms may be classified into families such as *sorting by exchange*, *sorting by insertion*, *sorting by distribution*, *sorting by selection* and so on. However in many cases, it is difficult to classify the algorithms as belonging to only a specific family.

A sorting technique is said to be *stable* if keys that are equal retain their relative orders of occurrence even after sorting. In other words, if  $K_1, K_2$  are two keys such that  $K_1 = K_2$ , and  $p(K_1) < p(K_2)$  where  $p(K_i)$  is the position index of the keys in the unsorted list, then after sorting,  $p'(K_1) < p'(K_2)$  where  $p'(K_i)$  is the index positions of the keys in the sorted list.

If the list of data or records to be sorted are small enough to be accommodated in the internal memory of the computer, then it is referred to as *internal sorting*. On the other hand if the data list or records to be sorted are voluminous and are accommodated in external storage devices such as tapes, disks and drums, then the sorting undertaken is referred to as *external sorting*. External sorting methods are quite different from internal sorting methods and are discussed in Chapter 17.

16.1 Introduction

16.2 Bubble Sort

16.3 Insertion Sort

16.4 Selection Sort

16.5 Merge Sort

16.6 Shell Sort

16.7 Quick Sort

16.8 Heap Sort

16.9 Radix Sort

In this chapter we discuss the internal sorting techniques of Bubble Sort, Insertion Sort, Selection sort, Merge Sort, Shell sort, Quick Sort, Heap Sort and Radix Sort.

## Bubble Sort

## 16.2

Bubble sort belongs to the family of *sorting by exchange or transposition*, where during the sorting process pairs of elements that are out of order are interchanged until the whole list is ordered. Given an unordered list  $L = \{K_1, K_2, K_3, \dots, K_n\}$  bubble sort orders the elements in their ascending order (i.e.),  $L = \{K_1, K_2, K_3, \dots, K_n\}, K_1 \leq K_2 \leq \dots \leq K_n$

Given the unordered list  $L = \{K_1, K_2, K_3, \dots, K_n\}$ , of keys, bubble sort compares pairs of elements  $K_i$  and  $K_j$  swapping them if  $K_i > K_j$ . At the end of the first pass of comparisons, the largest element in the list  $L$  moves to the last position in the list. In the next pass, the sublist  $\{K_1, K_2, K_3, \dots, K_{n-1}\}$  is considered for sorting. Once again the pair wise comparison of elements in the sub list results in the next largest element floating to the last position of the sublist. Thus in  $(n-1)$  passes where  $n$  is the number of elements in the list, the list  $L$  is sorted. The sorting is called bubble sorting for the reason that, with each pass the next largest element of the list floats or "bubbles" to its appropriate position in the sorted list.

Algorithm 16.1 illustrates the working of bubble sort.

### Algorithm 16.1: Procedure for Bubble sort

```

procedure BUBBLE_SORT(L, n)
    /* L[1:n] is an unordered list of data elements to be
       sorted in the ascending order */

    for i = 1 to n-1 do /* n-1 passes*/
        for j = 1 to n-i do
            if (L[j] > L[j+1]) swap(L[j], L[j+1]);
                /* swap pair wise elements*/
            end      /* the next largest element "bubbles" to the last position*/
        end
    end BUBBLE_SORT.

```



**Example 16.1** Let  $L = \{92, 78, 34, 23, 56, 90, 17, 52, 67, 81, 18\}$  be an unordered list. As the first step in the first pass of bubble sort, 92 is compared with 78. Since  $92 > 78$ , the elements are swapped yielding the list  $\{78, 92, 34, 23, 56, 90, 17, 52, 67, 81, 18\}$ . The swapped elements are shown in bold. Now the pair 92 and 34 are compared resulting in a swap which yields the list  $\{78, 34, 92, 23, 56, 90, 17, 52, 67, 81, 18\}$ . It is easy to see that at the end of pass one, the largest element of the list viz., 92 would have moved to the last position in the list. At the end of pass one, the list would be  $\{78, 34, 23, 56, 90, 17, 52, 67, 81, 18, 92\}$ .

In the second pass the list considered for sorting discounts the last element viz., 92 since 92 has found its appropriate position in the sorted list. At the end of the second pass, the next largest element viz., 90 would have moved to the end of the list. The partially sorted list at this point would be  $\{34, 23, 56, 78, 17, 52, 67, 81, 18, 90, 92\}$ . The elements shown in grey indicate elements discounted from the sorting process. In pass 10 the whole list would be completely sorted.

The trace of algorithm BUBBLE\_SORT (Algorithm 16.1) over  $L$  is shown in Table 16.1. Here  $i$  keeps count of the passes and  $j$  keeps track of the pair wise element comparisons within a pass. The lower ( $l$ ) and upper ( $u$ ) bounds of the loop controlled by  $j$  in each pass is shown as  $l..u$ . Elements shown in grey and underlined in the list  $L$  at the end of pass  $i$ , indicate those discounted from the sorting process.

**Table 16.1** Trace of Algorithm 16.1 over the list  $L = \{92, 78, 34, 23, 56, 90, 17, 52, 67, 81, 18\}$

(Pass) $i$	$j$	List $L$ at the end of Pass $i$
1	1..10	{ 78, 34, 23, 56, 90, 17, 52, 67, 81, 18, 92 }
2	1..9	{ 34, 23, 56, 78, 17, 52, 67, 81, 18, 90, <u>92</u> }
3	1..8	{ 23, 34, 56, 17, 52, 67, 78, 18, 81, <u>90, 92</u> }
4	1..7	{ 23, 34, 17, 52, 56, 67, 18, 78, <u>81, 90, 92</u> }
5	1..6	{ 23, 17, 34, 52, 56, 18, 67, <u>78, 81, 90, 92</u> }
6	1..5	{ 17, 23, 34, 52, 18, 56, <u>67, 78, 81, 90, 92</u> }
7	1..4	{ 17, 23, 34, 18, 52, <u>56, 67, 78, 81, 90, 92</u> }
8	1..3	{ 17, 23, 18, 34, <u>52, 56, 67, 78, 81, 90, 92</u> }
9	1..2	{ 17, 18, 23, <u>34, 52, 56, 67, 78, 81, 90, 92</u> }
10	1..1	{ 17, 18, <u>23, 34, 52, 56, 67, 78, 81, 90, 92</u> }

### Stability and performance analysis

Bubble sort is a stable sort since equal keys do not undergo swapping, as can be observed in Algorithm 16.1, and this contributes to the keys maintaining their relative orders of occurrence in the sorted list.

**Example 16.2** Consider the unordered list  $L = \{7^1, 7^2, 7^3, 6\}$ . The repeating keys have been distinguished using their orders of occurrence as superscripts. The partially sorted lists at the end of each pass of the bubble sort algorithm are shown below:

Pass 1: {  $7^1, 7^2, 6, 7^3$  }

Pass 2: {  $7^1, 6, 7^2, 7^3$  }

Pass 3: {  $6, 7^1, 7^2, 7^3$  }

Observe how the equal keys  $7^1, 7^2, 7^3$  maintain their relative orders of occurrence in the sorted list as well, verifying the stability of bubble sort.

The time complexity of bubble sort in terms of key comparisons is given by  $O(n^2)$ . It is easy to see this since the procedure involves two loops with their total frequency count given by  $O(n^2)$ .

### Insertion Sort

### 16.3

Insertion sort as the name indicates belongs to the family of *sorting by insertion* which is based on the principle that a new key  $K$  is *inserted* at its appropriate position in an already sorted sub list.

Given an unordered list  $L = \{K_1, K_2, K_3, \dots, K_n\}$ , insertion sort employs the principle of constructing the list  $L = \{K_1, K_2, K_3, \dots, K_i, K, K_j, K_{j+1}, \dots, K_n\}$ ,  $K_1 \leq K_2 \leq \dots \leq K_i$  and inserting a key  $K$  at its appropriate position by comparing it with its sorted sublist of predecessors  $\{K_1, K_2, K_3, \dots, K_i\}$ ,  $K_1 \leq K_2 \leq \dots \leq K_i$  for every key  $K$  ( $K = K_i = 2, 3, \dots, n$ ) belonging to the unordered list  $L$ .

