



# Data Structures and Algorithms

Concepts, Techniques and Applications

## The Author

G A Vijayalakshmi Pai, Ph D, is Assistant Professor of Computer Applications at PSG College of Technology, Coimbatore, India. She has over 20 years of experience in teaching graduate students, besides research. Her research interests span Computational Intelligence, Computational Finance and Pattern Recognition. Recipient of the AICTE Career Award for Young Teachers, 2001, awarded by the All India Council of Technical Education, New Delhi, she has published around 40 papers in various international and national journals and conferences, and has also been the investigator for many research projects.

She is also the adaptation author for the Schaum's Outlines Series book on *Data Structures* by Lipschutz published by Mc-Graw Hill Education (India) Ltd., New Delhi. She can be visited at [vijipai@vsnl.com](mailto:vijipai@vsnl.com)

# Data Structures and Algorithms

## Concepts, Techniques and Applications

**G A Vijayalakshmi Pai**

*Department of Computer Applications  
PSG College of Technology  
Coimbatore*



**Tata McGraw-Hill Publishing Company Limited**  
NEW DELHI

*McGraw-Hill Offices*

New Delhi New York St Louis San Francisco Auckland Bogotá Caracas  
Kuala Lumpur Lisbon London Madrid Mexico City Milan Montreal  
San Juan Santiago Singapore Sydney Tokyo Toronto



**Tata McGraw-Hill**

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

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

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

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

ISBN (13): 978-0-07-066726-6  
ISBN (10): 0-07-066726-8

Managing Director: *Ajay Shukla*

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

Asst. Sponsoring Editor: *Shalini Jha*

Editorial Executive: *Nilanjan Chakravarty*

Executive—Editorial Services: *Sohini Mukherjee*

Senior Proof Reader: *Suneeta S Bohra*

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

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

Controller—Production: *Rajender P Ghansela*

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

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

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

Cover Printer: Rashtriya Printers

RQLCRLXRAQQX



## ADVANCE PRAISE

**“For understanding data structure concepts, this book will be of great help to the students because of its simplicity and self-explanatory examples.”**

*Bhupesh Deka  
Sikkim Manipal Institute of Technology*

**“Pseudocode-based algorithms are provided so that implementation can be done using any programming language.”**

*Jibi Abraham  
M. S. Ramaiah Institute of Technology*

**“The presentation and the working of the algorithms is very fresh and unique. This style will be appreciated both by teachers and students alike.”**

*Dr. T. V. Gopal  
Anna University, Chennai*

**“The comparisons between the different types of linked lists are good.”**

*Dr. M. P. Sebastian  
National Institute of Technology, Calicut*



## MORE FROM THE REVIEWERS

“Inclusion of ADT at the end of chapters is a good feature. Also, the programming assignments will help the faculty in teaching the subject.”

“The presentation of this text is very effective and better than the other popular books. The writing style is very precise and effective.”

“I can assure you, I will definitely refer to this text.”

“A wide variety of examples and exercises are given. A Level Order Traversal, for example, is a good exercise and generally missing in other texts.”

“The major strengths are that its description is extremely clear and readable; its organization is excellent, and its exercises are motivating.”

“I would definitely adopt this book and recommend it to my students and friends. It is a nice book that covers all the topics and the advanced topics as well. Any student who would want to master the subject of data structures must read this book.”

## Dedication

**In fond memory of my father, Prof. G A Krishna Pai**

*"...one of the greatest lessons I have learnt in my life is to pay as much attention to the means of work as to its end..."*

*I have been always learning great lessons from that one principle and it appears to be that all the secret of success is there; to pay as much attention to the means as to the end...*

*...With the means alright the end must come...."*

—Swami Vivekananda

(Delivered at Los Angeles, California, 4 Jan, 1900)



# PREFACE

Efficient problem-solving using computers, irrespective of the discipline or application, calls for the design of efficient algorithms. Inclusion of appropriate data structures is of critical importance to the design of efficient algorithms. In other words, *good algorithm design must go hand in hand with appropriate data structures for efficient program design to solve a problem.*

**Data structures** is a fundamental course in Computer Science which most undergraduate and graduate programmes in Computer Science, Computer Science and Engineering, and other allied engineering disciplines such as Computer Integrated Manufacturing, Product Design and Commerce and Communication Engineering , to list a few, offer during the first year or first semester of the programme. It is offered as a core or an elective course, enabling students to have the much needed foundation for efficient programming, leading to better problem-solving in their respective disciplines. Besides regular academic programmes, training programmes of the IT corporate sector and other institutes also offer a course on data structures either by way of certificate courses, diploma or post-diploma programmes.

Most of the well-known textbooks/monographs on this subject have discussed the concepts in relation to a programming language—beginning with Pascal and spanning a spectrum of them such as C, C++, C#, Java, and so on—essentially calling for a fair knowledge of the language, before one proceeds to understand the data structure. There does remain a justification in this, when one argues that the implementation of data structures in a specific programming language needs to be demonstrated or that the algorithms pertaining to the data structure need a convenient medium of presentation and when this is so, why not a programming language?

Again, while some authors have insisted on using their books for an advanced level course, there are some who insist on a working knowledge of the specific programming language as a pre-requisite to using the book. However, in the case of a core course, as it is in most academic programmes, it is not uncommon for a novice or a sophomore, to be bewildered by the ‘miles of code’ that demonstrate or explain a data structure, rendering the subject difficult. In fact, the effort that one needs to put in to comprehend the data structure and its applications, is distracted by the necessity to garner sufficient programming knowledge to follow the code. It is indeed ironical that while a novice is taught data structures to appreciate programming, in reality it turns out that one learns programming to appreciate data structures!

In my decades-old experience of offering the course to graduate programmes which admits students from heterogeneous undergraduate disciplines, with little or less strong knowledge of programming, I had several occasions to observe this malady.

In fact, it is not uncommon for some academic programmes, especially graduate programmes, which due to their shorter duration have a course in Programming and Data Structures running in parallel in the same semester, (much to the chagrin of the novice learner) that a novice is forced to learn data structures through its implementation (in a specific programming language), when in reality it ought to be learning augmented with implementation of the data structures, failure of which has been the reason behind the fallout.

A solution to this problem would be to (i) frame the course such that the theory deals with the concepts, techniques and applications of data structures, not taking recourse to any specific programming language, but instead settling for a pseudo-language which clearly expounds the data structure and supplementing the course material with illustrative problems and exercises to reinforce the students' grasp of the concepts, and (ii) augment the theory with laboratory sessions to enable the student implement the data structure in itself or as embedded in an application, in a language of his/her own choice or as insisted upon in the curriculum. This would enable the student who has acquired sufficient knowledge and insight into the data structures, to appreciate the beauty and the merits of employing the data structure by programming it himself or herself, rather than 'look' for the data structure in a pre-written code.

This means that textbooks catering to the fundamental understanding of the data structure concepts for use as course material in the classroom are as much needed as those books which cater to the implementation of data structures in a programming language for use in the laboratory sessions. While most books in the market conform to the latter, to bring out a book for use as classroom course material by instructors handling a course on data structures and comprehensive enough for the novice students to benefit, has been the main motivation in writing this book. In this direction, the book details concepts, techniques and applications pertaining to data structures, independent of any programming language, discusses several illustrative problems and poses review questions to reinforce the understanding of the theory, and presents a suggestive list of programming assignments to aid implementation of the data structures. In fact, the book may be independently used as a textbook since it is self-contained or serves as a companion for books discussing data structures implemented in a specific programming language such as C, C++, Java, etc.

The book lays an all-round emphasis on Theory, Applications, Illustrative Problems, Review Questions and Programming Assignments to enable the students comprehend, implement and appreciate data structures. The whole book is divided into five parts.

As an introduction, the need for data structures and some basic concepts pertaining to analysis of algorithms which is essential to appreciate algorithms associated with data structures, have been presented in chapters 1–2.

**Part I** details sequential linear data structures, viz., *arrays, stacks, queues, priority queues* and *dequeues*, and comprises chapters 3–5. **Part II** details linked linear data structures, viz., *linked lists, linked stacks* and *linked queues*, and comprises chapters 6–7. **Part III** elucidates the nonlinear data structures of *trees, binary trees* and *graphs* covering chapters 8–9. **Part IV** highlights the advanced data structures of *binary search trees, AVL trees, B trees, tries, red black trees, splay trees, hash tables* and *files*, which spans chapters 10–14. **Part V** spans chapters 15–17 and discusses searching algorithms of *linear search, transpose sequential search, interpolation search, binary search, Fibonacci search, and other search techniques*, and internal sorting techniques of *bubble sort, insertion sort, selection sort, merge sort, shell sort, quick sort, heap sort* and *radix sort*, and external sorting techniques of *sorting with tapes, sorting with disks, polyphase merge sort* and *cascade merge sort*.

The concepts and techniques behind each data structure and their applications have been explained. Every chapter includes a variety of Illustrative Problems pertaining to the data structure(s) detailed, a summary of the technical content of the chapter and a list of Review Questions, to reinforce the comprehension of the concepts. A set of Programming Assignments to be implemented in the laboratory sessions, have also been listed at the end of the appropriate chapters.

The book could be used both as an introductory or an advanced-level textbook for the undergraduate, graduate and research programmes which offer data structures as a core or an elective course. While the book is primarily meant to serve as a course material for use in the classroom, it could be used as a companion guide during the laboratory sessions to nurture better understanding of the theoretical concepts.

The book could also serve as a course material for various diploma, post-diploma programmes and certificate courses conducted by various IT and related institutes and corporate sectors.

An introductory level course for a duration of one semester, targeting an undergraduate programme or a first-year graduate programme or a diploma programme or a certificate course, could include chapters 1–2, PART I, PART II, chapter 8 of PART III, chapter 13 of PART IV, chapter 15 (Sec. 15.1–15.2, 15.5) and chapter 16 (Sec. 16.1–16.3, 16.5, 16.7) of PART V in its curriculum.

A middle-level course for a duration of one semester, targeting senior graduate-level programmes and research programmes such as MS/Ph D, could include chapters 1–2, PART I, PART II, PART III, chapters 10, 11 and 13 of PART IV, and selective sections of chapters 15–16 of PART V.

An advanced-level course could include parts IV and V besides selections from the rest, based on the prerequisite courses satisfied.

Chapters 8, 10, 11 (Sec. 11.10–11.3), 13, 14 and 17 could be useful for inclusion in a curriculum that serves as a prerequisite for a course on Database Management Systems.

The salient features of the book are as follows:

- All-round emphasis on theory, problems, applications and programming assignments
- Simple and lucid explanation of the theory
- Inclusion of several applications to illustrate the use of data structures
- Several worked-out examples as Illustrative Problems in each chapter
- List of Programming Assignments at the end of each chapter
- Review Questions to strengthen understanding
- Self-contained text for use as a textbook for either an introductory or advanced-level course

The book is accompanied by a web supplement that can be accessed at [www.mhhe.com/pai/dsa](http://www.mhhe.com/pai/dsa). It includes the following online material:

- **Slide Presentation**  
The slides illustrative of the technical content in each chapter of the book could be effectively used by the instructor to supplement classroom teaching.
- **Solution Manual**  
Solutions to selected problems in each chapter are given here.
- **C Programs**  
C implementation of algorithms, demonstrative of selective data structures, discussed in the book have been given here.

I express my sincere thanks to the Management and Principal, PSG College of Technology, Coimbatore, for the encouragement and support provided by them. I also express my appreciation for the editorial and production teams of McGraw-Hill Education (India) Limited, New Delhi, for the excellent production values.

Thanks are also due to all the reviewers who went through the text and provided noteworthy suggestions and advice. Their names are listed below.

<b>Bhupesh Deka</b>	<i>Department of Computer Science Engineering, Sikkim Manipal Institute of Technology, East Sikkim</i>
<b>Basudev Halder</b>	<i>Department of Computer Science Engineering, Institute of Technology &amp; Marine Engineering College, Jingha</i>
<b>Amitava Nag</b>	<i>Department of Computer Science &amp; Engineering, Academy of Technology, Hooghly</i>
<b>S.R. Biradar</b>	<i>Department of Computer Science Engineering, Sikkim Manipal Institute of Technology, East Sikkim</i>
<b>Debasis Chakraborty</b>	<i>Department of Computer Science, Assansol Engineering College, Assansol</i>
<b>N. K. Kamila</b>	<i>Department of Computer Science, C.V Raman College of Engineering, Bhubaneswar</i>
<b>Sanjay Goswami</b>	<i>Department of Computer Applications, Narula Institute of Technology, Kolkata</i>
<b>Sanjoy Kumar Saha</b>	<i>Department of Computer Science and Engineering, Jadavpur University, Kolkata</i>
<b>P. Sampath</b>	<i>Computer Science and Engineering, Bannari Amman Institute of Technology, Sathyamangalam</i>
<b>T.V. Gopal</b>	<i>Department of Computer Science and Engineering, Anna University, Chennai</i>
<b>Suganthi Jeyaraj</b>	<i>Department of Computer Science and Engineering, PSG College of Technology, Coimbatore</i>
<b>Jibi Abraham</b>	<i>Department of Computer Science and Engineering, M.S. Ramaiah Institute of Technology, Bangalore</i>
<b>T. Ramesh</b>	<i>Department of Computer Science, National Institute of Technology, Warangal</i>
<b>Sameer Bhave</b>	<i>Department of Computer Engineering IPS Inst of Engineering and Sciences, Indore</i>
<b>Prashant Lakkadwala</b>	<i>Department of Computer Engineering, Venakateshwar Engineering College, Indore</i>
<b>Manish Manoria</b>	<i>Department of Computer Engineering, Truba College of Science and Technology, Bhopal</i>
<b>Abhay Kothari</b>	<i>Sanghvi Institute of Management and Sciences, Indore</i>

<b>Sachin Tripathi</b>	<i>Department of Computer Science Engineering, Indian School of Mines, Dhanbad</i>
<b>R K Gupta</b>	<i>Department of Computer Engineering, Madhav Institute of Science and Technology, Gwalior</i>
<b>H N Verma</b>	<i>Department of Computer Science, Sanjay Institute of Engineering and Management, Mathura</i>
<b>Dilkeshwar Pandey</b>	<i>Department of Computer Science and Engineering ABES Engineering College</i>
<b>Lalitsen Sharma</b>	<i>Department of Computer Science, University of Jammu, Jammu</i>
<b>Bhavna Jain</b>	<i>Department of Electronics Engineering, Hitkarni College of Engineering, Jabalpur</i>
<b>Akhilesh Kumar Srivastava</b>	<i>CSE Department, Inderprastha Engineering College, Ghaziabad</i>
<b>Shashank Dwivedi</b>	<i>Department of Computer Science United College of Engineering and Research, Allahabad</i>
<b>Sanjay Kumar Pandey</b>	<i>Department of Computer Science United College of Engineering and Research, Allahabad</i>
<b>Rajiv Pandey</b>	<i>Department of Computer Science and Engineering Amity University, Lucknow</i>
<b>Mayank Aggarwal</b>	<i>Department of Computer Science and Engineering, Faculty of Engineering and Technology Gurukul Kangri Vishwavidyalaya, Haridwar</i>
<b>Amit Jain</b>	<i>Computer Science/Information Technology, Radha Govind Engineering College, Meerut</i>
<b>Nilima Fulmare</b>	<i>Hindustan College of Science and Technology, Agra</i>

I would like to place on record my reverence for my mother whose blessings and prayers have been a source of inspiration and great strength. True to the Indian spiritual tradition, I offer my reverent salutations to my spiritual guru Srimat Swami Vireswaranandaji Maharaj, the tenth President of the Ramakrishna Math and Mission. Lastly, the infinite support, encouragement and help provided by my sisters Rekha and Udaya in all my endeavors, are affectionately remembered.

While I hope that the book would be beneficial to novices and sophomores alike, constructive feedback and suggestions for improvement may kindly be mailed to [vijipai@vsnl.com](mailto:vijipai@vsnl.com)

**G A V Pai**



# CONTENTS

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

<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

## Contents

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-way search trees: Definition and Operations</i>	262
11.3	B Trees: Definition and Operations	269
11.4	Tries: Definition and Operations	277
11.5	Applications	281
	<i>Summary</i>	284
	<i>Illustrative Problems</i>	285
	<i>Review Questions</i>	290
	<i>Programming Assignments</i>	292
<b>12.</b>	<b>Red-Black Trees and Splay Trees</b>	<b>293</b>
12.1	Red-Black Trees	293
12.2	Splay Trees	311

12.3 Applications	318
<i>Summary</i>	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
	<b>Index</b>	<b>457</b>

# Visual Walkthrough

xii	Contents
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 Circularly Linked Lists 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>	
6. Linked Lists	84
6.1 Introduction 84	
6.2 Singly Linked Lists 87	
6.3 Circularly Linked Lists 93	
6.4 Doubly Linked Lists 98	
6.5 Multiply Linked Lists 103	
6.6 Applications 105	
Summary 112	
Illustrative Problems 113	
Review Questions 118	
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>	
8. Trees and Binary Trees	151
8.1 Introduction 151	
8.2 Trees: Definition and Basic Terminologies 151	
8.3 Representation of Trees 153	

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

Each chapter lists the topics covered.

CHAPTER
6

**LINKED LISTS**

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

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

In this chapter we detail linear data structures having a linked representation. We first list the elements of the sequential data 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.

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 of size 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 synonyms. When a bucket is full and a synonym may be divided into slots to accommodate synonyms. When a bucket is full and a synonym is unable to find space in the bucket then an overflow is said to have occurred.
- The characteristics of a hash function are that it must be easy to compute and at the same time minimize collisions. Folding, truncation and modular arithmetic are some of the commonly used hash functions.
- A hash table can be implemented using a sequential data structure such as arrays. In such a case the method of handling overflows where the closest slot that is vacant is utilized to accommodate the synonym key is called linear open addressing or linear probing. However, in course of time, linear probing can lead to the problem of clustering thereby deteriorating the performance of the hash table to a mere sequential search!
- The other alternative methods of handling overflows are rehashing, quadratic probing and random probing.

Chapter-end summary for use as quick reference.

386 Data Structure and Algorithms

### Illustrative Problems

**Problem 15.1** For the list CHANNELS=( AXN, ZEE, ETC, CNN, DDDN, HBO, GOOD, FEAS, MNDS, SSON, CCAF, NNGE, BBC, FPRO) trace through sequential search for the list.

**External Sorting**

### Summary

- External sorting deals with sorting of files or lists that are too huge 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.
- Unit 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  $2^k$  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 term of cylinders) needed to store the employee file in the disk?

**Solution:**  
(a) The size of the employee record = 118 bytes  
Number of employee records that can be held in a sector =  $512/118 = 4$  records  
Number of sectors needed to hold the whole employee file =  $100000/4 = 25,000$  sectors

Extensive Illustrative Problems throughout.

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

### Review Questions

1. A minimal superkey is in fact a \_\_\_\_\_
  - (a) secondary key
  - (b) primary key
  - (c) non key
  - (d) none of these
2. State whether true or false:
  - (i) A primary key field with variable length yields 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
3. An index consisting of variable length entries where each index entry would be of the form  $(K, B_1\uparrow, B_2\uparrow, B_3\uparrow, \dots, B_l\uparrow)$  where  $B_i\uparrow$ '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
- DEQUEUE (QUEUE, ITEM, FRONT, REAR)
- Check if queue QUEUE is empty
- Check if queue QUEUE is full
- Insert ITEM into 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

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

## 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 (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  $[1, 5, 15, 15, 26, 34, 76]$ .

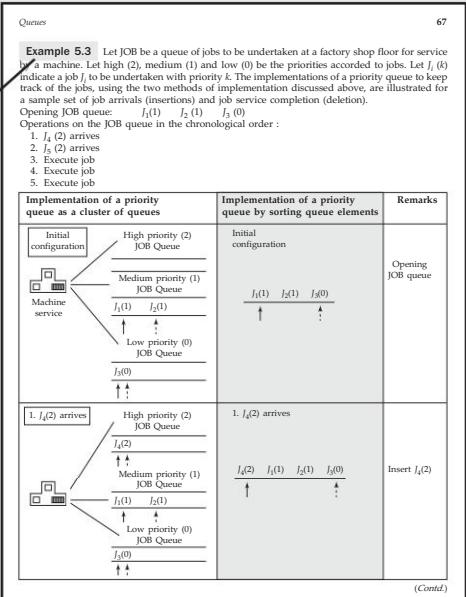
```
Algorithm 16.8: Procedure for Quick Sort
procedure QUICK_SORT(L, first, last)
    /* L[first:last] is the unordered list of elements to be
       quick sorted. The call to the procedure to sort the
       list L[1:n] would be QUICK_SORT(L, 1, n)*/
if (first < last) then
    { PARTITION(L, first, 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.

Extensive examples are given to illustrate theoretical concepts.

Pseudo-code algorithms are given for better comprehension



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

## CHAPTER

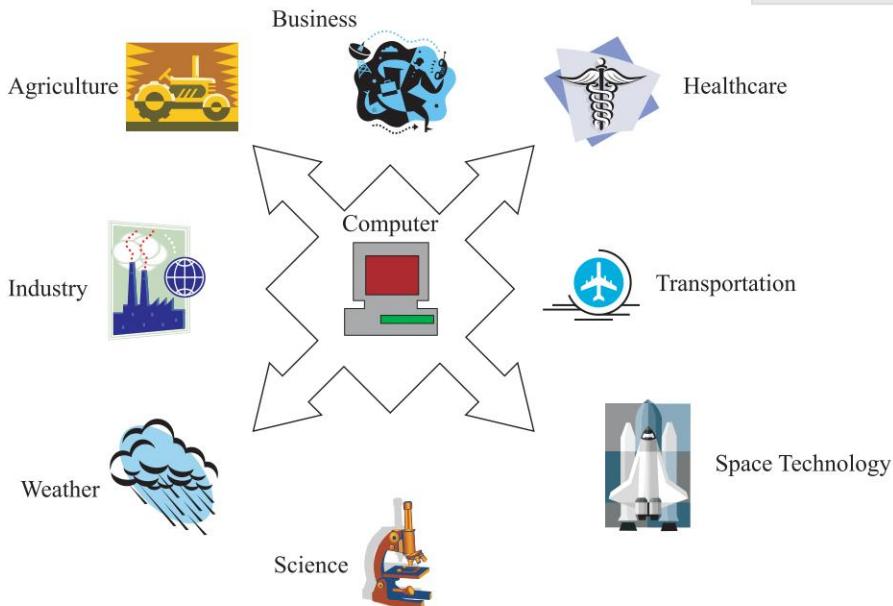


## INTRODUCTION

## 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:

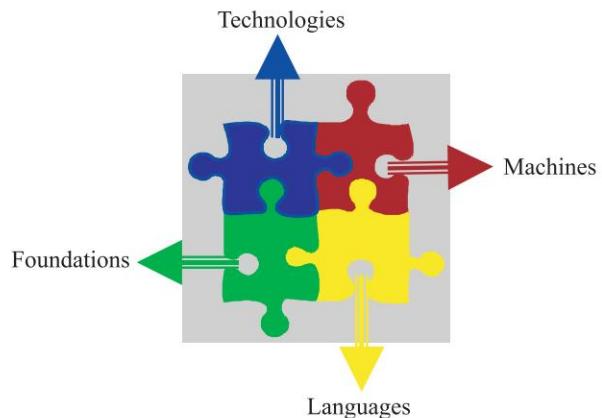
- *Machines* What machines are appropriate or available for the solution of a problem? What is the machine configuration – its processing power, memory capacity etc., that would be required for the efficient execution of the solution?
- *Languages* What is the language or software with which the solution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

$$\begin{array}{rcl}
 987 + & & 987 + \\
 & 76 & \Rightarrow & 76 + \\
 & & & \text{Carry } 1 \\
 \hline
 & \text{(Carry 1) } 3 & \hline
 & & \hline
 & & \text{Carry } 1 \\
 & & \hline
 & & 1063
 \end{array}$$

## Definition, Structure and Properties of Algorithms

1.2

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

### Structure and properties

An algorithm has the following structure:

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

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

An algorithm is endowed with the following properties:

<b>Finiteness</b>	an algorithm must terminate after a finite number of steps.
<b>Definiteness</b>	the steps of the algorithm must be precisely defined or unambiguously specified.
<b>Generality</b>	an algorithm must be generic enough to solve all problems of a particular class.
<b>Effectiveness</b>	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!
<b>Input-Output</b>	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

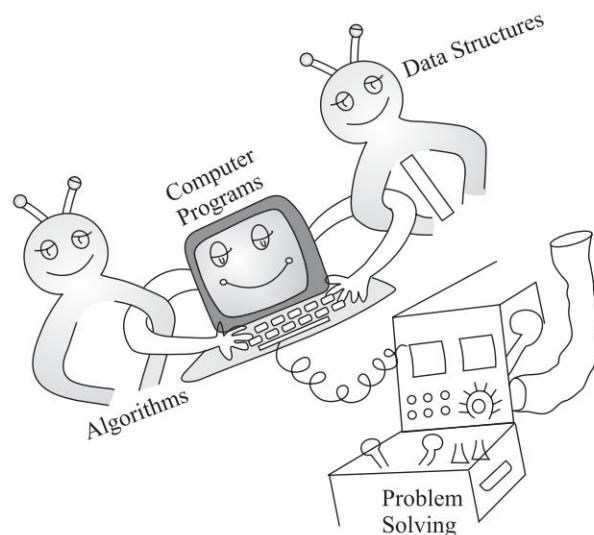
As was detailed in the previous section, the design of an *efficient* algorithm for the solution of the problem calls for the *inclusion of appropriate data structures*. A clear, unambiguous set of instructions following the properties of the algorithm alone does not contribute to the efficiency of the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

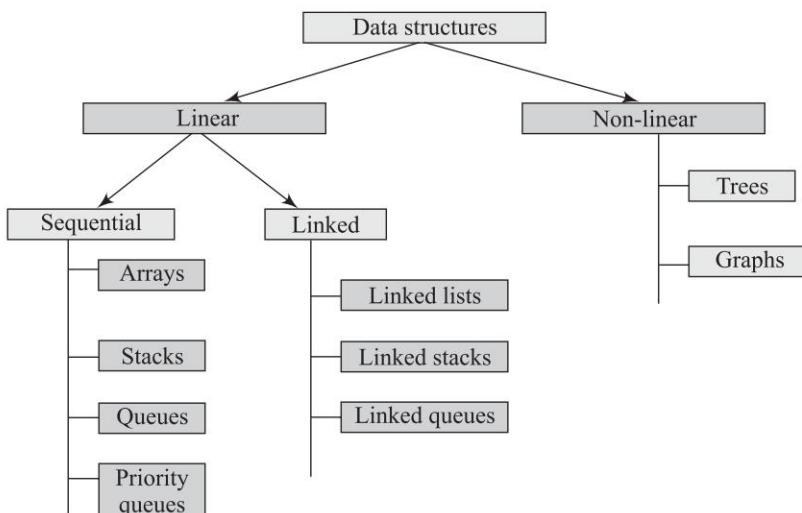
An ADT promotes ***data abstraction*** and focuses on *what* a data structure does rather than *how* it does. It is easier to comprehend a data structure by means of its ADT since it helps a designer to plan on the implementation of the data objects and its supportive operations in any programming language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

## CHAPTER



# ANALYSIS OF ALGORITHMS

# 2

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

## Efficiency of Algorithms

## 2.1

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

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

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

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

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

2.1 *Efficiency of Algorithms*

2.2 *Apriori Analysis*

2.3 *Asymptotic Notations*

2.4 *Time Complexity of an Algorithm using O Notation*

2.5 *Polynomial vs Exponential Algorithms*

2.6 *Average, Best and Worst Case Complexities*

2.7 *Analyzing Recursive Programs*

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

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

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

## Apriori Analysis

## 2.2

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

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

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

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

Program segment A

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

Program segment B

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

Program segment C

```
...
for j = 1 to n do
for x = 1 to n do
x = x + 2;
end
end
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
...	
<b>Total frequency count</b>	1

**Table 2.2 Total frequency count of program segment B**

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

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

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

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**

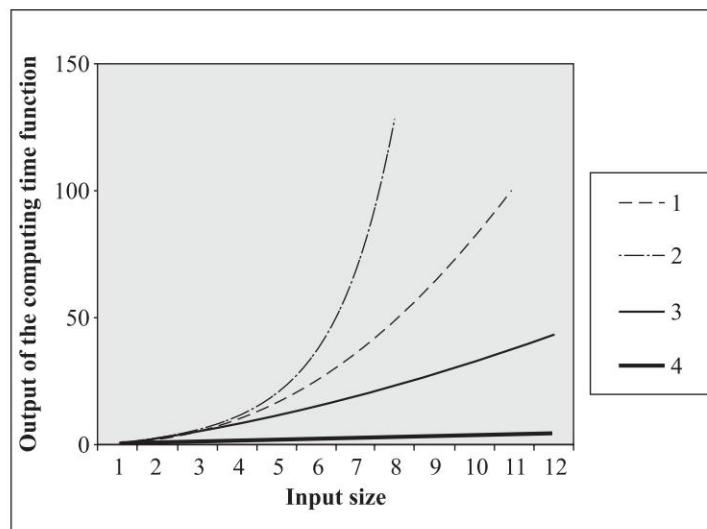
If  $n$  is the size of the input instance, then the number of operations for polynomial algorithms are of the form  $P(n)$  where  $P$  is a polynomial. In terms of  $O$  notation, polynomial algorithms have time complexities of the form  $O(n^k)$ , where  $k$  is a constant.

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
<b>Time complexity function</b>			
$n^2$	$10^{-4}$ 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



1 :  $n^2$       2:  $2^n$       3:  $n \log_2 n$       4:  $\log_2 n$

**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.

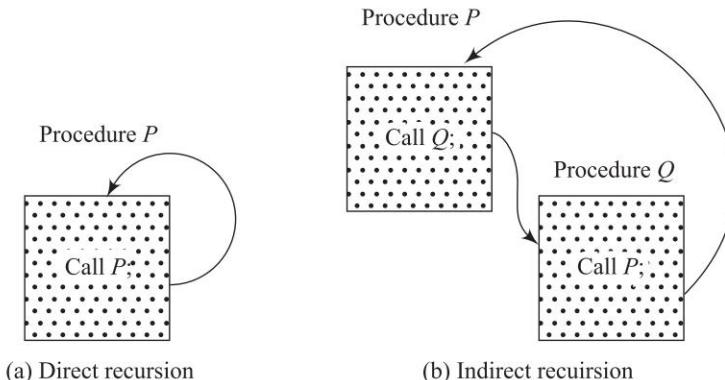
#### Recursive procedures

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

- 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.
- 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:

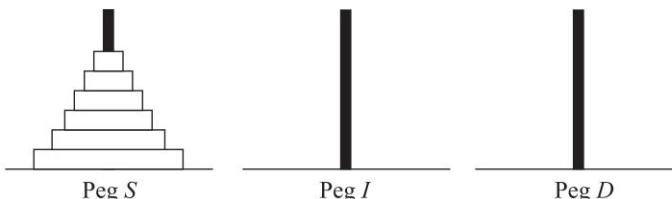
$$S(n) = \begin{cases} 0, & \text{if } n = 1 \text{ (base criterion)} \\ S(n/2) + 1, & \text{if } n > 1 \end{cases}$$

**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  $D$

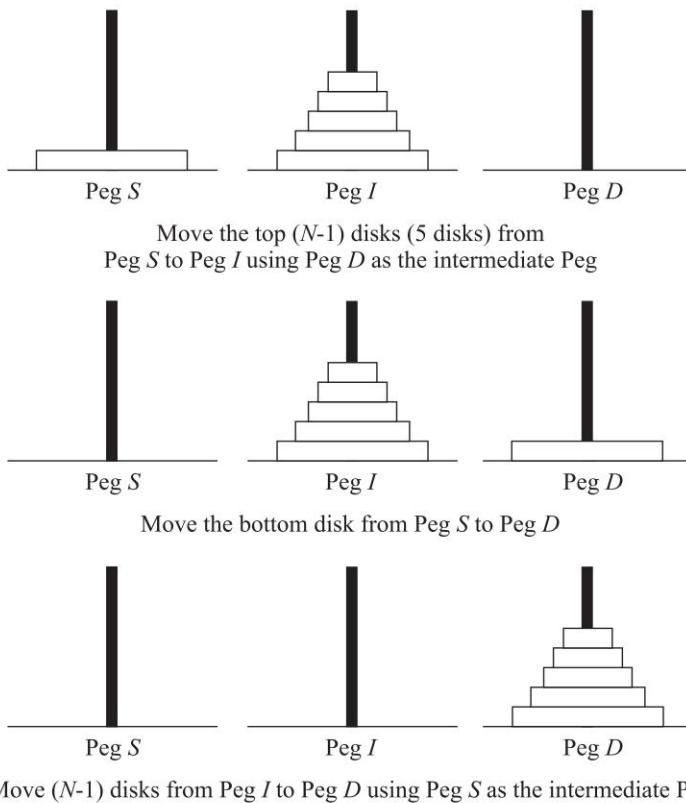


**Fig. 2.3** Tower of Hanoi puzzle (initial configuration)

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.

$$T(n) = c + T(n - 1) \quad \dots(\text{step 1})$$

$$= c + (c + T(n - 2)) \quad \dots(\text{step 2})$$

$$= 2c + T(n - 2) \quad \dots(\text{step 2})$$

$$= 2c + (c + T(n - 3)) \quad \dots(\text{step 3})$$

$$= 3c + T(n - 3) \quad \dots(\text{step 3})$$

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

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

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

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

$$\therefore x = \sqrt{10} \times n \approx 3.16n$$

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
<b>Total frequency count</b>	<b>4</b>	<b>5</b>	<b>6</b>	<b><math>5n + 3</math></b>

**Problem 2.5** Obtain the time complexity of the following program:

```

procedure whirlpool (m)
begin

```

```

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

```

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

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

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

Solving for the recurrence relation yields the following steps:

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

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)  $f(n) = 30n^22^n + 6n2^n + 8n^2 = O(2^n)$
- (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  $n = 2k$ :

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

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

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

In the  $i^{\text{th}}$  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  $i = (k - 1)$ ,

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

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



## Review Questions

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

```

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

```

12. Solve the recurrence relation:

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



# ARRAYS

# 3

## Introduction

## 3.1

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

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

$A[1 : 5]$ <table border="1" style="border-collapse: collapse; width: 100px; margin-left: auto; margin-right: auto;"> <tr><td style="padding: 5px;">6</td><td style="padding: 5px;">-4</td><td style="padding: 5px;">3</td><td style="padding: 5px;">2</td><td style="padding: 5px;">11</td></tr> </table>	6	-4	3	2	11	$B[1 : 3, 1 : 2]$ $\begin{matrix} 1 & 2 \\ -6 & 4 \\ 3 & 2 \\ 7 & -5 \end{matrix}$
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 ( $A, 3, 17$ )  RETRIEVE ( $A, 2$ )	$A \begin{array}{ c c c c c } \hline 6 & -4 & 17 & 2 & 11 \\ \hline [1] & [2] & [3] & [4] & [5] \\ \hline \end{array}$ -4

Fig. 3.2 Array operations: Store and Retrieve

## Number of Elements in an Array

## 3.3

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

### One-dimensional array

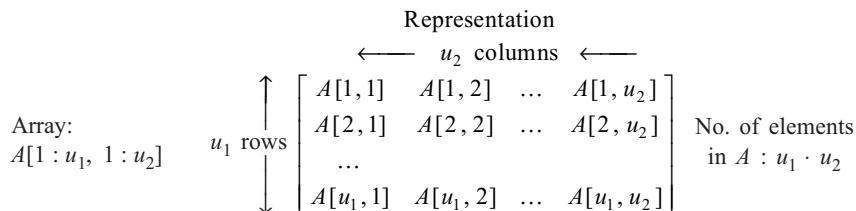
Let  $A[1:u]$  be a one-dimensional array. The size of the array, as is evident is  $u$  and the elements are  $A[1], A[2], \dots, A[u-1], A[u]$ . In the case of the array  $A[l : u]$  where  $l$  is the lower bound and  $u$  is the upper bound of the index range, the number of elements is given by 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

**Example 3.1** The number of elements in

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

### Two-dimensional array

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

**Fig. 3.3** Size of a two-dimensional array**Example 3.2** The number of elements in

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

## Multi-dimensional array

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

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

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

**Fig. 3.4** Size of a three-dimensional array**Example 3.3** The number of elements in

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

## Representation of Arrays in Memory

## 3.4

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

## Arrays

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

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

## One-dimensional array

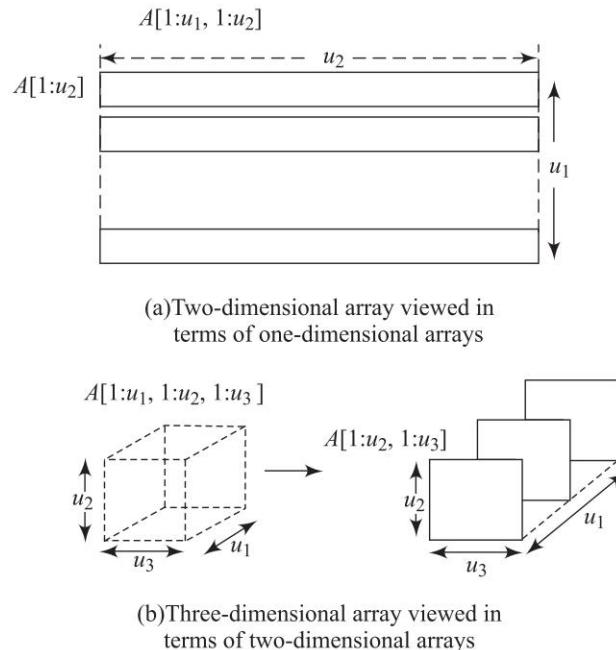
Consider the array  $A(1 : u_1)$  and let  $\alpha$  be the address of the starting memory location referred to as the *base address* of the array. Here as is evident,  $A[1]$  occupies the memory location whose address is  $\alpha$ ,  $A(2)$  occupies  $\alpha + 1$  and so on. In general, the address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

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

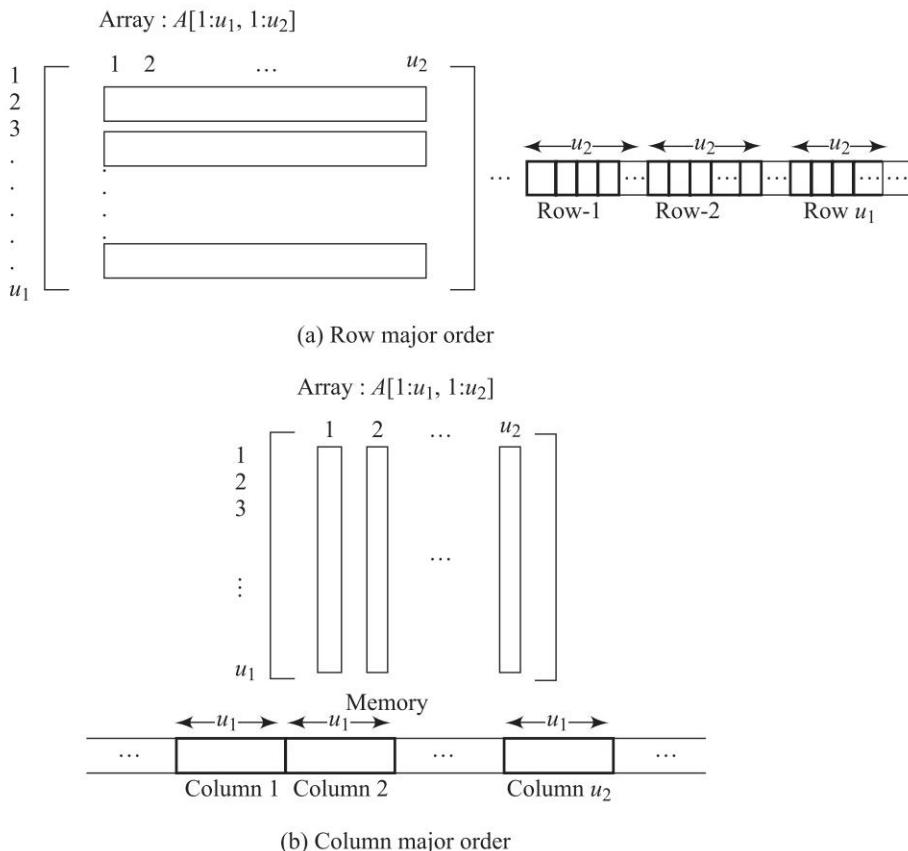
Array	Element	Address
(i) $A[1:17]$	$A[7]$	$\alpha + (7 - 1) = 100 + 6 = 106$
(ii) $A[-2:23]$	$A[16]$	$\alpha + (16 - (-2)) = 100 + 18 = 118$

## Two-dimensional array

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



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



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

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

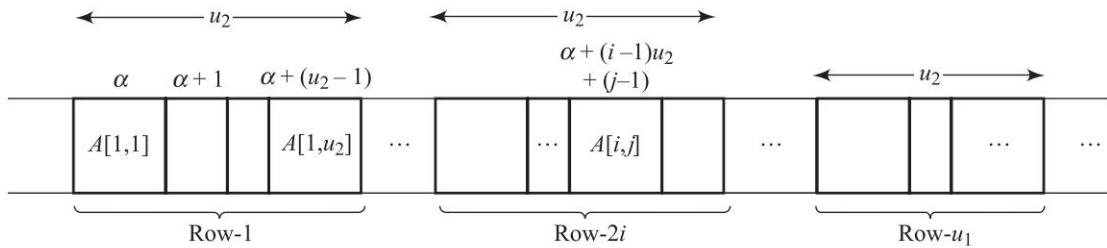
**Fig. 3.7** Representation of one-dimensional arrays in memory

left behind before one knocks at the first room in the  $i^{\text{th}}$  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:

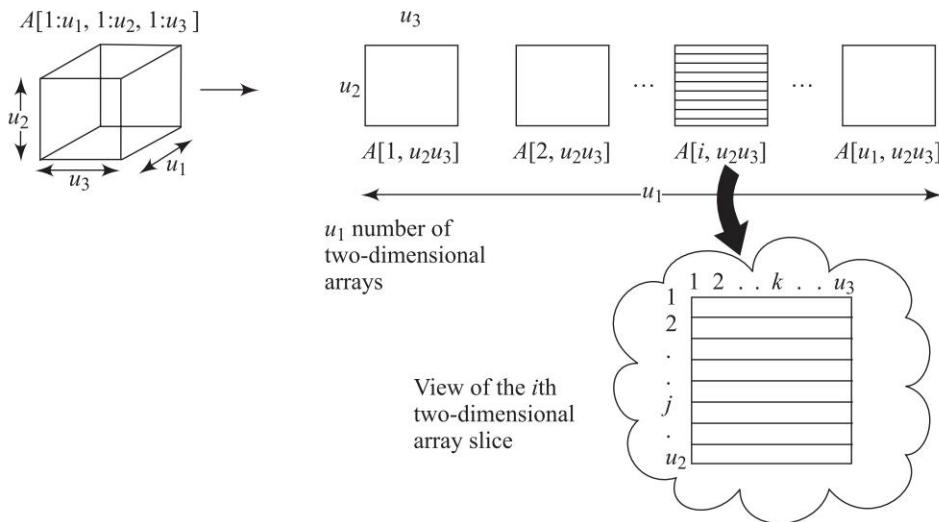
Array	Element	Address
$A[1 : 10, 1 : 5]$	$A[8, 3]$	$220 + (8 - 1)5 + (3 - 1) = 257$
$A[-2 : 4, -6 : 10]$	$A[3, -5]$	$220 + (3 - (-2))(10 - (-6) + 1) + (-5 - (-6)) = 306$

### 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  $u_2 u_3$ . Reverting to the analogy of building-floor-rooms, the three dimensional array  $A[i, j, k]$  could be viewed as a colony of  $i$  buildings each having  $j$  floors with each floor accommodating  $k$  rooms.

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

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



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

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

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

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

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

## $N$ -dimensional array

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

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

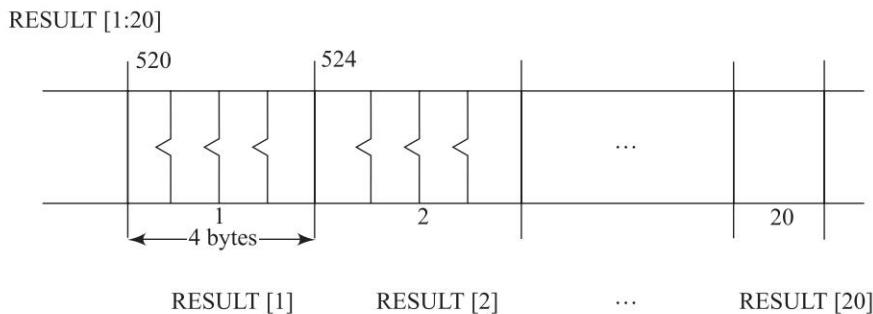
- Two concepts viz., sparse matrices and ordered lists, of use to computer science have been briefly described as applications of arrays.

## 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:*

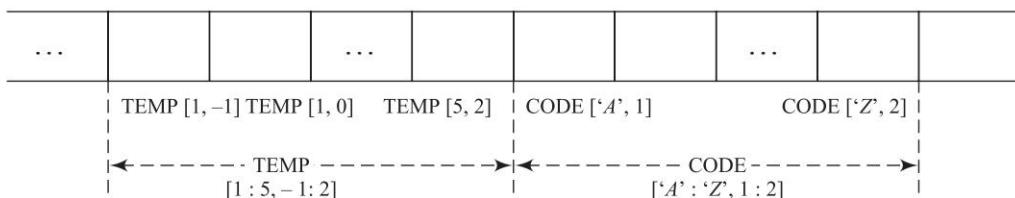
- (i) The number of elements in  $B$  is  $16 \times 6 = 96$
- (ii) The space occupied by  $B$  is  $96 \times 4 = 384$  bytes
- (iii) The address of  $B[7, 2]$  is given by

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

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

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

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



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

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

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

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

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

Therefore the base address of CODE is given by 520.

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

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

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

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

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

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

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



## Review Questions

1. Which among the following pairs of operations is supported by an array ADT?
  - (i) store and retrieve
  - (ii) insert and delete
  - (iii) copy and delete
  - (iv) append and copy

(a) (i)	(b) (ii)	(c) (iii)	(d) (iv)
---------	----------	-----------	----------
2. The number of elements in an array  $\text{ARRAY}[l_1 : u_1, l_2 : u_2]$  is given by
 

(a) $(u_1 - l_1 - 1)(u_2 - l_2 - 1)$	(b) $(u_1 \cdot u_2)$
(c) $(u_1 - l_1)(u_2 - l_2)$	(d) $(u_1 - l_1 + 1)(u_2 - l_2 + 1)$
3. A multi-dimensional array  $\text{OPEN}[0 : 2, 10 : 20, 3 : 4, -10 : 2]$  contains \_\_\_\_\_ elements.
 

(a) 240	(b) 858	(c) 390	(d) 160
---------	---------	---------	---------
4. For the array  $A[1 : u_1, 1 : u_2]$  where  $\alpha$  is the base address,  $A[i, 1]$  has its address given by
 

(a) $(i - 1)u_2$	(b) $\alpha + (i - 1)u_2$	(c) $\alpha + i \cdot u_2$	(d) $\alpha + (i - 1) \cdot u_1$
------------------	---------------------------	----------------------------	----------------------------------
5. For the array,  $A[1 : u_1, 1 : u_2, 1 : u_3]$  where  $\alpha$  is the base address, the address of  $A[i, j, 1]$  is given by
 

(a) $\alpha + (i - 1)u_2u_3 + (j - 1)u_3$	(b) $\alpha + i \cdot u_2u_3 + j \cdot u_3$
(c) $\alpha + (i - 1)u_1u_2 + (j - 1)u_2$	(d) $\alpha + i \cdot u_1u_2 + j \cdot u_2$
6. Distinguish between the row major and column major ordering of an array.
7. For an  $n$ -dimensional array  $A[1 : u_1, 1 : u_2, \dots, 1 : u_N]$  obtain the address of the element  $A[i_1, i_2, i_3, \dots, i_N]$  given  $\beta$  to be the home address.
8. For the following sparse matrix obtain an array representation.

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



## Programming Assignments

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

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

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

Write a program to

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



# STACKS

# 4

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.

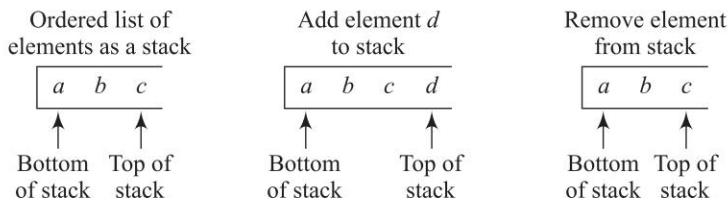
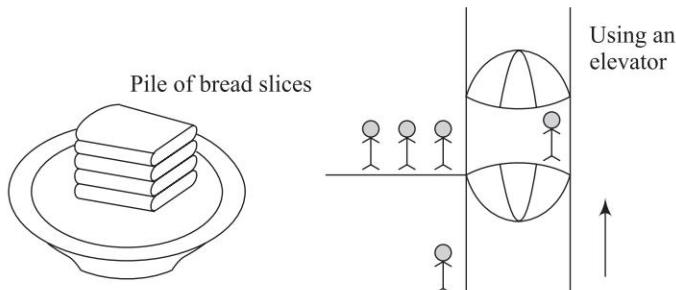


Fig. 4.1 Stack and its functionality

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

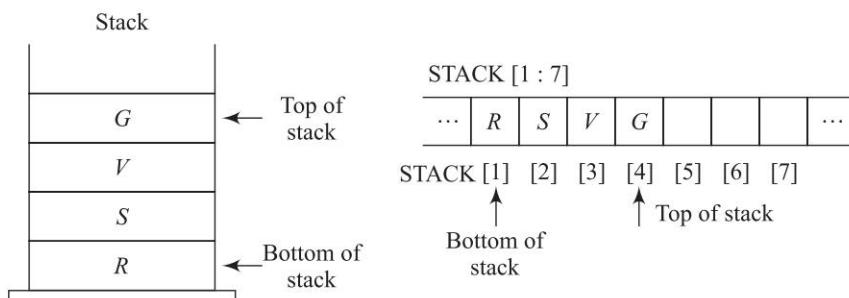
- (i) Insertion or addition of elements known as *Push*
- (ii) Deletion or removal of elements known as *Pop*

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.

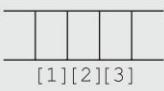
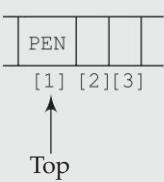
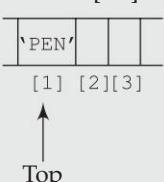
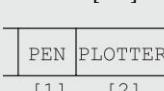
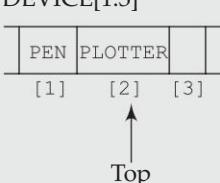
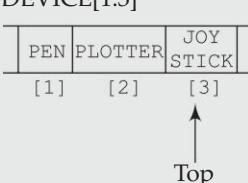
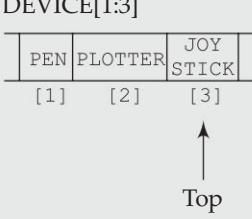
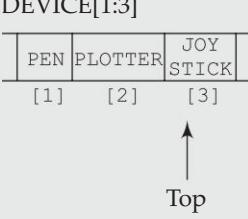
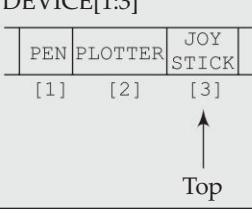
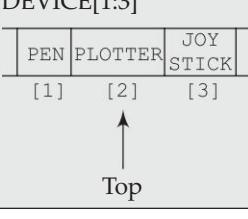
### Algorithm 4.2: Implementation of pop operation on a stack

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

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

## Applications

## 4.3

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

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

### Recursive programming

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

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

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

**Fig. 4.4** Recursive procedure to compute  $n!$

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

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

$n$	5
$x$	
$y$	

(a) Invocation of FACTORIAL (5)

$n$	5
$x$	5
$y$	↗

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

$n$	5	4
$x$	4	3
$y$	↗	↗

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

$n$	5	4	3	2	1
$x$	4	3	2	1	0
$y$	↗	↗	↗	↗	↗

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

$n$	5	4	3	2	1	0
$x$	4	3	2	1	0	
$y$	↗	↗	↗	↗	↗	

(e) Invocation of FACTORIAL (0)

$n$	5	4	3	2	1
$x$	4	3	2	1	0
$y$	↗	↗	↗	↗	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

$n$	5	4	3	2
$x$	4	3	2	1
$y$	↗	↗	↗	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

$n$	5
$x$	4
$y$	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  $y = \text{FACTORIZATION}(4) = 24$  and proceeds to obtain  $\text{FACTORIZATION} = n * y = 5 * 24 = 120$  which is the desired result.

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

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

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

## Evaluation of expressions

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

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

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

Expression	Sub expression chosen based on rules of hierarchy, precedence and associativity	Postfix expression
(i) $a + b * c - d$ \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) $(\textcircled{1} - \textcircled{2}) \uparrow d$ \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 $C \uparrow D = 2 \uparrow 3 = 8$ 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	<u>          </u>	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
```

$X$  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;`
2. `CLEAR(STACK);`
3. **repeat**
4. **if** `TERM <= 12` **then**
5.     `PUSH(STACK, TERM);`
6. **else**
7.     `POP(STACK, TERM);`
8.     `PRINT(TERM);`
9.     `TERM = 3 * TERM + 2;`
10. **until** `EMPTY_STACK(STACK)`
- 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  $\text{mod}(23, 7)$ .

```

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.

23	7	16	↑
$a$	$b$	$x_1$	$y_1$

(a) call mod(23, 7)

16	7	9	↑
23	7	16	↑

(b) call mod(16, 7)

9	7	2	↑
16	7	9	↑
23	7	16	↑

(c) call mod(9, 7)

2	7	2	↑
9	7	2	↑
16	7	9	↑
23	7	16	↑

(d) call mod(2, 7)

9	7	2	2
16	7	9	↑
23	7	16	↑

(e) After termination of mod(2, 7)

23	7	16	2
$a$	$b$	$x_1$	$y_1$

(f) After termination of mod(9, 7) and mod(16, 7)


output: 2

(g) After termination of mod(23, 7)

**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
$\underline{(-(A+B+C) \uparrow D)^{*}(G+H)}$ ①	( $A + B + C$ )  [Note: $(A + B + C)$ is equivalent to the two subexpressions  $\begin{array}{c} (A+B+C) \\ \xrightarrow{\text{①'}} \\ (\textcircled{1'} + C) \end{array}$ ] ②	①: $AB + C+$  ②: $\text{++ } ABC$	

(Contd.)

(Contd.)

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

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

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

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

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

(Contd.)

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



## Review Questions

1. Which among the following properties does not hold good in a stack?
  - (i) A stack supports the principle of Last In First Out
  - (ii) A push operation decrements the top pointer
  - (iii) A pop operation deletes an item from the stack
  - (iv) A linear stack has limited capacity

(a) (i)                         (b) (ii)   (c) (iii)   (d) (iv)
2. A linear stack  $S$  is implemented using an array as shown below. The TOP pointer which points to the top most element of the stack is set as shown.

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

Bottom of  $\uparrow$  stack

$\uparrow$  TOP

- Execution of the operation  $PUSH(S, 'W')$  would result in
- (a)  $TOP = 4$    (b)  $TOP = 5$
  - (c) Stack full condition   (d)  $TOP = 3$
  3. For the linear stack shown in Review Question 2, execution of the operations  $POP(S)$ ,  $POP(S)$ ,  $PUSH(S, 'U')$ ,  $POP(S)$  in a sequential fashion would leave the element \_\_\_\_\_ on top of the stack with the TOP pointer set to the value \_\_\_\_\_.
    - (a)  $Y, 2$    (b)  $U, 3$    (c)  $U, 1$    (d)  $U, 4$
  4. The equivalent post fix expression for the infix expression  $a + b + c$  is
    - (a)  $abc++$    (b)  $ab+c+$    (c)  $ab++c$    (d)  $a++bc$
  5. The equivalent post fix expression for the infix expression  $a \uparrow b \uparrow c \uparrow d$  is
    - (a)  $ab \uparrow cd \uparrow \uparrow$    (b)  $abc \uparrow \uparrow \uparrow d$    (c)  $ab \uparrow c \uparrow d \uparrow$    (d)  $abcd \uparrow \uparrow \uparrow$
  6. How are insert operations carried out in a stack?
  7. What are the demerits of a linear stack?
  8. If a stack  $S[1 : n]$  were to be implemented with the bottom of the stack at  $S[n]$ , write a procedure to undertake push operation on  $S$ .
  9. For the stack  $S[1 : n]$  introduced in Review Question 8 of Chapter 4, write a procedure to undertake the pop operation on  $S$ .
  10. For the following logical expression  

$$(a \text{ and } b \text{ and } c) \text{ or } d \text{ or } e \text{ or } (\text{not } h)$$
    - (i) obtain the equivalent postfix expression
    - (ii) evaluate the post fix expression for  $a = \text{true}$ ,  $b = \text{false}$ ,  $c = \text{true}$ ,  $d = \text{true}$ ,  $e = \text{true}$ ,  $h = \text{false}$ .



## 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  $\underline{(-(A*B)/D)} \uparrow C + E - F * H * I$  for  $A = 1, B = 2, D = 3, C = 14, E = 110, F = 220, H = 16.78, I = 364.621$ .



# QUEUES

# 5

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

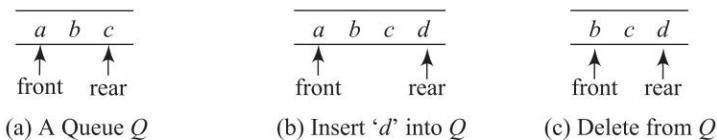
## Introduction

## 5.1

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

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

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



**Fig. 5.1** A queue and its functionality

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

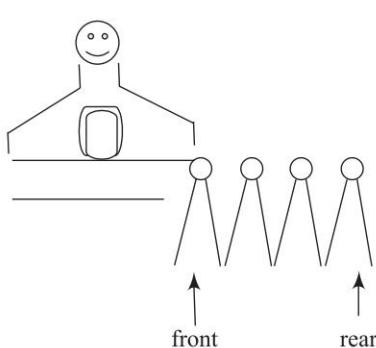
## Operations on Queues

5.2

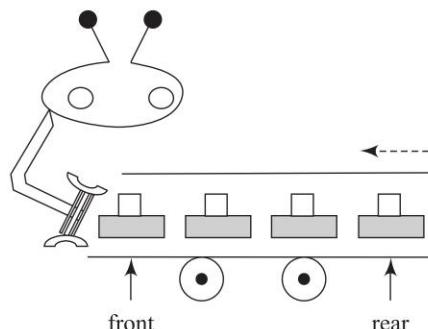
The queue data structure supports two operations, viz.,

- (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



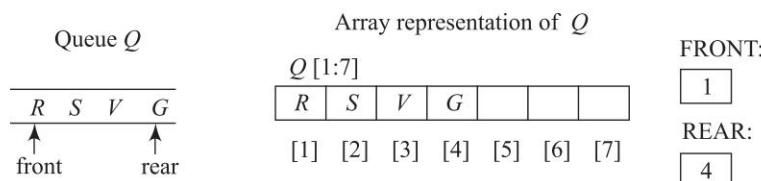
(b) Queue of components in an assembly line

**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.

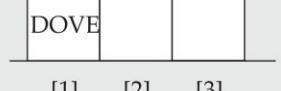
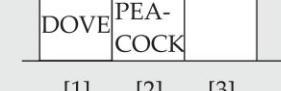
Soon after the queue  $Q$  has been initialized,  $\text{FRONT} = \text{REAR} = 0$ . Hence the condition ( $\text{FRONT} = \text{REAR}$ ) ensures that the queue is empty. Again after a sequence of operations when  $Q$  has become partially or completely full and delete operations are repeatedly invoked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

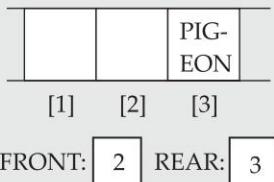
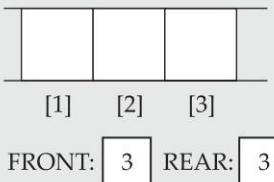
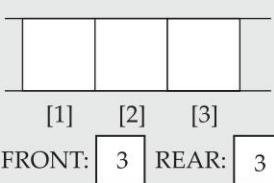
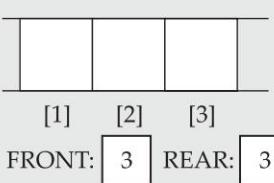
Operation	Queue before operation	Algorithm	Queue after operation	Remarks
1. Insert 'DOVE' into $\text{BIRDS}[1:3]$	BIRDS [1:3]  [1] [2] [3] FRONT: <span style="border: 1px solid black; padding: 2px;">0</span> REAR: <span style="border: 1px solid black; padding: 2px;">0</span>	INSERTQ (BIRDS, 3, 'DOVE', 0)	BIRDS [1:3]  [1] [2] [3] FRONT: <span style="border: 1px solid black; padding: 2px;">0</span> REAR: <span style="border: 1px solid black; padding: 2px;">1</span>	Insert 'DOVE' successful
2. Insert 'PEACOCK' into $\text{BIRDS}[1:3]$	BIRDS [1:3]  [1] [2] [3] FRONT: <span style="border: 1px solid black; padding: 2px;">0</span> REAR: <span style="border: 1px solid black; padding: 2px;">1</span>	INSERTQ (BIRDS, 3, 'PEACOCK', 1)	BIRDS [1:3]  [1] [2] [3] FRONT: <span style="border: 1px solid black; padding: 2px;">0</span> REAR: <span style="border: 1px solid black; padding: 2px;">2</span>	Insert 'PEACOCK' successful

(Contd.)

(Contd.)

3. Insert 'PIGEON' in to BIRDS [1:3]	<p>BIRDS [1:3]</p> <table border="1" data-bbox="287 282 582 447"> <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: <span style="border: 1px solid black; padding: 2px;">0</span> REAR: <span style="border: 1px solid black; padding: 2px;">2</span></p>	DOVE	PEA-COCK		[1]	[2]	[3]	INSERTQ(BIRDS, 3, 'PIGEON', 2)	<p>BIRDS [1:3]</p> <table border="1" data-bbox="792 282 1087 447"> <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: <span style="border: 1px solid black; padding: 2px;">0</span> REAR: <span style="border: 1px solid black; padding: 2px;">3</span></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="287 538 582 702"> <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: <span style="border: 1px solid black; padding: 2px;">0</span> REAR: <span style="border: 1px solid black; padding: 2px;">3</span></p>	DOVE	PEA-COCK	PIG-EON	[1]	[2]	[3]	INSERTQ(BIRDS, 3, 'SWAN', 3)	<p>BIRDS [1:3]</p> <table border="1" data-bbox="792 538 1087 702"> <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: <span style="border: 1px solid black; padding: 2px;">0</span> REAR: <span style="border: 1px solid black; padding: 2px;">3</span></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="287 803 582 967"> <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: <span style="border: 1px solid black; padding: 2px;">0</span> REAR: <span style="border: 1px solid black; padding: 2px;">3</span></p>	DOVE	PEA-COCK	PIG-EON	[1]	[2]	[3]	DELETEQ(BIRDS, 0, 3, ITEM)	<p>BIRDS [1:3]</p> <table border="1" data-bbox="792 803 1087 967"> <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: <span style="border: 1px solid black; padding: 2px;">1</span> REAR: <span style="border: 1px solid black; padding: 2px;">3</span></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="287 1076 582 1241"> <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: <span style="border: 1px solid black; padding: 2px;">1</span> REAR: <span style="border: 1px solid black; padding: 2px;">3</span></p>		PEA-COCK	PIG-EON	[1]	[2]	[3]	DELETEQ(BIRDS, 0, 3, ITEM)	<p>BIRDS [1:3]</p> <table border="1" data-bbox="792 1076 1087 1241"> <tr><td></td><td></td><td>PIG-EON</td></tr> <tr><td>[1]</td><td>[2]</td><td>[3]</td></tr> </table> <p>FRONT: <span style="border: 1px solid black; padding: 2px;">2</span> REAR: <span style="border: 1px solid black; padding: 2px;">3</span></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="287 1332 582 1496"> <tr><td></td><td></td><td>PIG-EON</td></tr> <tr><td>[1]</td><td>[2]</td><td>[3]</td></tr> </table> <p>FRONT: <span style="border: 1px solid black; padding: 2px;">2</span> REAR: <span style="border: 1px solid black; padding: 2px;">3</span></p>			PIG-EON	[1]	[2]	[3]	INSERTQ(BIRDS, 3, 'SWAN', 3)	<p>BIRDS [1:3]</p> <table border="1" data-bbox="792 1332 1087 1496"> <tr><td></td><td></td><td>PIG-EON</td></tr> <tr><td>[1]</td><td>[2]</td><td>[3]</td></tr> </table> <p>FRONT: <span style="border: 1px solid black; padding: 2px;">2</span> REAR: <span style="border: 1px solid black; padding: 2px;">3</span></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

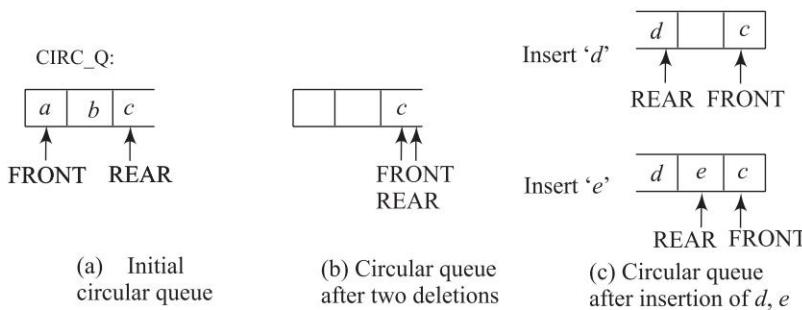
## 5.3

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)).

