

# 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>
4.1 Introduction	39
4.2 Stack Operations	40
4.3 Applications	43
Summary	48
Illustrative Problems	49
Review Questions	54
Programming Assignments	55
<b>5. Queues</b>	<b>56</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	81
Programming Assignments	82
<b>Part II</b>	
<b>6. Linked Lists</b>	<b>84</b>
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	121
<b>7. Linked Stacks and Linked Queues</b>	<b>123</b>
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>	<b>151</b>
8.1 Introduction	151

8.2	Trees: Definition and Basic Terminologies	151
8.3	Representation of Trees	153
8.4	Binary Trees: Basic Terminologies and Types	155
8.5	Representation of Binary Trees	156
8.6	Binary Tree Traversals	158
8.7	Threaded Binary Trees	167
8.8	Application	169
	<i>Summary</i>	175
	<i>Illustrative Problems</i>	175
	<i>Review Questions</i>	184
	<i>Programming Assignments</i>	185
<b>9.</b>	<b>Graphs</b>	<b>186</b>
9.1	Introduction	186
9.2	Definitions and Basic Terminologies	187
9.3	Representations of Graphs	195
9.4	Graph Traversals	199
9.5	Applications	203
	<i>Summary</i>	209
	<i>Illustrative Problems</i>	209
	<i>Review Questions</i>	214
	<i>Programming Assignments</i>	216
<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</i> -way search trees: Definition and Operations	262
11.3	B Trees: Definition and Operations	269
11.4	Tries: Definition and Operations	277
11.5	Applications	281
	<i>Summary</i>	284
	<i>Illustrative Problems</i>	285
	<i>Review Questions</i>	290
	<i>Programming Assignments</i>	292
<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>	319
<i>Illustrative Problems</i>	319
<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>	346
<i>Illustrative Problems</i>	347
<i>Review Questions</i>	351
<i>Programming Assignments</i>	352
<b>14. File Organizations</b>	<b>353</b>
14.1 Introduction	353
14.2 Files	354
14.3 Keys	355
14.4 Basic File Operations	356
14.5 Heap or Pile Organization	356
14.6 Sequential File Organisation	357
14.7 Indexed Sequential File Organization	358
14.8 Direct File Organization	363
<i>Illustrative Problems</i>	365
<i>Summary</i>	369
<i>Review Questions</i>	370
<i>Programming Assignments</i>	371
<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
15.4 Interpolation Search	376
15.5 Binary Search	378
15.6 Fibonacci Search	381
15.7 Other Search Techniques	384
<i>Summary</i>	385
<i>Illustrative Problems</i>	386
<i>Review Questions</i>	391
<i>Programming Assignments</i>	393
<b>16. Internal Sorting</b>	<b>394</b>
16.1 Introduction	394

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

# Visual Walkthrough

	Contents
Part I	
4. Stacks and Queues	42
4.1 Stack Operations	42
4.2 Applications	43
Summary	43
Illustrative Problems	44
Review Questions	44
Programming Assignments	45
5. Queues	46
5.1 Introduction	46
5.2 Operations on Queues	47
5.3 Circular Queues	48
5.4 Other Types of Queues	49
5.5 Applications	50
Summary	50
Illustrative Problems	51
Review Questions	52
Programming Assignments	52
Part II	54
6. Linked Lists	64
6.1 Introduction	64
6.2 Singly Linked Lists	67
6.3 Circularly Linked Lists	68
6.4 Doubly Linked Lists	69
6.5 Multiply Linked Lists	70
6.6 Applications	70
Summary	72
Illustrative Problems	73
Review Questions	73
Programming Assignments	72
7. Linked Stacks and Linked Queues	123
7.1 Introduction	123
7.2 Operations on Linked Stacks and Linked Queues	124
7.3 Dynamic Memory Management and Linked Stacks	124
7.4 Implementation of Linked Representations	125
7.5 Applications	125
Summary	127
Illustrative Problems	127
Review Questions	128
Programming Assignments	128
Part III	151
8. Trees and Binary Trees	151
8.1 Introduction	151
8.2 Trees: Definition and Basic Terminologies	151
8.3 Representation of Trees	151

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**

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 merits of the sequential data structures before introducing the need for a linked representation. Next, the linked data structures of singly linked list, circularly linked list, doubly linked list and multiply linked list are elaborately presented. Finally, two problems, viz., Polynomial addition and Sparse matrix representation, demonstrating the application of linked lists are discussed.

**6.1 Introduction**

**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 minimise 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 Structures and Algorithms

### Illustrative Problems

**Problem 15.1** For the list (NAMELE, I, KORN, 2288, 1000, 284, 1128, HIRSH, 5000, 1745, 8817, 1205, 2245, 1000, 284, 1128) trace through sequential search for the element HIRSH.

### 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 to obtain a single run at which stage the file is deemed sorted. The merge patterns called for by the strategies, are influenced by external storage medium on which the runs reside, viz., disks or tapes.
- Magnetic tapes are sequential devices built on the principle of audio tape devices. Data is stored in blocks occurring sequentially. Magnetic disks are random access storage devices. Data stored in a disk is addressed by its cylinder, track and sector numbers.
- Balanced merge sort is a technique that can be adopted on files residing on both disks and tapes. In its general form, a  $k$ -way merging could be undertaken during the runs. For the efficient management of merging runs, buffer handling and selection tree mechanisms are employed.
- Balanced  $k$ -way merge sort on tapes calls for the use of  $2k$  tapes for an efficient management of runs. Polyphase merge sort is a clever strategy that makes use of only  $(k+1)$  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)	(6)	(20)	(20)	(6)

(a) What is the storage space (in terms of bytes) needed to store the employee file in the disk?  
 (b) What is the storage space (in 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  
 (a) (i) true (ii) true (b) (i) true (ii) false (c) (i) false (ii) true (d) (i) false (ii) false
- 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 front and rear 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`  
`ENQUEUE(QUEUE, ITEM)`
- Delete element from queue `QUEUE` and output the element deleted in item  
`DEQUEUE(QUEUE, ITEM)`

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

Programming Assignments are given at the end of each chapter.

### Programming Assignment

- 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
  - the left subtree of `NODE` is empty, and
  - the left subtree of `NODE` is non-empty.

Internal Sorting 413

```

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 L[left] ≥ pivot_elt;
        repeat
            right = right-1; /* pivot element moves right to left*/
        until L[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  $[1, 5, 15, 26, 34, 76]$ .

```

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.