In the first pass of insertion sort,  $K_2$  is compared with its sorted sublist of predecessors viz.,  $K_1$ .  $K_2$  inserts itself at the appropriate position to obtain the sorted sublist  $\{K_1, K_2\}$ . In the second pass,  $K_3$  compares itself with its sorted sublist of predecessors viz.,  $\{K_1, K_2\}$  to insert itself at its appropriate position yielding the sorted list  $\{K_1, K_2, K_3\}$  and so on. In the  $(n-1)^{\text{th}}$  pass,  $K_n$  compares itself with its sorted sublist of predecessors  $\{K_1, K_2, \dots, K_{n-1}\}$  and having inserted itself at the appropriate position yields the final sorted list  $L = \{K_1, K_2, K_3, \dots, K_i, \dots, K_j, \dots, K_n\}$ ,  $K_1 \leq K_2 \leq \dots \leq K_i \leq \dots \leq K_j \leq \dots \leq K_n$ . Since each key  $K$  finds its appropriate position in the sorted list, such a technique is referred to as *sinking* or *sifting* technique.

Algorithm 16.2 illustrates the working of Insertion sort. The **for** loop in the algorithm keeps count of the passes and the **while** loop implements the comparison of the key **key** with its sorted sublist of predecessors. So long as the preceding element in the sorted sublist is greater than **key** the swapping of the element pair is done. If the preceding element in the sorted sublist is less than or equal to **key**, then **key** is left at its current position and the current pass terminates.

**Example 16.3** Let  $L = \{16, 36, 4, 22, 100, 1, 54\}$  be an unordered list of elements. The various passes of the insertion sort procedure are shown below. The snapshots of the list before and after each pass is shown. The key chosen for insertion in each pass is shown in bold and the sorted sublist of predecessors against which the key is compared are shown in brackets.

Pass 1 (Insert 36)	{ [16] 36, 4, 22, 100, 1, 54}
After Pass 1	{ [16 36] 4, 22, 100, 1, 54}
Pass 2 (Insert 4)	{ [16 36] 4, 22, 100, 1, 54}
After Pass 2	{ [4 16 36] 22, 100, 1, 54}
Pass 3 (Insert 22)	{ [4 16 36] 22, 100, 1, 54}
After Pass 3	{ [4 16 22 36] 100, 1, 54}
Pass 4 (Insert 100)	{ [4 16 22 36] 100, 1, 54}
After Pass 4	{ [4 16 22 36 100] 1, 54}
Pass 5 (Insert 1)	{ [4 16 22 36 100] 1, 54}
After Pass 5	{ [1 4 16 22 36 100] 54}
Pass 6 (Insert 54)	{ [1 4 16 22 36 100] 54}
After Pass 6	{ [1 4 16 22 36 54 100]}

#### Algorithm 16.2: Procedure for Insertion sort

```

procedure INSERTION_SORT(L, n)
    /* L[1:n] is an unordered list of data elements to be sorted
       in the ascending order */

    for i = 2 to n do           /* n-1 passes*/
        key = L[i];          /* key is the key to be inserted
                                    and position its location in the
                                    unordered list*/

```

```

position = i;
    /* compare key with its sorted
       sublist of predecessors for insertion
       at the appropriate position*/
while (position > 1) and (L[position-1] > key) do
    L[position] = L[position-1];
    position = position -1;
    L[position] = key;
end
end
end INSERTION_SORT.

```

## Stability and performance analysis

Insertion sort is a stable sort. It is evident from the algorithm that the insertion of key K at its appropriate position in the sorted sublist affects the position index of the elements in the sublist so long as the elements in the sorted sublist are greater than K. When the elements are less than or equal to the key K, there is no displacement of elements and this contributes to retaining the original order of keys which are equal, in the sorted sublists.

**Example 16.4** Consider the list  $L = \{3^1, 1, 2^1, 3^2, 3^3, 2^2\}$  where the repeated keys have been superscripted with numbers indicative of their relative orders of occurrence. The keys for insertion are shown in bold and the sorted sublists are bracketed.

The passes of the insertion sort are shown below:

Pass 1 (Insert 1)	{ [3 <sup>1</sup> ] <b>1</b> , 2 <sup>1</sup> , 3 <sup>2</sup> , 3 <sup>3</sup> , 2 <sup>2</sup> }
After Pass 1	{ [1 3 <sup>1</sup> ] 2 <sup>1</sup> , 3 <sup>2</sup> , 3 <sup>3</sup> , 2 <sup>2</sup> }
Pass 2 (Insert 2)	{ [1 3 <sup>1</sup> ] <b>2<sup>1</sup></b> , 3 <sup>2</sup> , 3 <sup>3</sup> , 2 <sup>2</sup> }
After Pass 2	{ [1 2 <sup>1</sup> 3 <sup>1</sup> ] 3 <sup>2</sup> , 3 <sup>3</sup> , 2 <sup>2</sup> }
Pass 3 (Insert 3)	{ [1 2 <sup>1</sup> 3 <sup>1</sup> ] <b>3<sup>2</sup></b> , 3 <sup>3</sup> , 2 <sup>2</sup> }
After Pass 3	{ [1 2 <sup>1</sup> 3 <sup>1</sup> 3 <sup>2</sup> ] 3 <sup>3</sup> , 2 <sup>2</sup> }
Pass 4 (Insert 3)	{ [1 2 <sup>1</sup> 3 <sup>1</sup> 3 <sup>2</sup> ] <b>3<sup>3</sup></b> , 2 <sup>2</sup> }
After Pass 4	{ [1 2 <sup>1</sup> 3 <sup>1</sup> 3 <sup>2</sup> 3 <sup>3</sup> ] 2 <sup>2</sup> }
Pass 5 (Insert 2)	{ [1 2 <sup>1</sup> 3 <sup>1</sup> 3 <sup>2</sup> 3 <sup>3</sup> ] <b>2<sup>2</sup></b> }
After Pass 5	{ [1 2 <sup>1</sup> 2 <sup>2</sup> 3 <sup>1</sup> 3 <sup>2</sup> 3 <sup>3</sup> ] }

The stability of insertion sort can be easily verified on this example. Observe how keys which are equal maintain their original relative orders of occurrence in the sorted list.

The worst case performance of insertion sort occurs when the elements in the list are already sorted in their descending order. It is easy to see that in such a case every key that is to be inserted has to move to the front of the list and therefore undertakes the maximum number of comparisons. Thus if the list  $L = \{K_1, K_2, K_3, \dots, K_n\}$ ,  $K_1 \geq K_2 \geq \dots \geq K_n$  is to be insertion sorted then the number of comparisons for the insertion of key  $K_i$  would be  $(i-1)$  since  $K_i$  would swap positions with each of the  $(i-1)$  keys occurring before it until it moves to position 1. Therefore the total number of comparisons for inserting each of the keys is given by

$$1 + 2 + 3 + \dots + (n-1) = \frac{(n-1)n}{2} = O(n^2)$$

The best case complexity of insertion sort arises when the list is already sorted in the ascending order. In such a case the complexity in terms of comparisons is given by  $O(n)$ .

The average case performance of insertion sort reports  $O(n^2)$  complexity.

## Selection Sort

16.4

Selection sort is built on the principle of repeated *selection* of elements satisfying a specific criterion to aid the sorting process.

The steps involved in the sorting process are listed below:

- (i) Given an unordered list  $L = \{K_1, K_2, K_3, \dots, K_j, \dots, K_n\}$ , select the minimum key  $K$
- (ii) Swap  $K$  with the element in the first position of the list  $L$ , viz.,  $K_1$ . By doing so the minimum element of the list has secured its rightful position of number one in the sorted list. This step is termed pass 1.
- (iii) Exclude the first element and select the minimum element  $K$ , from amongst the remaining elements of the list  $L$ . Swap  $K$  with the element in the second position of the list viz.,  $K_2$ . This is termed pass 2.
- (iv) Exclude the first two elements which have occupied their rightful positions in the sorted list  $L$ . Repeat the process of selecting the next minimum element and swapping it with the appropriate element, until the entire list  $L$  gets sorted in the ascending order. The entire sorting gets done in  $(n-1)$  passes.

Selection sort can also undertake sorting in the descending order by selecting the *maximum element* instead of the minimum element and swapping it with the element in the *last position* of the list  $L$ .

Algorithm 16.3 illustrates the working of selection sort. The procedure `FIND_MINIMUM(L, i, n)` selects the minimum element from the array  $L[i:n]$  and returns the position index of the minimum element to procedure `SELECTION_SORT`. The `for` loop in the `SELECTION_SORT` procedure represents the  $(n-1)$  passes needed to sort the array  $L[1:n]$  in the ascending order. Function `swap` swaps the elements input to it.