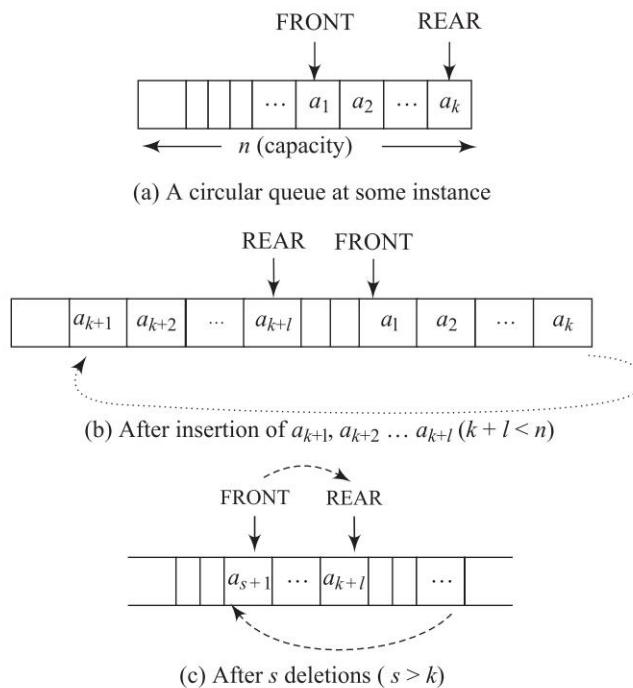
Figure 5.5 emphasizes this circular movement of `FRONT` and `REAR` variables over a general circular queue during a sequence of insertions/deletions.

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.

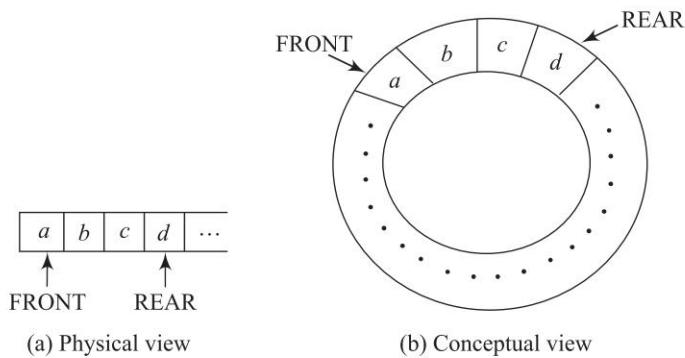
### Implementation of insertion and deletion operations in a circular queue

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]`.

## Queues



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



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

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

```

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

**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]  FRONT: <input type="text" value="0"/> REAR: <input type="text" value="0"/>	INSERT_CIRCQ(COLOURS, 0, 0, 4, 'ORANGE')	COLOURS [0:3]  FRONT: <input type="text" value="0"/> REAR: <input type="text" value="1"/>	Insert 'ORANGE' successful
2. Insert 'BLUE' into COLOURS [0:3]	COLOURS [0:3]  FRONT: <input type="text" value="0"/> REAR: <input type="text" value="1"/>	INSERT_CIRCQ(COLOURS, 0, 1, 4, 'BLUE')	COLOURS [0:3]  FRONT: <input type="text" value="0"/> REAR: <input type="text" value="2"/>	Insert 'BLUE' successful
3. Insert 'WHITE' into COLOURS [0:3]	COLOURS [0:3]  FRONT: <input type="text" value="0"/> REAR: <input type="text" value="2"/>	INSERT_CIRCQ(COLOURS, 0, 2, 4, 'WHITE')	COLOURS [0:3]  FRONT: <input type="text" value="0"/> REAR: <input type="text" value="3"/>	Insert 'WHITE' successful

(Contd.)

## Queues

(Contd.)

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

## Other Types of Queues

## 5.4

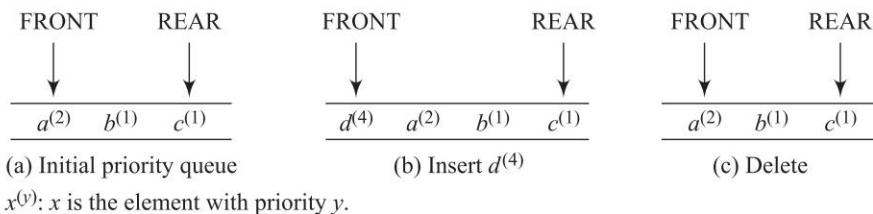
### Priority queues

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

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

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

A common method of implementation of a priority queue is to open as many queues as there are priority factors. A low priority queue will be operated for deletion only when all its high priority predecessors are empty. In other words, deletion of an element in a priority queue  $q_1$  with priority  $p_i$  is possible only when those queues  $q_j$  with priorities  $p_j$  ( $p_j > p_i$ ) are empty. However, with regard to insertions, an element  $e_k$  with priority  $p_1$  joins the respective queue obeying the scheme of FIFO with regard to the queue  $q_l$  alone.



**Fig. 5.7** A priority queue

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

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

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

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

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

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

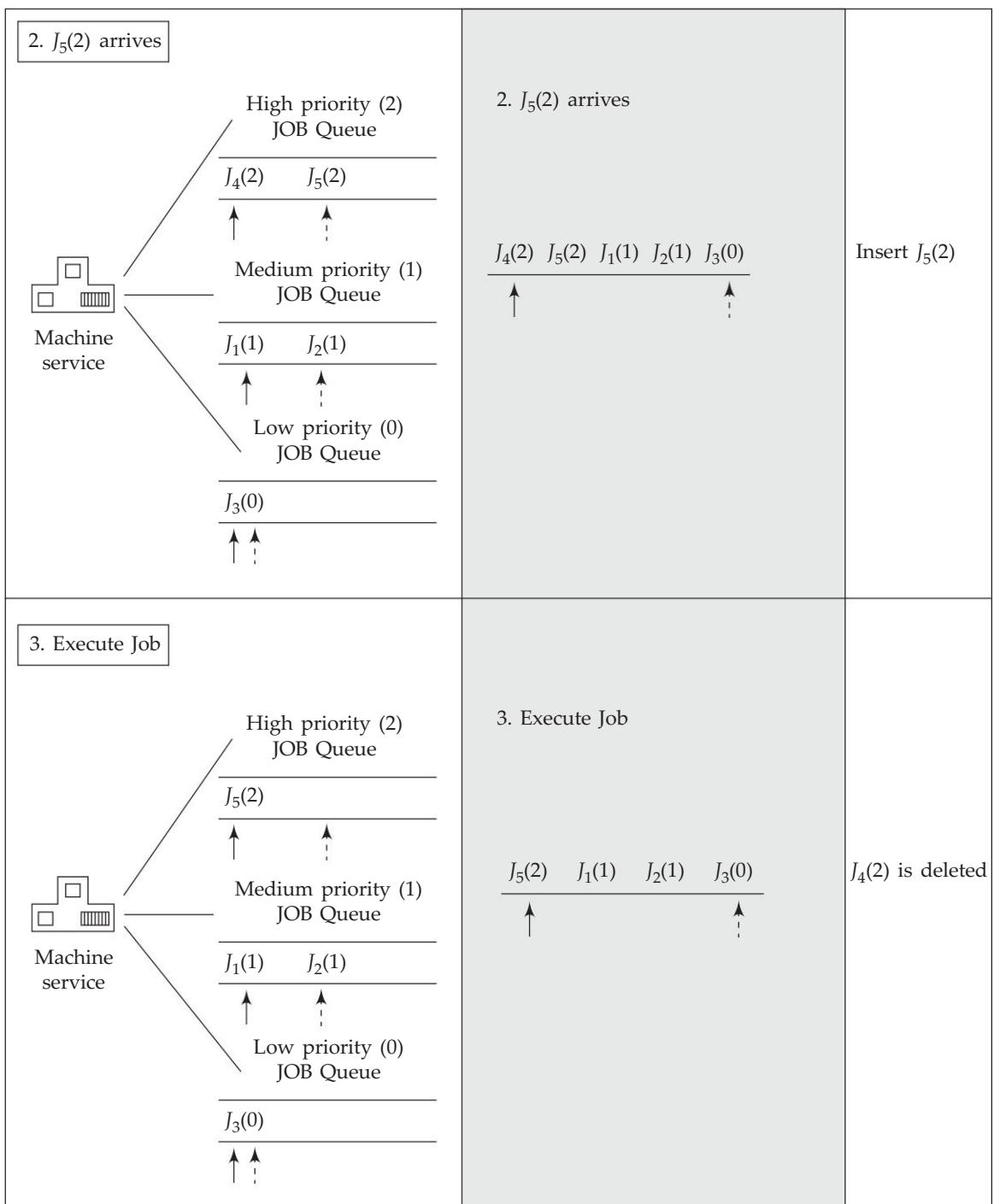
Operations on the JOB queue in the chronological order :

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

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

(Contd.)

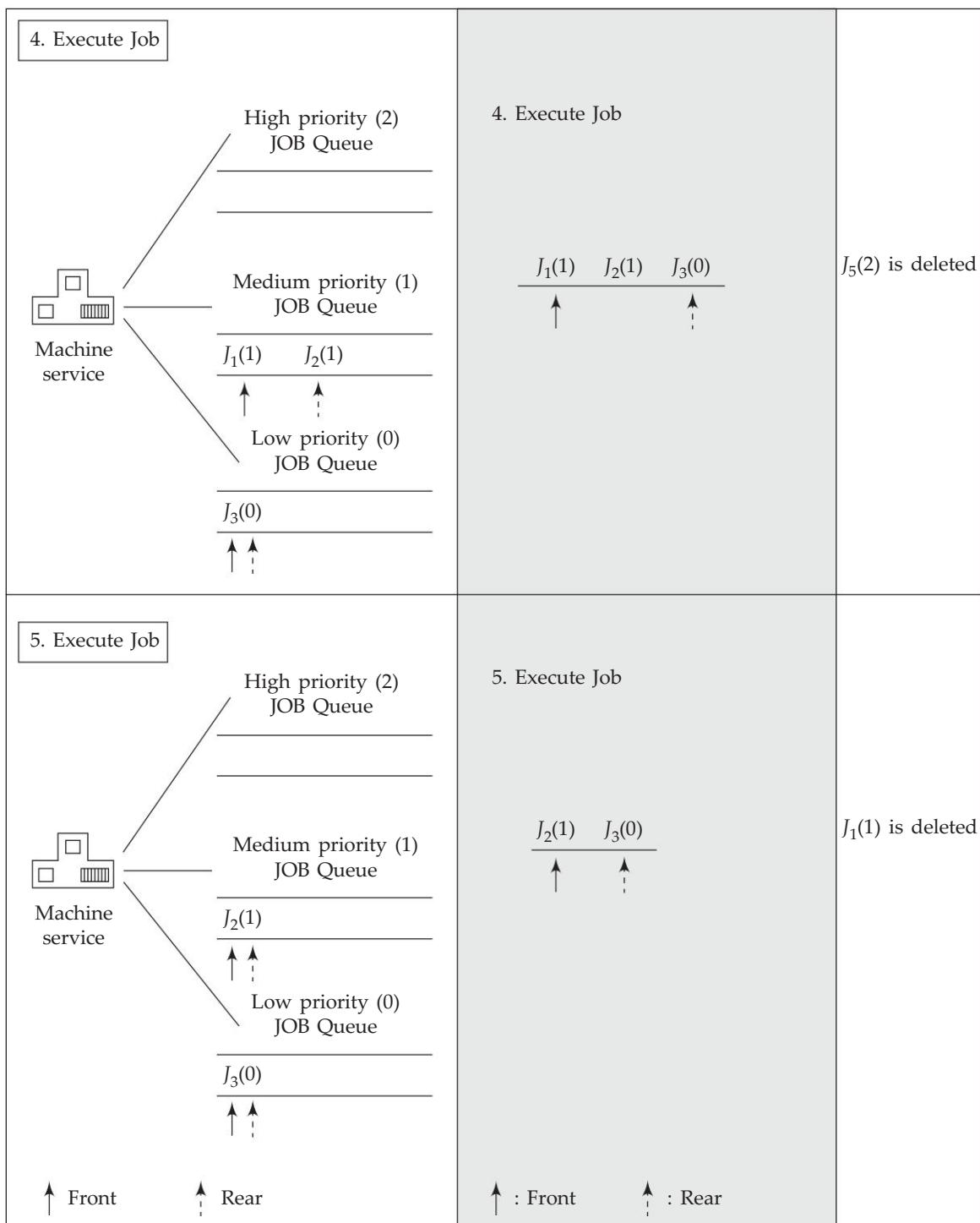
(Contd.)



(Contd.)

## Queues

(Contd.)

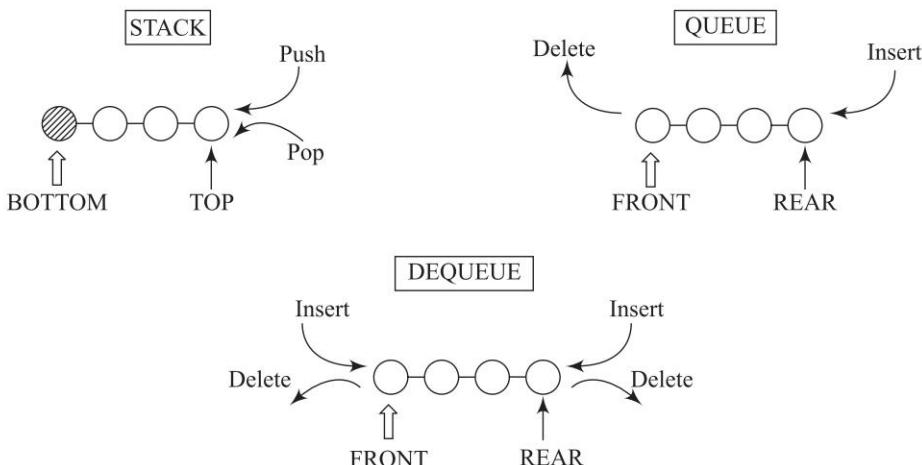


A variant of the implementation of a priority queue using multiple queues is to make use of a single two dimensional array to represent the list of queues and their contents. The number of rows in the array is equal to the number of priorities accorded to the data elements and the columns are equal to 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.

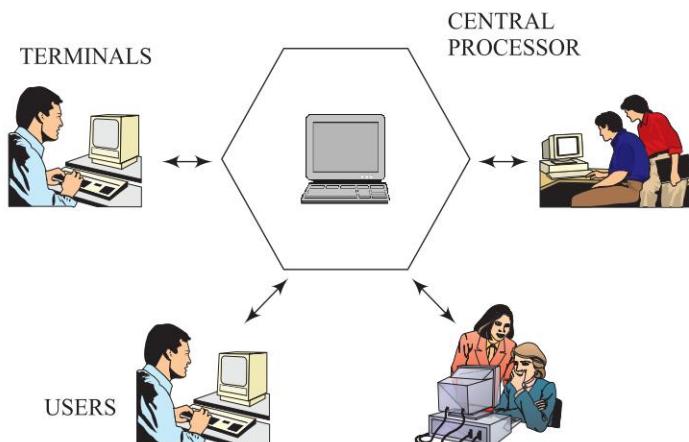
## Applications

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.

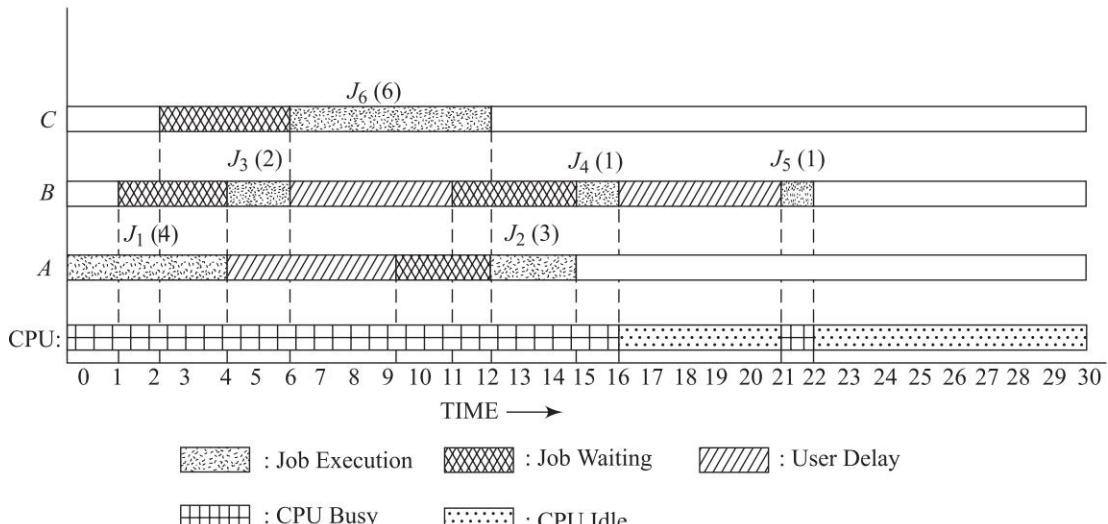
**Example 5.5** The following is a table of three users  $A, B, C$  with their job requests  $J_i (k)$  where  $i$  is the job number and  $k$  is the time required to execute the job.

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

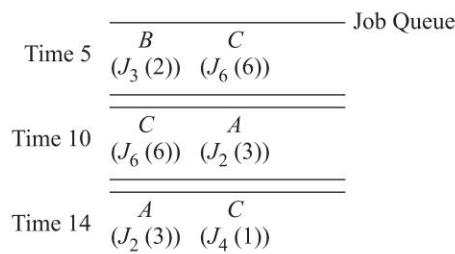
## Queues



**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.



**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

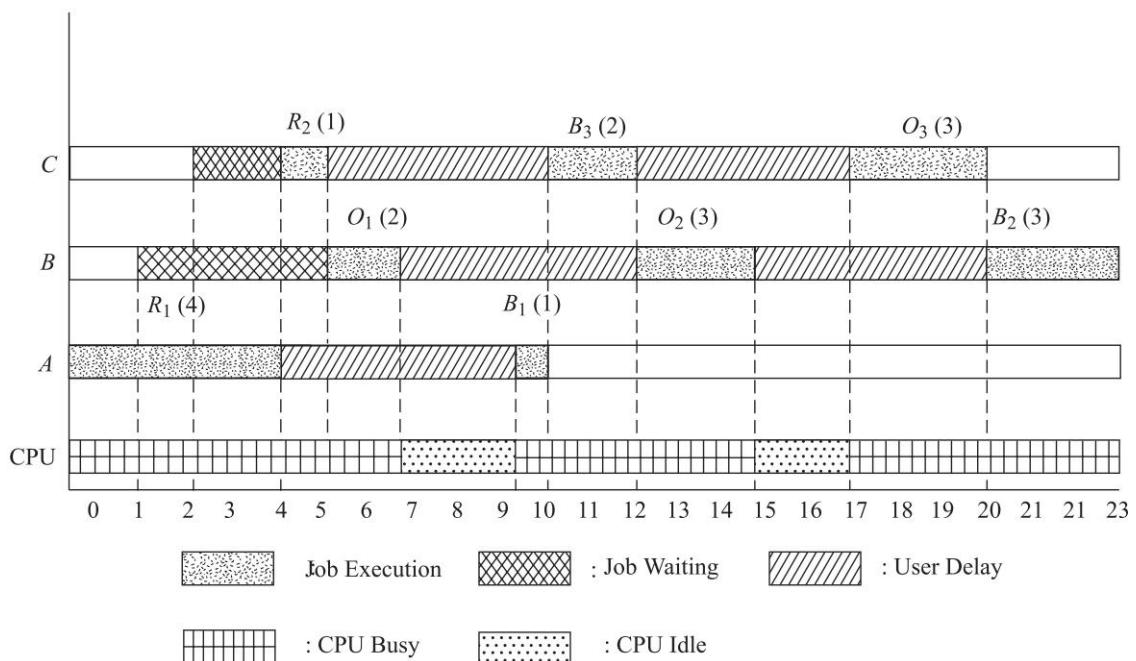
execution of a job request from  $O$ . In the case of queue  $B$ , before executing a job in queue  $B$ , the queues  $R$  and  $O$  should be empty. Example 5.6 illustrates the application of a priority queue in a time-sharing system with priority-based job requests.

**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.

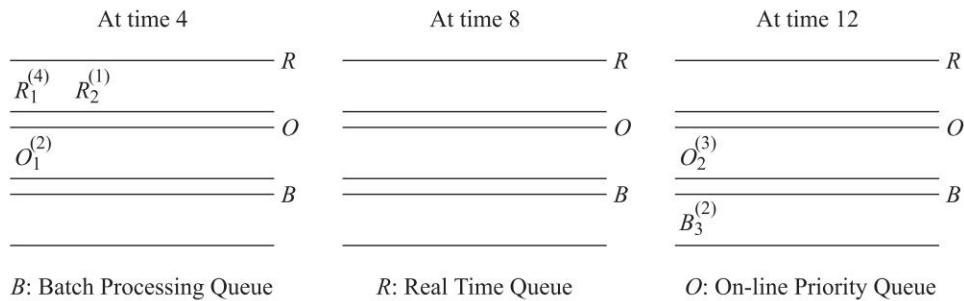
User	Job requests and their execution time in $\mu$ secs		
A	$R_1(4)$		$B_1(1)$
B	$O_1(2)$	$O_2(3)$	$B_2(3)$
C	$R_2(1)$	$B_3(2)$	$O_3(3)$

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

```

**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     | <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] | ... | ... | ... |
|       |                                                                                                                                                                                                                 |           |     |     |    |    |     |     |     |     |     |     |     |     |
| [0]   | [1]                                                                                                                                                                                                             | [2]       | [3] | [4] |    |    |     |     |     |     |     |     |     |     |
| 2-3   | <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] | 56  | 77  | ... |
|       |                                                                                                                                                                                                                 |           |     |     |    |    |     |     |     |     |     |     |     |     |
| [0]   | [1]                                                                                                                                                                                                             | [2]       | [3] | [4] |    |    |     |     |     |     |     |     |     |     |
| 4     | <table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td></td><td>56</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>       |           | 56  |     |    |    | [0] | [1] | [2] | [3] | [4] | 56  | 77  | ... |
|       | 56                                                                                                                                                                                                              |           |     |     |    |    |     |     |     |     |     |     |     |     |
| [0]   | [1]                                                                                                                                                                                                             | [2]       | [3] | [4] |    |    |     |     |     |     |     |     |     |     |
| 5     | <table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td></td><td>56</td><td>50</td><td></td><td></td></tr> <tr><td>[0]</td><td>[1]</td><td>[2]</td><td>[3]</td><td>[4]</td></tr> </table>     |           | 56  | 50  |    |    | [0] | [1] | [2] | [3] | [4] | 56  | 77  | ... |
|       | 56                                                                                                                                                                                                              | 50        |     |     |    |    |     |     |     |     |     |     |     |     |
| [0]   | [1]                                                                                                                                                                                                             | [2]       | [3] | [4] |    |    |     |     |     |     |     |     |     |     |
| 6     | <table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td></td><td>56</td><td>50</td><td>77</td><td></td></tr> <tr><td>[0]</td><td>[1]</td><td>[2]</td><td>[3]</td><td>[4]</td></tr> </table>   |           | 56  | 50  | 77 |    | [0] | [1] | [2] | [3] | [4] | 56  | 77  | ... |
|       | 56                                                                                                                                                                                                              | 50        | 77  |     |    |    |     |     |     |     |     |     |     |     |
| [0]   | [1]                                                                                                                                                                                                             | [2]       | [3] | [4] |    |    |     |     |     |     |     |     |     |     |
| 7     | <table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td></td><td></td><td>50</td><td>77</td><td></td></tr> <tr><td>[0]</td><td>[1]</td><td>[2]</td><td>[3]</td><td>[4]</td></tr> </table>     |           |     | 50  | 77 |    | [0] | [1] | [2] | [3] | [4] | 56  | 56  | ... |
|       |                                                                                                                                                                                                                 | 50        | 77  |     |    |    |     |     |     |     |     |     |     |     |
| [0]   | [1]                                                                                                                                                                                                             | [2]       | [3] | [4] |    |    |     |     |     |     |     |     |     |     |
| 8     | <table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td></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>   |           |     | 50  | 77 | 22 | [0] | [1] | [2] | [3] | [4] | 56  | 56  | ... |
|       |                                                                                                                                                                                                                 | 50        | 77  | 22  |    |    |     |     |     |     |     |     |     |     |
| [0]   | [1]                                                                                                                                                                                                             | [2]       | [3] | [4] |    |    |     |     |     |     |     |     |     |     |
| 9     | <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] | 56  | 56  | ... |
| 56    |                                                                                                                                                                                                                 | 50        | 77  | 22  |    |    |     |     |     |     |     |     |     |     |
| [0]   | [1]                                                                                                                                                                                                             | [2]       | [3] | [4] |    |    |     |     |     |     |     |     |     |     |
| 10    | <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] | 56  | 56  | ... |
| 56    |                                                                                                                                                                                                                 | 50        | 77  | 22  |    |    |     |     |     |     |     |     |     |     |
| [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></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] | 56    56    10<br><br>50    56    0<br>77    56    0<br>22    56    0<br>56    56    0 |
| 56    |                                                                                                                                                                                                                 | 50  | 77  | 22  |    |    |     |     |     |     |     |                                                                                                                                                                                                         |  |  |  |  |  |     |     |     |     |     |                                                                                        |
| [0]   | [1]                                                                                                                                                                                                             | [2] | [3] | [4] |    |    |     |     |     |     |     |                                                                                                                                                                                                         |  |  |  |  |  |     |     |     |     |     |                                                                                        |
|       |                                                                                                                                                                                                                 |     |     |     |    |    |     |     |     |     |     |                                                                                                                                                                                                         |  |  |  |  |  |     |     |     |     |     |                                                                                        |
| [0]   | [1]                                                                                                                                                                                                             | [2] | [3] | [4] |    |    |     |     |     |     |     |                                                                                                                                                                                                         |  |  |  |  |  |     |     |     |     |     |                                                                                        |
| 12-16 |                                                                                                                                                                                                                 |     |     |     |    |    |     |     |     |     |     |                                                                                                                                                                                                         |  |  |  |  |  |     |     |     |     |     |                                                                                        |

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.)

## Queues

(Contd.)

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

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]

$$\begin{matrix} 1 & \left[ \begin{matrix} - & - \end{matrix} \right] \\ 2 & \left[ \begin{matrix} - & - \end{matrix} \right] \\ 3 & \left[ \begin{matrix} - & - \end{matrix} \right] \end{matrix}$$

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

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

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

$DEQ:$

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

LEFT: 2

RIGHT: 5

Execute the following insertions and deletions on  $DEQ$ :

- (i)  $INSERT('S5', \text{LEFT})$
- (ii)  $INSERT('K9', \text{RIGHT})$
- (iii)  $DELETE()$
- (iv)  $INSERT('V7', \text{LEFT})$
- (v)  $INSERT('T5', \text{LEFT})$

**Solution:**

- (i)  $DEQ$  after the execution of operations
  - (i)  $INSERT('S5', \text{LEFT})$
  - (ii)  $INSERT('K9', \text{RIGHT})$

$DEQ:$

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

LEFT: 1

RIGHT: 6

- (ii)  $DEQ$  after the execution of  $DELETE()$

$DEQ:$

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

LEFT: 2

RIGHT: 6

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

(v) INSERT('T5', LEFT)

LEFT: 1

RIGHT: 6

DEQ:

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

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



## Review Questions

- Which among the following properties does not hold good in a queue?
  - A queue supports the principle of First come First served.
  - An enqueueing operation shrinks the queue length
  - A dequeuing operation affects the front end of the queue.
  - An enqueueing operation affects the rear end of the queue

(a) (i)                   (b) (ii)                   (c) (iii)                   (d) (iv)
- A linear queue  $Q$  is implemented using an array as shown below. The FRONT and REAR pointers which point to the physical front and rear of the queue, are also shown.

FRONT: 2      REAR: 3

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

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

- 2 and 4                   (b) 3 and 3                   (c) 3 and 4                   (d) 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                   (b) A, 2, 2                   (c) Y, 3, 3                   (d) A, 2, 3
- Given the following array implementation of a circular queue, with FRONT and REAR pointing to the physical front and rear of the queue,

FRONT: 3      REAR: 4

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

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

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

|     |     |     |     |     | 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 | $J_9(3), J_{10}(3), J_{11}(6)$                     | 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  $a_1^{(p_1)}, a_2^{(p_2)}, a_n^{(p_n)}$  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

# 6

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

In this chapter we detail linear data structures having a linked representation. We first list the 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.

## 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)

### 6.1 Introduction

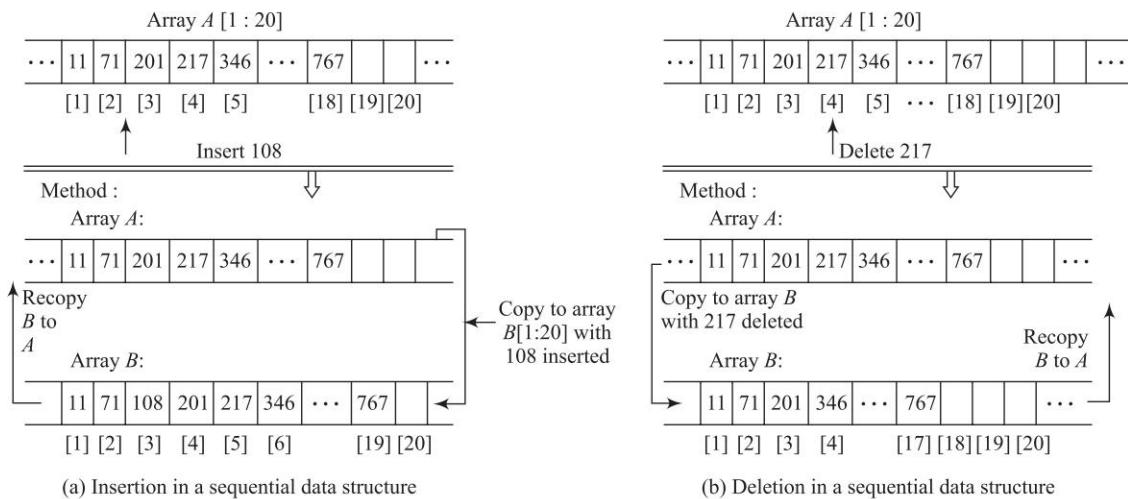
#### 6.2 Singly Linked Lists

#### 6.3 Circularly Linked Lists

#### 6.4 Doubly Linked Lists

#### 6.5 Multiply Linked Lists

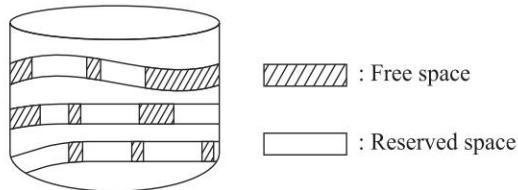
#### 6.6 Applications



**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

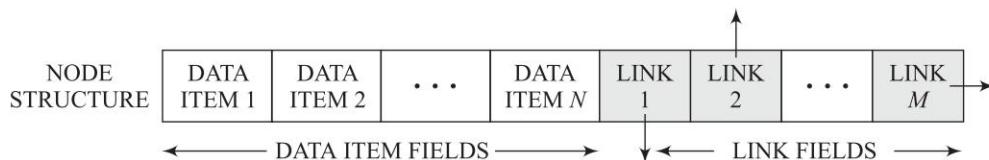
A linked representation serves to counteract the drawbacks of sequential representation by exhibiting the following merits:

- 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:

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

- 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.

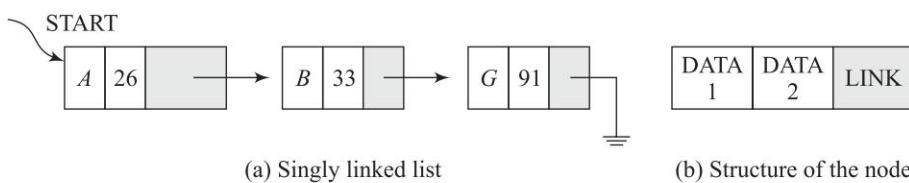
## Singly Linked Lists

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).

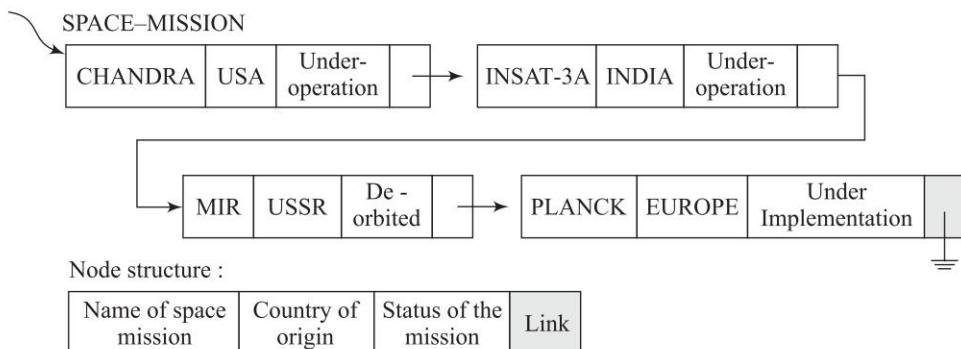
Figure 6.4 illustrates an example of a singly linked list and its node structure. Observe that the node in the list carries a single link which points to the node representing its immediate successor in the list of data elements.



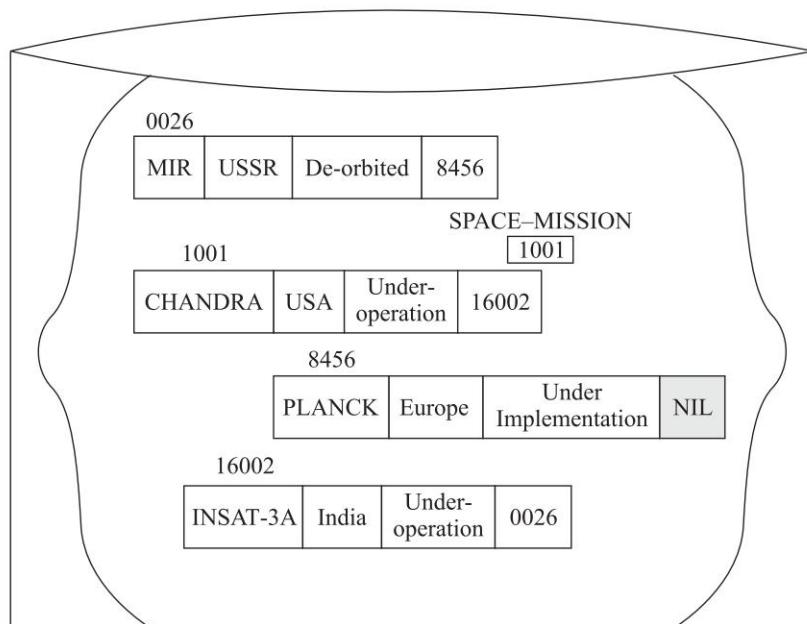
**Fig. 6.4** A singly linked list and its node structure

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.

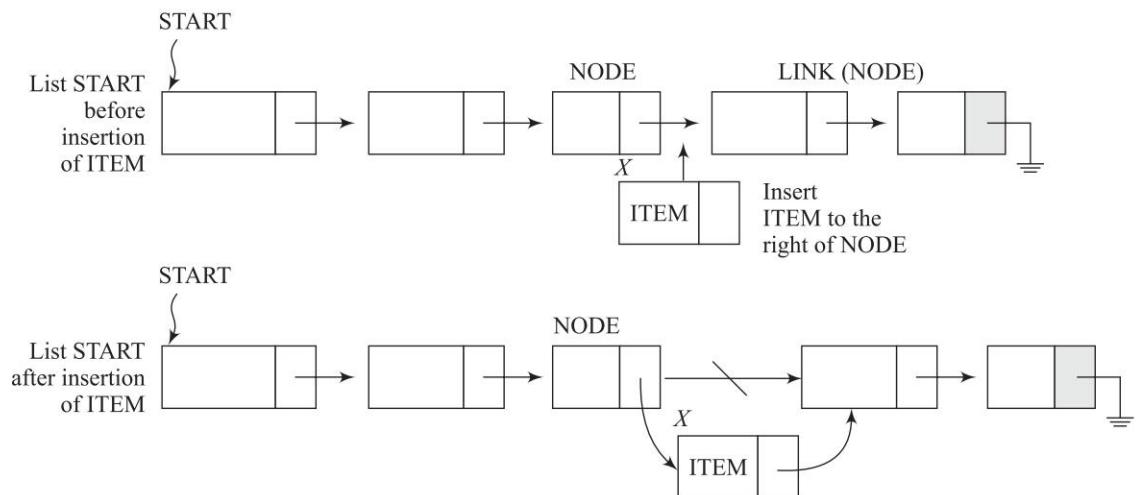


Fig. 6.6 Logical representation of insertion in a singly linked list

**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) 

|        |     |        |
|--------|-----|--------|
| APOLLO | 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).

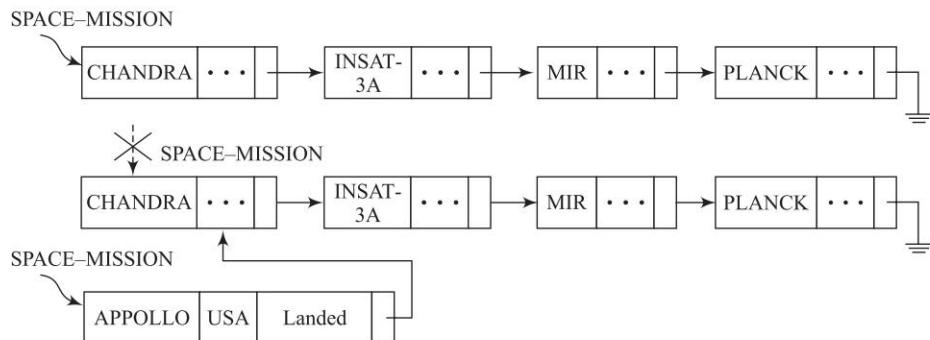
**Delete operation** Given a singly linked list *START*, the delete operation can acquire various forms such as deletion of a node *NODEY* next to that of a specific node *NODEX*, or more commonly deletion of a particular element in a list and so on. We now illustrate the deletion of a node which is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

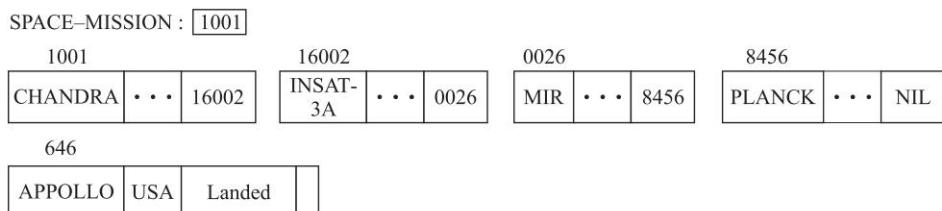
## Linked Lists

SPACE-MISSION  
list before  
insertion of  
APOLLO



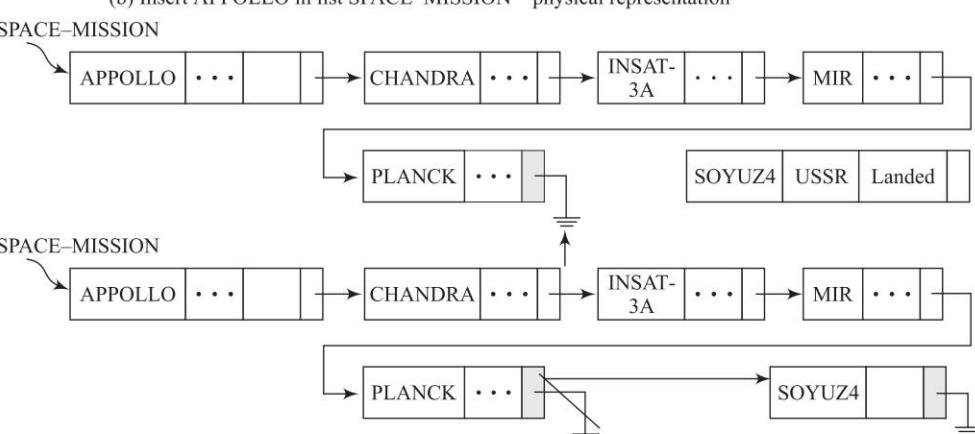
(a) Insert APOLLO in list SPACE-MISSION—logical representation

SPACE-MISSION  
list before  
insertion of  
APOLLO

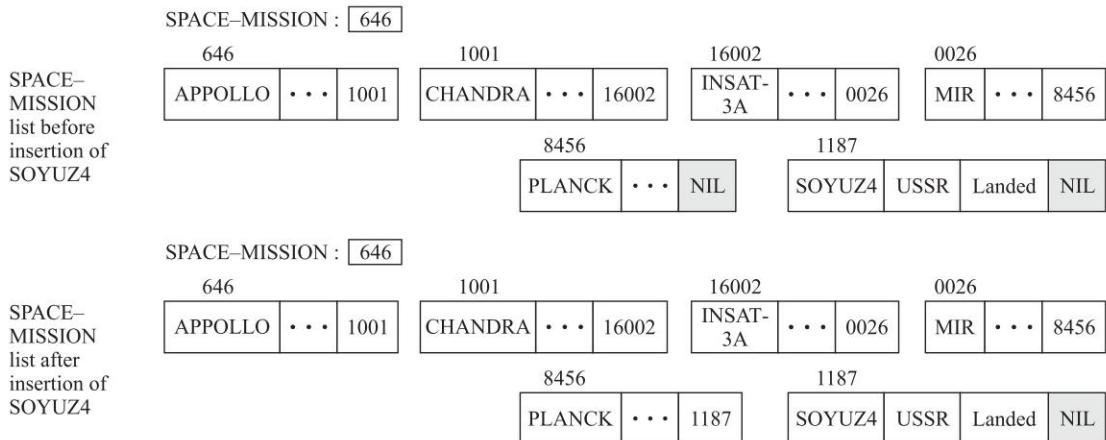


(b) Insert APOLLO in list SPACE-MISSION—physical representation

SPACE-MISSION  
list after  
insertion of  
APOLLO



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



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

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

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

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

```

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

```

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

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

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

- Delete CHANDRA
- Delete PLANCK

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

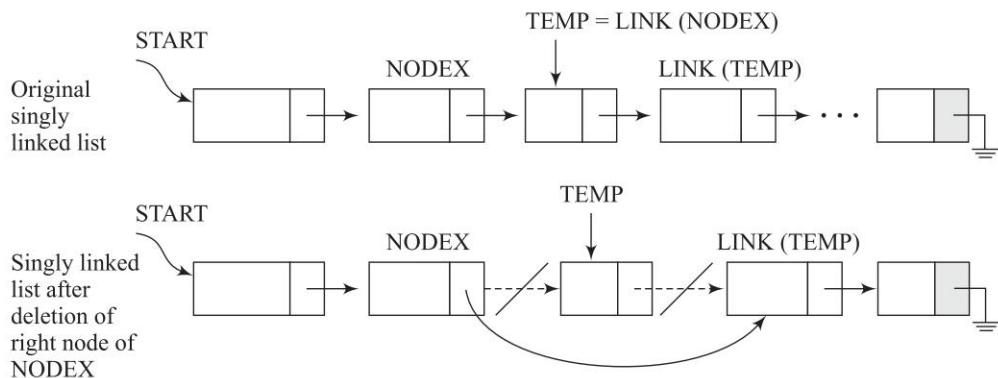


Fig. 6.8 Logical representation of deletion in a singly linked list

## Circularly Linked Lists

6.3

### Representation

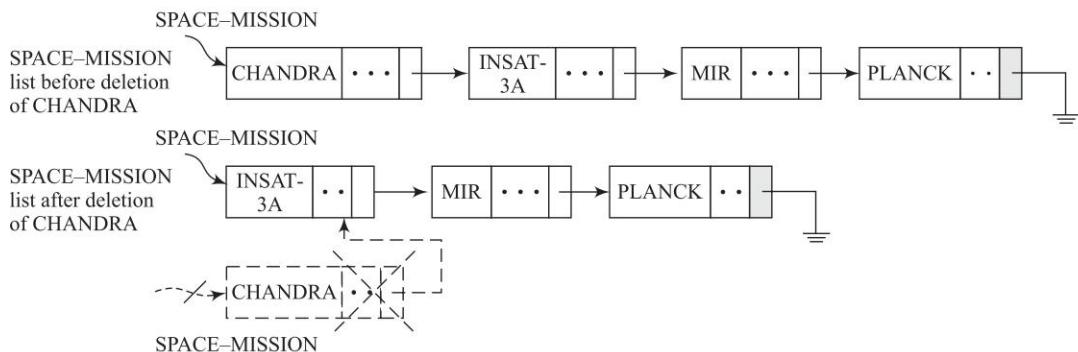
A normal singly linked list has its last node carrying a null pointer. For further improvement in processing one may replace the null pointer in the last node with the address of the first node in the list. Such a list is called as a *circularly linked list* or a *circular linked list* or simply a *circular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

### Advantages of circularly linked lists over singly linked lists

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

### Disadvantages of circularly linked lists

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



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

SPACE-MISSION : [1001]

|      |                     |       |                     |      |                |      |                  |
|------|---------------------|-------|---------------------|------|----------------|------|------------------|
| 1001 | CHANDRA   •   16002 | 16002 | INSAT-3A   •   0026 | 0026 | MIR   •   8456 | 8456 | PLANCK   •   NIL |
|------|---------------------|-------|---------------------|------|----------------|------|------------------|

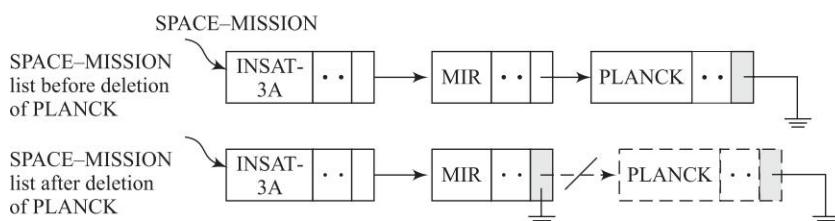
SPACE-MISSION list before insertion of CHANDRA

SPACE-MISSION : [16002]

|       |                     |      |                |      |                  |
|-------|---------------------|------|----------------|------|------------------|
| 16002 | INSAT-3A   •   0026 | 0026 | MIR   •   8456 | 8456 | PLANCK   •   NIL |
|-------|---------------------|------|----------------|------|------------------|

SPACE-MISSION list after deletion of CHANDRA

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



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

SPACE-MISSION : [16002]

|       |                     |      |                |      |                  |
|-------|---------------------|------|----------------|------|------------------|
| 16002 | INSAT-3A   •   0026 | 0026 | MIR   •   8456 | 8456 | PLANCK   •   NIL |
|-------|---------------------|------|----------------|------|------------------|

SPACE-MISSION list before insertion of PLANCK

SPACE-MISSION : [16002]

|       |                     |      |               |
|-------|---------------------|------|---------------|
| 16002 | INSAT-3A   •   0026 | 0026 | MIR   •   NIL |
|-------|---------------------|------|---------------|

SPACE-MISSION list after deletion of PLANCK

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

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

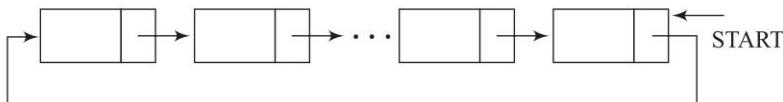


Fig. 6.10 Representation of a circular list

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

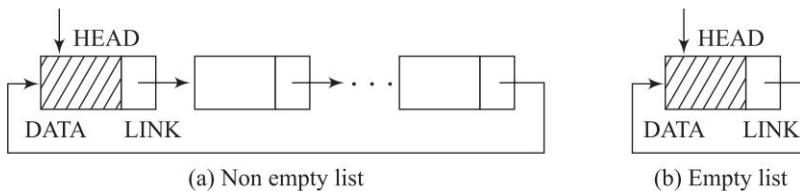


Fig. 6.11 A headed circularly linked list

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

Example 6.3 illustrates the functioning of circularly linked lists.

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

### Primitive operations on circularly linked lists

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

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

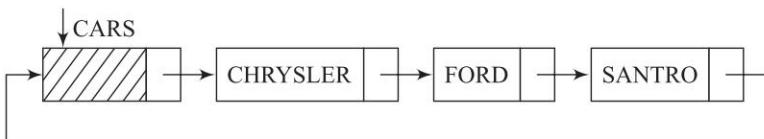
The sequence of operations to execute the insertion is:

```

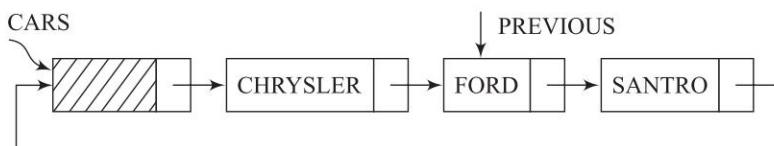
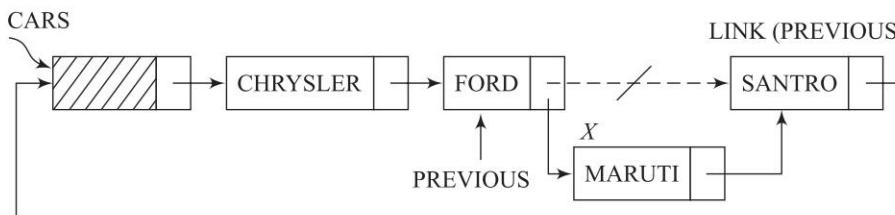
Call  GETNODE  (X);
      DATA  (X)  =  A;
      LINK  (X)  =  LINK (P);
      LINK  (P)  =  X;
  
```

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

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



(a) The headed circularly linked list CARS

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

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

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

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

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

$P = X$

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

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

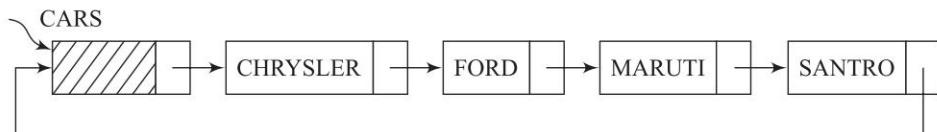
The sequence of operations to execute the deletion are:

```

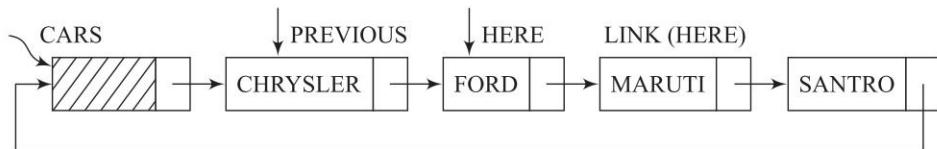
PTR = LINK(P);
Y = DATA(PTR);
LINK(P) = LINK(PTR);
Call RETURN(PTR);
    
```

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

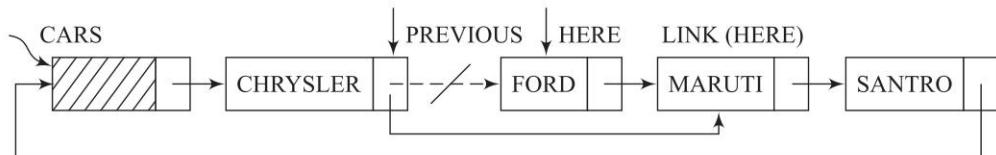
Observe that the primitive operations (i) and (iii) when combined, results in the circularly linked list working as a stack and operations (ii) and (iii) when combined, results in the circularly linked list working as a queue.



(a) The headed circularly linked list CARS

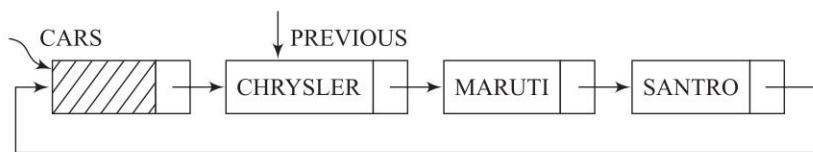


(b) Obtain the address of the node containing FORD (node HERE) by searching for it in the list CARS and its predecessor node (node PREVIOUS)



(c) Reset links to delete FORD

LINK (PREVIOUS) = LINK (HERE)



(d) Dispose node HERE

RETURN (HERE)

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

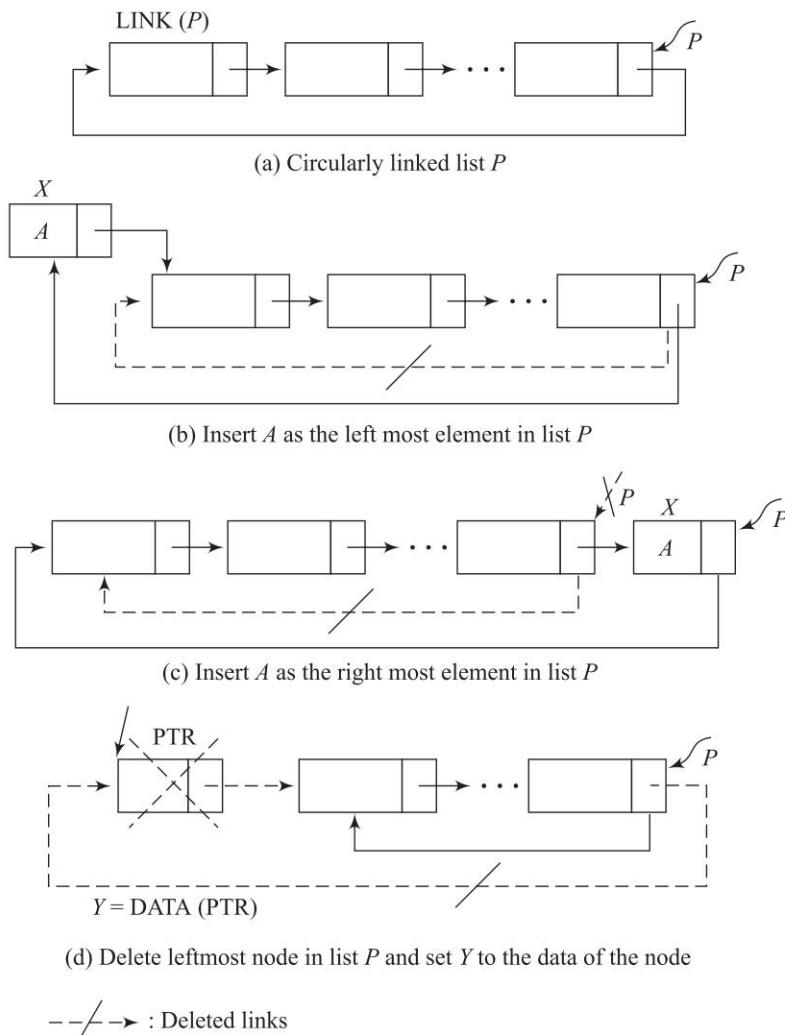
## Other operations on circularly linked lists

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

```

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

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

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

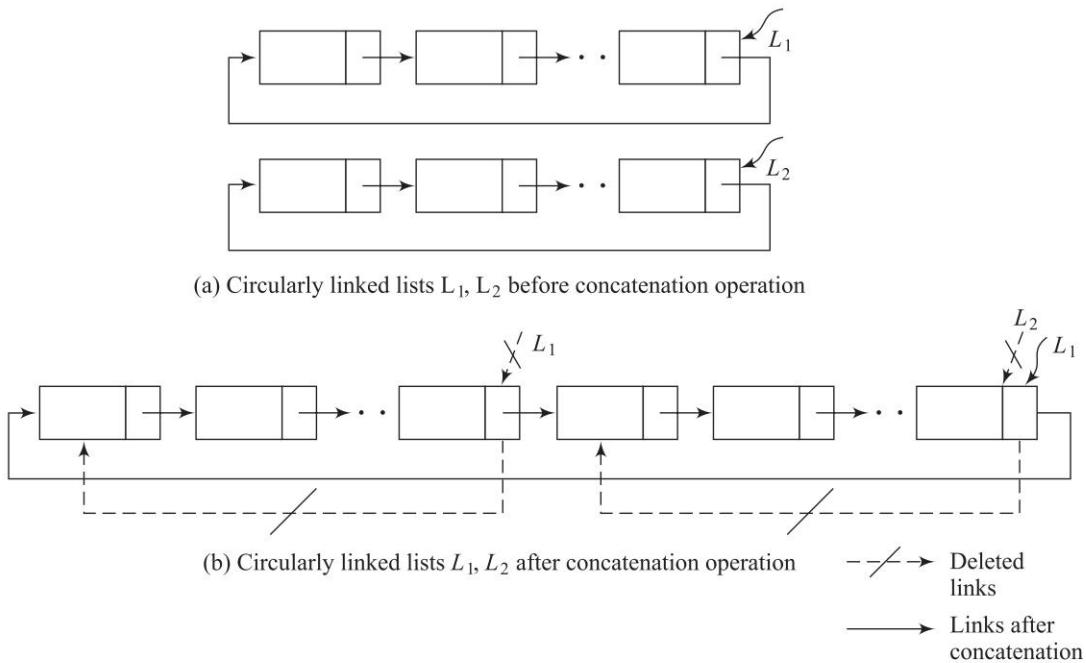
## Doubly Linked Lists

6.4

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

### Representation of a doubly linked list

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



**Fig. 6.15 Concatenation of two circularly linked lists**

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

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

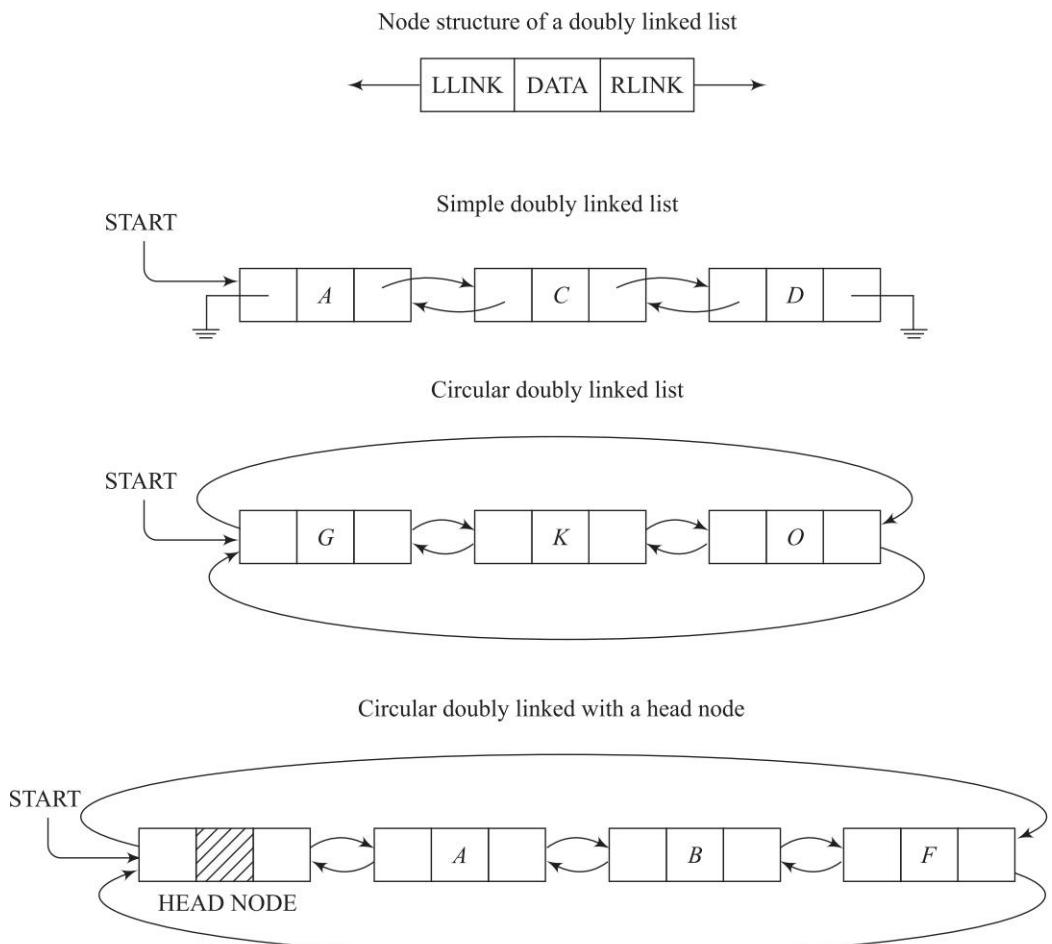
Example 6.4 illustrates a doubly linked list and its logical and physical representations.

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

### Advantages and disadvantages of a doubly linked list

Doubly linked lists have the following advantages:

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



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

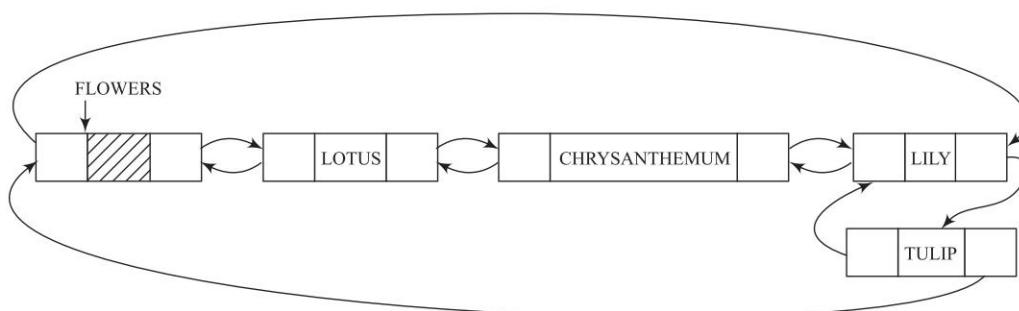
endowed with the capability to move round the list to find the predecessor node. However, in the latter case, if the list is too long it may render the delete operation inefficient.

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

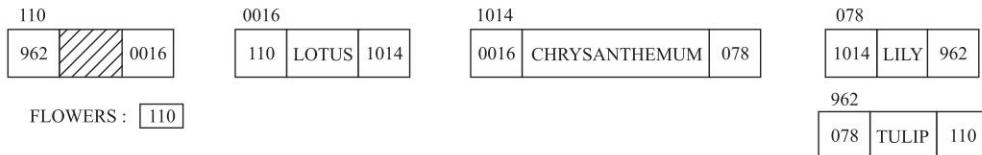
## Operations on doubly linked lists

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

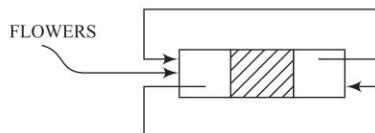
**Insert Operation** Let  $P$  be a headed circular doubly linked list which is non empty. Algorithm 6.4 illustrates the insertion of a node  $X$  to the right of node  $Y$ . Figure 6.18(a) shows the logical representation of list  $P$  before and after insertion.



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



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



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



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

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

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

```

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

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

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

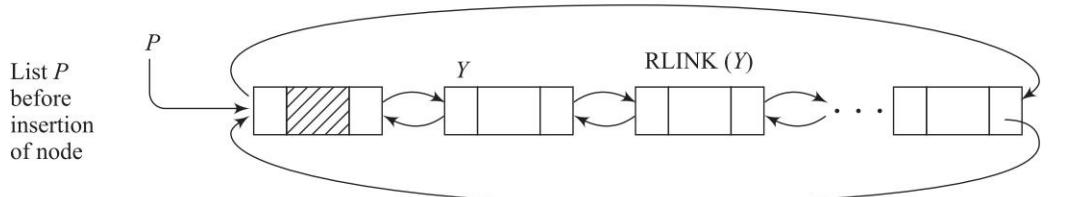
**Algorithm 6.5:** Delete node X from a headed circular doubly linked list P

```

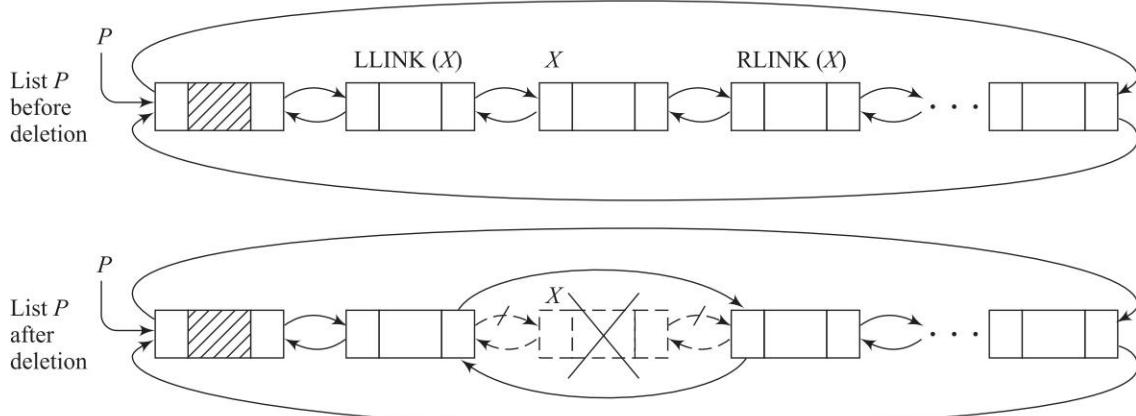
procedure DELETE_DL(P, X)
    if (X = P) then ABANDON_DELETE;
    else
        {RLINK(LLINK(X)) = RLINK(X);
        LLINK(RLINK(X)) = LLINK(X);
        Call RETURN(X); }
    end DELETE_DL.
```

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

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



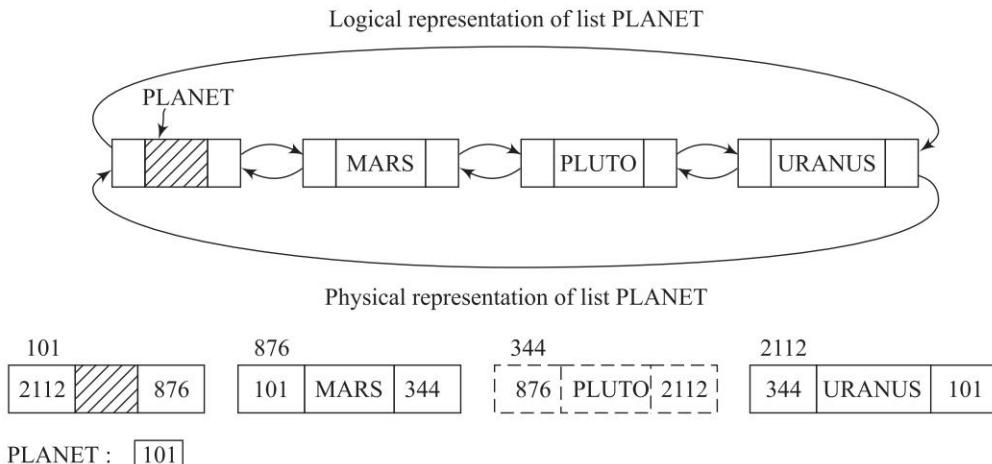
(a) Insertion of node X into a headed circular doubly linked list P, after node Y



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

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

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



**Fig. 6.19** Logical and physical representation of list PLANET

## Multiply Linked Lists

## 6.5

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

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

Example 6.6 illustrates an example of a multiply linked list.