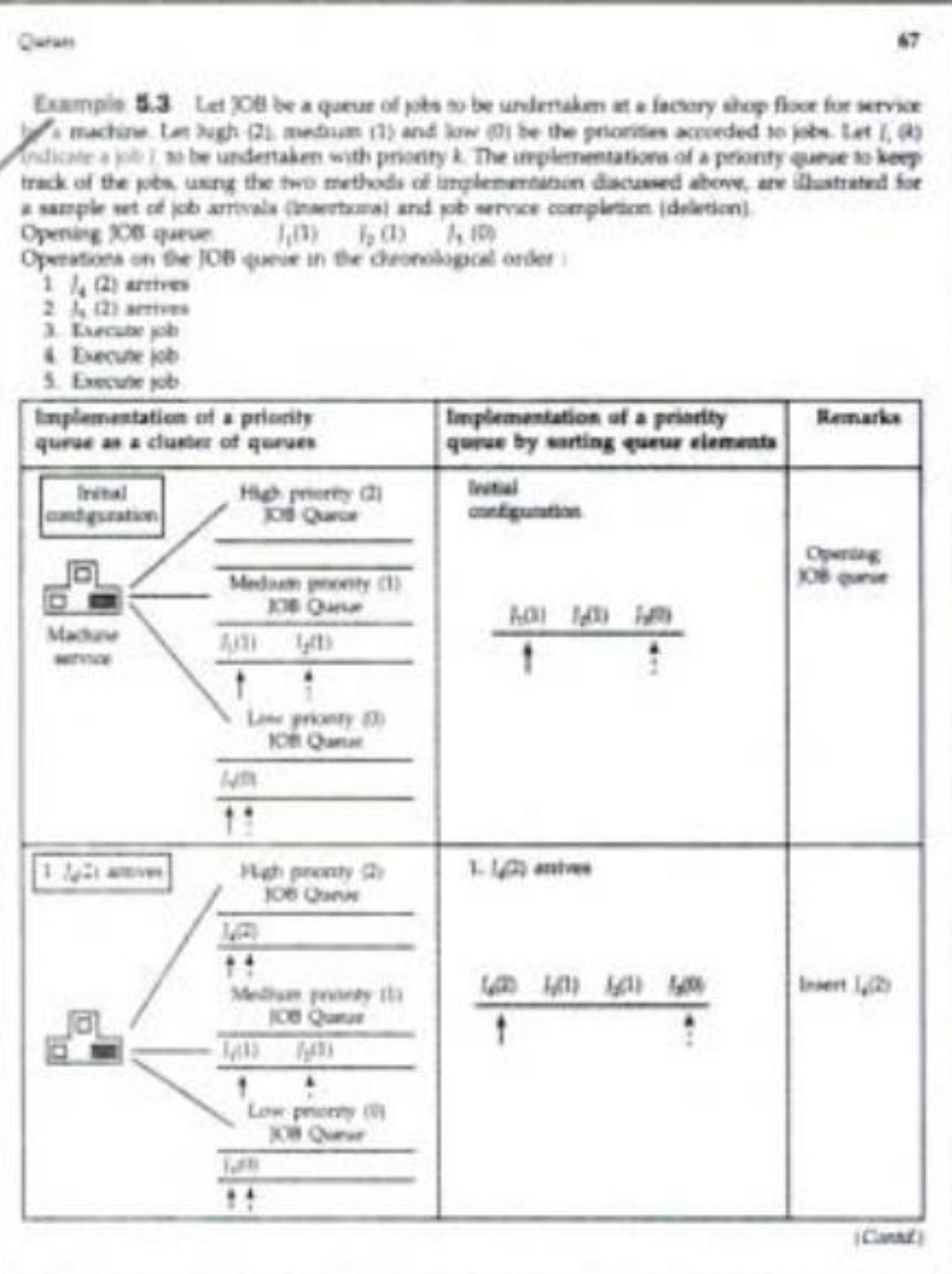
```

**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.



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.

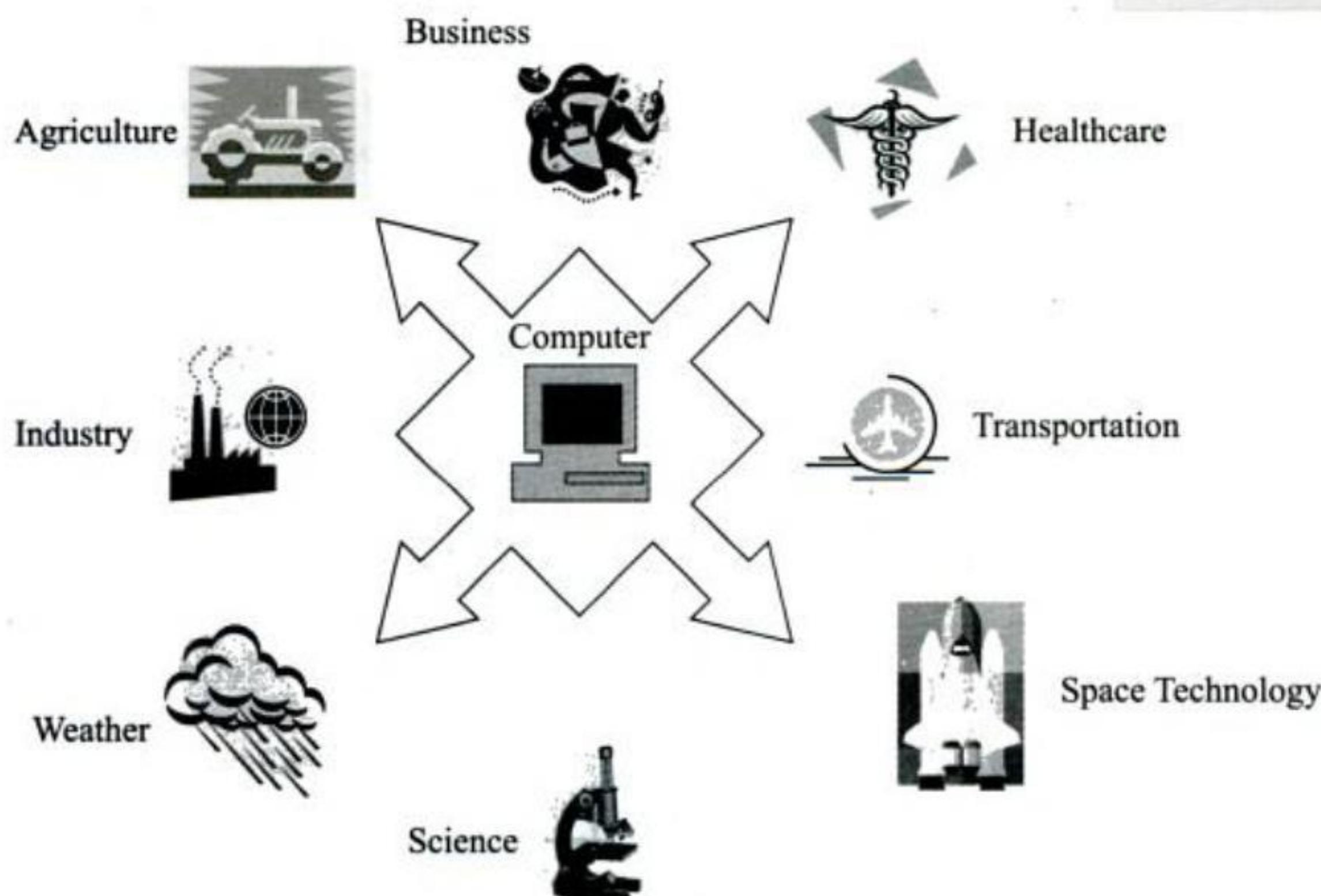


# INTRODUCTION

# 1

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:

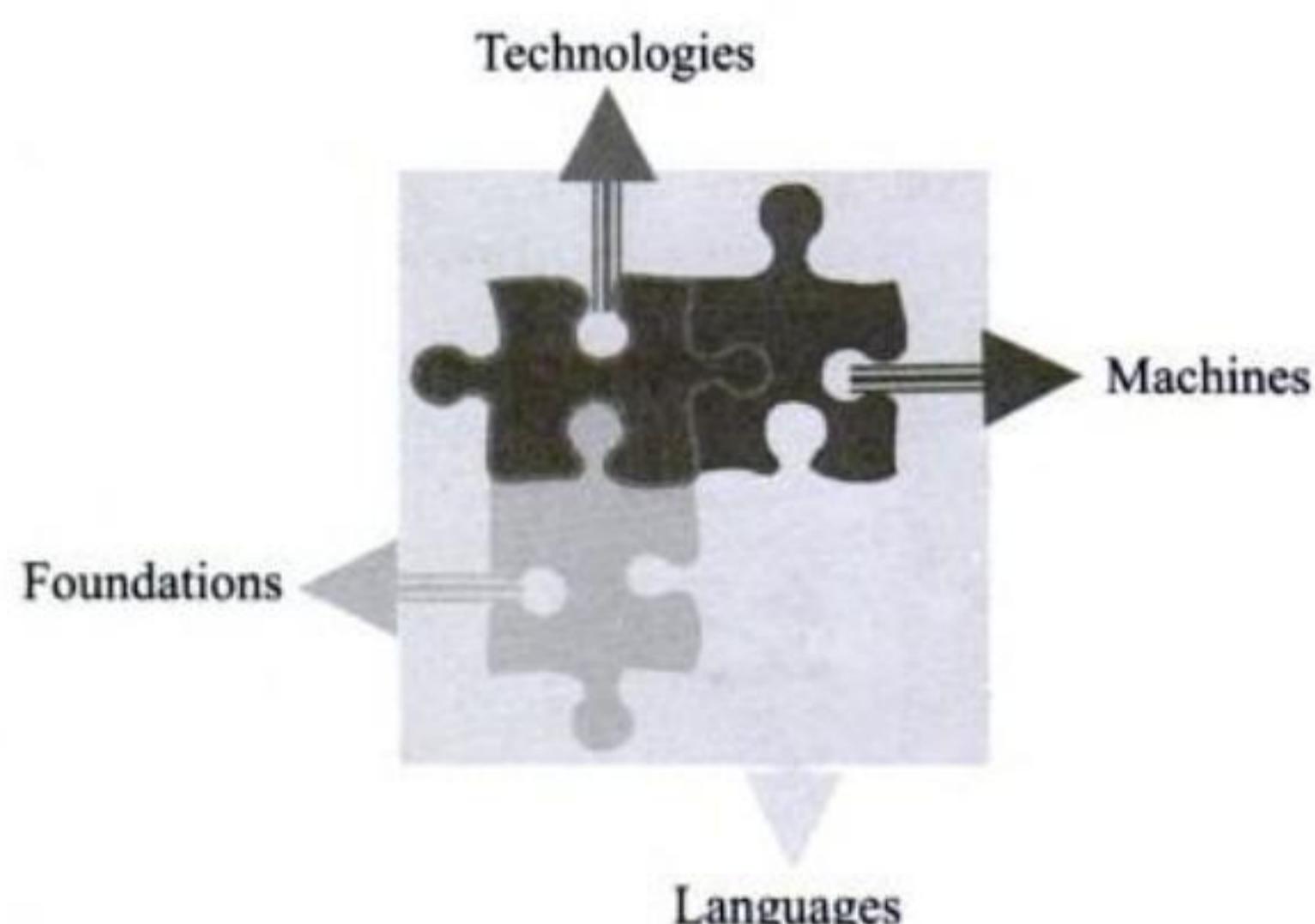
- *Machines* What machines are appropriate or available for the solution of a problem? What is the machine configuration – its processing power, memory capacity etc., that would be required for the efficient execution of the solution?
- *Languages* What is the language or software with which the solution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

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

## 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

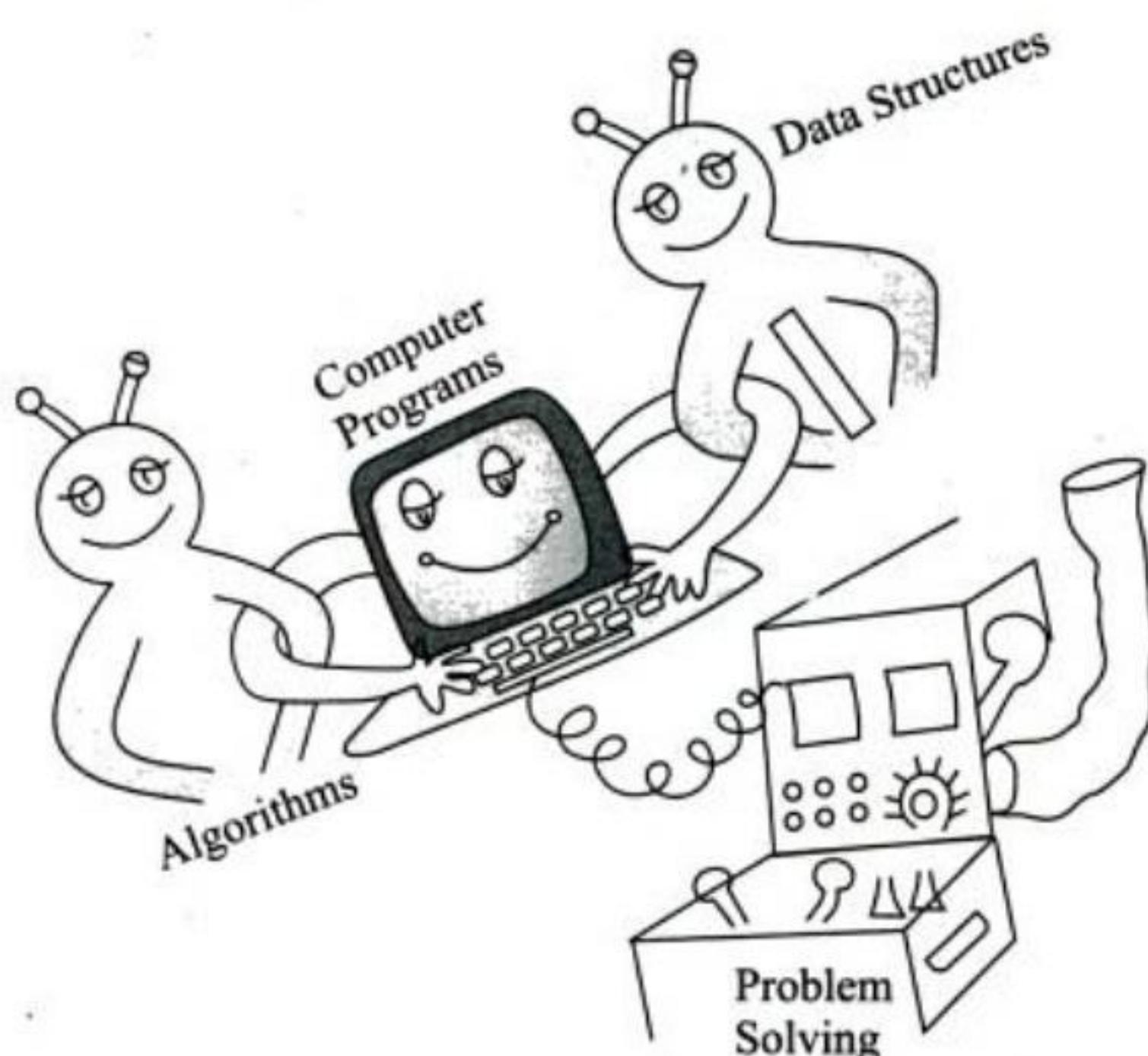
As was detailed in the previous section, the design of an *efficient* algorithm for the solution of the problem calls for the *inclusion of appropriate data structures*. A clear, unambiguous set of instructions following the properties of the algorithm alone does not contribute to the efficiency of the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

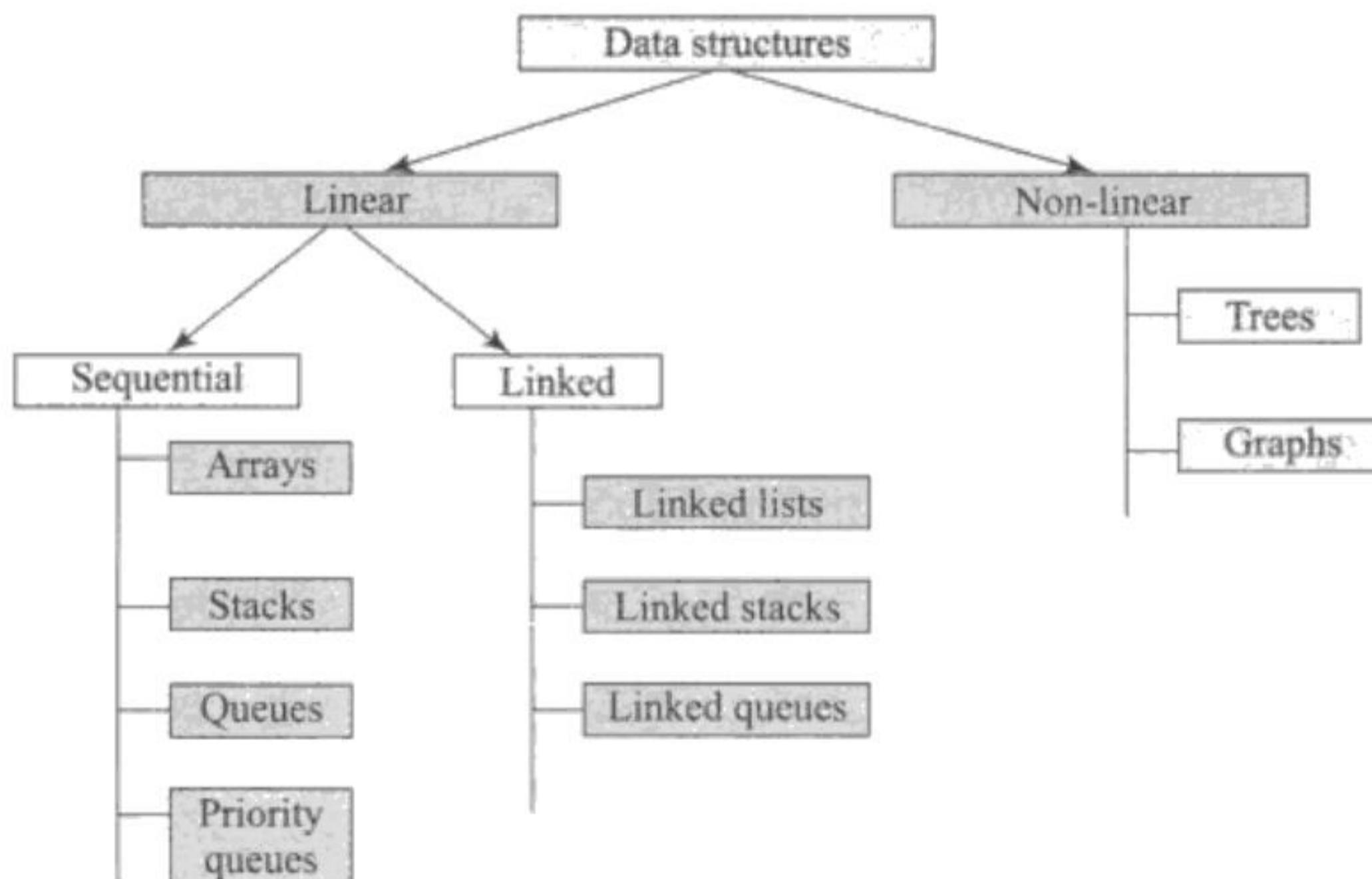
An ADT promotes ***data abstraction*** and focuses on *what* a data structure does rather than *how* it does. It is easier to comprehend a data structure by means of its ADT since it helps a designer to plan on the implementation of the data objects and its supportive operations in any programming language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

- 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

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.

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>	$(n + 1)$
$x = x + 2;$	$n$
end	$n$
...	
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>	$\sum_{j=1}^n (n + 1) = (n + 1)n$
$x = x + 2;$	$n^2$
end	$\sum_{j=1}^n n = n^2$
end	$n$
...	
Total frequency count	$3n^2 + 3n + 1$

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

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

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

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

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

## Asymptotic Notations

## 2.3

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

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

### Example

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

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

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

### Example

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

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

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

**Example**

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

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

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

**Example**

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

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

2.4

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

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

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

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

**Polynomial Vs Exponential Algorithms**

2.5

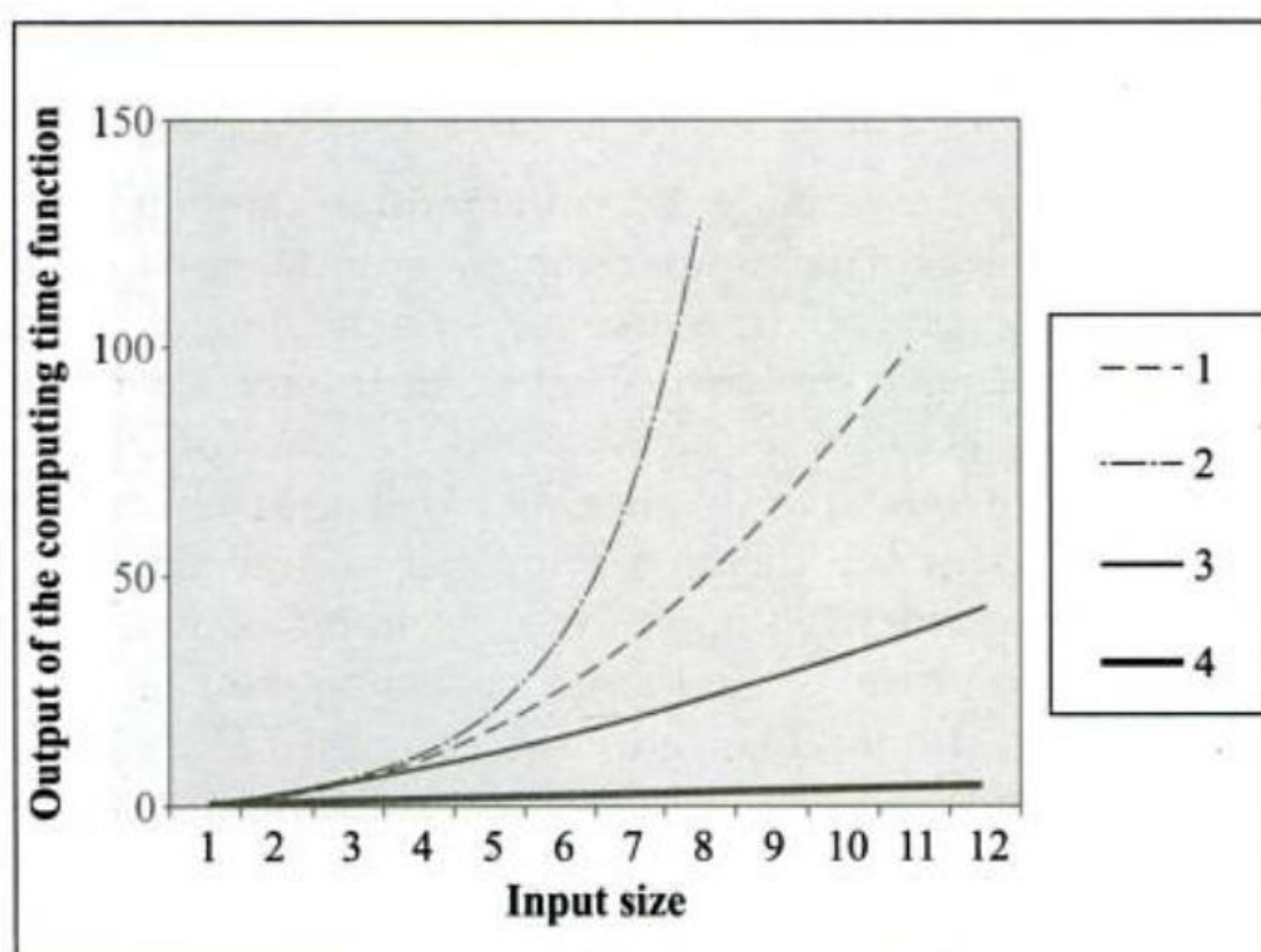
If  $n$  is the size of the input instance, then the number of operations for polynomial algorithms are of the form  $P(n)$  where  $P$  is a polynomial. In terms of  $O$  notation, polynomial algorithms have time complexities of the form  $O(n^k)$ , where  $k$  is a constant.

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

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

**Table 2.4 Comparison of polynomial and exponential algorithms**

Size	10	20	50
Time complexity function			
$n^2$	$10^{-4}$ sec	$4 \times 10^{-4}$ 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.

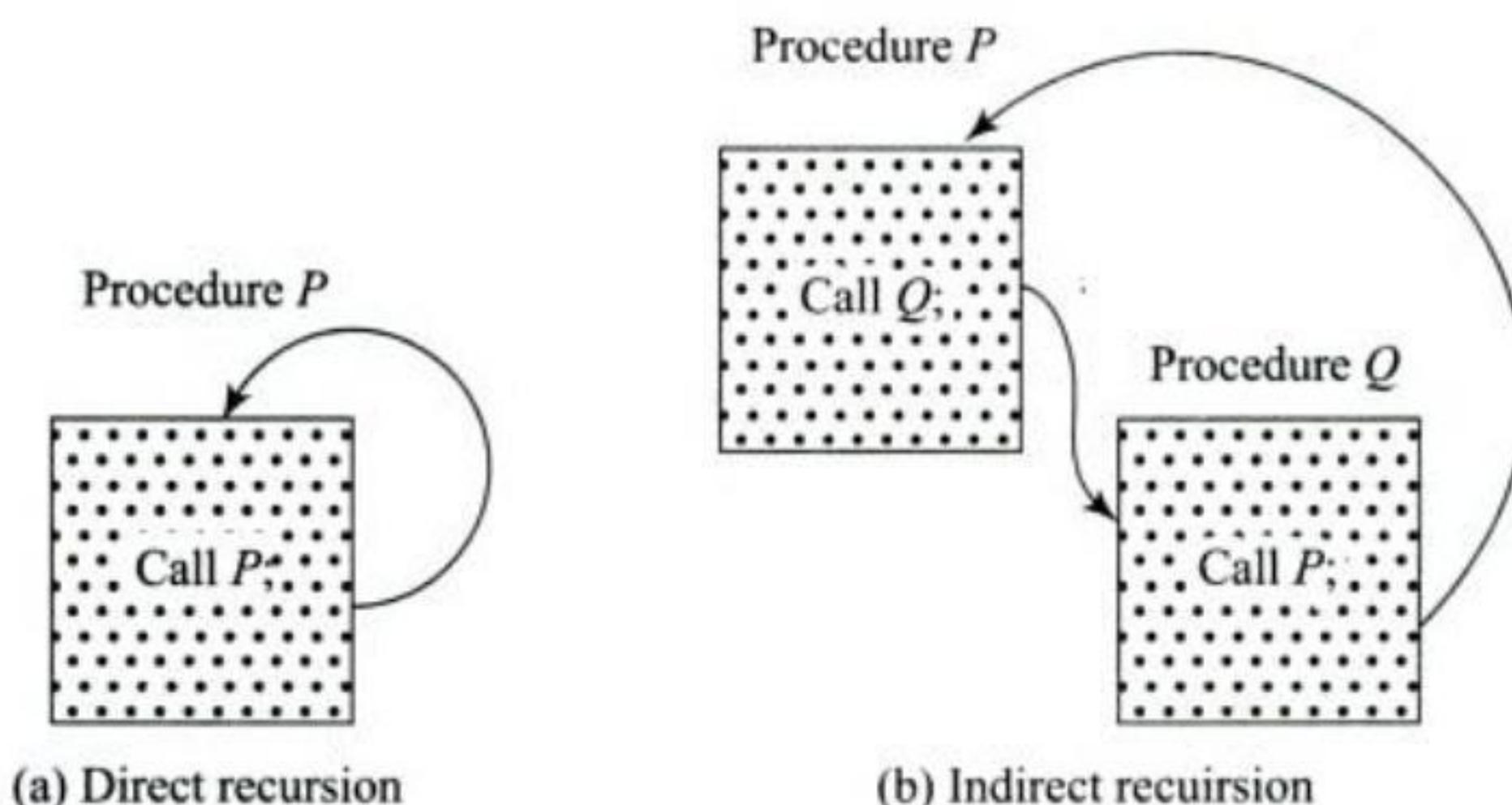
#### Recursive procedures

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:

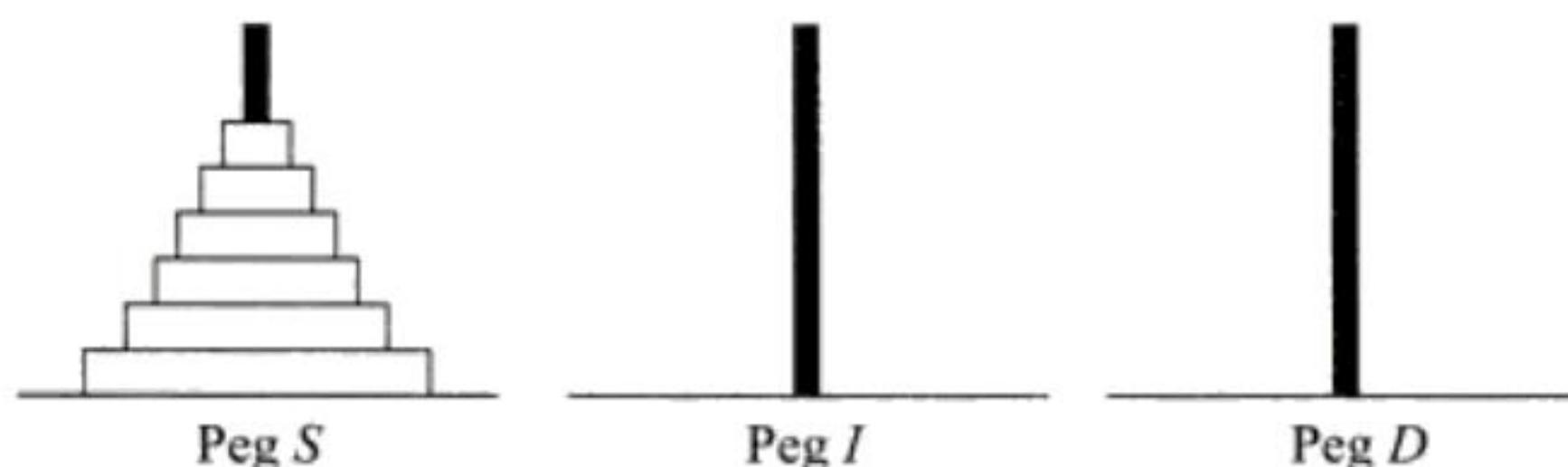
$$S(n) = \begin{cases} 0, & \text{if } n = 1 \text{ (base criterion)} \\ S(n/2) + 1, & \text{if } n > 1 \end{cases}$$

**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*

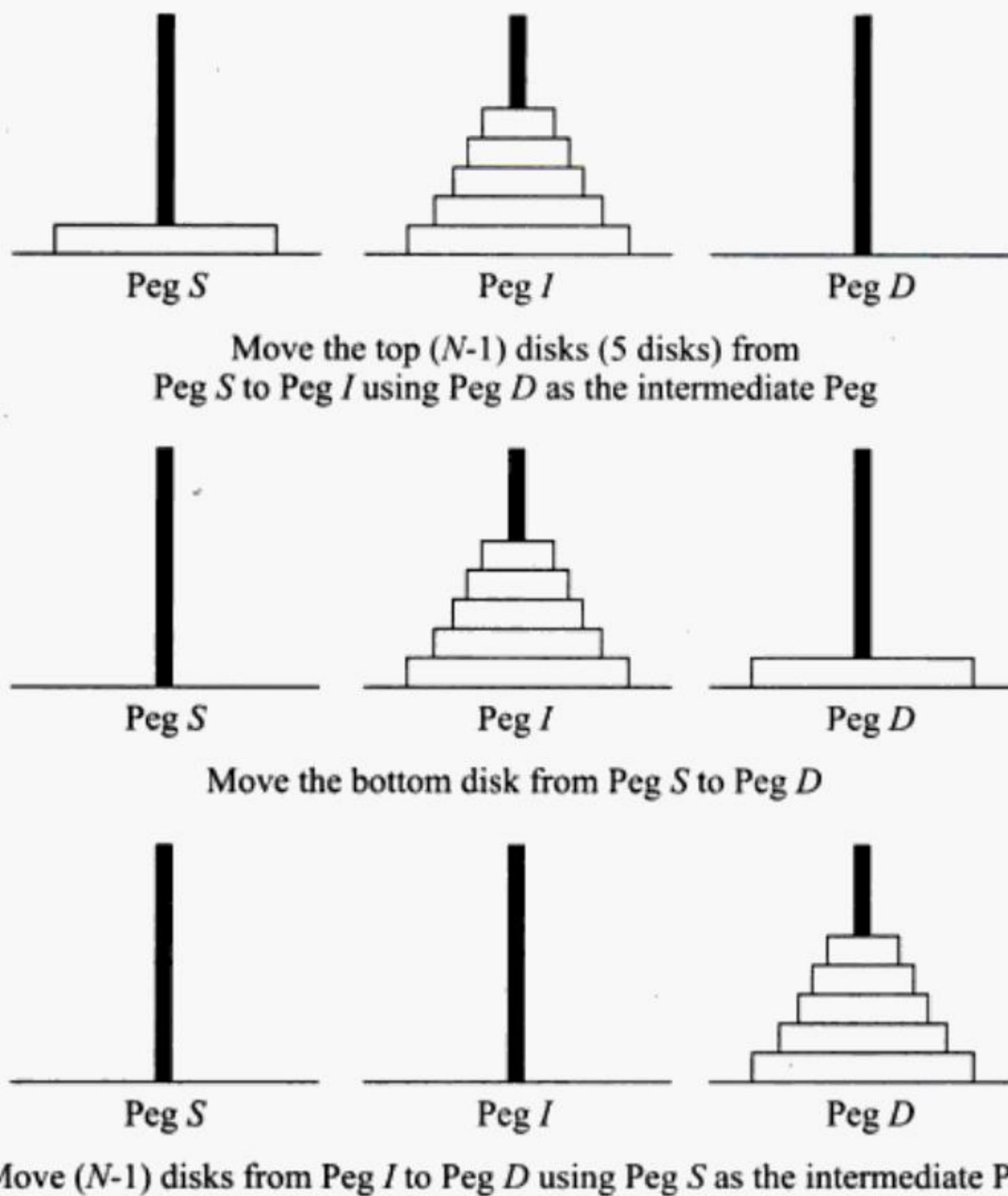


**Fig. 2.3** Tower of Hanoi puzzle (initial configuration)

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	$B[1 : 3, 1 : 2]$ $\begin{bmatrix} 1 & 2 \\ -6 & 4 \\ 3 & 2 \\ 7 & -5 \end{bmatrix}$
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
$A[1 : 5]$	$A \begin{array}{ c c c c c } \hline & 6 & -4 & 3 & 2 & 11 \\ \hline [1] & [2] & [3] & [4] & [5] \\ \hline \end{array}$	STORE ( $A, 3, 17$ )	$A \begin{array}{ c c c c c } \hline & 6 & -4 & 17 & 2 & 11 \\ \hline [1] & [2] & [3] & [4] & [5] \\ \hline \end{array}$
		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

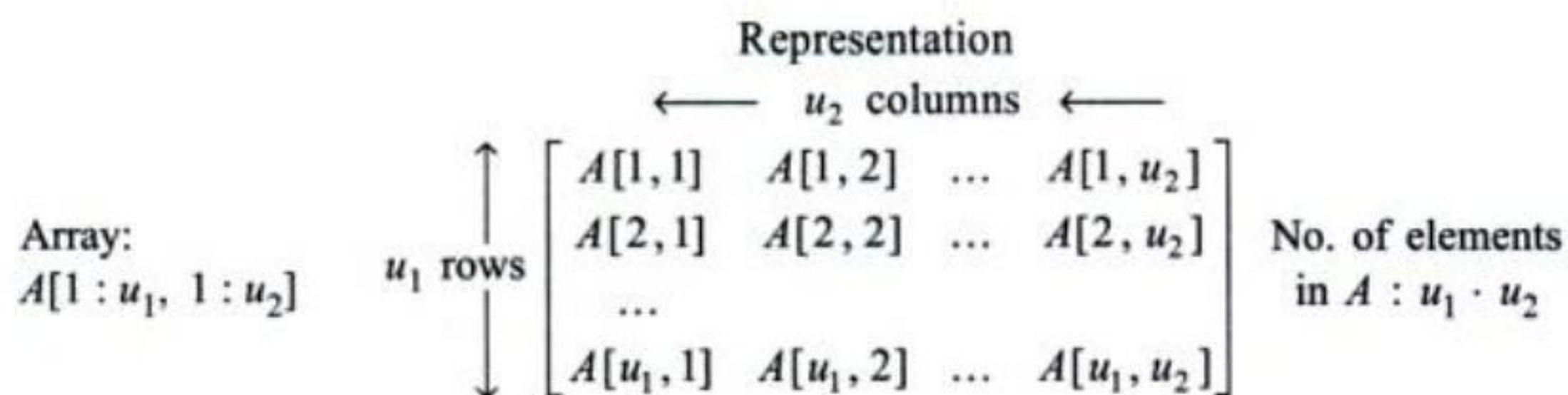
Let  $A[1:u]$  be a one-dimensional array. The size of the array, as is evident is  $u$  and the elements are  $A[1], A[2], \dots, A[u - 1], A[u]$ . In the case of the array  $A[l : u]$  where  $l$  is the lower bound and  $u$  is the upper bound of the index range, the number of elements is given by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

**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

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

### Multi-dimensional array

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

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

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

**Fig. 3.4** Size of a three-dimensional array

**Example 3.3** The number of elements in

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

## Representation of Arrays in Memory

## 3.4

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

In this respect, it is convenient to imagine a two-dimensional array  $A[1 : u_1, 1 : u_2]$  as  $u_1$  number of one-dimensional arrays whose dimension is  $u_2$ . Again, in the case of three-dimensional arrays  $A[1 : u_1, 1 : u_2, 1 : u_3]$  it can be viewed as  $u_1$  number of two-dimensional arrays of size  $u_2 \cdot u_3$ . 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

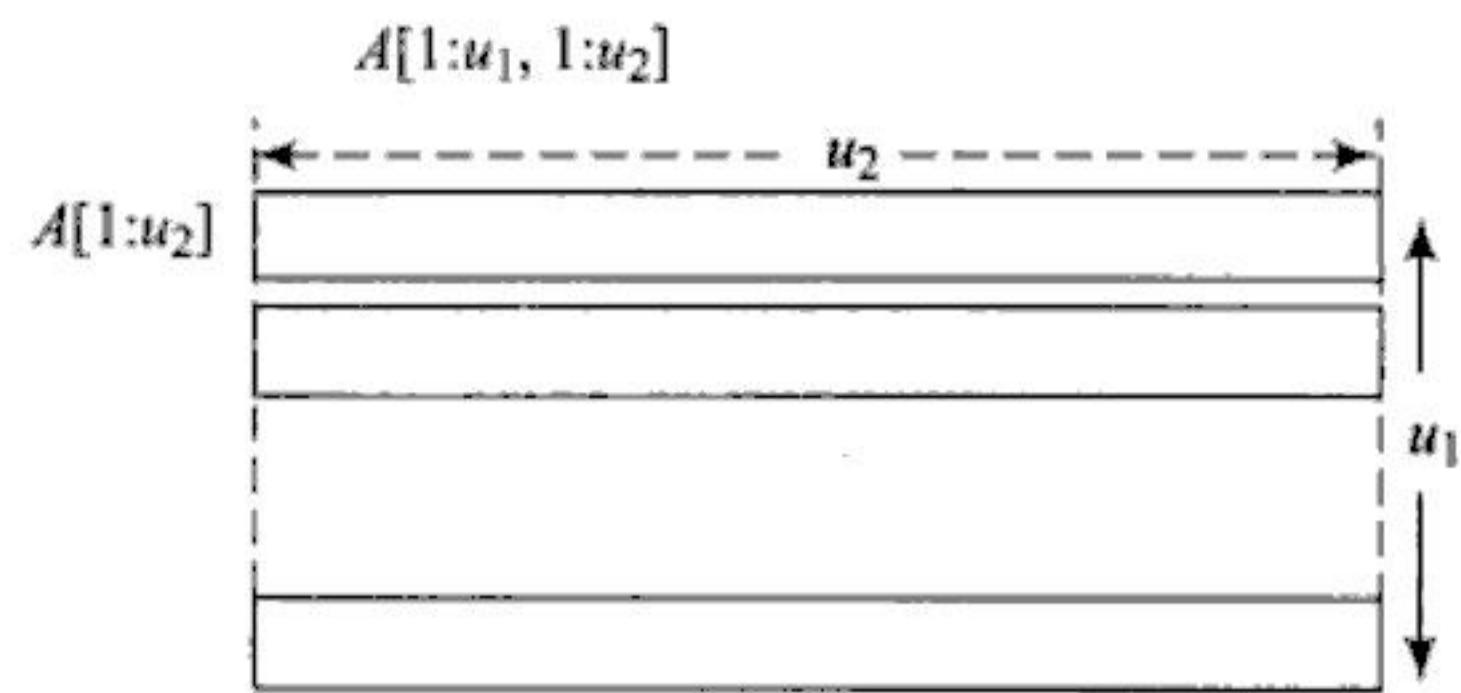
Consider the array  $A(1 : u_1)$  and let  $\alpha$  be the address of the starting memory location referred to as the *base address* of the array. Here as is evident,  $A[1]$  occupies the memory location whose address is  $\alpha$ ,  $A(2)$  occupies  $\alpha + 1$  and so on. In general, the address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