### Algorithm 16.3: Procedure for Selection sort

```

procedure SELECTION_SORT(L, n)
    /* L[1:n] is an unordered list of data elements to be
       sorted in the ascending order */
    for i = 1 to n-1 do                                /* n-1 passes*/
        minimum_index = FIND_MINIMUM(L, i, n);      /* find minimum element
                                                       of the list L[i:n] and store the position index of
                                                       the element in minimum_index */
        swap(L[i], L[minimum_index]);
    end
end SELECTION_SORT

```

```

procedure FIND_MINIMUM(L, i, n)
    /* the position index of the minimum element in the array
       L[i : n] is returned */
    min_indx = i;
    for j = i + 1 to n do
        if (L[j] < L[min_indx]) min_indx = j;
    end
    return (min_indx)
end FIND_MINIMUM

```

**Example 16.5** Let  $L = \{71, 17, 86, 100, 54, 27\}$  be an unordered list of elements. Each pass of selection sort is traced below. The minimum element is shown in bold and the arrows indicate the swap of the elements concerned. The elements in gray indicate their exclusion in the passes concerned.

Pass	List <i>L</i> (During Pass)	List <i>L</i> (After Pass)
1	{71, 17, 86, 100, 54, 27}	{17, 71, 86, 100, 54, 27}
2	{17, 71, 86, 100, 54, 27}	{17, 27, 86, 100, 54, 71}
3	{17, 27, 86, 100, 54, 71}	{17, 27, 54, 100, 86, 71}
4	{17, 27, 54, 100, 86, 71}	{17, 27, 54, 71, 86, 100}
5	{17, 27, 54, 71, 86, 100}	{17, 27, 54, 71, 86, 100} (Sorted list)

### Stability and performance analysis

Selection sort is not stable. Example 16.6 illustrates a case. The computationally expensive portion of selection sort occurs when the minimum element has to be selected in each pass. The time complexity of FIND\_MINIMUM procedure is  $O(n)$ . The time complexity of SELECTION\_SORT procedure is therefore  $O(n^2)$ .

**Example 16.6** Consider the list  $L = \{6^1, 6^2, 2\}$ . The repeating keys have been superscripted with numbers indicative of their relative orders of occurrence. A trace of the selection sort procedure is shown below. The minimum element is shown in bold and the swapping is indicated by the curved arrow. The elements excluded from the pass are shown in gray.

Pass	List <i>L</i> (During Pass)	List <i>L</i> (After Pass)
1	{ 6 <sup>1</sup> , 6 <sup>2</sup> , 2 }	{ 2, 6 <sup>2</sup> , 6 <sup>1</sup> }
2	{ 2, 6 <sup>2</sup> , 6 <sup>1</sup> }	{ 2, 6 <sup>2</sup> , 6 <sup>1</sup> } (Sorted list)

The selection sort on the given list  $L$  is therefore not stable.

## Merge Sort

## 16.5

*Merging* or *collating* is a process by which two ordered lists of elements are combined or merged into a single ordered list. *Merge sort* makes use of the principle of merge to sort an unordered list of elements and hence the name. In fact a variety of sorting algorithms belonging to the family of *sorting by merge* exist. Some of the well known external sorting algorithms belong to this class.

### Two-way Merging

Two-way merging deals with the merging of two ordered lists.

Let  $L_1 = \{a_1, a_2, \dots, a_i, \dots, a_n\}$  where  $a_1 \leq a_2 \leq \dots \leq a_i \leq \dots \leq a_n$  and  $L_2 = \{b_1, b_2, \dots, b_j, \dots, b_m\}$  where  $b_1 \leq b_2 \leq \dots \leq b_j \leq \dots \leq b_m$  be two ordered lists. Merging combines the two lists into a single list  $L$  by making use of the following cases of comparison between the keys  $a_i$  and  $b_j$  belonging to  $L_1$  and  $L_2$  respectively:

- A1.** If ( $a_i < b_j$ ) then drop  $a_i$  into the list  $L$
- A2.** If ( $a_i > b_j$ ) then drop  $b_j$  into the list  $L$
- A3.** If ( $a_i = b_j$ ) then drop both  $a_i$  and  $b_j$  into the list  $L$

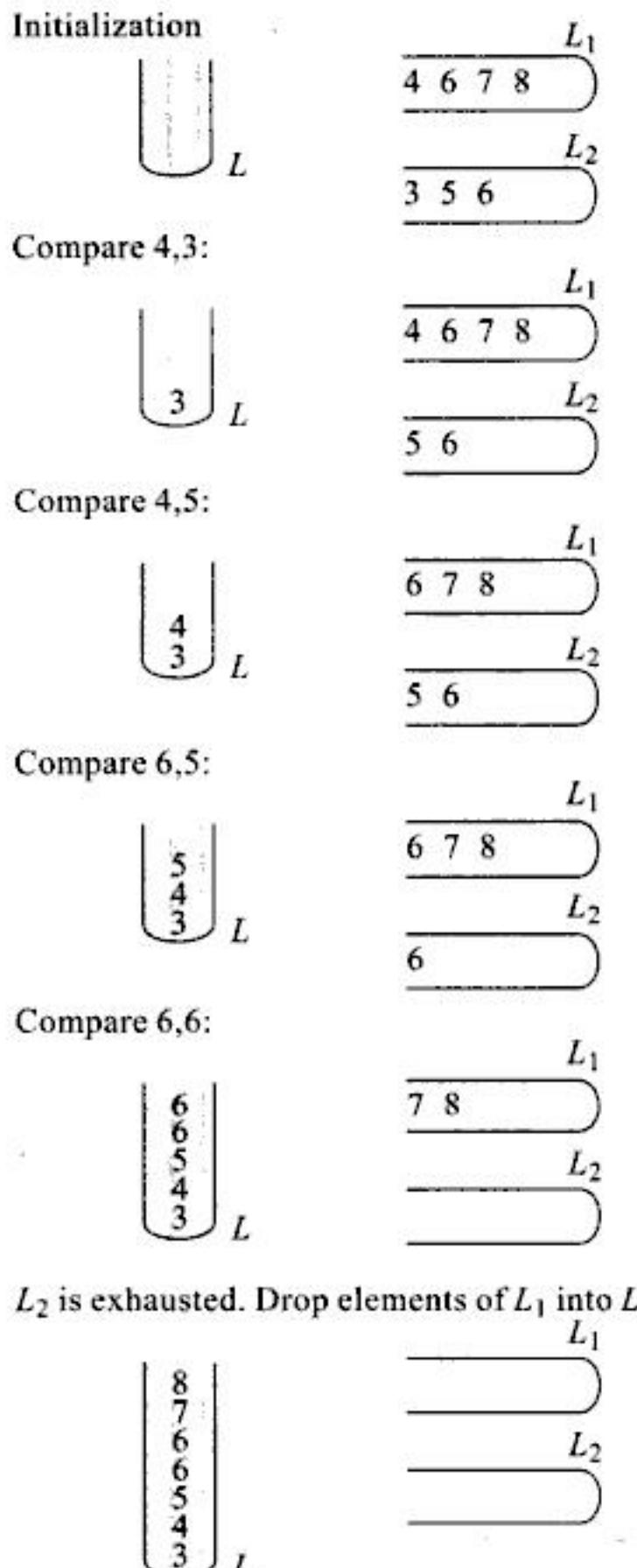
In the case of **A1**, once  $a_i$  is dropped into the list  $L$  the next comparison of  $b_j$  proceeds with  $a_{i+1}$ . In the case of **A2**, once  $b_j$  is dropped into the list  $L$  the next comparison of  $a_i$  proceeds with  $b_{j+1}$ . In the case of **A3** the next comparison proceeds with  $a_{i+1}$  and  $b_{j+1}$ . At the end of merge, list  $L$  contains  $(n+m)$  ordered elements.

The series of comparisons between pairs of elements from the lists  $L_1$  and  $L_2$  and the dropping of the relatively smaller elements into the list  $L$  proceeds until one of the following cases happens:

- B1.**  $L_1$  gets exhausted earlier to that of  $L_2$ . In such a case, the remaining elements in list  $L_2$  are dropped into the list  $L$  in the order of their occurrence in  $L_2$  and the merge is done.
- B2.**  $L_2$  gets exhausted earlier to that of  $L_1$ . In such a case the remaining elements in list  $L_1$  are dropped into the list  $L$  in the order of their occurrence in  $L_1$  and the merge is done.
- B3.** Both  $L_1$  and  $L_2$  are exhausted, in which case merge is done.

**Example 16.7** Consider the two ordered lists  $L_1 = \{4, 6, 7, 8\}$  and  $L_2 = \{3, 5, 6\}$ . Let us merge the two lists to get the ordered list  $L$ .  $L$  contains 7 elements in all. Figure 16.1 illustrates the snapshots of the merge process. Observe how when the elements 6, 6 are compared both the elements drop into the list  $L$ . Also note how list  $L_2$  gets exhausted earlier to  $L_1$  resulting in all the remaining elements of list  $L_1$  getting flushed into list  $L$ .