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

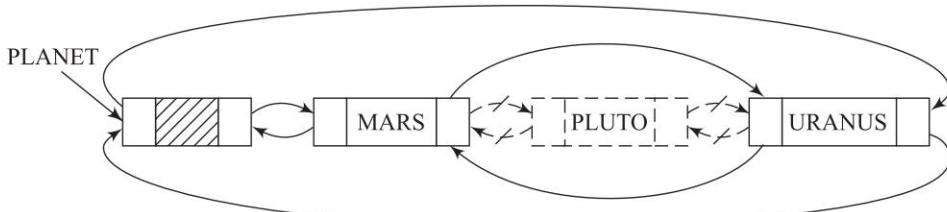
Consider Table 6.1 illustrating details pertaining to 6 students.

**Table 6.1** Student details for representation as a multiply linked list

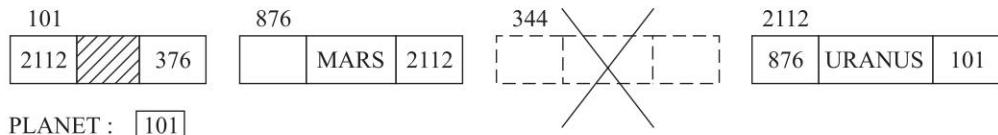
| Name of the Student | Roll # | Number of Credits Registered | Sports Club Membership | Day Student | Department       |
|---------------------|--------|------------------------------|------------------------|-------------|------------------|
| AKBAR               | CS02   | 200                    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

(Contd.)

Logical representation of list PLANET after deletion of PLUTO

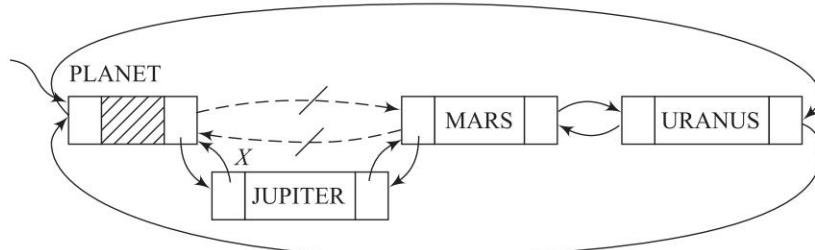


Physical representation of list PLANET after deletion of PLUTO

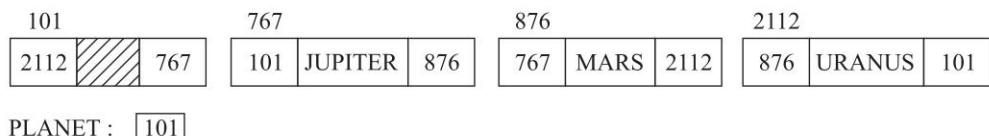


(a) Logical and physical representation of list PLANET after deletion of PLUTO

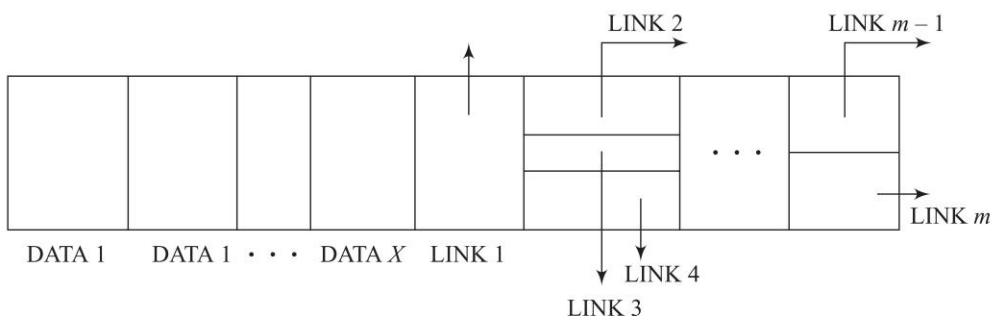
Logical representation of list PLANET after insertion of JUPITER

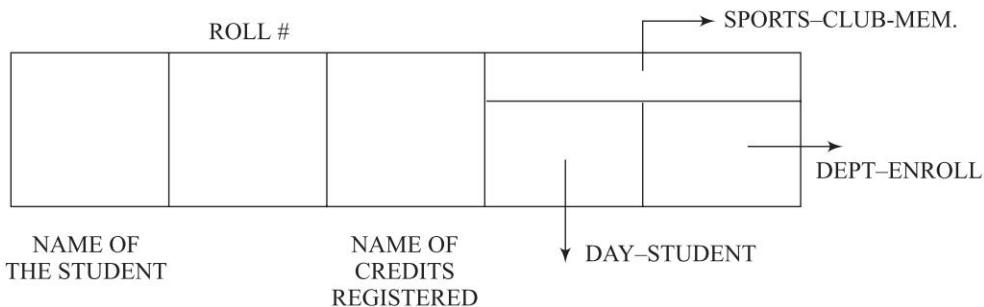


Physical representation of list PLANET after insertion of JUPITER



(b) Logical and physical representation of list PLANET after insertion of JUPITER

**Fig. 6.20** *Deletion of PLUTO and insertion of JUPITER in list PLANET***Fig. 6.21** *The node structure of a multiply linked list*



**Fig. 6.22** Node structure of the multiply linked list STUDENT

(Contd.)

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

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

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

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

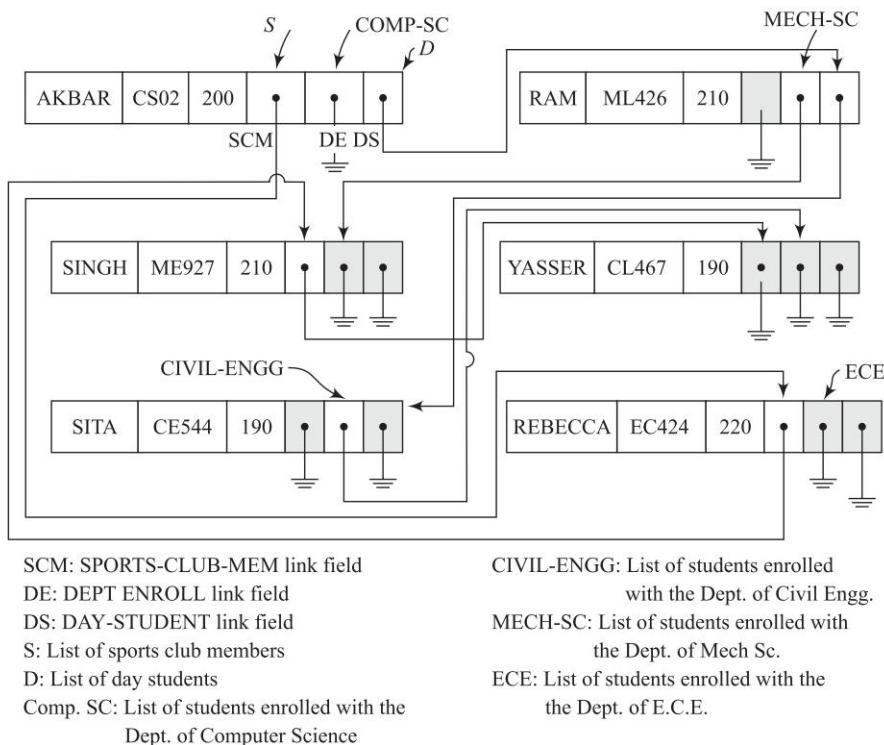
To delete REBECCA from the list of sports club members of the multiply linked list STUDENT, we undertake a sequence of operations as shown in Fig. 6.25. Observe how the node for REBECCA continues to participate in the other lists despite its deletion from the list *S*.

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

## Applications

## 6.6

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



**Fig. 6.23** Multiply linked list structure of list STUDENT

- Addition of polynomials and
- Representation of a sparse matrix

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

## Addition of polynomials

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

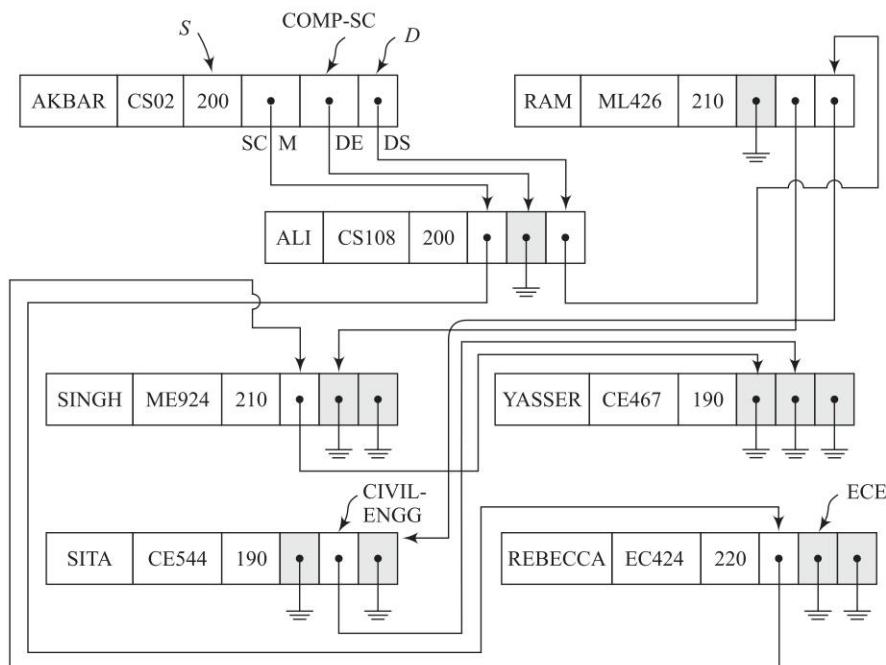
$$\begin{aligned} \text{Let } P_1 &: 2x^6 + x^3 + 5x + 4 \text{ and} \\ P_2 &: 7x^6 + 8x^5 - 9x^3 + 10x^2 + 14 \end{aligned}$$

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

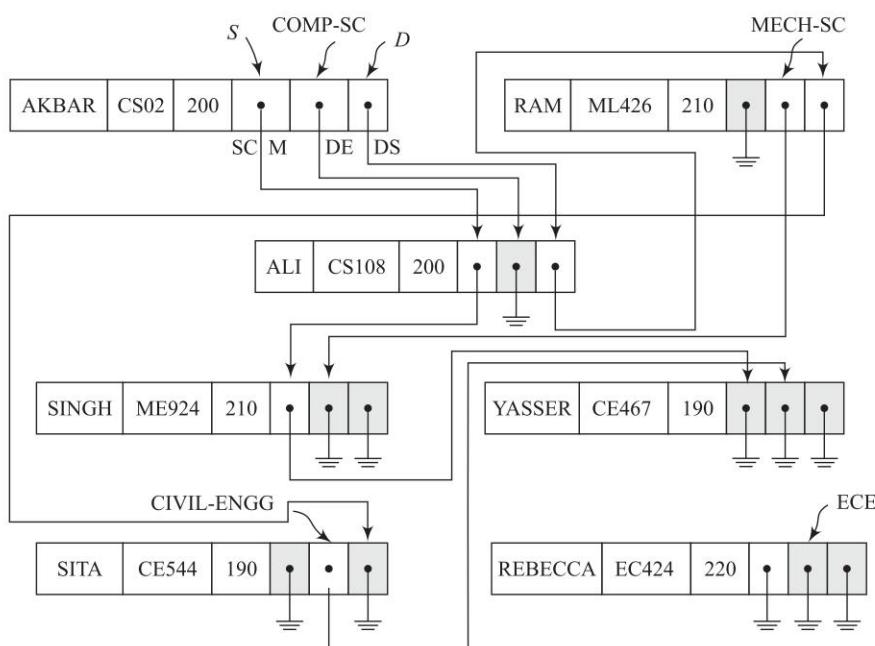
$$P_1 + P_2 = 9x^6 + 8x^5 - 8x^3 + 10x^2 + 5x + 18$$

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

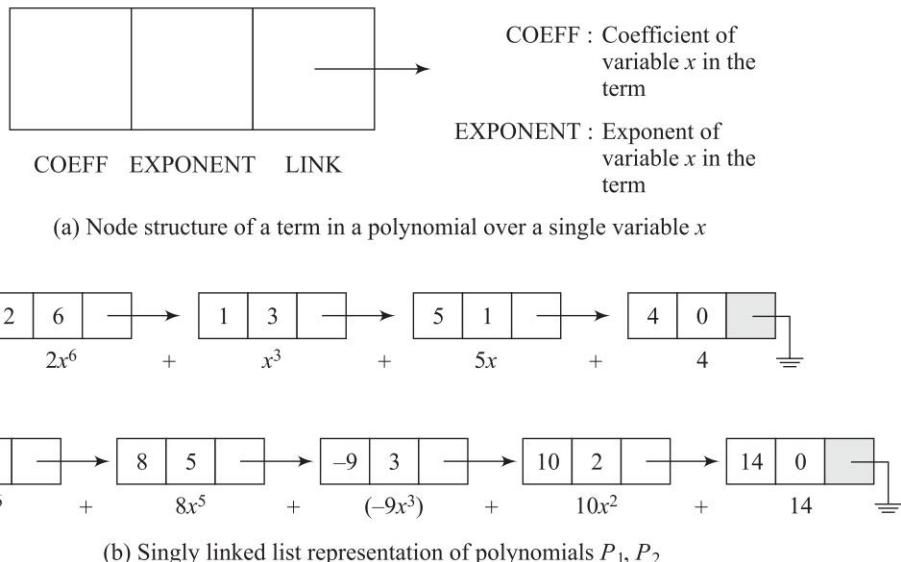
To add the two polynomials, we presume that the singly linked lists have their nodes arranged in the decreasing order of the exponents of the variable  $x$ .



**Fig. 6.24 Insert ALI into the multiply linked list STUDENT**



**Fig. 6.25 Delete REBECCA from the sports club membership list of the multiply linked list STUDENTS**



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

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

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

```

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

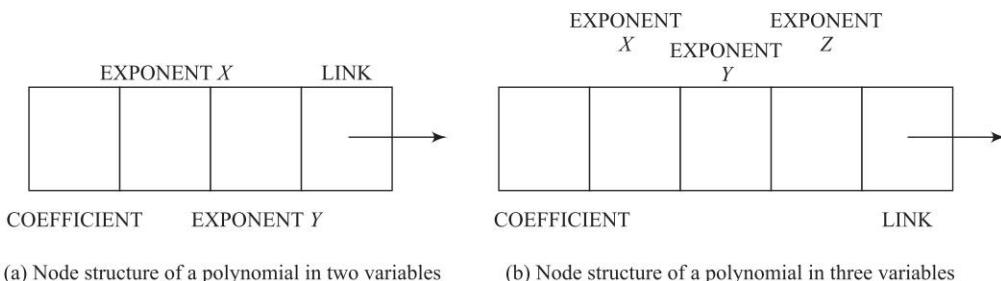
```

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

```

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

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



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

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

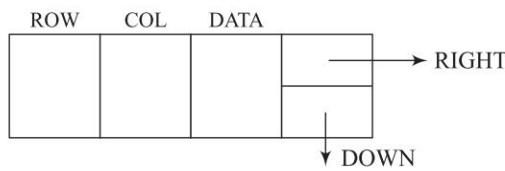
## Sparse matrix representation

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

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

|    |    |   |   |   |   |
|----|----|---|---|---|---|
| 0  | 1  | 0 | 0 | 0 | 0 |
| 0  | 0  | 0 | 0 | 0 | 0 |
| -2 | 0  | 0 | 1 | 0 | 0 |
| 0  | 0  | 0 | 0 | 0 | 0 |
| 0  | -3 | 0 | 0 | 0 | 0 |
| 0  | 0  | 0 | 0 | 0 | 0 |
| 0  | 0  | 0 | 0 | 0 | 1 |

(a) Sparse matrix



(b) Node structure of the multiply linked list

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

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

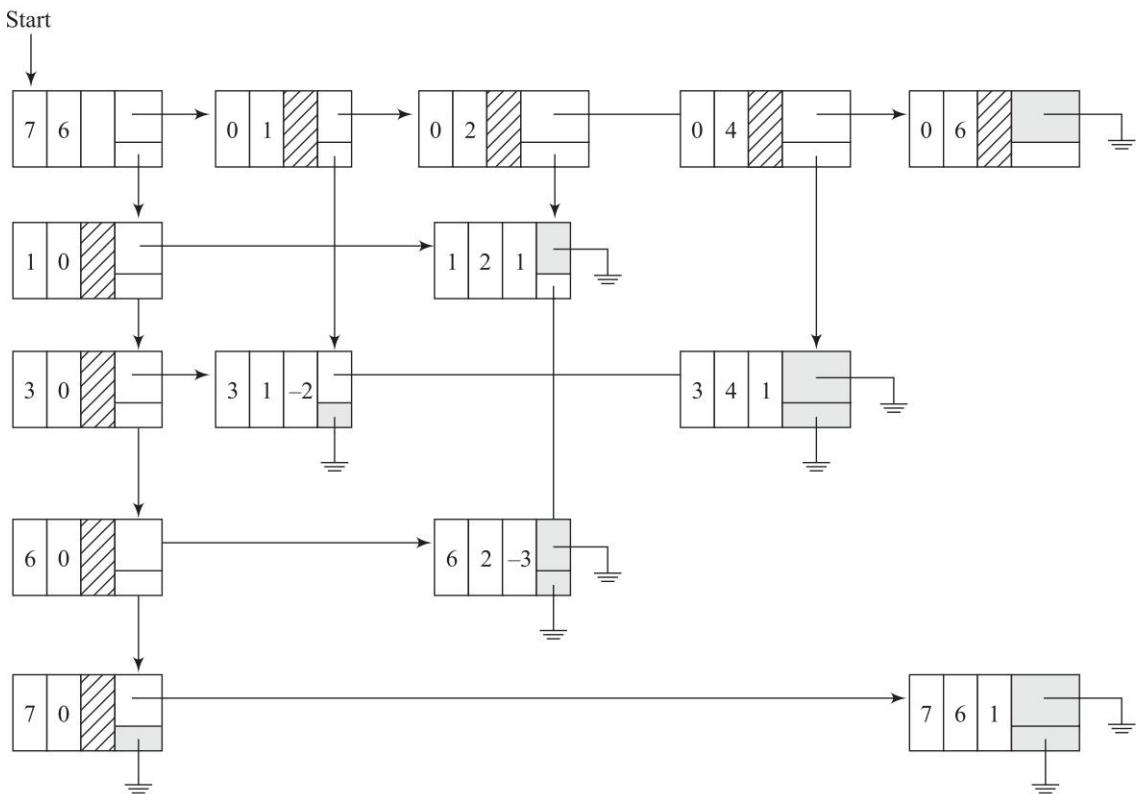
### ADT for Links

#### Data objects:

Addresses of the nodes holding data and null links

#### Operations:

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



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

### ADT for Singly Linked Lists

#### Data objects:

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

#### Operations:

- Check if list LIST is empty  
CHECK\_LIST\_EMPTY (LIST) (Boolean function)
- Insert ITEM into the list LIST as the first element  
INSERT\_FIRST (LIST, ITEM)
- Insert ITEM into the list LIST as the last element  
INSERT\_LAST (LIST, ITEM)
- Insert ITEM into the list LIST in order  
INSERT\_ORDER (LIST, ITEM)
- Delete the first node from the list LIST  
DELETE\_FIRST (LIST)
- Delete the last node from the list LIST  
DELETE\_LAST (LIST)

- Delete ITEM from the list LIST  
DELETE\_ELEMENT (LIST, ITEM)
- Advance Link to traverse down the list  
ADVANCE\_LINK (LINK)
- Store ITEM into a node whose address is X  
STORE\_DATA (X, ITEM)
- Retrieve data of a node whose address is X and return it in ITEM  
RETRIEVE\_DATA (X, ITEM)
- Retrieve link of a node whose address is X and return the value in LINK1  
RETRIEVE\_LINK (X, LINK1)



## Summary

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

## Illustrative Problems

**Problem 6.1** Write a pseudocode procedure to insert NEW\_DATA as the first element in a singly linked list  $T$ .

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

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

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

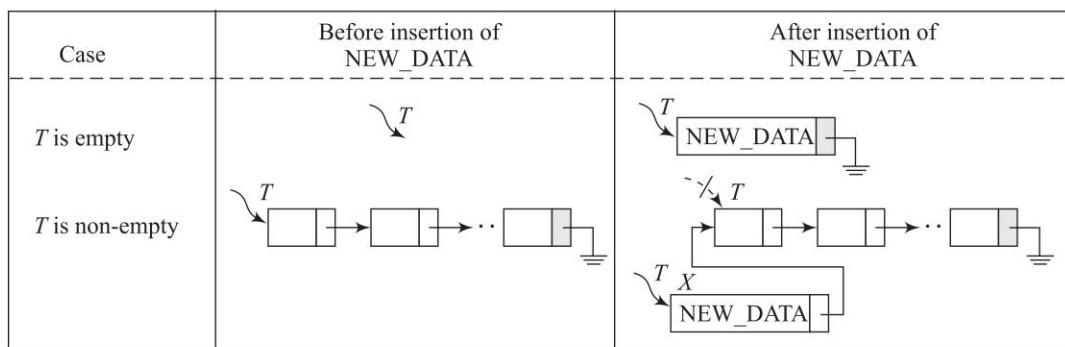


Fig. I 6.1

The general procedure in pseudocode:

```

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

```

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

**Solution:** The logical representation of the list  $T$  before and after insertion of NEW\_DATA as the  $k^{\text{th}}$  element in the list is shown in Fig. I 6.2.

The pseudocode procedure is:

```

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

```

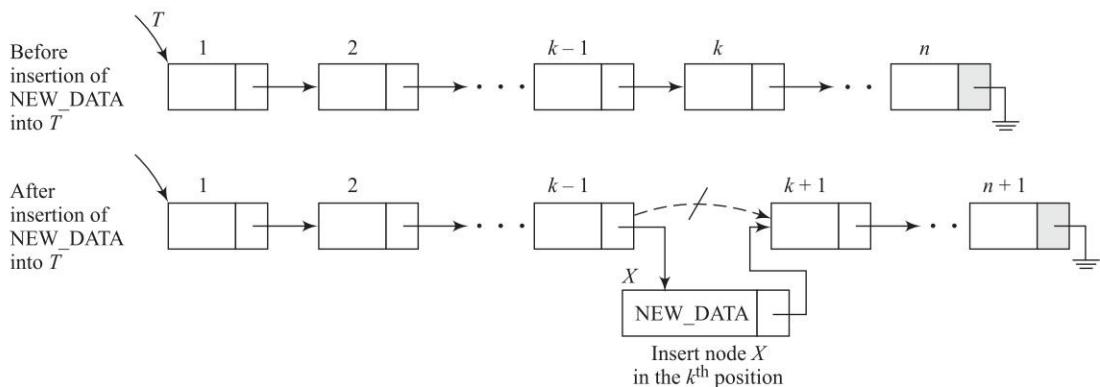


Fig. I 6.2

```

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

```

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

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

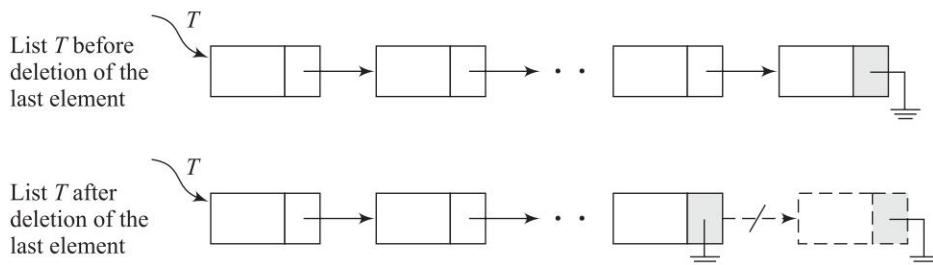


Fig. I 6.3

The pseudocode procedure is

```

procedure DELETE_LAST ( $T$ )
    if ( $T$  = NIL) then {call ABANDON_DELETE;}
    else
        { TEMP =  $T$ 
        while (LINK(TEMP)  $\neq$  NIL)

```

```

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

```

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

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

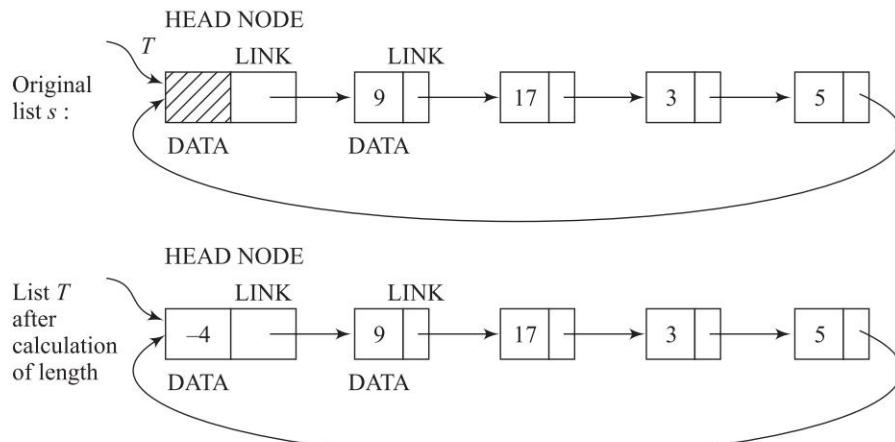


Fig. I 6.4

The pseudocode procedure is:

```

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

```

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

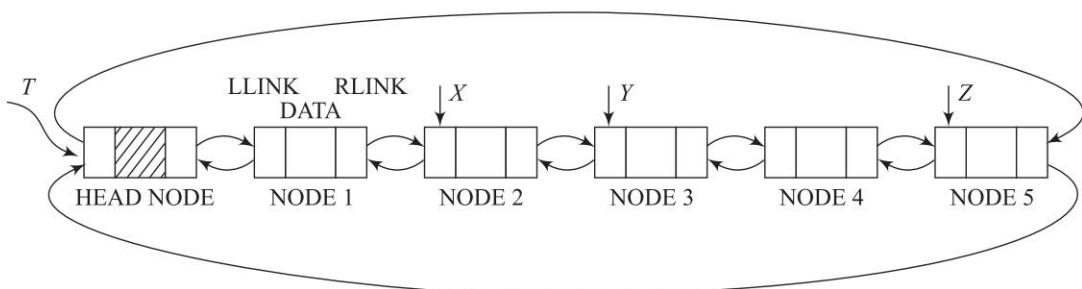


Fig. I 6.5

- (i) Express the DATA field of Node 5
- (ii) Express the DATA field of Node 1 as referenced from head node  $T$
- (iii) Express the left link of Node 1 as referenced from Node 2
- (iv) Express the right link of Node 4 as referenced from Node 5

**Solution:**

- (i) DATA (Z)
- (ii) DATA (RLINK( $T$ ))
- (iii) LLINK(LLINK(X))
- (iv) RLINK(LLINK(Z))

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

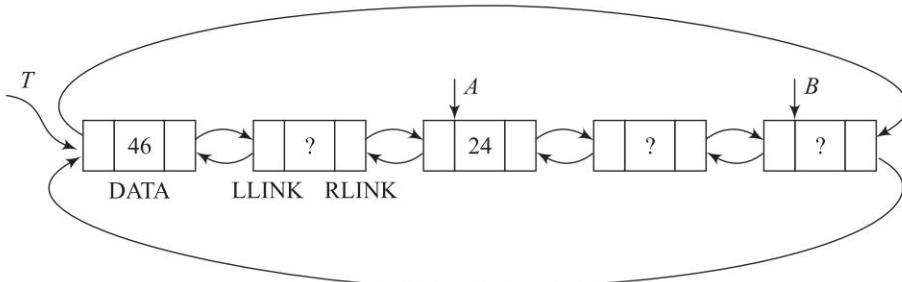


Fig. I 6.6(a)

- (i) DATA (B) = DATA (LLINK(RLINK(A))) + DATA (LLINK(RLINK(T)))
- (ii) DATA (LLINK(B)) = DATA (B) + 10
- (iii) DATA (RLINK(RLINK(B))) = DATA (LLINK(LLINK (B)))

**Solution:**

- (i) DATA(B) = DATA(A) + DATA(T)  

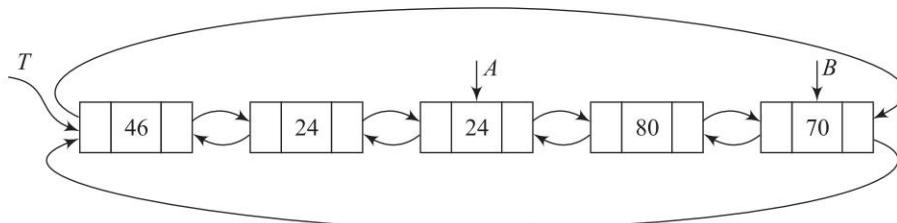
$$\begin{aligned} & (\because \text{LLINK(RLINK(A))} = A \text{ and } \text{LLINK(RLINK(T))} = T) \\ & = 24 + 46 \\ & = 70 \end{aligned}$$
- (ii) DATA (LLINK(B)) = DATA(B) + 10  

$$\begin{aligned} & = 70 + 10 \\ & = 80 \end{aligned}$$

## Linked Lists

$$\begin{aligned}
 \text{(iii) } \text{DATA}(\text{RLINK}(\text{RLINK}(B))) &= \text{DATA}(A) \\
 &= 24 \\
 (\because \text{LLINK(LLINK}(B)) &= A)
 \end{aligned}$$

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



**Fig. I 6.6(b)**

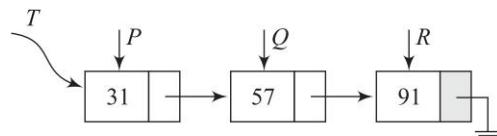
**Problem 6.7** In a programming language (Pascal) the declaration of a node in a singly linked list is shown in Fig. I 6.7(a). The list referred to for the problem is shown in Fig. I 6.7(b). Given  $P$  to be a pointer to a node, the instructions  $\text{DATA}(P)$  and  $\text{LINK}(P)$  referring to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

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

```

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

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

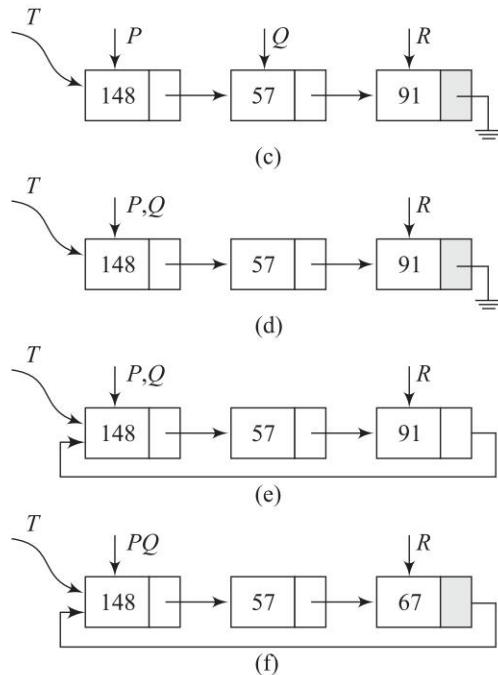


(b) A single linked list  $T$

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

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

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



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

$$\begin{aligned} \text{(i)} \quad & P \uparrow .\text{DATA} := Q \uparrow .\text{DATA} + R \uparrow .\text{DATA} \\ & P \uparrow .\text{DATA} := 57 + 91 = 148 \end{aligned}$$

$$\text{(ii)} \quad Q := P$$

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

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

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

$$\begin{aligned} \text{(iii)} \quad & R \uparrow .\text{DATA} := Q \uparrow .\text{LINK} \uparrow .\text{DATA} + 10 \\ & := 57 + 10 \\ & := 67 \end{aligned}$$

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

**Solution:**

- (i)  $\text{RLINK}(\text{RLINK}(X)) = \text{NIL}$   
or  
 $\text{RLINK}(\text{LLINK}(T)) = \text{NIL}$
- (ii)  $\text{LINK}(\text{LINK}(Y)) = T$
- (iii)  $\text{RLINK}(T) = \text{RLINK}(\text{RLINK}(T))$

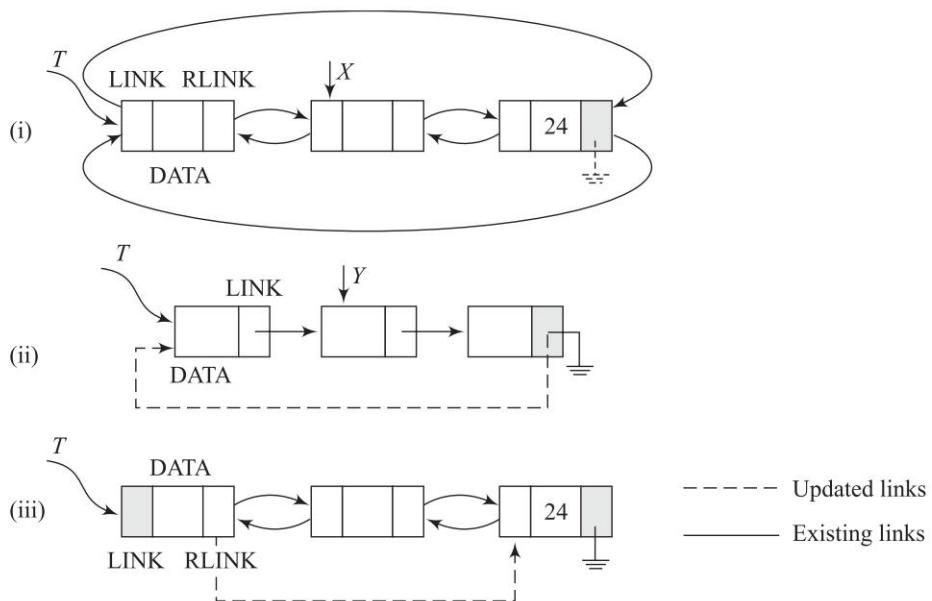


Fig. I 6.8



## Review Questions

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

| TENNIS_STARS: |   |
|---------------|---|
|               | 2 |

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

- The number of data elements in the list **TENNIS\_STARS** is
  - (a) 3
  - (b) 2
  - (c) 5
  - (d) 7

2. The successor of 'graf' in the list TENNIS\_STARS is  
 (a) navaratilova      (b) sabatini      (c) nirupama      (d) chris
3. In the list TENNIS\_STARS, DATA ( RLINK(LLINK(5)) ) = -----  
 (a) mirza      (b) graf      (c) nirupama      (d) chris
4. Given the singly linked list  $T$  shown in Fig. R6.4, the following code inserts the node containing the data "where\_am\_i"

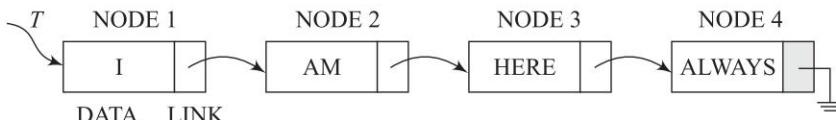


Fig. R6.4

```

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

```

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

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

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

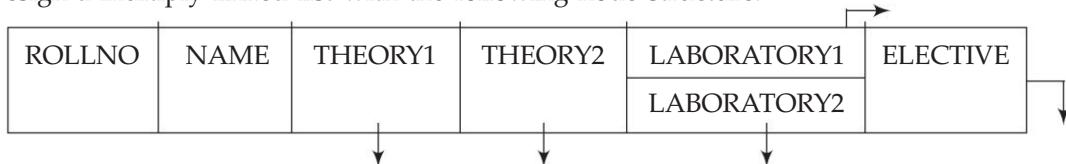
$$P_1 : 19x^6 + 78x^4 + 6x^3 - 23x^2 - 34$$

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



## Programming Assignments

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



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

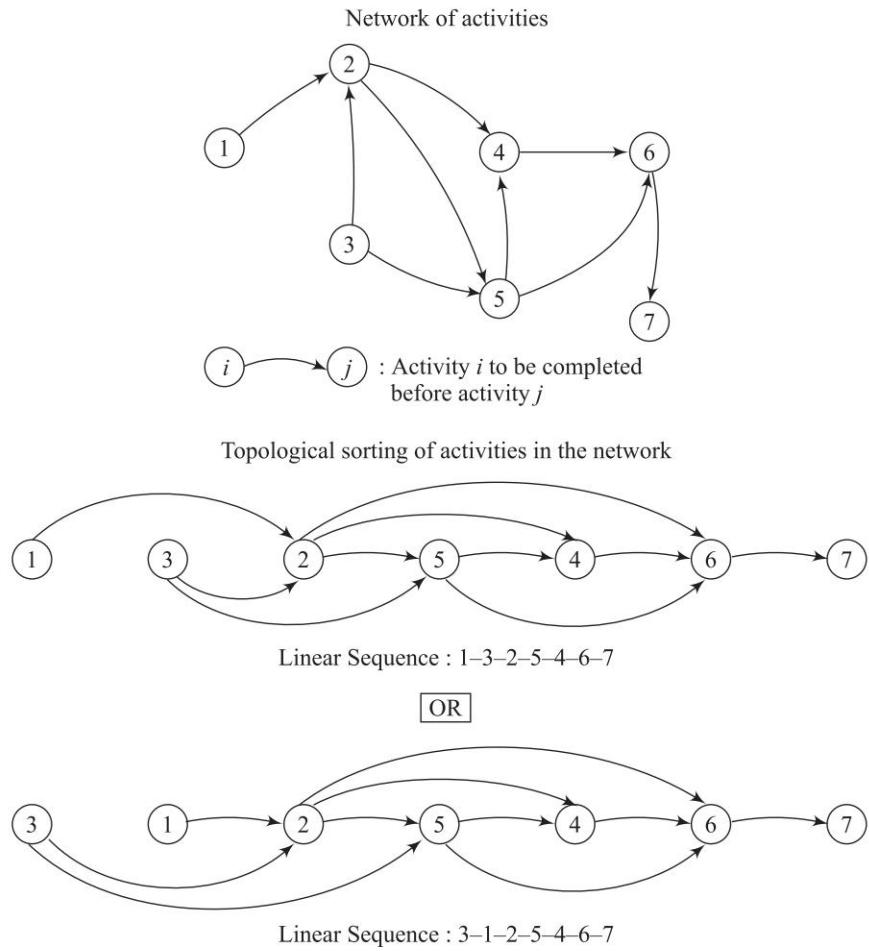
Execute a program to implement the multiply linked list with provisions to insert nodes, to update information besides generating reports as needed by the department.

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

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

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

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



**Fig. P6.5** Topological sorting of a network

## CHAPTER

# LINKED STACKS AND LINKED QUEUES

# 7



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

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

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

- 7.1 *Introduction*
- 7.2 *Operations on Linked Stacks and Linked Queues*
- 7.3 *Dynamic Memory Management and Linked Stacks*
- 7.4 *Implementation of Linked Representations*
- 7.5 *Applications*

## Introduction

## 7.1

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

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

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

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

The drawbacks of sequential representation of a queue are

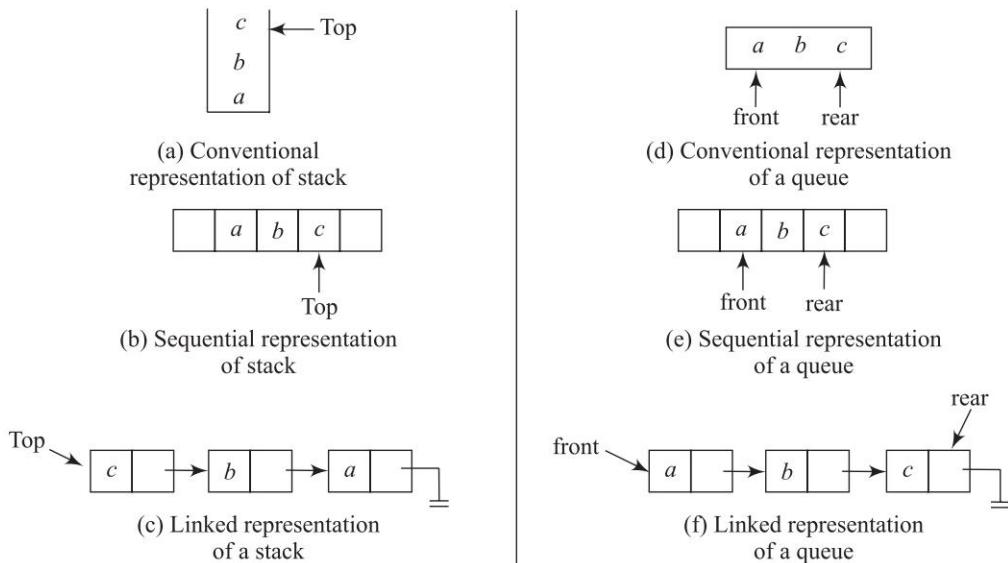
- (i) finite capacity of the queue, and

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

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

## Linked stack

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



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

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

## Linked queues

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

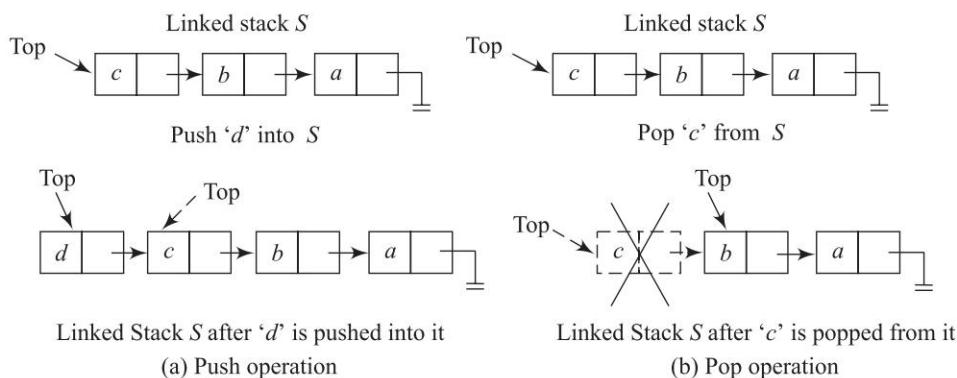
## Operations on Linked Stacks and Linked Queues

## 7.2

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

## Linked stack operations

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

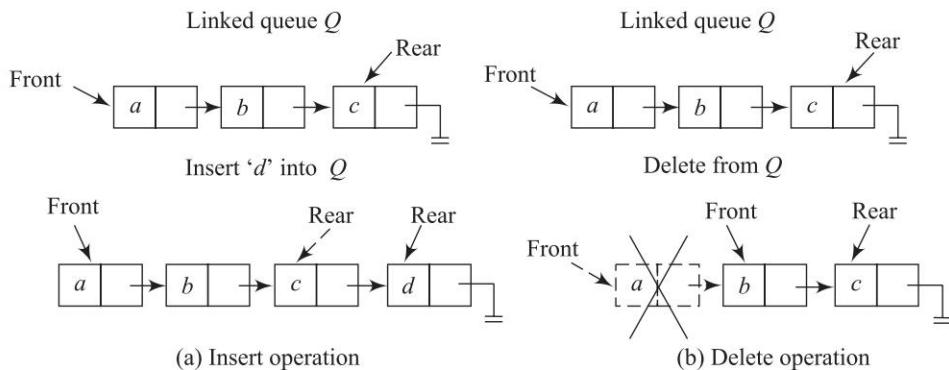


**Fig. 7.2 Push and pop operation on a linked stack  $S$**

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

## Linked queue operations

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



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

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

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

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

### Algorithms for push/pop operations on a linked stack

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

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

```

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

```

Note the absence of the STACK\_FULL condition. The time complexity of a push operation is  $O(1)$ .

**Algorithm 7.2:** Pop from a linked stack  $S$  and output the element through ITEM

```

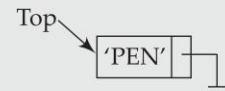
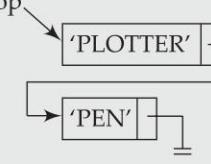
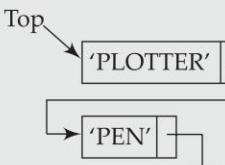
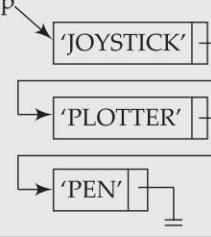
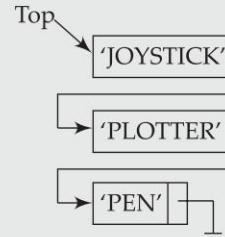
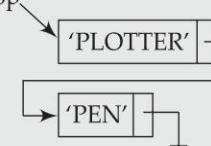
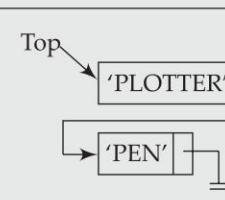
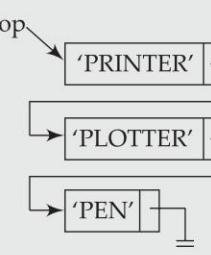
procedure POP_LINKSTACK (TOP, ITEM)
    /* pop element from stack and set ITEM to the element */
    if (TOP = 0) then call LINKSTACK_EMPTY
        /* check if linked stack is empty */
    else { TEMP = TOP
        ITEM = DATA(TOP)
        TOP = LINK(TOP)
    }
    call RETURN(TEMP) ;
end POP_LINKSTACK.

```

The time complexity of a pop operation is  $O(1)$ . Example 7.1 illustrates the push and pop operation on a linked stack.

**Example 7.1** Consider the stack DEVICE of peripheral devices illustrated in Example 4.1. We implement the same as a linked stack. The insertion of PEN, PLOTTER, JOYSTICK and PRINTER and a deletion operation are illustrated in Table 7.1. We assume the list to be initially empty and TOP to be the top pointer of the stack.

**Table 7.1** Insert and delete operations on linked stack DEVICE

| Stack Operation                | Stack DEVICE before operation                                                       | Algorithm invocation             | Stack DEVICE after operation                                                                             | Remarks                                          |
|--------------------------------|-------------------------------------------------------------------------------------|----------------------------------|----------------------------------------------------------------------------------------------------------|--------------------------------------------------|
| 1. Push 'PEN' into DEVICE      |    | PUSH_LINKSTACK (Top, 'PEN')      |                        | Set Top to point to the first node.              |
| 2. Push 'PLOTTER' into DEVICE  |    | PUSH_LINKSTACK (Top, 'PLOTTER')  |                        | Insert PLOTTER as the first node and reset Top.  |
| 3. Push 'JOYSTICK' into DEVICE |    | PUSH_LINKSTACK (Top, 'JOYSTICK') |                        | Insert JOYSTICK as the first node and reset Top. |
| 4. Pop from DEVICE             |   | Pop_LINKSTACK (Top, 'ITEM')      | <br>ITEM = 'JOYSTICK' | Return the first node and reset Top.             |
| 5. Push 'PRINTER' into DEVICE  |  | PUSH_LINKSTACK (Top, 'PRINTER')  |                      | Insert PRINTER as the first node and reset Top.  |

### Algorithms for insert and delete operations in a linked queue

Let  $Q$  be a linked queue. Algorithms 7.3 and 7.4 illustrate the insert and delete operations on the queue  $Q$ .

**Algorithm 7.3:** Push item *ITEM* into a linear queue *Q* with FRONT and REAR as the front and rear pointer to the queue

```

procedure INSERT_LINKQUEUE (FRONT, REAR, ITEM)
Call GETNODE (X);
DATA (X) = ITEM;
LINK (X) = NIL; /* Node with ITEM is ready to be inserted into Q */
if (FRONT = 0) then FRONT = REAR = X;
/* If Q is empty then ITEM is the first element in the queue
   Q */
else {LINK (REAR) = X;
       REAR = X
}
end INSERT_LINKQUEUE.
```



Observe the absence of QUEUE\_FULL condition in the insert procedure. The time complexity of an insert operation is  $O(1)$ .

**Algorithm 7.4:** Delete element from the linked queue *Q* through ITEM with FRONT and REAR as the front and rear pointers

```

procedure DELETE_LINKQUEUE (FRONT, ITEM)
if (FRONT = 0) then call LINKQUEUE_EMPTY;
/* Test condition to avoid deletion in an empty queue */
else {TEMP = FRONT;
       ITEM = DATA (TEMP);
       FRONT = LINK (TEMP);
}
call RETURN (TEMP); /* return the node TEMP to the free pool */
end DELETE_LINKQUEUE.
```



The time complexity of a delete operation is  $O(1)$ . Example 7.2 Illustrates the insert and delete operations on a linked queue.

**Example 7.2** Consider the queue BIRDS illustrated in Example 5.1. The insertion of DOVE, PEACOCK, PIGEON and SWAN, and two deletions are shown in Table 7.2.

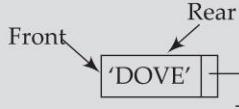
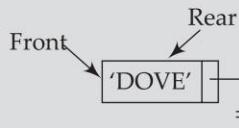
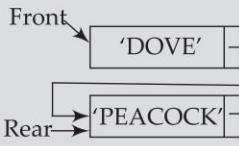
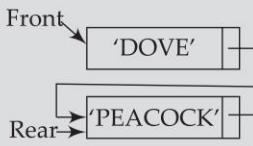
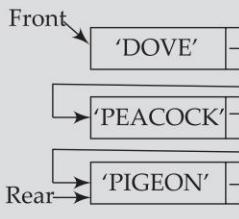
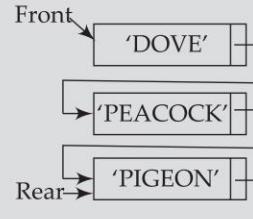
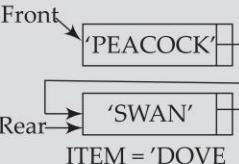
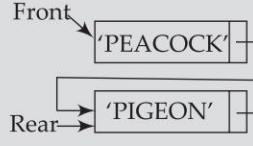
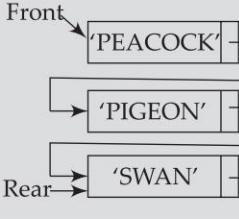
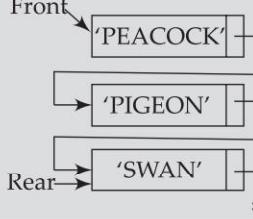
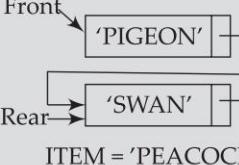
Owing to the linked representation there is no limitation on the capacity of the stack or queue. In fact, the stack or queue can hold as many elements as the storage memory can accommodate! This dispenses with the need to check for STACK\_FULL or QUEUE\_FULL conditions during push or insert operations respectively.

The merits of linked stacks and linked queues are therefore

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

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

**Table 7.2** Insert and delete operations on a linked queue BIRDS

| Linked queue                   | Linked queue before operation                                                       | Algorithm Invocation                            | Linked queue after operation                                                                             | Remarks                                                                                          |
|--------------------------------|-------------------------------------------------------------------------------------|-------------------------------------------------|----------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------|
| 1. Insert 'DOVE' into BIRDS    |    | INSERT_LINKQUEUE<br>(Front, Rear,<br>'DOVE')    |                        | Since the queue BIRDS is empty, insert DOVE as the first node. Front and Rear point to the node. |
| 2. Insert 'PEACOCK' into BIRDS |    | INSERT_LINKQUEUE<br>(Front, Rear,<br>'PEACOCK') |                        | Insert PEACOCK as the last node. Reset Rear pointer.                                             |
| 3. Insert 'PIGEON' into BIRDS  |    | INSERT_LINKQUEUE<br>(Front, Rear,<br>'PIGEON')  |                        | Insert PIGEON as the last node. Reset Rear pointer.                                              |
| 4. Delete from BIRDS           |   | DELETE_LINKQUEUE<br>(Front, Rear,<br>ITEM)      | <br>ITEM = 'DOVE'     | Delete node pointed to by Front. Reset Front.                                                    |
| 5. Insert SWAN into BIRDS      |  | INSERT_LINKQUEUE<br>(Front, Rear,,<br>SWAN)     |                      | Insert SWAN as the last node. Reset Rear.                                                        |
| 6. Delete from BIRDS           |  | DELETE_LINKQUEUE<br>(Front, Rear,<br>ITEM)      | <br>ITEM = 'PEACOCK' | Delete node pointed to by Front. Reset Front.                                                    |

## Dynamic Memory Management and Linked Stacks

## 7.3

**Dynamic memory management** deals with methods of allocating storage and recycling unused space for future use. The automatic recycling of dynamically allocated memory is also known as *Garbage collection*.

If the memory storage pool is thought of as a repository of nodes, then dynamic memory management primarily revolves round the two actions of *allocating nodes* (for use by the application) and *liberating nodes* (after their release by the application). Several intelligent strategies for the efficient allocation and liberation of nodes have been discussed in the literature. However, we choose to discuss this topic from the perspective of a linked stack application.

Every linked representation, which makes use of nodes to accommodate data elements, executes procedure `GETNODE ()` to have the desired node allocated to it, from the free storage pool and procedure `RETURN ()` to dispose or liberate the node released by it, into the storage pool. Free storage pool is also referred to as *Available Space* (`AVAIL_SPACE`).

When the application invokes `GETNODE ()`, a node from the available space data structure is deleted, to be handed over for use by the program and when `RETURN ()` is invoked, the node disposed off by the application is inserted into the available space for future use.

The most commonly used data structure for management of `AVAIL_SPACE` and its insert / delete operation is the linked stack. The list of free nodes in `AVAIL_ SPACE` are all linked together and maintained as a linked stack with a top pointer (`AV_SP`). When `GETNODE ()` is invoked, a pop operation of the linked stack is done releasing a node for use by the application and when `RETURN ()` is invoked, a push operation of the linked stack is done. Figure 7.4 illustrates the association between the `GETNODE ()` and `RETURN ()` procedures and `AVAIL_SPACE` maintained as a linked stack.

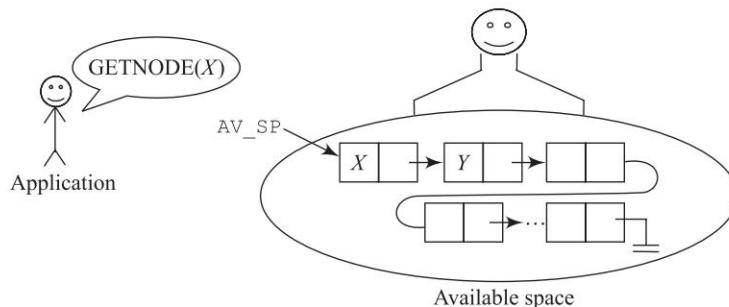
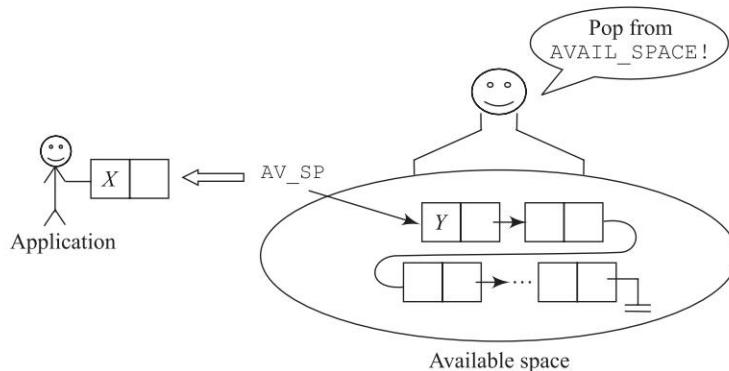
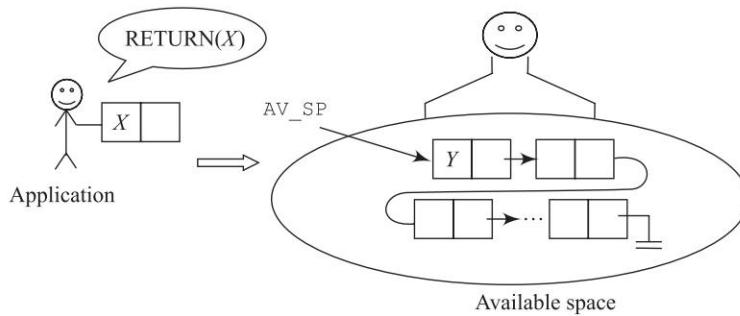
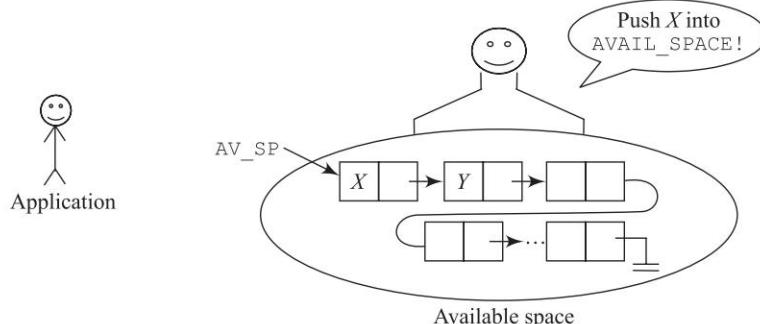
We now implement `GETNODE ()` and `RETURN ()` procedures which in fact are nothing but POP and PUSH operations on the linked stack `AVAIL_SPACE`. Algorithms 7.5 and 7.6 illustrates the implementation of the procedures.

It is obvious that at a given instance the adjacent or other nodes in the `AVAIL_ SPACE` are neighbors that are physically contiguous in the memory but lie scattered in the list. This may eventually lead to holes in the memory leading to inefficient use of memory. When variable size nodes are in use, it is desirable to compact memory so that all free nodes form a contiguous block of memory. Such a thing is termed as *memory compaction*.

It now becomes essential that the storage manager, for efficient management of memory, every time a node is returned to the free pool, ensures that the neighboring blocks of memory that are free are coalesced into a single block of memory, so as to satisfy large requests for memory. This is however easier said than done. To look for neighboring nodes which are free, a 'brute free approach' calls for a complete search through `AVAIL_ SPACE` list before collapsing the adjacent free nodes into a single block.

Allocation strategies such as Boundary Tag method and Buddy system method, with efficient reservation and liberation of nodes have been proposed in the literature.

## Linked Stacks and Linked Queues

(a) Available space before execution of `GETNODE()` procedure(b) Available space after execution of `GETNODE()` procedure(c) Available space before execution of `RETURN()` procedure(d) Available space after execution of `RETURN()` procedure**Fig. 7.4** Association between `GETNODE()`, `RETURN()` procedures and `AVAIL_SPACE`

**Algorithm 7.5:** Implementation of procedure GETNODE (X) where AV is the pointer to the linked stack implementation of AVAIL\_SPACE

```

procedure GETNODE (X)
if (AV = 0) then call NO_FREE_NODES;
    /* AVAIL_SPACE has no free nodes to allocate */
else { X = AV;
        AV = LINK (AV); /* Return the address X of the top node in
                           AVAIL_SPACE */
end GETNODE.
```

**Algorithm 7.6:** Implementation of procedure RETURN (X) where AV is the pointer to the linked stack implementation of AVAIL\_SPACE

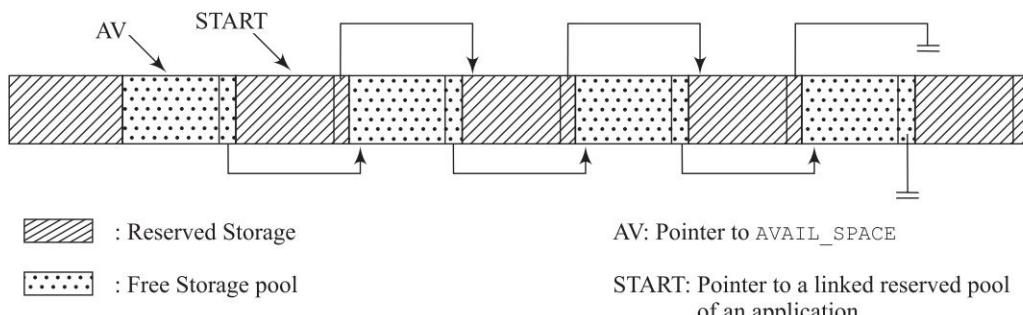
```

procedure RETURN (X)
LINK (X)= AV; /* Push node X into AVAIL_SPACE and reset AV */
AV = X;
end RETURN.
```

## Implementation of Linked Representations

## 7.4

It is emphasized here that nodes belonging to the reserved pool, that is nodes which are currently in use, coexist with the nodes of the free pool in the same storage area. It is therefore not uncommon to have a reserved node having a free node as its physically contiguous neighbor. While the link fields of the free nodes, which in its simplest form is a linked stack, keeps track of the free nodes in the list, the link fields of the reserved pool similarly keep track of the reserved nodes in the list. Figure 7.5 illustrates a simple scheme of reserved pool intertwined with the free pool in the memory storage.



**Fig. 7.5** The scheme of reserved storage pool and free storage pool in the memory storage

Example 7.3 illustrates the implementation of a linked representation. For simplicity we consider a singly linked list occupying the reserved pool.

**Example 7.3** A snap shot of the memory storage is shown in Fig. 7.6. The reserved pool accommodates a singly linked list (START). The free storage pool of used and disposed nodes is maintained as a linked stack with top pointer AV.

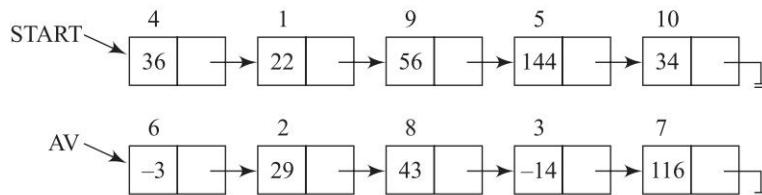
|    | DATA | Link |  |
|----|------|------|--|
| 1  | 22   | 9    |  |
| 2  | 29   | 8    |  |
| 3  | -14  | 7    |  |
| 4  | 36   | 1    |  |
| 5  | 144  | 10   |  |
| 6  | -3   | 2    |  |
| 7  | 116  | 0    |  |
| 8  | 43   | 3    |  |
| 9  | 56   | 5    |  |
| 10 | 34   | 0    |  |

AV: 6

START: 4

**Fig. 7.6** A Snapshot of the memory accommodating a singly linked list in its reserved pool and the free storage pool

Note the memory locations AV and START. AV records the address of the first node in the free storage pool and START the same of the singly linked list in the reserved pool. The logical representation of the singly linked list and the available space are illustrated in Fig. 7.7.



**Fig. 7.7** Logical representation of the singly linked list and the AVAIL\_SPACE shown in Fig. 7.6

## Applications

## 7.5

All applications of linear queues and linear stacks can be implemented as linked stacks and linked queues. In this section we discuss the following problems,

- (i) Balancing symbols,
  - (ii) Polynomial representation
- as application of linked stacks and linked queues respectively.

### Balancing symbols

An important activity performed by compilers is to check for syntax errors in the program code. One such error checking mechanism is the balancing of symbols or specifically, balancing of parentheses in the context of expressions, which is exclusive to this discussion.

For the balancing of parentheses, the left parentheses or braces or brackets as allowed by the language syntax, must have closing or matching right parentheses, braces or brackets respectively. Thus the usage of ( ), or { } or [ ] are correct whereas, (, [, } are incorrect, the former indicative of a balanced occurrence and the latter of an imbalanced occurrence in an expression.

The arithmetic expressions shown in Example 7.4 are balanced in parentheses while those listed in Example 7.5 are imbalanced forcing the compiler to report errors.

**Example 7.4** Balanced arithmetic expressions

- (i)  $((A + B) \uparrow C - D) + E - F$
- (ii)  $( - (A + B) * (C - D)) \uparrow F$

**Example 7.5** Imbalanced arithmetic expressions

- (i)  $(A + B)^* - (C + D + F)$
- (ii)  $-((A + B + C)^* - (E + F))$

The solution to the problem is an easy but elegant use of a stack to check for mismatched parentheses. The general pseudocode procedure for the problem is illustrated in Algorithm 7.5. Appropriate to the discussion, we choose a linked representation for the stack in the algorithm. Examples 7.6 and 7.7 illustrate the working of the algorithm on two expressions with balanced and unbalanced symbols respectively.