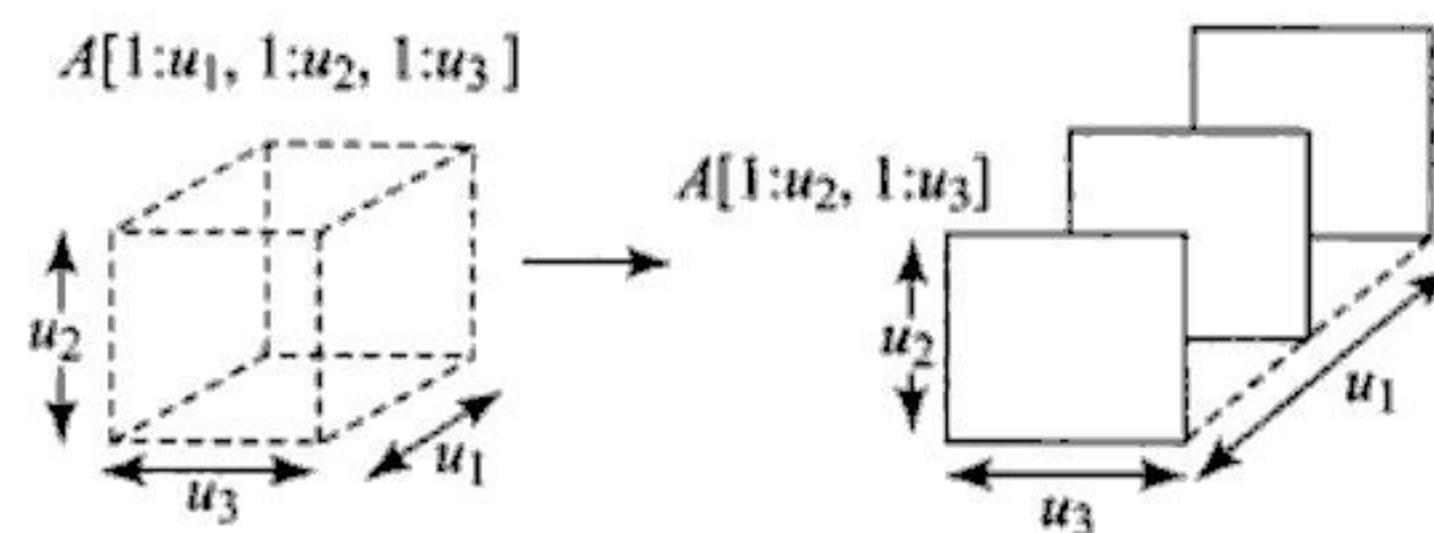
Array	Element	Address
(i) $A[1:17]$	$A[7]$	$\alpha + (7 - 1) = 100 + 6 = 106$
(ii) $A[-2:23]$	$A[16]$	$\alpha + (16 - (-2)) = 100 + 18 = 118$

## 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

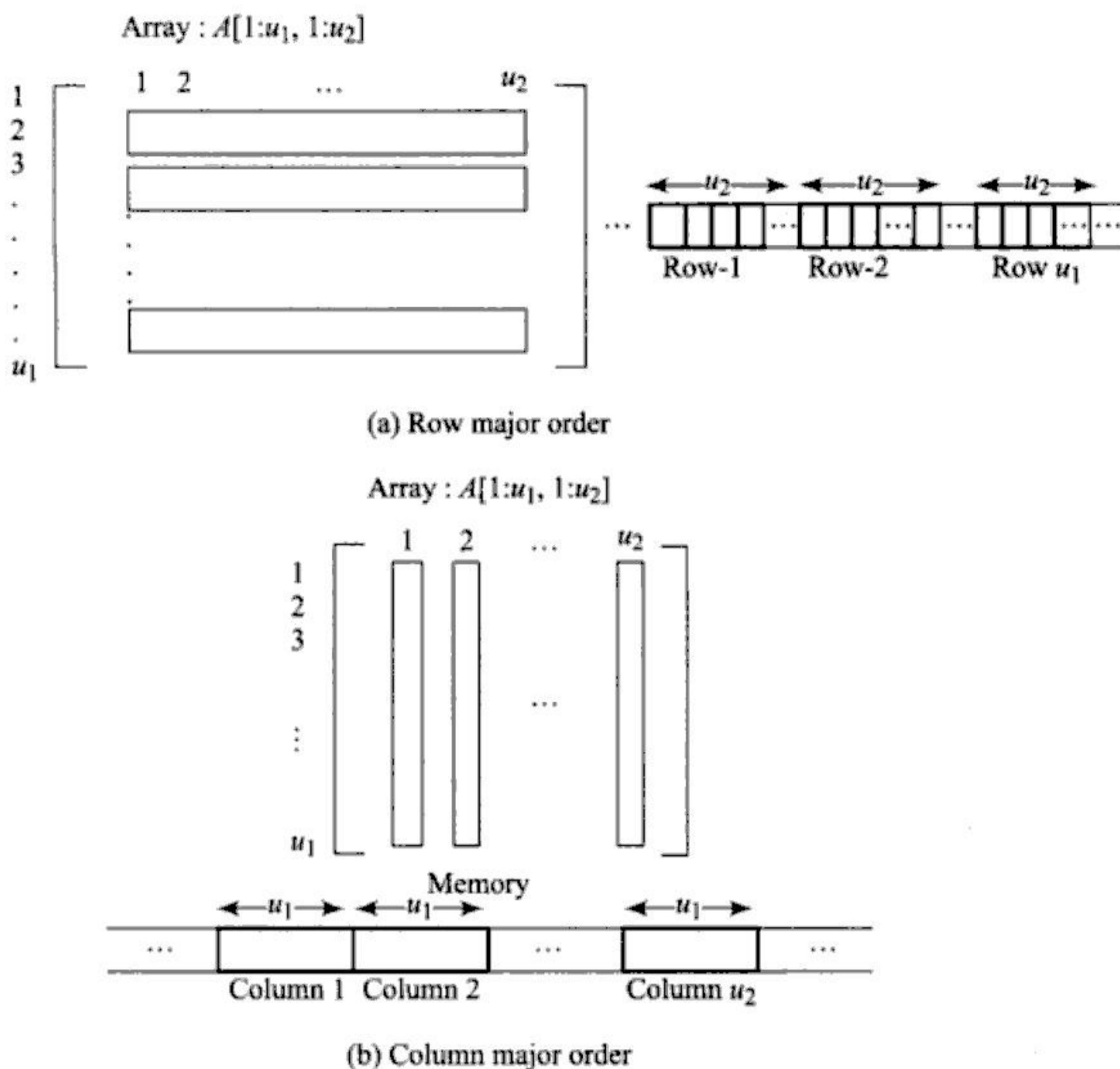


(a) Two-dimensional array viewed in terms of one-dimensional arrays



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

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



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

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

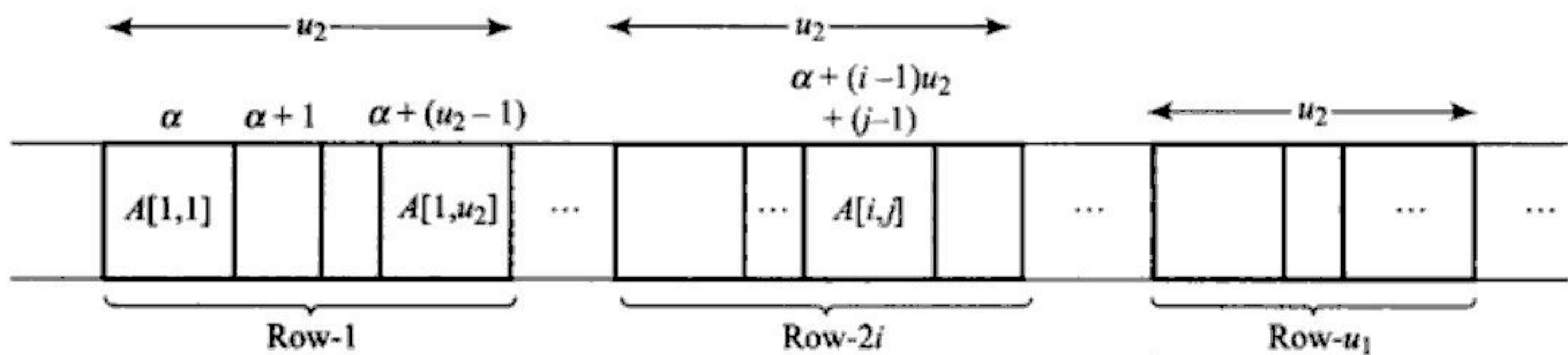
**Fig. 3.7 Representation of one-dimensional arrays in memory**

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:

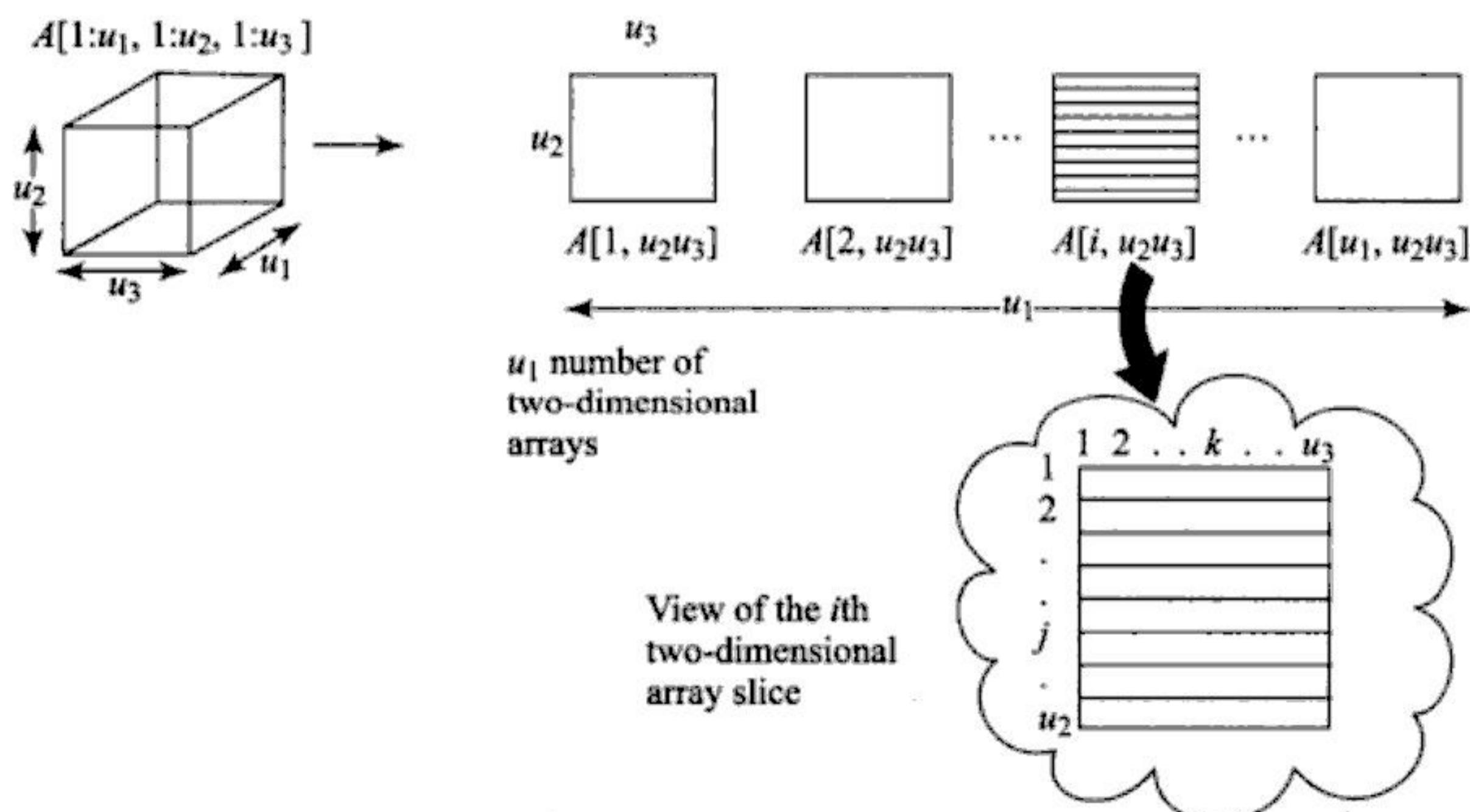
Array	Element	Address
$A[1 : 10, 1 : 5]$	$A[8, 3]$	$220 + (8 - 1)5 + (3 - 1) = 257$
$A[-2 : 4, -6 : 10]$	$A[3, -5]$	$220 + (3 - (-2))(10 - (-6) + 1) + (-5 - (-6)) = 306$

### 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  $\alpha + (i - 1)u_2 \cdot u_3$ . 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  $m.n$  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

$$\begin{array}{c} A[1 : 7, 1 : 6] \\ \left[ \begin{array}{cccccc} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ -2 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{array} \right] \end{array} \quad \begin{array}{c} B[0 : 5, 1 : 3] \\ \left[ \begin{array}{ccc} 7 & 6 & 5 \\ 1 & 2 & 1 \\ 3 & 1 & -2 \\ 3 & 4 & 1 \\ 6 & 2 & -3 \\ 7 & 6 & 1 \end{array} \right] \end{array}$$

**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.

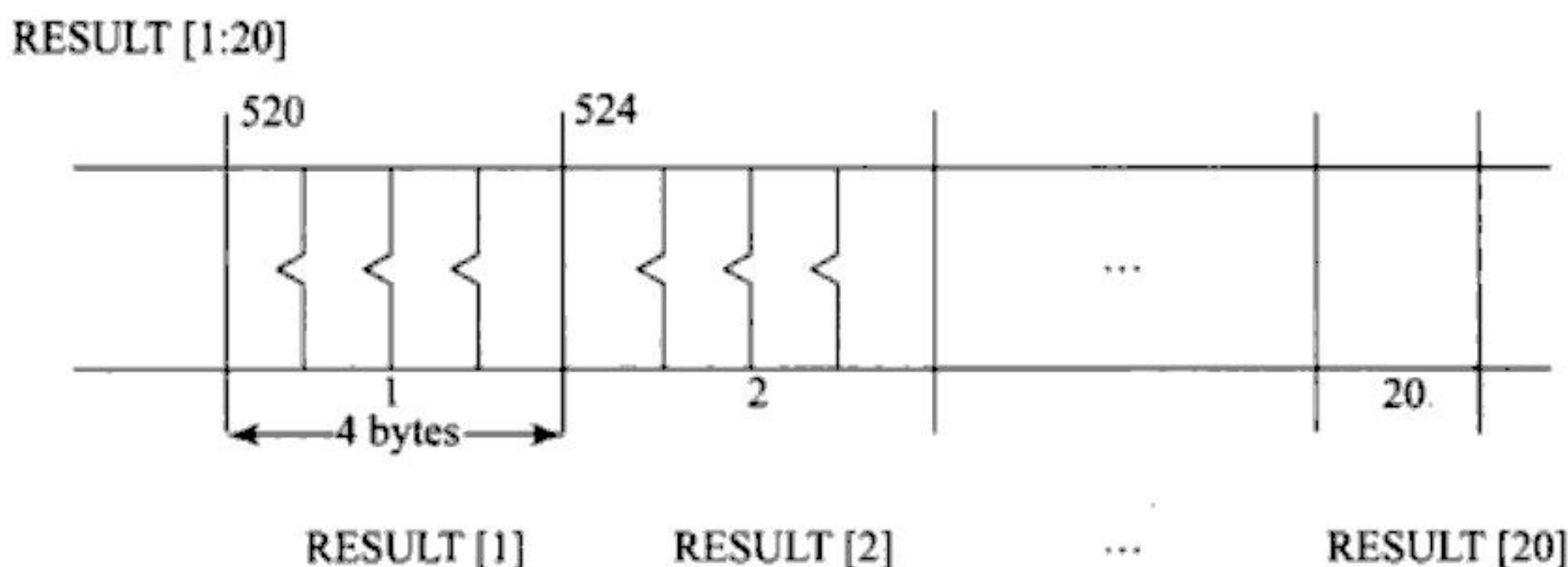
- Two concepts viz., sparse matrices and ordered lists, of use to computer science have been briefly described as applications of arrays.

## Illustrative Problems

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

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

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



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

**Problem 3.2** For the following array  $B$ , compute

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

Array : B

Column index: 0:5

Base address : 1003

Size of the memory location : 4 bytes

Row index : 0:15

*Solution:*

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

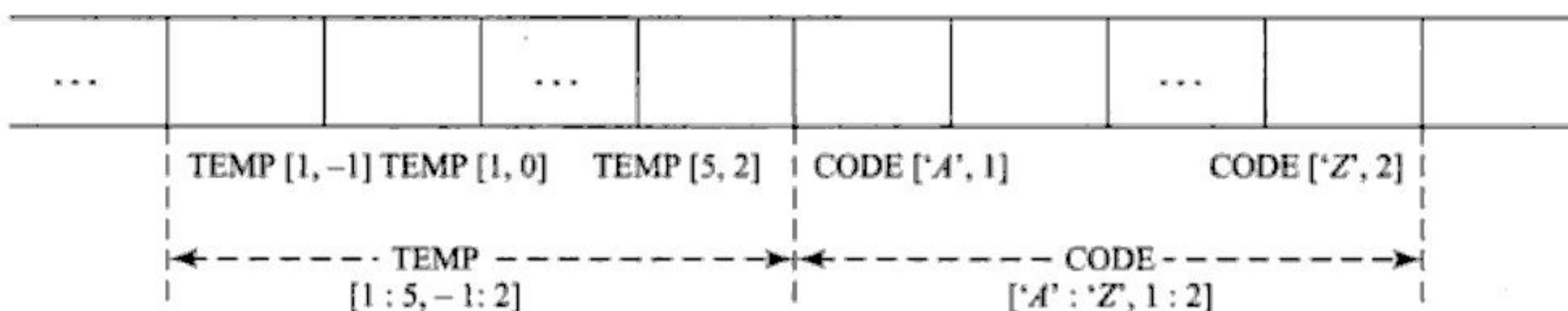
$$1003 + (7 - 0) \cdot 6 + (2 - 0) = 1003 + 42 + 2$$

$$= 1047$$

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

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

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



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

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

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

Hence the address of TEMP[5, 2] is computed as

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

Therefore the base address of CODE is given by 520.

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

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

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

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

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

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

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



## Review Questions



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



# Programming Assignments

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

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

*A: 10 × 12*

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.

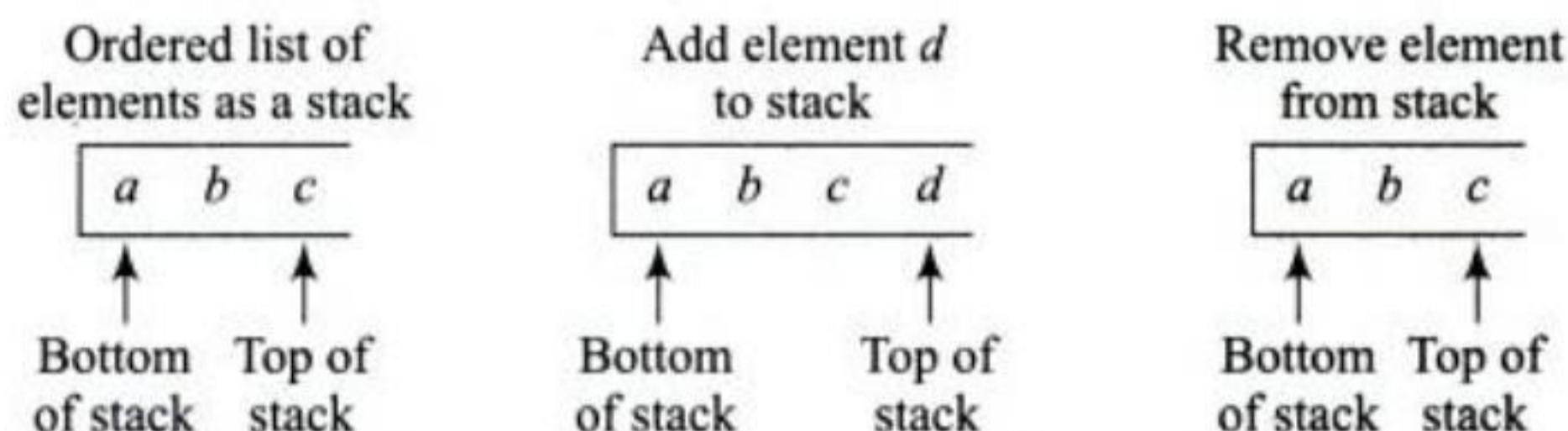
- 4.1 Introduction
- 4.2 Stack Operations
- 4.3 Applications

## 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.

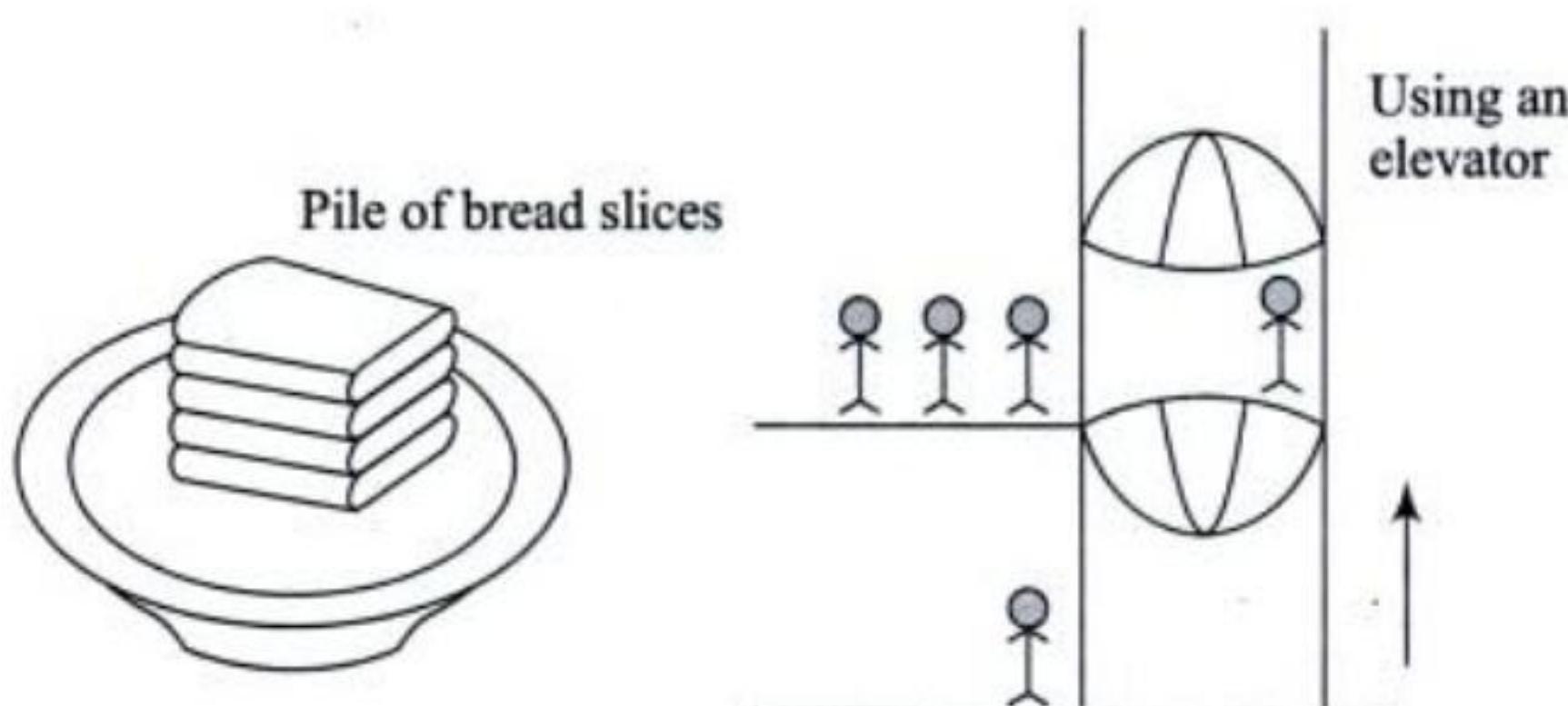


**Fig. 4.1** Stack and its functionality

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

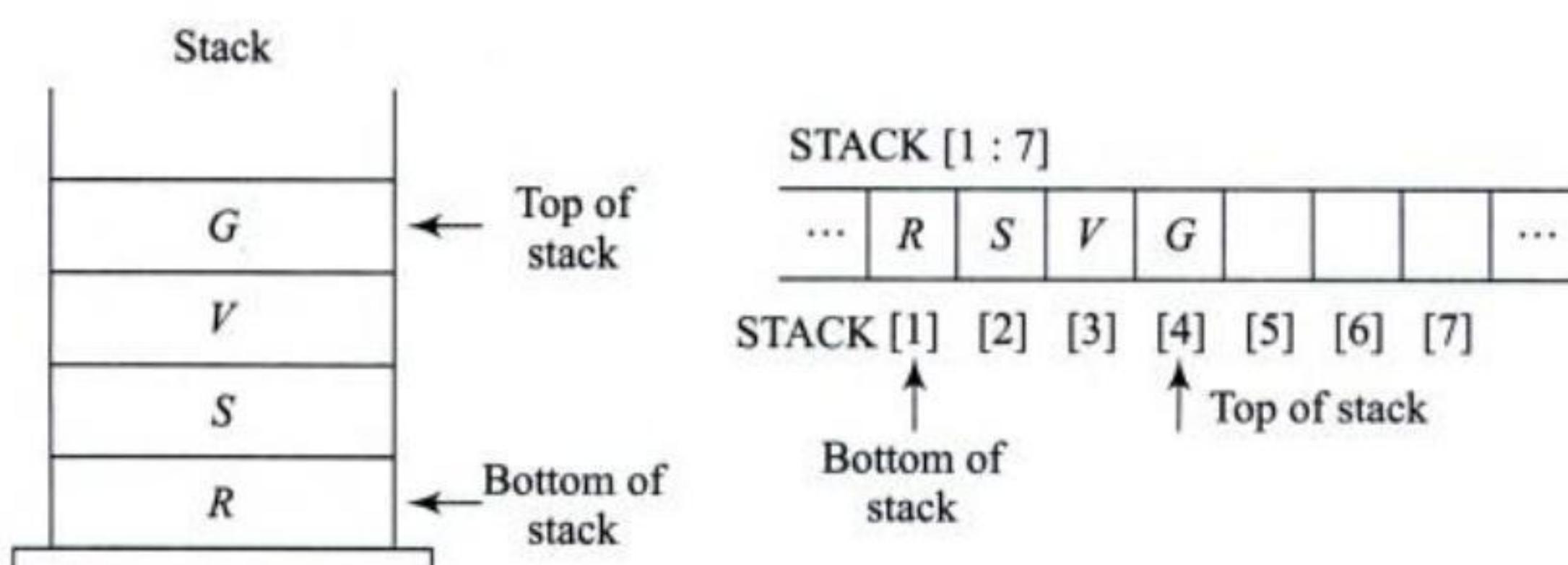
- (i) Insertion or addition of elements known as *Push*
- (ii) Deletion or removal of elements known as *Pop*

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.

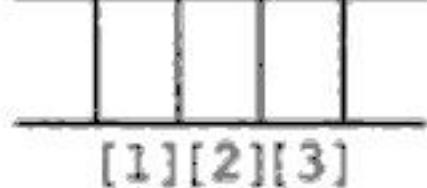
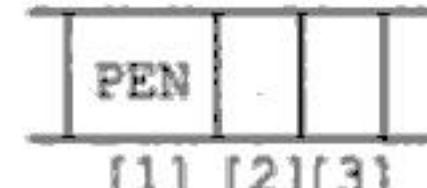
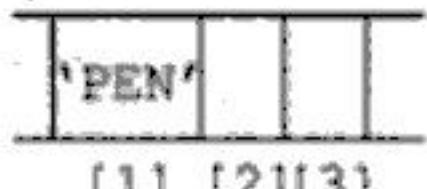
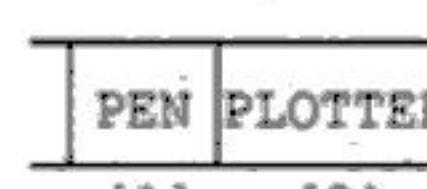
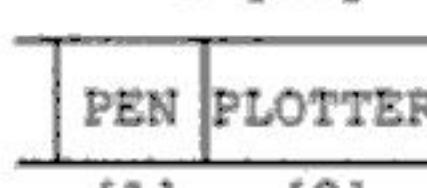
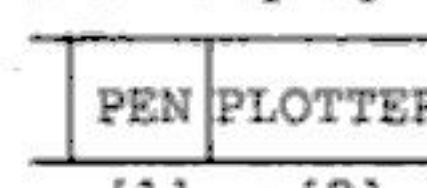
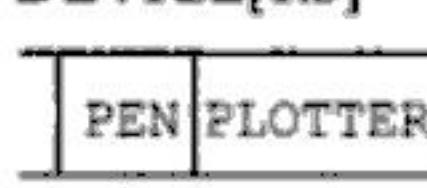
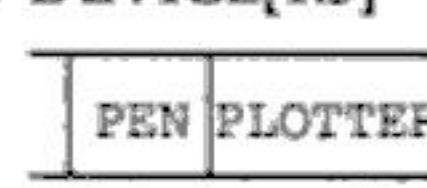
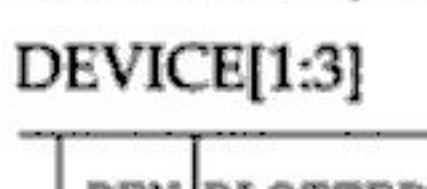
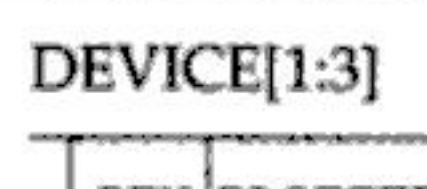
### Algorithm 4.2: Implementation of pop operation on a stack

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

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

During the execution of FACTORIAL(5), the first and the oldest call to be made,  $y$  in Step 3 computes  $y = \text{FACTORIAL}(4) = 24$  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> <operand> <operator>* and is known as *postfix expression*. For example, the infix expression  $a + b$  would have the equivalent postfix expression  $a\ b+$ . A third category of expression is the one which follows the scheme of *<operator> <operand> <operand>* and is known as *prefix expression*. For example, the equivalent prefix expression corresponding to  $a + b$  is  $+a\ b$ . Examples 4.2, 4.3 illustrate the hand computation of prefix and postfix expressions from a given infix expression.

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

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

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) $a * b - f * h) \uparrow d$ \underbrace{\hspace{1cm}}_{\textcircled{1}}	$a * b$	\textcircled{1}: * ab
(ii) $(\textcircled{1} - f * h) \uparrow d$ \underbrace{\hspace{1cm}}_{\textcircled{2}}	$f * h$	\textcircled{2}: * fh
(iii) $(\textcircled{1} - \textcircled{2}) \uparrow d$ \underbrace{\hspace{1cm}}_{\textcircled{3}}	$(\textcircled{1} - \textcircled{2})$	\textcircled{3}: -\textcircled{1}\textcircled{2} (i.e) $- * ab * fh$
(iv) $\textcircled{3} \uparrow d$ \underbrace{\hspace{1cm}}_{\textcircled{4}}	$\textcircled{3} \uparrow d$	\textcircled{4}: \uparrow \textcircled{3}d (i.e) $\uparrow - * ab * fh d$

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

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

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

**Algorithm 4.3:** Procedure to evaluate a postfix expression  $E$