Algorithm 16.4 illustrates the procedure for merge. Here the two ordered lists to be merged are given as  $(x_1, x_2, \dots, x_t)$  and  $(x_{t+1}, x_{t+2}, \dots, x_n)$  to enable reuse of the algorithm for merge sort to be discussed subsequently. The input parameters to procedure MERGE is given as  $(x, \text{first}, \text{mid}, \text{last})$  where **first** is the starting index of the first list, **mid** the index related to the end/beginning of the first and second list respectively and **last** the ending index of the second list. The call to merge the two lists,  $(x_1, x_2, \dots, x_t)$  and  $(x_{t+1}, x_{t+2}, \dots, x_n)$  would be  $\text{MERGE}(x, 1, t, n)$ . While the first while loop in the procedure performs the pair wise comparison of elements in the two lists as discussed in cases **A1-A3**, the second while loop takes care of the case **B1** and the third loop that of the case **B2**. Case **B3** is inherently taken care of in the first while loop.

**Fig. 16.1** Two-way merge

## Performance analysis

The first while loop in Algorithm 16.4 executes at most  $(\text{last}-\text{first}+1)$  times and plays a significant role in the time complexity of the algorithm. The rest of the while loops only move the elements of the unexhausted lists into the list  $L$ . The complexity of the first while loop and hence the algorithm is given by  $O(\text{last}-\text{first}+1)$ . In the case of merging two lists  $(x_1, x_2, \dots, x_t), (x_{t+1}, x_{t+2}, \dots, x_n)$  where the number of elements in the two lists sums to  $n$ , the time complexity of MERGE is given by  $O(n)$ .

## *k-way merging*

The two-way merge principle could be extended to  $k$  ordered lists in which case it is termed as *k-way merging*. Here  $k$  ordered lists

$$L_1 = \{a_{11}, a_{12}, \dots, a_{1i} \dots a_{1n_1}\}, \quad a_{11} \leq a_{12} \leq \dots \leq a_{1i} \leq \dots a_{1n_1},$$

$$L_2 = \{a_{21}, a_{22}, \dots, a_{2i} \dots a_{2n_2}\}, \quad a_{21} \leq a_{22} \leq \dots \leq a_{2i} \leq \dots a_{2n_2}$$

$$L_k = \{a_{k1}, a_{k2}, \dots, a_{ki} \dots a_{kn_k}\}, \quad a_{k1} \leq a_{k2} \leq \dots \leq a_{ki} \leq \dots a_{kn_k}$$

each comprising  $n_1, n_2, \dots, n_k$  number of elements are merged into a single ordered list  $L$  comprising  $(n_1 + n_2 + \dots + n_k)$  number of elements. At every stage of comparison,  $k$  keys  $a_{ij}$ , one from each list, are compared before the smallest of the keys are dropped into the list  $L$ . Cases **A1-A3** and **B1-B3** discussed in Sec. 16.5 with regard to two-way merge, hold good in this case as well but as extended to  $k$  lists. Illustrative Problem 16.3 discusses an example *k-way merge*.

## Non recursive merge sort procedure

Given a list  $L = \{K_1, K_2, K_3, \dots, K_n\}$  of unordered elements, merge sort sorts the list making use of procedure MERGE repeatedly over several passes.

The non recursive version of merge sort merely treats the list  $L$  of  $n$  elements as  $n$  independent ordered lists of one element each. In pass one, the  $n$  singleton lists are pair wise merged. At the end of pass 1, the merged lists would have a size of 2 elements each. In pass 2, the lists of size 2 are pair wise merged to obtain ordered lists of size 4 and so on. In the  $i^{\text{th}}$  pass the lists of size  $2^{(i-1)}$  are merged to obtain ordered lists of size  $2^i$ .

During the passes, if any of the lists are unable to find a pair for their respective merge operation, then they are simply carried forward to the next pass.

**Example 16.8** Consider the list  $L = \{12, 56, 1, 34, 89, 78, 43, 10\}$  to be merge sorted using its non recursive formulation. Figure 16.2 illustrates the pair wise merging undertaken in each of the passes. The sublists in each pass are shown in brackets. Observe how pass 1 treats the list  $L$  as 8 ordered sublists of one element each and at the end of merge sort, pass 3 obtains a single list of size 8 which is the final sorted list.

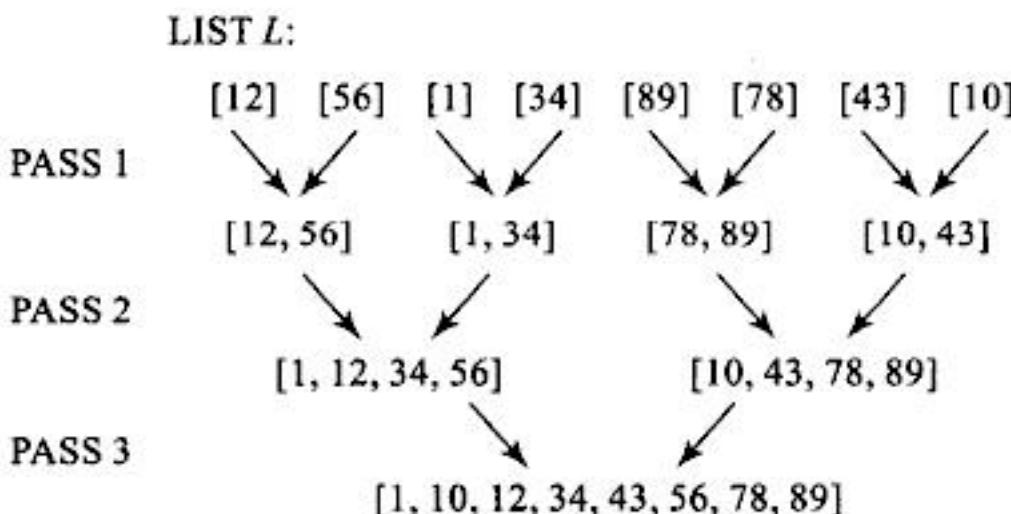


Fig. 16.2 Non recursive merge sort of list  $L = \{12, 56, 1, 34, 89, 78, 43, 10\}$  (Example 16.8)

**Algorithm 16.5:** Procedure for Recursive Merge Sort

```

procedure MERGE_SORT(a, first, last)
    /* a[first:last] is the unordered list of elements to be
       merge sorted. The call to the procedure to sort the
       list a[1:n] would be MERGE_SORT(a, 1, n) */

if (first < last) then
    {  $mid = \left\lfloor \frac{(first+last)}{2} \right\rfloor$ ;           /* divide the list into two sublists*/
      MERGE_SORT(a, first, mid); /* merge sort the sublist a[first,mid]*/
      MERGE_SORT(a, mid+1, last); /* merge sort the sublist a[mid+1, last]*/
      MERGE(a, first, mid, last); /* merge the two sublists a[first,mid] and
                                    a[mid+1, last]*/
    }
end MERGE SORT.

```

Example 16.10 illustrates shell sort on the given list  $L$  for an increment sequence  $\{8, 4, 2, 1\}$ .

**Example 16.10** Trace the shell sort procedure on the unordered list  $L$  of keys given by  $L = \{24, 37, 46, 11, 85, 47, 33, 66, 22, 84, 95, 55, 14, 09, 76, 35\}$  for an increment sequence  $\{h_3, h_2, h_1, h_0\} = \{8, 4, 2, 1\}$ .

The steps traced are shown in Fig. 16.4. Pass 1 for an increment 8, divides the unordered list  $L$  into 8 sublists each comprising 2 keys, that are 8 units apart. After each of the sublists have been individually insertion sorted, they are gathered together for the next pass.

In Pass 2, for an increment 4, the list gets divided into 4 groups, each comprising elements which are 4 units apart in the list  $L$ . The individual sub lists are again insertion sorted and gathered together for the next pass and so on, until in Pass 4 the entire list gets sorted for an increment 1.

The shell sort, in fact could work for any sequence of increments so long as  $h_0$  equals 1. Several empirical results and theoretical investigations have been undertaken regarding the conditions to be followed by the sequence of increments. Example 16.11 illustrates shell sort for the same list  $L$  used in Example 16.10 but for a different sequence of increments, viz.,  $\{7, 5, 3, 1\}$ .

**Example 16.11** Trace the shell sort procedure on the unordered list  $L$  of keys given by  $L = \{24, 37, 46, 11, 85, 47, 33, 66, 22, 84, 95, 55, 14, 09, 76, 35\}$  for an increment sequence  $\{h_3, h_2, h_1, h_0\} = \{7, 5, 3, 1\}$ .