**Algorithm 7.5:** To check for the balancing of parentheses in a string procedure  
BALANCE\_EXPR( $E$ )

```
/* $E$  is the expression padded with a $ to indicate end of input*/
clear stack;
while not end_of_string( $E$ )
    read character; /* read a character from string  $E$  */
    if the character is an open symbol then push character in to stack;
    if the character is a close symbol then
        if stack is empty then ERROR( )
        else {pop the stack;
            if the character popped is
                not the matching symbol
            then ERROR( );
        }
    endwhile
    if stack not empty then ERROR();
end BALANCE_EXPR.
```

**Example 7.6** Consider the arithmetic expression  $((A+B)^* C) - D$  which has balanced parentheses. Table 7.3 illustrates the working of the algorithm on the expression.

**Example 7.7** Consider the expression  $((A+B)^* C + G$  which has unbalanced parentheses. Table 7.4 illustrates the working of the algorithm on the expression.

## Polynomial representation

In Chapter 6, Sec. 6 we had discussed the problem of addition of polynomials as an application of linked lists. In this section, we highlight the representation of polynomials as an application of linear queues.

Consider a polynomial  $9x^6 - 2x^4 + 3x^2 + 4$ . Adopting the node structure shown in Fig. 7.8(a) (reproduction of Fig. 6.26(a)) the linked list for the polynomial is as shown in Fig. 7.8(b).

**Table 7.3** Working of Algorithm BALANCE\_EXPR ( ) on the expression  $((A + B)^* C) - D$

| Input string (E)       | Stack (S)                                                                                                               | Remarks                                                         |
|------------------------|-------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------|
| $((A + B) * C) - D \$$ | S →                                                                                                                     | Initialization. Note E is padded with \$ as end of input symbol |
| $(A + B) * C) - D \$$  | S →    | Push '(' into S                                                 |
| $A + B) * C) - D \$$   | S →    | Push '(' into S                                                 |
| $+ B) * C) - D \$$     | S →    | Ignore character 'A'                                            |
| $B) * C) - D \$$       | S →    | Ignore character '+'                                            |
| $) * C) - D \$$        | S →    | Ignore character 'B'                                            |
| $* C) - D \$$          | S →   | Pop symbol from S.<br>Matching symbol to ")" found.<br>Proceed. |
| $C) - D \$$            | S →  | Ignore character '*'                                            |
| $) - D \$$             | S →  | Ignore character 'C'                                            |
| $- D \$$               | S →                                                                                                                     | Pop symbol from S.<br>Matching symbol to ')' found.<br>Proceed. |
| $D \$$                 | S →                                                                                                                     | Ignore character '-'                                            |
| \$                     | S →                                                                                                                     | Ignore character 'D'                                            |
| \$                     | S →                                                                                                                     | End of input encountered.<br>Stack is empty.<br>Success.        |

**Table 7.4** Working of the algorithm BALANCE\_EXPR ( ) on the expression  $((A+B) * C \uparrow G$ 

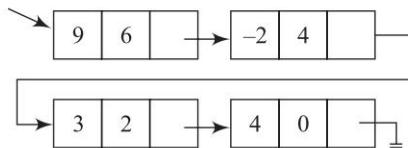
| Input string (E)                  | Stack (S)                                                                                                               | Remarks                                                         |
|-----------------------------------|-------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------|
| $((A + B) * C \uparrow G \$$<br>↑ | S →                                                                                                                     | Initialization.<br>E is padded with \$ as end of input symbol.  |
| $(A + B) * C \uparrow G \$$<br>↑  | S →    | Push '(' into S                                                 |
| $A + B) * C \uparrow G \$$<br>↑   | S →    | Push '(' into S                                                 |
| $+ B) * C \uparrow G \$$<br>↑     | S →    | Ignore character 'A'                                            |
| $B) * C \uparrow G \$$<br>↑       | S →    | Ignore character '+'                                            |
| $) * C \uparrow G \$$<br>↑        | S →    | Ignore character 'B'                                            |
| $* C \uparrow G \$$<br>↑          | S →    | Pop symbol from S.<br>Matching symbol to ")" found.<br>Proceed. |
| $C \uparrow G \$$<br>↑            | S →                                                                                                                     | Ignore character '*'                                            |
| $\uparrow G \$$<br>↑              | S →    | Ignore character 'C'                                            |
| $G \$$<br>↑                       | S →  | Ignore character '↑ '                                           |
| \$                                | S →  | Ignore character 'G'                                            |
| \$                                | S →  | End of input encountered.<br>Stack is not empty.<br>Error.      |

For easy manipulation of the linked list, we represent the polynomial in its decreasing order of exponents of the variable (in the case of uni-variable polynomials). It would therefore be easy for the function handling the reading of the polynomial to implement the linked list as a linear queue, since this would entail an elegant construction of the list from the symbolic representation of the polynomial, by enqueueing the linear queue with the next highest exponent term. The linear queue representation for the polynomial  $9x^6 - 2x^4 + 3x^2 + 4$  is shown in Fig. 7.9.

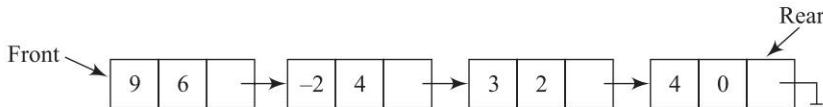


COEFF: Coefficient of the term  
EXP: Exponent of the variable

(a) Node structure

(b) Linked list representation of the polynomial  $9x^6 - 2x^4 + 3x^2 + 4$ **Fig. 7.8** Linked list representation of a polynomial

Also, after the manipulation of the polynomials (addition, subtraction etc.) the resulting polynomial could also be elegantly represented as a linear queue. This merely calls for enqueueing the linear queue with the just manipulated term. Recall the problem of addition of polynomials discussed in Sec. 6.6. Maintaining the added polynomial as a linear queue would only call for ‘appending’ the added terms (coefficients of terms with like exponents) to the rear of the list. However, during the manipulation, the linear queue representation of the polynomials are to be treated as traversable queues. A *traversable queue* while retaining the operations of enqueueing and dequeuing, permits traversal of the list in which nodes may be examined.

**Fig. 7.9** Linear queue representation of the polynomial  $9x^6 - 2x^4 + 3x^2 + 4$ 

## Summary

- Sequential representation of stacks and queues suffer from the limitation of finite capacity besides checking for the STACK\_FULL and QUEUE\_FULL conditions, each time a push or insert operation is executed respectively.
- Linked stacks and linked queues are singly linked list implementation of stacks and queues, though a circularly linked list representation can also be attempted without hampering the LIFO or FIFO principle of the respective data structures.
- Linked stacks and linked queues display the merits of conceptual and computational simplicity of insert and delete operations besides absence of limited capacity. However, the requirement of additional space to accommodate the link fields can be viewed as a demerit.
- The maintenance of available space list calls for the application of linked stacks.
- The problems of balancing of symbols and polynomial representation have been discussed to demonstrate the application of linked stack and linked queue respectively.



## Illustrative Problems

**Problem 7.1** Given the following memory snap shot where START and AV\_SP store the start pointers of the linked list and the available space respectively,

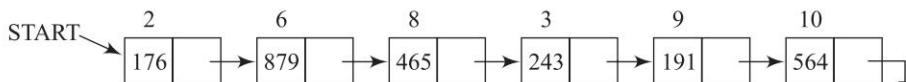
- Identify the linked list
- Show how the linked list and the available space list are affected when the following operations are carried out:

- Insert 116 at the end of the list
  - Delete 243
  - Obtain the memory snap shot after the execution of operations listed in (a) and (b)
- DATA LINK

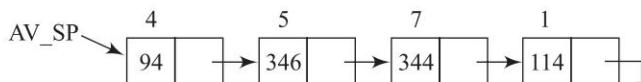
|    |     |    |
|----|-----|----|
| 1  | 114 | 0  |
| 2  | 176 | 6  |
| 3  | 243 | 9  |
| 4  | 94  | 5  |
| 5  | 346 | 7  |
| 6  | 879 | 8  |
| 7  | 344 | 1  |
| 8  | 465 | 3  |
| 9  | 191 | 10 |
| 10 | 564 | 0  |

START: 2 AV\_SP: 4

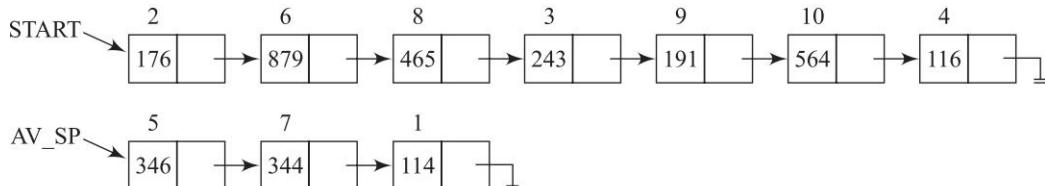
**Solution:** (i) Since the linked list starts at node whose address is 2, the logical representation of the list is as given below:



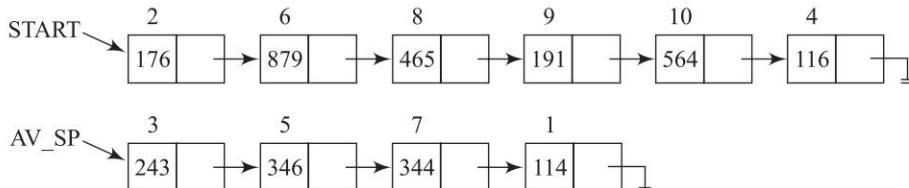
The available space list which functions as a linked stack and starts from node whose address is 4, is given by:



- (a) To insert 116 at the end of the list START, we get a node from the available space list (invoke GETNODE ( )). The node released has address 4. The resultant list and the available space list are as follows



- (b) To delete 243, the node holding the element has to be returned to the available space list (invoke (RETURN ( ) )). The resultant list and the available space list are as



(c) The memory snapshot after the execution of (a) and (b) is as given below:

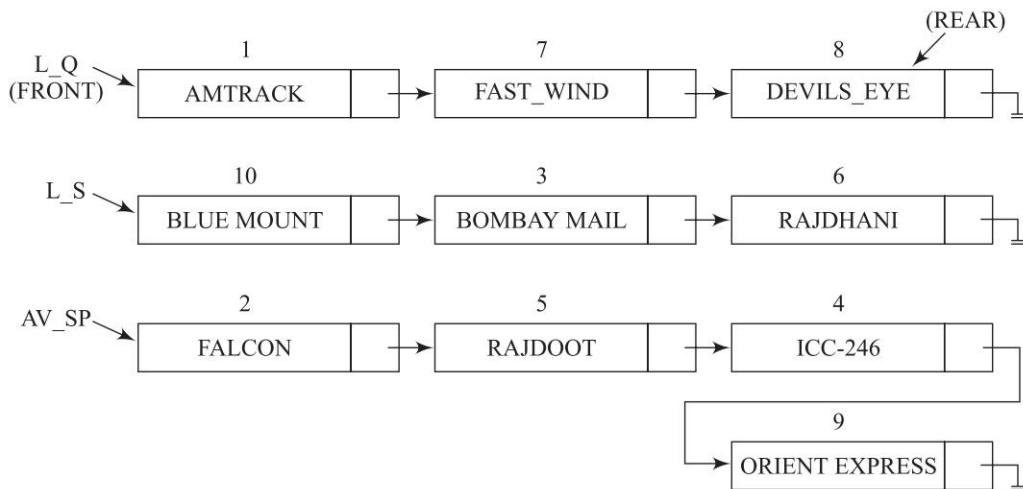
|    | DATA | LINK |                                       |                                       |
|----|------|------|---------------------------------------|---------------------------------------|
| 1  | 114  | 0    | START: <input type="text" value="2"/> | AV_SP: <input type="text" value="3"/> |
| 2  | 176  | 6    |                                       |                                       |
| 3  | 243  | 5    |                                       |                                       |
| 4  | 116  | 0    |                                       |                                       |
| 5  | 346  | 7    |                                       |                                       |
| 6  | 879  | 8    |                                       |                                       |
| 7  | 344  | 1    |                                       |                                       |
| 8  | 465  | 9    |                                       |                                       |
| 9  | 191  | 10   |                                       |                                       |
| 10 | 564  | 4    |                                       |                                       |

**Problem 7.2** Given the following memory snapshot which stores a linked stack L\_S and a linked queue L\_Q beginning at the respective addresses, obtain the resulting memory snapshot after the following operations are carried out sequentially.

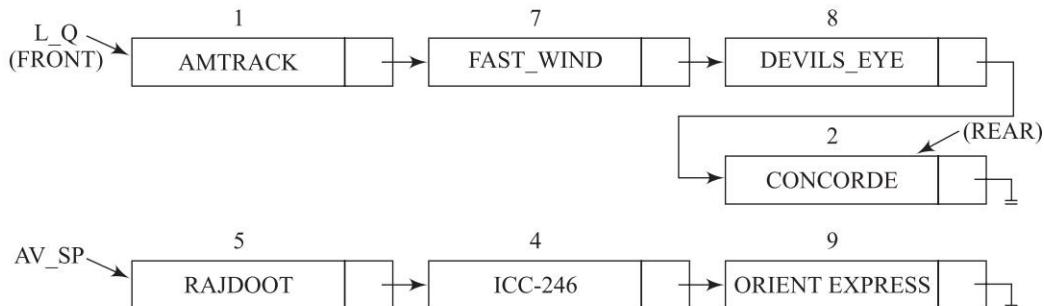
- (i) Enqueue CONCORDE into L\_Q.
- (ii) Pop from L\_S
- (iii) Dequeue from L\_Q
- (iv) Push PALACE\_ON\_WHEELS into L\_S

|    | DATA           | LINK |                                     |                                                                            |
|----|----------------|------|-------------------------------------|----------------------------------------------------------------------------|
| 1  | AMTRACK        | 7    | L_Q: <input type="text" value="1"/> | L_S: <input type="text" value="10"/> AV_SP: <input type="text" value="2"/> |
| 2  | FALCON         | 5    | (FRONT)                             |                                                                            |
| 3  | BOMBAY_MAIL    | 6    | L_Q <input type="text" value="8"/>  |                                                                            |
| 4  | ICC 246        | 9    | (REAR)                              |                                                                            |
| 5  | RAJDOOT        | 4    |                                     |                                                                            |
| 6  | RAJDHANI       | 0    |                                     |                                                                            |
| 7  | FAST_WIND      | 8    |                                     |                                                                            |
| 8  | DEVILS_EYE     | 0    |                                     |                                                                            |
| 9  | ORIENT EXPRESS | 0    |                                     |                                                                            |
| 10 | BLUE MOUNT     | 3    |                                     |                                                                            |

**Solution:** It is easier to perform the operations on the logical representations of the lists and available space extracted from the memory, before obtaining the final memory snapshot. The lists are:

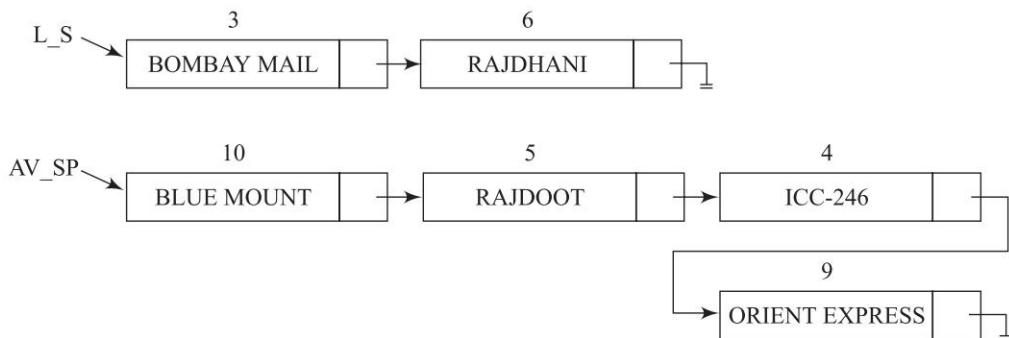


- (i) Enqueue CONCORDE into L\_Q yields:



Here node 2 is popped from AV\_SP to accommodate CONCORDE which is inserted at the rear of L\_Q.

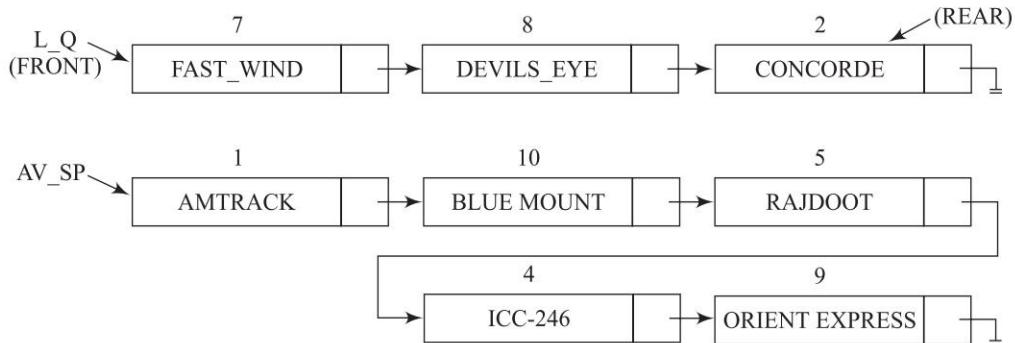
- (ii) Pop from L\_S yields:



Here node 10 from L\_S is deleted and pushed into AV\_SP.

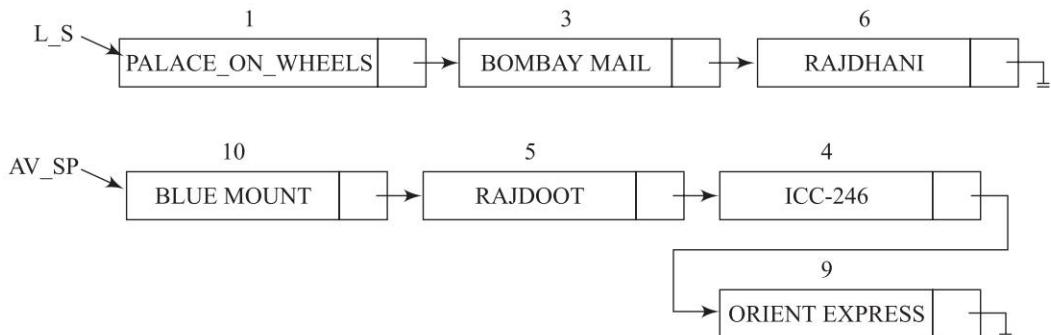
## Linked Stacks and Linked Queues

(iii) Dequeue from L\_Q yields:



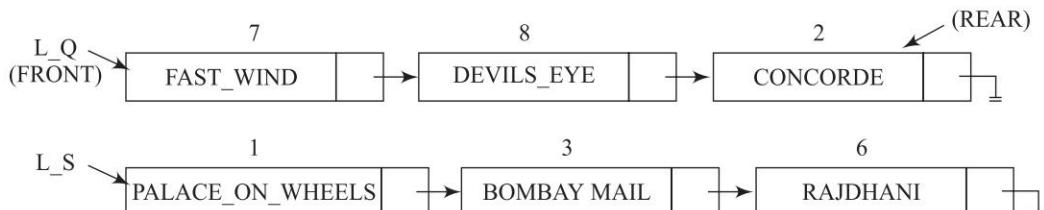
Here, node 1 from  $L_Q$  is deleted and pushed into  $AV\_SP$ .

(iv) Push "PALACE\_ON\_WHEELS" into  $L_S$  yields:



Here, node 1 from  $AV\_SP$  is popped to accommodate "PALACE\_ON\_WHEELS" before pushing the node into  $L_S$ .

The final lists are



The memory snapshot is given by:

|   | DATA             | LINK |                  |                      |
|---|------------------|------|------------------|----------------------|
| 1 | PALACE ON WHEELS | 3    | $L_Q: \boxed{7}$ | $L_S: \boxed{1}$     |
| 2 | CONCORDE         | 0    | (FRONT)          | $AV\_SP: \boxed{10}$ |
| 3 | BOMBAY MAIL      | 6    | $L_Q \boxed{2}$  |                      |
| 4 | ICC-246          | 9    | (REAR)           |                      |

|    |                |   |
|----|----------------|---|
| 5  | RAJDOOT        | 4 |
| 6  | RAJDHANI       | 0 |
| 7  | FAST_WIND      | 8 |
| 8  | DEVILS_EYE     | 2 |
| 9  | ORIENT EXPRESS | 0 |
| 10 | BLUE MOUNT     | 5 |

**Problem 7.3** Implement an abstract data type STAQUE which is a combination of a linked stack and a linked queue. Develop procedures to perform an Insert and a Delete operation, termed PUSHINS and POPDEL respectively, on a non empty STAQUE. PUSHINS inserts an element at the top or rear of the STAQUE based on an indication given to the procedure and POPDEL deletes elements from the top or front of the list.

**Solution:** The procedure PUSHINS performs the insertion of an element in the top or rear of the list based on whether the STAQUE is viewed as a stack or queue respectively. On the other hand, the procedure POPDEL which performs a pop or deletion of element, is common to a STAQUE, since in both the cases first element in the list alone is deleted.

```

procedure PUSHINS(WHERE, TOP, REAR, ITEM)
/* WHERE indicates whether the insertion of ITEM
is to be done as on a stack or as on a queue*/
Call GETNODE (X);
DATA (X) = ITEM;
if (WHERE = 'Stack') then {LINK (X) = TOP;
                           TOP = X;
                           }
else
                           {LINK (REAR) = X;
                            LINK (X) = Nil;
                            REAR = X;
                           }
end PUSHINS

```

```

procedure POPDEL (TOP, ITEM)
TEMP = TOP;
ITEM = DATA (TEMP); /* delete top
element of the list through
ITEM*/
TOP=LINK (TEMP);
RETURN (TEMP);
end POPDEL.

```

**Problem 7.4** Write a procedure to convert a linked stack into a linked queue.

**Solution:** An elegant and an easy solution to the problem is to undertake the conversion by returning the addresses of the first and last nodes of the linked stack as FRONT and REAR thereby turning the linked stack into a linked queue.

```

procedure CONVERT_LINKSTACK(TOP, FRONT, REAR)
/* FRONT and REAR are the variables which return the addresses of
the first and last node of the list converting the linked stack into
a linked queue*/

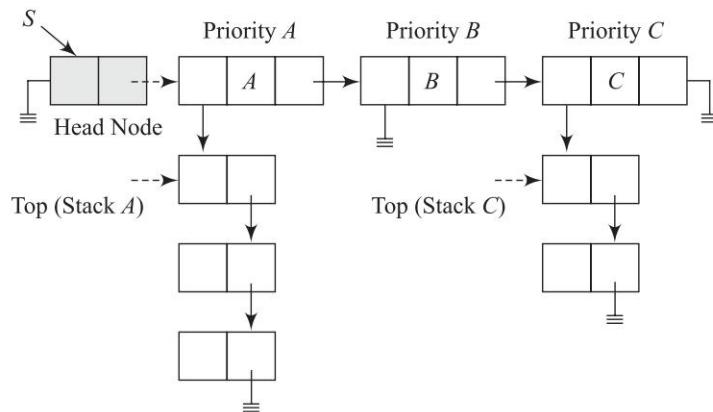
```

```

if (TOP = Nil) then print ("Conversion not possible");
else {FRONT=TOP;
        TEMP=TOP;
        while (LINK (TEMP) procedure ^ Nil)
            TEMP=LINK (TEMP);
            REAR=TEMP;
        endwhile
    }
end CONVERT_LINKSTACK.

```

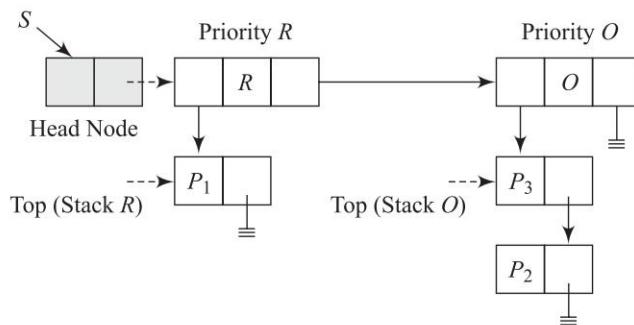
**Problem 7.5** An Abstract Data Type STACKLIST is a list of linked stacks stored according to a priority factor viz., A, B, C etc, where A means highest priority, B the next and so on. Elements having the same priority are stored as a linked stack. The following is a structure of the STACKLIST S



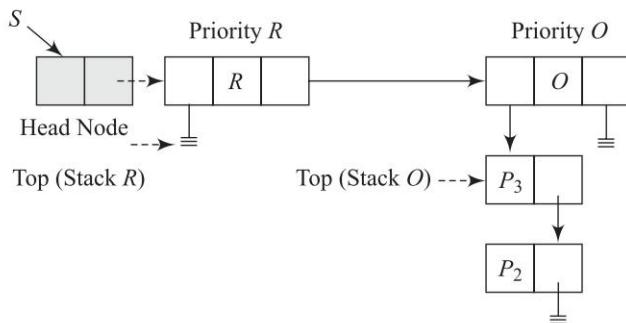
Create a STACKLIST for the following application of Process Scheduling with the processes having two priorities, viz., R (Real time) and O(Online) listed within brackets.

|                                        |                                        |
|----------------------------------------|----------------------------------------|
| 1. Initiate Process $P_1$ ( R )        | 5. Initiate Process $P_5$ ( O )        |
| 2. Initiate Process $P_2$ ( O )        | 6. Initiate Process $P_6$ ( R )        |
| 3. Initiate Process $P_3$ ( O )        | 7. Terminate Process in Linked Stack O |
| 4. Terminate Process in Linked Stack R | 8. Initiate Process $P_7$ ( R )        |

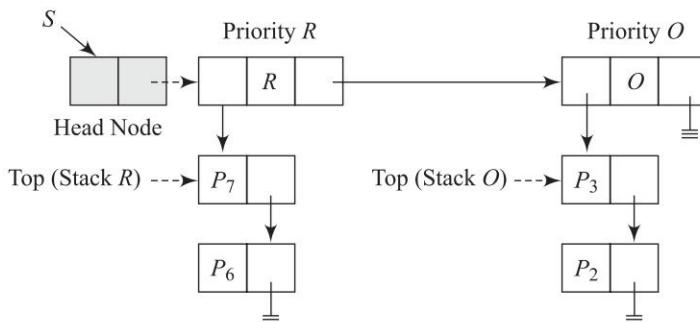
**Solution:** The STACKLIST at the end of Schedules 1-3 is shown as follows



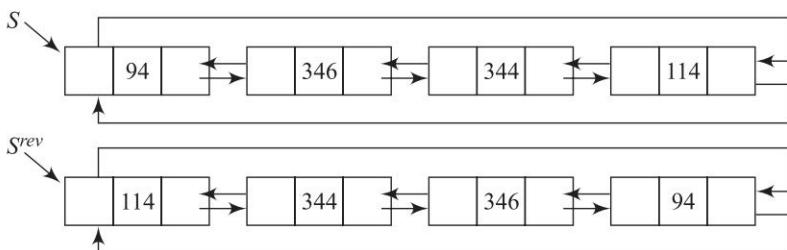
The STACKLIST at the end of Schedule 4 is as given below:



The STACKLIST at the end of Schedule 5-8 is as shown below:



**Problem 7.6** Write a procedure to reverse a linked stack implemented as a doubly linked list, with the original top and bottom positions of the stack reversed as bottom and top respectively. For example, a linked stack *S* and its reversed version *S*<sup>rev</sup> are shown below:

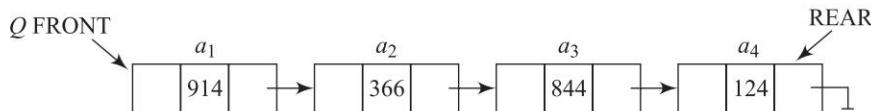


**Solution:** An elegant solution would be to merely swap the LLINK and RLINK pointers of each of the doubly linked list to reverse the list and remember the address of the last node in the original stack  $S$  as the TOP pointer. The procedure is given below:

```

procedure REVERSE_STACK(TOP)
/* TEMP and HOLD are temporary variables to hold the addresses of nodes*/
TEMP=TOP;
Repeat
    HOLD=LLINK(TEMP);
    LLINK(TEMP)=RLINK(TEMP);
    RLINK(TEMP)=HOLD;           /* Swap left and right links for each node*/
    TEMP=LLINK(TEMP);          /* Move to the next node*/
until (TEMP=TOP)
TOP=RLINK(TEMP);
end REVERSE_STACK.
```

**Problem 7.7** What does the following pseudocode do to the linked queue  $Q$  with the addresses of nodes, as shown below:

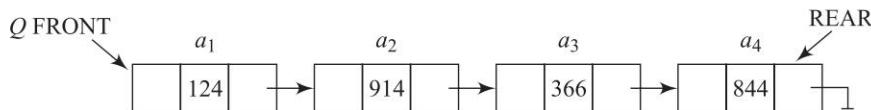


```

procedure WHAT_DO_I_DO(FRONT, REAR)
/* HAVE, HOLD and HUG are temporary variables to hold the link or data
fields of the nodes as the case may be*/
HAVE=FRONT;
HOLD=DATA(HAVE);

while LINK(HAVE) ≠ Nil
HUG= DATA(LINK(HAVE));
DATA(LINK(HAVE))=HOLD;
HOLD=HUG;
HAVE=LINK(HAVE);
endwhile
DATA(FRONT)=HOLD;
end WHAT_DO_I_DO
```

**Solution:** The procedure WHAT\_DO\_I\_DO rotates the data items of linked queue  $Q$  to obtain the resultant list given below:



**Problem 7.8** Write a procedure to remove the  $n^{\text{th}}$  element (from the top) of a linked stack with the rest of the elements unchanged. Contrast this with a sequential stack implementation for the same problem (Refer Illustrative Problem 4.2 (iii) of Chapter 4).

**Solution:** To remove the  $n^{\text{th}}$  element leaving the other elements unchanged, a linked implementation of the stack merely calls for sliding down the list which is easily done, and for a reset of a link to remove the node concerned. The procedure is given below. In contrast, a sequential implementation as illustrated in Illustrative Problem 4.2(iii), calls for the use of another temporary stack to hold the elements popped out from the original stack before pushing them back into it.

```

procedure REMOVE(TOP, ITEM, n)
/* The  $n^{\text{th}}$  element is removed through ITEM*/
TEMP=TOP;
COUNT=1;
while (COUNT  $\neq$  n) do
PREVIOUS=TEMP;
TEMP = LINK (TEMP);
COUNT=COUNT+1;
endwhile
LINK(PREVIOUS) =LINK(TEMP);
ITEM=DATA(TEMP);
RETURN (ITEM);
end REMOVE

```

**Problem 7.9** Given a linked stack L\_S and a linked queue L\_Q with equal lengths, what do the following procedures to do the lists? Here TOP is the top pointer of L\_S and FRONT and REAR are the front and rear of L\_Q. What are your observations regarding the functionality of the two procedures?

```

Procedure WHAT_IS_COOKING1(TOP,
FRONT, REAR)
/* TEMP, TEMP1, TEMP2 and TEMP3 are
temporary variables*/
TEMP1= FRONT;
TEMP2=TOP;
while (TEMP1 $\neq$  Nil AND TEMP2 $\neq$  Nil) do
TEMP3=DATA(FRONT);
DATA(FRONT)=DATA(TOP);
DATA(TOP)=TEMP3;
TEMP1=LINK(TEMP1);
PREVIOUS=TEMP2;
TEMP2= LINK(TEMP2);
endwhile
TEMP=TOP;
TOP=FRONT;
FRONT=TEMP;
REAR=PREVIOUS;
end WHAT_IS_COOKING 1

```

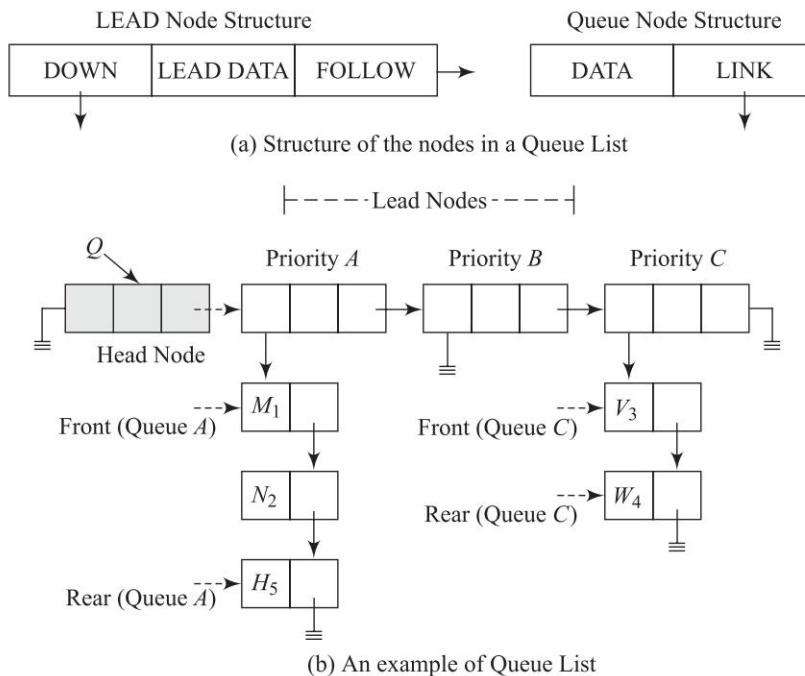
```

procedure WHAT_IS_COOKING2(TOP,
FRONT, REAR)
/* TEMP, TEMP1, TEMP2 and TEMP3 are
temporary variables*/
TEMP= TOP;
while (LINK(TEMP) $\neq$  Nil) do
TEMP=LINK(TEMP);
endwhile
TEMP1=TOP;
REAR=TEMP;
TOP=FRONT;
FRONT=TEMP1;
end WHAT_IS_COOKING2

```

**Solution:** Both the procedures swap the contents of the Linked stack L\_S and linked queue L\_Q. While WHAT\_IS\_COOKING1 does by exchanging the data items of the lists, WHAT\_IS\_COOKING2 does it by merely manipulating the pointers and hence is an elegant presentation.

**Problem 7.10** A Queue List  $Q$  is a list of linked queues stored according to orders of priority viz.,  $A, B, C$ , etc., with  $A$  accorded the highest priority and so on. The LEAD nodes serve as head nodes for each of the priority based queues. Elements with the same priority are stored as a normal linked queue. Figure I 7.10 (a-b) illustrate the node structure and an example of Queue List respectively.



|    | DOWN | LEAD DATA | FOLLOW |
|----|------|-----------|--------|
| 10 | 604  | 5         | 7      |
| 11 | 26   | -4        | 561    |
| 12 | 566  | 1         | 13     |
| 13 | 0    | 2         | 15     |
| 14 | 3    | 4         | 591    |
| 15 | 573  | 3         | 0      |
| 16 | 0    | -3        | 12     |

|      |      |
|------|------|
| DATA | LINK |
| $g$  | 0    |
| $k$  | 571  |
| $a$  | 572  |
| $l$  | 384  |
| $v$  | 0    |
| $m$  | 570  |
| $n$  | 0    |
| $u$  | 568  |
| $x$  | 564  |
| $h$  | 565  |

START  
16

AVAILABLE  
SPACE 10

(c) A snap shot of a Queue List

**Fig. I 7.10**

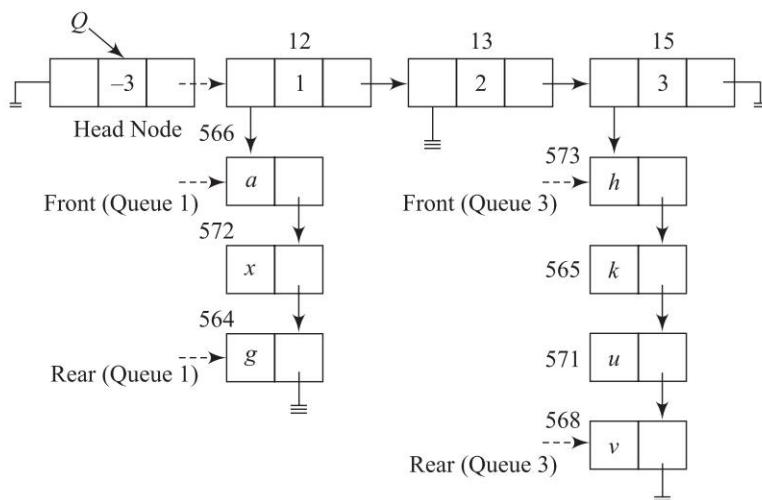
The FOLLOW link links together the head nodes of the queues and DOWN link connects it to the first node in the respective queue. The LEAD DATA field may be used to store the priority factor of the queue.

Here is a QUEUELIST  $Q$  stored in the memory, a snapshot of which is shown in Fig. I 7.10(c)

There are three queues  $Q_1, Q_2, Q_3$  with priorities 1, 2, and 3. The Head node of QUEUELIST stores the number of queues in the list as a negative number. The LEAD DATA field stores the priority factor of each of the three queues. START points to the head node of the QUEUELIST and AVAILABLE SPACE the pointer to the free storage pool.

Obtain the QUEUELIST by tracing the lead nodes and nodes of the linked queues.

**Solution:** The structure of the QUEUELIST is as shown below:



## Review Questions

The following is a snap shot of a memory which stores a linked stack VEGETABLES and a linked queue FRUITS beginning at the respective addresses. Answer the following questions with regard to operations on the linked stack and queue, each of which is assumed to be independently performed on the original linked stack and queue.

|    | DATA     | LINK |
|----|----------|------|
| 1  | CABBAGE  | 7    |
| 2  | CUCUMBER | 5    |
| 3  | PEAR     | 6    |
| 4  | ONION    | 9    |
| 5  | ORANGE   | 4    |
| 6  | PEACH    | 0    |
| 7  | CELERY   | 8    |
| 8  | CARROTS  | 0    |
| 9  | LEMON    | 0    |
| 10 | PLUM     | 3    |

| FRUITS           | VEGETABLES | AV_SP |
|------------------|------------|-------|
| FRONT<br>10<br>6 | TOP<br>1   | 2     |
| REAR:            |            |       |

1. Inserting PAPAYA into the linked queue FRUITS results in the following changes to the FRONT, REAR and AV\_SP pointers respectively, as given in:  
 (a) 10 2 2                    (b) 2 6 2                    (c) 2 6 5                    (d) 10 2 5
2. Undertaking pop operation on VEGETABLES results in the following changes to the TOP and AV\_SP pointers respectively, as given in:  
 (a) 7 1                    (b) 7 2                    (c) 8 2                    (d) 8 1
3. Undertaking delete operation on FRUITS results in the following changes to the FRONT, REAR and AV\_SP pointers respectively, as given in :  
 (a) 3 6 2                    (b) 10 3 6                    (c) 3 6 10                    (d) 10 3 2
4. Pushing TURNIPS into VEGETABLES results in the following changes to the TOP and AV\_SP pointers respectively, as given in:  
 (a) 2 5                    (b) 2 9                    (c) 1 5                    (d) 1 9
5. After the push operation of TURNIPS into VEGETABLES (undertaken in Review Question 7.4), DATA( 2 ) = ----- and DATA (LINK (2)) = -----  
 (a) TURNIPS and CABBAGE                    (b) CUCUMBER and CABBAGE  
 (c) TURNIPS and CUCUMBER                    (d) CUCUMBER and ORANGE
6. What are the merits of linked stacks and queues over their sequential counterparts?
7. How is the memory storage pool associated with a linked stack data structure for its operations?
8. How are push and pop operations implemented on a linked stack?
9. What are traversable queues?
10. Outline the node structure and a linked queue to represent the polynomial:  

$$17x^5 + 18x^2 + 9x + 89$$
11. Trace Algorithm 7.5 on the following expression to check whether parentheses are balanced:  

$$((X + Y + Z) * H) + (D * T) - 2$$



## Programming Assignments

1. Execute a program to implement a linked stack to check for the balancing of the following pairs of symbols in a Pascal program. The name of the source Pascal program is the sole input to the program.  
 Symbols: *begin end , ( ), [ ], { }.*  
 (i) Output errors encountered during mismatch of symbols.  
 (ii) Modify the program to set right the errors.
2. Evaluate a postfix expression using a linked stack implementation.
3. Implement the simulation of a time sharing system discussed in Chapter 5, Sec. 5.5, using linked queues.
4. Develop a program to implement a Queue List (Refer Illustrative Problem 7.10) which is a list of linked queues stored according to an order of priority.  
 Test for the insertion and deletion of the following jobs with their priorities listed within brackets, on a Queue List JOB\_MANAGER with three queues A, B and C listed according to their order of priorities:

|    |                      |     |                      |
|----|----------------------|-----|----------------------|
| 1. | Insert Job $J_1$ (A) | 6.  | Insert Job $J_5$ (C) |
| 2. | Insert Job $J_2$ (B) | 7.  | Insert Job $J_6$ (C) |
| 3. | Insert Job $J_3$ (A) | 8.  | Insert Job $J_7$ (A) |
| 4. | Insert Job $J_4$ (B) | 9.  | Delete Queue C       |
| 5. | Delete Queue B       | 10. | Insert Job $J_8$ (A) |

5. Develop a program to simulate a calculator which performs the addition, subtraction, multiplication and division of polynomials.

## CHAPTER



# TREES AND BINARY TREES

# 8

## Introduction

## 8.1

In Chapters 3–5 we discussed the sequential data structures of arrays, stacks and queues. These are termed as *linear data structures* as they are inherently uni-dimensional in structure. In other words, the items form a sequence or a linear list. In contrast, the data structures of trees and graphs are termed *non linear data structures* as they are inherently two dimensional in structure. Trees and their variants, binary trees and graphs, have emerged as truly powerful data structures registering immense contribution to the development of efficient algorithms or efficient solutions to various problems in science and engineering.

In this chapter, we first discuss the tree data structure, the basic terminologies and representation schemes. An important variant of the tree viz., binary tree, its basic concepts, representation schemes and traversals are elaborately discussed next. A useful modification to the binary tree viz., threaded binary tree is introduced. Finally, expression trees and its related concepts are discussed as an application of binary trees.

## Trees: Definition and Basic Terminologies

## 8.2

### Definition of trees

A *tree* is defined as a finite set of one or more nodes such that

- (i) there is a specially designated *node* called the *root* and
- (ii) the rest of the nodes could be partitioned into  $t$  disjoint sets ( $t \geq 0$ ) each set representing a tree  $T_i$ ,  $i = 1, 2, \dots, t$  known as *subtree* of the tree.

A *node* in the definition of the tree represents an item of information, and the links between the nodes termed as *branches*, represent an association between the items of information. Figure 8.1 illustrates a tree.

The definition of the tree emphasizes on the aspect of (i) *connectedness* and (ii) absence of closed loops or what are termed *cycles*. Beginning from the root node, the structure of the tree

8.1 *Introduction*

8.2 *Trees: Definition and Basic Terminologies*

8.3 *Representation of Trees*

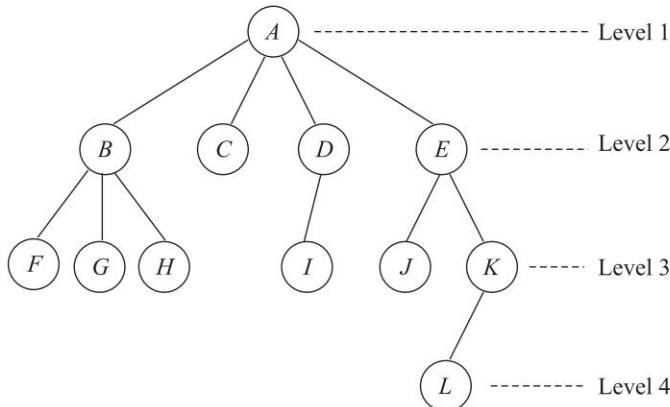
8.4 *Binary Trees: Basic Terminologies and Types*

8.5 *Representation of Binary Trees*

8.6 *Binary Tree Traversals*

8.7 *Threaded Binary Trees*

8.8 *Applications*

**Fig. 8.1** An example tree

permits connectivity of the root to every other node in the tree. In general, any node is reachable from anywhere in the tree. Also, with branches providing the links between the nodes, the structure ensures that no set of nodes link together to form a closed loop or a cycle.

## Basic terminologies of trees

There are several basic terminologies associated with the tree. The specially designated node called root has already been introduced in the definition. The number of subtrees of a node is known as the *degree of the node*. Nodes that have zero degree are called *leaf nodes* or *terminal nodes*. The rest of them are called as *non terminal nodes*. These nodes which hang from branches emanating from a node are known as *children* and the node from which the branches emanate is known as the *parent node*. Children of the same parent node are referred to as *siblings*. The *ancestors* of a given node are those nodes that occur on the path from the root to the given node. The *degree of a tree* is the maximum degree of the node in the tree. The level of a node is defined by letting the root to occupy level 1 (some authors let the root occupy level 0). The rest of the nodes occupy various levels depending on their association. Thus if a parent node occupies level  $i$ , its children should occupy level  $i+1$ . This renders the tree to have a *hierarchical structure* with root occupying the top most level of 1. The *height* or *depth* of a tree is defined to be the maximum level of any node in the tree. Some authors define depth of a node to be the length of the longest path from the root node to that node, which yields the relation,

$$\text{depth of the tree} = \text{height of the tree} - 1$$

A *forest* is a set of zero or more disjoint trees. The removal of the root node from a tree results in a forest (of its subtrees!).

In Fig. 8.1,  $A$  is the root node. The degree of node  $E$  is 2 and  $L$  is 0.  $F, G, H, C, I, J$  and  $L$  are leaf or terminal nodes and all the remaining nodes are non leaf or non terminal nodes. Nodes  $F, G$  and  $H$  are children of  $B$  and  $B$  is a parent node. Nodes  $J, K$  and nodes  $F, G, H$  are sibling nodes with  $E$  and  $B$  as their respective parents. For the node  $L$ , nodes  $A, E$  and  $K$  are ancestors. The degree of the tree is 4 which is the maximum degree reported by node  $A$ . While node  $A$  which is the root node occupies level 1, its children  $B, C, D$  and  $E$  occupy level 2 and so on. The height of the tree is its maximum level which is 4. Removal of  $A$  yields a forest of four disjoint (sub) trees viz.,  $\{B, F, G, H\}$ ,  $\{C\}$ ,  $\{D, I\}$  and  $\{E, J, K, L\}$ .

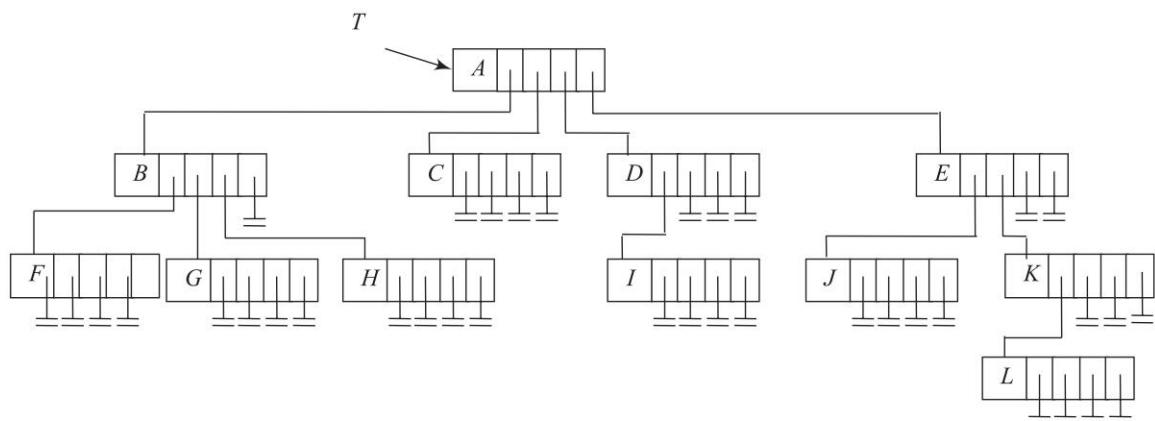
## Representation of Trees

### 8.3

Though trees are better understood in their pictorial forms, a common representation of a tree to suit its storage in the memory of a computer, is a *list*. The tree of Fig. 8.1 could be represented in its list form as  $(A (B(F,G,H), C, D(I), E(J,K(L))))$ . The root node comes first followed by the list of subtrees of the node. This is repeated for each subtree in the tree. This list form of a tree, paves way for a naïve representation of the tree as a linked list. The node structure of the linked list is shown in Fig. 8.2(a).

|      |        |        |     |          |
|------|--------|--------|-----|----------|
| DATA | LINK 1 | LINK 2 | ... | LINK $n$ |
|------|--------|--------|-----|----------|

(a) General node structure



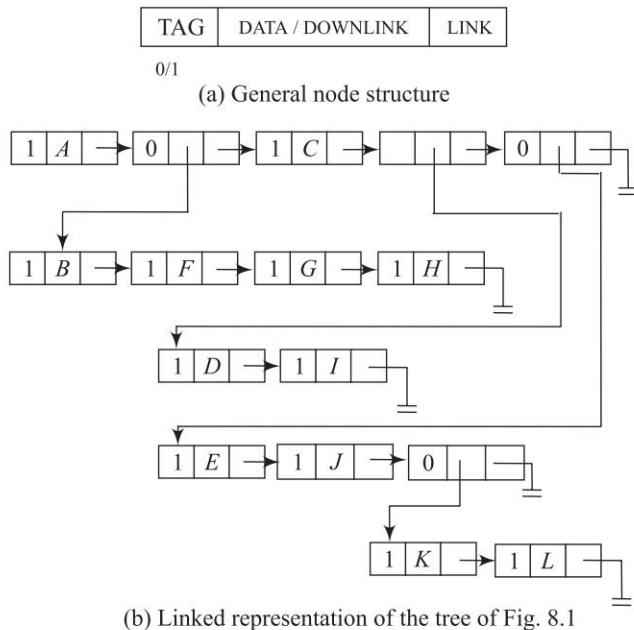
(b) Linked list representation of the tree shown in Fig. 8.1

**Fig. 8.2** *Linked list representation of a tree*

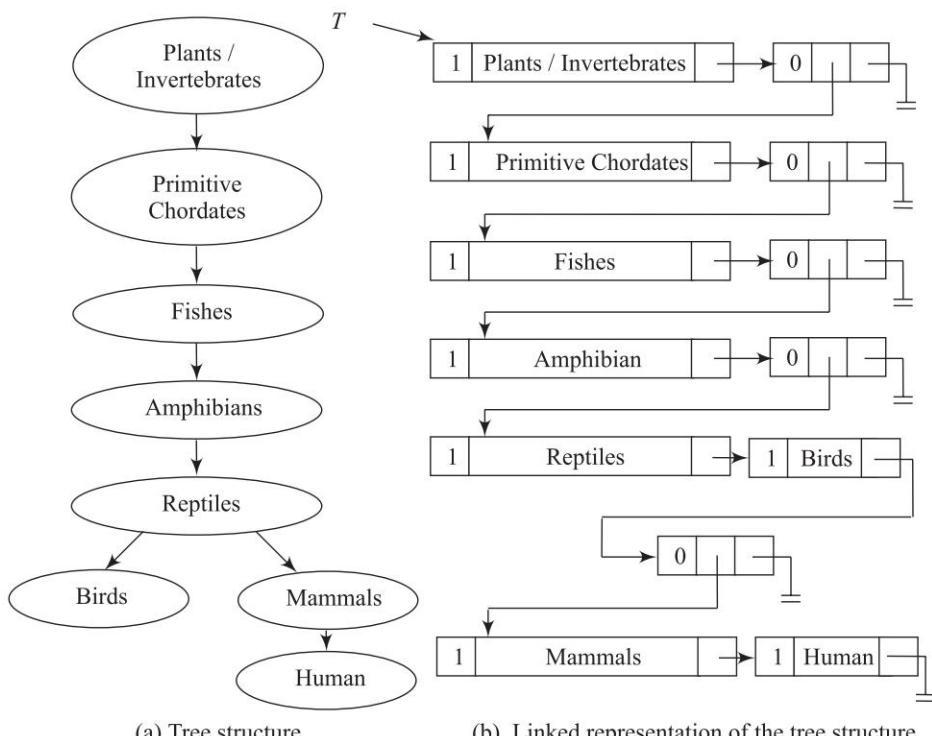
The DATA field of the node stores the information content of the tree node. A fixed set of LINK fields accommodate the pointers to the child nodes of the given node. In fact the *maximum number of links the node would require is equal to the degree of the tree*. The linked representation of the tree shown in Fig. 8.1 is illustrated in Fig. 8.2 (b). Observe the colossal wastage of space by way of null pointers!

An alternative representation would be to use a node structure as shown in Fig. 8.3(a). Here TAG = 1 indicates that the next field (DATA / DOWN LINK) is occupied by data (DATA) and TAG = 0 indicates that the same is used to hold a link (DOWN LINK). The node structure of the linked list holds a DOWNLINK whenever it encounters a child node which gives rise to a subtree. Thus the root node A has four child nodes, three of which viz.,  $B$ ,  $D$  and  $E$  give rise to subtrees. Note the DOWNLINK active fields of the nodes in these cases with TAG set to 0. In contrast, observe the linked list node corresponding to  $C$  which has no subtree. The DATA field records  $C$  with TAG set to 1.

**Example 8.1** We illustrate a tree structure in the organic evolution which deals with the derivation of new species of plants and animals from the first formed life by descent with modification. Figure 8.4 (a) illustrates the tree and Fig. 8.4 (b) shows its linked representation.



**Fig. 8.3 An alternative elegant linked representation of a tree**



**Fig. 8.4 Tree structure of organic evolution**

## Binary Trees: Basic Terminologies and Types

8.4

### Basic terminologies

A **binary tree** has the characteristic of all nodes having at most two branches, that is, all nodes have a *degree of at most 2*. A binary tree can therefore be *empty* or consist of a root node and two disjointed binary trees termed *left subtree* and *right subtree*. Figure 8.5 illustrates a binary tree.

It is essential that the distinction between trees and binary trees are brought out clearly. While a binary tree can be empty with zero nodes, a tree can never be empty. Again while the ordering of the subtrees in a tree is immaterial, in a binary tree the distinction of left and right subtrees are very clearly maintained. All other terminologies applicable to trees such as levels, degree, height, leaf nodes, parent, child, siblings etc. are also applicable to binary trees. However, there are some important observations regarding binary trees.

- (i) The maximum number of nodes on level  $i$  of a binary tree is  $2^{i-1}$ ,  $i \geq 1$ .
- (ii) The maximum number of nodes in a binary tree of height  $h$  is  $2^h - 1$ ,  $h \geq 1$ . (for proof refer Illustrative Problem 6 of Chapter 8)
- (iii) For any non empty binary tree, if  $t_0$  is the number of terminal nodes and  $t_2$  is the number of nodes of degree 2, then  $t_0 = t_2 + 1$  (for proof refer Illustrative Problem 8.7)

These observations could be easily verified on the binary tree shown in Fig. 8.5. The maximum number of nodes on level 3 is  $2^{3-2} = 2^2 = 4$ . Also with the height of the binary tree being 3, the maximum number of nodes =  $2^3 - 1 = 7$ . Again  $t_0 = 4$  and  $t_2 = 3$  which yields  $t_0 = t_2 + 1$ .

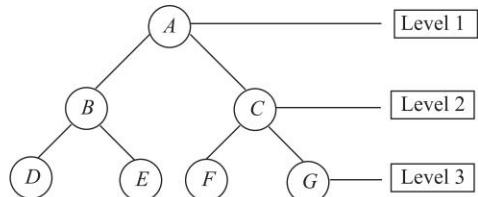
### Types of binary trees

A binary tree of height  $h$  which has all its permissible maximum number of nodes viz.,  $2^h - 1$  intact is known as a **full binary tree of height  $h$** . Figure 8.6(a) illustrates a full binary tree of height 4. Note the specific method of numbering the nodes.

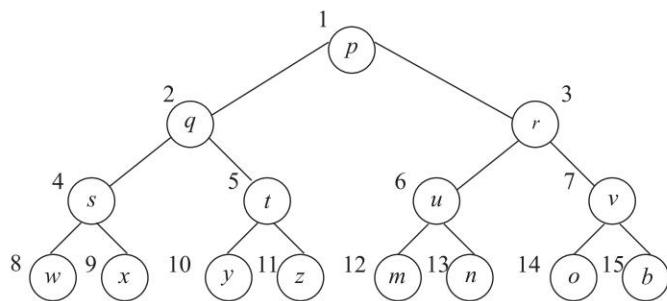
A binary tree with  $n'$  nodes and height  $h$  is *complete* if its nodes correspond to the nodes which are numbered 1 to  $n$  ( $n' \leq n$ ) in a full binary tree of height  $h$ . In other words, a **complete binary tree** is one in which its nodes follow a sequential numbering that increments from a left-to-right and top-to-bottom fashion. A full binary tree is therefore a special case of a complete binary tree. Also, the height of a complete binary tree with  $n$  elements has a height  $h$  given by  $h = \lceil \log_2 (n + 1) \rceil$ . A complete binary tree obeys the following properties with regard to its node numbering:

- (i) If a parent node has a number  $i$  then its left child has the number  $2i$  ( $2i \leq n$ ). If  $2i > n$  then  $i$  has no left child.
- (ii) If a parent node has a number  $i$ , then its right child has the number  $2i + 1$  ( $2i + 1 \leq n$ ). If  $2i + 1 > n$  then  $i$  has no right child.
- (iii) If a child node (left or right) has a number  $i$  then the parent node has the number  $\lfloor i/2 \rfloor$  if  $i \neq 1$ . If  $i = 1$  then  $i$  is the root and hence has no parent.

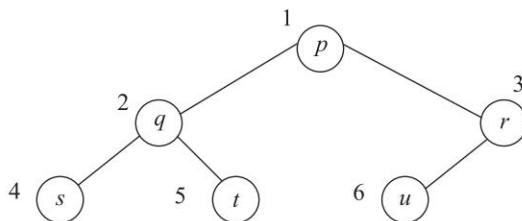
In the full binary tree of height 4 illustrated in Fig. 8.6(a), observe how the parent-child numbering is satisfied. For example, consider node  $s$  (number 4), its left child  $w$  has the number  $2*4=8$  and its right child has the number  $2*4+1=9$ . Again the parent of node  $v$  (number 7) is the node with number  $\lfloor 7/2 \rfloor = 3$  (i.e.) node 3 which is  $r$ .



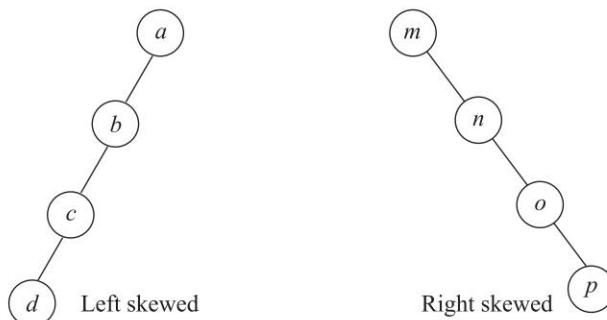
**Fig. 8.5** An example of binary tree



(a) Full binary tree of height 4



(b) A complete binary tree of height 3



(c) Skewed binary tree

**Fig. 8.6** Examples of full binary tree, complete binary tree and skewed binary trees

Figure 8.6(b) illustrates an example complete binary tree. A binary tree which is dominated solely by left child nodes or right child nodes is called a **skewed binary tree** or more specifically **left skewed binary tree** or **right skewed binary tree** respectively. Figure 8.6(c) illustrates examples of skewed binary trees.

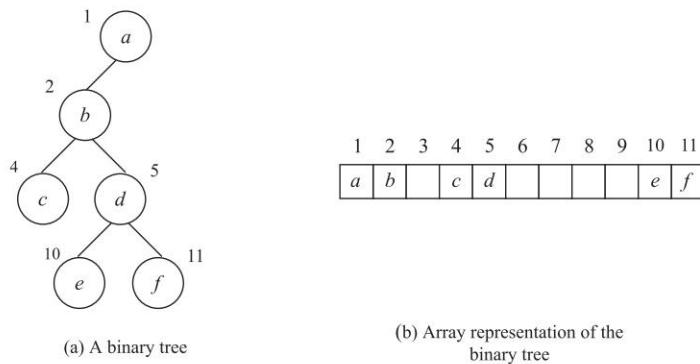
## Representation of Binary Trees

## 8.5

A binary tree could be represented using a sequential data structure (arrays) as well as linked data structure.

## Array representation of binary trees

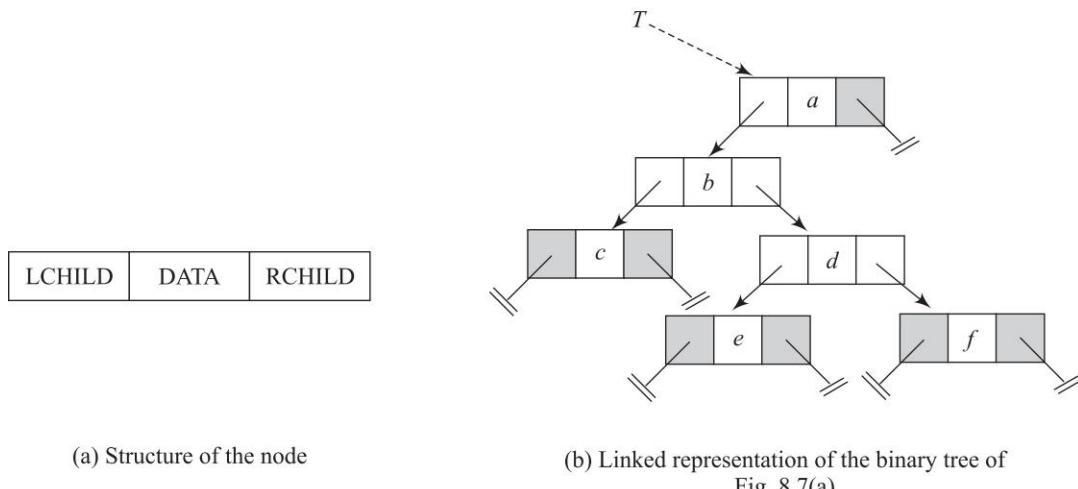
To represent the binary tree as an array, the sequential numbering system emphasized by a complete binary tree comes in handy. Consider the binary tree shown in Fig. 8.7(a). The array representation is as shown in Fig. 8.7(b). The association of numbers pertaining to parent and left/right child nodes makes it convenient to access the appropriate cells of the array. However, the missing nodes in the binary tree and hence the corresponding array locations, are left empty in the array. This obviously leads to a lot of wastage of space. However, the array representation ideally suits a full binary tree due to its non wastage of space.



**Fig. 8.7** Array representation of a binary tree

## Linked representation of binary trees

The linked representation of a binary tree has the node structure shown in Fig. 8.8(a). Here, the node, besides the DATA field, needs two pointers LCHILD and RCHILD to point to the left and right child nodes respectively. The tree is accessed by remembering the pointer to the root node of the tree.



**Fig. 8.8** Linked representation of a binary tree

In the binary tree  $T$  shown in Fig. 8.8(b), LCHILD ( $T$ ) refers to the node storing  $b$  and RCHILD (LCHILD ( $T$ )) refers to the node storing  $d$  and so on. The following are some of the important observations regarding the linked representation of a binary tree:

- If a binary tree has  $n$  nodes then the number of pointers used in its linked representation is  $2 * n$ .
- The number of null pointers used in the linked representation of a binary tree with  $n$  nodes is  $n + 1$ .

However, in a linked representation it is difficult to determine a parent given a child node. In any case if an application so requires, a fourth field PARENT may also be included in the structure.

## Binary Tree Traversals

## 8.6

An important operation that is performed on a binary tree is its *traversal*. A traversal of a binary tree is where its nodes are visited in a particular but repetitive order, rendering a linear order of the nodes or information represented by them.

A traversal is governed by three actions, viz. *Move left* (L), *Move Right* (R) and *Process Node* (P). In all, it yields six different combinations of LPR, LRP, PLR, PRL and RLP. Of these, three have emerged significant in computer science. They are,

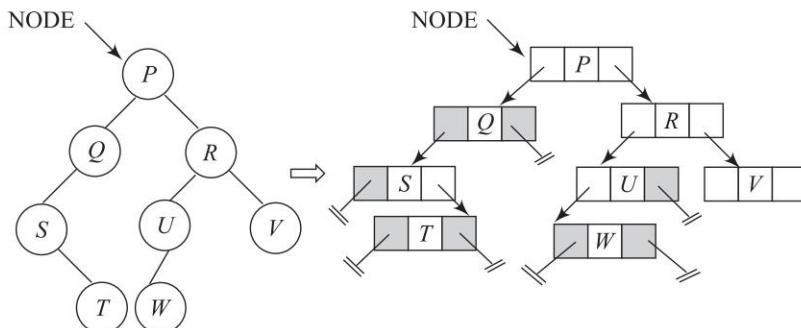
- LPR — *Inorder traversal*
- LRP — *Postorder traversal*
- PLR — *Preorder traversal*.

The algorithms for each of the traversals are elaborated here.

### Inorder Traversal

The traversal keeps moving left in the binary tree until one can move no further, processes the node and moves to the right to continue its traversal again. In the absence of any node to the right, it retracts backwards by a node and continues the traversal.

Algorithm 8.1 illustrates a recursive procedure to perform inorder traversal of a binary tree. For clarity of application, the action *Process Node* (P) is interpreted as *Print node*. Observe the recursive procedure reflect the maxim LPR repetitively. Example 8.2 illustrates the inorder traversal of the binary tree shown in Fig. 8.9.