```

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

```

The evaluation of a postfix expression using Algorithm EVAL\_POSTFIX is illustrated in Example 4.4.

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

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

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

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

(Contd.)

(Contd.)

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

## ADT for Stacks

**Data objects:**

A finite set of elements of the same type

**Operations:**

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



## Summary

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

## Illustrative Problems

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

- |                          |                                                       |
|--------------------------|-------------------------------------------------------|
| 1. $X := 30;$            | 9. $\text{PUSH}(S, Z);$                               |
| 2. $Y := 15;$            | 10. $\text{POP}(S, X);$                               |
| 3. $Z := 20;$            | 11. $\text{PUSH}(S, 20);$                             |
| 4. $\text{PUSH}(S, X);$  | 12. $\text{PUSH}(S, X);$                              |
| 5. $\text{PUSH}(S, 40);$ | 13. <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.

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

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

**Solution:**

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

$X$  holds the element at the bottom of the stack.

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

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

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

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

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

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

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

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

The output is 12, 6, 3, 11.

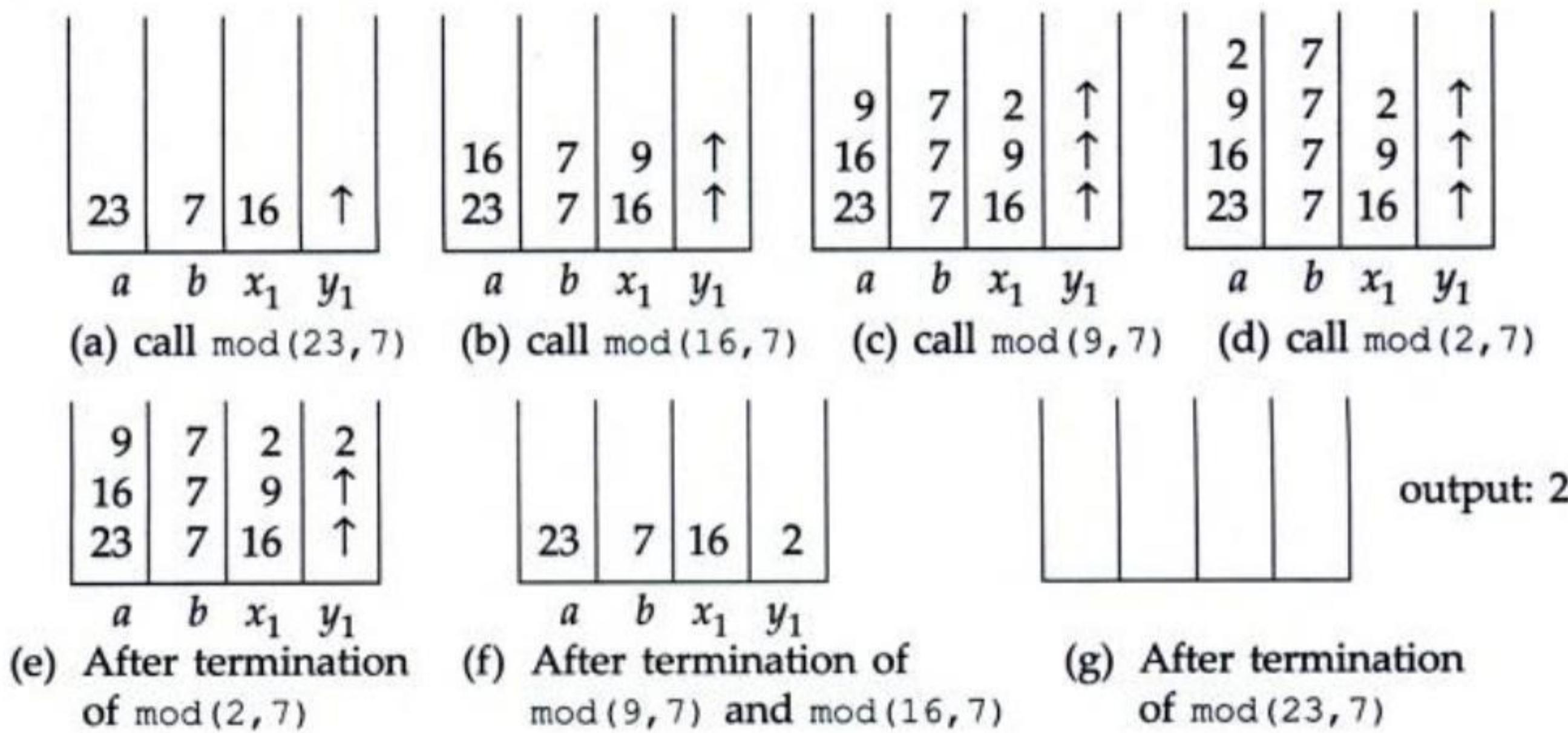
**Problem 4.4** For the following pseudo code of a recursive program mod which computes  $a \bmod b$  given  $a, b$  as inputs, trace the stack contents during the execution of the call mod(23, 7).

```

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

```

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



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

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

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

(Contd.)

(Contd.)

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

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

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

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

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

(Contd.)

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



## Review Questions



## Programming Assignments

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

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

(e.g.)

Stack  $S$ Output stack  $S$ 

Elements:	a	b	c	d
-----------	---	---	---	---

a	c	
---	---	--

Position: 1 2 3 4

1 2

2. Write a recursive program to obtain the  $n^{\text{th}}$  order Fibonacci sequence number. Include appropriate input / output statements to track the variables participating in recursion. Do you observe the 'invisible' stack at work? Record your observations.
3. Implement a program to evaluate any given postfix expression. Test your program for the evaluation of the equivalent postfix form of the expression  $-(A*B)/D \uparrow C + E - F * H * I$  for  $A = 1$ ,  $B = 2$ ,  $D = 3$ ,  $C = 14$ ,  $E = 110$ ,  $F = 220$ ,  $H = 16.78$ ,  $I = 364.621$ .



## QUEUES

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

### Introduction

### 5.1

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

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

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

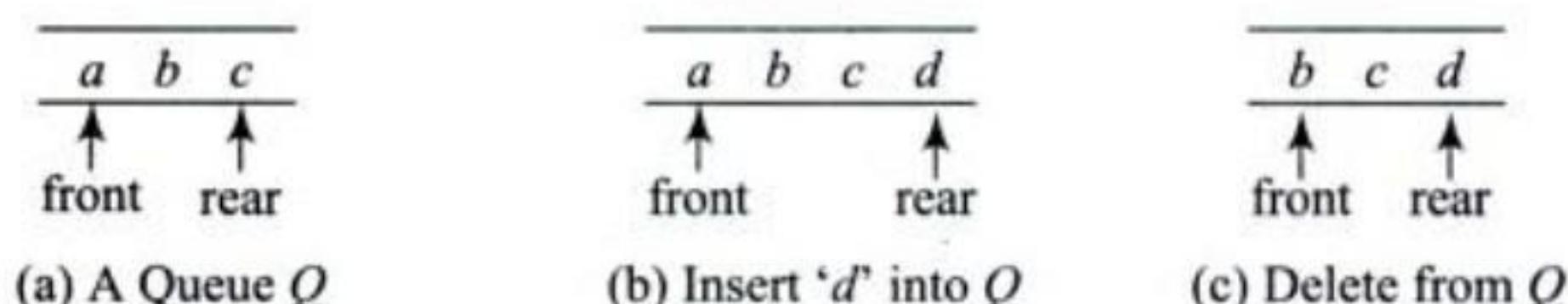


Fig. 5.1 A queue and its functionality

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

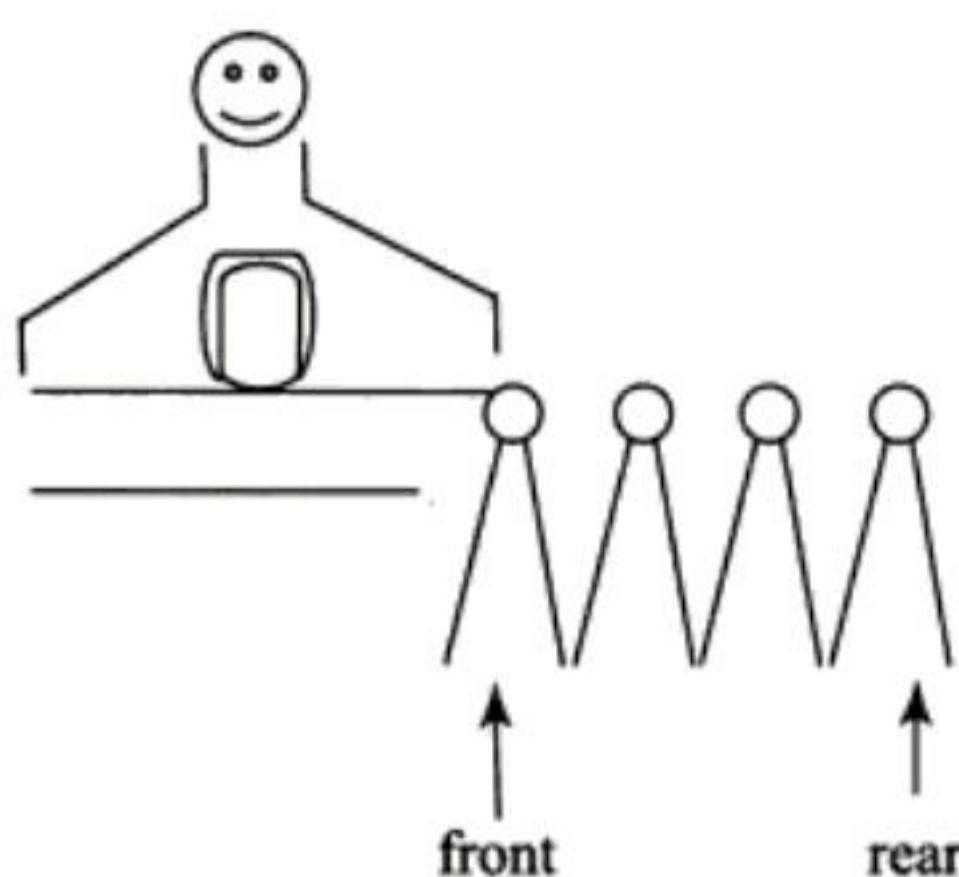
## Operations on Queues

### 5.2

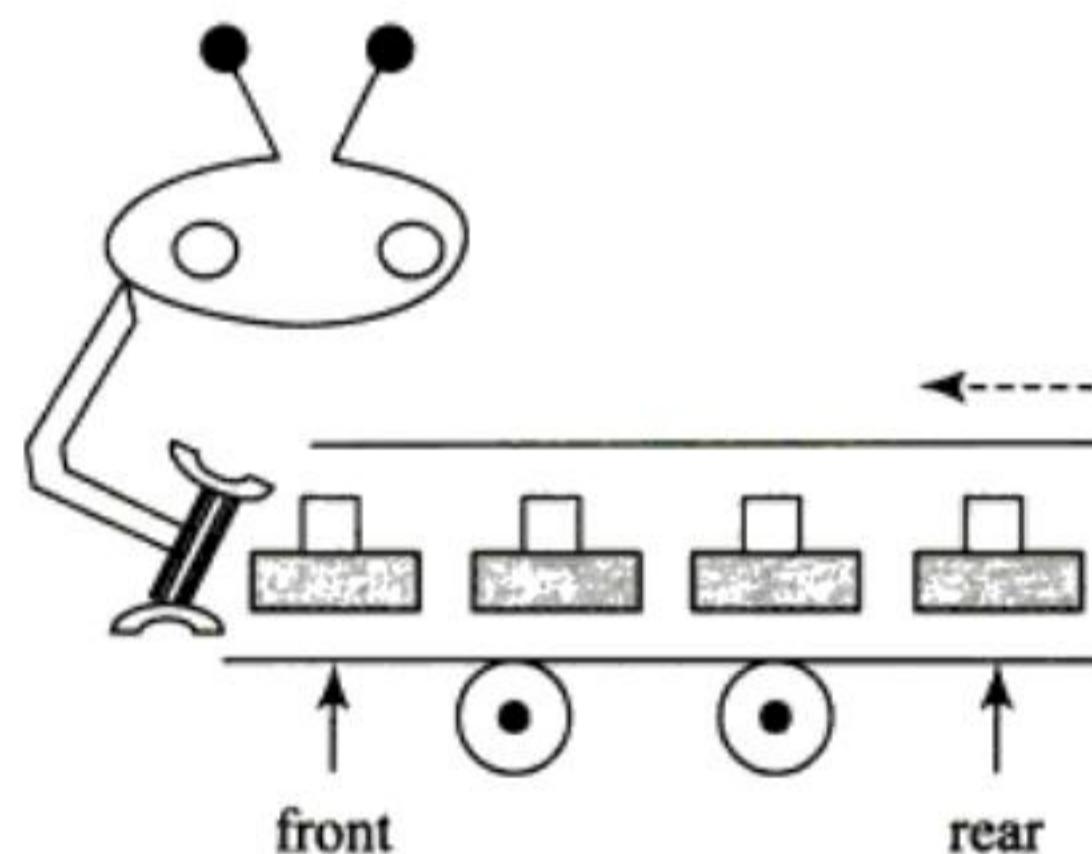
The queue data structure supports two operations, viz.,

- (i) Insertion or addition of elements to a queue
- (ii) Deletion or removal of elements from a queue

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



(a) Queue before a booking counter



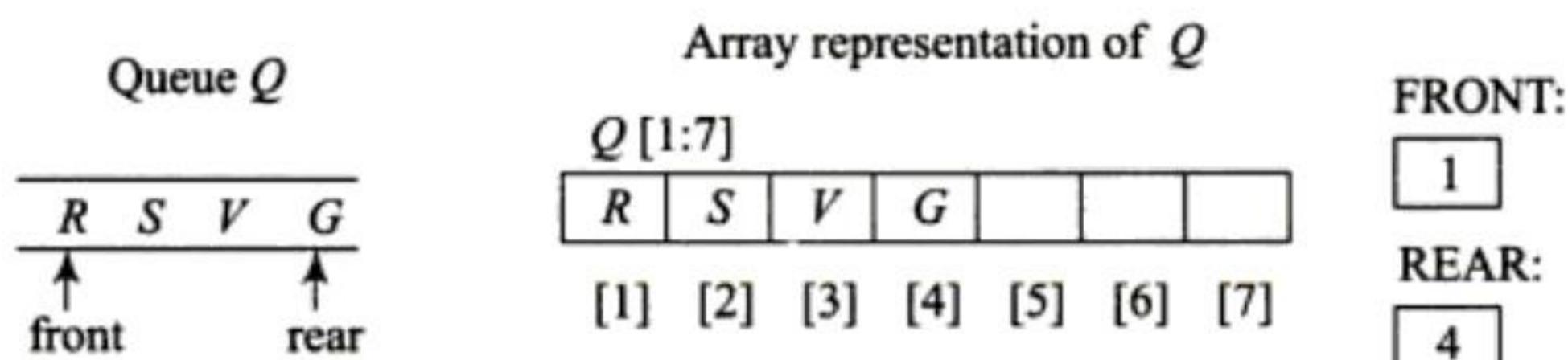
(b) Queue of components in an assembly line

**Fig. 5.2 Common examples of queues**

## Queue Implementation

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

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



**Fig. 5.3 Array implementation of a queue**

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

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

### Implementation of insert and delete operations on a queue

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

Algorithm 5.1 illustrates the insert operation on a queue.

#### Algorithm 5.1: Implementation of an insert operation on a queue

```

procedure INSERTQ (Q, n, ITEM, REAR)
    /* insert item ITEM into Q with capacity n */
    if (REAR = n) then QUEUE_FULL;
    REAR = REAR + 1;    /* Increment REAR*/
    Q[REAR] = ITEM;     /* Insert ITEM as the rear element*/
end INSERTQ

```

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

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

#### Algorithm 5.2: Implementation of a delete operation on a queue