Figure 16.5 illustrates the steps involved in the sorting process. In Pass 1, the increment of 7 divides the sublist  $L$  into 7 groups of varying number of elements. The sub lists are insertion sorted and gathered for the next pass. In Pass 2, for an increment of 5, the list  $L$  gets divided into 5 groups of varying number of elements. As before they are insertion sorted and so on until in Pass 4 the entire list gets sorted for an increment of 1.

Algorithm 16.6 describes the skeletal shell sort procedure. The array  $L[1:n]$  represents the unordered list of keys,  $L = \{K_1, K_2, K_3, \dots, K_j, \dots, K_n\}$ .  $H$  is the sequence of increments  $\{h_t, h_{t-1}, h_{t-2}, \dots, h_2, h_1, h_0\}$ .

**Unordered list  $L$ :**

$K_1$	$K_2$	$K_3$	$K_4$	$K_5$	$K_6$	$K_7$	$K_8$	$K_9$	$K_{10}$	$K_{11}$	$K_{12}$	$K_{13}$	$K_{14}$	$K_{15}$	$K_{16}$
24	37	46	11	85	47	33	66	22	84	95	55	14	09	76	35

**Pass 1 (increment  $h_3 = 7$ )**

$(K_1 \ K_8 \ K_{15})$	$(K_2 \ K_9 \ K_{16})$	$(K_3 \ K_{10})$	$(K_4 \ K_{11})$	$(K_5 \ K_{12})$	$(K_6 \ K_{13})$	$(K_7 \ K_{14})$
$(24 \ 66 \ 76)$	$(37 \ 22 \ 35)$	$(46 \ 84)$	$(11 \ 95)$	$(85 \ 55)$	$(47 \ 14)$	$(33 \ 09)$

After insertion sort:

$(24 \ 66 \ 76)$	$(22 \ 35 \ 37)$	$(46 \ 84)$	$(11 \ 95)$	$(55 \ 85)$	$(14 \ 47)$	$(09 \ 33)$
------------------	------------------	-------------	-------------	-------------	-------------	-------------

**List  $L$  after Pass 1:**

$K_1$	$K_2$	$K_3$	$K_4$	$K_5$	$K_6$	$K_7$	$K_8$	$K_9$	$K_{10}$	$K_{11}$	$K_{12}$	$K_{13}$	$K_{14}$	$K_{15}$	$K_{16}$
24	22	46	11	55	14	09	66	35	84	95	85	47	33	76	37

**Pass 2 (increment  $h_2 = 5$ )**

$(K_1 \ K_6 \ K_{11} \ K_{16})$	$(K_2 \ K_7 \ K_{12})$	$(K_3 \ K_8 \ K_{13})$	$(K_4 \ K_9 \ K_{14})$	$(K_5 \ K_{10} \ K_{15})$
$(24 \ 14 \ 95 \ 37)$	$(22 \ 09 \ 85)$	$(46 \ 66 \ 47)$	$(11 \ 35 \ 33)$	$(55 \ 84 \ 76)$

After insertion sort:

$(14 \ 24 \ 37 \ 95)$	$(09 \ 22 \ 85)$	$(46 \ 47 \ 66)$	$(11 \ 33 \ 35)$	$(55 \ 76 \ 84)$
-----------------------	------------------	------------------	------------------	------------------

**List  $L$  after Pass 3:**

$K_1$	$K_2$	$K_3$	$K_4$	$K_5$	$K_6$	$K_7$	$K_8$	$K_9$	$K_{10}$	$K_{11}$	$K_{12}$	$K_{13}$	$K_{14}$	$K_{15}$	$K_{16}$
14	09	46	11	55	24	22	47	33	76	37	85	66	35	84	95

**Pass 3 (increment  $h_1 = 3$ )**

$(K_1 \ K_4 \ K_7 \ K_{10} \ K_{13} \ K_{16})$	$(K_2 \ K_5 \ K_8 \ K_{11} \ K_{14})$	$(K_3 \ K_6 \ K_9 \ K_{12} \ K_{15})$
$(14 \ 11 \ 22 \ 76 \ 66 \ 95)$	$(09 \ 55 \ 47 \ 37 \ 35)$	$(46 \ 24 \ 33 \ 85 \ 84)$

After insertion sort:

$(11 \ 14 \ 22 \ 66 \ 76 \ 95)$	$(09 \ 35 \ 37 \ 47 \ 55)$	$(24 \ 33 \ 46 \ 84 \ 85)$
---------------------------------	----------------------------	----------------------------

**List  $L$  after Pass 2**

$K_1$	$K_2$	$K_3$	$K_4$	$K_5$	$K_6$	$K_7$	$K_8$	$K_9$	$K_{10}$	$K_{11}$	$K_{12}$	$K_{13}$	$K_{14}$	$K_{15}$	$K_{16}$
11	09	24	14	35	33	22	37	46	66	47	84	76	55	85	95

**Pass 4 (increment  $h_0 = 1$ )**

$(K_1 \ K_2 \ K_3 \ K_4 \ K_5 \ K_6 \ K_7 \ K_8 \ K_9 \ K_{10} \ K_{11} \ K_{12} \ K_{13} \ K_{14} \ K_{15} \ K_{16})$
$(11 \ 09 \ 24 \ 14 \ 35 \ 33 \ 22 \ 37 \ 46 \ 66 \ 47 \ 84 \ 76 \ 55 \ 85 \ 95)$

After insertion sort:

$(09 \ 11 \ 14 \ 22 \ 24 \ 33 \ 35 \ 37 \ 46 \ 47 \ 55 \ 66 \ 76 \ 84 \ 85 \ 95)$
---

**Sorted List  $L$** 

$K_1$	$K_2$	$K_3$	$K_4$	$K_5$	$K_6$	$K_7$	$K_8$	$K_9$	$K_{10}$	$K_{11}$	$K_{12}$	$K_{13}$	$K_{14}$	$K_{15}$	$K_{16}$
09	11	14	22	24	33	35	37	46	47	55	66	76	84	85	95

**Fig. 16.5** Shell sorting of  $L = \{24, 37, 46, 11, 85, 47, 33, 66, 22, 84, 95, 55, 14, 09, 76, 35\}$  for the increment sequence  $\{7, 5, 3, 1\}$ .