**Fig. 8.9** Binary tree to demonstrate Inorder, Postorder and Preorder traversals

**Example 8.2** An easy method to obtain the traversal would be to run one's fingers on the binary tree with the maxim: *move left until no more nodes, process node, then move right and continue the traversal.*

An alternative method is to trace the recursive steps of the algorithm using the following scheme:

**Algorithm 8.1:** Recursive procedure to perform Inorder traversal of a binary tree

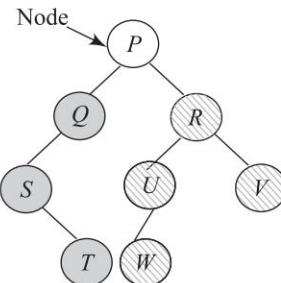
```

procedure INORDER_TRAVERSAL (NODE)
    /* NODE refers to the Root node of the binary tree
       in its first call to the procedure. Root node is the
       starting point of the traversal */
    if NODE ≠ NIL then
        { call INORDER_TRAVERSAL (LCHILD(NODE));
          /* Inorder traverse the left subtree (L) */
          print (DATA (NODE)) ;
          /* Process node (P) */
          call INORDER_TRAVERSAL (RCHILD(NODE));
          /* Inorder traverse the right subtree (R) */
        }
    end INORDER_TRAVERSAL.

```

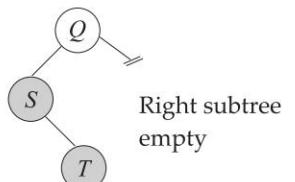
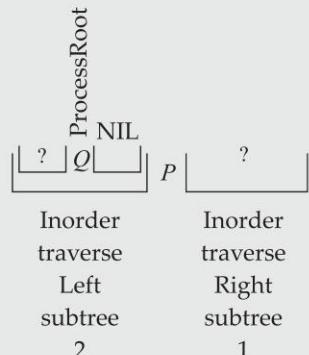
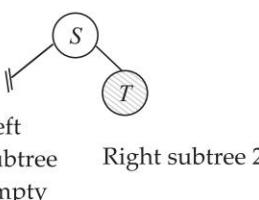
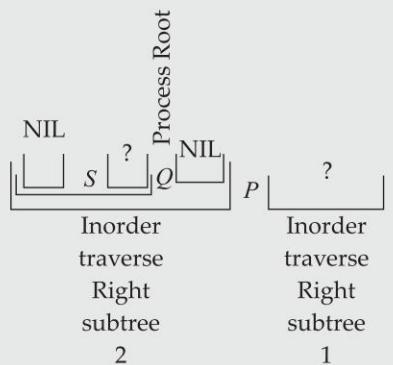
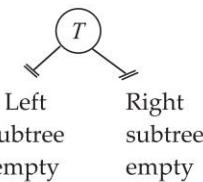
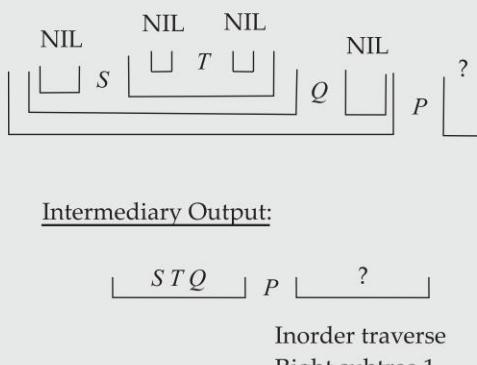
Execute the traversal of the binary tree as *traverse left subtree, process root node and traverse right subtree*. Repeat the same for each of the left and right subtrees encountered. Table 8.1. illustrates the traversal of the binary tree shown in Fig. 8.9, using this scheme. Each open box in the inorder traversal output (Column 2 of Table 8.1) represents the output of the call to the procedure INORDER\_TRAVERSAL with the root of the appropriate subtree as its input. The final output of the inorder traversal is S T Q P W U R V.

**Table 8.1** Inorder traversal of binary tree shown in Fig. 8.9

| Binary Tree                                                                                                                                                                                                     | Inorder Traversal Output                                                                                                                                                                                                                                                                                                                                                                                                                                            | Remarks |   |   |                                          |                                           |  |                                                                                               |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------|---|---|------------------------------------------|-------------------------------------------|--|-----------------------------------------------------------------------------------------------|
| <u>Step 1</u><br>Inorder Traverse Binary Tree<br><br>Node → P<br>S → T, W<br>R → U, V<br>Left subtree 1      Right subtree 1 | <table style="width: 100%; border-collapse: collapse;"> <tr> <td style="text-align: center; padding: 10px;">?</td> <td style="text-align: center; padding: 10px;">P</td> <td style="text-align: center; padding: 10px;">?</td> </tr> <tr> <td style="text-align: center; padding: 10px;">Inorder<br/>traverse<br/>Left<br/>subtree 1</td> <td style="text-align: center; padding: 10px;">Inorder<br/>traverse<br/>Right<br/>subtree 1</td> <td></td> </tr> </table> | ?       | P | ? | Inorder<br>traverse<br>Left<br>subtree 1 | Inorder<br>traverse<br>Right<br>subtree 1 |  | Inorder traversals of the Left and Right subtrees of the root node are to yield their output. |
| ?                                                                                                                                                                                                               | P                                                                                                                                                                                                                                                                                                                                                                                                                                                                   | ?       |   |   |                                          |                                           |  |                                                                                               |
| Inorder<br>traverse<br>Left<br>subtree 1                                                                                                                                                                        | Inorder<br>traverse<br>Right<br>subtree 1                                                                                                                                                                                                                                                                                                                                                                                                                           |         |   |   |                                          |                                           |  |                                                                                               |

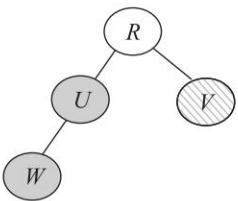
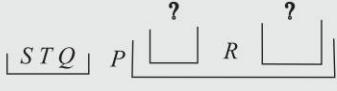
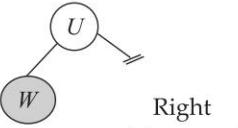
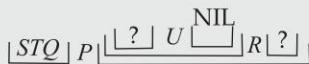
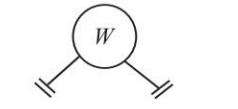
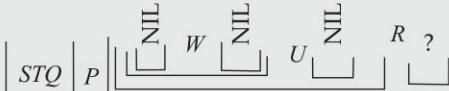
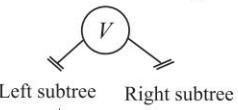
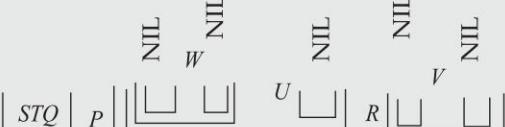
(Contd.)

(Contd.)

|                                                                                                                                                                                                  |                                                                                                                                                                                                            |                                                                                                                                                                                                                                                |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p><u>Step 2</u><br/>Inorder Traverse Left subtree 1</p>  <p>Right subtree empty</p> <p>Left subtree 2</p>      |  <p>Inorder traversal<br/>Left subtree 2      Inorder traversal<br/>Right subtree 1</p>                                   | <p>Inorder traversal of the Left subtree 1 yields Inorder traversal of Left subtree 2, process root <math>Q</math>, and Inorder traverse Right subtree.</p> <p>However, since the Right subtree is empty, its traversal yields NIL output.</p> |
| <p><u>Step 3</u><br/>Inorder Traverse Left subtree 2</p>  <p>Left subtree empty      Right subtree 2</p>       |  <p>Inorder traversal<br/>Right subtree 2      Inorder traversal<br/>Right subtree 1</p>                                 |                                                                                                                                                                                                                                                |
| <p><u>Step 4</u><br/>Inorder Traverse Right subtree 2</p>  <p>Left subtree empty      Right subtree empty</p> |  <p>NIL      NIL      NIL      NIL</p> <p><u>Intermediary Output:</u></p> <p>Inorder traversal<br/>Right subtree 1</p> | <p>Inorder traversal of Left subtree 1 is done.</p> <p>Gathering the traversal's output for Left subtree 1 yields <math>STQ</math>.</p> <p>The Inorder Traversal of Right subtree1 needs to be performed.</p>                                  |

(Contd.)

(Contd.)

|                                                                                                                                                                                                   |                                                                                                                                                                          |                                                                                                                                                        |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p><u>Step 5</u></p> <p>Inorder Traverse Right subtree 1</p>  <p>Left subtree 3    Right subtree empty</p>       | <p>Inorder traverse<br/>Left subtree 3</p>  <p>Inorder traverse<br/>Right subtree 3</p> | <p>Inorder Traversal of the Left subtree 3 and Right subtree 3 are to yield their output.</p>                                                          |
| <p><u>Step 6</u></p> <p>Inorder Traverse Left subtree 3</p>  <p>Right subtree empty</p> <p>Left subtree 4</p>    |  <p>Inorder traverse<br/>Left subtree 4      Inorder traverse<br/>Right subtree 3</p>   |                                                                                                                                                        |
| <p><u>Step 7</u></p> <p>Inorder Traverse Left subtree 4</p>  <p>Left subtree empty    Right subtree empty</p>  |  <p>Inorder traverse<br/>Right subtree 3</p>                                          |                                                                                                                                                        |
| <p><u>Step 8</u></p> <p>Inorder Traverse Right subtree 3</p>  <p>Left subtree empty    Right subtree empty</p> |  <p>Final Output:</p> <p>STQPWURV</p>                                                | <p>Inorder traversal of Right subtree 1 is done.<br/>Gathering the traversal's output yields WURV.</p> <p><b>Final output:</b><br/><b>STQPWURV</b></p> |

## Postorder Traversal

The traversal proceeds by keeping to the left until it is no further possible, turns right to begin again or if there is no node to the right, processes the node and retraces its direction by one node to continue its traversal.

Algorithm 8.2 illustrates a recursive procedure to perform post order traversal of a binary tree. The recursive procedure reflects the maxim LRP invoked repetitively. Example 8.3 illustrates the postorder traversal of the binary tree shown in Fig. 8.9. The traversal output is *TSQWUVRP*.

**Algorithm 8.2:** Recursive procedure to perform Postorder traversal of a binary tree

```

procedure POSTORDER_TRAVERSAL (NODE)
    /* NODE refers to the Root node of the binary tree
       in its first call to the procedure. Root node is the
       starting point of the traversal */
If NODE ≠ NIL then
    { call POSTORDER_TRAVERSAL (LCHILD(NODE)) ;
      /* Postorder traverse the left subtree (L) */
      call POSTORDER_TRAVERSAL (RCHILD(NODE)) ;
      /* Postorder traverse the right subtree (R) */
      print (DATA (NODE)) ;
      /* Process node (P) */
    }
end POSTORDER_TRAVERSAL.

```

**Example 8.3** As pointed out in Example 8.2, an easy method would be to run one's fingers on the binary tree with the maxim: *move left until there are no more nodes and turn right to continue traversal. If there is no right node, process node, retract by one node and continue traversal.*

An alternative method would be to trace the recursive steps of the algorithm using the scheme: *Traverse left subtree, Traverse right subtree and Process root node*. Table 8.2 illustrates the traversal of the binary tree shown in Fig. 8.9 using this scheme.

## Preorder Traversal

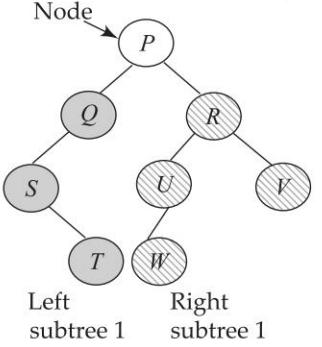
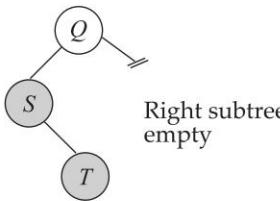
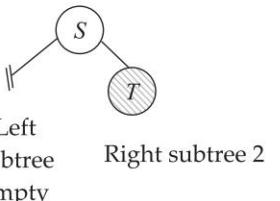
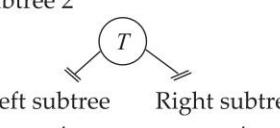
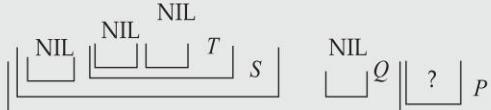
The traversal processes every node as it moves left until it can move no further. Now it turns right to begin again or if there is no node in the right, retracts until it can move right to continue its traversal.

The recursive procedure for the preorder traversal is illustrated in Algorithm 8.3. The recursive procedure reflects the maxim PLR invoked repetitively. Example 8.4 illustrates the preorder traversal of the binary tree shown in Fig. 8.9. The traversal output is *PQSTRUWV*.

**Example 8.4** An easy method as discussed before would be to trace the traversal on the binary tree using the maxim: *Process nodes while moving left until no more nodes, turn right, and otherwise retract to continue the traversal.*

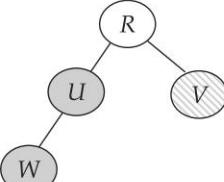
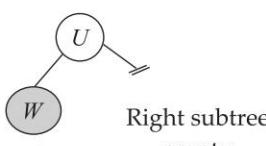
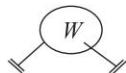
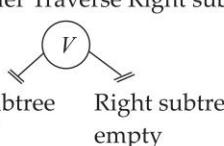
An alternative method is to trace the recursive steps of the algorithm using the following scheme:

**Table 8.2** Postorder traversal of binary tree shown in Fig. 8.9

| Binary Tree                                                                                                                                                                               | Postorder Traversal Output                                                                                                                                                     | Remarks                                                                                                              |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------|
| <u>Step 1</u><br>Postorder traverse Binary Tree Node<br><br>Left subtree 1      Right subtree 1          | <br>Postorder traverse Left subtree 1      Postorder traverse Right subtree 1<br>Process Root | Postorder Traversal of the Left and Right subtrees of the root is yet to yield their output.                         |
| <u>Step 2</u><br>Postorder Traverse Left subtree 1<br><br>Left subtree 2                                 | Postorder traverse Left subtree 2<br><br>Postorder traverse Right subtree 1                   |                                                                                                                      |
| <u>Step 3</u><br>Postorder Traverse Left subtree 2<br><br>Left subtree 2      Right subtree 2          | <br>Postorder traverse Right subtree 2      Postorder traverse Right subtree 1              |                                                                                                                      |
| <u>Step 4</u><br>Postorder traverse Right subtree 2<br><br>Left subtree empty      Right subtree empty | <br><hr/><br><u>Intermediary Output:</u> <u>TSQ</u>   [ ] ?   P                            | Postorder traversal of Left subtree 1 is done.<br><br>Gathering the traversal's output for Left subtree 1 yields TSQ |

(Contd.)

(Contd.)

|                                                                                                                                                                                                    |                                                                                                                                                                                        |                                                                                                                                                            |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p><u>Step 5</u><br/>Postorder traverse Right subtree 1</p>  <p>Left subtree 3      Right subtree 3</p>           | <p>Postorder traverse<br/>Right subtree 3</p> <p><math>  T S Q   [ ? ] [ ? ] R   P</math></p> <p>Postorder traverse<br/>Left subtree 3</p>                                             |                                                                                                                                                            |
| <p><u>Step 6</u><br/>Postorder Traverse Left subtree 3</p>  <p>Left subtree 4      Right subtree empty</p>        | <p><math>  T S Q   [ ? ] [ NIL ] U [ ? ] R   P</math></p> <p>Postorder Traverse Left subtree 4      Postorder Traverse Right subtree 3</p>                                             |                                                                                                                                                            |
| <p><u>Step 7</u><br/>Postorder traverse left subtree 4</p>  <p>Left subtree empty      Right subtree empty</p>  | <p><math>  NIL   NIL   W</math></p> <p><math>  T S Q   [ ? ] [ NIL ] U [ ? ] R   P</math></p> <p>Postorder traverse<br/>Right subtree 3</p>                                            |                                                                                                                                                            |
| <p><u>Step 8</u><br/>Postorder Traverse Right subtree 3</p>  <p>Left subtree empty      Right subtree empty</p> | <p><math>  NIL   NIL   W   NIL   NIL   V</math></p> <p><math>  T S Q   [ ? ] [ NIL ] U [ ? ] R   P</math></p> <hr/> <p>Final Output:</p> <p><math>  T S Q   [ W U V R ]   P</math></p> | <p>Postorder traversal of Right subtree 1 is done. Gathering the output yields <math>WUVR</math></p> <p><b>Final output:</b><br/><math>TSQWUVRP</math></p> |

**Algorithm 8.3:** Recursive procedure to perform Preorder traversal of a binary tree

```

procedure PREORDER_TRAVERSAL (NODE)
    /* NODE refers to the Root node of the binary tree
       in its first call to the procedure. Root node is the
       starting point of the traversal */
If NODE ≠ NIL then
    { print (DATA (NODE)) ;
      /* Process node (P) */
      call PREORDER_TRAVERSAL (LCHILD(NODE));
      /* Preorder traverse the left subtree (L) */
      call PREORDER_TRAVERSAL (RCHILD(NODE));
      /* Preorder traverse the right subtree (R) */
    }
end PREORDER_TRAVERSAL.

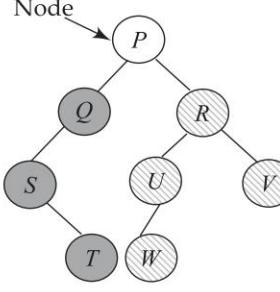
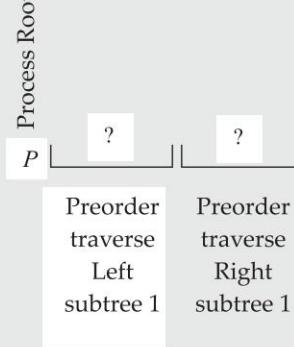
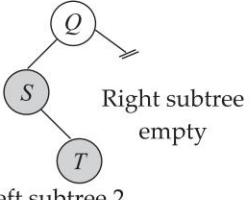
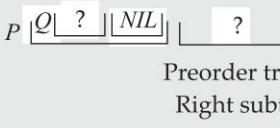
```



Execute the traversal of the binary tree as, *process root node, traverse left subtree and traverse right subtree*, repeating the same for each of the left and right subtrees encountered.

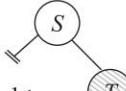
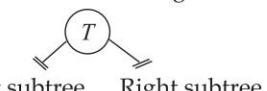
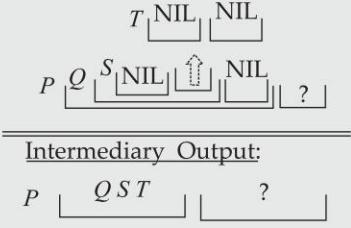
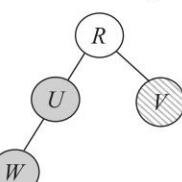
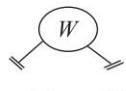
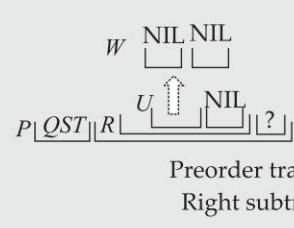
Table 8.3 illustrates the preorder traversal of the binary tree shown in Fig. 8.9 using this scheme. Some significant observations pertaining to the traversals of a binary tree are the following

**Table 8.3** Preorder traversal of binary tree shown in Fig. 8.9

| Binary Tree                                                                                                                                                                         | Preorder Traversal Output                                                                                                                                                     | Remarks                                                                              |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------|
| <u>Step 1</u><br>Preorder traversal of Binary Tree<br><br>Left subtree 1      Right subtree 1     | Process Root<br><br>Preorder traverse Left subtree 1      Preorder traverse Right subtree 1 | Preorder traversals of the Left subtree 1 and Right subtree 1 to yield their output. |
| <u>Step 2</u><br>Preorder traverse Left subtree 1<br><br>Left subtree 2      Right subtree empty | Preorder traverse Left subtree 2<br><br>Preorder traverse Right subtree 1                  |                                                                                      |

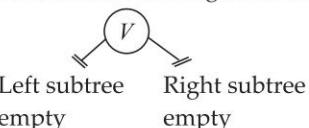
(Contd.)

(Contd.)

|                                                                                                                                                                                                     |                                                                                                                                                                                           |  |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--|
| <p><u>Step 3</u></p> <p>Preorder traverse<br/>Left subtree 2</p>  <p>Left subtree empty      Right subtree 2</p>   | <p>Preorder traverse<br/>Right subtree 2</p> $P \boxed{Q \boxed{S \boxed{\text{NIL}} \boxed{?} \boxed{\text{NIL}}} \boxed{?}}$ <p>Preorder traverse<br/>Right subtree 1</p>               |  |
| <p><u>Step 4</u></p> <p>Preorder traverse Right subtree 2</p>  <p>Left subtree empty      Right subtree empty</p>  |  <p>Preorder traversal of Left subtree 1 is done. Gathering the traversal's output yields <i>QST</i></p> |  |
| <p><u>Step 5</u></p> <p>Preorder traverse Right subtree 1</p>  <p>Left subtree 3      Right subtree 3</p>          | <p>Preorder traverse<br/>Left subtree 3</p> $P \boxed{Q \boxed{S \boxed{T}} \boxed{R \boxed{?} \boxed{?}}}$ <p>Preorder traverse<br/>Right subtree 3</p>                                  |  |
| <p><u>Step 6</u></p> <p>Preorder traverse Left subtree 3</p>  <p>Left subtree 4      Right subtree empty</p>     | <p>Preorder traverse<br/>Left subtree 4</p> $P \boxed{Q \boxed{S \boxed{T}} \boxed{R \boxed{U \boxed{?} \boxed{\text{NIL}}} \boxed{?}}}$ <p>Preorder traverse<br/>Right subtree 3</p>     |  |
| <p><u>Step 7</u></p> <p>Preorder traverse left subtree 4</p>  <p>Left subtree empty      Right subtree empty</p> |  <p>Preorder traverse<br/>Right subtree 3</p>                                                          |  |

(Contd.)

(Contd.)

|                                                                                                                                                                                                 |                                                                                                                     |                                                                                                                                                         |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p><b>Step 8</b><br/>Preorder traverse Right subtree 3</p>  <p>Left subtree empty      Right subtree empty</p> | <p>W NIL NIL      V NIL NIL</p> <p>P Q S T R U NIL NIL</p> <hr/> <p><b>Final Output:</b></p> <p>P Q S T R U W V</p> | <p>Preorder traversal of Right subtree 1 is done.<br/>Gathering the traversal's output yields RUWV.</p> <p><b>Final output:</b><br/><b>PQSTRUWV</b></p> |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------|

- (i) Given a preorder traversal of a binary tree, the root node is the first occurring item in the list.
- (ii) Given a postorder traversal of a binary tree, the root node is the last occurring item in the list.
- (iii) Inorder traversal does not directly reveal the root node of the binary tree.
- (iv) An inorder traversal coupled with any one of preorder or post order traversal helps trace back the structure of the binary tree (Refer Illustrative Problems 8.3, 8.4.).

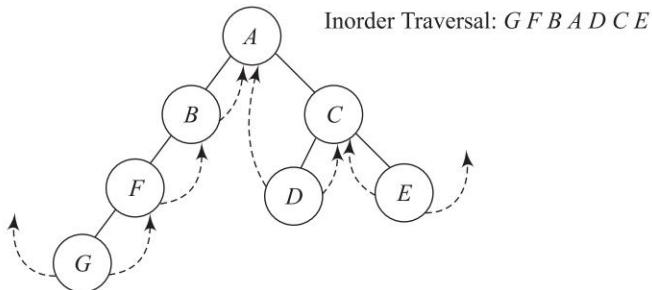
## Threaded Binary Trees

8.7

The linked representation of the binary tree discussed in Section 8.5 showed that for a binary tree with  $n$  nodes,  $2n$  pointers are required of which  $(n+1)$  are null pointers. A.J. Perlis and C.Thornton devised a prudent method to utilize these  $(n+1)$  empty pointers, introducing what are called *threads*. Threads are also links or pointers but replace null pointers by pointing to some useful information in the binary tree. Thus, for a node NODE if RCHILD(NODE) is NIL then the null pointer is replaced by a thread which points to the node which would occur after NODE when the binary tree is traversed in inorder. Again if LCHILD (NODE) is NIL then the null pointer is replaced by a thread to the node which would immediately precede NODE when the binary tree is traversed in inorder.

Figure 8.10. illustrates a threaded binary tree. The threads are indicated using broken lines to distinguish them from the normal links indicated with solid lines. The inorder traversal of the binary tree is also shown in the figure.

Note that the left child of G and the right child of E have threads which are left dangling due to the absence of an inorder predecessor and successor respectively.

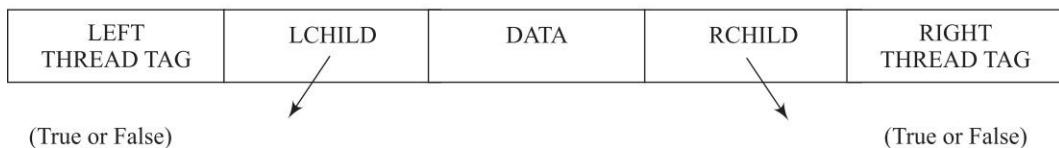


**Fig. 8.10** A threaded binary tree

There are many ways to thread a binary tree  $T$ , corresponding to the specific traversal chosen. In this work, we have the threading correspond to an Inorder traversal. Also the threading can be of two representations viz., *one-way threading* and *two-way threading*. One-way threading is where a thread appears only on the RCHILD field of a node, when it is null, pointing to the inorder successor of the node (Refer Illustrative Problem I8.10). On the other hand, in two-way threading, which had been introduced above, a thread appears in the LCHILD field also, if it is null, which points to the inorder predecessor of the node. However, the first and the last of the nodes in the inorder traversal will carry dangling threads.

### Linked representation of a threaded binary tree

A linked representation of the threaded binary tree (two-way threading) has a node structure as shown in Fig. 8.11.



**Fig. 8.11 Node Structure of a linked representation of a threaded binary tree**

Since the LCHILD and RCHILD fields are utilized to represent both links and threads it becomes essential for the node structure to clearly distinguish between them to avoid confusion while processing the threaded binary tree. Hence it is necessary that the node structure includes two more fields which act as flags to indicate if the LCHILD and RCHILD fields represent a thread or a link.

If the LEFT THREAD TAG or RIGHT THREAD TAG is marked *true* then LCHILD and RCHILD fields represent threads otherwise they represent links. Also, to tuck in the dangling threads which are bound to arise, the linked representation of a threaded binary tree includes a *head node*. The dangling threads point to the head node. The head node by convention has its LCHILD pointing to the root node of the threaded binary tree and therefore has its LEFT THREAD TAG set to *false*. THE RIGHT THREAD TAG field is also set to *false* but the RCHILD link points to the head node itself. Figure 8.12(a) shows the linked representation of an empty threaded binary tree and Fig. 8.12(b) that of a non-empty threaded binary tree.

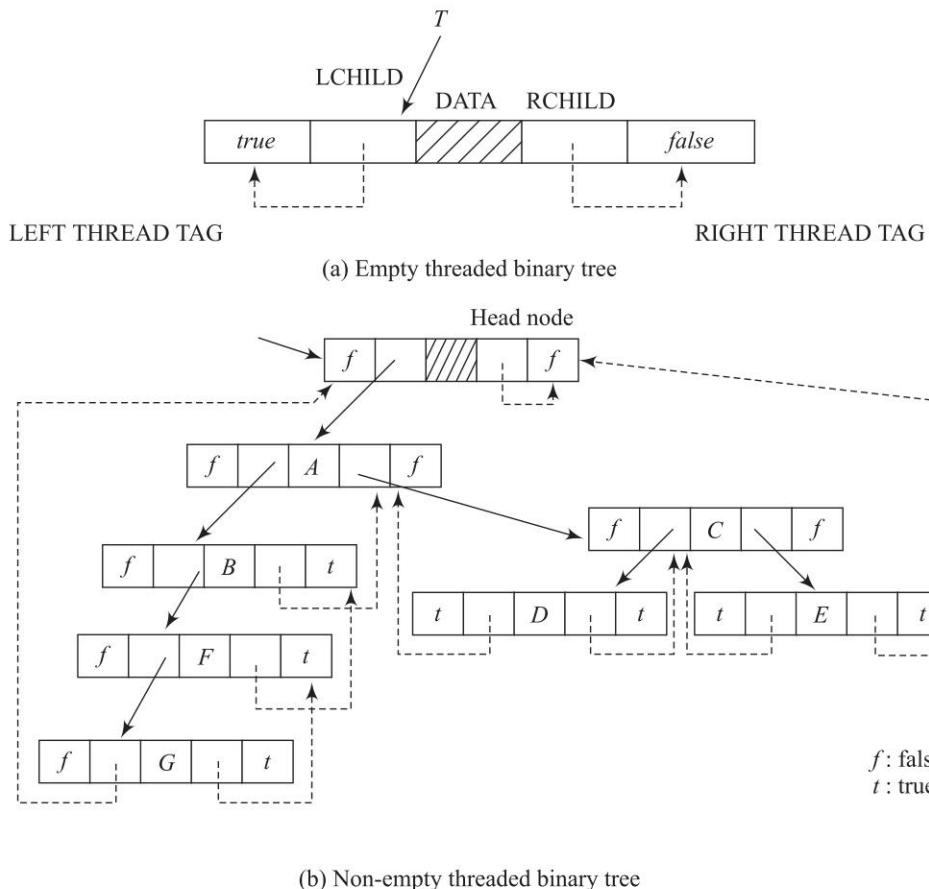
### Growing threaded binary trees

Here we discuss the insertion of nodes contributing to the growth of threaded binary trees. The insertion of a node calls not only for the realignment of links but also of the threads involved.

Consider the case of inserting a node NEW to the right of a node NODE in the threaded binary tree. If the node NODE had no right subtree then the case is trivial. Attach NEW as right child of NODE and appropriately reset the threads (LCHILD and RCHILD) of NEW to point to its inorder predecessor and successor respectively. Figure 8.13(a) illustrates this insertion.

In the next case, if NODE already had a right subtree then attach NEW as the right child of the node NODE and link the previous right subtree of NODE to the right of node NEW. When this is done, the threads of the appropriate nodes are reset as shown in Fig. 8.13(b).

A similar procedure is followed to insert a node in the left subtree of a threaded binary tree.

Fig. 8.12 *Linked representation of threaded binary trees*

## Application

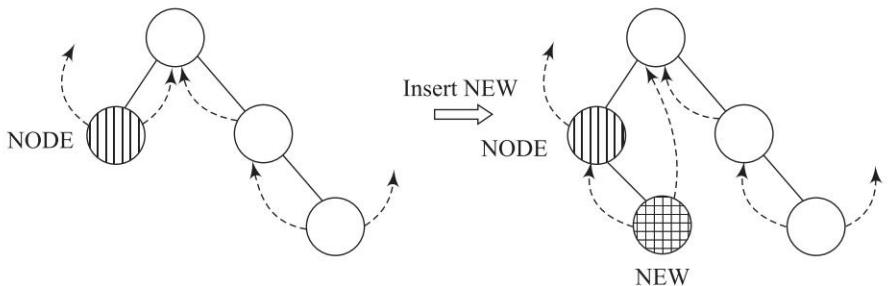
## 8.8

In this section we discuss an application of binary trees in expression trees which have a significant role to play in the principles of compiler design.

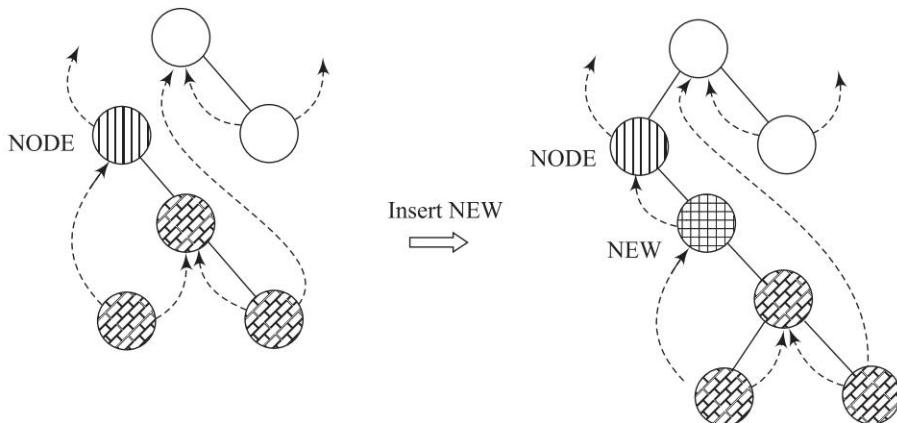
### Expression trees

Expressions—arithmetic and logical—are an inherent component of programming languages. The following are examples of arithmetic and logical expressions.

- $((A + B) * C - D) \uparrow G$  (arithmetic expression)
- $(\neg A \wedge B) \vee (B \wedge E) \wedge \neg F$  (logical expression)
- $(T < W) \vee (A \leq B) \wedge (C \neq E)$  (logical expression)



(a) Insertion of node NEW to the right of NODE: Right subtree of NODE is empty



(b) Insertion of node NEW to the right of NODE: Right subtree of NODE is non-empty

**Fig. 8.13** Insertion of a node in the right subtree of a threaded binary tree

That expressions are represented in three forms viz., infix, postfix and prefix were detailed in Sec. 4.3. To quickly review, an infix expression which is the commonly used representation of an expression follows the scheme  $\langle \text{operand} \rangle \text{ } \langle \text{operator} \rangle \text{ } \langle \text{operand} \rangle$ .

Examples are  $A + B$ ,  $A * B$ .

Post fix expressions follow the scheme  $\langle \text{operand} \rangle \text{ } \langle \text{operand} \rangle \text{ } \langle \text{operator} \rangle$ .

Examples are  $AB+$ ,  $AB^*$ .

Prefix expressions follow the scheme  $\langle \text{operator} \rangle \text{ } \langle \text{operand} \rangle \text{ } \langle \text{operand} \rangle$ .

Examples are  $+AB$ ,  $*AB$ .

Binary trees have found an application in the representation of expressions. An *expression tree* has the operands of the expression as its terminal or leaf nodes and the operators as its non terminal nodes. The arity of the operator is therefore restricted to be 1 or 2 (unary or binary) and this is what is commonly encountered in arithmetic and logical expressions. Figure 8.14 illustrates example expression trees.

The hierarchical precedence and associativity rules of the operators in terms of the expressions are reflected in the orientation of the subtrees or the sibling nodes. Table 8.4 illustrates some examples showing the orientation of the binary tree in accordance with the precedence and associativity rules of the operators in the expression terms.

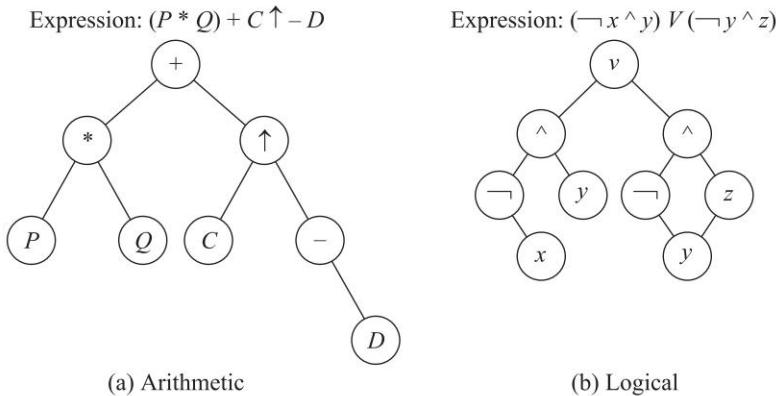


Fig. 8.14 Example of expression trees

Table 8.4 Orientation of the binary trees with regard to expressions

| Expression                | Expression Tree | Remarks                                                                                                                                                              |
|---------------------------|-----------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $A + B$                   |                 | Observe the orientation of the sibling nodes. The left operand $A$ and the right operand $B$ become the left and right child nodes of the operator $+$ respectively. |
| $B + A$                   |                 |                                                                                                                                                                      |
| $A + B + C$               |                 | The left associativity rule satisfied by $+$ is reflected in the orientation of the subtrees.                                                                        |
| $A \uparrow B \uparrow C$ |                 | The right associate rule satisfied by $\uparrow$ is reflected in the orientation of the subtrees                                                                     |
| $A * B - C / D$           |                 | The hierarchical precedence relation among the operators decides the orientation of the subtrees.                                                                    |

## Traversals of an expression tree

Section 8.6. detailed the traversals of a binary tree. With an expression tree essentially being a binary tree, the traversal of an expression tree also yields significant results. Thus, the inorder traversal of an expression tree yields an infix expression, the postorder traversal, a postfix expression and preorder traversal, a prefix expression. The output of the algorithms `INORDER_TRAVERSAL()`, `PREORDER_TRAVERSAL()` and `POSTORDER_TRAVERSAL()` on any given expression tree can be verified against the hand computed infix, prefix and postfix expressions (discussed in Sec. 4.3).

## Conversion of infix expression to postfix expression

We utilize this opportunity to introduce a significant concept of infix to postfix expression conversion which finds a useful place in the theory of compiler design.

Given an infix expression, for example  $A+B*C$ , the objective is to obtain its postfix equivalent  $ABC^*+$ . In Chapter 4, Sec. 4.3, a hand coded method of conversion was illustrated. In this section, we introduce an algorithm to perform the same.

The algorithm makes use of a stack as its work space and is governed by two priority factors viz., *In Stack priority* (ISP) and *Incoming Priority* (ICP) of the operators participating in the conversion. Thus those operators already pushed into the stack during the process of conversion, command an ISP in relation to those which are just about to be pushed in to the stack (ICP). Table 8.5 illustrates the ISP and ICP of a common list of arithmetic operators.

**Table 8.5** ISP and ICP of a common list of arithmetic operators

| Operator | ISP | ICP |
|----------|-----|-----|
| )        | -   | -   |
| ↑        | 3   | 4   |
| *, /     | 2   | 2   |
| +, -     | 1   | 1   |
| (        | 0   | 4   |

The rule which operates during conversion of infix to post fix expression is : *pop operators out of the work stack so long as the ICP of the incoming operator is less than or equal to the ISP of the operators already available in the stack.*

The input infix expression is padded with a “\$” to signal end of input. The bottom of the work stack is also signaled with a “\$” symbol with  $\text{ISP}(\$) = -1$ . Algorithm 8.4 illustrates the pseudo-code procedure to convert an infix expression into postfix expression.

**Algorithm 8.4:** Procedure to convert infix expression to postfix expression

```

procedure INFIX_POSTFIX_CONV(E)
    /* to convert an infix expression E padded with a "$"
       as its end-of-input symbol into its equivalent
       postfix expression */
    x := getnextchar (E);
    /* obtain the next character from E */

```

```

while x ≠ "$" do
    case x of
        : x is an operand: print x;
        : x = ')' : while (top element of stack ≠ '(') do
            print top element of stack and pop stack;
        end while;
        Pop '(' from stack;
        : else : while ICP (x) ≤ ISP (top element of stack) do
            print top element of stack and pop stack;
        end while
        push x into stack;
    end case
    x: = getnextchar (E);
end while
while stack is non empty do
    print top element of stack and pop stack;
end while
end INFIX_POSTFIX_CONV.

```



Example 8.5 illustrates the conversion of an infix expression into its equivalent postfix expression using Algorithm INFIX\_POSTFIX \_CONV ( ).

**Example 8.5** Consider an infix expression  $A^*(B+C)-G$ . Table 8.6 illustrates the conversion into its postfix equivalent.

**Table 8.6** Conversion of  $A^*(B+C)-G\$$  into its postfix form

| Input character fetched by getnextchar ( ) | Work stack | Postfix expression | Remarks                                                                |
|--------------------------------------------|------------|--------------------|------------------------------------------------------------------------|
| A                                          | \$         | A                  | Print A                                                                |
| *                                          | \$*        | A                  | Since ISP(*) > ISP (\$) push * into stack.                             |
| (                                          | \$*(       | A                  | Since ICP (' ') > ISP (*) push ( into stack.                           |
| B                                          | \$*(       | AB                 | Print B.                                                               |
| +                                          | \$*(+      | AB                 | Since ICP (+) > ISP('') push + into stack.                             |
| C                                          | \$*(+      | ABC                | Print C                                                                |
| )                                          | \$*        | ABC+               | Pop elements from stack until '(' is reached. Also pop '(' from stack. |
| -                                          | \$-        | ABC+*              | Since ICP (-) < ISP (*) pop * from the stack. Push '-' into stack      |
| G                                          | \$-        | ABC+*G             | Print G                                                                |
| \$                                         | \$         | ABC+*G-            | End of input (\$) reached. Empty contents of stack.                    |

## ADT for Binary Trees

**Data objects:**

A binary tree of nodes each holding one (or more) data field(s) DATA and two link fields, LCHILD and RCHILD. T points to the root node of the binary tree.

**Operations:**

- Check if binary tree  $T$  is empty  
 $\text{CHECK\_BINTREE\_EMPTY} (T)$  (Boolean function)
- Make a binary tree  $T$  empty by setting  $T$  to NIL  
 $\text{MAKE\_BINTREE\_EMPTY} (T)$
- Move to the left subtree of a node  $X$  by moving down its LCHILD pointer  
 $\text{MOVE\_LEFT\_SUBTREE}(X)$
- Move to the right subtree of a node  $X$  by moving down its RCHILD pointer  
 $\text{MOVE\_RIGHT\_SUBTREE}(X)$
- Insert node containing element  $ITEM$  as the root of the binary tree  $T$ ; Ensure that  $T$  does not point to any node before execution  
 $\text{INSERT\_ROOT} (T, ITEM)$
- Insert node containing  $ITEM$  as the left child of node  $X$ ; Ensure that  $X$  does not have a left child node before execution  
 $\text{INSERT\_LEFT} (X, ITEM)$
- Insert node containing  $ITEM$  as the right child of node  $X$ ; Ensure that  $X$  does not have a right child node before execution  
 $\text{INSERT\_RIGHT} (X, ITEM)$
- Delete root node of binary tree  $T$ ; Ensure that the root does not have child nodes  
 $\text{DELETE\_ROOT} (T)$
- Delete node pointed to by  $X$  from the binary tree and set  $X$  to point to the left child of the node; Ensure that the node pointed to by  $X$  does not have a right child  
 $\text{DELETE\_POINT\_LEFTCHILD}(X)$
- Delete node pointed to by  $X$  from the binary tree and set  $X$  to point to the right child of the node; Ensure that the node pointed to by  $X$  does not have a left child  
 $\text{DELETE\_POINT\_RIGHTCHILD}(X)$
- Store  $ITEM$  into a node whose address is  $X$   
 $\text{STORE\_DATA}(X, ITEM)$
- Retrieve data of a node whose address is  $X$  and return it in  $ITEM$   
 $\text{RETRIEVE\_DATA}(X, ITEM)$
- Perform Inorder traversal of binary tree  $T$   
 $\text{INORDER\_TRAVERSAL}(T)$
- Perform Preorder traversal of binary tree  $T$   
 $\text{PREORDER\_TRAVERSAL}(T)$
- Perform Postorder traversal of binary tree  $T$   
 $\text{POSTORDER\_TRAVERSAL}(T)$



## Summary

- Trees and binary trees are non-linear data structures, which are inherently two dimensional in structure.
- While trees are non empty and may have nodes of any degree, a binary tree may be empty or hold nodes of degree, at most two.
- The terminologies of root node, height, level, parent, children, sibling, ancestors, leaf or terminal nodes and non-terminal nodes are applicable to both trees and binary trees.
- While trees are efficiently represented using linked representations, binary trees are represented using both array and linked representations.
- The traversals of a binary tree are inorder, postorder and preorder.
- A prudent use of null pointers in the linked representation of a binary tree yields a threaded binary tree.
- The application of binary trees has been demonstrated on expression trees and its related concepts.
- The ADT of the binary tree is presented.



## Illustrative Problems

**Problem 8.1** For the binary tree shown in Fig. I 8.1,

- Identify
  - Root
  - children of G
  - parent of D
  - siblings of Z
  - Level of C
  - Ancestors of Y
  - leaf nodes
  - height of the binary tree
- Obtain the inorder, postorder and preorder traversals of the binary tree.

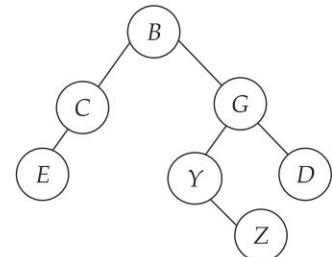
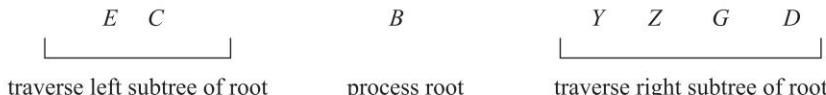


Fig. I 8.1

**Solution:**

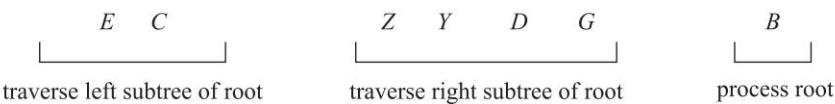
- (i) Root : B (ii) Children of G: Y,D (iii) Parent of D: G (iv) Siblings of Z: None (v) Level of C: 2 (vi) Ancestors of Y : G, B (vii) Leaf nodes : E, Z, D (viii) Height of the binary tree :4.
- (b) Inorder traversal : ECYZGD

The output of the traversal which follows the scheme of Algorithm 8.1 can be dissected as



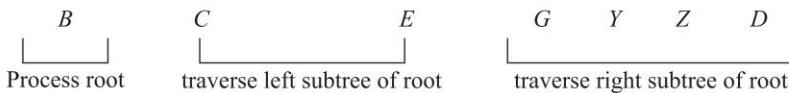
Postorder traversal : ECZYDGB.

The output of the traversal following the scheme of Algorithm 8.2 can be dissected as.



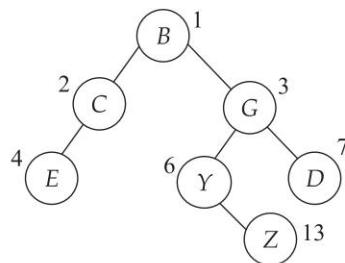
Preorder traversal : BCEGYZD

The output of the traversal following the scheme of Algorithm 8.3 can be dissected as



**Problem 8.2** Obtain an array representation and a linked representation of the binary tree shown in Fig. I 8.1.

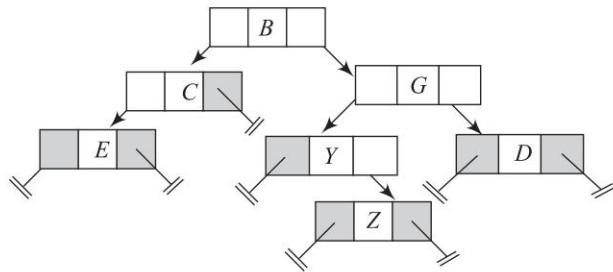
**Solution:** To obtain the array representation we first number the nodes of the binary tree akin to that of a complete binary tree, as shown below:



The array representation is given as:

|     |     |     |     |     |     |     |     |     |      |      |      |      |     |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|------|------|------|------|-----|
| $B$ | $C$ | $G$ | $E$ |     | $Y$ | $D$ |     |     |      |      |      |      | $Z$ |
| [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9] | [10] | [11] | [12] | [13] |     |

The linked representation is given as:



**Problem 8.3** A binary tree  $T$  has 9 nodes. The inorder and preorder traversals of  $T$  yield the following:

Inorder traversal (I) :     $E$        $A$        $C$        $K$        $F$        $H$        $D$        $B$        $G$   
 Preorder traversal (P) :     $F$        $A$        $E$        $K$        $C$        $D$        $H$        $G$        $B$

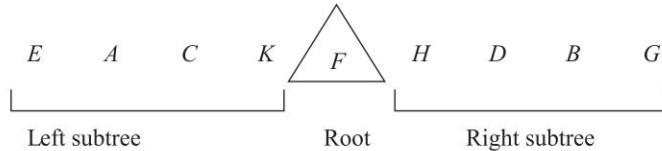
Draw the binary tree  $T$ .

**Solution:** The key to the solution of this problem is the observation that the first occurring node in a preorder traversal is the root of the binary tree and that, once the root is known, the nodes forming the left and the right subtrees can be extracted from the Inorder traversal list. Application

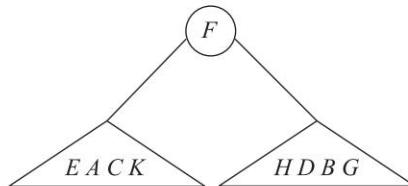
of this key to each of the left and right subtree by obtaining their respective roots from the preorder traversal and moving on to inorder traversal to obtain the nodes forming the sub-subtrees can eventually lead to the tracing of the binary tree.

From P: Root of the binary tree is  $F$ .

From I : the nodes forming the left and right subtrees of  $F$  are

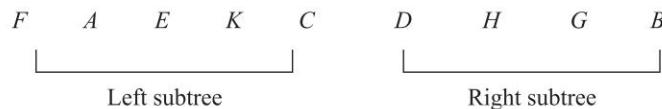


The binary tree can be roughly traced as shown below:

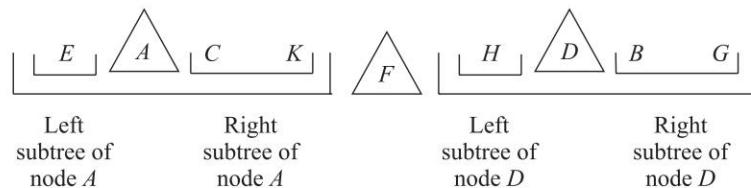


In the next step we proceed to obtain the structure of the left and right subtrees.

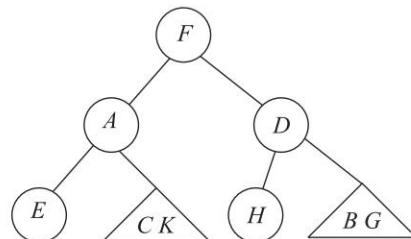
From P : Root of the left subtree is  $A$  and root of the right subtree is  $D$



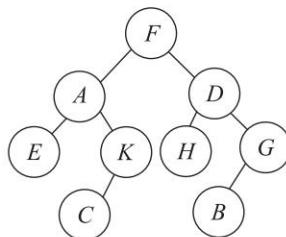
From I : the nodes forming the left and right sub-subtrees are



Tracing the binary tree yields,



Proceeding in a similar fashion, we obtain the roots of the subtrees  $\{C, K\}$  and  $\{B, G\}$  to be  $K$  and  $G$  respectively. The final trace yields the binary tree:



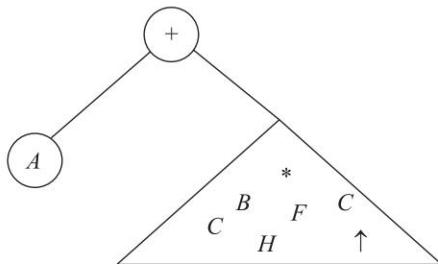
**Example 8.4** Make use of the infix and postfix expressions given below to trace the corresponding expression tree

Infix :  $A + B * C / F \uparrow H$

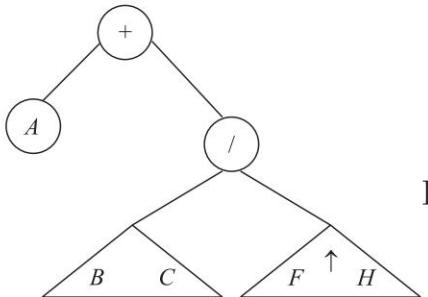
Postfix :  $A B C * F H \uparrow / +$

**Solution:** The key to the problem is similar to the one discussed in Illustrative Problem 8.3 but for the difference that the root node to be picked from the postfix expression is the last occurring node.

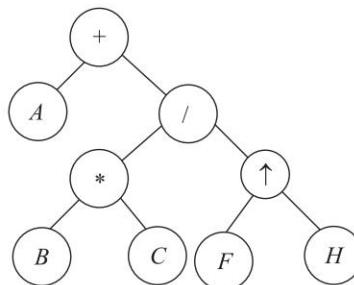
Thus the expression tree traced in the first step is:



In the next step the expression tree traced would be



Progressing in this way, the final expression tree is obtained as shown in the adjacent figure.



**Note:** Though the expression tree could be easily traced from infix expression alone, the objective of the problem is to emphasize the fact that a binary tree can be traced from its inorder and postorder traversals as well.

**Example 8.5** What does the following pseudocode procedure do to the binary tree given in Fig. I 8.5, when invoked as `WHAT_DO_I_DO (THIS)`?

```

procedure WHAT_DO_I_DO (HERE)
    if HERE ≠ NIL then
        { call WHAT_DO_I_DO(LCHILD (HERE));
          if ( LCHILD (HERE) = NIL) and (RCHILD (HERE) = NIL)
          then print DATA (HERE) ;
          call WHAT_DO_I_DO (RCHILD(HERE)) ;
        }
    end WHAT_DO_I_DO.
  
```

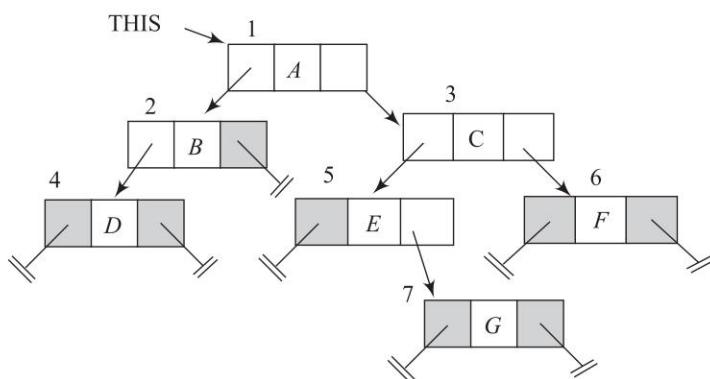
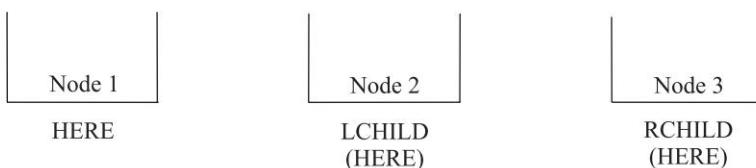


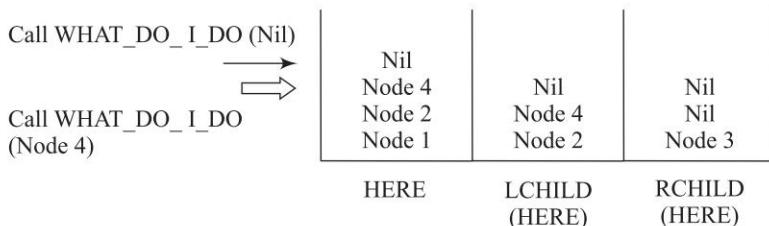
Fig. I 8.5

**Solution:** We trace the recursive procedure using a stack and for convenience of representing the nodes in the stack, have numbered the nodes from 1 to 7. For every call of `WHAT_DO_I_DO()` we keep track of HERE, LCHILD (HERE) and RCHILD (HERE).

The first call of `WHAT_DO_I_DO (THIS)` results in the following snap shot of the stack :



In the subsequent calls the snapshot of the stack is given by



when `HERE = NIL` that call of `WHAT_DO_I_DO (HERE)` (marked  $\rightarrow$ ) terminates and the control returns to the previous call viz., `WHAT_DO_I_DO (Node 4)` (marked  $\Rightarrow$ ). Here `LCHILD (HERE) = RCHILD (HERE) = NIL`. Hence `DATA(Node 4)` viz., D is printed. Now the control moves further to invoke the call `WHAT_DO_I_DO (RCHILD(Node 4))` that is `WHAT_DO_I_DO (NIL)` which again terminates. Now the control returns to the call `WHAT_DO_I_DO (Node 2)` and so on. It is easy to see that `WHAT_DO_I_DO (THIS)` prints the data fields of all leaf nodes of the binary tree. Hence the output is D, G and F.

**Example 8.6** Show that the maximum number of nodes in a binary tree of height  $h$  is  $2^h - 1$ ,  $h \geq 1$ .

**Solution:** It is known that the maximum number of nodes in level  $i$  of a binary tree is  $2^{i-1}$ . Given the height of the binary tree to be  $h$  which is the maximum level, the maximum number of nodes is given by

$$\sum_{i=1}^h 2^{i-1} = 1 + 2 + 2^2 + \dots + 2^{h-1} = 2^h - 1$$

**Example 8.7** Show that for a non-empty binary tree  $T$  if  $n_o$  is the number of leaf nodes,  $n_2$  the number of nodes of degree 2 then  $n_o = n_2 + 1$ .

**Solution:** Let  $n$  be the number of nodes in a non empty binary tree and let  $n_1$  be the number of nodes of degree 1.

Now,

$$n = n_o + n_1 + n_2 \quad \dots(i)$$

Again if  $b$  is the number of links or branches in the binary tree, all nodes except the root node hang from a branch yielding the relation

$$b = n - 1 \quad \dots(ii)$$

Also, each branch emanates from a node whose degree is either 1 or 2. Hence,

$$b = n_1 + 2.n_2 \quad \dots \text{(iii)}$$

Subtracting (iii) from (ii) yields.

$$n = n_1 + 2n_2 + 1 \quad \dots \text{(iv)}$$

From (iv) and (i) we obtain

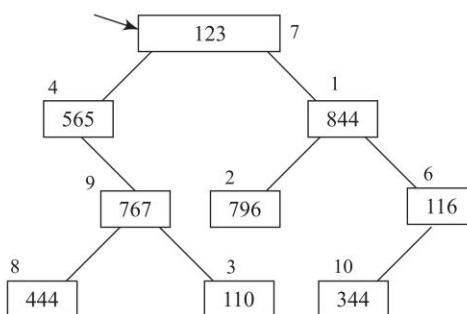
$$n_0 = n_2 + 1$$

**Example 8.8** A binary tree is stored in the memory of a computer as shown below. Trace the structure of the binary tree.

|    | LCHILD | DATA | RCHILD |  |
|----|--------|------|--------|--|
| 1  | 2      | 844  | 6      |  |
| 2  | 0      | 796  | 0      |  |
| 3  | 0      | 110  | 0      |  |
| 4  | 0      | 565  | 9      |  |
| 5  | 12     | 444  | 0      |  |
| 6  | 10     | 116  | 0      |  |
| 7  | 4      | 123  | 1      |  |
| 8  | 0      | 444  | 0      |  |
| 9  | 8      | 767  | 3      |  |
| 10 | 0      | 344  | 0      |  |

Root : 7

**Solution:** Given the root node's address to be 7 we begin tracing the binary tree from the root onwards. The binary tree is given by:



**Example 8.9** Outline a linked representation for the tree and threaded binary tree representation for the binary tree shown in Fig. I 8.9(a) and (b), respectively.

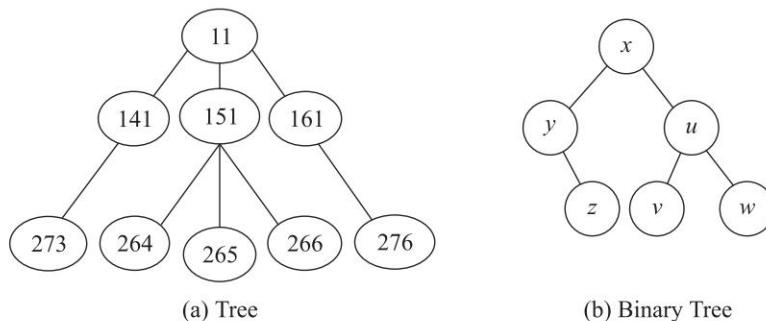
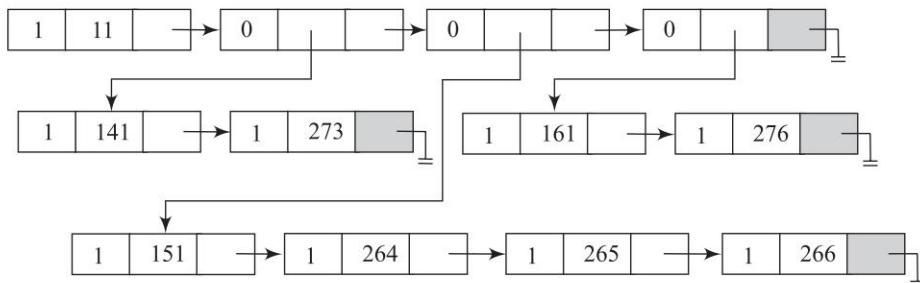
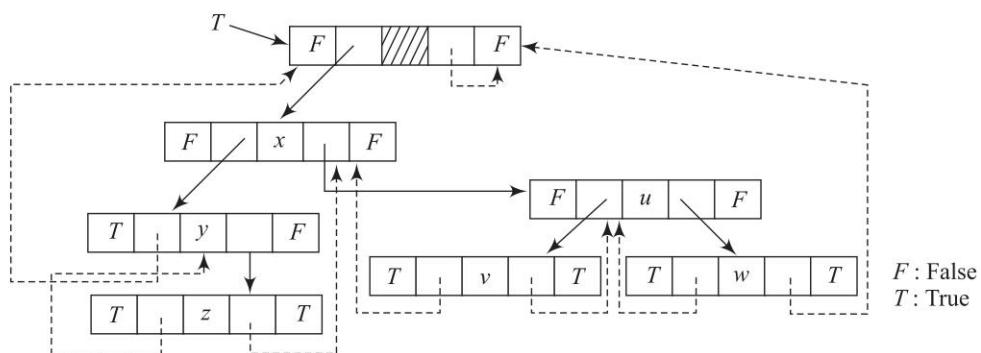


Fig. I 8.9

**Solution:** Following the node structure of TAG, DATA/DOWNLINK, LINK illustrated in Sec. 8.3 the linked representation of the tree is given by



The threaded binary representation of Fig. I 8.9(b) is illustrated below and is obtained by following the node structure detailed in Sec. 8.7.



The inorder traversal sequence to be tracked by the threads is :  $y \ z \ x \ v \ u \ w$ . The threads are linked to the appropriate inorder successors and predecessors.

**Example 8.10** For the binary tree  $T$  given in Fig. I 8.10 obtain (i) a one-way inorder threading of  $T$  and (ii) one-way preorder threading of  $T$ .

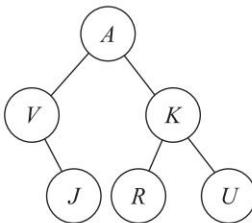
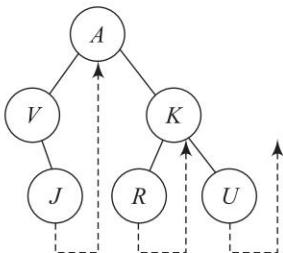


Fig. I 8.10

*Solution:*

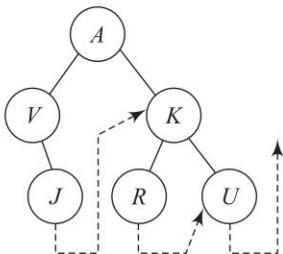
- (i) The inorder traversal of the binary tree  $T$  yields:  $V \ J \ A \ R \ K \ U$

A one-way threading of  $T$  is obtained by replacing the RCHILD links of the nodes, which are null, by threads pointing to the inorder successor of the node. Thus, the RCHILD link of  $J$  points to  $A$ , that of  $R$  points to  $K$  and that of  $U$  is either kept dangling (or if there is a head node points to the same). The threaded binary tree is shown below:



- (ii) The preorder traversal of binary  $T$  yields:  $A \ V \ J \ K \ R \ U$ .