```

procedure DELETEQ (Q, FRONT, REAR, ITEM )
if (FRONT = REAR) then QUEUE_EMPTY;
FRONT = FRONT + 1;
ITEM = Q[FRONT];
end DELETEQ.

```

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

Soon after the queue  $Q$  has been initialized,  $\text{FRONT} = \text{REAR} = 0$ . Hence the condition ( $\text{FRONT} = \text{REAR}$ ) ensures that the queue is empty. Again after a sequence of operations when  $Q$  has become partially or completely full and delete operations are repeatedly invoked to empty the queue, it may be observed how  $\text{FRONT}$  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  $\text{BIRDS}[1:3]$

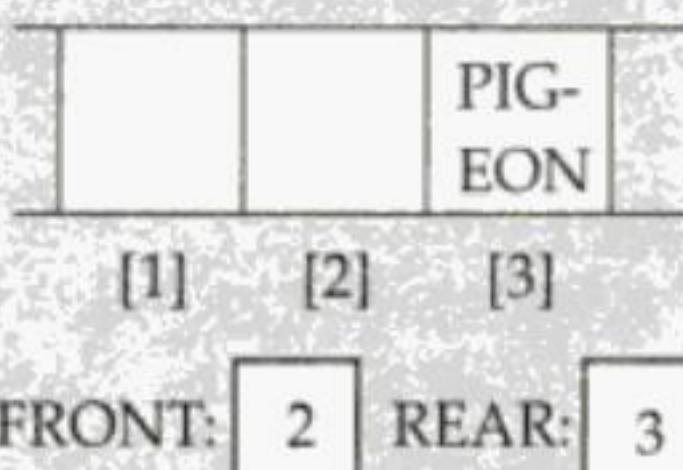
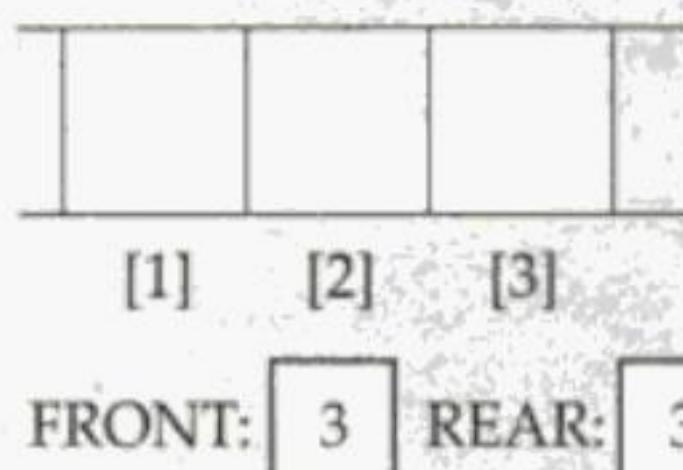
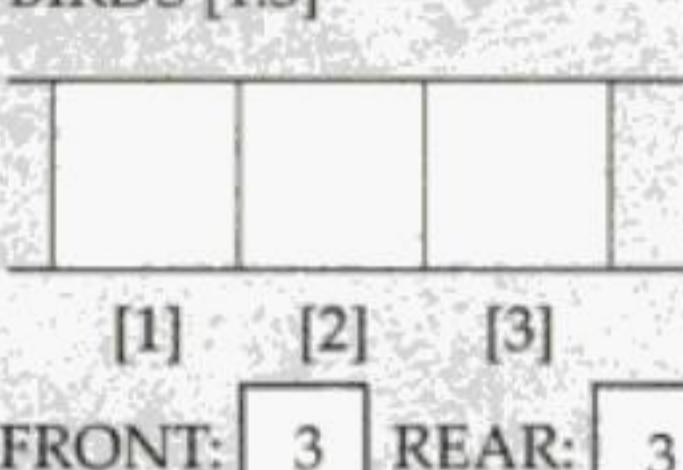
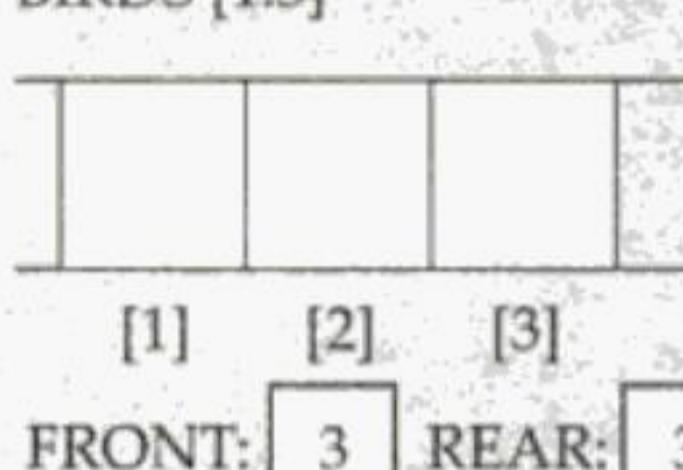
Operation	Queue before operation	Algorithm	Queue after operation	Remarks
1. Insert 'DOVE' into $\text{BIRDS}[1:3]$	BIRDS [1:3]  [1] [2] [3] FRONT: 0 REAR: 0	INSERTQ (BIRDS, 3, 'DOVE', 0)	BIRDS [1:3]  [1] [2] [3] FRONT: 0 REAR: 1	Insert 'DOVE' successful
2. Insert 'PEACOCK' into $\text{BIRDS}[1:3]$	BIRDS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

(Contd.)

(Contd.)

3. Insert 'PIGEON' in to BIRDS [1:3]	<p>BIRDS [1:3]</p> <table border="1" data-bbox="614 682 1348 965"> <tr> <td>DOVE</td> <td>PEA-COCK</td> <td></td> </tr> <tr> <td>[1]</td> <td>[2]</td> <td>[3]</td> </tr> </table> <p>FRONT: <input type="text" value="0"/> REAR: <input type="text" value="2"/></p>	DOVE	PEA-COCK		[1]	[2]	[3]	INSERTQ(BIRDS, 3, 'PIGEON', 2)	<p>BIRDS [1:3]</p> <table border="1" data-bbox="1911 682 2645 965"> <tr> <td>DOVE</td> <td>PEA-COCK</td> <td>PIG-EON</td> </tr> <tr> <td>[1]</td> <td>[2]</td> <td>[3]</td> </tr> </table> <p>FRONT: <input type="text" value="0"/> REAR: <input type="text" value="3"/></p>	DOVE	PEA-COCK	PIG-EON	[1]	[2]	[3]	Insert 'PIGEON' successful
DOVE	PEA-COCK															
[1]	[2]	[3]														
DOVE	PEA-COCK	PIG-EON														
[1]	[2]	[3]														
4. Insert 'SWAN' in to BIRDS [1:3]	<p>BIRDS [1:3]</p> <table border="1" data-bbox="614 1389 1348 1672"> <tr> <td>DOVE</td> <td>PEA-COCK</td> <td>PIG-EON</td> </tr> <tr> <td>[1]</td> <td>[2]</td> <td>[3]</td> </tr> </table> <p>FRONT: <input type="text" value="0"/> REAR: <input type="text" value="3"/></p>	DOVE	PEA-COCK	PIG-EON	[1]	[2]	[3]	INSERTQ(BIRDS, 3, 'SWAN', 3)	<p>BIRDS [1:3]</p> <table border="1" data-bbox="1911 1389 2645 1672"> <tr> <td>DOVE</td> <td>PEA-COCK</td> <td>PIG-EON</td> </tr> <tr> <td>[1]</td> <td>[2]</td> <td>[3]</td> </tr> </table> <p>FRONT: <input type="text" value="0"/> REAR: <input type="text" value="3"/></p>	DOVE	PEA-COCK	PIG-EON	[1]	[2]	[3]	Insert 'SWAN' failure! QUEUE_FULL condition invoked.
DOVE	PEA-COCK	PIG-EON														
[1]	[2]	[3]														
DOVE	PEA-COCK	PIG-EON														
[1]	[2]	[3]														
5. Delete	<p>BIRDS [1:3]</p> <table border="1" data-bbox="614 2095 1348 2378"> <tr> <td>DOVE</td> <td>PEA-COCK</td> <td>PIG-EON</td> </tr> <tr> <td>[1]</td> <td>[2]</td> <td>[3]</td> </tr> </table> <p>FRONT: <input type="text" value="0"/> REAR: <input type="text" value="3"/></p>	DOVE	PEA-COCK	PIG-EON	[1]	[2]	[3]	DELETEQ(BIRDS, 0, 3, ITEM)	<p>BIRDS [1:3]</p> <table border="1" data-bbox="1911 2095 2645 2378"> <tr> <td></td> <td>PEA-COCK</td> <td>PIG-EON</td> </tr> <tr> <td>[1]</td> <td>[2]</td> <td>[3]</td> </tr> </table> <p>FRONT: <input type="text" value="1"/> REAR: <input type="text" value="3"/></p>		PEA-COCK	PIG-EON	[1]	[2]	[3]	Delete successful. ITEM =DOVE
DOVE	PEA-COCK	PIG-EON														
[1]	[2]	[3]														
	PEA-COCK	PIG-EON														
[1]	[2]	[3]														
6. Delete	<p>BIRDS [1:3]</p> <table border="1" data-bbox="614 2755 1348 3037"> <tr> <td></td> <td>PEA-COCK</td> <td>PIG-EON</td> </tr> <tr> <td>[1]</td> <td>[2]</td> <td>[3]</td> </tr> </table> <p>FRONT: <input type="text" value="1"/> REAR: <input type="text" value="3"/></p>		PEA-COCK	PIG-EON	[1]	[2]	[3]	DELETEQ(BIRDS, 0, 3, ITEM)	<p>BIRDS [1:3]</p> <table border="1" data-bbox="1911 2755 2645 3037"> <tr> <td></td> <td></td> <td>PIG-EON</td> </tr> <tr> <td>[1]</td> <td>[2]</td> <td>[3]</td> </tr> </table> <p>FRONT: <input type="text" value="2"/> REAR: <input type="text" value="3"/></p>			PIG-EON	[1]	[2]	[3]	Delete successful. ITEM =PEACOCK
	PEA-COCK	PIG-EON														
[1]	[2]	[3]														
		PIG-EON														
[1]	[2]	[3]														
7. Insert 'SWAN' in to BIRDS [1:3]	<p>BIRDS [1:3]</p> <table border="1" data-bbox="614 3438 1348 3720"> <tr> <td></td> <td></td> <td>PIG-EON</td> </tr> <tr> <td>[1]</td> <td>[2]</td> <td>[3]</td> </tr> </table> <p>FRONT: <input type="text" value="2"/> REAR: <input type="text" value="3"/></p>			PIG-EON	[1]	[2]	[3]	INSERTQ(BIRDS, 3, 'SWAN', 3)	<p>BIRDS [1:3]</p> <table border="1" data-bbox="1911 3438 2645 3720"> <tr> <td></td> <td></td> <td>PIG-EON</td> </tr> <tr> <td>[1]</td> <td>[2]</td> <td>[3]</td> </tr> </table> <p>FRONT: <input type="text" value="2"/> REAR: <input type="text" value="3"/></p>			PIG-EON	[1]	[2]	[3]	Insert 'SWAN' failure! QUEUE_FULL condition invoked.
		PIG-EON														
[1]	[2]	[3]														
		PIG-EON														
[1]	[2]	[3]														

(Contd.)

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

**invocation****Limitations of linear queues**

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

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

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

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

## Circular Queues

## 5.3

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

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

### Operations on a circular queue

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

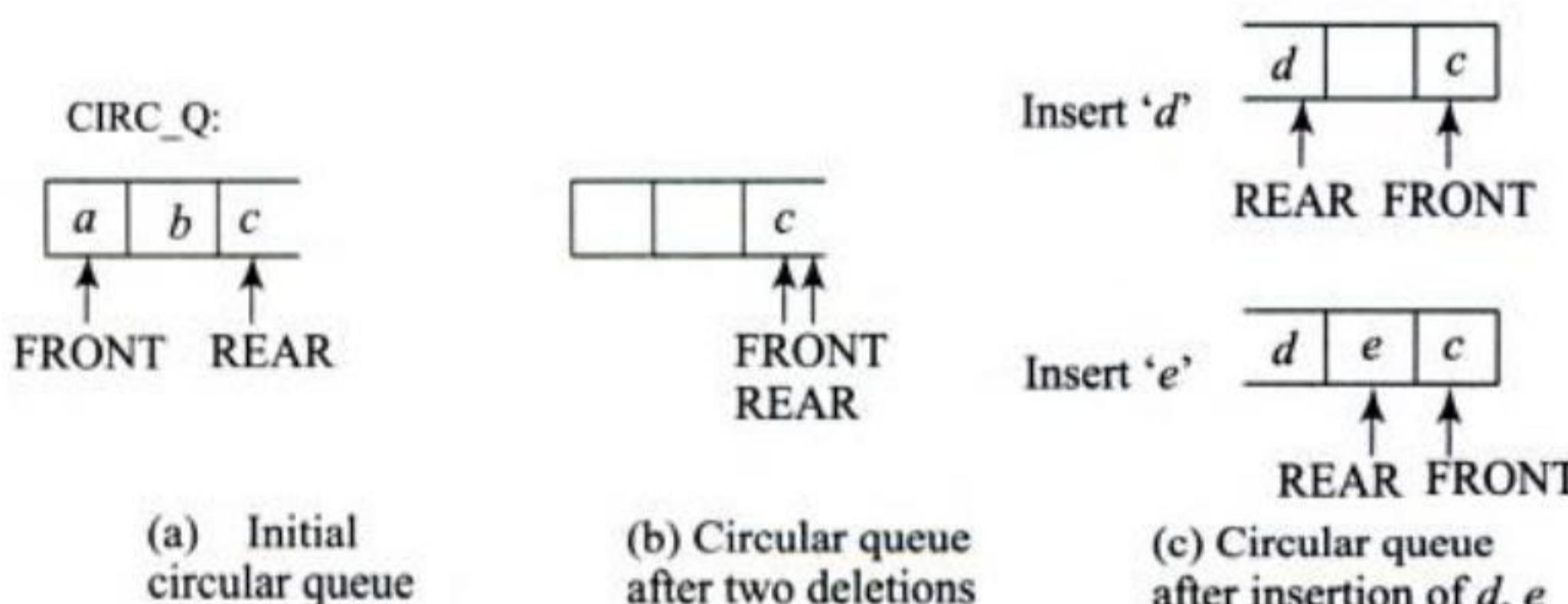


Fig. 5.4 Working of a circular queue

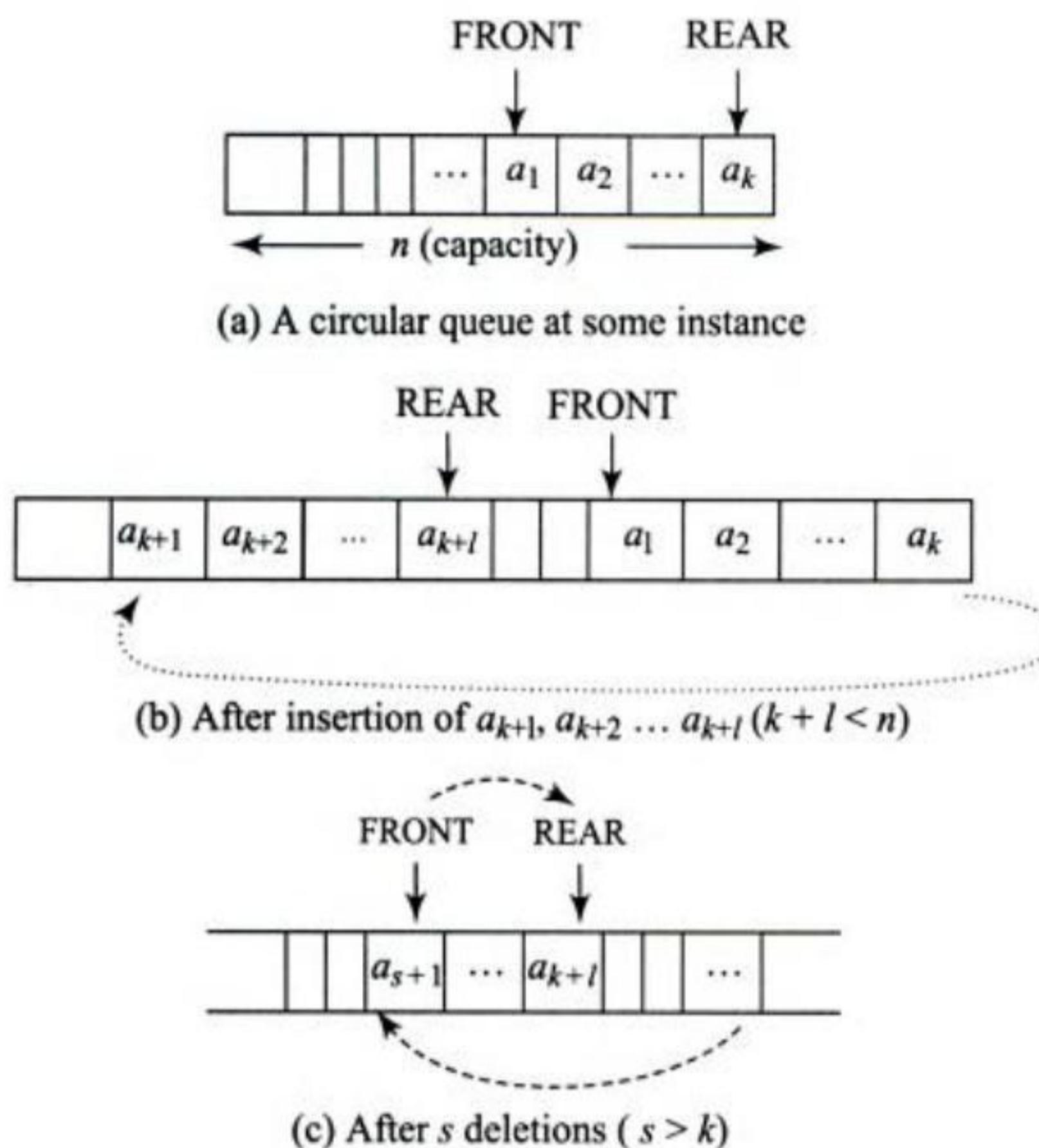
Observe the circular movement of the FRONT and REAR variables. After two deletions, FRONT moves towards REAR and points to 'c' as the current front element of CIRC\_Q (Fig. 5.4(b)). When 'd' is inserted, unlike linear queues, REAR curls back in a clock wise fashion to accommodate 'd' in the vacant space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

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

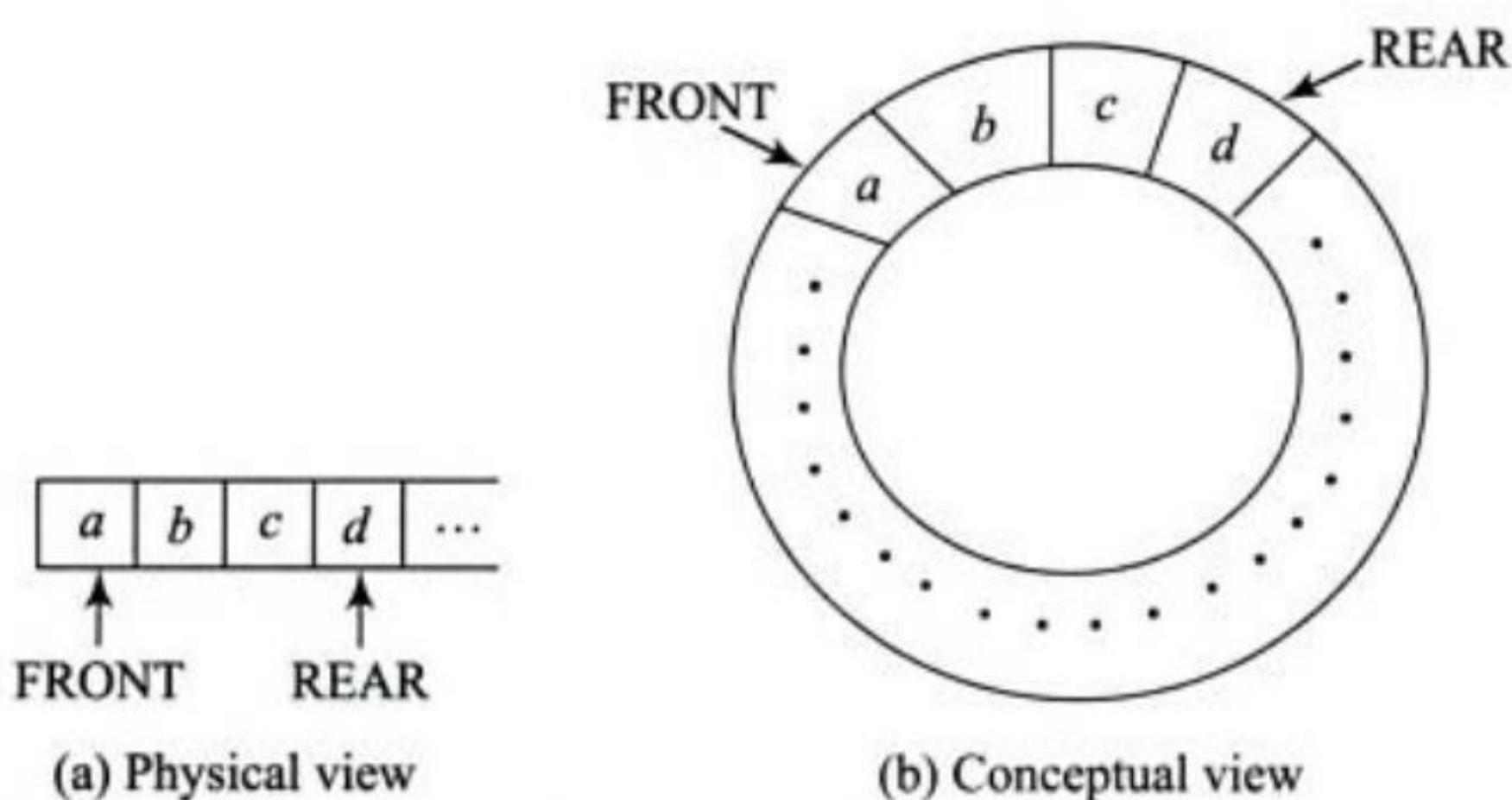
A circular queue when implemented using arrays is not different from linear queues in their physical storage. In other words, a linear queue is conceptually viewed to have a circular form to understand the clockwise movement of FRONT and REAR variables as shown in Fig. 5.6.

### Implementation of insertion and deletion operations in a circular queue

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



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



**Fig. 5.6** Physical and conceptual view of a circular queue

**Algorithm 5.3:** Implementation of insert operation on a circular queue

```

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

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

```

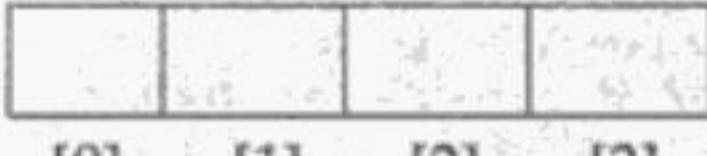
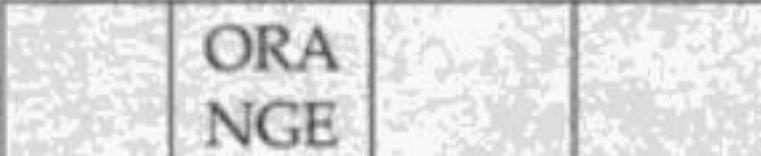
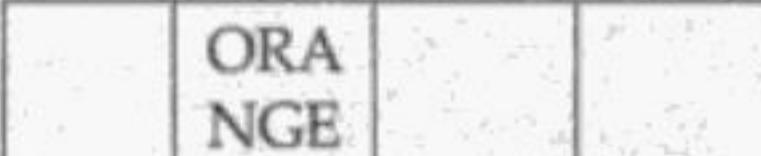
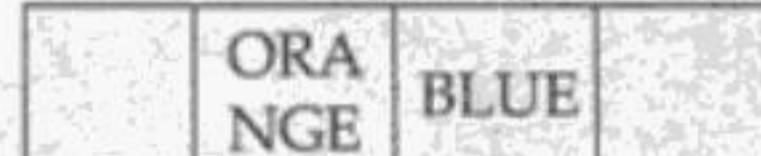
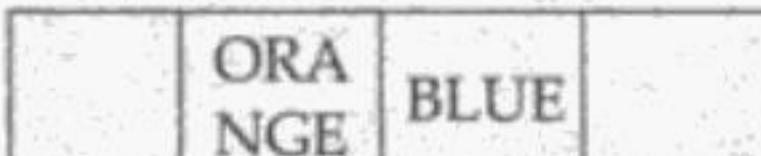
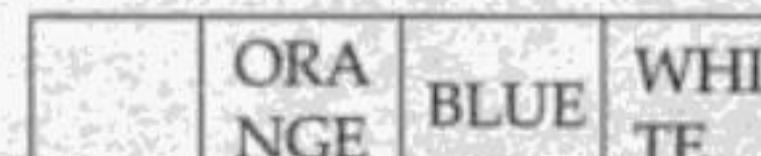
procedure DELETE_CIRCQ(CIRC_Q, FRONT, REAR, n, ITEM)
  If (FRONT = REAR) then CIRCQ_EMPTY; /* CIRC_Q is physically empty*/
  FRONT = (FRONT+1) mod n;
  ITEM = CIRC_Q [FRONT];
end DELETE_CIRCQ

```

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

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

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

Circular Queue operation	Circular queue before operation	Algorithm Invocation	Circular queue after operation	Remarks
1. Insert 'ORANGE' into COLOURS [0:3]	COLOURS [0: 3]  [0] [1] [2] [3] FRONT: 0 REAR: 0	INSERT_CIRCQ(COLOURS, 0, 0, 4, 'ORANGE')	COLOURS [0:3]  [0] ORA NGE [2] [3] FRONT: 0 REAR: 1	Insert 'ORANGE' successful
2. Insert 'BLUE' into COLOURS [0:3]	COLOURS [0:3]  [0] ORA NGE [2] [3] FRONT: 0 REAR: 1	INSERT_CIRCQ(COLOURS, 0, 1, 4, 'BLUE')	COLOURS [0:3]  [0] ORA NGE [2] [3] FRONT: 0 REAR: 2	Insert 'BLUE' successful
3. Insert 'WHITE' into COLOURS [0:3]	COLOURS [0:3]  [0] ORA NGE [2] [3] FRONT: 0 REAR: 2	INSERT_CIRCQ(COLOURS, 0, 2, 4, 'WHITE')	COLOURS [0:3]  [0] ORA NGE [2] WHI TE [3] FRONT: 0 REAR: 3	Insert 'WHITE' successful

(Contd.)

(Contd.)