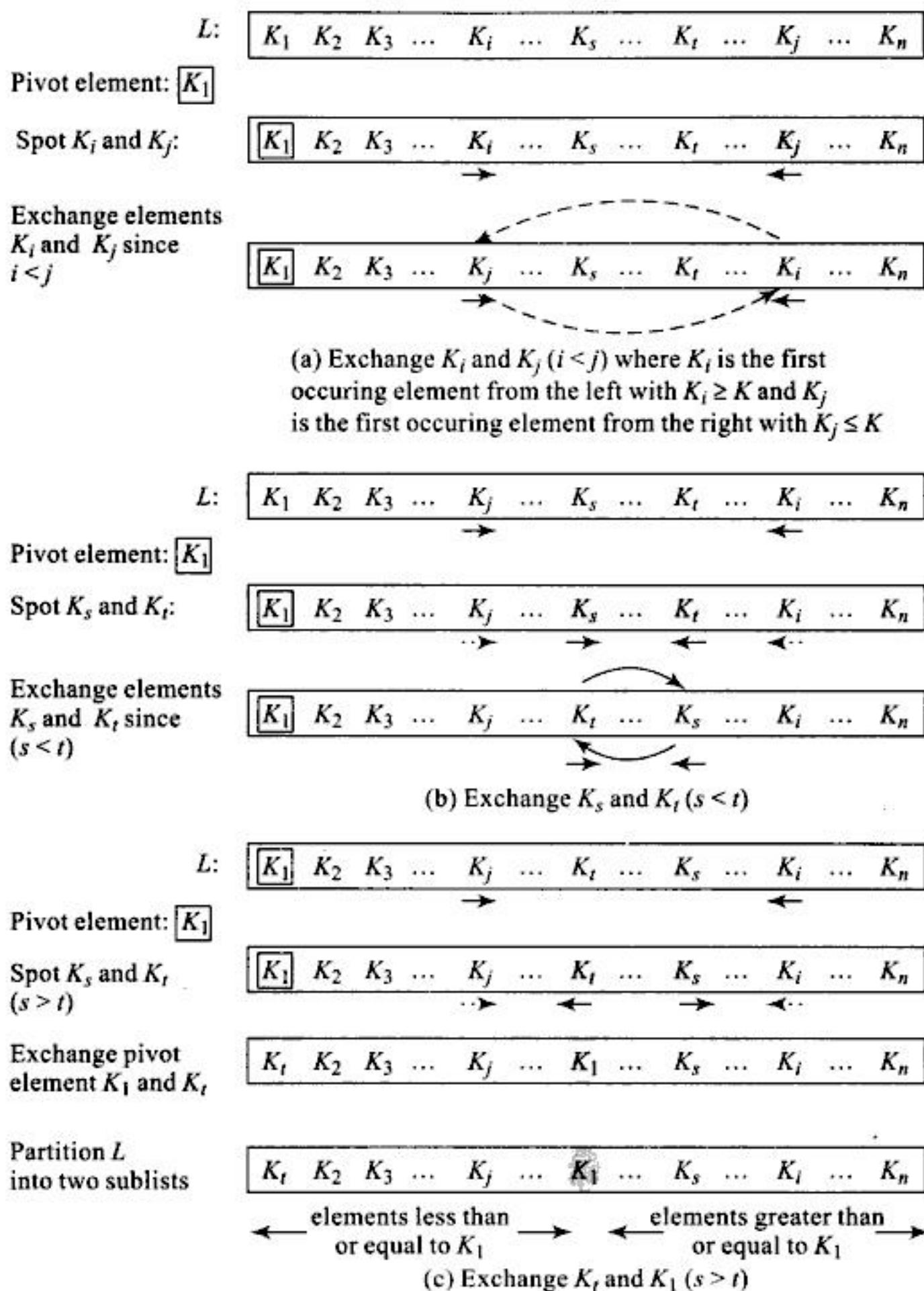


Fig. 16.6 Partitioning in Quick Sort

(Fig. 16.6(b)). If  $s > t$ , then  $K$  exchanges itself with  $K_t$  -the key which is smaller of  $K_s$  and  $K_t$ . At this stage a *partition* is said to occur. The pivot element  $K$  which has now exchanged position with  $K_t$  is the median around which the list partitions itself or splits itself into two. Figure 16.6(c) illustrates partition. Now what do we observe about the partitioned sublists and the pivot element?

- The sublist occurring to the left of the pivot element  $K$  (now at position  $t$ ) has all its elements less than or equal to  $K$  and the sublist occurring to the right of the pivot element  $K$  has all its elements greater than or equal to  $K$ .



every pass of the sort, the smallest or the largest key is selected by a well devised method and added to the output list and when all the elements have been selected the output list yields the sorted list.

Heap sort is built on a data structure called *heap* and hence the name heap sort. The heap data structure aids the selection of the largest (or smallest) key from the remaining elements of the list. Heap sort proceeds in two phases viz.,

- construction of a heap where the unordered list of elements to be sorted are converted into a heap, and
- repeated selection and inclusion of the root node key of the heap into the output list after reconstructing the remaining tree into a heap.

## Heap

A heap is a complete binary tree in which each parent node  $u$  labeled by a key or element  $e(u)$  and its respective child nodes  $v, w$  labeled  $e(v), e(w)$  respectively are such that  $e(u) \geq e(v)$  and  $e(u) \geq e(w)$ . Since the parent node keys are greater than or equal to their respective child node keys at each level, the key at the root node would turn out to be the largest amongst all the keys represented as a heap.

It is also possible to define the heap such that the root holds the smallest key for which every parent node key should be less than or equal to that of its child nodes. However, by convention a heap sticks to the principle of the root holding the largest element.

**Example 16.15** The binary tree shown in Fig. 16.10(a) is a heap while that shown in Fig. 16.10(b) is not.

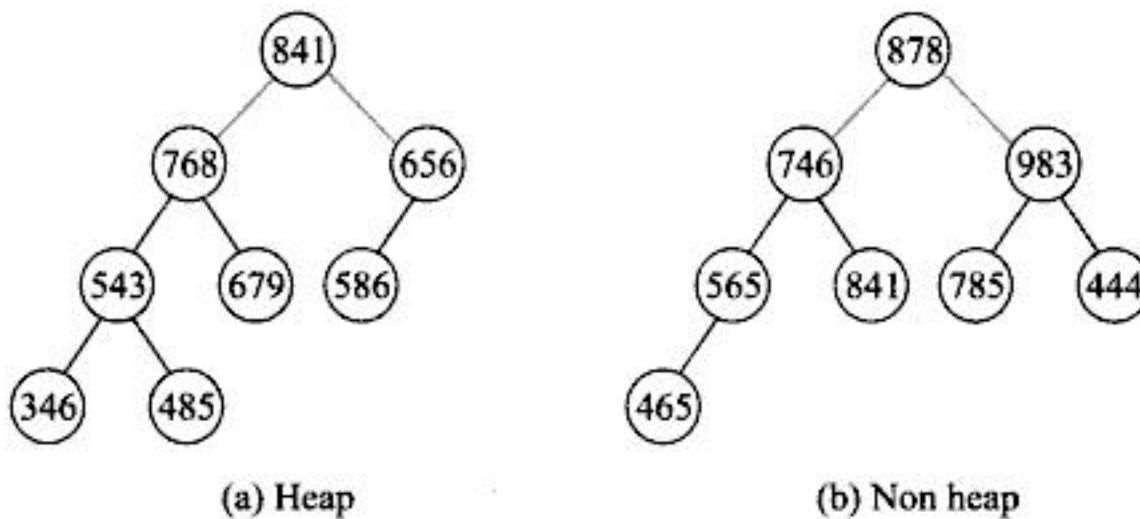


Fig. 16.10 An example heap and non heap

It may be observed in Fig. 16.10(a) how each parent node key is greater than or equal to that of its child node keys. As a result the root represents the largest key in the heap. In contrast the non heap shown in Fig. 16.10(b) violates the above characteristics.

## Construction of heap

Given an unordered list of elements it is essential that a heap is first constructed before heap sort works on it to yield the sorted list. Let  $L = \{K_1, K_2, K_3, \dots, K_n\}$  be the unordered list. The construction of the heap proceeds by inserting keys from  $L$  one by one into an existing heap.

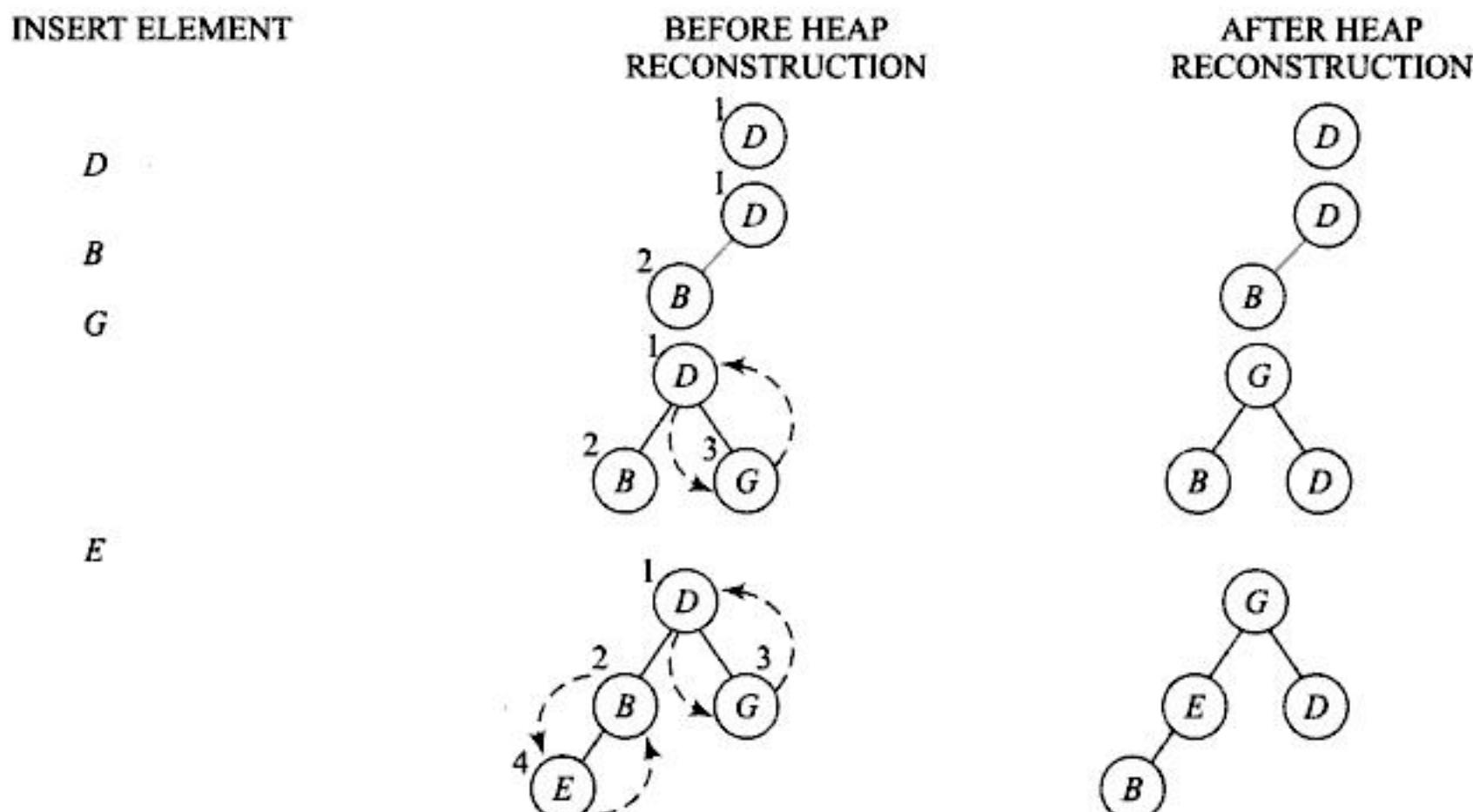
$K_1$  is inserted into the initially empty heap as its root.  $K_2$  is inserted as the left child of  $K_1$ . If the property of heap is violated then  $K_1$  and  $K_2$  swap positions to construct a heap out of themselves. Next  $K_3$  is inserted as the right child of node  $K_1$ . If  $K_3$  violates the property of heap it swaps position with its parent  $K_1$  and so on.