For the one-way preorder threading, the RCHILD links of the nodes, which are null, are set to point to the preorder successors of the node. Thus, the RCHILD link of  $J$  points to  $K$ , that of  $R$  points to  $U$  and the same of  $U$  is a dangling thread or may be connected to the head node if available. The threaded tree for the same is shown below:





## Review Questions

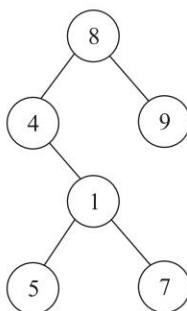
1. Which among the following is not a property of a tree?
  - (i) There is a specially designated node called the root
  - (ii) The rest of the nodes could be partitioned into  $t$  disjoint sets ( $t \geq 0$ ) each set representing a tree  $T_i$ ,  $i = 1, 2, \dots, t$  known as subtree of the tree.
  - (iii) Any node should be reachable from anywhere in the tree
  - (iv) At most one cycle could be present in the tree
 

|         |          |           |          |
|---------|----------|-----------|----------|
| (a) (i) | (b) (ii) | (c) (iii) | (d) (iv) |
|---------|----------|-----------|----------|
2. The maximum number of nodes in a binary tree of depth  $k$  is
 

|               |                     |               |                   |
|---------------|---------------------|---------------|-------------------|
| (a) $2^{k-1}$ | (b) $2^{(k+1)} - 1$ | (c) $2^{k-1}$ | (d) $2^{(k+1)-1}$ |
|---------------|---------------------|---------------|-------------------|
3. For a binary tree of  $2.k$  nodes,  $k > 1$ , the number of pointers and the number of null pointers that the tree would use for its representation is respectively given by
 

|                   |                         |                         |                         |
|-------------------|-------------------------|-------------------------|-------------------------|
| (a) $k$ and $k+1$ | (b) $2.k$ and $2.k + 1$ | (c) $4.k$ and $4.k + 1$ | (d) $4.k$ and $2.k + 1$ |
|-------------------|-------------------------|-------------------------|-------------------------|
4. An inorder and postorder traversal of a binary tree was 'claimed' to yield the following sequence:  
 Inorder traversal : HAT GLOVE SOCKS SCARF GLASSES  
 Post order traversal : GLOVE SCARF HAT GLASSES SOCKS  
 What are your observations?  
  - (i) HAT is the root of the binary tree
  - (ii) SOCKS is the root of the binary tree
  - (iii) the binary tree is a skewed binary tree
  - (iv) the traversals are incorrect
 

|         |          |           |          |
|---------|----------|-----------|----------|
| (a) (i) | (b) (ii) | (c) (iii) | (d) (iv) |
|---------|----------|-----------|----------|
5. Which of the following observations with regard to binary tree traversals is incorrect?
  - (i) Given a preorder traversal of a binary tree, the root node is the first occurring item in the list.
  - (ii) Given a postorder traversal of a binary tree, the root node is the last occurring item in the list.
  - (iii) Inorder traversal does not directly reveal the root node of the binary tree.
  - (iv) To trace back the structure of the binary tree, inorder, postorder and preorder traversal sequences are needed.
6. Sketch (i) an array representation and (ii) a linked list representation for the following binary tree:



7. Sketch a linked representation for a threaded binary tree equivalent of the binary tree shown in Review Questions 6 (Chapter 8).
8. Obtain inorder and post order traversals for the binary tree shown in Review Questions 6 (Chapter 8).
9. Draw an expression tree for the following logical expression:  
 $p \text{ and } (q \text{ or not } k) \text{ and } (s \text{ or } b \text{ or } h)$
10. Undertake post order traversal of the expression tree obtained in Review Questions 9 (Chapter 8) and compare it with the hand computed postfix form of the logical expression.
11. Given the following inorder and preorder traversals, trace the binary tree.  
 Inorder traversal :  $B F G H P R S T W Y Z$   
 Preorder traversal :  $P F B H G S R Y T W Z$
12. Making use of Algorithm 8.4, convert the following infix expression to its equivalent postfix form and evaluate the postfix expression for the specified values:  

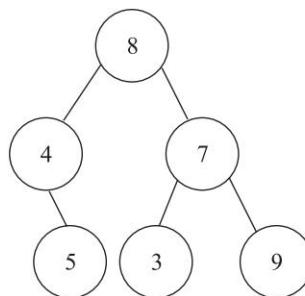
$$(x + y + z) \uparrow (a + b) - g * n * m + r$$

$$x = 1, y = 2, z = -1, a = 1, b = 2, g = 5, n = 2, m = 1, r = 7$$



## Programming Assignments

1. Write a program to input a binary tree implemented as a linked representation. Execute Algorithms 8.1–8.3 to perform inorder, postorder and preorder traversals of the binary tree.
2. Implement Algorithm 8.4 to convert an infix expression into its postfix form.
3. Write a recursive procedure to count the number of nodes in a binary tree.
4. Implement a threaded binary tree. Write procedures to insert a node NEW to the left of node NODE when
  - (i) the left subtree of NODE is empty, and
  - (ii) the left subtree of NODE is non-empty.
5. Write non-recursive procedures to perform the inorder, postorder and preorder traversals of a binary tree.
6. **Level order traversal:** It is a kind of binary tree traversal where elements in the binary tree are traversed by levels, top to bottom and within levels, left to right. Write a procedure to execute the level order traversal of a binary tree.(Hint: Use a Queue data structure)  
 Example: Level order traversal of the following binary tree is: 8 4 7 5 3 9



7. Implement the ADT of a binary tree in a language of your choice. Include operations to
  - (i) obtain the height of a binary tree and (ii) the list of leaf nodes.

## CHAPTER



## GRAPHS

## 9

In Chapter 8 we introduced trees and graphs as examples of non linear data structures. To recall, non-linear data structures unlike linear data structures which are uni dimensional in structure (for example arrays), are inherently two dimensional in structure.

Though in the field of computer science, trees have been recognized as efficient non linear data structures with their own set of terminologies and concepts to suit the needs of the digital computer, graph theory which has emerged as an independent field, encompasses studies on trees as well. In other words, in the field of graph theory, a tree is a special kind of graph holding a definition which in principle agrees with that of a tree data structure, but is devoid of most of the terminologies and concepts tagged to it from the view point of data structures. This distinction needs to be borne in mind when one defines a tree- rather 'redefines' tree as a special kind of graph in this chapter.

Though graph theory has turned out to be a vast area with innumerable applications, we restrict the scope of this chapter to introducing graphs as effective data structures only. Hence only those concepts and terminologies needed to promote this aspect of graphs are dealt with.

## Introduction

## 9.1

The history of graphs dates back to 1736 in what is now referred to as the classical *Koenigsberg bridge problem*. In the town of Koenigsberg in Eastern Prussia, the island of Kneiphof existed in the middle of the river Pregel. The river bifurcated itself bordering the land areas as shown in Fig. 9.1. There were seven bridges connecting the land areas as shown in the figure. The problem was to find if the people of the town could walk on the seven bridges once only, starting from any land area, and returning to the starting land area after traversing all the bridges.

An example walk is listed below:

Start from the land area  $P$ -traverse bridge1; land area  $R$ -traverse bridge 3; land area  $Q$ -traverse bridge 4; land area  $R$ -traverse bridge 5; land area  $S$ -traverse bridge 7; land area  $Q$ -traverse bridge 7; land area  $S$ -traverse bridge 6; land area  $P$ -traverse bridge 2; land area  $R$ .

This walk, neither does it traverse all bridges once only nor does it reach its starting point which is land area  $P$ .

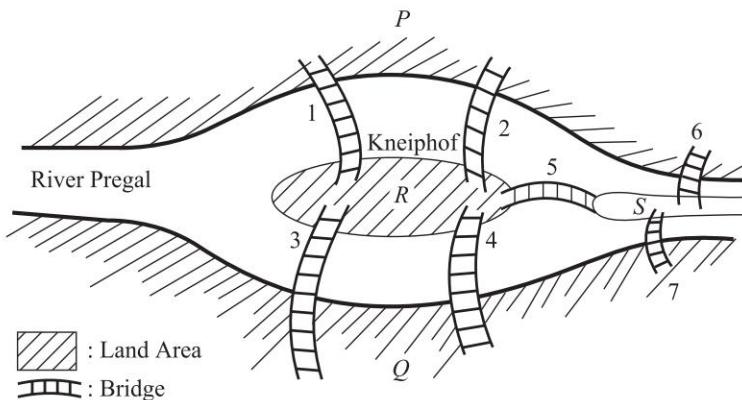
9.1 *Introduction*

9.2 *Definitions and Basic Terminologies*

9.3 *Representations of Graphs*

9.4 *Graph Traversals*

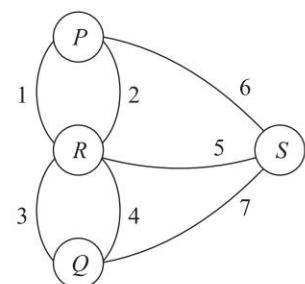
9.5 *Applications*



**Fig. 9.1** The Koenigsberg bridge problem

It was left to Euler to solve the puzzle of Koenigsberg bridge problem by stating that there is no way that people could walk across the bridges once only and return to the starting point. The solution to the problem was arrived at by representing the land areas as circles called *vertices* and bridges as arcs called *links* or *edges* connecting the circles. Defining the *degree of a vertex* to be the number of arcs converging on it, or in other words, the number of bridges which descend on a land area, Euler showed that *a walk is possible only when all the vertices have even degree*. That is, every land area needs to have only even number of bridges descending on it. In the case of the Koenigsberg bridge problem, all the vertices turned out to have an *odd degree*. Figure 9.2 illustrates the graph representation of the Koenigsberg bridge problem. This vertex-edge representation is what came to be known as a *graph* (here it is a *multigraph*). The walk which beginning from a vertex and returning to it after traversing all edges in the graph came to be known as an *Eulerian walk*.

Since this first application, graph theory has grown *in leaps and bounds* to encompass a wide range of applications in the fields of cybernetics, electrical sciences, genetics and linguistics, to quote a few.



**Fig. 9.2** Graph representation of the Koenigsberg bridge problem

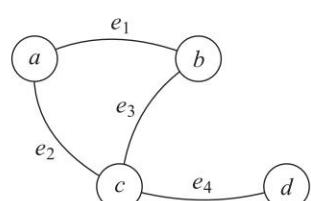
## Definitions and Basic Terminologies

## 9.2

### Graph

A *graph*  $G = (V, E)$  consists of a finite non empty set of *vertices*  $V$  also called *points* or *nodes* and a finite set  $E$  of unordered pairs of distinct vertices called *edges* or *arcs* or *links*.

**Example** Figure 9.3 illustrates a graph. Here  $V = \{a, b, c, d\}$  and  $E = \{(a, b), (a, c), (b, c), (c, d)\}$ . However it is convenient to represent edges using labels as shown in the figure.



**Fig. 9.3** A graph

$V$  : Vertices : { $a, b, c, d$ }

$E$  : Edges : { $e_1, e_2, e_3, e_4$ }

A graph  $G = (V, E)$  where  $E = \emptyset$ , is called as a *null* or *empty graph*. A graph with one vertex and no edges is called a *trivial graph*.

## Multigraph

A *multigraph*  $G = (V, E)$  also consists of a set of vertices and edges except that  $E$  may contain *multiple edges* (i.e.) edges connecting the same pair of vertices, or may contain *loops* or *self edges* (i.e.) an edge whose end points are the same vertex.

**Example** Figure 9.4 illustrates a multigraph

Observe the multiple edges  $e_1, e_2$  connecting vertices  $a, b$  and  $e_5, e_6, e_7$  connecting vertices  $c, d$  respectively. Also note the self edge  $e_4$ .

However, it has to be made clear that graphs do not contain multiple edges or loops and hence are different from multigraphs. The definitions and terminologies to be discussed in this section are applicable only to graphs.

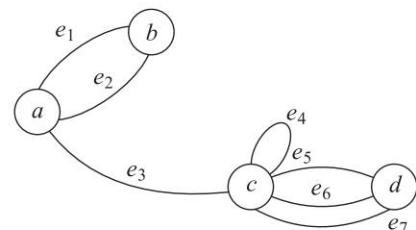


Fig. 9.4 A multigraph

## Directed and undirected graphs

A graph whose definition (stated in Sec. 9.2) makes reference to *unordered pairs of vertices* as edges is known as *an undirected graph*. The edge  $e_{ij}$  of such an undirected graph is represented as  $(v_i, v_j)$  where  $v_i, v_j$  are distinct vertices. Thus an undirected edge  $(v_i, v_j)$  is equivalent to  $(v_j, v_i)$ .

On the other hand, *directed graphs* or *digraphs* make reference to edges which are directed (i.e.) edges which are *ordered pairs of vertices*. The edge  $e_{ij}$  is referred to as  $\langle v_i, v_j \rangle$  which is distinct from  $\langle v_j, v_i \rangle$  where  $v_i, v_j$  are distinct vertices. In  $\langle v_i, v_j \rangle$ ,  $v_i$  is known as *tail* of the edge and  $v_j$  as the *head*.

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

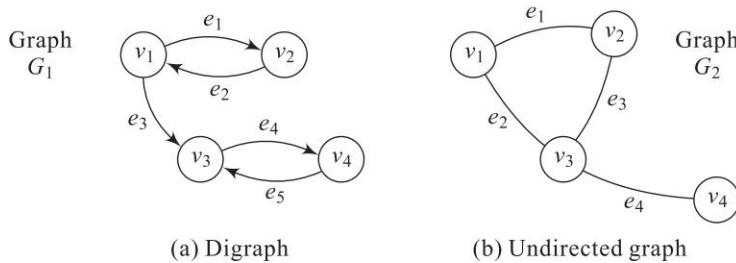


Fig. 9.5 A digraph and an undirected graph

In Fig. 9.5(a),  $e_1$  is a directed edge between  $v_1$  and  $v_2$ , (i.e.)  $e_1 = \langle v_1, v_2 \rangle$ , whereas in Fig. 9.5(b)  $e_1$  is an undirected edge between  $v_1$  and  $v_2$ , (i.e.)  $e_1 = (v_1, v_2)$ .

The list of vertices and edges of graphs  $G_1$  and  $G_2$  are:

Vertices ( $G_1$ ) :  $\{v_1, v_2, v_3, v_4\}$

Vertices ( $G_2$ ) :  $\{v_1, v_2, v_3, v_4\}$

Edges ( $G_1$ ) :  $\{\langle v_1, v_2 \rangle, \langle v_2, v_1 \rangle, \langle v_1, v_3 \rangle, \langle v_3, v_4 \rangle, \langle v_4, v_3 \rangle\}$  or  $\{e_1, e_2, e_3, e_4, e_5\}$

Edges ( $G_2$ ) :  $\{(v_1, v_2), (v_1, v_3), (v_2, v_3), (v_3, v_4)\}$  or  $\{e_1, e_2, e_3, e_4\}$

In the case of an undirected edge  $(v_i, v_j)$  in a graph, the vertices  $v_i, v_j$  are said to be *adjacent* or the edge  $(v_i, v_j)$  is said to be *incident on vertices*  $v_i, v_j$ . Thus in Fig. 9.5(b) vertices  $v_1, v_3$  are adjacent to vertex  $v_2$  and edges  $e_1: (v_1, v_2), e_3: (v_2, v_3)$  are incident on vertex  $v_2$ .

On the other hand, if  $\langle v_i, v_j \rangle$  is a directed edge, then  $v_i$  is said to be *adjacent to*  $v_j$  and  $v_j$  is said to be *adjacent from*  $v_i$ . The edge  $\langle v_i, v_j \rangle$  is *incident* to both  $v_i$  and  $v_j$ . Thus in Fig. 9.5(a) vertices  $v_2$  and  $v_3$  are adjacent from  $v_1$ , and  $v_1$  is adjacent to vertices  $v_2$  and  $v_3$ . The edges incident to vertex  $v_3$  are  $\langle v_1, v_3 \rangle, \langle v_3, v_4 \rangle$  and  $\langle v_4, v_3 \rangle$ .

## Complete graphs

The number of distinct unordered pairs  $(v_i, v_j)$ ,  $v_i \neq v_j$  in a graph with  $n$  vertices is  ${}^n C_2 = \frac{n \cdot (n - 1)}{2}$

An  $n$  vertex undirected graph with exactly  $\frac{n \cdot (n - 1)}{2}$  edges is said to be *complete*.

**Example** Figure 9.6 illustrates a complete graph. The undirected graph with 4 vertices has all its  ${}^4 C_2 = 6$  edges intact.

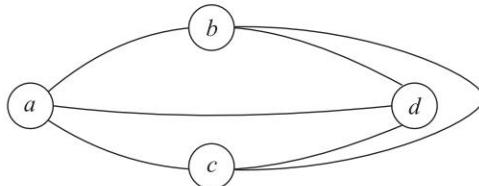


Fig. 9.6 A complete graph

In the case of a digraph with  $n$  vertices, the maximum number of edges is given by  ${}^n P_2 = n \cdot (n - 1)$ . Such a graph with exactly  $n \cdot (n - 1)$  edges is said to be a *complete digraph*.

**Example** Figure 9.7(a) illustrates a digraph which is complete and Fig. 9.7(b) a graph which is not complete.

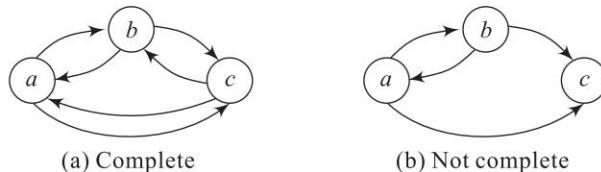
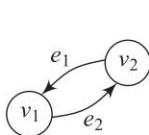
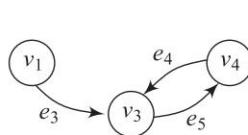
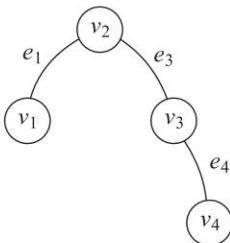
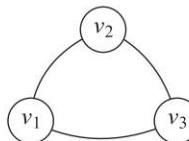


Fig. 9.7 Digraphs which are complete and not complete

## Subgraph

A *subgraph*  $G' = (V', E')$  of a graph  $G = (V, E)$  is such that  $V' \subseteq V$  and  $E' \subseteq E$ .

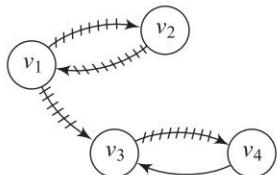
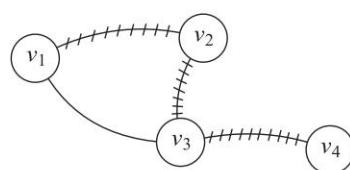
**Example** Figure 9.8 illustrates some subgraphs of the directed and undirected graphs shown in Fig. 9.5 (Graphs  $G_1$  and  $G_2$ )

(a) Subgraphs of  $G_1$ (b) Subgraphs of  $G_2$ (c) Subgraphs of  $G_2$ **Fig. 9.8** Subgraphs of graphs  $G_1$  and  $G_2$  (Fig. 9.5)

## Path

A *path* from a vertex  $v_i$  to vertex  $v_j$  in an undirected graph  $G$  is a sequence of vertices  $v_i, v_{l_1}, v_{l_2}, \dots, v_{l_k}, v_j$  such that  $(v_i, v_{l_1}), (v_{l_1}, v_{l_2}), \dots, (v_{l_k}, v_j)$  are edges in  $G$ . If  $G$  is directed then the path from  $v_i$  to  $v_j$  more specially known as a *directed path* consists of edges  $\langle v_i, v_{l_1} \rangle, \langle v_{l_1}, v_{l_2} \rangle, \dots, \langle v_{l_k}, v_j \rangle$  in  $G$ .

**Example** Figure 9.9(a) illustrates a path  $P_1$  from vertex  $v_1$  to  $v_4$  in graph  $G_1$  of Fig. 9.5(a) and Fig. 9.9(b) illustrates a path  $P_2$  from vertex  $v_1$  to  $v_4$  of graph  $G_2$  of Fig. 9.5(b).

(a) A path from  $v_1$  to  $v_4$  in directed graph  $G_1$   
 $P_1 = \{v_1, v_2, v_1, v_3, v_4\}$ (b) A path from  $v_1$  to  $v_4$  in undirected graph  $G_2$   
 $P_2 = \{v_1, v_2, v_3, v_4\}$ **Fig. 9.9** Path between vertices of a graph (Fig. 9.5)

The *length* of a path is the number of edges on it.

**Example** In Fig. 9.9 the length of path  $P_1$  is 4 and the length of path  $P_2$  is 3.  
A *simple path* is a path in which all the vertices except possibly the first and last vertices are distinct.

**Example** In graph  $G_2$  (Fig. 9.5(b)), the path from  $v_1$  to  $v_4$  given by  $\{(v_1, v_2), (v_2, v_3), (v_3, v_4)\}$  and written as  $\{v_1, v_2, v_3, v_4\}$  is a simple path where as the path from  $v_3$  to  $v_4$  given by  $\{(v_3, v_1), (v_1, v_2), (v_2, v_3), (v_3, v_4)\}$  and written as  $\{v_3, v_1, v_2, v_3, v_4\}$  is not a simple path but a path due to the repetition of vertices.

Also in graph  $G_1$  (Fig. 9.5(a)) the path from  $v_1$  to  $v_3$  given by  $\{<v_1, v_2>, <v_2, v_1>, <v_1, v_3>\}$  written as  $\{v_1, v_2, v_1, v_3\}$  is not a simple path but a mere path due to the repetition of vertices. However, the path from  $v_2$  to  $v_4$  given by  $\{<v_2, v_1>, <v_1, v_3>, <v_3, v_4>\}$  written as  $\{v_2, v_1, v_3, v_4\}$  is a simple path.

A *cycle* is a simple path in which the first and last vertices are the same. A cycle is also known as a *circuit*, *elementary cycle*, *circular path* or *polygon*.

**Example** In graph  $G_2$  (Fig. 9.5(b)) the path  $\{v_1, v_2, v_3, v_1\}$  is a cycle. Also, in graph  $G_1$  (Fig. 9.5(a)) the path  $\{v_1, v_2, v_1\}$  is a cycle or more specifically a *directed cycle*.

## Connected graphs

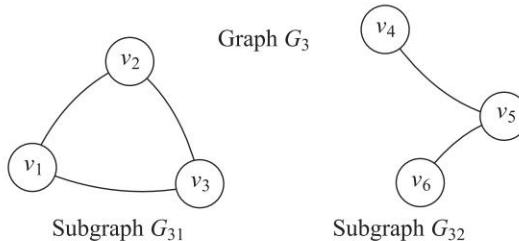
Two vertices  $v_i, v_j$  in a graph  $G$  are said to be *connected* only if there is a path in  $G$  between  $v_i$  and  $v_j$ . In an undirected graph if  $v_i$  and  $v_j$  are connected then it automatically holds that  $v_j$  and  $v_i$  are also connected.

An undirected graph is said to be a *connected graph* if every pair of distinct vertices  $v_i, v_j$  are connected.

**Example** Graph  $G_2$  (Fig. 9.5(b)) is connected where as graph  $G_3$  shown in Fig. 9.10 is not connected.

In the case of an undirected graph which is not connected, the maximal *connected subgraph* is called as a *connected component* or simply a *component*.

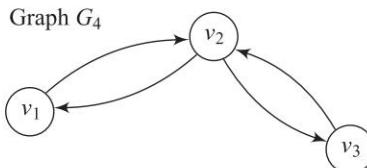
**Example** Graph  $G_3$  (Fig. 9.10) has two connected components viz., graph  $G_{31}$  and  $G_{32}$ .



**Fig. 9.10** An undirected graph with two connected components

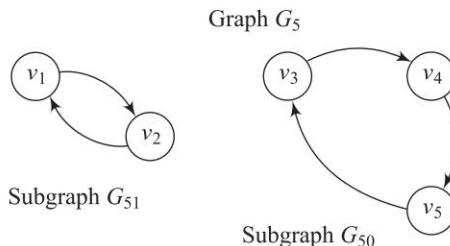
A directed graph is said to be *strongly connected* if every pair of distinct vertices  $v_i, v_j$  are connected (by means of a directed path). Thus if there exists a directed path from  $v_i$  to  $v_j$  then there also exists a directed path from  $v_j$  to  $v_i$ .

**Example** Graph  $G_4$  shown in Fig. 9.11 is strongly connected.



**Fig. 9.11** A strongly connected graph

However, the digraph shown in Fig. 9.12 is not strongly connected but is said to possess two *strongly connected components*. A strongly connected component is a maximal subgraph that is strongly connected.



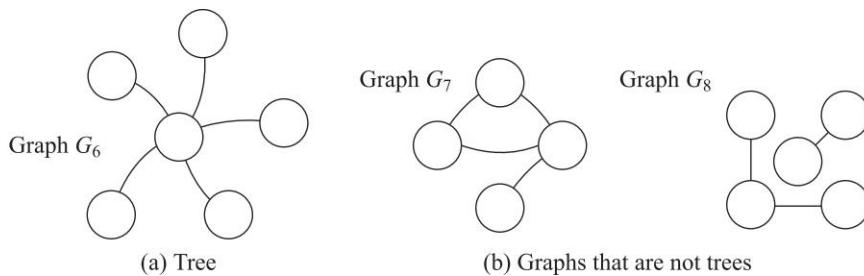
**Fig. 9.12** Strongly connected components of a digraph

## Trees

A *tree* is defined to be a connected acyclic graph. The following properties are satisfied by a tree:

- (i) There exists a path between any two vertices of the tree, and
- (ii) No cycles must be present in the tree. In other words, trees are acyclic.

**Example** Figure 9.13(a) illustrates a tree. Figure 9.13(b) illustrates graphs which are not trees due to the violation of the property of acyclicity and connectedness respectively.



**Fig. 9.13** Graphs which are trees and not trees

Note the marked absence of any hierarchical structure and its allied terminologies of parent, child, sibling, ancestor, level etc insisted upon in the tree data structure. However, both the definitions of trees—as a data structure and a type of graph—agree on the principles of connectedness and acyclicity.

## Degree

The *degree* of a vertex in an undirected graph is the number of edges incident to that vertex. A vertex with degree one is called as a *pendant vertex* or *end vertex*. A vertex with degree zero and hence has no incident edges is called an *isolated vertex*.

**Example** In graph  $G_2$  (Fig. 9.5(b)) the degree of vertex  $v_3$  is 3 and that of vertex  $v_2$  is 2. In the case of digraphs, we define the *indegree* of a vertex  $v$  to be the number of edges with  $v$  as the head and the *outdegree* of a vertex to be number of edges with  $v$  as the tail.

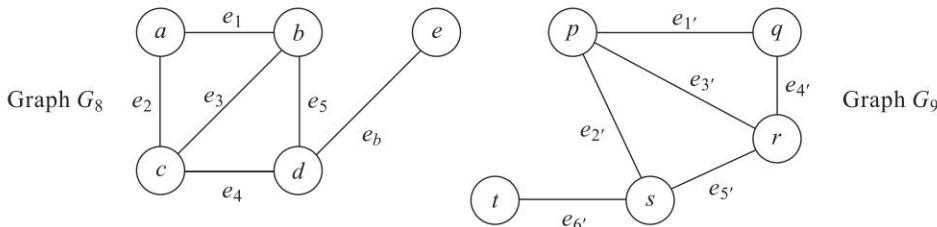
**Example** In graph  $G_1$  (Fig. 9.5(a)) the indegree of vertex  $v_3$  is 2 and the out degree of vertex  $v_4$  is 1.

## Isomorphic graphs

Two graphs are said to be *isomorphic* if,

- (i) they have the same number of vertices
- (ii) they have the same number of edges
- (iii) they have an equal number of vertices with a given degree

**Example** Figure 9.14 illustrates two graphs which are isomorphic.



**Fig. 9.14** Isomorphic graphs

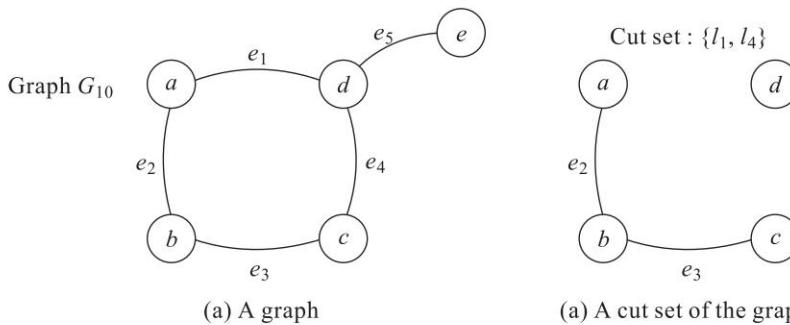
The property of isomorphism can be verified on the lists of vertices and edges of the two graphs  $G_8$  and  $G_9$  when superimposed as shown below:

|                          |        |        |        |        |        |        |
|--------------------------|--------|--------|--------|--------|--------|--------|
| Vertices ( $G_8$ ) :     | $a$    | $b$    | $c$    | $d$    | $e$    |        |
|                          | ↑      | ↑      | ↑      | ↑      | ↑      |        |
| Vertices ( $G_9$ ) :     | $q$    | $p$    | $r$    | $s$    | $t$    |        |
| Degree of the vertices : | 2      | 3      | 3      | 3      | 1      |        |
| Edges ( $G_8$ ) :        | $e_1$  | $e_2$  | $e_3$  | $e_4$  | $e_5$  | $e_6$  |
| Edges ( $G_9$ ) :        | $e'_1$ | $e'_4$ | $e'_3$ | $e'_2$ | $e'_5$ | $e'_6$ |

## Cut set

A *cut set* in a connected graph  $G$  is the set of edges whose removal from  $G$  leaves  $G$  disconnected, provided the removal of no proper subset of these edges disconnects the graph  $G$ . Cut sets are also known as *proper cut set* or *cocycle* or *minimal cut set*.

**Example** Figure 9.15 illustrates the cut set of the graph  $G_{10}$ . The cut set  $\{e_1, e_4\}$  disconnects the graph into two components as shown in the figure.  $\{e_5\}$  is also another cut set of the graph.

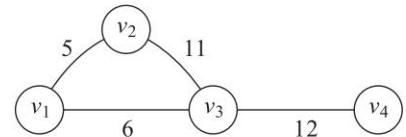


**Fig. 9.15** A cut set of a graph

## Labeled graphs

A graph  $G$  is called a *labeled graph* if its edges and / or vertices are assigned some data. In particular if the edge  $e$  is assigned a non negative number  $l(e)$  then it is called the *weight* or *length* of the edge  $e$ .

**Example** Figure 9.16 illustrates a labeled graph. A graph with weighted edges is also known as a *network*.



**Fig. 9.16** A labeled graph

A walk starting at any vertex going through each edge exactly once and terminating at the start vertex is called an *Eulerian walk* or *Euler line*.

The Koenigsberg bridge problem was in fact a problem of obtaining an Eulerian walk for the graph concerned. The solution to the problem discussed in Sec. 9.1 can be rephrased as, an Eulerian walk is possible only if the degree of each vertex in the graph is even.

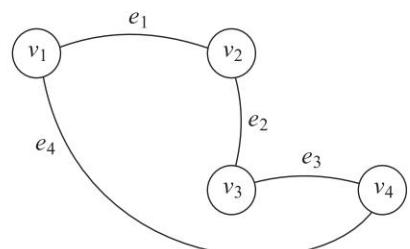
Given a connected graph  $G$ ,  $G$  is an *Euler graph* iff all the vertices are of even degree.

**Example** Figure 9.17 illustrates an Euler graph.  $\{e_1, e_2, e_3, e_4\}$  shows a Eulerian walk. The even degree of the vertices may be noted.

## Hamiltonian circuit

A *Hamiltonian circuit* in a connected graph is defined as a closed walk that traverses every vertex of  $G$  exactly once, except of course the starting vertex at which the walk terminates.

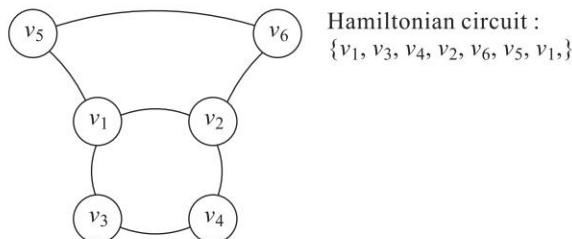
A *circuit* in a connected graph  $G$  is said to be *Hamiltonian* if it includes every vertex of  $G$ . If any edge is removed from a Hamiltonian circuit then what remains is referred to as a *Hamiltonian path*. Hamiltonian path traverses every vertex of  $G$ .



**Fig. 9.17** An Euler graph

**Example**

Figure 9.18 illustrates a Hamiltonian circuit.



**Fig. 9.18** A Hamiltonian circuit

## Representations of Graphs

## 9.3

The representation of graphs in a computer can be categorized as (i) *sequential representation* and (ii) *linked representation*. Of the two, though sequential representation has several methods, all of them follow a matrix representation thereby calling for their implementation using arrays.

The linked representation of a graph makes use of a singly linked list as its fundamental data structure.

### Sequential representation of graphs

The sequential or the matrix representation of graphs have the following methods:

- (i) *Adjacency matrix representation*
- (ii) *Incidence matrix representation*
- (iii) *Circuit matrix representation*
- (iv) *Cut set matrix representation*
- (v) *Path matrix representation*

### Adjacency matrix representation

The *adjacency matrix* of a graph  $G$  with  $n$  vertices is an  $n \times n$  symmetric binary matrix given by  $A = [a_{ij}]$  defined as

$$\begin{aligned} a_{ij} &= 1 && \text{if the } i^{\text{th}} \text{ and } j^{\text{th}} \text{ vertices are adjacent (i.e.) there is an edge} \\ &&& \text{connecting the } i^{\text{th}} \text{ and } j^{\text{th}} \text{ vertices} \\ &= 0 && \text{otherwise, (i.e.) if there is no edge linking the vertices.} \end{aligned}$$

**Example**

Figure 9.19(a) illustrates an undirected graph whose adjacency matrix is shown in Fig. 9.19(b).

It can easily be seen that while adjacency matrices of undirected graphs are symmetric, nothing can be said about the symmetry of the adjacency matrix of digraphs.

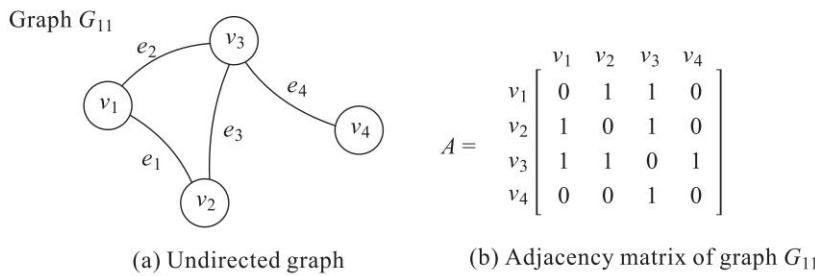
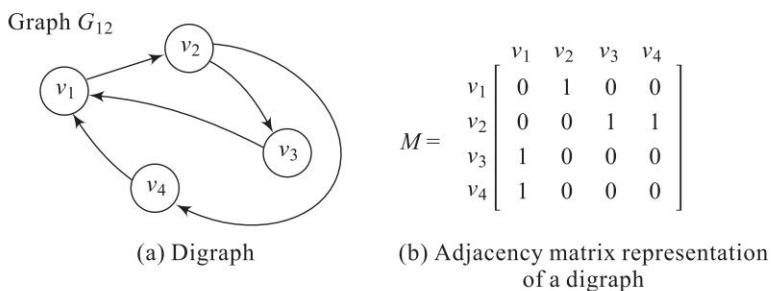
**Fig. 9.19** Adjacency matrix of an undirected graph**Example**

Figure 9.20(a-b) illustrates a digraph and its adjacency matrix representation.

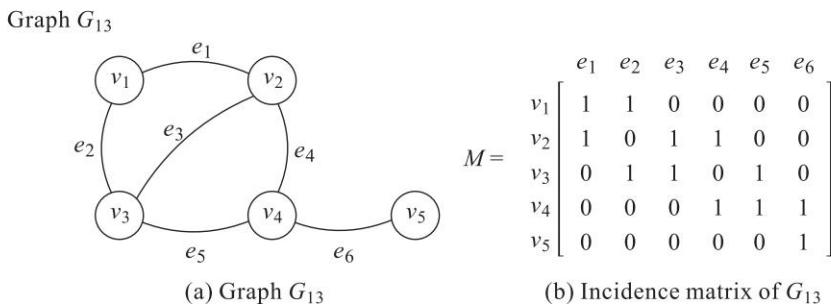
**Fig. 9.20** Adjacency matrix representation of a digraph**Incidence matrix representation**

Let  $G$  be a graph with  $n$  vertices and  $e$  edges. Define an  $n \times e$  matrix  $M = [m_{ij}]$  whose  $n$  rows correspond to  $n$  vertices and  $e$  columns correspond to  $e$  edges, as

$$m_{ij} = \begin{cases} 1 & \text{if the } j^{\text{th}} \text{ edge } e_j \text{ is incident on the } i^{\text{th}} \text{ vertex } v_i, \\ 0 & \text{otherwise} \end{cases}$$

Matrix  $M$  is known as the *incidence matrix* representation of the graph  $G$ .

**Example** Consider the graph  $G_{13}$  shown in Fig. 9.21(a), the incidence matrix representation for the graph is given in Fig. 9.21(b).

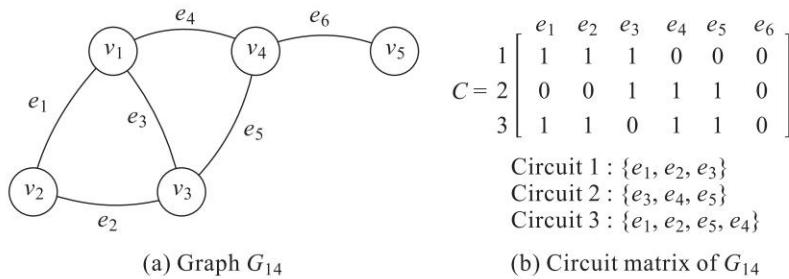
**Fig. 9.21** Incidence matrix representation of a graph

## Circuit matrix representation

For a graph  $G$  let the number of different circuits be  $t$  and the number of edges be  $e$ . Then the circuit matrix  $C = [C_{ij}]$  of  $G$  is a  $t \times e$  matrix defined as

$$C_{ij} = \begin{cases} 1 & \text{if the } i^{\text{th}} \text{ circuit includes the } j^{\text{th}} \text{ edge, otherwise} \\ 0 & \end{cases}$$

**Example** Consider the graph  $G_{14}$  shown in Fig. 9.22(a). The circuits for this graph expressed in terms of their edges are 1:  $\{e_1, e_2, e_3\}$  2:  $\{e_3, e_4, e_5\}$  3:  $\{e_1, e_2, e_5, e_4\}$ . The circuit matrix  $C$  of order  $3 \times 6$  is shown in Fig. 9.22(b).



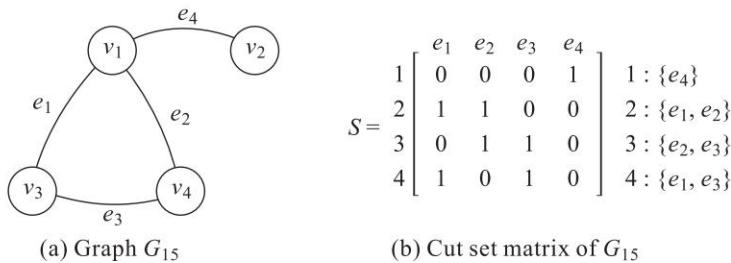
**Fig. 9.22** Circuit matrix representation of a graph

## Cut set matrix representation

For a graph  $G$ , a matrix  $S = [s_{ij}]$  whose rows correspond to cut sets and columns correspond to edges of the graph is defined to be a cut set matrix if

$$s_{ij} = \begin{cases} 1 & \text{if the } i^{\text{th}} \text{ cut set contains the } j^{\text{th}} \text{ edge, otherwise} \\ 0 & \end{cases}$$

**Example** Consider the graph  $G_{15}$  shown in Fig. 9.23(a). The cut sets of the graph are 1: $\{e_4\}$  2: $\{e_1, e_2\}$  3: $\{e_2, e_3\}$  and 4: $\{e_1, e_3\}$ . The cut set matrix representation is shown in Fig. 9.23(b).



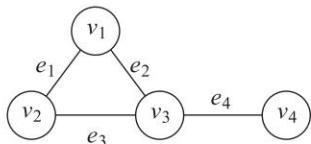
**Fig. 9.23** Cut set matrix representation of a graph

## Path matrix representation

A path matrix is generally defined for a specific pair of vertices in a graph. If  $(u, v)$  is a pair of vertices then the path matrix denoted as  $P(u,v) = [p_{ij}]$  is given by

$$p_{ij} = \begin{cases} 1 & \text{if the } j^{\text{th}} \text{ edge lies in the } i^{\text{th}} \text{ path between vertices } u \text{ and } v, \text{ otherwise} \\ 0 & \end{cases}$$

**Example** Consider the graph  $G_{16}$  shown in Fig. 9.24(a). The paths between vertices  $v_1$  and  $v_4$  are 1:{ $e_2, e_4$ } and 2:{ $e_1, e_3, e_4$ }. The path matrix representation is shown in Fig. 9.24(b).

(a) Graph  $G_{16}$ 

$$P(v_1, v_4) = \begin{bmatrix} 1 & 0 & 1 & 1 \\ 2 & 1 & 0 & 1 \end{bmatrix} \quad \text{Paths :} \\ \begin{array}{ll} 1 : \{e_2, e_4\} \\ 2 : \{e_1, e_3, e_4\} \end{array}$$

(b) Path matrix between  $v_1 v_4$  of  $G_{16}$ **Fig. 9.24 Path matrix representation**

Of all these sequential representations, adjacency matrix representation represents the graph best and is the most widely used representation. The adjacency matrix  $A$  of a graph  $G$  with  $n$  vertices has an order of  $n \times n$ . As a consequence, graph algorithms which make use of the adjacency matrix representation are bound to report a time complexity of  $O(n^2)$  since at least  $n^2 - n$  entries (excluding the diagonal elements) are to be examined.

## Linked representation of graphs

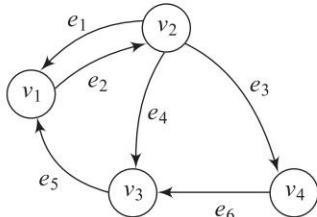
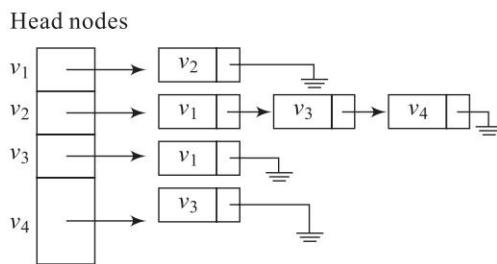
The linked representation of graphs is referred to as *adjacency list representation* and is comparatively efficient with regard to adjacency matrix representation.

Given a graph  $G$  with  $n$  vertices and  $e$  edges, the adjacency list opens  $n$  head nodes corresponding to the  $n$  vertices of graph  $G$ , each of which points to a singly linked list of nodes, which are adjacent to the vertex representing the head node.

**Example** Figure 9.25 illustrates a graph and its adjacency list representation.

It can easily be seen that if the graph is undirected, then the number of nodes in the singly linked lists put together is  $2e$  where as in the case of digraphs the number of nodes is just  $e$ , where  $e$  is the number of edges in the graph.

In contrast to adjacency matrix representations, graph algorithms which make use of an adjacency list representation would generally report a time complexity of  $O(n + e)$  or  $O(n + 2e)$  based on whether the graph is directed or undirected respectively, thereby rendering them efficient.

(a) Graph  $G_{17}$ (b) Adjacency list representation of  $G_{17}$ **Fig. 9.25 Adjacency list representation of a graph**

## Graph Traversals

## 9.4

Just as tree data structures support traversals of Inorder, Preorder and Postorder, graphs support the following traversals:

*Breadth first Traversal*, and  
*Depth first Traversal*.

A traversal, to recall, is a systematic walk which visits the nodes comprising the data structure (graphs in this case) in a specific order.

### Breadth first traversal

We discuss the *breadth first traversal* of an undirected graph in this section. The traversal starts from a vertex  $u$  which is said to be visited. Now all nodes  $v_i$ , adjacent to  $u$  are visited. The unvisited vertices  $w_{ij}$  adjacent to each of  $v_i$  are visited next and so on. The traversal terminates when there are no more nodes to visit. The process calls for the maintenance of a queue to keep track of the order of nodes whose adjacent nodes are to be visited.

Algorithm 9.1 illustrates the procedure for breadth first traversal of a graph  $G$ .

#### Algorithm 9.1: Breadth first traversal

```

Procedure BFT(s)
    /* s is the start vertex of the traversal in an undirected graph G */
    /* Q is a queue which keeps track of the vertices whose adjacent nodes
       are to be visited */

    /* Vertices which have been visited have their 'visited' flags set to
       1 (i.e.) visited (vertex) = 1.
    Initially, visited (vertex) = 0 for all vertices of graph G */

    Initialize queue Q;
    visited(s) = 1;
    call ENQUEUE (Q,s);                                /* insert s into Q */
    while not EMPTY_QUEUE(Q) do                      /* process until Q is empty */
        call DEQUEUE (Q,s);                            /* delete s from Q */
        print (s);                                     /* output vertex visited */
        for all vertices v adjacent to s do
            if (visited (v) = 0) then
                { call ENQUEUE (Q, v);
                  visited (v) =1; }
        end
    endwhile
end BFT.

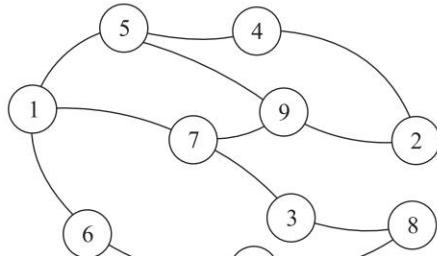
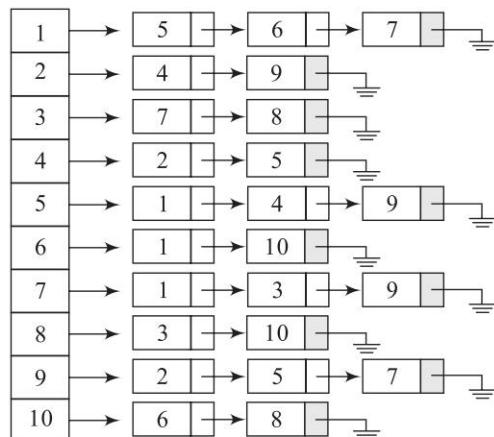
```



Breadth first traversal as its name indicates traverses the successors of the start node, generation after generation in a horizontal or linear fashion. This “breadth wise” traversal is clearly visible when the traversal is worked over a graph represented as an adjacency list

Example 9.1 illustrates the breadth first traversal of an undirected graph represented as an adjacency list.

**Example 9.1** Consider the undirected graph  $G$  shown in Fig. 9.26(a) and its adjacency list representation shown in Fig. 9.26(b). The trace of procedure BFT(1) where the start vertex is 1, is shown in Table 9.1.

(a) Graph  $G$ (b) Adjacency list of Graph  $G$ **Fig. 9.26** A graph and its adjacency list representation to demonstrate breadth first traversal**Table 9.1** Trace of the Breadth first traversal procedure on graph  $G$  (Fig. 9.26)

| Current Vertex      | Queue $Q$ | Traversal output | Status of visited flag of vertices {1, 2, 3, 4, 5, 6, 7, 8, 9, 10} of graph $G$ |
|---------------------|-----------|------------------|---------------------------------------------------------------------------------|
| 1<br>(Start vertex) | 1         |                  | 1 2 3 4 5 6 7 8 9 10<br>1 0 0 0 0 0 0 0 0 0 0                                   |
| 1                   | 5 6 7     | 1                | 1 2 3 4 5 6 7 8 9 10<br>1 0 0 0 1 1 1 0 0 0 0                                   |
| 5                   | 5 7 4 9   | 1 5              | 1 2 3 4 5 6 7 8 9 10<br>1 0 0 1 1 1 1 0 1 0                                     |
| 6                   | 7 4 9 10  | 1 5 6            | 1 2 3 4 5 6 7 8 9 10<br>1 0 0 1 1 1 1 0 1 1                                     |
| 7                   | 4 9 10 3  | 1 5 6 7          | 1 2 3 4 5 6 7 8 9 10<br>1 0 1 1 1 1 1 0 1 1                                     |
| 4                   | 9 10 3 2  | 1 5 6 7 4        | 1 2 3 4 5 6 7 8 9 10<br>1 1 1 1 1 1 1 0 1 1                                     |
| 9                   | 10 3 2    | 1 5 6 7 4 9      | 1 2 3 4 5 6 7 8 9 10<br>1 0 1 1 1 1 1 1 0 1 1                                   |

(Contd.)

(Contd.)

|    |       |                      |                                             |
|----|-------|----------------------|---------------------------------------------|
| 10 | 3 2 8 | 1 5 6 7 4 9 10       | 1 2 3 4 5 6 7 8 9 10<br>1 1 1 1 1 1 1 1 1 1 |
| 3  | 2 8   | 1 5 6 7 4 9 10 3     | 1 2 3 4 5 6 7 8 9 10<br>1 1 1 1 1 1 1 1 1 1 |
| 2  | 8     | 1 5 6 7 4 9 10 3 2   | 1 2 3 4 5 6 7 8 9 10<br>1 1 1 1 1 1 1 1 1 1 |
| 8  |       | 1 5 6 7 4 9 10 3 2 8 | 1 2 3 4 5 6 7 8 9 10<br>1 1 1 1 1 1 1 1 1 1 |
|    |       | 1 5 6 7 4 9 10 3 2 8 | Breadth first traversal ends                |

The breadth first traversal starts from vertex 1 and visits vertices 5,6,7 which are adjacent to it, while enqueueing them into queue Q. In the next shot, vertex 5 is dequeued and its adjacent, unvisited vertices 4, 9 are visited next and so on. The process continues until the queue Q which keeps track of the adjacent vertices is empty.

If an adjacency matrix representation had been used, the time complexity of the algorithm would have been  $O(n^2)$  since to visit each vertex, the while loop incurs a time complexity of  $O(n)$ . On the other hand, the use of adjacency list only calls for the examination of those nodes which are adjacent to the given node thereby curtailing the time complexity of the loop to  $O(e)$ .

## Depth first traversal

In this section we discuss the depth first traversal of an undirected graph. The traversal starts from a vertex  $u$  which is said to be visited. Now, all the nodes  $v_i$  adjacent to vertex  $u$  are collected and the first occurring vertex  $v_1$  is visited, deferring the visits to other vertices. The nodes adjacent to  $v_1$  viz.,  $w_{1k}$  are collected and the first occurring adjacent vertex viz.,  $w_{11}$  is visited deferring the visit to other adjacent nodes and so on. The traversal progresses until there are no more visits possible.

Algorithm 9.2 illustrates a recursive procedure to perform the depth first traversal of graph  $G$ .

### Algorithm 9.2: Depth first traversal

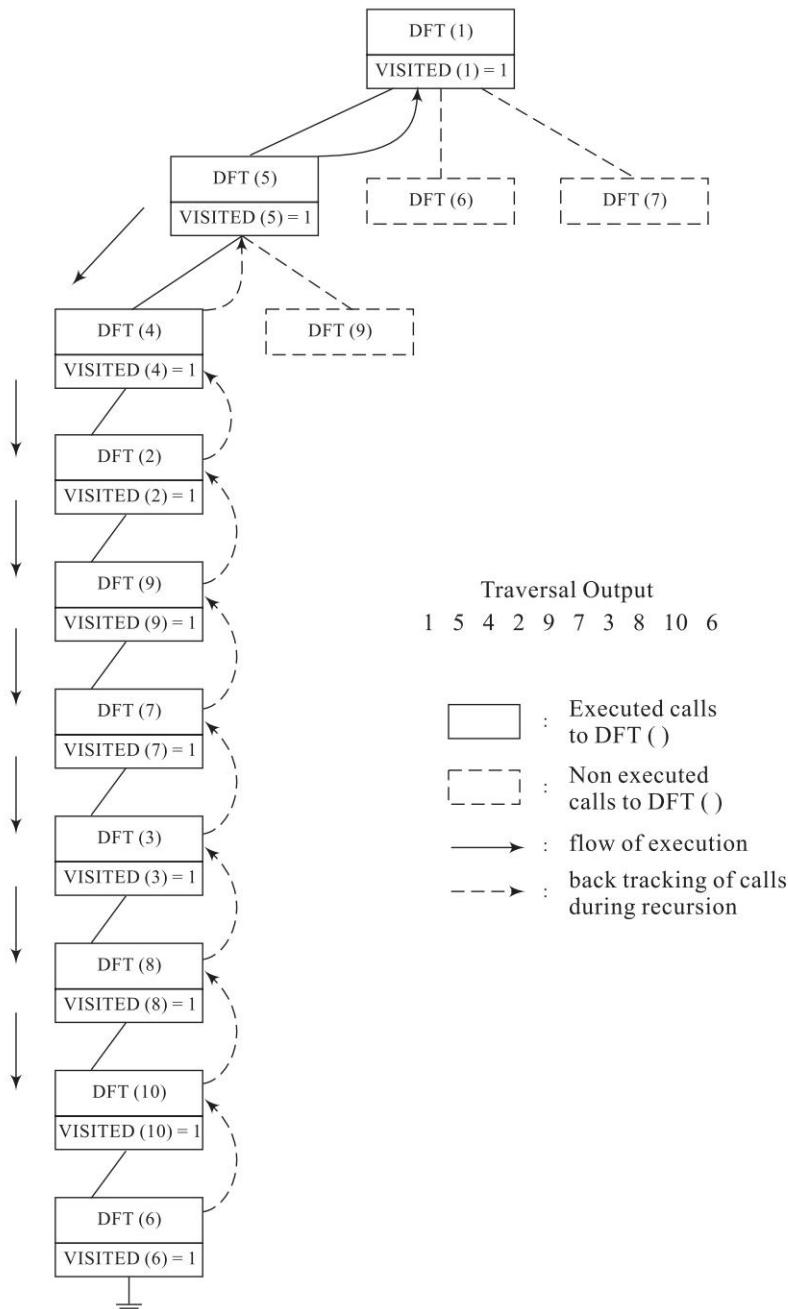
```

Procedure DFT(s)
    /* s is the start vertex */
    visited(s) = 1;
    print (s); /* Output visited vertex */
    for each vertex v adjacent to s do
        if visited(v) = 0 then call DFT(v);
    end
end DFT

```

The depth first traversal as its name indicates visits each node, that is, the first occurring among its adjacent nodes and successively repeats the operation, thus moving 'deeper and deeper' into the graph. In contrast, breadth first traversal moves side ways or breadth ways in the graph. Example 9.2 illustrates a depth first traversal of a undirected graph.

**Example 9.2** Consider the undirected graph  $G$  and its adjacency list representation shown in Fig. 9.26. Figure 9.27 shows a tree of recursive calls which represents a trace of the procedure DFT(1) on the graph  $G$  with start vertex 1.



**Fig. 9.27** Tree of recursive calls showing the trace of procedure DFT(1) on graph  $G$  (Fig. 9.26)

The tree of recursive calls illustrates the working of the DFT procedure. The first call `DFT(1)` visits start vertex 1 and releases 1 as the traversal output. Vertex 1 has vertices 5, 6, 7 as its adjacent nodes. `DFT(1)` now invokes `DFT(5)`, visiting vertex 5 and releasing it as the next traversal output. However `DFT(6)` and `DFT(7)` are kept in waiting for `DFT(5)` to complete its execution. Such procedure calls waiting to be executed are shown in broken line boxes in the tree of recursive calls.

Now `DFT(5)` invokes `DFT(4)` releasing vertex 4 as the traversal output while `DFT(9)` is kept in abeyance. Note that though vertex 1 is an adjacent node of vertex 5, since no `DFT( )` calls to vertices already visited are invoked, `DFT(1)` is not called for. The process continues until `DFT(6)` completes its execution with no more nodes left to visit. During recursion the calls made to `DFT( )` procedure are indicated using solid arrows in the forward direction.

Once `DFT(6)` finishes execution, back tracking takes place which is indicated using broken arrows in the reverse direction. Once `DFT(1)` completes execution the traversal output is gathered to be 1 5 4 2 9 7 3 8 10 6.

In the adjacency list representation of graph  $G$ , `DFT( )` examines each node in the adjacency list at most once. Since there are  $2e$  list nodes, the time complexity turns out to be  $O(e)$ . On the other hand, the adjacency matrix implementation for procedure `DFT( )` records a time complexity of  $O(n^2)$ .

Both breadth first and depth first traversals, irrespective of the vertex they start from, visit all vertices of the graph that are connected to it. Hence, if the graph is connected, both traversals would visit all the vertices of the graph. On the other hand, if the graphs were not connected, both traversals would yield only their connected components. Thus breadth first traversal and depth first traversal can be useful in testing for the connectivity of graphs. If after executing the traversal algorithms, there are any vertices which are left unvisited, then it implies that the graph is disconnected.

## Applications

9.5

We illustrate the application of graphs for

- (i) Determination of shortest path (Single Source Shortest path problem), and
- (ii) Extraction of minimum cost spanning trees.

### Single-source, shortest-path problem

Given a network of cities and the distances between them, the objective of the *single-source, shortest-path problem* is to find the shortest path from a city (termed *source*) to all other cities connected to it.

The network of cities with their distances is represented as a weighted digraph. Algorithm 9.3 illustrates Dijkstra's algorithm for the single source shortest path problem.

Let  $V$  be a set of  $N$  cities (vertices) of the digraph. Here the source is city 1. The set  $T$  is initialized to city 1. The DISTANCE vector, `DISTANCE [2:N]` initially records the distances of cities 2 to  $N$  connected to the source by an edge (not path!). If there is no edge directly connecting the city to the source, then we initialize its DISTANCE value to  $\infty$ . Once the algorithm completes its iterations, the DISTANCE vector holds the shortest distance of cities 2 to  $N$  from the source city 1.

It is convenient to represent the weighted digraph using its cost matrix  $\text{COST}_{N \times N}$ . The cost matrix records the distances between cities connected by an edge.

Dijkstra's algorithm has a complexity of  $O(N^2)$  where  $N$  is the number of vertices (cities) in the weighted digraph.

**Algorithm 9.3:** Dijkstra's algorithm for the single source shortest path problem

**Procedure** DIJKSTRA\_SSSP( $N$ , COST)

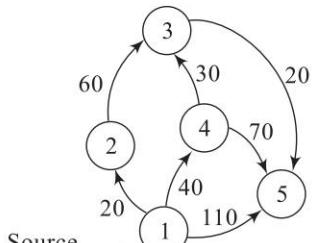
```

/*N is the number of vertices labeled { 1, 2, 3,...N} of the weighted
digraph. COST[1:N,1:N] is the cost matrix of the graph. If there is
no edge then COST [i, j] = ∞*/
/* The procedure computes the cost of the shortest path from vertex
1 the source, to every other vertex of the weighted digraph */
T = {1}; /* Initialize T to source vertex */
for i = 2 to N do
  DISTANCE[i] = COST[1,i]; /*Initialize DISTANCE vector
  end                                to the cost of the edges connecting
   vertex i with the source vertex 1. If
   there is no edge then COST [1, i] = ∞*/
for i = 1 to N -1 do
  Choose a vertex u in V - T such that DISTANCE[u]
  is a minimum;
  Add u to T;
  for each vertex w in V-T do
    DISTANCE [w] = minimum (DISTANCE[w],
                           DISTANCE [u] + COST [u, w] );
  end
end
end DIJKSTRA-SSSP

```



**Example 9.3** Consider the weighted digraph of cities and its cost matrix shown in Fig. 9.28. Table 9.2 shows the trace of the Dijkstra's algorithm.



(a) Weighted digraph

|   | 1        | 2        | 3        | 4        | 5   |
|---|----------|----------|----------|----------|-----|
| 1 | 0        | 20       | $\infty$ | 40       | 110 |
| 2 | $\infty$ | 0        | 60       | $\infty$ | 0   |
| 3 | $\infty$ | $\infty$ | 0        | $\infty$ | 20  |
| 4 | $\infty$ | $\infty$ | 30       | 0        | 70  |
| 5 | $\infty$ | $\infty$ | $\infty$ | $\infty$ | 0   |

(b) Cost matrix  $C_{5 \times 5}$

**Fig. 9.28** A weighted digraph and its cost matrix

**Table 9.2** Trace of Dijkstra's algorithm on the weighted digraph (Fig. 9.28)

| Iteration | T               | u | DISTANCE |     |     |     |
|-----------|-----------------|---|----------|-----|-----|-----|
|           |                 |   | [2]      | [3] | [4] | [5] |
| Initial   | {1}             | — | 20       | ∞   | 40  | 110 |
| 1         | {1, 2}          | 2 | 20       | 80  | 40  | 110 |
| 2         | {1, 2, 4}       | 4 | 20       | 70  | 40  | 110 |
| 3         | {1, 2, 4, 3}    | 3 | 20       | 70  | 40  | 90  |
| 4         | {1, 2, 4, 3, 5} | 5 | 20       | 70  | 40  | 90  |

The DISTANCE vector in the last iteration records the shortest distance of the vertices {2, 3, 4, 5} from the source vertex 1.

To reconstruct the shortest path from the source vertex to all other vertices, a vector PREDECESSOR[1:N] where PREDECESSOR[v] records the predecessor of vertex  $v$  in the shortest path, is maintained. PREDECESSOR[v] is initialized to source for all  $v \neq$  source. PREDECESSOR[v] is updated by inserting the statement

```
if (DISTANCE[u] + COST[u, w]) < DISTANCE[w]
then PREDECESSOR[w] = u
```

soon after  $\text{DISTANCE}[w] = \min(\text{DISTANCE}[w], \text{DISTANCE}[u] + \text{COST}[u, w])$  is computed in procedure DIJKSTRA\_SSSP. To trace the shortest path we move backwards from the destination vertex, hopping on the predecessors recorded by the PREDECESSOR vector until the source vertex is reached.

**Example 9.4** To trace the shortest paths of the vertices from vertex 1 using Dijkstra's algorithm, inclusion of the statement updating PREDECESSOR vector results in the following:

|                                | PREDECESSOR |     |     |     |     |
|--------------------------------|-------------|-----|-----|-----|-----|
|                                | [1]         | [2] | [3] | [4] | [5] |
| Initialization                 | —           | 1   | 1   | 1   | 1   |
| Final values after iteration 4 |             | 1   | 4   | 1   | 3   |

To trace the shortest path from source 1 to vertex 5, we move in the reverse direction from vertex 5 (shown in dotted lines) hopping on the predecessors until the source vertex is reached. The shortest path is given by:

$$\text{PREDECESSOR}(5) = 3 \quad \text{PREDECESSOR}(3) = 4 \quad \text{PREDECESSOR}(4) = 1$$



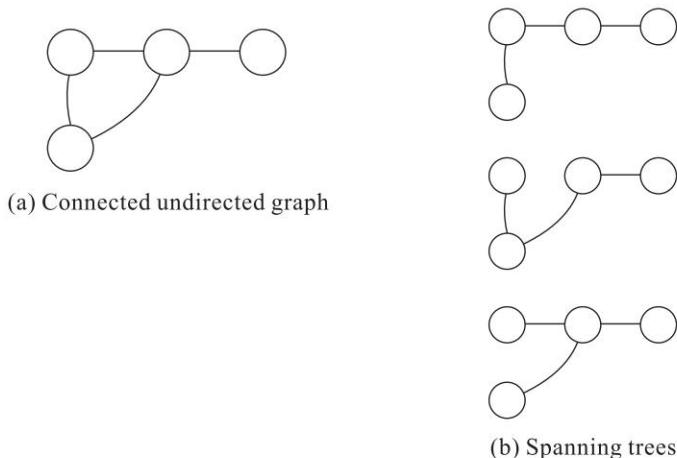
Thus the shortest path between vertex 1 and vertex 5 is 1 – 4 – 3 – 5 and the distance is given by DISTANCE[5] is 90.

## Minimum cost spanning trees

Consider an application where  $n$  stations are to be linked using a communication network. The laying of communication links between any two stations involves a cost. The problem is to obtain a network of communication links which while preserving the connectivity between stations does it with minimum cost. If the problem were to be modeled as a weighted graph, the ideal solution to the problem would be to extract a subgraph termed *minimum cost spanning tree* which while preserving the connectedness of the graph yields minimum cost.

Let  $G = (V, E)$  be an undirected connected graph. A subgraph  $T = (V, E')$  of  $G$  is a *spanning tree* of  $G$  iff  $T$  is a tree.

**Example** For the connected graph undirected shown in Fig. 9.29 (a), some of the spanning trees extracted from the graph are shown in Fig. 9.29(b).



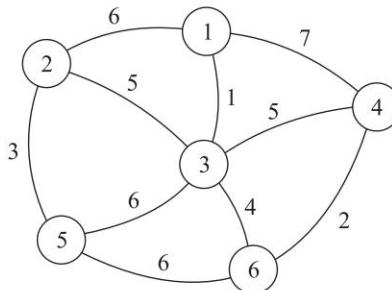
**Fig. 9.29** Spanning trees of a graph

Given a connected undirected graph there are several spanning trees that may be extracted from it. Now given  $G = (V, E)$  to be a connected, weighted undirected graph where each edge involves a cost, the extraction of a spanning tree extends itself to the extraction of a minimum cost spanning tree. A minimum cost spanning tree is a spanning tree which has a minimum total cost. Algorithm 9.4 illustrates Prims algorithm for the extraction of minimum cost spanning trees.

A spanning tree of a graph  $G$  with  $n$  vertices will have  $(n - 1)$  edges. This is due to the property that a tree with  $n$  vertices has always  $(n - 1)$  edges (Refer Illustrative Problem 9.9). Also, addition of even one single edge results in the spanning tree losing its property of acyclicity and removal of one single edge results in its losing the property of connectivity.

The time complexity of Prims algorithm is  $O(n^2)$ .

**Example 9.5** Consider the connected, weighted, undirected graph shown in Fig. 9.30. Table 9.3 illustrates the trace of the PRIM's algorithm on the graph.



**Fig. 9.30** A connected weighted undirected graph for the extraction of minimum cost spanning tree using Prims algorithm

**Algorithm 9.4:** Prims Algorithm to obtain the minimum cost spanning tree from a connected undirected graph

```

procedure PRIM( $G$ )
    /*  $G = (V, E)$  is a weighted, connected undirected graph and  $E'$  is
       the set of edges which are to be extracted to obtain the minimum
       cost spanning tree */
     $E' = \emptyset$ ; /* Initialize  $E'$  to a null set */
    Select a minimum cost edge  $(u, v)$  from  $E$ ;
     $V' = \{u\}$  /* Include  $u$  in  $V'$  */
    while  $V' \neq V$  do
        Let  $(u, v)$  be the lowest cost edge such that  $u$  is in  $V'$ 
        and  $v$  is in  $V - V'$ ;
        Add edge  $(u, v)$  to set  $E'$ ;
        Add  $v$  to set  $V'$ ;
    endwhile
end PRIM

```

**Table 9.3** Trace of the Prims algorithm on the connected weighted undirected graph (Fig. 9.30)

| Edge     | Cost of the edge | $V'$          | $E'$                 | Spanning tree |
|----------|------------------|---------------|----------------------|---------------|
| $(1, 3)$ | 1                | 1             | —                    | —             |
| $(1, 3)$ | 1                | $\{1, 3\}$    | $\{(1, 3)\}$         |               |
| $(3, 6)$ | 4                | $\{1, 3, 6\}$ | $\{(1, 3), (3, 6)\}$ |               |

(Contd.)

(Contd.)

|        |   |                    |                                          |  |
|--------|---|--------------------|------------------------------------------|--|
|        |   |                    |                                          |  |
| (6, 4) | 2 | {1, 3, 6, 4}       | {(1, 3), (3, 6), (6, 4)}                 |  |
| (3, 2) | 5 | {1, 3, 6, 4, 2}    | {(1, 3), (3, 6), (6, 4), (3, 2)}         |  |
| (2, 5) | 3 | {1, 3, 6, 4, 2, 5} | {(1, 3), (3, 6), (6, 4), (3, 2), (2, 5)} |  |

We first initialize  $V'$  to a vertex of the lowest cost edge of the graph  $G$ . Then with each iteration we look for a lowest cost edge that has one of its end points in  $V'$ , all the while ensuring that the edge chosen does not destroy the property of connectedness and acyclicity insisted upon by spanning tree. Once  $V' = V$  the algorithm terminates obtaining the minimum cost spanning tree. In this example the minimum cost spanning tree has a cost of 15.