<p>4. Insert 'RED' into COLOURS [0:3]</p>	<p>COLOURS [0:3]</p> <table border="1" data-bbox="710 648 1495 944"> <tr> <td></td><td>ORANGE</td><td>BLUE</td><td>WHITE</td></tr> <tr> <td>[0]</td><td>[1]</td><td>[2]</td><td>[3]</td></tr> </table> <p>FRONT: 0 REAR: 3</p>		ORANGE	BLUE	WHITE	[0]	[1]	[2]	[3]	<p>INSERT_CIRCQ (COLOURS, 0, 3, 4, 'RED')</p>	<p>COLOURS [0:3]</p> <table border="1" data-bbox="2024 648 2809 944"> <tr> <td></td><td>ORANGE</td><td>BLUE</td><td>WHITE</td></tr> <tr> <td>[0]</td><td>[1]</td><td>[2]</td><td>[3]</td></tr> </table> <p>FRONT: 0 REAR: 3</p>		ORANGE	BLUE	WHITE	[0]	[1]	[2]	[3]	<p>CIRCQ_FULL condition is invoked. Insert 'RED' failure!</p> <p>Note: REAR retracts to its previous value of 3.</p>
	ORANGE	BLUE	WHITE																	
[0]	[1]	[2]	[3]																	
	ORANGE	BLUE	WHITE																	
[0]	[1]	[2]	[3]																	
<p>5, 6. Delete twice from COLOURS [0:3]</p>	<p>COLOURS [0:3]</p> <table border="1" data-bbox="710 1833 1495 2129"> <tr> <td></td><td>ORANGE</td><td>BLUE</td><td>WHITE</td></tr> <tr> <td>[0]</td><td>[1]</td><td>[2]</td><td>[3]</td></tr> </table> <p>FRONT: 0 REAR: 3</p>		ORANGE	BLUE	WHITE	[0]	[1]	[2]	[3]	<p>DELETE_CIRCQ (COLOURS, 0, 3, 4, 'ITEM') DELETE_CIRCQ (COLOURS, 1, 3, 4, ITEMS)</p>	<p>COLOURS [0:3]</p> <table border="1" data-bbox="2024 1833 2809 2129"> <tr> <td></td><td></td><td></td><td>WHITE</td></tr> <tr> <td>[0]</td><td>[1]</td><td>[2]</td><td>[3]</td></tr> </table> <p>FRONT: 2 REAR: 3</p>				WHITE	[0]	[1]	[2]	[3]	<p>DELETE operation successful ITEM = ORANGE ITEM = BLUE</p>
	ORANGE	BLUE	WHITE																	
[0]	[1]	[2]	[3]																	
			WHITE																	
[0]	[1]	[2]	[3]																	
<p>7. Insert 'YELLOW' into COLOURS [0:3]</p>	<p>COLOURS [0:3]</p> <table border="1" data-bbox="710 2773 1495 3069"> <tr> <td></td><td></td><td></td><td>WHITE</td></tr> <tr> <td>[0]</td><td>[1]</td><td>[2]</td><td>[3]</td></tr> </table> <p>FRONT: 2 REAR: 3</p>				WHITE	[0]	[1]	[2]	[3]	<p>INSERT_CIRCQ (COLOURS, 2, 3, 4, 'YELLOW')</p>	<p>COLOURS [0:3]</p> <table border="1" data-bbox="2024 2773 2809 3069"> <tr> <td>YELLOW</td><td></td><td></td><td>WHITE</td></tr> <tr> <td>[0]</td><td>[1]</td><td>[2]</td><td>[3]</td></tr> </table> <p>FRONT: 2 REAR: 0</p>	YELLOW			WHITE	[0]	[1]	[2]	[3]	<p>Insert 'YELLOW' successful</p>
			WHITE																	
[0]	[1]	[2]	[3]																	
YELLOW			WHITE																	
[0]	[1]	[2]	[3]																	
<p>8. Insert 'VIOLET' into COLOURS [0:3]</p>	<p>COLOURS [0:3]</p> <table border="1" data-bbox="710 3431 1495 3727"> <tr> <td>YELLOW</td><td></td><td></td><td>WHITE</td></tr> <tr> <td>[0]</td><td>[1]</td><td>[2]</td><td>[3]</td></tr> </table> <p>FRONT: 2 REAR: 0</p>	YELLOW			WHITE	[0]	[1]	[2]	[3]	<p>INSERT_CIRCQ (COLOURS, 2, 0, 4, 'VIOLET')</p>	<p>COLOURS [0:3]</p> <table border="1" data-bbox="2024 3431 2809 3727"> <tr> <td>YELLOW</td><td>VIOLET</td><td></td><td>WHITE</td></tr> <tr> <td>[0]</td><td>[1]</td><td>[2]</td><td>[3]</td></tr> </table> <p>FRONT: 2 REAR: 1</p>	YELLOW	VIOLET		WHITE	[0]	[1]	[2]	[3]	<p>Insert 'VIOLET' successful</p>
YELLOW			WHITE																	
[0]	[1]	[2]	[3]																	
YELLOW	VIOLET		WHITE																	
[0]	[1]	[2]	[3]																	

## Other Types of Queues

### 5.4

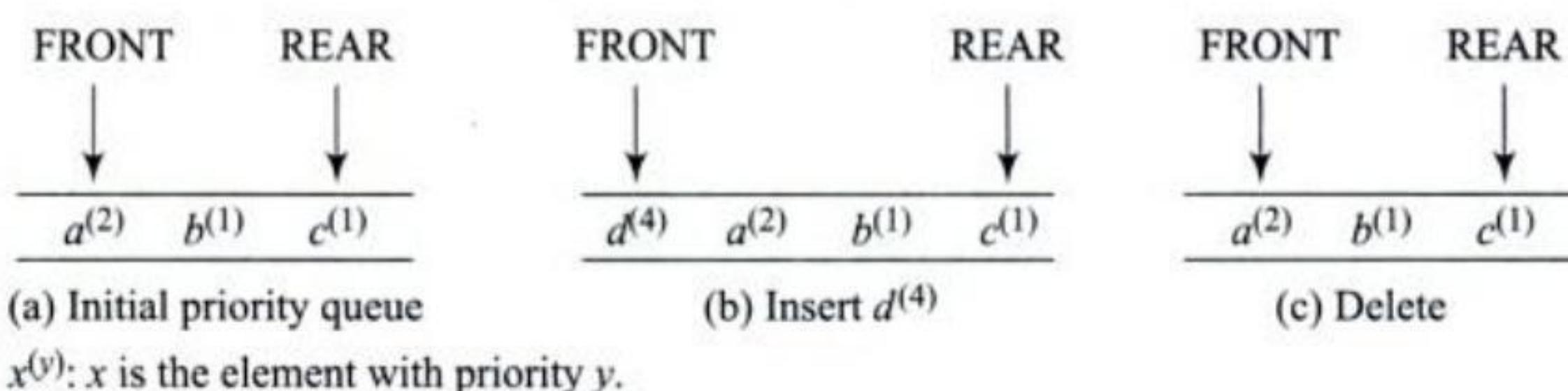
#### Priority queues

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

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

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

A common method of implementation of a priority queue is to open as many queues as there are priority factors. A low priority queue will be operated for deletion only when all its high priority predecessors are empty. In other words, deletion of an element in a priority queue  $q_i$  with priority  $p_i$  is possible only when those queues  $q_j$  with priorities  $p_j$  ( $p_j > p_i$ ) are empty. However, with regard to insertions, an element  $e_k$  with priority  $p_1$  joins the respective queue obeying the scheme of FIFO with regard to the queue  $q_i$  alone.



**Fig. 5.7 A priority queue**

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

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

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

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

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

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

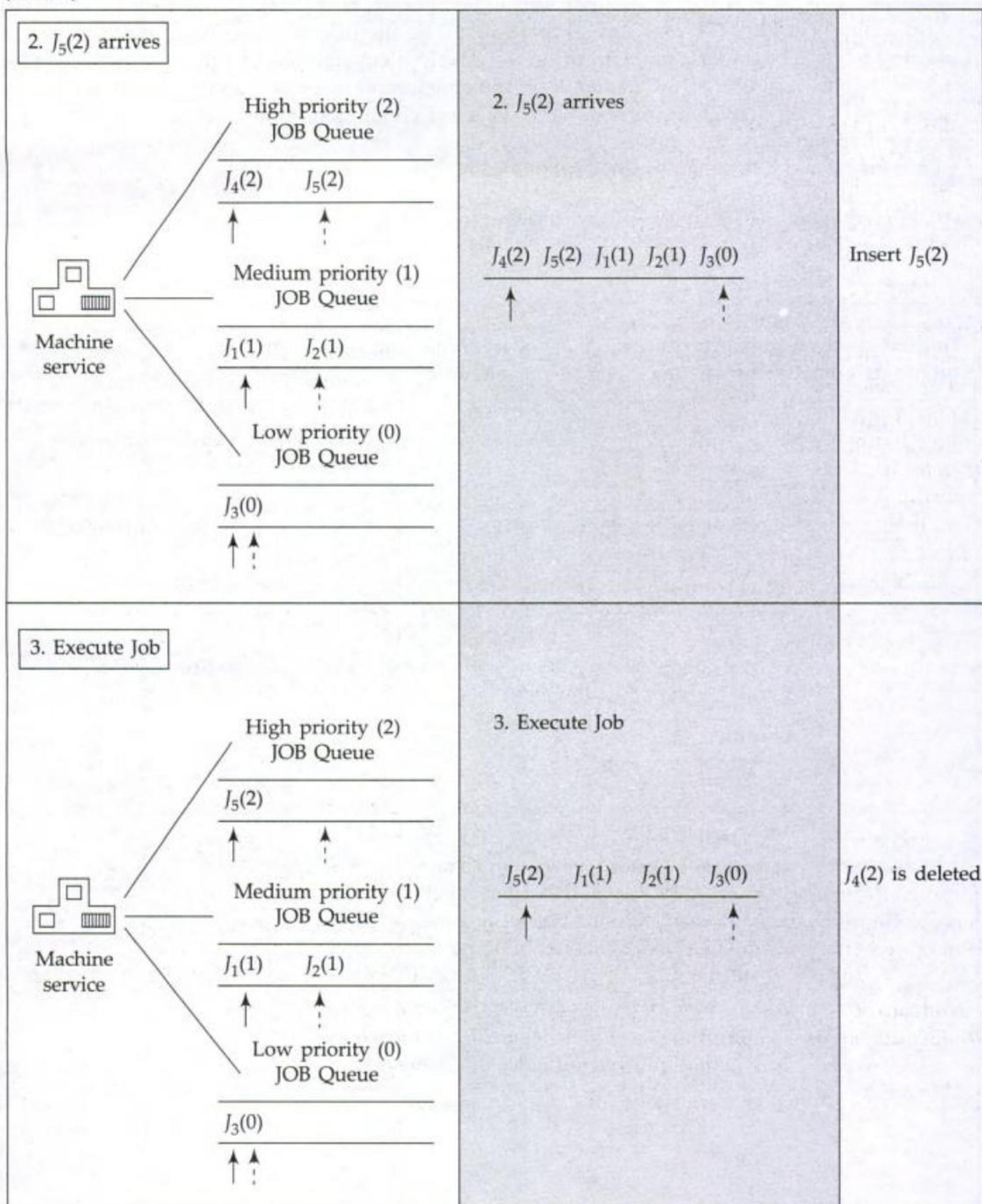
Operations on the JOB queue in the chronological order :

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

Implementation of a priority queue as a cluster of queues	Implementation of a priority queue by sorting queue elements	Remarks
<p>Initial configuration</p> <p>Machine service</p>	<p>Initial configuration</p> <p><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>

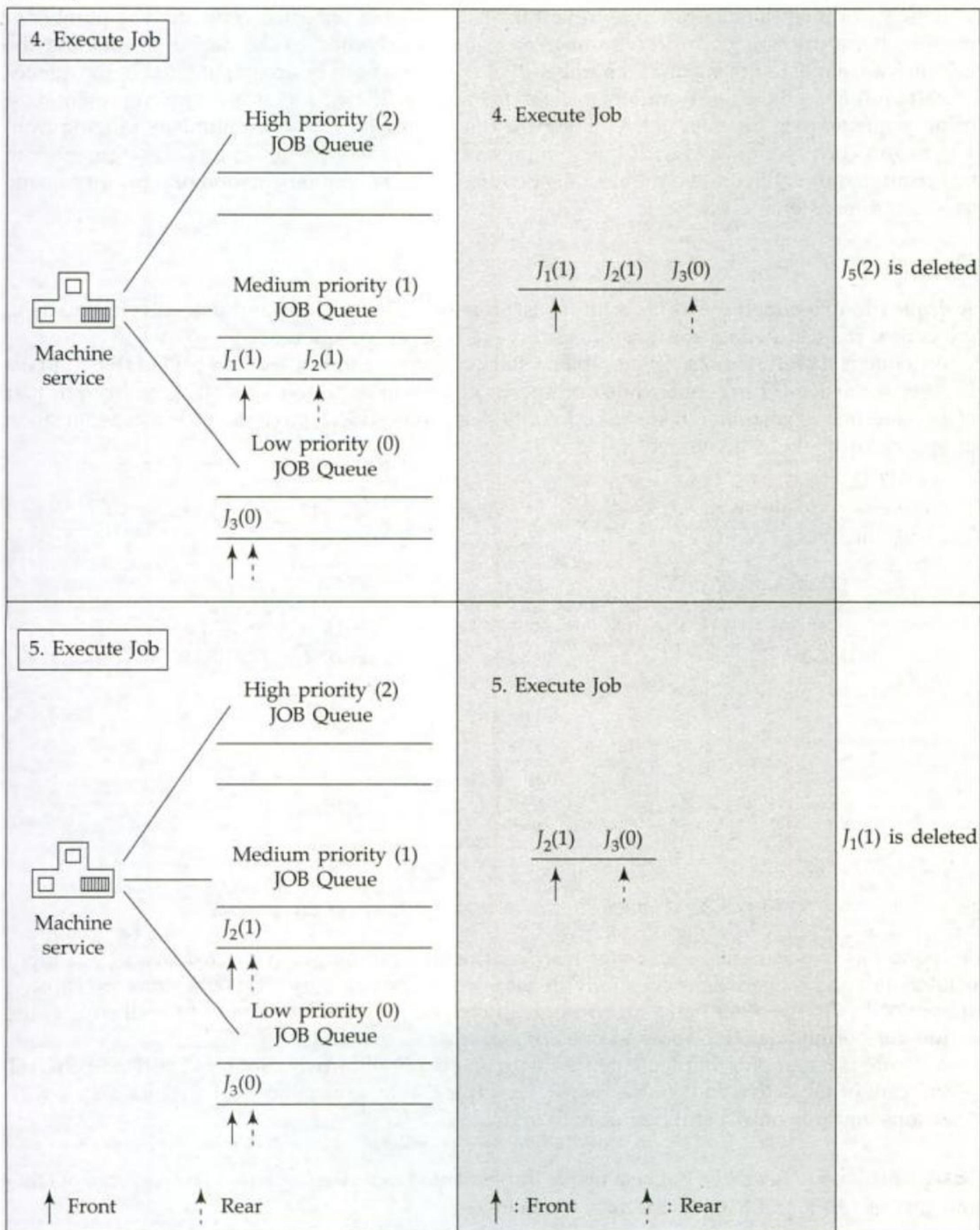
(Contd.)

(Contd.)



(Contd.)

(Contd.)



A variant of the implementation of a priority queue using multiple queues is to make use of a single two dimensional array to represent the list of queues and their contents. The number of rows in the array is equal to the number of priorities accorded to the data elements and the columns are equal to the maximum number of elements that can be accommodated in the queues corresponding to the priority number. Thus, if PRIO\_QUE[1:m, 1:n] is an array representing a priority queue, then the data items joining the queue may have priority numbers ranging from 1 to  $m$  and corresponding to each queue representing a priority, a maximum of  $n$  elements can be accommodated. Illustrative problem 5.4 demonstrates the implementation of a priority queue as a two dimensional array.

## Deques

A *deque* (double ended queue) is a linear list in which all insertions and deletions are made at the end of the list. A deque is pronounced as 'deck' or 'de queue'.

A deque is therefore more general than a stack or queue and is a sort of FLIFLO (First in Last In or First out Last Out). Thus while one speaks of the top or bottom of a stack, or front or rear of a queue, one refers to the *right end* or *left end* of a deque. The fact that deque is a generalization of a stack or queue is illustrated in Fig. 5.8.

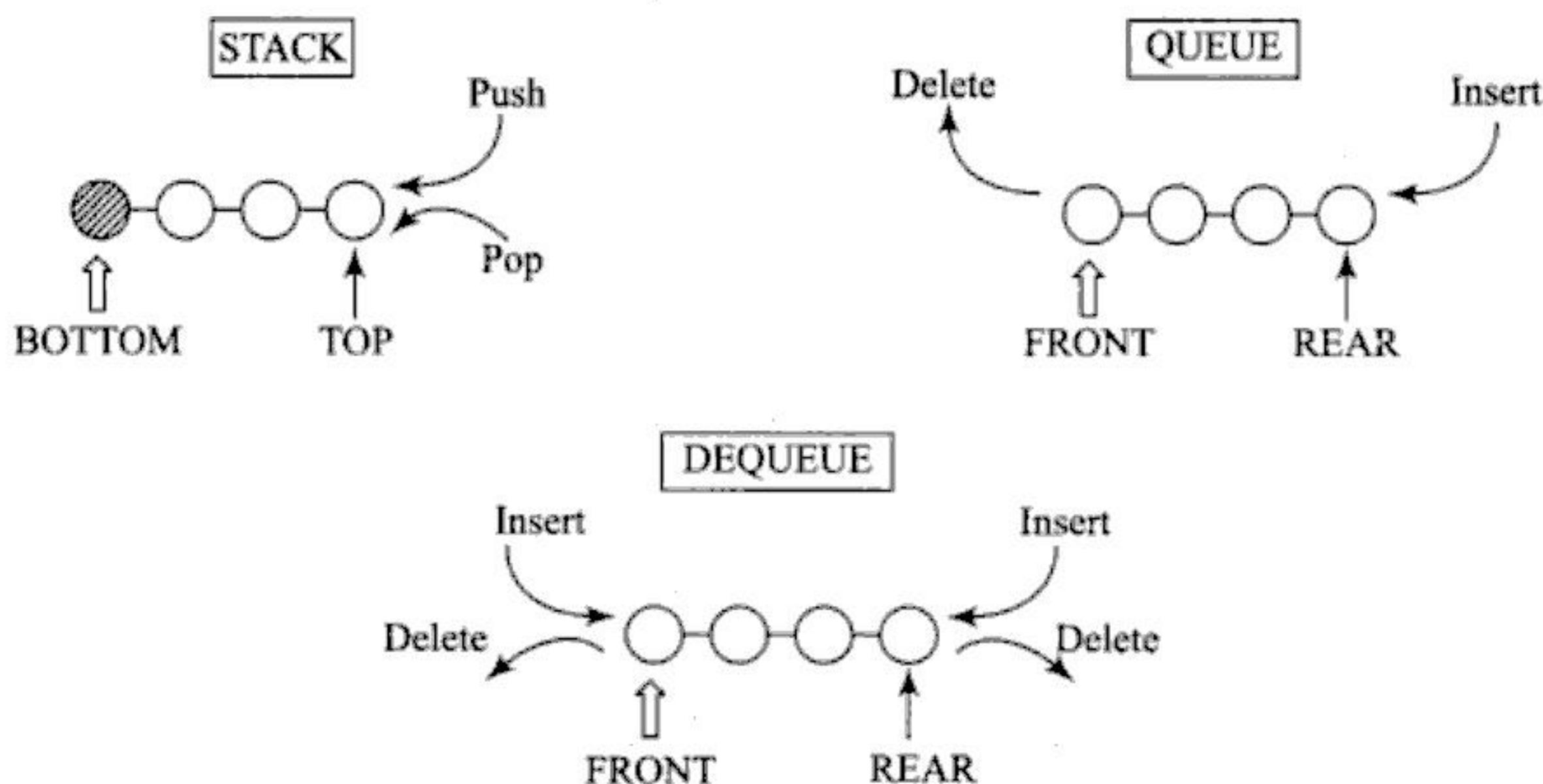


Fig. 5.8 A stack, a queue and a deque—a comparison

A deque has two variants, viz., *input restricted* deque and *output restricted* deque. An input restricted deque is one where insertions are allowed at one end only while deletions are allowed at both ends. On the other hand, an output restricted deque allows insertions at both ends of the deque but permits deletions only at one end.

A deque is commonly implemented as a circular array with two variables LEFT and RIGHT taking care of the active ends of the deque. Example 5.4 illustrates the working of a deque with insertions and deletions permitted at both ends.

**Example 5.4** Let DEQ[1:6] be a deque implemented as a circular array. The contents of DEQ and that of LEFT and RIGHT are as given below:

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

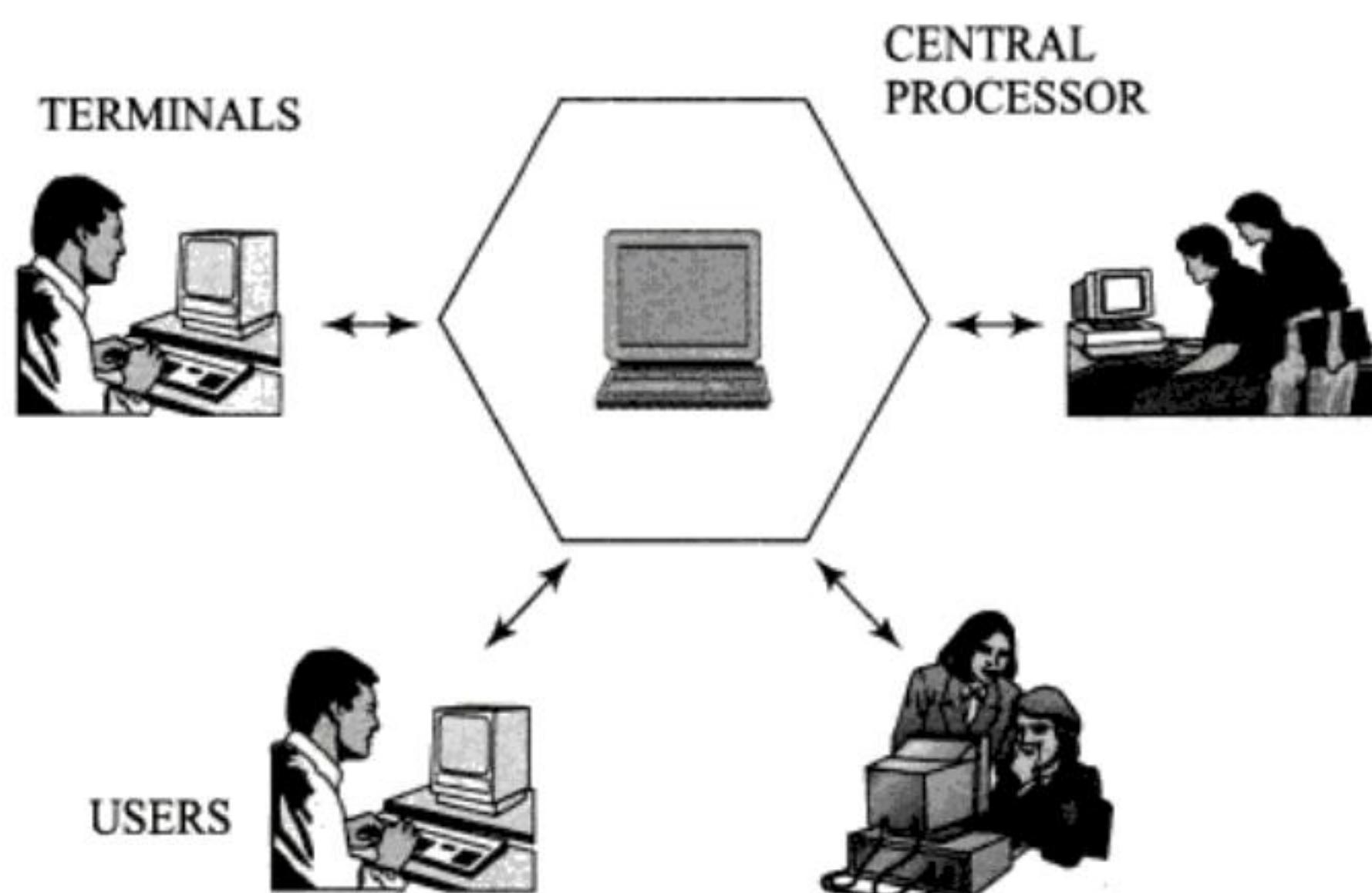
## Applications

5.5

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

### Application of a linear queue

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



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

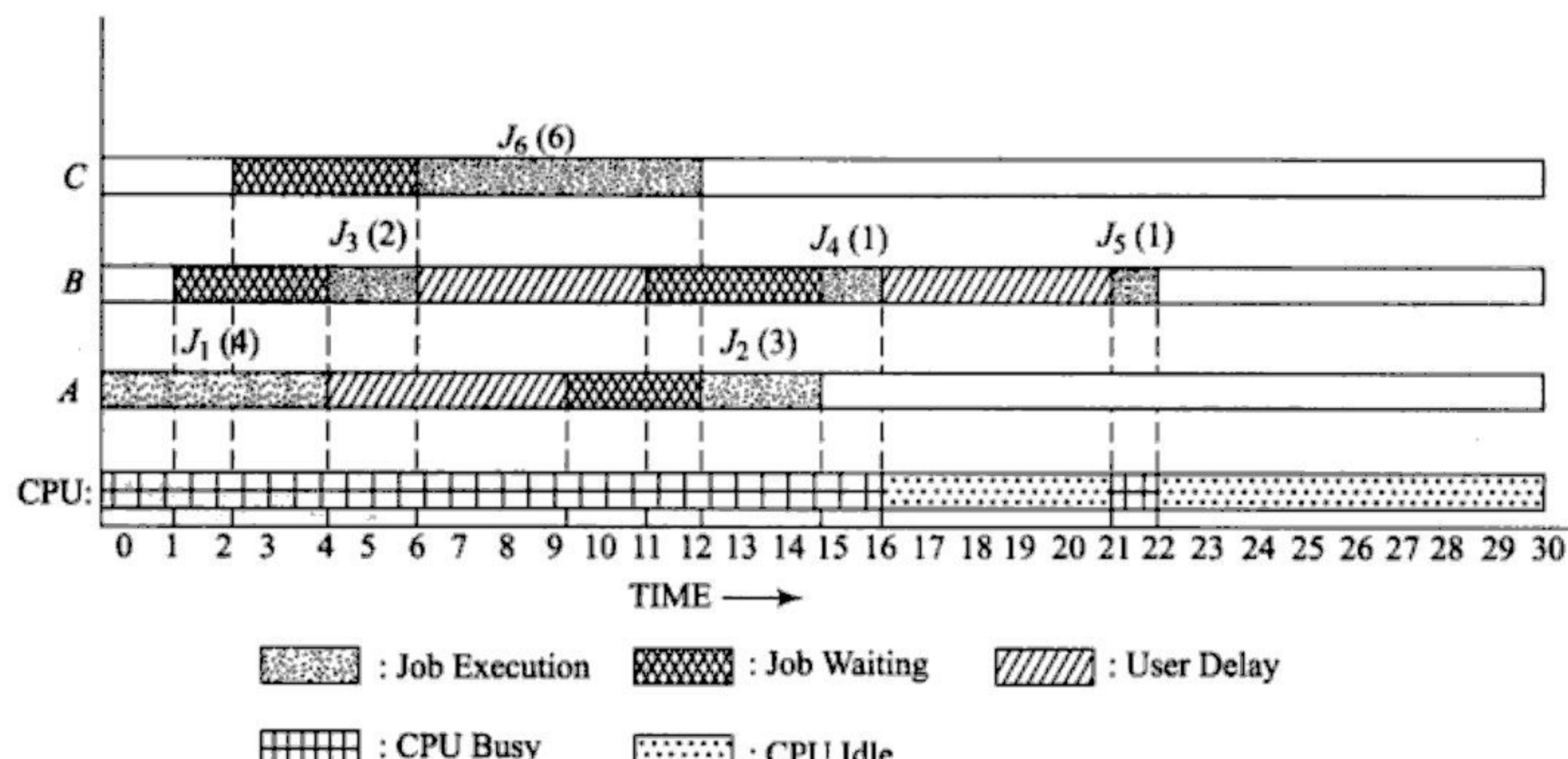
and memory resources is done by allotting a definite time slice of the processor's attention on the users and in a round-robin fashion. In a system such as this, the users are unaware of the presence of other users and are led to believe that their job receives the undivided attention of the CPU. However,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