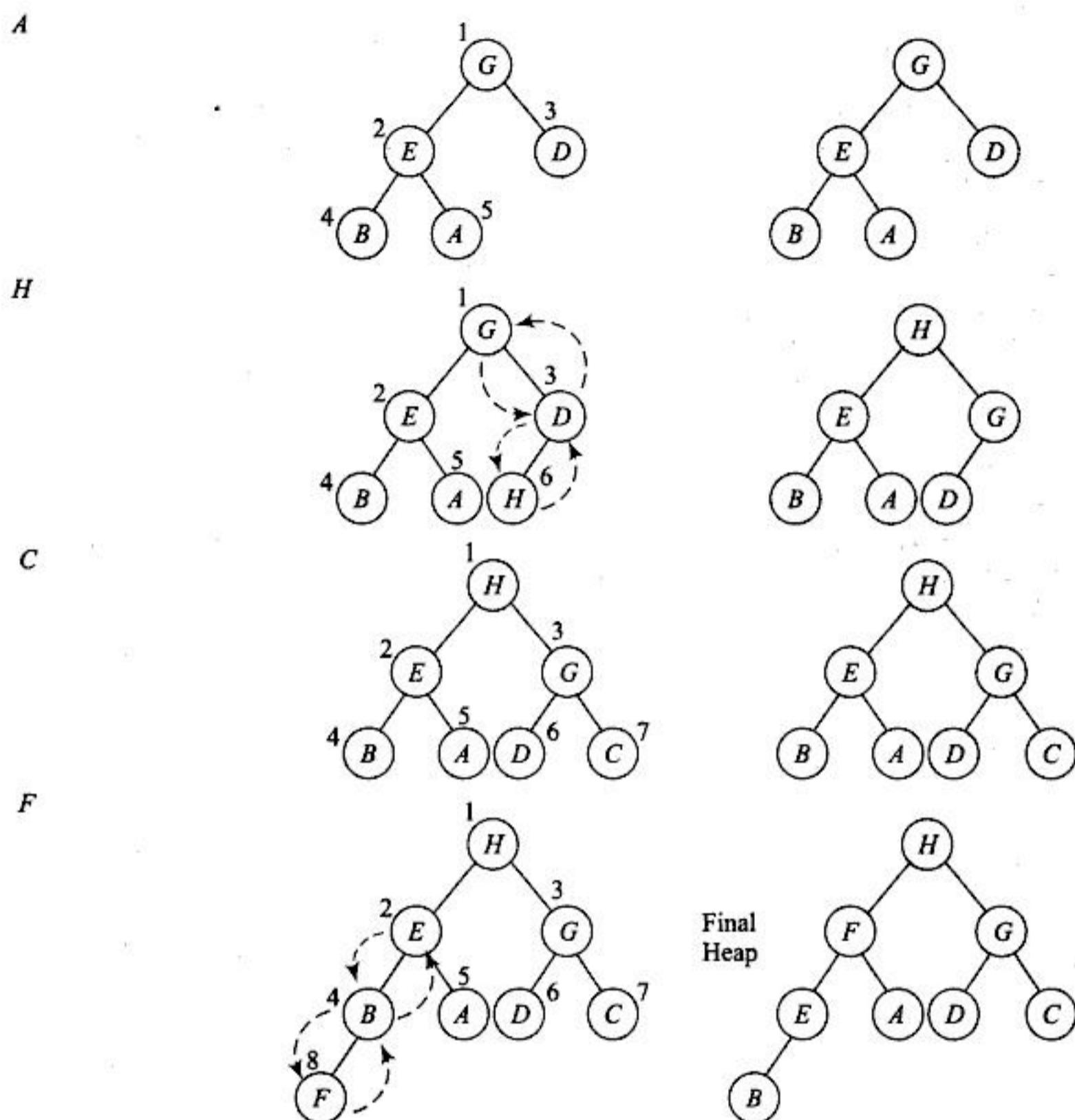
In general, a key  $K_i$  is inserted into the heap as the child of node  $\left\lfloor \frac{i}{2} \right\rfloor$  following the principle of complete binary tree (the parent of child  $i$  is given by  $\left\lfloor \frac{i}{2} \right\rfloor$  and the right and left child of  $i$  is given by  $2i$  and  $(2i+1)$  respectively). If the property of the heap is violated then it calls for a swap between  $K_i$  and  $K_{\left\lfloor \frac{i}{2} \right\rfloor}$  which in turn may trigger further adjustments between  $K_{\left\lfloor \frac{i}{2} \right\rfloor}$  and its parent and so on. In short a major adjustment across the tree may have to be carried out to reconstruct the heap.

Though a heap is a binary tree, the principle of complete binary tree which it follows favors its representation as an array (see Sec. 8.5). The algorithms pertaining to heap and heap sort employ arrays for their implementation of heaps.

**Example 16.16** Let us construct a heap out of  $L = \{D, B, G, E, A, H, C, F\}$ . Figure 16.11 illustrates the step by step process of insertion and heap reconstruction before the final heap is obtained. The adjustments made between the keys of the node during the heap reconstruction are shown in dotted lines.

List  $L$ :  $\{D \ 2 \ B \ 3 \ G \ 4 \ E \ 5 \ A \ 6 \ H \ 7 \ C \ 8 \ F \ 8\}$





**Fig. 16.11** Construction of heap (Example 16.16)

As mentioned earlier, for the implementation of the algorithm for the construction of a heap, it is convenient make use of an array representation. Thus if the list  $L = \{D, B, G, E, A, H, C, F\}$  shown in Example 16.16 is represented as an array then the same after construction of the heap would be as shown in Fig. 16.12. Algorithm 16.9 illustrates the procedure for inserting a key  $K$  ( $L[\text{child\_index}]$ ) into an existing heap  $L[1:\text{child\_index}-1]$ .

List $L$ as an array $L[1 : 8]$ before heap construction	$L$ $\dots   D   B   G   E   A   H   C   F   \dots$ [1] [2] [3] [4] [5] [6] [7] [8]
List $L$ as an array $L[1 : 8]$ after heap construction	$L$ $\dots   H   F   G   E   A   D   C   B   \dots$ [1] [2] [3] [4] [5] [6] [7] [8]

**Fig. 16.12** Array representation of a heap for the list  $L = \{D, B, G, E, A, H, C, F\}$

**Algorithm 16.9:** Procedure for inserting a key into a heap

```

procedure INSERT_HEAP(L, child_index)
    /* L[1:child_index-1] is an existing heap into which
       L[child_index] is to be included*/
heap = false;
parent_index =  $\left\lfloor \frac{\text{child\_index}}{2} \right\rfloor$ ; /* identify parent*/
while (not heap) and (child_index > 1) do
    if (L[parent_index] < L[child_index]) then /* heap property
        violated- swap
        parent and child*/
        { swap(L[parent_index], L[child_index]);
        child_index = parent_index;
        parent_index =  $\left\lfloor \frac{\text{child\_index}}{2} \right\rfloor$ ;
        }
    else
    {
        heap = true;
    }
end
end INSERT_HEAP.
```

To build a heap out of a list  $L[1 : n]$ , each element beginning from  $L[2]$  to  $L[n]$  will have to be inserted one by one into the constructed heap. Algorithm 16.10 illustrates the procedure of constructing a heap out of  $L[1 : n]$ . Illustrative Problem 16.8 illustrates the trace of the algorithm for the construction of a heap given a list of elements.