## ADT for Graphs

**Data objects:**

A graph  $G$  of vertices and edges. Vertices represent data.

**Operations:**

- Check if graph  $G$  is empty  
 $\text{CHECK\_GRAPH\_EMPTY } (G)$  (Boolean function)
- Insert an isolated vertex  $V$  into a graph  $G$ . Ensure that  $V$  does not exist in  $G$  before insertion.  
 $\text{INSERT\_VERTEX } (G, V)$
- Insert an edge connecting vertices  $U, V$  into a graph  $G$ . Ensure that such an edge does not exist in  $G$  before insertion.  
 $\text{INSERT\_EDGE}(G, U, V)$
- Delete vertex  $V$  and all the edges incident on it from the graph  $G$ . Ensure that such a vertex exists in the graph before deletion.  
 $\text{DELETE\_VERTEX } (G, V)$
- Delete an edge from the graph  $G$  connecting the vertices  $U, V$ . Ensure that such an edge exists before deletion.  
 $\text{DELETE\_EDGE } (G, U, V)$
- Store ITEM into a vertex  $V$  of graph  $G$   
 $\text{STORE\_DATA}(G, V, ITEM)$
- Retrieve data of a vertex  $V$  in the graph  $G$  and return it in ITEM  
 $\text{RETRIEVE\_DATA } (G, V, ITEM)$
- Perform Breadth first traversal of a graph  $G$ .  
 $\text{BFT } (G)$
- Perform Depth first traversal of a graph  $G$ .  
 $\text{DFT } (G)$



## Summary

- Graphs are non-linear data structures. The history of graph theory originated from the classical Koenigsberg bridge problem.
- A graph  $G = (V, E)$  consists of a finite set of vertices  $V$  and edges  $E$ . Undirected graph, digraph, complete graph, subgraph, tree, isomorphic graphs and labeled graphs are graphs which satisfy special properties.
- Path, simple path, cycle, degree, cut set, pendant vertex, Eulerian walk, Hamiltonian circuit are terminologies associated with graphs.
- For problem solving using computers, graphs are represented using two popular methods viz., adjacency matrix representation and adjacency list representation which belong to the class of sequential and linked representations respectively.
- The other matrix representations for graphs are incidence matrix, circuit matrix, cut set matrix and path matrix.
- Graphs support the traversals of breadth first and depth first. The traversal techniques can be employed to test for the connectedness of the graph.
- Two applications of graphs viz., single source shortest path problem and extraction of minimal spanning trees have been discussed.



## Illustrative Problems

**Problem 9.1** Draw the graphs:

$$G_1: V_1 = \{a, b, c, d\} \quad E_1 = \{\langle a, b \rangle, \langle b, c \rangle, \langle d, c \rangle, \langle c, a \rangle\}$$

$$G_2: V_2 = \{a, b, c, d\} \quad E_2 = \{(a, b), (b, c), (a, d), (b, d)\}$$

**Solution:** Figure I 9.1 illustrates the graphs. Here  $G_1$  is a digraph and  $G_2$  is an undirected graph.

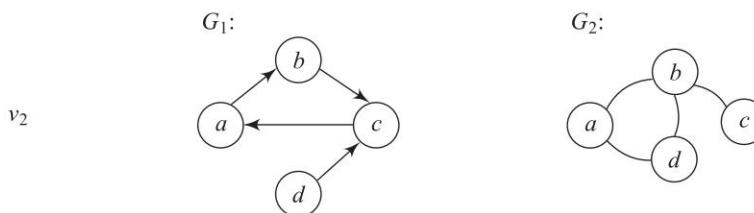


Fig. I 9.1

**Problem 9.2** For the graph given in Fig. I 9.2 find

- an isolated vertex
- degree of node  $b$
- a simple path from  $a$  to  $c$
- a path which is not simple from  $a$  to  $c$
- a cycle
- a pendant vertex

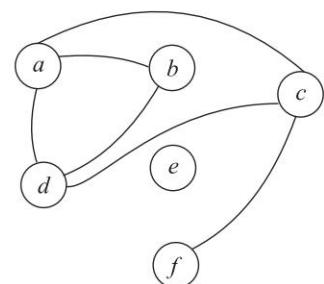


Fig. I 9.2

**Solution:**

- (i) e
- (ii) 2
- (iii) {a, d, c} or {a, b, d, c}
- (iv) {a, b, d, a, c}
- (v) {a, b, d}
- (vi) f

**Problem 9.3** For the graph given in Fig. I 9.3

- (a) obtain
  - (i) The cut sets
  - (ii) An Eulerian walk
  - (iii) A Hamiltonian path
- (b) Is the graph complete?

**Solution:**

- (a) (i) The cut set is {C} since removal of the vertex with all the edges incident on it disconnects the graph.
- (ii) No, an Eulerian walk does not exist since not all nodes have even degree.
- (iii) A Hamiltonian path is given by D B C A E
- (b) No, the graph is not complete since all the  ${}^5C_2$  edges are not available.

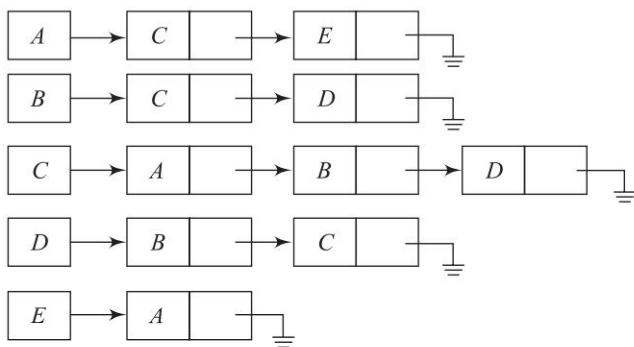
**Problem 9.4** Represent the graph shown in Fig. I 9.3 using

- (i) Adjacency matrix
- (ii) Adjacency list and
- (iii) Incidence matrix

**Solution:**

$$\begin{array}{c} \begin{matrix} & A & B & C & D & E \\ A & 0 & 0 & 1 & 0 & 1 \\ B & 0 & 0 & 1 & 1 & 0 \\ C & 1 & 1 & 0 & 1 & 0 \\ D & 0 & 1 & 1 & 0 & 0 \\ E & 1 & 0 & 0 & 0 & 0 \end{matrix} \end{array}$$

- (i) Adjacency matrix



- (ii) Adjacency List

(iii) Incidence matrix

$$A \begin{pmatrix} e_1 & e_2 & e_3 & e_4 & e_5 \\ 1 & 0 & 0 & 0 & 1 \\ B & 0 & 1 & 1 & 0 & 0 \\ C & 1 & 1 & 0 & 1 & 0 \\ D & 0 & 0 & 1 & 1 & 0 \\ E & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

**Problem 9.5** Show that if  $d_i$  is the degree of vertex  $i$  in a graph  $G$  with  $n$  vertices and  $e$  edges then

$$e = \frac{\sum_{i=1}^n d_i}{2}$$

**Solution:** Since each edge contributes to the degree of its two end vertices,

$$\begin{aligned} d_1 + d_2 + \dots + d_n &= 2e \\ (\text{i.e.}) \quad e &= \frac{(d_1 + d_2 + \dots + d_n)}{2} \end{aligned}$$

$$= \frac{\sum_{i=1}^n d_i}{2}$$

Hence, the result.

**Problem 9.6** With the help of the result  $\sum_{i=1}^n d_i = 2e$  (proved in Illustrative Problem 9.5) show that the number of vertices of odd degree is even.

$$\begin{aligned} \text{Solution:} \quad \sum_{i=1}^n d_i &= \sum_{\substack{\text{odd degree} \\ \text{vertices}}} d_k + \sum_{\substack{\text{even degree} \\ \text{vertices}}} d_j \end{aligned}$$

$$\therefore \sum_{\substack{\text{odd degree} \\ \text{vertices}}} d_k = 2e - \sum_{\substack{\text{even degree} \\ \text{vertices}}} d_j$$

$$\Rightarrow \sum_{\substack{\text{odd degree} \\ \text{vertices}}} d_k = \text{an even number } c$$

As each  $d_k$  of the summation is an odd number, for the summation to be an even number (denoted as  $c$ ), the number of terms must be even. Hence the number of vertices of odd degree is even.

**Problem 9.7** There is one and only one path between every pair of vertices in a tree  $T$ . Prove.

**Solution:** If there is more than one path between a pair of distinct vertices  $v_i, v_j$  in a tree  $T$  then it means that a circuit exists. Hence  $T$  is no more a tree. Therefore there exists one and only path between every pair of vertices in a tree  $T$ .

**Problem 9.8** If in a graph  $G$  there is one and only one path between every pair of vertices then  $G$  is a tree.

**Solution:** For  $G$  to be a tree, (i)  $G$  should be connected and (ii)  $G$  should have no cycles.

(i) is true since there exists a path between every pair of vertices

(ii) is also true since there is one and only one path between every pair of vertices which ensures absence of cycles (Illustrative Problem 9.7)

Hence  $G$  is a tree.

**Problem 9.9** A tree  $T$  with  $n$  vertices has  $(n - 1)$  edges. Prove.

**Solution:** We prove this by induction.

For  $n = 1$ , a tree has no edge or has  $(1 - 1) = 0$  edges.

For  $n = 2$ , a tree has one edge or has  $(2 - 1) = 1$  edge.

The statement is therefore true for  $n = 1$  and  $n = 2$ . Let us suppose the theorem holds for a tree  $T$  with  $n-1$  vertices. Now to prove that it is true for a tree  $T$  with  $n$  vertices. Consider a tree  $T$  with  $n$  vertices. Remove an edge  $e$  from  $T$ . This disconnects  $T$  and results in two trees  $T_1, T_2$  each with  $n'$  and  $n-n'$  nodes respectively, for some  $n'$ . Since  $n'$  and  $n-n'$  are fewer than  $n$ , the total number of edges in  $T_1$  and  $T_2$  put together is  $(n'-1) + (n - n'-1) = n-2$ . Replacing  $e$ , the tree  $T$  has  $(n-2+1) = n-1$  edges. Hence the proof.

**Problem 9.10** Extract a minimum cost spanning tree for the graph shown in Fig. I 9.10.

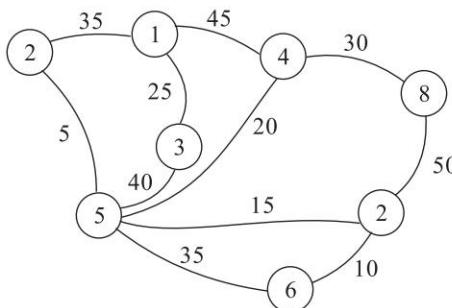
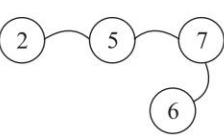
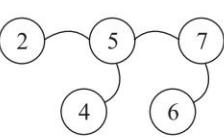
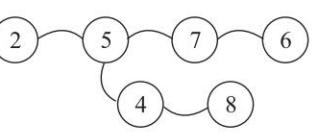
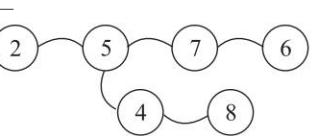
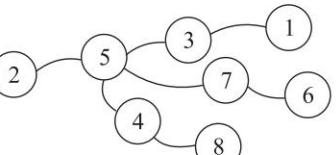


Fig. I 9.10

**Solution:** Table I 9.1 illustrates the trace of the Prims algorithm in obtaining the minimum cost spanning tree.

**Table I 9.1**

| Edge             | Cost of the edge | $V'$                     | $E'$                                               | Minimum Cost Spanning tree                                                           |
|------------------|------------------|--------------------------|----------------------------------------------------|--------------------------------------------------------------------------------------|
| (2, 5)           | —                | {2}                      | —                                                  | —                                                                                    |
| (2, 5)           | 5                | {2, 5}                   | {(2, 5)}                                           |    |
| (5, 7)           | 15               | {2, 5, 7}                | {(2, 5) (5, 7)}                                    |    |
| (7, 6)           | 10               | {2, 5, 7, 6}             | {(2, 5) (5, 7) (7, 6)}                             |    |
| (5, 4)           | 20               | {2, 5, 7, 6, 4}          | {(2, 5) (5, 7) (7, 6) (5, 4)}                      |    |
| (4, 8)           | 30               | {2, 5, 7, 6, 4, 8}       | {(2, 5) (5, 7) (7, 6) (5, 4) (4, 8)}               |    |
| (5, 6)<br>Reject | —                | —                        | —                                                  | —                                                                                    |
| (5, 3)           | 40               | {2, 5, 7, 6, 4, 8, 3}    | {(2, 5) (5, 7) (7, 6) (5, 4) (4, 8) (5, 3)}        |   |
| (1, 3)           | 25               | {2, 5, 7, 6, 4, 8, 3, 1} | {(2, 5) (5, 7) (7, 6) (5, 4) (4, 8) (5, 3) (1, 3)} |  |

Total Cost of the minimum spanning tree is 145.

**Problem 9.11** Obtain a solution to the single-source, shortest-path problem defined on the digraph shown in Fig. I 9.11.

**Solution:** Table I 9.2 illustrates the trace of the Dijkstra's algorithm on the single-source, shortest-path problem.

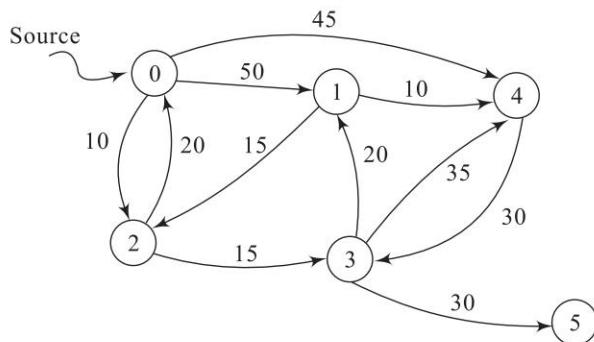


Fig. I 9.11

Table I 9.2

| Iteration  | T                  | U | DISTANCE |     |          |     |          | PREDECESSER |     |     |     |     |
|------------|--------------------|---|----------|-----|----------|-----|----------|-------------|-----|-----|-----|-----|
|            |                    |   | [1]      | [2] | [3]      | [4] | [5]      | [1]         | [2] | [3] | [4] | [5] |
| Initialize | {0}                | — | 50       | 10  | $\infty$ | 45  | $\infty$ | 0           | 0   | 0   | 0   | 0   |
| 1          | {0, 2}             | 2 | 50       | 10  | 25       | 45  | $\infty$ | 0           | 0   | 2   | 0   | 0   |
| 2          | {0, 2, 3}          | 3 | 45       | 10  | 25       | 45  | 55       | 3           | 0   | 2   | 0   | 3   |
| 3          | {0, 2, 3, 1}       | 1 | 45       | 10  | 25       | 45  | 55       | 3           | 0   | 2   | 0   | 3   |
| 4          | {0, 2, 3, 1, 4}    | 4 | 45       | 10  | 25       | 45  | 55       | 3           | 0   | 2   | 0   | 3   |
| 5          | {0, 2, 3, 1, 4, 5} | 5 | 45       | 10  | 25       | 45  | 55       | 3           | 0   | 2   | 0   | 3   |

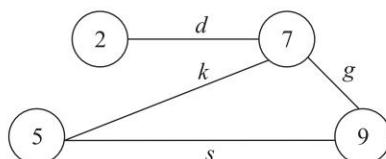
The shortest paths and distances are:

| Source | Destination | Shortest Path | Shortest Distance |
|--------|-------------|---------------|-------------------|
| 0      | 1           | 0 - 2 - 3 - 1 | 45                |
| 0      | 2           | 0 - 2         | 10                |
| 0      | 3           | 0 - 2 - 3     | 25                |
| 0      | 4           | 0 - 4         | 45                |
| 0      | 5           | 0 - 2 - 3 - 5 | 55                |



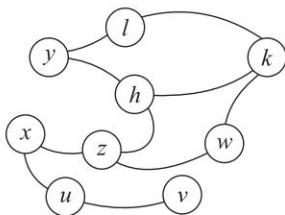
## Review Questions

1. Which of the following does not hold good for the given graph G?



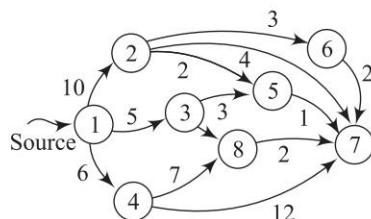
- (i) an Eulerian walk exists for the graph
- (ii) the graph is an undirected graph

- (iii) the graph has a cycle  
 (iv) the graph has a pendant vertex  
     (a) (i)                                          (b) (ii)                                          (c) (iii)                                          (d) (iv)
2. Which of the following properties is not satisfied by two graphs that are isomorphic?  
 (i) They have the same number of vertices  
 (ii) They have the same number of edges  
 (iii) They have an equal number of vertices with a given degree  
 (iv) There must exist at least one cycle  
     (a) (i)                                                  (b) (ii)                                          (c) (iii)                                          (d) (iv)
3. For the graph shown in Review Question 1 (Chapter 9), the following matrix represents its
- $$\begin{array}{c}
 \begin{matrix} & d & g & k & s \\ \hline
 2 & 1 & 0 & 0 & 0 \\ 
 5 & 0 & 0 & 1 & 1 \\ 
 7 & 1 & 1 & 1 & 0 \\ 
 9 & 0 & 1 & 0 & 1 \end{matrix}
 \end{array}$$
- (i) Adjacency matrix representation  
 (ii) Incidence matrix representation  
 (iii) Circuit matrix representation  
 (iv) Cut set matrix representation  
     (a) (i)                                                  (b) (ii)                                          (c) (iii)                                          (d) (iv)
4. In the context of graph traversals, state whether true or false:  
 (i) graph traversals could be employed to check for the connectedness of a graph  
 (ii) for any graph, graph traversals always visit all vertices of the graph  
     (a) (i) true (ii) true                                          (b) (i) true (ii) false  
     (c) (i) false (ii) true                                          (d) (i) false (ii) false
5. Which among the following properties is not satisfied by a minimum cost spanning tree  $T$  extracted from a graph  $G$  with  $n$  vertices?  
 (i)  $T$  has a cycle  
 (ii)  $T$  has  $(n-1)$  edges  
 (iii)  $T$  has  $n$  vertices  
 (iv)  $T$  is connected  
     (a) (i)                                                  (b) (ii)                                          (c) (iii)                                          (d) (iv)
6. Distinguish between digraphs and undirected graphs?
7. For a graph of your choice, trace its (i) adjacency matrix and (ii) adjacency list representations.
8. Draw graphs that contain (i) an Eulerian walk, and (ii) a Hamiltonian circuit
9. How can graph traversal procedures help in detecting graph connectivity?
10. Discuss an application of minimum cost spanning trees.
11. Trace (i) Breadth first traversal and (ii) Depth first traversal on the graph shown in Fig. R9.11, beginning from vertex  $y$ .



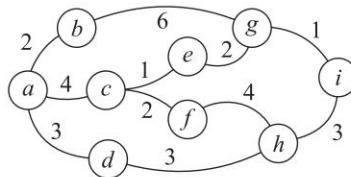
**Fig. R9.11** A strongly connected graph

12. For the graph shown in Fig. R9.12, obtain the shortest path from vertex 1 to all other vertices:



**Fig. R9.12** Strongly connected components of a diagram

13. For the graph shown in Fig. 9.13, extract a minimum cost spanning tree.



**Fig. R9.13** Graphs which are trees and not trees



## Programming Assignments

1. Execute a program to input a graph  $G = (V, E)$  as an adjacency matrix. Include functions to
  - (i) test if  $G$  is complete
  - (ii) obtain a path and a simple path from vertex  $u$  to vertex  $v$ .
  - (iii) obtain the degree of a node  $u$ , if  $G$  is undirected, and indegree and outdegree of node  $u$  if  $G$  is directed.
2. Execute a program to input a graph  $G = (V, E)$  as an adjacency list. Include two functions BFT and DFT to undertake a breadth first and depth first traversal of the graph. Making use of the traversal procedures, test whether the graph is connected.

3. Implement Dijkstra's algorithm to obtain the shortest paths from the source vertex 1 to every other vertex of the graph  $G$  given below:

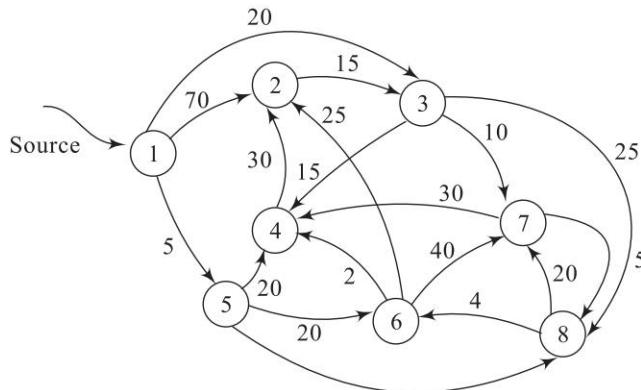


Fig. P 9.3

4. Design and implement an algorithm to obtain a spanning tree of a connected, undirected graph using breadth first or depth first traversal.
5. Design and implement an algorithm to execute depth first traversal of a graph represented by its incidence matrix.
6. Design and implement an algorithm to obtain an Eulerian walk of an undirected graph in the event of such a walk being available.
7. Implement the ADT for graphs in a programming language of your choice choosing a linked representation for the graphs.

## CHAPTER



# BINARY SEARCH TREES 10 AND AVL TREES

## Introduction

## 10.1

In Chapter 8, the tree and binary tree data structures were discussed. Binary search trees and AVL trees are a category of binary trees which facilitate efficient retrievals. In this chapter the definition of a binary search tree and its operations viz., retrieval, insertion and deletion are discussed. However, binary search trees can have their setbacks too, the rectification of which yields an AVL tree. The definition of the AVL search tree and the operations of retrieval, insertion and deletion on the tree are elaborated next. The application of the two data structures to the representation of symbol tables in compiler design have been detailed last.

## Binary Search Trees: Definition and Operations

## 10.2

### Definition

A *binary search tree*  $T$  may be an empty binary tree. If non-empty, then for a set  $S$ ,  $T$  is a labeled binary tree in which each node  $u$  is labeled by an element or key  $e(u) \in S$  such that

- (i) for each node  $u$  in the left subtree of  $v$ ,  $e(u) < e(v)$
- (ii) for each node  $u$  in the right subtree of  $v$ ,  $e(u) > e(v)$
- (iii) for each element  $a \in S$  there is exactly one node  $u$  such that  $e(u) = a$ .

In other words, a binary search tree  $T$  satisfies the following norms:

- (i) all keys of the binary search tree must be distinct
- (ii) all keys in the left subtree of  $T$  are less than the root element
- (iii) all keys in the right subtree of  $T$  are greater than the root element and
- (iv) the left and right subtrees of  $T$  are also binary search trees.

Figure 10.1 illustrates an empty binary search tree and a non empty binary search tree defined for the set  $S = \{G, M, B, E, K, I, Q, Z\}$ . It needs to be emphasized here that for a given set  $S$  more than one binary search tree can be constructed.

- 10.1 *Introduction*
- 10.2 *Binary Search Trees: Definition and Operations*
- 10.3 *AVL Trees: Definition and Operations*
- 10.4 *Applications*

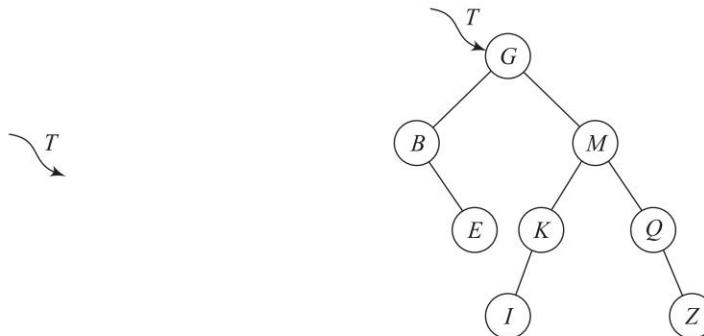
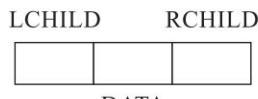


Fig. 10.1 Example binary search trees

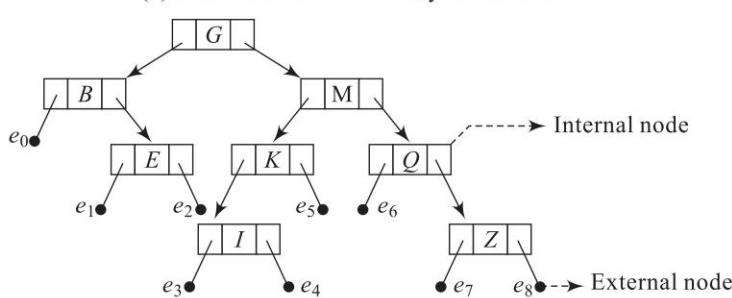
The inorder traversal of a binary search tree  $T$  yields the elements of the associated set  $S$  in the ascending order. If  $S = \{a_i, i = 1, 2, \dots, n\}$ , then the inorder traversal of the binary search tree yields the elements in its ascending sequence, for example  $a_1 < a_2 < a_3 < \dots < a_n$ . Thus, the inorder traversal of the binary search tree shown in Fig. 10.1 results in  $\{B, E, G, I, K, M, Q, Z\}$  which are the elements of  $S$  in the ascending order.

## Representation of a binary search tree

A binary search tree is commonly represented using a linked representation in the same way as that of a binary tree (Sec. 8.5). The node structure and the linked representation of the binary search tree shown in Fig. 10.1, is illustrated in Fig. 10.2. However, the null pointers of the nodes may be emphatically represented using fictitious nodes called *external nodes*. The external nodes labeled as  $e_i$  are shown as solid circles in Fig. 10.2. Thus a linked representation of a binary search tree is viewed as a bundle of external nodes which represent the null pointers and *internal nodes* which represent the keys. Such a binary tree is referred to as an *extended binary tree*. Obviously, the number of external nodes in a binary search tree comprising  $n$  internal nodes is  $n+1$ . The path from the root to an external node is called as an *external path*.



(a) Node structure of a binary search tree



(b) Linked representation of the binary search tree of Fig. 10.1(b)

Fig. 10.2 Linked representation of a binary search tree

## Retrieval from a binary search tree

Let  $T$  be a binary search tree. To retrieve a key  $u$  from  $T$ ,  $u$  is first compared with the root key  $r$  of  $T$ . If  $u = r$  then the search is done. If  $u < r$  then the search begins at the left subtree of  $T$ . If  $u > r$  then the search begins at the right subtree of  $T$ . The search is repeated recursively in the left and right sub-subtrees with  $u$  compared against the respective root keys, until the key  $u$  is either found or not found. If the key is found the search is termed *successful* and if not found, is termed *unsuccessful*.

While all successful searches terminate at the appropriate internal nodes in the binary search tree, all unsuccessful searches terminate only at the external nodes in the appropriate portion of the binary search tree. Hence external nodes are also referred to as *failure nodes*. Thus if the inorder traversal of a binary search tree yields the keys in the sequence  $a_1 < a_2 < a_3 < \dots < a_n$  then the failure nodes  $e_0, e_1, e_2, e_3, \dots, e_n$  are all equivalence classes which represent cases of unsuccessful searches on the binary search tree. While  $e_0$  traps all unsuccessful searches of keys that are less than  $a_1$ ,  $e_1$  traps those that are greater than and less than  $a_2$  and so on. In general,  $e_i$  traps all keys between  $a_i$  and  $a_{i+1}$  which are unsuccessfully searched. For example, in Fig. 10.2(b), all keys less than  $B$  which result in unsuccessful searches terminate at the external node and those which are greater than  $Z$  terminate at the external node and so on.

Algorithm 10.1 illustrates the procedure to retrieve the location LOC of the node containing the element ITEM, from a binary search tree  $T$ .

**Algorithm 10.1:** Procedure to retrieve ITEM from a binary search tree  $T$

```

procedure FIND_BST( $T$ , ITEM, LOC)
    /* LOC is the address of the node containing ITEM which
       is to be retrieved from the binary search tree  $T$ . In case
       of unsuccessful search the procedure prints the message
       "ITEM not found" and returns LOC as NIL.*/
    if  $T = \text{NIL}$  then {print (" binary search tree  $T$  is empty");
                        exit; }                      /* exit procedure*/
    else
        LOC =  $T$ ;
        while (LOC  $\neq \text{NIL}$ ) do
            case
                :ITEM = DATA(LOC): return (LOC);      /* ITEM found in node LOC*/
                :ITEM < DATA(LOC): LOC = LCHILD(LOC); /* search left subtree*/
                :ITEM > DATA(LOC): LOC= RCHILD(LOC); /* search right subtree*/
            endcase
        endwhile
        if (LOC=NIL) then {print("ITEM not found"); return (LOC)}
                           /* unsuccessful search*/
    end FIND_BST

```

Why are binary search tree retrievals more efficient than sequential list retrievals?

For a list of  $n$  elements stored as a sequential list, the worst case time complexity of searching an element both in the case of successful search or unsuccessful search is  $O(n)$ . This is so since in the worst case, the search key needs to be compared with every element of the list. However, in the

case of a binary search tree as is evident in Algorithm 10.1, searching for a given key  $k$  results in discounting half the binary search tree at every stage of its comparison with a node on its path from the root downwards. The best case time complexity for the retrieval operation is therefore  $O(1)$  when the search key  $k$  is found in the root itself. The worst case occurs when the search key is found in one of the leaf nodes whose level is equal to the height  $h$  of the binary search tree. The time complexity of the search is then given by  $O(h)$ . In some cases binary search trees may grow to heights that equal  $n$ , the number of elements in the associated set, thereby increasing the time complexity of a retrieval operation to  $O(n)$  in the worst case (see Sec. 10.2). However, on an average assuming random insertions/deletions, we obtain the height  $h$  of the binary search tree to be  $O(\log n)$  yielding a time complexity of  $O(\log n)$  for a retrieval operation.

**Example 10.1** Consider the set  $S = \{416, 891, 456, 765, 111, 654, 345, 256, 333\}$  whose associated binary search tree  $T$  is shown in Fig. 10.3.

Let us retrieve the keys 333 and 777 from the binary search tree. Tables 10.1 and 10.2 show the trace of the Algorithm FIND for the retrieval of the two keys respectively. Here  $\#(n)$  where  $n \in S$  indicates the location (address) of the node containing the key  $n$ . While retrieval of 333 yields a successful search terminating at node  $\#(333)$ , retrieval of 777 results in an unsuccessful search terminating at the appropriate external node.

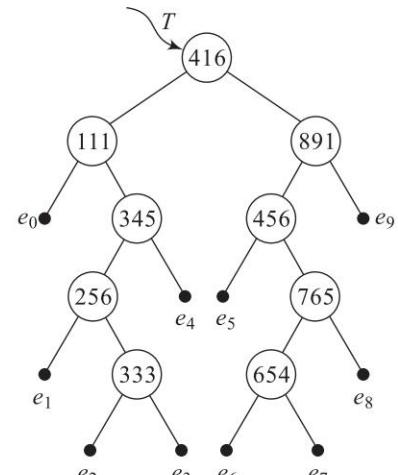


Fig. 10.3 A binary search tree

Table 10.1 Trace of Algorithm 10.1 for the retrieval of ITEM=333

| LOC                           | <<br>ITEM = DATA(LOC) ?<br>> | Updated LOC                                       |
|-------------------------------|------------------------------|---------------------------------------------------|
| Initially<br>LOC = T = #(416) | 333 < 416                    | LOC = LCHILD(#(416)) = #(111)                     |
| LOC = #(111)                  | 333 > 111                    | LOC = RCHILD(#(111)) = #(345)                     |
| LOC = #(345)                  | 333 < 345                    | LOC = LCHILD(#(345)) = #(256)                     |
| LOC = #(256)                  | 333 > 256                    | LOC = RCHILD(#(256)) = #(333)                     |
| LOC = #(333)                  | 333 = 333                    | RETURN(#(333))<br>Element found and node returned |

Table 10.2 Trace of Algorithm 10.1 for the retrieval of ITEM=777

| LOC                                                                                         | <<br>ITEM = DATA(LOC) ?<br>>                     | Updated LOC                                                                                                                                                            |
|---------------------------------------------------------------------------------------------|--------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Initially LOC = T = # (416)<br>LOC = # (891)<br>LOC = # (456)<br>LOC = # (765)<br>LOC = NIL | 777 > 416<br>777 < 891<br>777 > 456<br>777 > 765 | LOC = RCHILD(# (416)) = # (891)<br>LOC = LCHILD(# (891)) = # (456)<br>LOC = RCHILD(# (456)) = # (765)<br>LOC = RCHILD(# (765)) = NIL<br>Element not found RETURN (NIL) |

## Insertion into a binary search tree

The insertion of a key into a binary search tree is similar to the retrieval operation. The insertion of a key  $u$  initially proceeds as if it were trying to retrieve the key from the binary search tree, but on reaching the null pointer (failure node) which it is sure to encounter since key  $u$  is not present in the tree, a new node containing the key  $u$  is inserted at that position.

**Example 10.2** Let us insert keys 701 and 332 into the binary search tree  $T$  associated with set  $S = \{416, 891, 456, 765, 111, 654, 345, 256, 333\}$ , shown in Fig. 10.3. Figure 10.4 (a) shows the insertion of 701. Note how the operation moves down the tree in the path shown and when it encounters a failure node  $e_7$ , the key 701 is inserted as the right child of node containing 654. Again the insertion of 332 which follows a similar procedure is illustrated in Fig. 10.4(b).

The algorithm for the insert procedure is only a minor modification of Algorithm 10.1. The time complexity of an insert operation is also  $O(\log n)$ .

## Deletion from a binary search tree

The deletion of a key from the binary search tree is comparatively not as straight as the insertion operation. We first search for the node containing the key by undertaking a retrieval operation. But once the node is identified, the following cases are tested before the node containing the key  $u$  is appropriately deleted from the binary search tree  $T$ :

- (i) key  $u$  is a leaf node
- (ii) key  $u$  has a lone subtree (left subtree or right subtree only)
- (iii) key  $u$  has both left subtree and right subtree

**Case (i)** If the key  $u$  to be deleted is a leaf node then the deletion is trivial since the appropriate link field of the parent node of key  $u$  only needs to be set as NIL. Figure 10.5(a) illustrates this case.

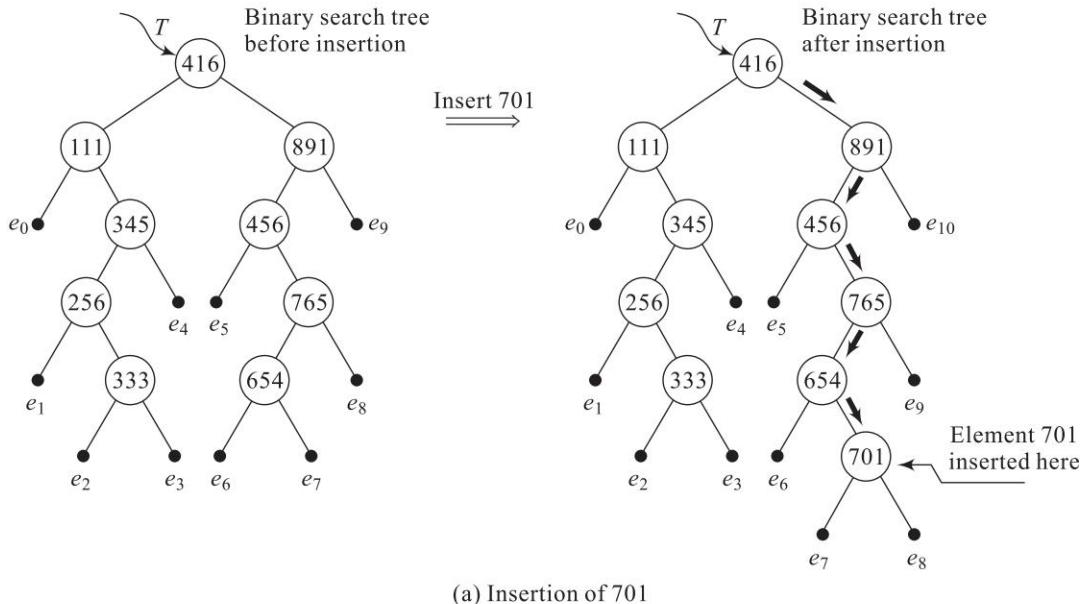
**Case (ii)** If the key  $u$  to be deleted has either a left subtree or a right subtree (but not both) then the link of the parent node of  $u$  is set to point to the appropriate subtree. Figure 10.5(b) illustrates the case.

**Case (iii)** If the key  $u$  to be deleted has both a left subtree and a right subtree, then the problem is complicated. In this case since the right subtree comprises keys that are greater than  $u$ , the parent node of key  $u$  is now set to point to the right subtree of  $u$ . Now where do we accommodate the left subtree of  $u$ ? Since all the keys of the left subtree of  $u$  are less than that of the right subtree of  $u$ , we move as far left as possible in the right subtree of  $u$  until an empty left subtree is found and link the left subtree of  $u$  at that position. Figure 10.5(c) illustrates the case.

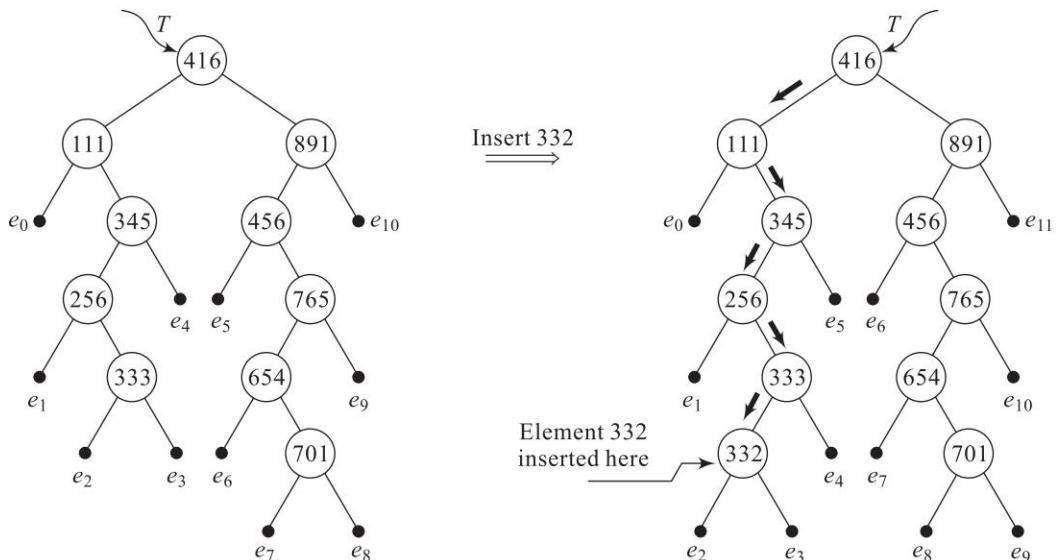
The other methods of deletion in this case include replacing the key  $u$  with either the largest key  $l$  occurring in the left subtree of  $u$  or the smallest key  $s$  in the right subtree of  $u$ . It is guaranteed that  $l$  or  $s$  will turn out to be a node with either empty subtrees or any one non empty subtree. After replacing  $u$  with  $l$  or  $s$  as the case may be, the nodes carrying  $l$  or  $s$  are deleted from the tree using the appropriate procedure [Case (i) or Case (ii)].

**Example 10.3** Delete keys 333, 891 and 416 in the order given, from the binary search tree  $T$  associated with set  $S = \{416, 891, 456, 765, 111, 654, 345, 256, 333\}$  shown in Fig. 10.3.

## Binary Search Trees and AVL Trees



(a) Insertion of 701

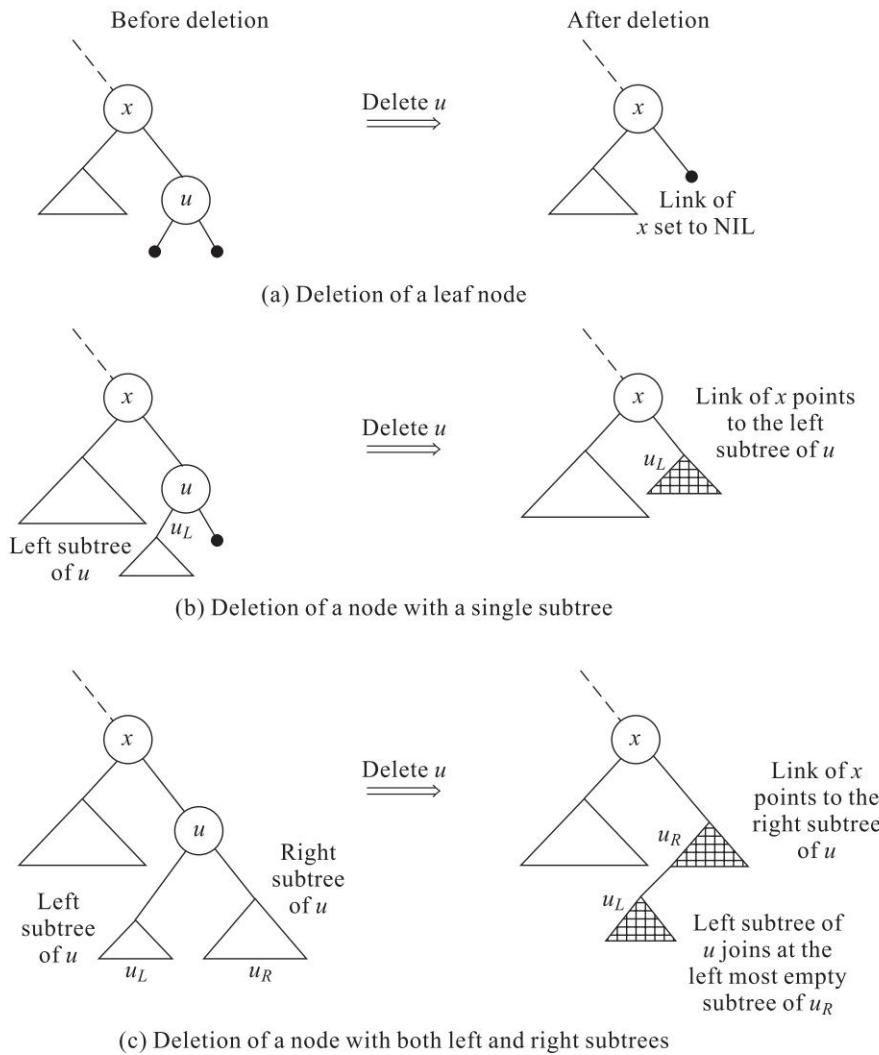


(b) Insertion of 332

**Fig. 10.4** Insertion of elements 701 and 332 into the binary search tree shown in Fig. 10.3

Deletion of 333, a leaf node, illustrates case(i). The RCHILD link of node #(256) is set to NIL. Figure 10.6(a) shows the deletion.

Deletion of 891, a node with a single subtree (left subtree), illustrates case(ii). In this case the RCHILD link of node #(416) is set to point to node #(456). Figure 10.6(b) illustrates the deletion.

**Fig. 10.5** *Deletion of a key from a binary search tree*

Lastly, the deletion of 416, the root node with both the left and right subtrees intact, results in node #(456) taking over as the root. However, the left subtree of the root viz., the subtree with node #(111) as the root, attaches itself as far left of the right subtree of node #(416). It therefore attaches itself to the LCHILD of node #(456). Figure 10.6(c ) illustrates this case.

Algorithm 10.2 illustrates the deletion procedure on a binary search tree given  $\text{NODE\_}_U$ , the node to be deleted and  $\text{NODE\_}_X$  its parent. For simplicity, the procedure illustrates only the deletion operation for all non empty nodes  $\text{NODE\_}_U$  other than the root. A general procedure to delete any  $ITEM$  from a binary search tree  $T$  can be easily attempted (Programming Assignment 1 (Chapter 10)). The time complexity of the delete operation on a binary search tree is  $O(\log n)$ .

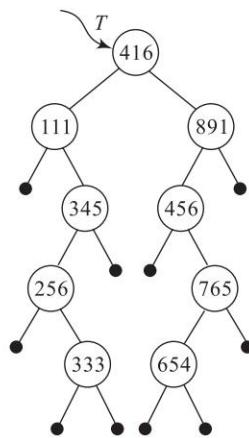
**Algorithm 10.2:** Procedure to delete a node NODE\_U from a binary search tree given its parent node NODE\_X

```

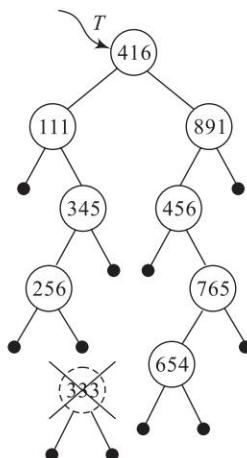
procedure DELETE (NODE_U, NODE_X)
    /* NODE_U is the node which is to be deleted from the
       binary search tree. NODE_X is the parent node for which
       NODE_U may be the left child or the right child.
       Procedure DELETE is applicable for deletion of all non
       empty nodes other than the root (i.e.) NODE_U ≠ NIL and
       NODE_X ≠ NIL */
case
: LCHILD(NODE_U) = RCHILD(NODE_U) = NIL: /* NODE_U is a leaf node*/
    Set RCHILD(NODE_X) or LCHILD(NODE_X) to NIL based on
        whether NODE_U is the right child or left child of
        NODE_X respectively;
    call RETURN(NODE_U); /* dispose node to the
                           Available space list*/
:LCHILD(NODE_U) <> NIL and RCHILD(NODE_U) <> NIL: /* NODE_U has both left
   and right subtrees*/
    /* attach right subtree of NODE_U to NODE_X */
    Set RCHILD(NODE_X) or LCHILD(NODE_X) to
    RCHILD(NODE_U) based on whether NODE_U is the
    right child or left child of NODE_X respectively;
    /* attach left subtree of NODE_U as far left of
       the right subtree of NODE_U as possible*/
    TEMP= RCHILD(NODE_U);
    while (LCHILD(TEMP) <> NIL) do
        TEMP=LCHILD(TEMP);
    endwhile
    LCHILD(TEMP) = LCHILD(NODE_U);
    call RETURN (NODE_U);
:LCHILD (NODE_U) <> NIL and RCHILD(NODE_U) = NIL: /*NODE_U has only left
   subtree*/
    TEMP=LCHILD(NODE_U);
    Set RCHILD(NODE_X) or LCHILD(NODE_X) to TEMP based
    on whether NODE_U is the right child or left child
    of NODE_X respectively;
    call RETURN(NODE_U);
:LCHILD(NODE_U) = NIL and RCHILD(NODE_U) <> NIL: /*NODE_U has only right
   subtree*/
    TEMP=RCHILD(NODE_U);
    Set RCHILD(NODE_X) or LCHILD(NODE_X) to TEMP based
    on whether NODE_U is the right child or left child
    of NODE_X respectively;
    call RETURN(NODE_U);
endcase
end DELETE

```

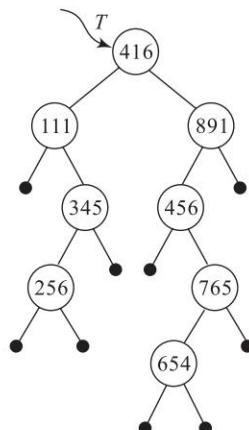
Binary search tree before deletion



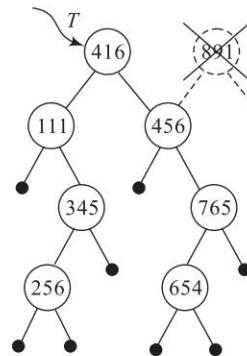
Binary search tree after deletion



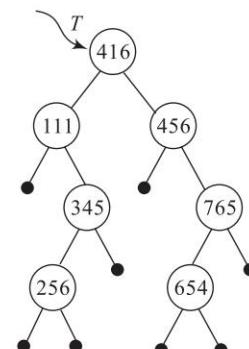
(a) Delete 333



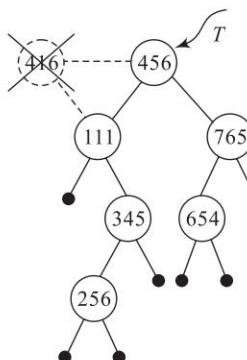
Delete 891



(b) Delete 891



Delete 416



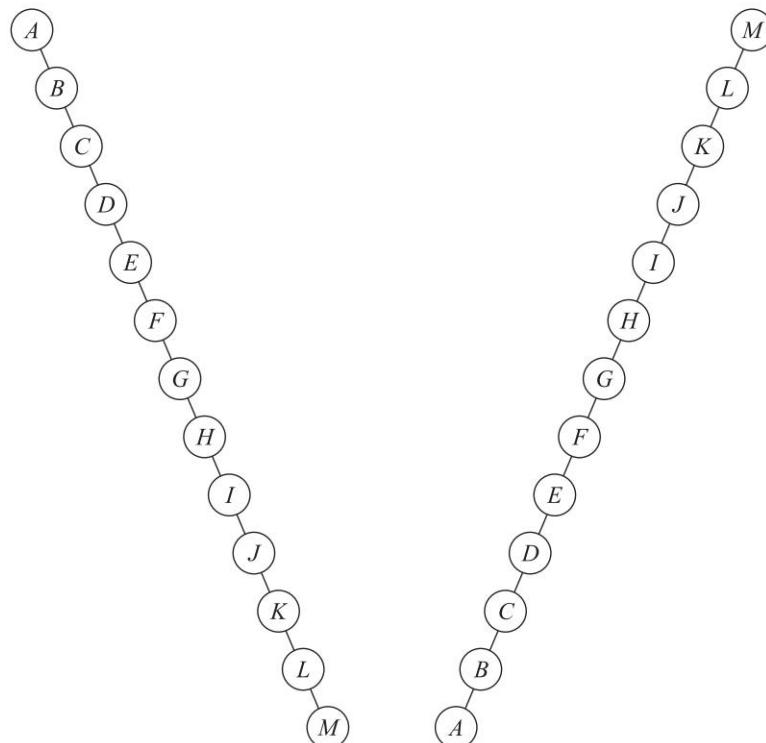
(c) Delete 416

**Fig. 10.6** Deletion of keys 333, 891 and 416 from the binary search tree shown in Fig. 10.3

## Drawbacks of a binary search tree

Though binary search trees in comparison to sequential lists report a better performance of  $O(\log n)$  time complexity for their insert, delete and retrieval operations, they are not without their setbacks. As pointed out in Sec. 10.2, there are instances where binary search trees may grow to heights that equal  $n$ , the number of elements to be represented as the tree, thereby deteriorating their performance. This may occur due to a sequence of insert operations or delete operations. Examples 10.4 and 10.5 illustrate instances when the height of a binary search tree reaches  $n$ .

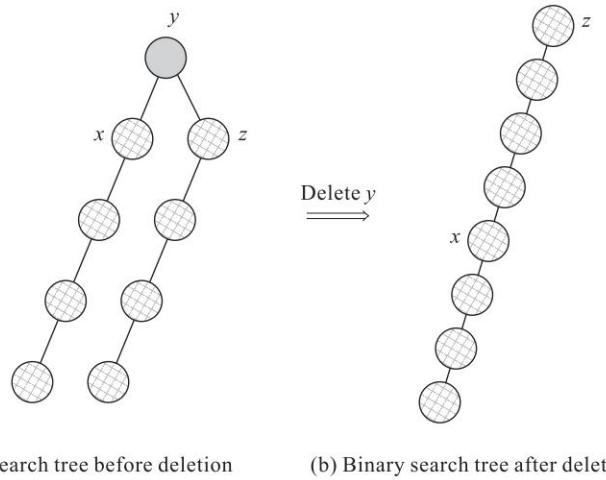
**Example 10.4** Let us construct binary search trees for the sets  $S_1 = \{A, B, C, D, E, F, G, H, I, J, K, L, M\}$  and  $S_2 = \{M, L, K, J, I, H, G, F, E, D, C, B, A\}$ . It can be seen that while the elements of  $S_1$  are in the ascending order of the alphabetical sequence, those in  $S_2$  are in the descending order of the sequence. The respective binary search trees are shown in Fig. 10.7.



**Fig. 10.7** Binary search trees for the sets  $S_1 = \{A, B, C, D, E, F, G, H, I, J, K, L, M\}$  and  $S_2 = \{M, L, K, I, H, G, F, E, D, C, B, A\}$

Observe that the two binary search trees are right skewed and left skewed respectively. In such a case, the height  $h$  of the binary search tree is equal to  $n$  and hence a search operation on these binary search trees in the worst case would yield  $O(n)$  time complexity.

**Example 10.5** Consider a skeletal binary search tree shown in Fig. 10.8 (a). Deletion of node  $y$  in the tree yields the one shown in Fig. 10.8(b). Here again it may be seen that the binary search tree after deletion has yielded a left skewed binary tree once again resulting in  $O(n)$  time complexity in the event of a search operation.



**Fig. 10.8** Deletion from a binary search tree resulting in a skewed binary tree

It is clear from the above examples that if the height of the binary search tree is left unchecked for it can result in skewed binary trees deteriorating their performance. In other words it is essential that the binary search trees are maintained so as to have a *balanced height*. Trees whose height in the worst case yields  $O(\log n)$  are known as *balanced trees*. *AVL trees* are one such trees and is discussed in Sec. 10.3.

## AVL Trees: Definition and Operations

## 10.3

In Sec. 10.2 it was pointed out how binary search trees can reach heights equal to  $n$ , the number of elements in the tree, thereby deteriorating its performance. To eliminate this drawback it is essential that during an insert or delete operation which can affect the structure of the tree and hence the height of the tree, it is ensured that the binary search tree remains of balanced height. In other words, there needs to be a mechanism to ensure that an insert or delete operation does not turn the tree into a skewed one. As mentioned earlier, trees whose height in the worst case turns out to be  $O(\log n)$  are known as *balanced trees* or *height balanced trees*. One such balanced tree viz., AVL trees are discussed in this section. AVL trees were proposed by Adelson-Velskii and Landis in 1962.

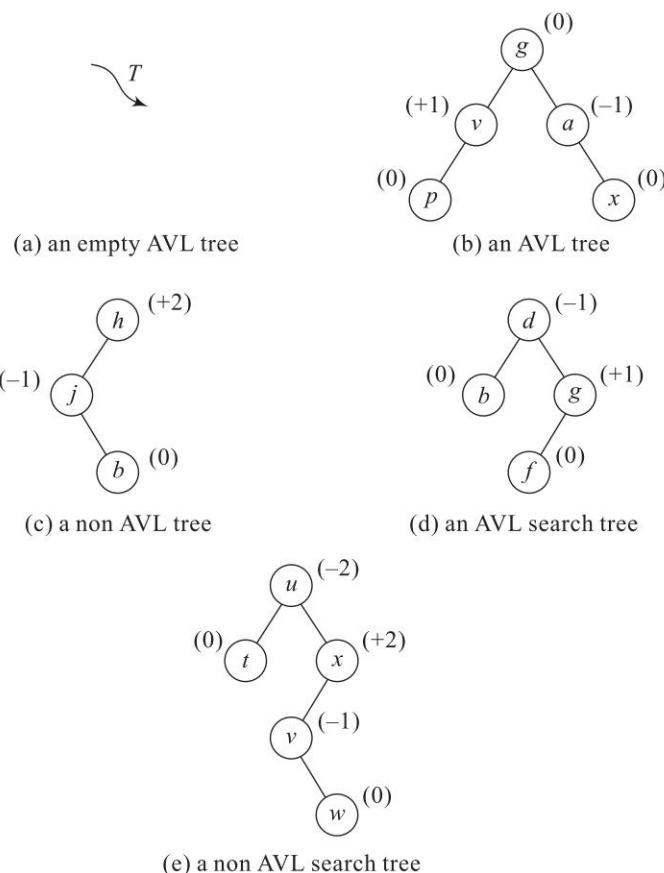
## Definition

An empty binary tree is an *AVL tree*. If non empty, the binary tree  $T$  is an AVL tree if (i)  $T_L$  and  $T_R$ , the left and right subtrees of  $T$  are also AVL trees and (ii)  $|h(T_L) - h(T_R)| \leq 1$ , where  $h(T_L)$  and  $h(T_R)$  are the heights of the left subtree and right subtree of  $T$  respectively.

For a node  $u$ ,  $bf(u) = (h(u_L) - h(u_R))$  where  $h(u_L)$  and  $h(u_R)$  are the heights of the left and right subtrees of the node  $u$  respectively, is known as the *balance factor* ( $bf$ ) of the node  $u$ . In an AVL tree therefore, every node  $u$  has a balance factor  $bf(u)$  which may be either 0 or +1 or -1.

A binary search tree  $T$  which is an AVL tree is referred to as an *AVL search tree*. This section elaborates on the operations of insert, delete and retrieval performed on AVL search trees.

Figure 10.9 illustrates examples of AVL trees and AVL search trees. The balance factor of each of the nodes is indicated by the side of the node within parentheses. Note how the balance factors of the nodes in the AVL trees are either 0 or +1 or -1.



**Fig. 10.9 Examples of AVL trees and non AVL trees**

AVL trees and AVL search trees just like binary trees or binary search trees may be represented using a linked representation adopting the same node structure (see Sec. 8.5 and Sec. 10.2). However to facilitate efficient rendering of insert and delete procedures, a field termed BF may be included in the node structure to record the balance factor of the specific node.

## Retrieval from an AVL search tree

The retrieval of a key from an AVL search tree is in no way different from the retrieval operation on a binary search tree. Algorithm 10.1 illustrating the find operation on a binary search tree  $T$  may be utilized for retrieval of an element from an AVL search tree as well. However, since the height of the AVL search tree of  $n$  elements is  $O(\log n)$ , the time complexity of the find procedure when applied on AVL search trees does not exceed  $O(\log n)$ .

## Insertion into an AVL search tree

The insertion of an element  $u$  into an AVL search tree  $T$  proceeds exactly as one would to insert  $u$  in a binary search tree. However, if after insertion the balance factors of any of the nodes turns out to be anything other than 0 or +1 or -1, then the tree is said to be *unbalanced*. To balance the tree we undertake what are called *rotations*. Rotations are mechanisms which shift some of the subtrees of the unbalanced tree to obtain a balanced tree.

With regard to rotations there are some important observations which are helpful in the implementation of the operations on AVL trees. For the initiation of rotations, it is required that the balance factors of all nodes in the unbalanced tree are limited to -2, -1, 0, 1, and +2. Also the rotation is initiated with respect to an ancestor node  $A$  that is closest to the newly inserted node  $u$  and whose balance factor is either +2 or -2. If a node  $w$  after insertion of node  $u$  reports a balance factor of  $bf(w) = +2$  or  $-2$  respectively, then its balance factor before insertion should have been +1 or -1 respectively. The insertion of a node can only change the balance factors of those nodes on the path from the root to the inserted node. If the closest ancestor node  $A$  of the inserted node  $u$  has a balance factor  $bf(A) = +2$  or  $-2$ , then prior to insertion the balance factors of all nodes on the path from  $A$  to  $u$  must have been 0. In fact these observations are vital to determining the closest ancestor  $A$  after insertion of  $u$ .

The rotations which are of four different types are listed below. The classification is based on the position of the inserted node  $u$  with respect to the ancestor node  $A$  which is closest to the node  $u$  and reports a balance factor of -2 or +2.

- (i) *LL rotation*—node  $u$  is inserted in the left subtree ( $L$ ) of left subtree ( $L$ ) of  $A$
- (ii) *LR rotation*—node  $u$  is inserted in the right subtree ( $R$ ) of left subtree ( $L$ ) of  $A$
- (iii) *RR rotation*—node  $u$  is inserted in the right subtree ( $R$ ) of right subtree ( $R$ ) of  $A$
- (iv) *RL rotation*—node  $u$  is inserted in the left subtree ( $L$ ) of right subtree ( $R$ ) of  $A$

Each of the four classes of rotations are illustrated with examples.

### LL rotation

Figure 10.10 illustrates a generic representation of *LL* type imbalance and the corresponding rotation that is undertaken to set right the imbalance. After insertion of node  $u$ , the closest ancestor node of node  $u$ , viz., node  $A$ , reporting an imbalance ( $bf(A) = +2$ ) is first found out. For simplicity of discussion, the generic tree shown in Fig. 10.10(a) has been so chosen to have the ancestor node  $A$  occurring at the root. In reality the ancestor node  $A$  may occur anywhere down the tree. Now with reference to the ancestor node  $A$ , we find that the node  $u$  has been inserted in the left subtree ( $L$ ) of left subtree ( $L$ ) of  $A$ . This implies there is an *LL* type of imbalance and to balance the tree an *LL* rotation is to be called for. The AVL tree before insertion of  $u$  (Fig. 10.10(a)), the unbalanced tree after insertion of  $u$  (Fig. 10.10(b)) and the balanced tree after the *LL* rotation (Fig. 10.10(c)) have been illustrated.

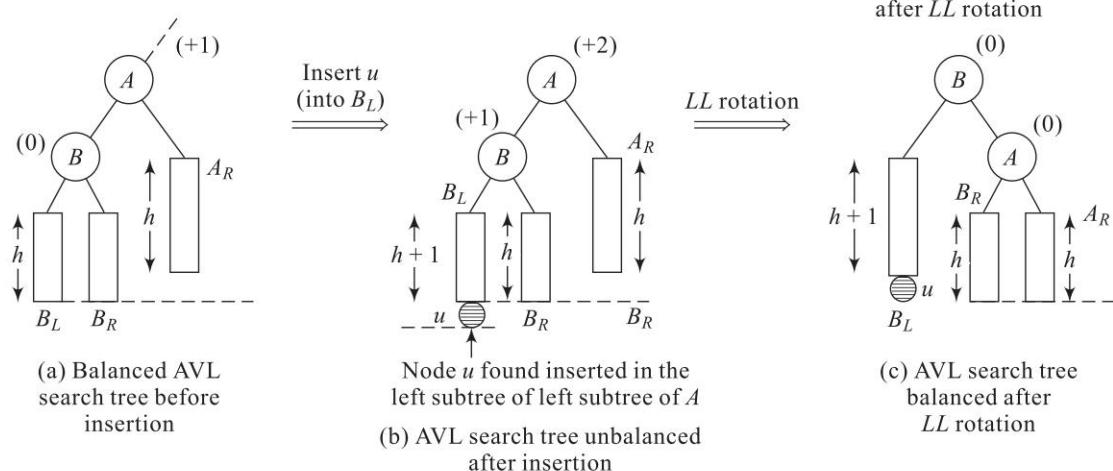


Fig. 10.10 Generic representation of an LL rotation

Here  $u$  is found inserted in the left subtree of  $B$ , viz.,  $B_L$  where  $B$  is in the left subtree of  $A$ . We assume the heights of the generic subtrees  $A_R$ ,  $B_L$  and  $B_R$  to be  $h$ . Observe the imbalance in the balance factor of  $A$  after insertion of  $u$ .  $bf(A)$  which was +1 before insertion of  $u$  changes to +2 after insertion. To balance the tree, the *LL* rotation pushes  $B$  up as the root of the AVL tree which results in node  $A$  slumping downwards to its left along with its right subtree  $A_R$ . Now the tree is rearranged by shifting the right subtree of  $B$ , viz.,  $B_R$  to join  $A$  as its left subtree, leaving  $B_L$  (holding the inserted node  $u$ ) undisturbed as the left subtree of  $B$ .

**Example 10.5** Consider the AVL search tree shown in Fig. 10.11(a). Let us insert  $C$  into the AVL search tree. To facilitate ease of understanding, the notations employed in the generic tree of Fig. 10.10 have been mapped to the given tree. Note how  $C$  finds itself inserted in the left subtree of left subtree of  $M$ , the closest ancestor node of  $C$  that shows  $bf(M) = +2$  after insertion.

The *LL* rotation pushes  $F$  up, to become the root of the tree and shifts the subtree with node  $K$  which was originally the right subtree of  $F$ , to the left subtree of  $M$ .

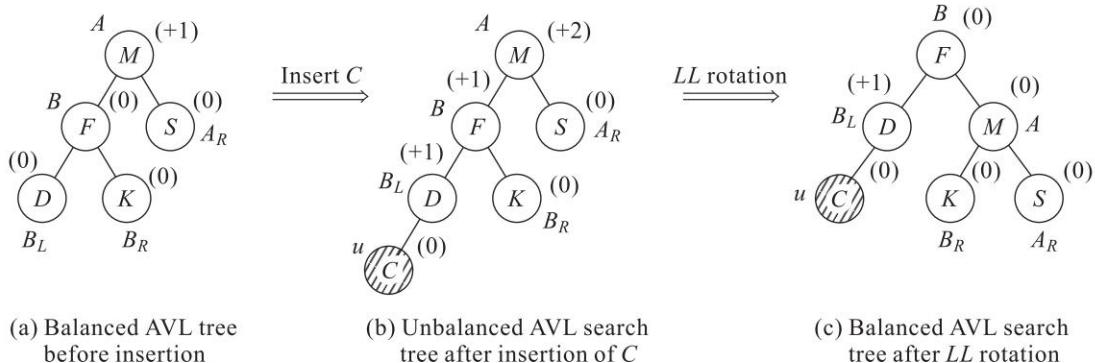


Fig. 10.11 An example of LL rotation

## LR rotation

Figure 10.12 illustrates the generic representation of an *LR* type of imbalance and the corresponding rotation that is undertaken to set right the imbalance. Here the node  $u$  on insertion finds  $A$  to be its closest ancestor node that is unbalanced and with reference to node  $A$  is inserted in the right subtree of left subtree of  $A$ . This therefore is an *LR* type of imbalance and calls for *LR* rotation to balance the tree. The AVL tree before insertion of  $u$  (Fig. 10.12 (a)), the unbalanced tree after insertion of  $u$  (Fig. 10.12(b)) and the balanced tree after the *LR* rotation (Fig. 10.12(c)) have been illustrated.