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

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

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

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



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

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

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

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

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

## Application of priority queues

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

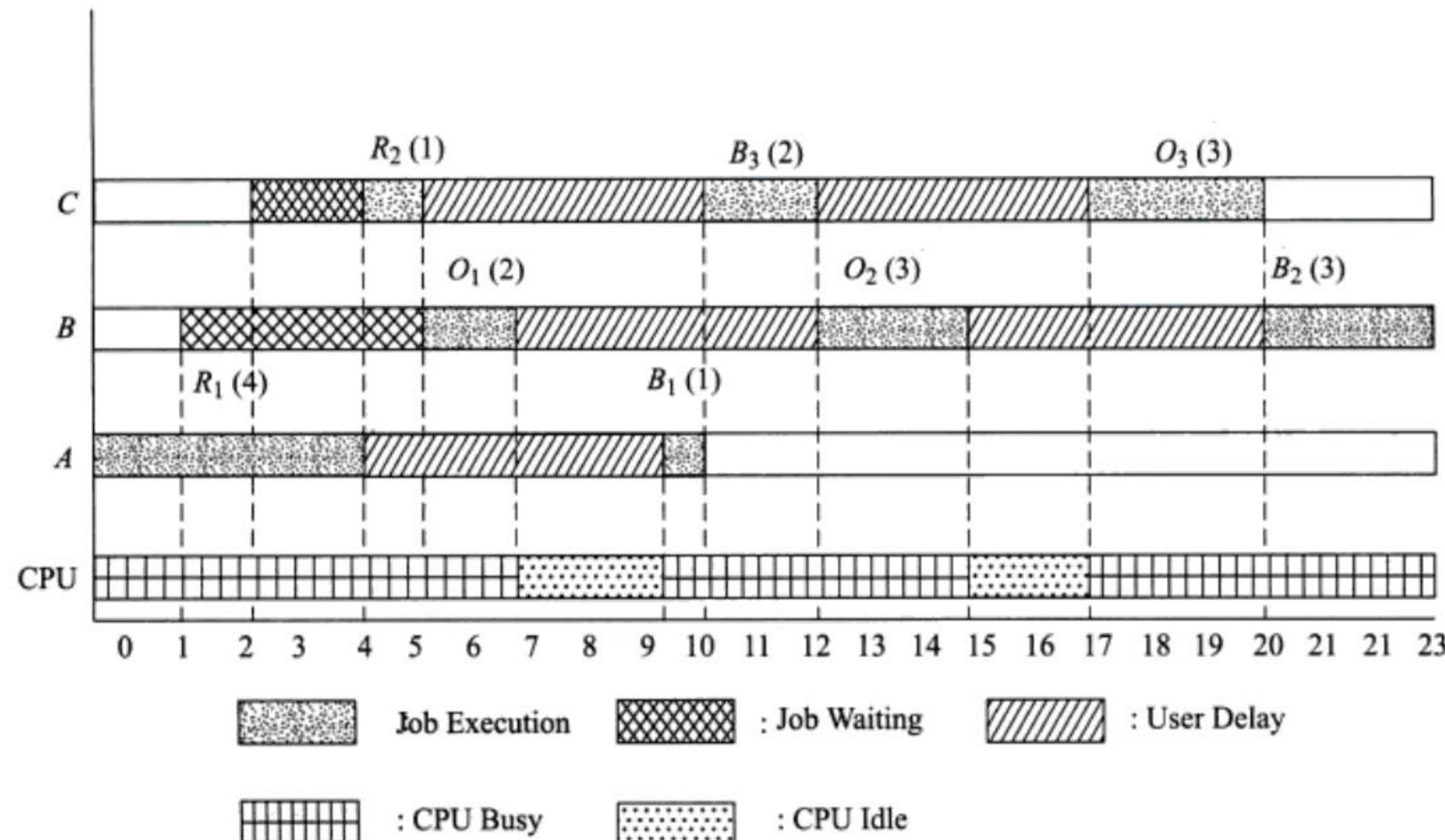
execution of a job request from  $O$ . In the case of queue  $B$ , before executing a job in queue  $B$ , the queues  $R$  and  $O$  should be empty. Example 5.6 illustrates the application of a priority queue in a time-sharing system with priority-based job requests.

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

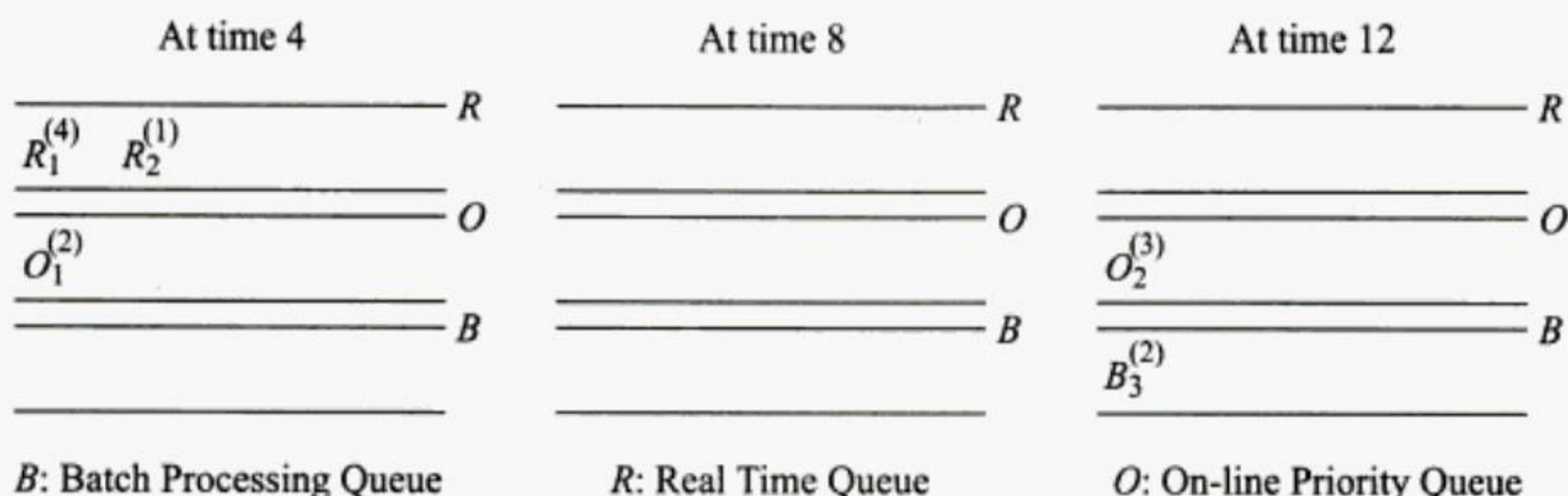
User	Job requests and their execution time in $\mu$ secs		
$A$	$R_1(4)$ $B_1(1)$		
$B$	$O_1(2)$	$O_2(3)$	$B_2(3)$
$C$	$R_2(1)$	$B_3(2)$	$O_3(3)$

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

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



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



**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.

## Illustrative Problems

**Problem 5.1** Let INITIALISE ( $Q$ ) be an operation which initializes a linear queue  $Q$  to be empty. Let ENQUEUE ( $Q, ITEM$ ) insert an  $ITEM$  into  $Q$  and DEQUEUE ( $Q, ITEM$ ) delete an element from  $Q$  through  $ITEM$ . EMPTY\_QUEUE ( $Q$ ) is a Boolean function which is true if  $Q$  is empty and false otherwise, and PRINT ( $ITEM$ ) is a function which displays the value of  $ITEM$ .

What is the output of the following pseudo code?

- |                        |                                                    |
|------------------------|----------------------------------------------------|
| 1. $X = Y = Z = 0;$    | 9. ENQUEUE ( $Q, Y+18$ )                           |
| 2. INITIALISE ( $Q$ )  | 10. DEQUEUE ( $Q, X$ )                             |
| 3. ENQUEUE ( $Q, 10$ ) | 11. DEQUEUE ( $Q, Y$ )                             |
| 4. ENQUEUE ( $Q, 70$ ) | 12. <b>while</b> not EMPTY_QUEUE ( $Q$ ) <b>do</b> |
| 5. ENQUEUE ( $Q, 88$ ) | 13. DEQUEUE ( $Q, X$ )                             |
| 6. DEQUEUE ( $Q, X$ )  | 14. PRINT ( $X$ )                                  |
| 7. DEQUEUE ( $Q, Z$ )  | 15. <b>end</b>                                     |
| 8. ENQUEUE ( $Q, X$ )  |                                                    |

**Solution:** The contents of the queue  $Q$  and the values of the variables  $X, Y, Z$  are tabulated below:

Step	Queue $Q$	Variables		
		X	Y	Z
1-2		0	0	0
3	10	0	0	0
4	10 70	0	0	0
5	10 70 88	0	0	0
6	70 88	10	0	0
7	88	10	0	70
8	88 10	10	0	70
9	88 10 18	10	0	70
10	10 18	88	0	70
11	18	88	10	70
12-14		18	10	70

The output of the program code is : 18

**Problem 5.2** Given  $Q'$  to be a circular queue implemented as an array  $Q'[0:4]$  and using procedures declared in problem I = 5.1, but suitable for implementation on  $Q'$ , what is the output of the following code? Illustrative Problem 5.1

[Note: The procedures ENQUEUE ( $Q'$ ,  $X$ ) and DEQUEUE ( $Q'$ ,  $X$ ) may be assumed to be implementation of Algorithms 5.3, 5.4]

- |                           |                                           |
|---------------------------|-------------------------------------------|
| 1. INITIALISE ( $Q'$ )    | 10. ENQUEUE ( $Q'$ , $Y$ )                |
| 2. $X := 56$              | 11. $Z = X - Y$                           |
| 3. $Y := 77$              | 12. if ( $Z = 0$ )                        |
| 4. ENQUEUE ( $Q'$ , $X$ ) | 13. then (while not EMPTY_QUEUE ( $Q'$ )) |
| 5. ENQUEUE ( $Q'$ , 50)   | 14. DEQUEUE ( $Q'$ , $X$ )                |
| 6. ENQUEUE ( $Q'$ , $Y$ ) | 15. PRINT ( $X$ )                         |
| 7. DEQUEUE ( $Q'$ , $Y$ ) | 16. end }                                 |
| 8. ENQUEUE ( $Q'$ , 22)   | 17. else PRINT ("Process Complete");      |
| 9. ENQUEUE ( $Q'$ , $X$ ) |                                           |

**Solution:** The contents of the circular queue  $Q' [0:4]$  and the values of the variable  $x$ ,  $y$ ,  $z$  are illustrated below.

Steps	Queue $Q'$					Variables		
						$X$	$Y$	$Z$
1	[ ]	[ ]	[ ]	[ ]	[ ]			...
	[0]	[1]	[2]	[3]	[4]			...
2-3	[ ]	[ ]	[ ]	[ ]	[ ]	56	77	...
	[0]	[1]	[2]	[3]	[4]			...
4	[ ]	56	[ ]	[ ]	[ ]	56	77	...
	[0]	[1]	[2]	[3]	[4]			...
5	[ ]	56	50	[ ]	[ ]	56	77	...
	[0]	[1]	[2]	[3]	[4]			...
6	[ ]	56	50	77	[ ]	56	77	...
	[0]	[1]	[2]	[3]	[4]			...
7	[ ]	[ ]	50	77	[ ]	56	56	...
	[0]	[1]	[2]	[3]	[4]			...
8	[ ]	[ ]	50	77	22	56	56	...
	[0]	[1]	[2]	[3]	[4]			...
9	56	[ ]	50	77	22	56	56	...
	[0]	[1]	[2]	[3]	[4]			...
10	56	[ ]	50	77	22	56	56	...
	[0]	[1]	[2]	[3]	[4]			...

Queue full. ENQUEUE ( $Q'$ ,  $Y$ ) fails

(Contd.)

(Contd.)

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

Output of the program code: 50 77 22 56

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

```

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

```

*Solution:*

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

(Contd.)

(Contd.)

7-10	<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
			Output: 21		
15-18			53	85	32
			53	85	53
			53	85	85
			Output: 32	53	85

The final output is: 21 32 53 85

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

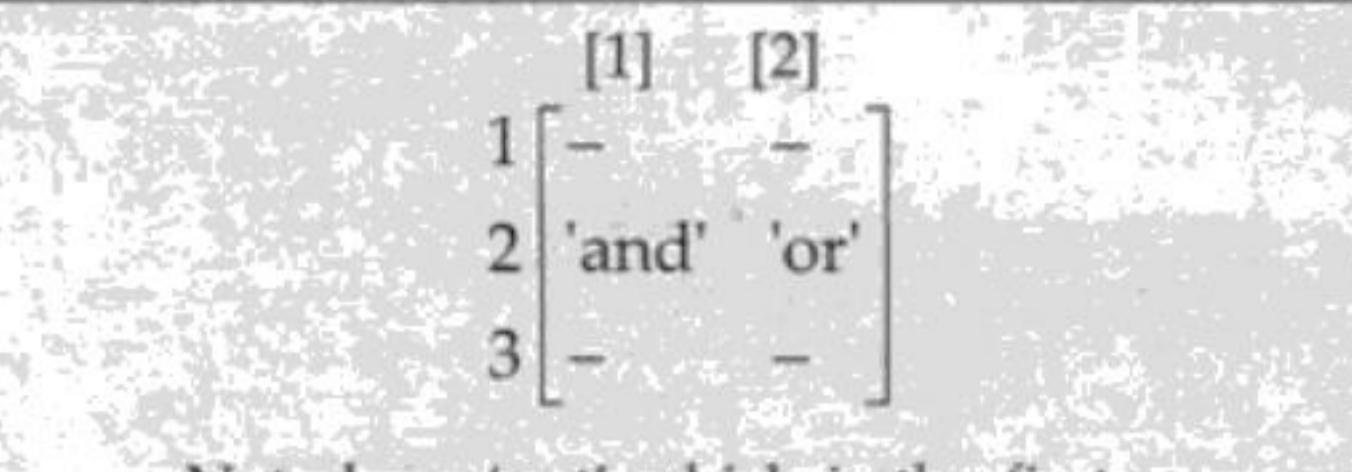
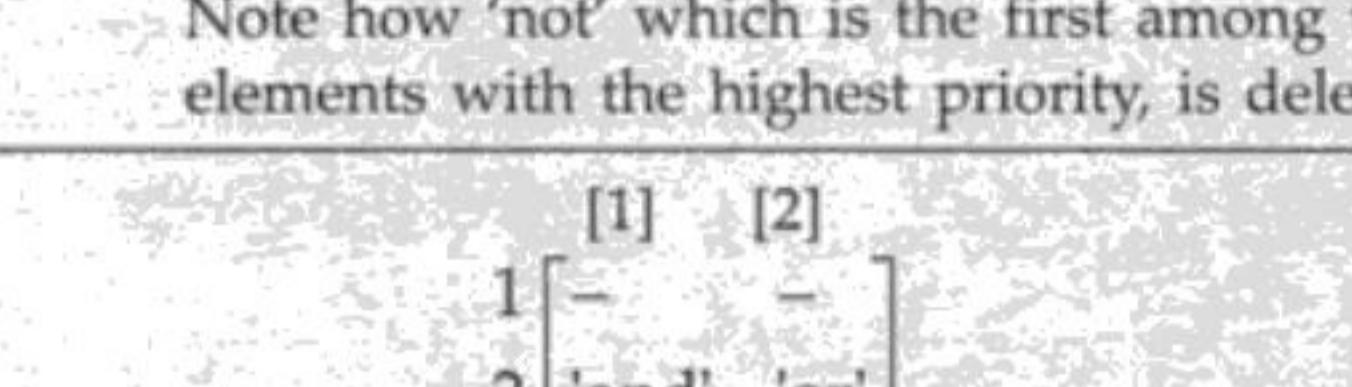
- (i) INSERT('not', 1)
- (ii) INSERT('and', 2)
- (iii) INSERT('or', 2)
- (iv) DELETE( )
- (v) INSERT('equ', 3);

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

TOKEN: [1] [2]

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

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

(i) INSERT ('not', 1) (ii) INSERT ('and', 2) (iii) INSERT ('or', 2)	
(iv) DELETE ( )	
(v) INSERT('equ', 3);	<p>Note how 'not' which is the first among the elements with the highest priority, is deleted</p> 

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

DEO:

LEFT: 2

RIGHT: 5

Execute the following insertions and deletions on DEO:

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

**Solution:**

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

DEQ:

LEFT: 1

RIGHT: 6

(ii) DEQ after the execution of DELETE( )

DEO

LEET: 2

PICHT, 6

[1]	[2]	[3]	[4]	[5]	[6]
C1	M1	N7	N6	K9	

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

DEQ:

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

LEFT: 1

RIGHT: 6

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



## Review Questions

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

FRONT: 2 REAR: 3

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

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

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

FRONT: 3 REAR: 4

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

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

- Invoking Queue full condition soon after ENQUEUE( Q, 'H') operation
  - Aborting the ENQUEUE( Q, 'T') operation
  - Yielding FRONT = 1 and REAR = 4, after the operations.
  - Yielding FRONT = 3 and REAR =1, after the operations
    - (i)
    - (ii)
    - (iii)
    - (iv)
- State whether true or false:  
 For the following implementation of a queue, where FRONT and REAR point to the physical front and rear of the queue,

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

Execution of the operation ENQUEUE(Q, 'C'),

- (i) if Q is a linear queue, would invoke the Queue full condition
- (ii) if Q is a circular queue would abort the enqueueing operation
- (a) (i) true (ii) true    (b) (i) true (ii) false    (c) (i) false (ii) false    (d) (i) false (ii) true

6. What are the disadvantages of linear queues?
7. How do circular queues help overcome the disadvantages of linear queues?
8. If FRONT and REAR were pointers to the physical front and rear of a linear queue, comment on the condition, FRONT = REAR.
9. If FRONT and REAR were pointers to the physical front and rear of a circular queue, comment on the condition, FRONT = REAR.
10. How are priority queues implemented using a single queue?
11. The following is a table of five users Tim, Shiv, Kali, Musa and Lobo, with their job requests  $J_i(k)$  where  $i$  is the job number and  $k$  is the time required to execute the job. The time at which the users logged in are also shown in the table.

User	Job requests and the execution time in $\mu$ secs.	Login time
Tim	$J_1(5), J_2(4)$	0
Shiv	$J_3(3), J_4(5), J_5(1)$	1
Kali	$J_6(6), J_7(3)$	2
Musa	$J_8(5), J_9(1)$	3
Lobo	$J_{10}(3), J_{11}(6)$	4

Throughout the simulation, assume a uniform user delay period of  $4 \mu$  secs between any two sequential job requests initiated by a user. Also to simplify simulation, assume that the CPU gives whole attention to the completion of a job request before moving to the next job request. Trace a graphical illustration of the simulation to demonstrate a time sharing system at work. Show snapshots of the linear queue used by the system, to implement the FIFO principle of attending to jobs by the CPU.

12. For the time sharing system discussed in Review Question 11 of Chapter 5, trace a graphical illustration of the simulation assuming that all job requests  $J_i(k)$  where  $i$  is even numbered have higher priority than those jobs  $J_i(k)$  where  $i$  is odd numbered. Show snapshots of the priority queue implementation.



## Programming Assignments

1. Waiting line simulation in a post office:

In a post office, a lone postal worker serves a single queue of customers. Every customer receives a token # (serial number) as soon as he/she enters the queue. After service, the token is returned to the postal worker and the customer leaves the queue. At any point of time the worker may want to know how many customers are yet to be served.

- (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  $a_1^{(p_1)}, a_2^{(p_2)}, \dots, a_n^{(p_n)}$  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.



# LINKED LISTS

In Part I of the book we dealt with arrays, stacks and queues which are linear sequential data structures (of these, stacks and queues have a linked representation as well, which will be discussed in Chapter 7)

In this chapter we detail linear data structures having a linked representation. We first list the demerits of the sequential data structure before introducing the need for a linked representation. Next, the linked data structures of singly linked list, circularly linked list, doubly linked list and multiply linked list are elaborately presented. Finally, two problems, viz., Polynomial addition and Sparse matrix representation, demonstrating the application of linked lists are discussed.

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

## Introduction

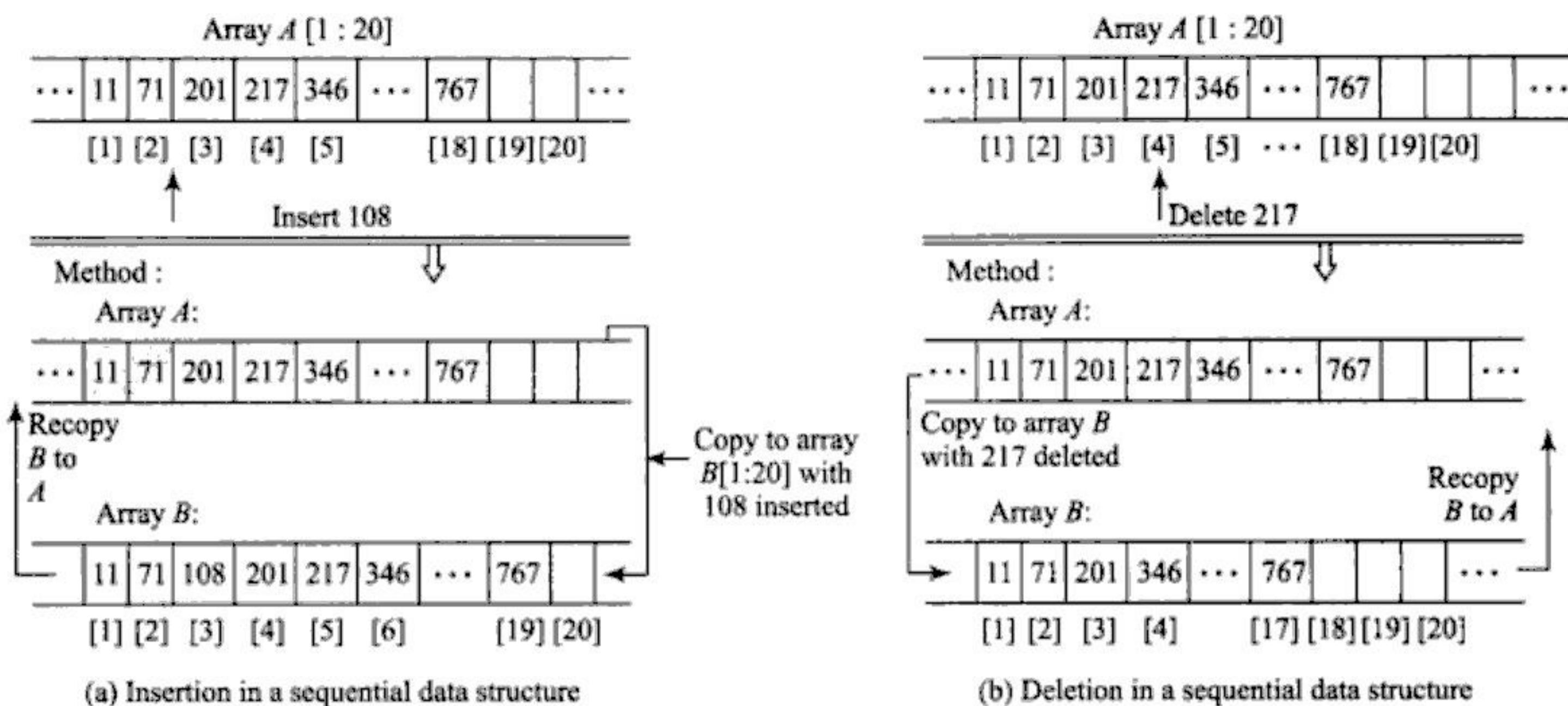
6.1

### Drawbacks of sequential data structures

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

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

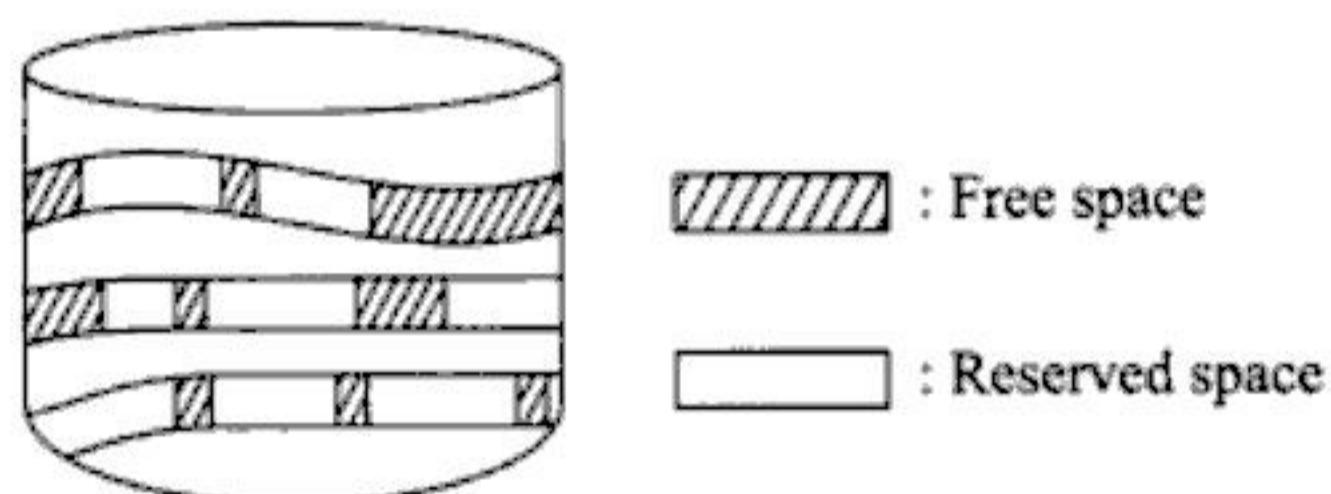
Let us consider an array  $A[1 : 20]$ . This means a contiguous set of twenty memory locations have been made available to accommodate the data elements of  $A$ . As shown in Fig. 6.1(a), let us suppose the array is partially full. Now, to insert a new element 108 in the position indicated, it is not 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)



**Fig. 6.1** Drawbacks of sequential data structures—Inefficient implementation of Insertion/Deletion operations

With regard to the second drawback of inefficient storage memory management, the need for allotting contiguous memory locations for every array declaration is bound to leave fragments of free memory space unworthy of allotment for future requests. This eventually may lead to inefficient storage management. In fact, fragmentation of memory is a significant problem to be reckoned with in computer science. Several methods have been proposed to counteract this problem.