**Algorithm 16.10:** Procedure for construction of heap

```

procedure CONSTRUCT_HEAP(L, n)
    /* L[1:n] is a list to be constructed into a heap*/
    for child_index = 2 to n do
        INSERT_HEAP(L, child_index); /* insert elements one by one
            into the heap*/
    end
end CONSTRUCT_HEAP.
```

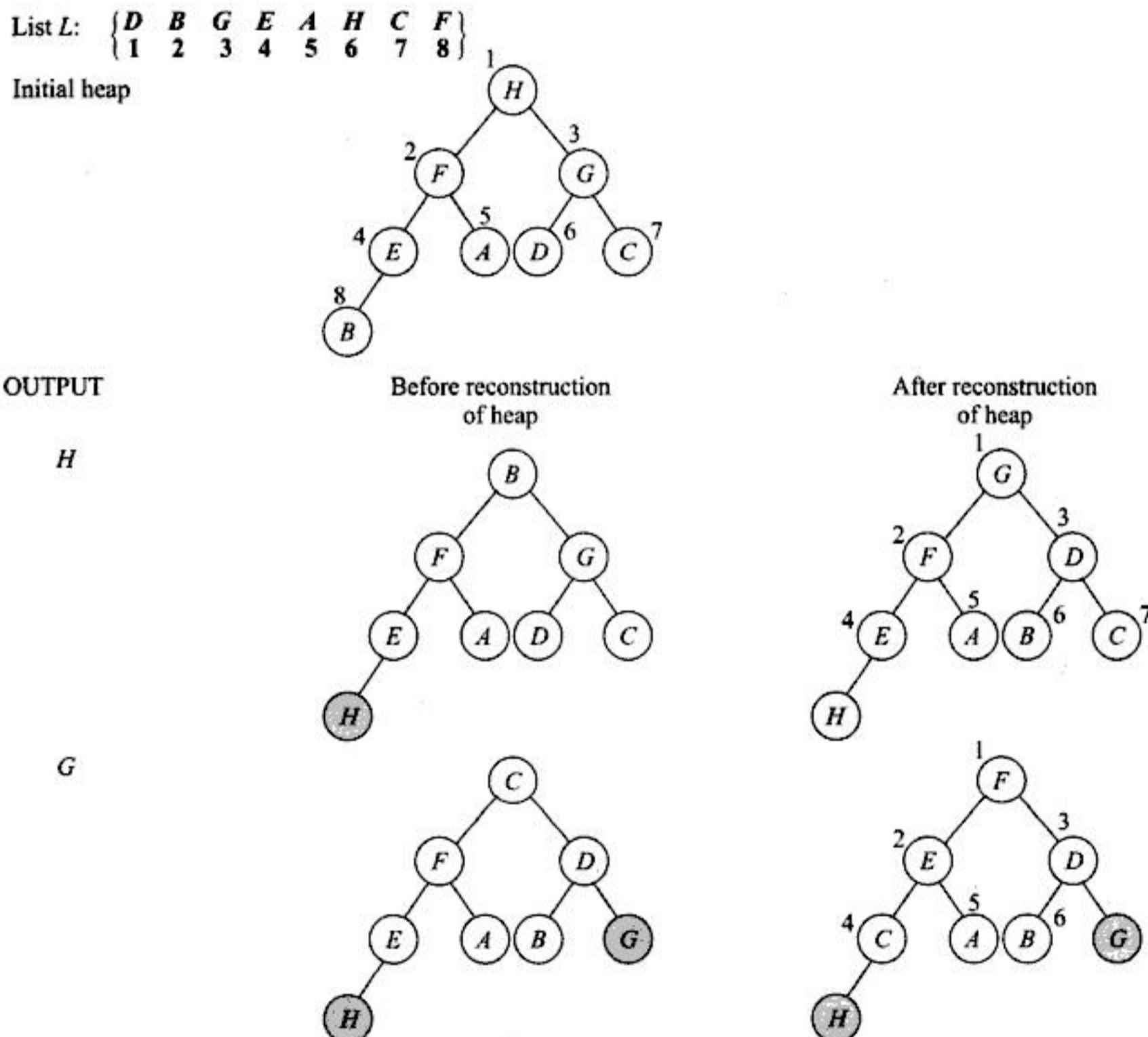
**Heap sort procedure**

To sort an unordered list  $L = \{K_1, K_2, K_3, \dots, K_n\}$ , heap sort procedure first constructs a heap out of  $L$ . The root which holds the largest element of  $L$  swaps places with the *largest numbered* node of the tree. The largest numbered node is now disabled from further participation in the heap reconstruction process. This is akin to the highest key of the list getting included in the output list. Now the remaining tree with  $(n-1)$  active nodes is again reconstructed to form a heap. The root node now holds the next largest element of the list. The swapping of the root node with the

next largest numbered node in the tree which is disabled thereafter, yields a tree with  $(n-2)$  active nodes and so on. This process of heap reconstruction and outputting the root node to the output list continues until the tree is left with no active nodes. At this stage heap sort is done and the output list contains the elements in the sorted order.

**Example 16.17** Let us heap sort the list  $L = \{D, B, G, E, A, H, C, F\}$  made use of in Example 16.16. The first phase of heap sort is to construct a heap out of the list. The heap constructed for the list  $L$  is shown in Fig. 16.11.

In the second stage the root node key is exchanged with the largest numbered node of the tree and the heap reconstruction of the remaining tree continues until the entire list is sorted. Figure 16.13 illustrates the second stage of heap sort. The disabled nodes of the tree are shown shaded in grey. After reconstruction of the heap the nodes are numbered to indicate the largest numbered node that is to be swapped with the root of the heap. The sorted list is obtained as  $L = \{A, B, C, D, E, F, G, H\}$ .





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



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



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



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



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



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



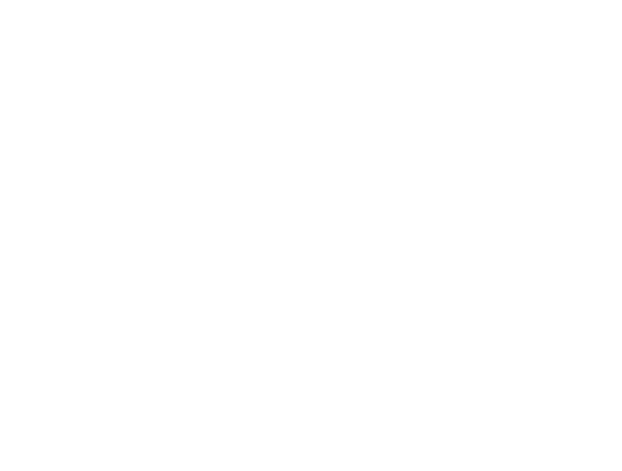
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



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



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



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



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



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



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



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



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



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



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



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



You have either reached a page that is unavailable for viewing or reached your 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
- Enqueueing 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 22
- 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  
 R0 rotation 240  
 R1 rotation 241  
 Radix sort 422  
 Random access storage devices 353  
 probing 339  
 Rb0, Rb1 304  
 Rb2 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  
 Rr0, Rr1 304  
 Rr2 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

# DATA STRUCTURES AND ALGORITHMS

This text details concepts, techniques, and applications pertaining to the subject in a lucid style. Independent of any programming language, the text discusses several illustrative problems to reinforce the understanding of the theory. It offers a plethora of programming assignments and problems to aid implementation of Data Structures.

## **Salient features:**

- Example driven approach employed, where introduction to the topic is followed by solved examples and algorithms
- A unique feature, whereby ADT for each Data Structure has been discussed in a separate section at the end of every chapter
- Exhaustive coverage on Binary Search Trees, AVL Trees, B-Trees and Tries, Red Black Trees and Splay Trees
- The use of Pseudocode provides flexibility in terms of language of implementation
- A dedicated website offers a number of tools including C Program implementation of most algorithms in the text

**URL:** <http://www.mhhe.com/pai/dsa>

- Exhaustive pedagogical features
  - ⇒ 124 Solved examples
  - ⇒ 215 Review questions
  - ⇒ 133 Illustrative Problems
  - ⇒ 74 Programming Assignments

Visit us at : [www.tatamcgrawhill.com](http://www.tatamcgrawhill.com)

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

ISBN-10: 0-07-066726-8



9 780070 667266



**Tata McGraw-Hill**

Copyrighted material