Figure 6.2 shows a simple diagram of a storage memory with fragmentation of free space.



**Fig. 6.2** Drawbacks of sequential data structures—Inefficient storage memory management

Note how fragments of free memory space, though put together, can be a huge chunk of free space, the lack of contiguity renders them unworthy of accommodating sequential data structures.

### Merits of linked data structures

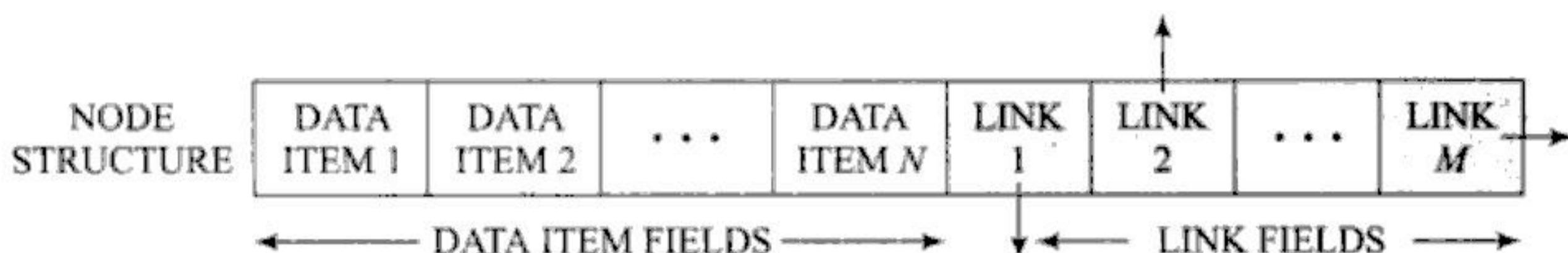
A linked representation serves to counteract the drawbacks of sequential representation by exhibiting the following merits:

- Efficient implementation of insertion and deletion operations. Unlike sequential data structures, there is complete absence of data movement of neighbouring elements during the execution of these operations.

- (ii) Efficient use of storage memory. The operation and management of linked data structures are less prone to create memory fragmentation.

A linked representation of data structure known as a *linked list* is a collection of *nodes*. Each node is a collection of *fields* categorized as *data items* and *links*. The data item fields hold the information content or data to be represented by the node. The link fields hold the addresses of the neighbouring nodes or of those nodes which are associated with the given node as dictated by the application.

Figure 6.3 illustrates the general node structure of a linked list. A node is represented by a rectangular box and the fields are shown by partitions in the box. Link fields are shown to carry arrows to indicate the nodes to which the given node is linked or connected.



**Fig. 6.3 A general structure of a node in a linked list**

This implies that unlike arrays, no two nodes in a linked list need be physically contiguous. All the nodes in a linked list data structure may in fact be strewn across the storage memory making effective use of what little space is available to represent a node. However, the link fields carry on themselves the onerous responsibility of remembering the addresses of the other neighbouring or associated nodes, to keep track of the data elements in the list.

In programming language parlance, the link fields are referred to as *pointers*. In this book, pointers and link fields will be interchangeably used in several contexts.

To implement linked lists the following mechanisms are essential:

- i) A mechanism to frame chunks of memory into nodes with the desired number of data items and fields.

In most programming languages, this mechanism is implemented by making use of a 'record' or 'structure' or its look-alikes or even associated structures, to represent the node and its fields.

- ii) A mechanism to determine which nodes are free and which have been allotted for use.

- iii) A mechanism to obtain nodes from the free storage area or storage pool for use.

These are fully provided and managed by the system. There is very little that an end user or a programmer can do to handle this mechanism by oneself. This is made possible in many programming languages by the provision of inbuilt functions which help execute requests for a node with the specific fields. In this book, we make use of a function GETNODE (X) to implement this mechanism. The GETNODE (X) function allots a node of the desired structure and the address of the node viz. X, is returned. In other words, X is an output parameter of the function GETNODE (X), whose value is determined and returned by the system.

- iv) A mechanism to return or dispose of nodes from the reserved area or pool to the free area after use.

This is also made possible in many programming languages by providing an in-built function which helps return or dispose of the node after use. In this book we make use of the function RETURN(X) to implement this mechanism. The RETURN(X) function returns

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.

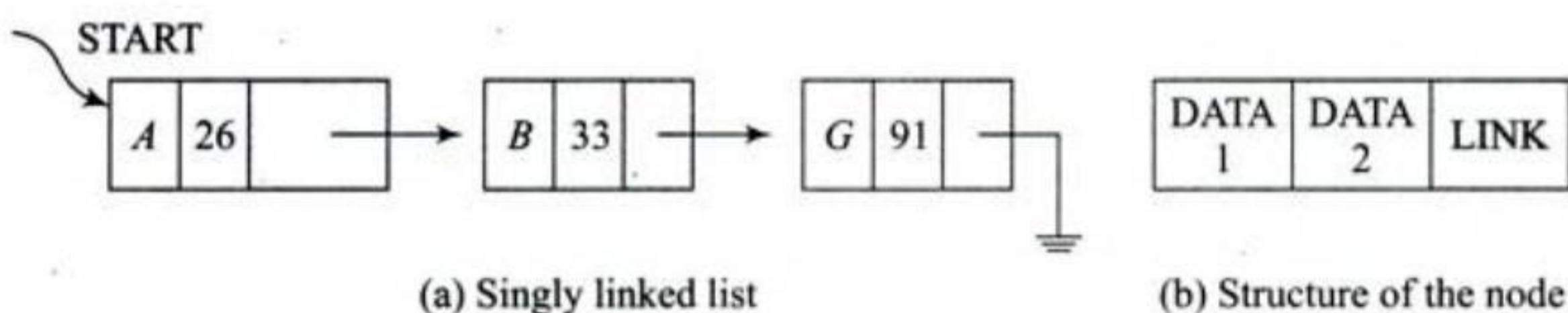
## Singly Linked Lists

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.

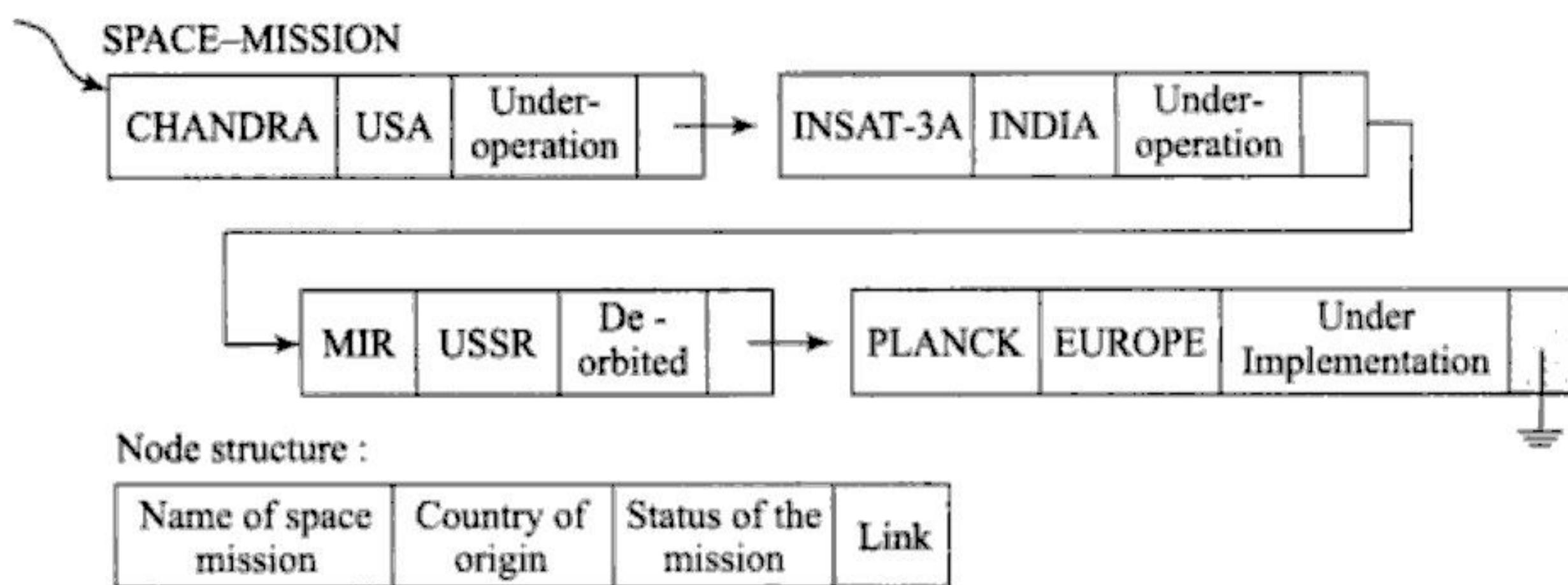


**Fig. 6.4 A singly linked list and its node structure**

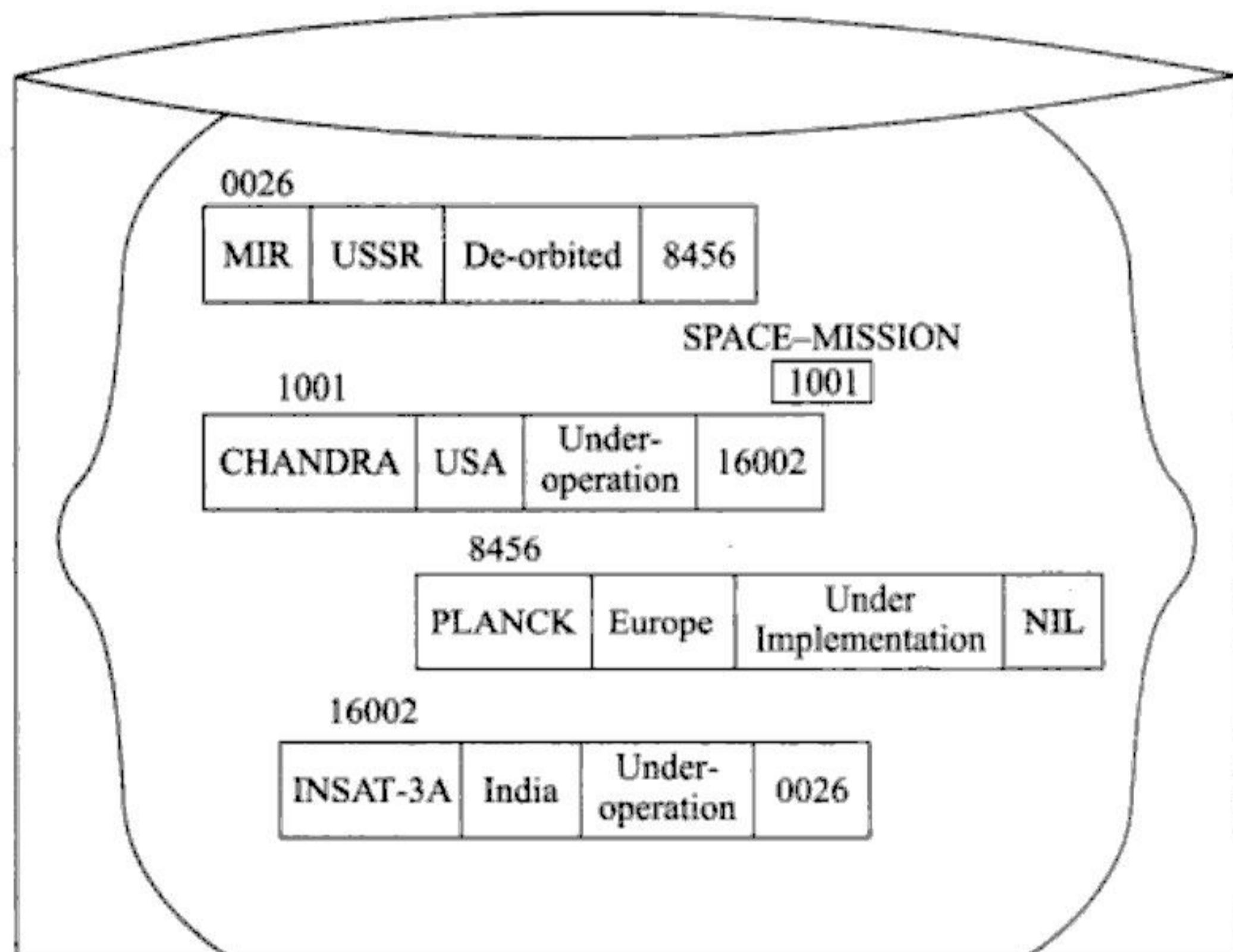
Every node which is basically a chunk of memory, carries an address. When a set of data elements to be used by an application are represented using a linked list, each data element is represented by a node. Depending on the information content of the data element, one or more data items may be opened 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



(a) Logical representation of SPACE-MISSION



(b) Physical representation of SPACE-MISSION

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

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

### Insertion and deletion in a singly linked list

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

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

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

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

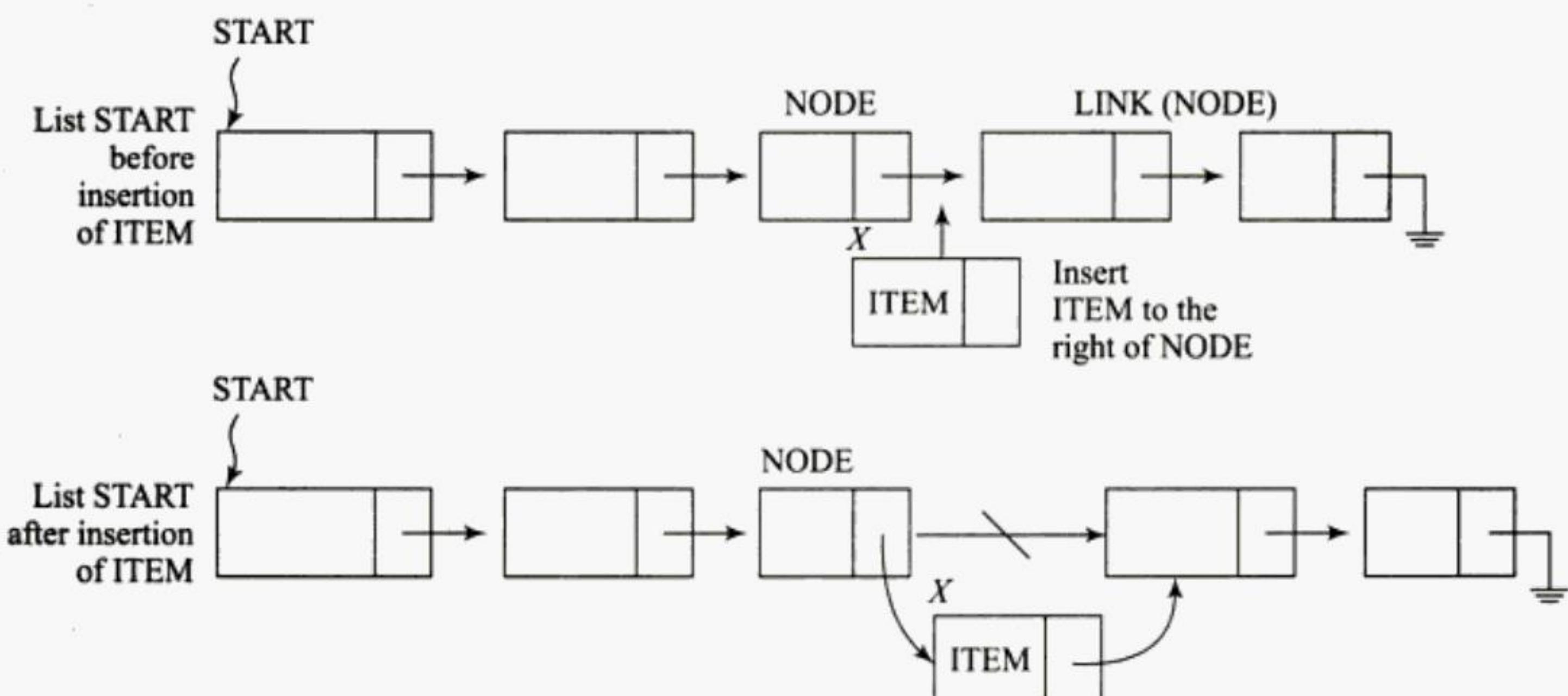


Fig. 6.6 Logical representation of insertion in a singly linked list

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

```

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

```

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

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

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

```

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

```

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

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

- (i) 

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

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

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

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

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

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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



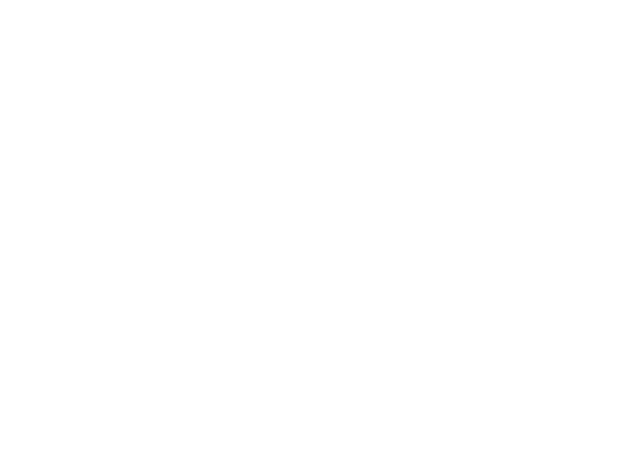
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



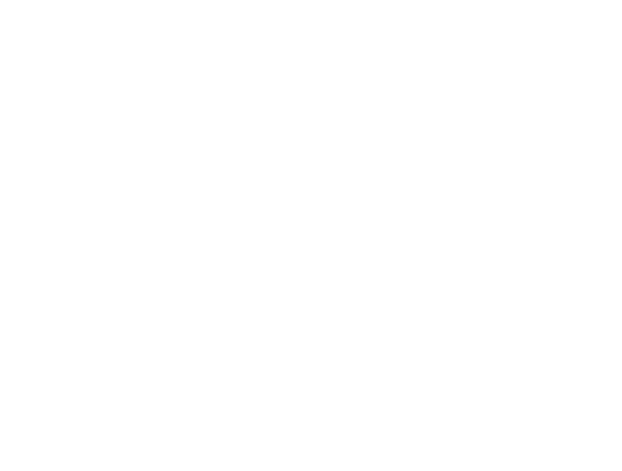
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



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



# INDEX

- 2-3** trees 270
- 2-3-4** trees 294
- 2-4** trees 270, 294
- Abstract Data Type **5**
- Addition of polynomials 106
- Adjacency list 198
  - matrix 195
  - matrix representation 195
- ADT
  - arrays **34**
  - binary trees 174
  - graphs 208
  - links 110
  - queues **75**
  - singly linked lists 111
  - stacks **48**
- Algorithm **2**
  - definition **3**
  - development **4**
  - properties **3**
  - structure **3**
- Alternate keys 355
- Amortized analysis of splay trees 317
- Apriori analysis **9**
  - recursive functions **17**
  - analysis **17**
  - approach **9**
- Array **27**
  - ADT **34**
  - multi-dimensional **28**
  - number of elements **27**
  - one-dimensional **27**
  - operations **27**
  - representation **28**
  - two-dimensional **27**
- Asymptotic notations **11**
- Available space 130
- Average case complexity **14**
- AVL search tree 229
  - deletion 236
  - insertion 230
  - retrieval 230
  - tree 229
- B tree of order  $m$  293
  - definition 269
  - deletion 273
  - height 277
  - inserting 270
  - searching 270
  - trees 269
  - trees of order **4** 293
- B+ trees 283
- Balance factor 229
- Balanced  $k$ -way merging on disks 442
  - merge sort 438, 441
  - $P$ -way merging on tapes 441
  - trees 228
- Balancing symbols 133
- Base address **29**
- Best case time complexity **14**
- Bin sort 422
- Binary search 378
  - ADT 174
  - basic terminologies 155
  - definition 218
  - deletion 222
  - drawbacks 227
  - growth 168
  - insertion 222
  - representation 156, 219
  - retrieval 220
  - representation 156

- search tree 218  
 tree traversals 158  
 traversals 158, 172  
 trees 155  
 types 155  
**Bisection** 378  
**Black condition** 295  
**Block anchor** 358  
**Branch node** 277  
**Breadth first traversal** 199  
**Bubble sort** 395  
**Bucket sort** 422  
**Buffer handling** 440
- Candidate keys** 355  
**Cascade merge sort** 447  
**Chained hash tables** 340  
**Chaining** 339  
**Circuit matrix** 195  
 matrix representation 197  
**Circular queues** 59, 62  
 operations 62  
**Circularly linked list** 87, 93  
 primitive operations 95  
 representation 93  
**Classification** 6  
**Cluster indexing** 360  
**Collating** 401  
**Collision** 333  
 resolution 338  
**Complexity** 8  
**Construction of heap** 415  
**Conversion of infix expression to postfix expression** 172  
**Cut set matrix** 195  
 matrix representation 197  
**Cycle** 191
- Data abstraction** 6  
 classification 5  
 definition 5  
 structure 2, 5  
 structures and algorithms 4  
 algorithms 4  
 type 5  
**Decision tree**  
 binary search 379  
 Fibonacci search 381  
**Deletion from a binary search tree** 222  
 from an AVL search tree 236  
**Dense index** 358
- Depth first traversal** 201  
**Deque** 70  
**Dequeueing a queue** 56  
**Development of an algorithm** 4  
**Dictionary** 331  
**Digital sort** 422  
**Dijkstra's algorithm** 203  
**Diminishing increment sort** 405  
**Direct file organization** 346, 363  
**Doubly linked lists** 87, 98  
 advantages and disadvantages 99  
 operations 100  
 representation 98  
**Drawbacks of a binary search tree** 227  
 of sequential data structures 84  
**Dynamic memory management** 130
- Enqueuing a queue** 56  
**Evaluation of expressions** 43  
**Exponential time complexities** 12  
**Expression trees** 169  
**External hashing** 363  
 memory 353  
 sorting 394, 435  
 storage devices 353, 436
- Fibonacci merge** 447  
 search 381  
**File indexing** 282  
 operations 356  
 organization 346  
**Files** 353, 354  
**First Come First Served (FCFS)** 56  
**In First Out (FIFO)** 56  
**FLIFLO (First in Last In or First out Last Out)** 70  
**Folding** 334  
**Free storage pool** 130
- Garbage collection** 130  
**Graph** 187  
 complete graphs 189  
 connected graphs 191  
 cut set 193  
 degree 193  
 directed 188  
 empty graph 188  
 Eulerian graph 194  
 Hamiltonian circuit 194  
 isomorphic graphs 193  
 labeled graphs 194

- multigraph 188  
path 190  
subgraph 190  
trees 192  
undirected 188  
**Graph** 188  
    search 384  
**Growth of threaded binary trees** 168
- Hard disks** 436  
**Hash function H** 332  
    functions 333  
    table 332  
**Hashing** 332  
**Head node** 95  
**Heap** 356, 415  
    sort 414  
**Height balanced trees** 228  
**Home bucket** 335  
**Huffman coding** 260
- Incidence matrix** 195  
    matrix representation 196  
**Index** 282  
**Indexed sequential file organization** 358  
    sequential search 385  
**Infix, prefix and postfix expressions** 45  
**Information node** 277  
**Inorder traversal** 158  
**Input buffers** 440  
    restricted deque 20  
**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  
**Lb0, Lb1 and Lb2 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  
**Lr0, Lr1 and Lr2 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