Here  $u$  finds itself inserted in the right subtree of left subtree of  $A$ , the closest ancestor node. The heights of the subtrees  $A_{R'}, B_L, B_R, C_L$  and  $C_R$  are as shown in the figure. Let us suppose  $u$  is found in  $C_L$  the left subtree of  $C$ . The procedure is no way different if  $u$  is found in  $C_R$  the right subtree of  $C$ . The *LR* rotation rearranges the tree by first shifting  $C$  to the root node. Then the left subtree  $C_L$  of  $C$  is shifted to the right subtree of  $B$  and  $C_R$  the right subtree of  $C$  is shifted to the left subtree of  $A$ . The rearranged tree is balanced. In the case of *LR* rotation, the following observations hold:

If  $BF(C) = 0$  after insertion of new node then  $BF(A)=BF(B)=0$  after rotation

If  $BF(C) = -1$  after insertion of new node then  $BF(A)= 0$ ,  $BF(B)=+1$  after rotation

If  $BF(C) = +1$  after insertion of new node then  $BF(A)=-1$ ,  $BF(B)=0$  after rotation

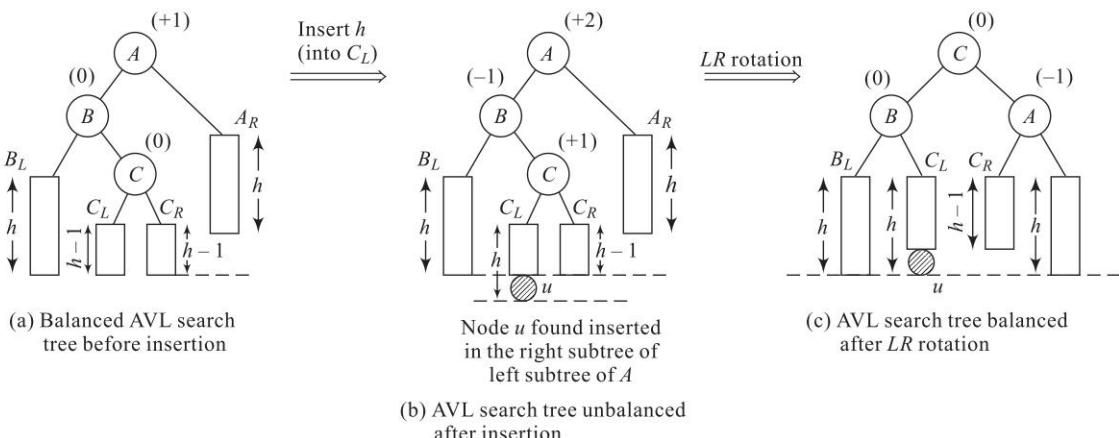


Fig. 10.12 Generic representation of an *LR* rotation

**Example 10.6** Consider the AVL search tree shown in Fig. 10.13. The subtrees  $C_L$  and  $C_R$  of node  $C$  in the generic representation shown in Fig. 10.12 are mapped to empty subtrees in this tree. In other words, the node labeled  $L$  has empty left and right subtrees. Let us insert  $H$  into the AVL search tree. Note how  $H$  gets inserted into the right subtree of left subtree of  $S$ , the closest ancestor node of  $H$  that shows  $bf(S)= +2$ . The *LR* rotation rearranges the tree by first pushing node  $L$  to be the root. As a result, node  $S$  slumps to its right along with its right subtree comprising the element  $W$ . Thereafter, the original left subtree of  $L$  holding the newly inserted node  $H$  is attached to  $F$  as its right subtree. In the absence of a right subtree for  $L$  (which was so before rotation), only an empty tree is attached as the left subtree of  $S$ .

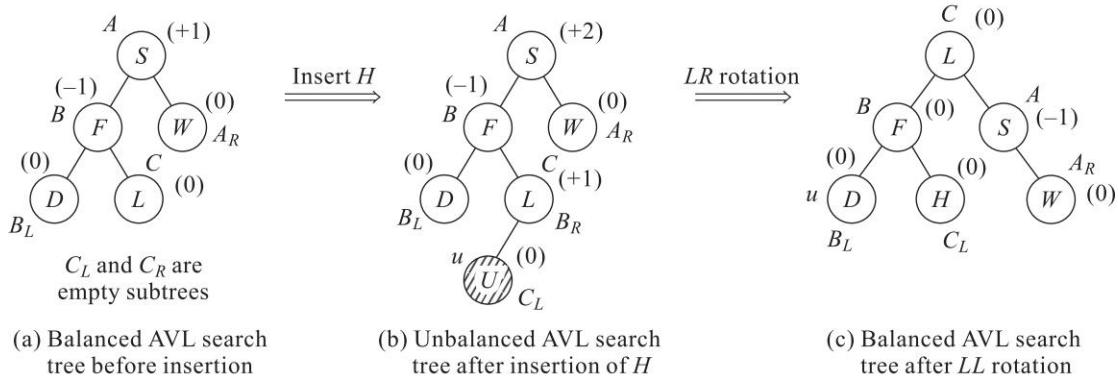


Fig. 10.13 An example of LR rotation

## RR rotation

The RR rotation is symmetric to the LL rotation. Figure 10.14 illustrates the generic representation of the RR rotation scheme. Observe how node *u* finds itself inserted in the right subtree of right subtree of *A*, the closest ancestor node that is unbalanced and the rotation is merely a mirror image of the LL rotation scheme.

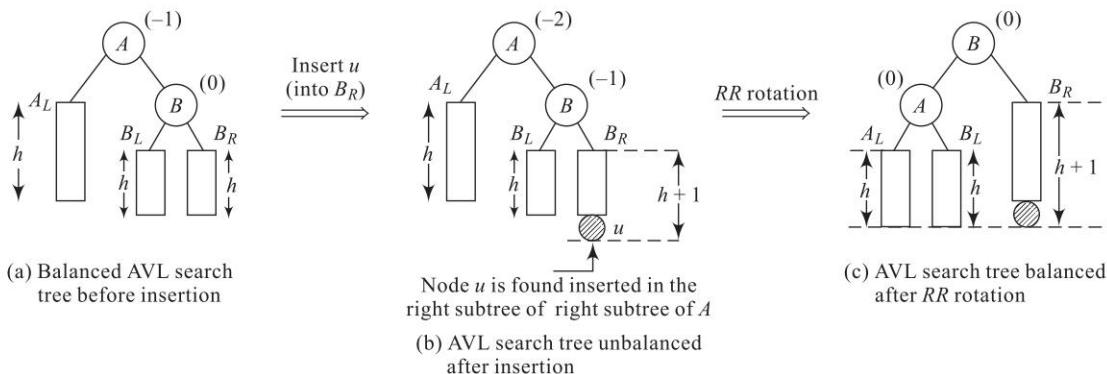
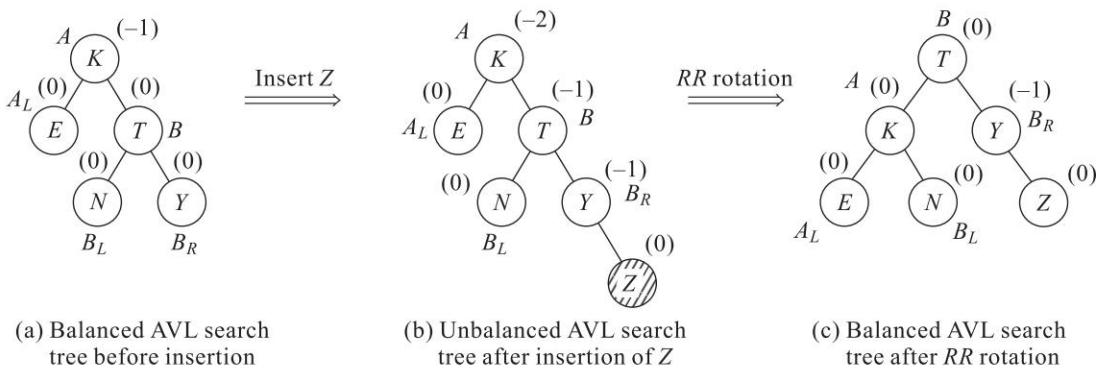
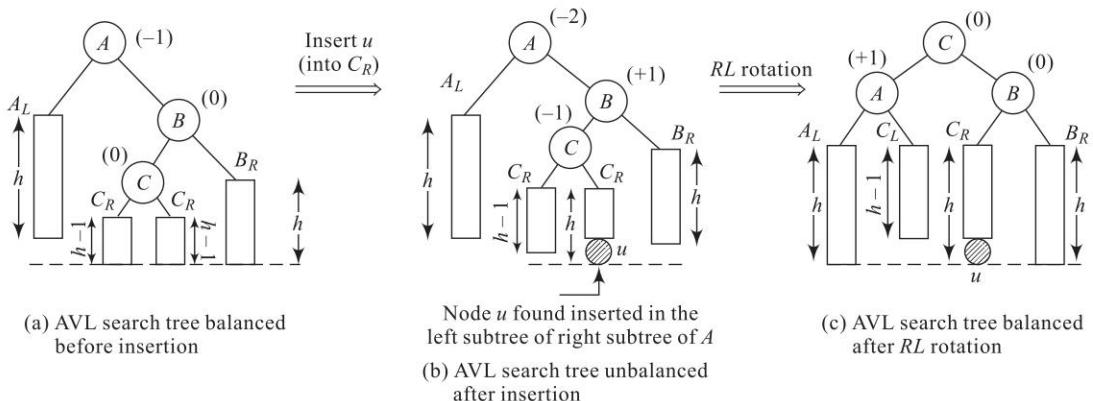


Fig. 10.14 Generic representation of an RR rotation

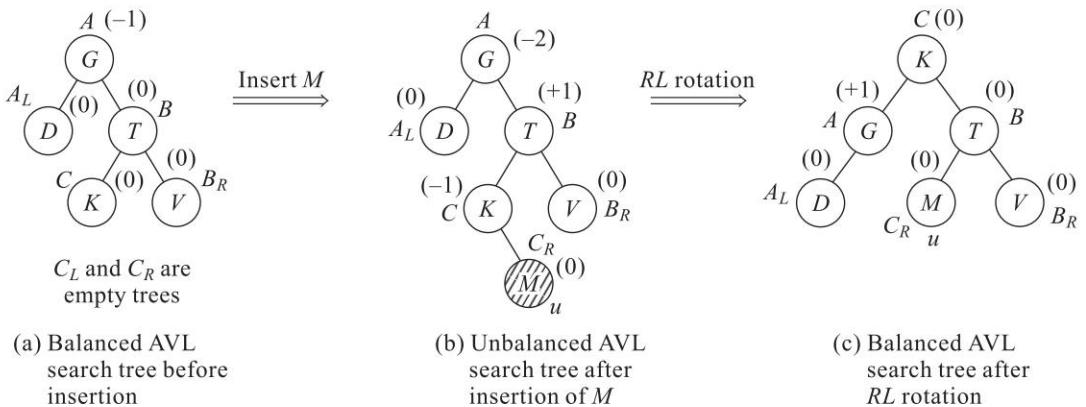
**Example 10.6** Consider the AVL search tree shown in Fig. 10.15. The insertion of *Z* calls for an RR rotation. The unbalanced AVL search tree and the balanced tree after RR rotation have been shown in Figs 10.15 and 10.15 respectively.

## RL rotation

RL rotation is symmetric to LR rotation. Figure 10.16 illustrates the generic representation of the RL rotation scheme. Here node *u* finds itself inserted in the left subtree of right subtree of node *A* which is the closest ancestor node that is unbalanced. Note how the RL rotation is the mirror image of the LR rotation scheme. As pointed out for the LR rotation scheme, the rotation procedure for RL remains the same irrespective of *u* being inserted in *C<sub>L</sub>* or *C<sub>R</sub>*, the left subtree and right subtree of *C* respectively.

Fig. 10.15 An example of  $RR$  rotationFig. 10.16 Generic representation of an  $RL$  rotation

**Example 10.7** Consider the AVL search tree shown in Fig. 10.17(a). The insertion of  $M$  calls for an  $RL$  rotation. The unbalanced AVL search tree and the balanced tree after  $RL$  rotation have been shown in Figs 10.17(b) and 10.17(c) respectively.

Fig. 10.17 An example of  $RL$  rotation

In the above classes of rotations, *LL* and *RR* are called as *single rotations* and *LR* and *RL* are called as *double rotations*. An *LR* rotation is a combination of *RR* rotation followed by an *LL* rotation and *RL* rotation is a combination of *LL* rotation followed by an *RR* rotation.

Algorithm 10.3 illustrates a skeletal procedure to insert an element into an AVL search tree. The procedure initially tries to identify the most recent ancestor node *A* of the inserted element whose  $bf(A) = \pm 1$ . If no such node *A* is found then all nodes in the path from the root to the newly inserted node have a balance factor of 0 at the time of insertion and hence the tree cannot go unbalanced due to the insertion. In such a case we only update the balance factors of the nodes in the path from the root to the newly inserted node by updating the BF value of the node to +1 if ITEM is inserted in its left subtree and to -1 if it is inserted in its right subtree.

**Algorithm 10.3:** Skeletal procedure to insert an ITEM into an AVL search tree *T*

```

procedure INSERT(T, ITEM)
    /*Steps to insert an ITEM into an AVL search tree T.
       The node structure comprises the fields LCHILD, DATA,
       BF and RCHILD representing left child link, data,
       balance factor and right child link*/
    call GETNODE(X);      /* get ready new node X containing ITEM/
    DATA(X)=ITEM,
    LCHILD(X)=RCHILD(X)=NIL and BF(X)=0;
                           /* AVL search tree T is empty*/
    if (T=NIL) then { Set T to X;
        exit;
    }
    /* AVL search tree T is non empty and ITEM is distinct
       from other elements in T */

Find node P where ITEM is to be inserted as either the left child or right child of P by following a path from the root onwards. Also, while traversing down the tree in search of the point of insertion of ITEM, take note of the most recent ancestor node A whose  $bf(A) = \pm 1$ ;  

Insert node X carrying ITEM as the left or right child of node P;  

/*if no ancestor node A is found the balance factors of
all nodes on the path from the root to the node
containing ITEM is 0. The tree will therefore remain
balanced even after insertion of ITEM. Merely update the
BF fields of all the nodes on the path from the root to
node P after insertion of ITEM and exit*/  

if (node A not found) then {TEMP = T;  

    /* update BF field of node to +1 if ITEM is inserted in
       its left subtree and to -1 if inserted in its right
       subtree*/  

    while ( TEMP <> X) do  

        if (DATA(X) > DATA(TEMP))
```

```

        then
            { BF(TEMP)=-1;
              TEMP= RCHILD(TEMP);
            }
        else { BF(TEMP) = +1;
              TEMP= LCHILD(TEMP);
            }
        endwhile
        exit;

        /* if node A exists and BF(A)= +1 with ITEM inserted in the right subtree of A or BF(A)= -1 with ITEM inserted in the left subtree of A, then set BF(A)=0. Update the balance factors of all nodes in the path from node A to the inserted node X*/
if (node A found)
then
    { if (BF (A)= +1 and ITEM was inserted in the right subtree of A) or (BF (A)= -1 and ITEM was inserted in the left subtree of A)

    then {BF (A)=0;
          Update the balance factors of all nodes in the path from node A to the inserted node X;
          exit;
        }
    }
else
    /* AVL search tree T is unbalanced. Classify the imbalance and perform the appropriate rotations*/
    {
        Identify the type of imbalance and apply the appropriate rotations.
        Update the balance factors of the nodes as required by the rotation scheme as well as reset the LCHILD and RCHILD links of the appropriate nodes.
    }
end INSERT

```



If node  $A$  exists and  $bf(A) = +1$  and the insertion is done in the right subtree of  $A$  or if  $bf(A) = -1$  and the insertion is done in the left subtree of  $A$  then we set  $bf(A)=0$ . Also, we update the balance factors of all nodes in the path from the node  $A$  to the newly inserted node. In all other cases, the type of imbalance is identified and the appropriate rotations are carried out. This may call for updating the balance factors of the involved nodes as well as resetting the link fields of the relevant nodes after identifying the appropriate  $B$ ,  $C$ ,  $A_L$ ,  $A_R$ ,  $B_L$ ,  $B_R$ ,  $C_L$  and  $C_R$  relevant to the rotation scheme.

The time complexity of the insert operation is  $O(\text{height}) = O(\log n)$ .

## Deletion from an AVL search tree

To delete an element from an AVL search tree we discuss the operation based on whether the

node  $t$  carrying the element to be deleted is either a leaf node or one with a single non empty subtree or with two non empty subtrees. A delete operation just like an insert operation may also imbalance an AVL search tree. Just as LL/ LR/ RL/ RR rotations are called for to rebalance the tree after insertion, a delete operation also calls for rotations categorized as L and R. While the L category is further classified as L0,L1 and L – 1 rotations, the R category is further classified as R0, R1 and R – 1 rotations. The **Classify rotations for deletion** section of Algorithm 10.3 details the mode of classification of the L and R rotations. However, it needs to be remembered that not all deletions call for rotations.

Considering the complex nature of the operation we present its skeletal work procedure in two algorithms viz., Algorithms 10.3 and 10.4. While Algorithm 10.3 illustrates the case when node  $t$  is a leaf node or one with a single non empty subtree, Algorithm 10.4 illustrates the case when node  $t$  is one with two non empty subtrees. For the invocation of the algorithms we assume that the node holding the *ITEM* which is to be deleted from the AVL search tree  $T$ , viz., node  $t$  has already been found.

**Algorithm 10.4:** Skeletal procedure to delete an *ITEM* from a non empty AVL search tree  $T$  where node  $t$  in  $T$  holding *ITEM* is either a leaf node or one with a single subtree

```

Procedure DELETE1( $T$ , node  $t$ )
    /*Steps to delete an ITEM from a non-empty AVL search tree  $T$ .
    node  $t$  that holds ITEM and is either a leaf node or one with
    a single non empty subtree has been identified. The node
    structure comprises the fields LCHILD, DATA, BF and RCHILD
    representing left child link, data, balance factor and right
    child link respectively */

if node  $t$  is a leaf node or a node with a single child
then
    { Let node  $p$  be its parent node;
        Delete node  $t$  and reset the links of node  $p$  appropriately so as
        to either have a null link or to point to the lone child of node
         $t$  as is the case;
    }
else call DELETE2( $T$ , node  $t$ ); /* call procedure DELETE2( $T$ , node  $t$ )*/
Update balance factors:

Rule 1: With regard to node  $p$ , if node  $t$ 's deletion occurred in its right
        subtree then  $bf(p)$  increases by 1 and if it occurred in its left
        subtree then  $bf(p)$  decreases by 1.
Rule 2: If the new  $bf(p)=0$  then the height of the tree is decreased by
        1 and therefore this calls for updating the balance factors of
        its parent node and/or its ancestor nodes.
Rule 3: If the new  $bf(p) = \pm 1$ , then the height of the tree is the same
        as it was before deletion and therefore the balance factors of
        the ancestor nodes remains unchanged.
Rule 4: If the new  $bf(p)= \pm 2$ , then the node  $p$  is unbalanced and the
        appropriate rotations need to be called for.

```

**Classify rotations for deletion:**

While propagating the balance factor updates from the node  $p$  upwards to the root node there may be nodes whose balance factors are updated to  $\pm 2$ . Let  $A$  be the first such node on the path from node  $p$  to the root.

```
if the deletion took place on the right of  $A$ 
then classify the rotation as  $R$  or else classify it as  $L$ ;
```

For the  $R$  classification, if  $bf(A)=+2$  then it should have been  $+1$  before deletion and  $A$  should have a left subtree with root  $B$ . Based on  $bf(B)$  being either  $0$  or  $+1$  or  $-1$ , classify the  $R$  rotations further as  $R0$ ,  $R1$  and  $R-1$  respectively/\*See Sec. 10.3:  $R$  category rotations associated with the delete operation\*/

For the  $L$  classification, if  $bf(A)=-2$  then it should have been  $-1$  before deletion and  $A$  should have a right subtree with root  $B$ . Based on  $bf(B)$  being either  $0$  or  $+1$  or  $-1$ , classify the  $L$  rotations further as  $L0$ ,  $L1$  and  $L-1$  respectively/\*See Sec. 10.3:  $L$  category rotations associated with the delete operation\*/

**Perform rotation:**

Perform the appropriate rotations to balance the tree.

```
end DELETE1.
```



**Algorithm 10.5:** Skeletal procedure to delete an *ITEM* from a non empty AVL search tree  $T$  where node  $t$  in  $T$  holding *ITEM* is one with both a left and a right subtree

```
Procedure DELETE2( $T$ , node  $t$ )
```

```
/*Steps to delete an ITEM from a non-empty AVL search tree
 $T$ . node  $t$  holding ITEM and which has both a left and a right
non empty subtree has been identified. The node structure
comprises the fields LCHILD, DATA, BF and RCHILD representing
left child link, data, balance factor and right
child link. */
```

```
if node  $t$  is one with a non-empty left and right subtree
then
```

```
{
```

Find the smallest key in the right subtree of node  $t$ . This is obtained by moving down the RCHILD link of node  $t$  to reach node  $u$  and traversing the LCHILD links of the left subtree of node  $u$  until it is empty;

Let node  $v$  be the last node reached while traversing the left subtree of node  $u$  down the left child nodes.

Now,  $LCHILD(\text{node } v)=\text{NIL}$ . Let  $SUCC= \text{DATA}(\text{node } v)$ ;

Replace  $\text{DATA}(\text{node } t)$  with  $SUCC$ ;

Delete node  $v$  using Procedure  $\text{DELETE1}(T, \text{node } v)$ , since node  $v$  will either be a leaf node or one with a single subtree.

```
}
```

```
end DELETE2.
```

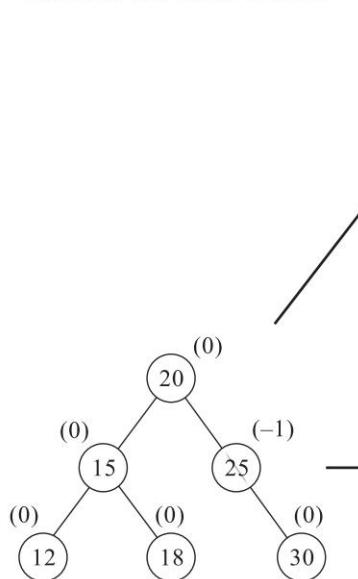


**Rotation free deletion of a leaf node** In the case of deletion of node  $t$  which is a leaf node, we physically delete the node and make the child link of its parent, viz., node  $p$  null. Now

update the balance factor of node  $p$  based on whether the deletion occurred to its right or left. If it had occurred in the right then we increase  $bf(\text{node } p)$  by 1 or else decrease  $bf(\text{node } p)$  by 1. The new updated value of  $bf(\text{node } p)$  is now tested against Rules 1–4 of Algorithm 10.3 for updating the balance factors of its ancestor nodes. In the event of any imbalance, the appropriate rotations viz.,  $L0/R0$ ,  $L1/R1$  or  $L-1/R-1$  are called for rendering the tree balanced.

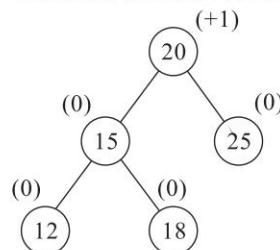
**Example 10.8** Consider the balanced AVL search tree shown in Fig. 10.18. Deletion of 30 is a case of deleting a leaf node. Figure 10.18(a) shows the balanced tree after deletion. Observe how the parent of node 30, viz., the node holding 25 resets its RCHILD link to NIL after the physical deletion of the node holding 30. Now how are the balance factors of the other nodes updated? Since the deletion of key 30 occurred to the right of key 25,  $bf(25)$  is increased by 1. Hence the new updated  $bf(25)$  is 0. As per Rule 2 outlined in Algorithm 10.3, this calls for the update of the balance factors of all ancestor nodes of key 25. Since the root (key 20) is the only ancestor node

Balanced tree before deletion

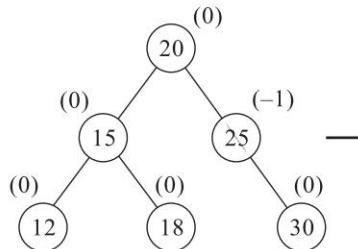


delete 30

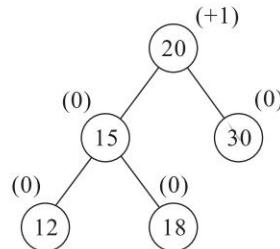
Balanced tree after deletion



(a) Deletion of a leaf node



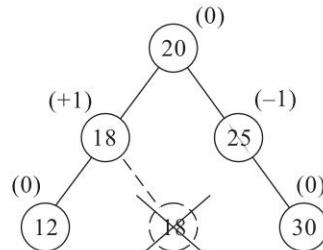
delete 25



(b) Deletion of a node with a single subtree



delete 15



(c) Deletion of a node with both left and right subtree

**Fig. 10.18** Rotation free deletion of nodes in an AVL search tree

available and the deletion took place to its right,  $bf(20)$  is increased by 1 which yields the new value as  $bf(20)=+1$ . We now see that the tree is automatically balanced and in this case no rotations were called for.

**Rotation free deletion of a node having a single subtree** In the case of deletion of node  $t$  with a single subtree, just as before, we reset the child link of the parent node, node  $p$ , to point to the child node of node  $t$ . The balance factors of node  $p$  and / or its ancestor nodes are updated using Rules 1–4 of Algorithm 10.3.

**Example 10.9** Deletion of key 25 illustrated in Fig. 10.18(b) is a case of deletion of a node with a single subtree. Observe how the RCHILD link of node 20 which is the root is reset to point to node 30, the right child of node 25. Since the deletion occurred to the right of node 20,  $bf(20)$  is updated to +1. Again the deletion does not call for any rotation to balance the tree.

**Rotation free deletion of a node having both the subtrees** In the case of deletion of node  $t$  which has both its subtrees to be non empty, the deletion is a little more involved (Algorithm 10.4). We first replace DATA (node  $t$ ) with the smallest key of the right subtree of node  $t$  or with the largest key of the left subtree of node  $t$ . Algorithm 10.4 illustrates replacement using the smallest key on the right subtree of node  $t$ . The smallest key of the right subtree of node  $t$  can be obtained by moving right and then moving deep left until the left child link is NIL. Similarly, moving left and then moving deep right until an empty right child link is seen will yield the largest element in the left subtree of node  $t$ . An important observation here is that the node representing the smallest value of the right subtree or the largest value of the left subtree will turn out to be either leaf nodes or nodes with a single subtree. Now we physically delete the node holding this replacement value using the procedure discussed for the deletion of a leaf node or a node with a single subtree (Algorithm 10.3).

**Example 10.10** Deletion of node 15 from the balanced AVL search tree shown in Fig. 10.18 yields the balanced tree shown in Fig. 10.18(c). Here the *value* 15 in the respective node is *replaced by* 18, the largest value in the right subtree of node 15. Now we physically delete the leaf node holding 18 using the case discussed earlier. Observe how  $bf(18)$  (the balance factor of the node which earlier had the value 15 but now holds the replaced value 18) is updated to +1 and applying Rule 3 of Algorithm 10.4, the balance factors of no other ancestor nodes are changed. The example once again illustrates a case of rotation free deletion.

As pointed out in Rule 4 of Algorithm 10.3, during the propagation of balance factor updates from the specific node to the root upwards, there could be cases of imbalance amongst the nodes. To set right the imbalance it is essential that the category of rotation viz.,  $L$  or  $R$  is identified before sub classifying it further as  $L0/R0$ ,  $L1/R1$  or  $L-1/R-1$ . All the rotations are performed with regard to the first ancestor node  $A$  encountered on the path to the root node and whose  $bf(A)=\pm 2$ . Each of the  $R$  category and  $L$  category of rotations are illustrated with examples.

## R category rotations associated with the delete operation

**$R0$  rotation** Figure 10.19 illustrates the generic representation of an  $R0$  rotation. Node  $t$  is to be deleted from a balanced tree with  $A$  shown as the root (for simplicity) and with  $A_R$  as its right subtree.  $B$  is the root of  $A$ 's left subtree and has two subtrees  $B_L$  and  $B_R$ . The heights of the

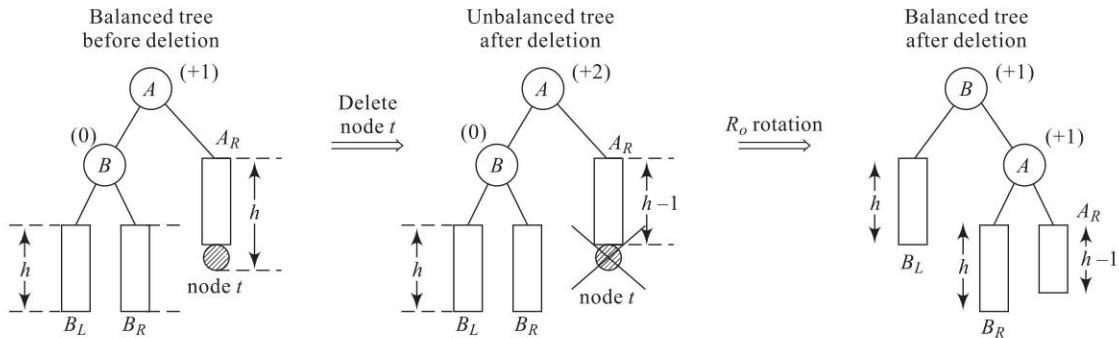


Fig. 10.19 Generic representation of an R0 rotation

subtrees are as shown in the figure. Now the deletion of node  $t$  results in an imbalance with  $bf(A)=+2$ . Since deletion of node  $t$  occurred to the right of  $A$  and since  $A$  is the first node on the path to the root, the situation calls for an  $R$  rotation with respect to  $A$ . Again since  $bf(B)=0$ , the rebalancing needs to be brought about using an  $R0$  rotation only. Here,  $B$  pushes itself up to occupy  $A$ 's position pushing node  $A$  to its right along with  $A_R$ . While  $B$  retains  $B_L$  as its left subtree,  $B_R$  is handed over to node  $A$  to function as its left subtree. Observe how the tree regains its balance.

**Example 10.11** Figure 10.20 illustrates the deletion of key 65 from the balanced tree shown. Since 65 occurred to the right of 55,  $bf(55)$  is updated to 0. This implies the balance factors of its ancestors nodes need to be updated. When we proceed to the node holding 50,  $bf(50)$  is updated to +2. This calls for an  $R0$  rotation with respect to node 50. The notations of the generic representation (Fig. 10.19) have been mapped to the given tree for ease of understanding. At the end of the rotation the tree is balanced with 30 as its root and the tree appropriately rearranged.

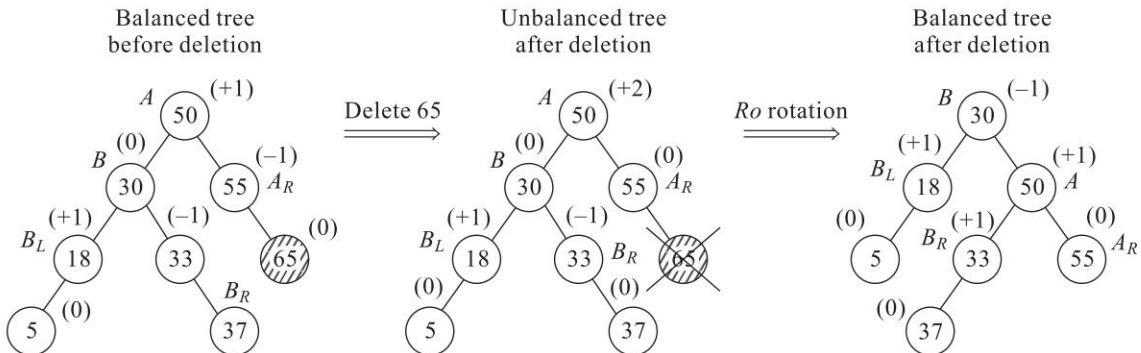


Fig. 10.20 Deletion of a node calling for R0 rotation

**R1 Rotation** Figure 10.21 illustrates the generic representation of an  $R1$  rotation. Deletion of node  $t$  occurs to the right of  $A$  the first ancestor node whose  $bf(A)=+2$ . But  $bf(B)=+1$  classifies it further as  $R1$  rotation. The rotation is similar to the  $R0$  rotation and yields a balanced tree.

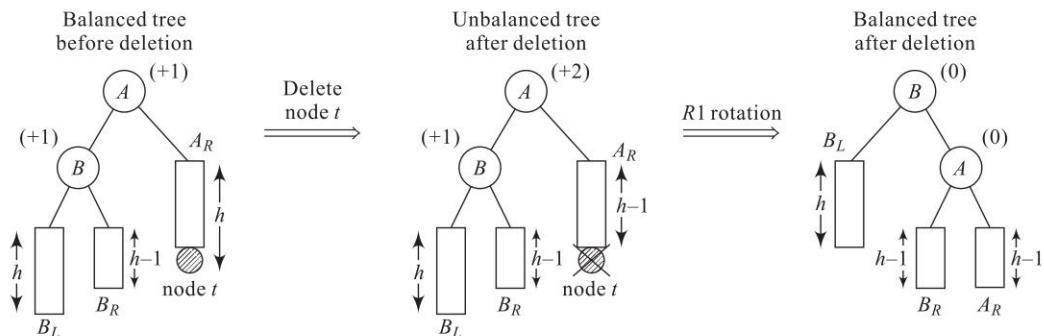


Fig. 10.21 Generic representation of an R1 rotation

**Example 10.12** Figure 10.22 shows the deletion of key 80 from the given balanced AVL search tree. The deletion occurs to the right of 76 and while updating the balance factors of the ancestor nodes yields  $bf(76)=+2$  which is the first and only ancestor node reporting imbalance. R1 rotation yields 60 as the root with the tree accordingly rearranged to balance it.

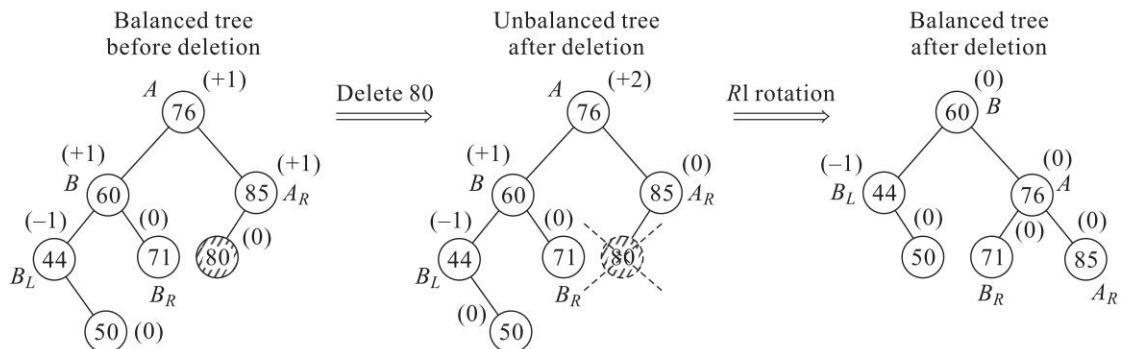


Fig. 10.22 Deletion of a node calling for R1 rotation

**R-1 rotation** The generic representation of an R-1 rotation is shown in Fig. 10.23. As in the other rotations deletion of node  $t$  results in the imbalance of the tree with regard to  $A$  and also

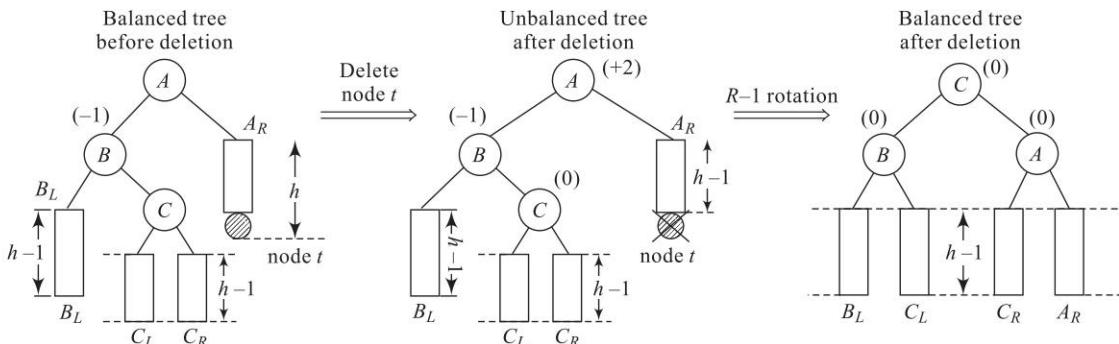


Fig. 10.23 Generic representation of an R-1 rotation

leaves  $bf(B) = -1$  calling for  $R-1$  rotation. Here let  $C$  be the root of the right subtree of  $B$  and  $C_L$  and  $C_R$  its left and right subtrees respectively. During the rotation  $C$  elevates itself to become the root pushing  $A$  along with its right subtree  $A_R$  to its right. The tree is now rearranged with  $C_L$  as the right subtree of  $B$  and  $C_R$  as the left subtree of  $A$ . The tree automatically gets balanced.  $R-1$  rotation is a case of double rotation where the rotation is once applied over  $B$  and then again over  $A$ .

**Example 10.13** Figure 10.24 illustrates the deletion of key 40 from the given balanced AVL search tree. Since 40 occurs to the left of the key 46,  $bf(46)$  is updated to 0 triggering an update of  $bf(35)$  which yields +2. Since  $bf(21) = -1$ , we resort to  $R-1$  rotation. The rest of the steps in the rotation follow those shown in the generic representation (Fig. 10.23).

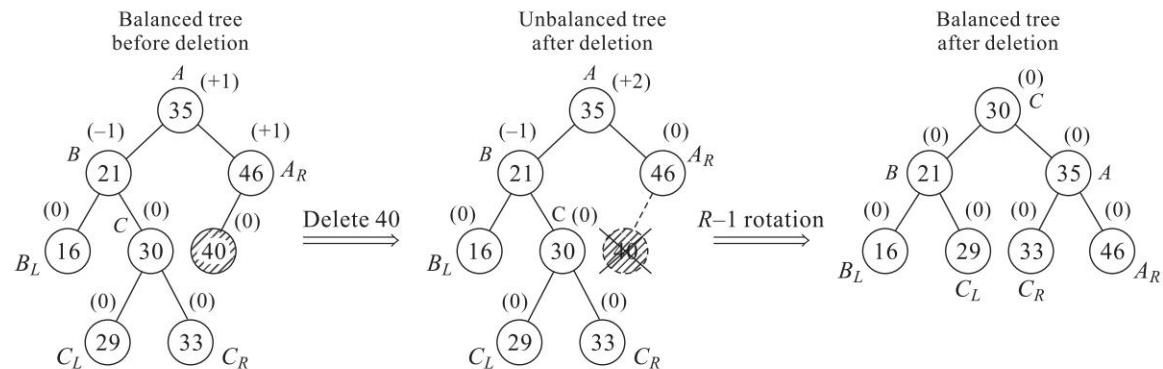


Fig. 10.24 Deletion of a node calling for  $R-1$  rotation

### L category rotations associated with the delete operation

If the deletion of node  $t$  occurs to the left of  $A$ , the first ancestor node on the path to the root reporting  $bf(A) = -2$ , then the category of rotation to be applied is  $L$ . As in  $R$  rotations, based on  $bf(B) = +1, -1$ , or 0 the  $L$  rotation is further classified as  $L0, L1$  and  $L-1$  respectively. The generic representations of the  $L0, L1$  and  $L-1$  rotations are shown in Fig. 10.25. An illustrative example for the  $L$  category of rotations is presented in Illustrative Problems I 10.9.

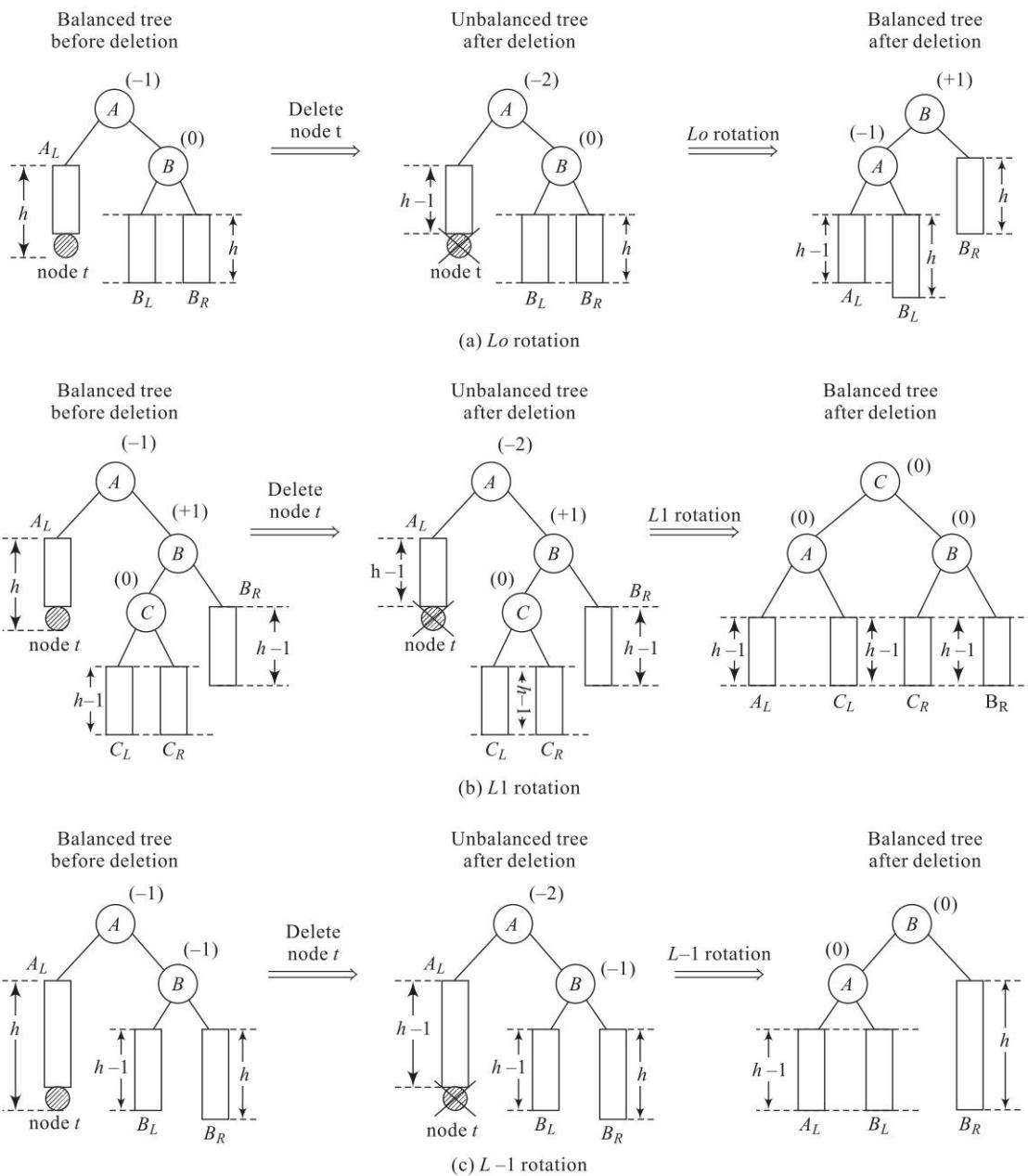
Unlike insertion, to rebalance a tree after a deletion it may be that more than one rotation is required. In fact the number of rotations required is  $O(\log n)$ . It can be observed that there are similarities between the  $LL, LR, RL$  and  $RR$  rotations undertaken during insertion and the  $L0/R0, L1/R1$  and  $L-1/R-1$  rotations undertaken during deletion.

## Applications

## 10.4

### Representation of symbol tables in compiler design

Compilers are translators that translate a source programming language code into a target programming language code viz., machine code or Assembly level language code. The various phases in the design of compilers include *Lexical analysis*, *Syntactic analysis*, *Semantic analysis*, *Intermediate code generation*, *Code optimization* and *Code generation*.



**Fig. 10.25** Generic representations of L0 L1 and L-1 rotations

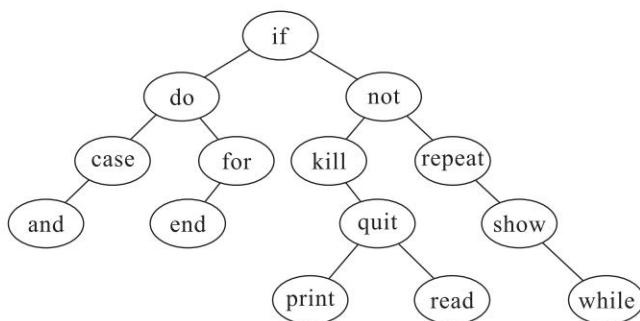
During lexical analysis, which is the first phase of the compiler, the source program is scanned character by character to identify the keywords, user identifiers, constants, labels etc which are termed as *tokens*. These tokens are stored in data structures called *symbol tables* which store

information pertaining to the tokens as a *name-attribute* pair. Thus there are individual symbol tables for keywords, user identifiers, constants etc.

Symbol tables which are constructed for a fixed set of data already known in advance and calling for no insert or delete operations after construction, but are only susceptible to search or retrieval operations are known as *static tables*. On the other hand those symbol tables which support insertion and deletion operations besides search are known as *dynamic tables*. While a keyword table is an example of a static symbol table, a user identifier table is an example of a dynamic table.

With regard to the keywords which are fixed for a given source language, a compiler stores them using a static symbol table using an appropriate data structure which favors efficient retrievals. This is so, since the Lexical Analyzer distinguishes a keyword token  $k$ , from a user identifier token  $u$ , by undertaking a search of the tokens  $k$  and  $u$  on the keyword table. While the search for  $k$  in the keyword table would yield a successful search, the same for  $u$  would yield an unsuccessful search. Those appropriately selected tokens which yield an unsuccessful search in the keyword table are classified as user id tokens and stored separately in a user id table. It is here that one sees the application of binary search trees and we terminate any further discussion on compilers at this point.

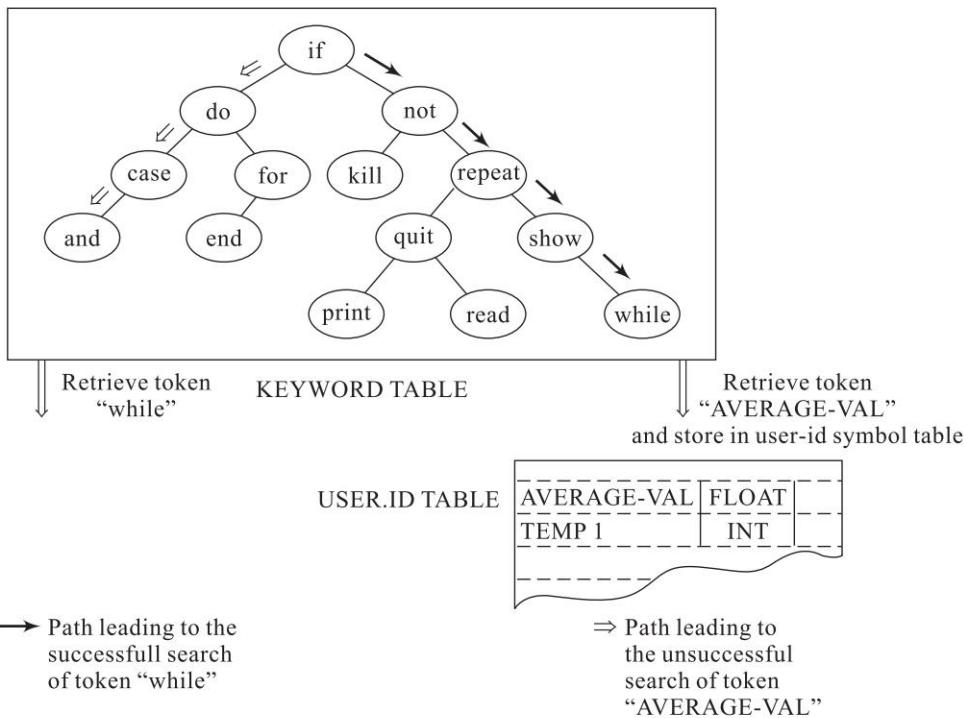
Since the keyword and user id tokens need searching for the presence or the absence of their respective tokens in the key word table, it is essential that the keyword table is represented using a data structure that supports efficient retrievals. A binary search tree is an excellent candidate for the representation of both static and dynamic symbol tables considering its  $O(\log n)$  average case complexity for insert, delete and retrieval operations. Figure 10.26 shows a sample keyword table represented using a binary search tree. Figure 10.27 illustrates a successful and an unsuccessful search of a token on the tree shown in Fig. 10.26. While the retrieval of token "while" yields success in 5 comparisons, the retrieval of "average\_val" which is a user id results in an unsuccessful search in 4 comparisons and hence find its place in the user id table shown.



**Fig. 10.26** Representation of a sample keyword table using binary search trees

The application of binary search trees for the representation of keyword symbol table in compiler design can be probed further to bring in the application of AVL search trees as well. It is known that for a given set  $K$  of keywords, a finite set of binary search trees may be constructed. However,

the problem now is to look for the most efficient representation amongst the binary search trees. Though a procedure to construct an *optimal binary search tree*, viz., a binary search tree which reports the minimum cost (see Illustrative Problem 10.4) exists, the optimal binary search tree may lead to inefficient retrievals comparatively due to the imbalance of nodes. It is in such a case that the application of AVL search trees becomes visible. Representing the keyword table as an AVL search tree ensures the retrieval of tokens in  $O(\log n)$  time in the worst case.



**Fig. 10.27** Searching for a keyword and a user id from a keyword symbol table represented using a binary search tree



## Summary

- Binary search trees may be empty or if otherwise, are labeled binary trees where the left child key is less than its parent node key and right child key is greater than the parent node key. All the keys forming a binary search tree are distinct. Binary search trees are represented using linked representations. However in many cases it is convenient to represent them as extended binary trees.
- A search or retrieval operation on a binary search tree is of  $O(\log n)$  complexity and hence is more efficient than the same over a sequential list. The insert and delete operations are also of  $O(\log n)$  complexity.

- The insertion of a key in a binary search tree is similar to that of searching for the key (unsuccessfully) and inserting it at the appropriate position as a leaf node. The deletion of a binary search tree is discussed depending on whether the deleted node is a leaf node or a node with a single subtree or a node with two subtrees.
- Binary search trees suffer from the drawback of becoming skewed especially when the keys are inserted in their sorted order.
- AVL trees are height balanced trees with the balance factor of the nodes being either 0 or 1 or -1. AVL search trees are height balanced binary search trees
- The search or retrieval operation on an AVL search tree is similar to that on a binary search tree.
- The insertion operation on an AVL search tree is similar to that of a binary search tree but when it leads to imbalance of the tree, any one of the rotations viz., LL, LR, RL and RR are undertaken to rebalance the tree.
- The deletion operation on an AVL search tree is classified as that of a leaf node, or a node with a single subtree or a node with two subtrees. In the event of any imbalance in the tree, the R category of rotations viz., R0, R1, R-1 or the L category of rotations, viz., L0, L1 and L-1 or a combination are called for to rebalance the tree.
- Binary search trees and AVL search trees find application in the representation of symbol tables in compiler design.



## Illustrative Problems

### Problem 10.1

- (a) Construct a binary search tree  $T$  for the following set  $S$  of elements in the order given:
- $$S = \{ \text{INDIGO, GREEN, CYAN, YELLOW, RED, ORANGE, VIOLET} \}$$
- (b) How many comparisons are made for the retrieval of "yellow" from the tree corresponding to the one drawn in 10.1(a)?
- (c) For what arrangements of elements of  $S$  will the associated binary search tree turn out to be skewed?
- (d) For the binary search tree(s) constructed in I 10.1(c) how many comparisons are made for the retrieval of "yellow"?

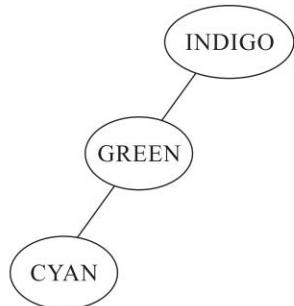
**Solution:** (a) The binary search tree constructed for  $S$  with the elements considered in the order given, is shown in Fig. I 10.1(a).

- (b) The number of comparisons for the retrieval of "yellow" is 2. The search key "yellow" is first compared against the root which is "indigo" and since "yellow" is greater than "indigo" moves to the right of "indigo" only to find itself after the second comparison.
- (c) If the elements of  $S$  are sorted either in the ascending or in the descending order then the associated binary search tree will be either right skewed or left skewed respectively. Figure I 10.1(b) illustrates the trees.
- (d) In the case of the left skewed tree the number of comparisons for "yellow" is 1 and in the case of the right skewed tree the number of comparisons is 7.

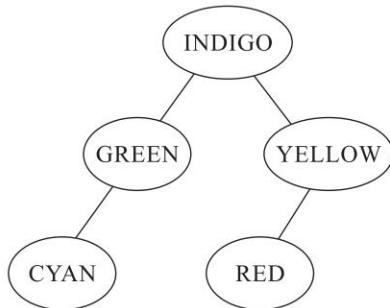
Insert: INDIGO



Insert: GREEN, CYAN



Insert: YELLOW, RED



Insert: ORANGE, VIOLET

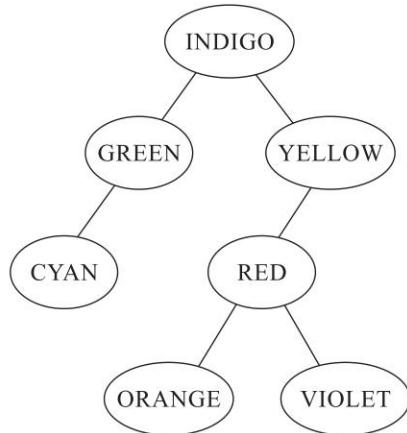
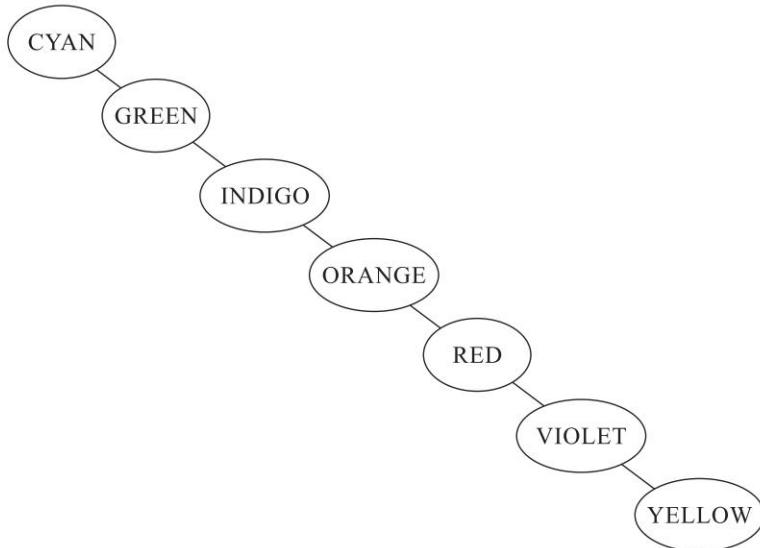


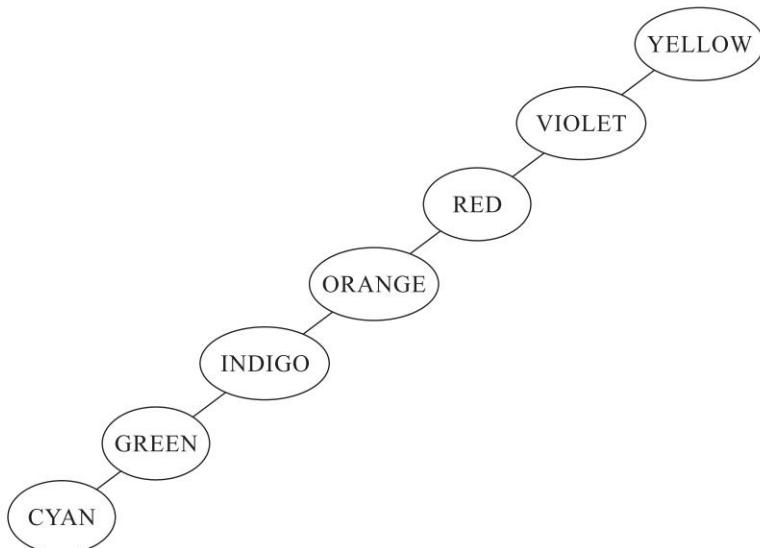
Fig. I 10.1(a)

Ascending order of S: {CYAN, GREEN, INDIGO, ORANGE, RED, VIOLET, YELLOW}

Corresponding binary search tree :

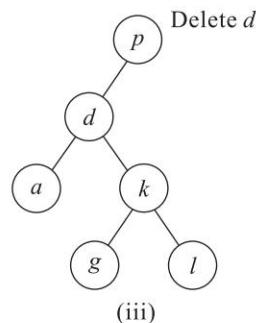
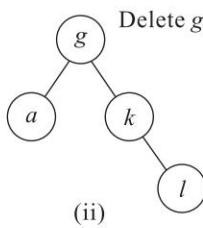
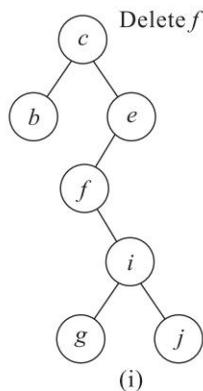


Descending order of S: {YELLOW, VIOLET, RED, ORANGE, INDIGO, GREEN, CYAN}

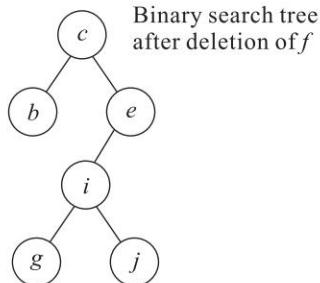


**Fig. I 10.1(b)**

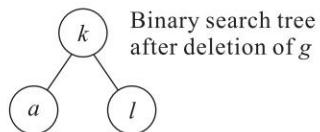
**Problem 10.2** Given the following binary search trees draw the same after the deletion of the specified elements in the respective binary search trees.



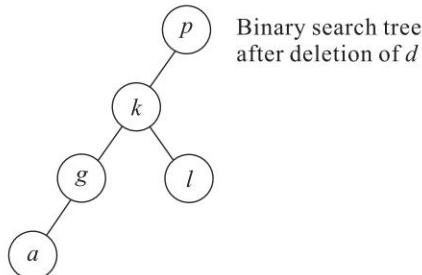
**Solution:** Deletion of *f* is a case of deletion of a node with a single non empty subtree with root *i*. Hence simply delete *f* and link the node *e* with node *i*.



Deletion of *g* is a case of deletion of a node with two non empty subtrees. Delete *g* and push *k* along with its subtree to be the root. *a* joins as the left child of *k*.



Deletion of *d* is again a case of deletion of a node with two non empty subtrees. After the deletion of *d*, *k* along with its subtrees moves up. *a* joins *g* as its left child.

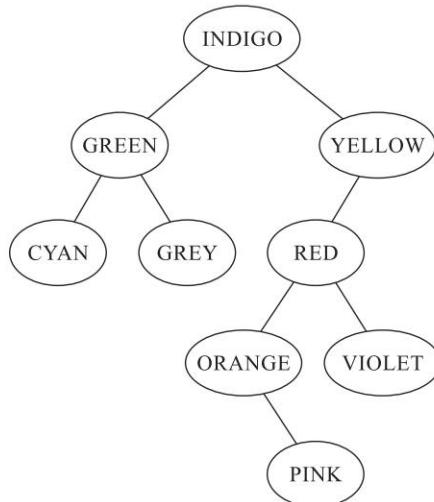


**Problem 10.3** On the binary search tree shown in Fig. I 10.1(a), perform the following operations in the order shown:

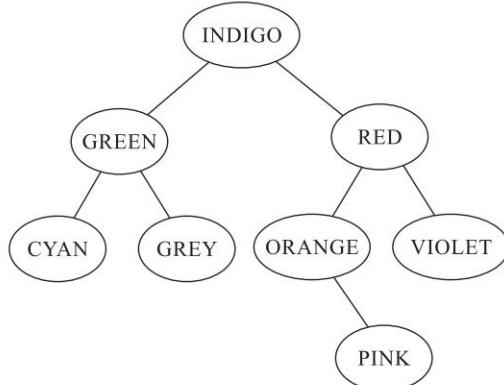
Insert GREY, Insert PINK, Delete YELLOW, Delete RED

**Solution:** The trees after the two insert operations and two delete operations in the given sequence are shown below:

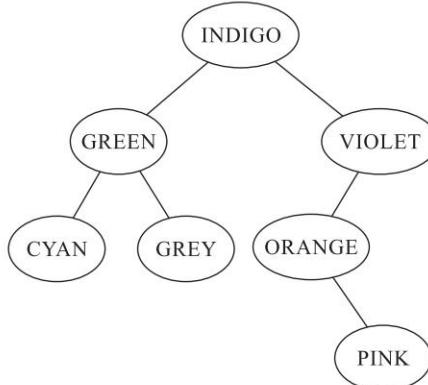
Insert GREY  
Insert PINK



Delete YELLOW



Delete RED



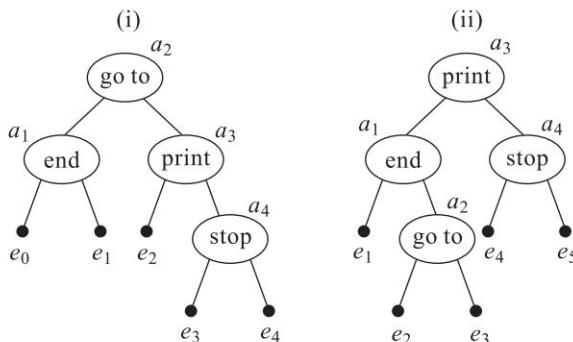
**Problem 10.4** [optimal binary search trees] Let  $S = \{a_1, a_2, a_3, \dots, a_n\}$  be a set of elements and  $T_k$  be the set of associated binary search trees that can be constructed out of  $S$ . Let  $p_i$ ,  $1 \leq i \leq n$ , be the probability with which  $a_i$  is searched for (probability of successful search) and let  $q_j$ ,  $0 \leq j \leq n$ , be the probability with which a key  $X$ ,  $a_j \leq X \leq a_{j+1}$  is unsuccessfully searched for (probability of unsuccessful search) on a binary search tree  $T_k$ . As explained in Sec. 10.2, the search for such an  $X$  will end up in an appropriate external node  $e_j$ . The cost of a binary search tree is given by

$$\sum_{1 \leq i \leq n} p_i \cdot \text{level}(a_i) + \sum_{0 \leq j \leq n} q_j \cdot (\text{level}(e_j) - 1)$$

An optimal binary search tree is a tree  $T \in T_k$  such that  $\text{cost}(T)$  is the minimum. The term  $\sum_{0 \leq j \leq n} q_j \cdot (\text{level}(e_j) - 1)$  when  $q_j, 0 \leq j \leq n$  represents weights associated with the external nodes is known as *weighted external path length*.

Consider a set  $S = \{\text{end}, \text{goto}, \text{print}, \text{stop}\}$ , let  $\{p_1, p_2, p_3, p_4\} = \left\{\frac{1}{20}, \frac{1}{5}, \frac{1}{10}, \frac{1}{20}\right\}$  and

$\{q_0, q_1, q_2, q_3, q_4\} = \left\{\frac{1}{5}, \frac{1}{10}, \frac{1}{5}, \frac{1}{20}, \frac{1}{20}\right\}$ . Find the cost of the following binary search trees and show of the two which one has the minimum cost?



**Solution:** The cost of tree (i) is given by

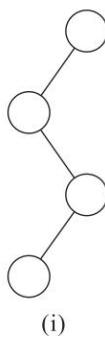
$$\left\{\frac{1}{20} \cdot 2 + \frac{1}{5} \cdot 1 + \frac{1}{10} \cdot 2 + \frac{1}{20} \cdot 3\right\} + \left\{\frac{1}{5} \cdot 2 + \frac{1}{10} \cdot 2 + \frac{1}{5} \cdot 2 + \frac{1}{20} \cdot 3 + \frac{1}{20} \cdot 3\right\} = \frac{13}{20} + \frac{26}{20} = \frac{39}{20}$$

The cost of tree(ii) is given by

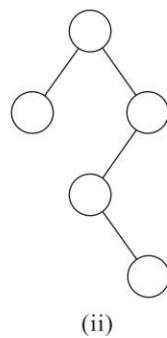
$$\left\{\frac{1}{20} \cdot 2 + \frac{1}{5} \cdot 3 + \frac{1}{10} \cdot 1 + \frac{1}{20} \cdot 2\right\} + \left\{\frac{1}{5} \cdot 2 + \frac{1}{10} \cdot 3 + \frac{1}{5} \cdot 3 + \frac{1}{20} \cdot 2 + \frac{1}{20} \cdot 2\right\} = \frac{18}{20} + \frac{30}{20} = \frac{48}{20}$$

Hence of the two binary search trees, tree(i) is a minimum cost binary search tree.

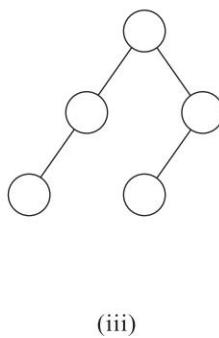
**Problem 10.5** Which of the following is an AVL tree?



(i)



(ii)



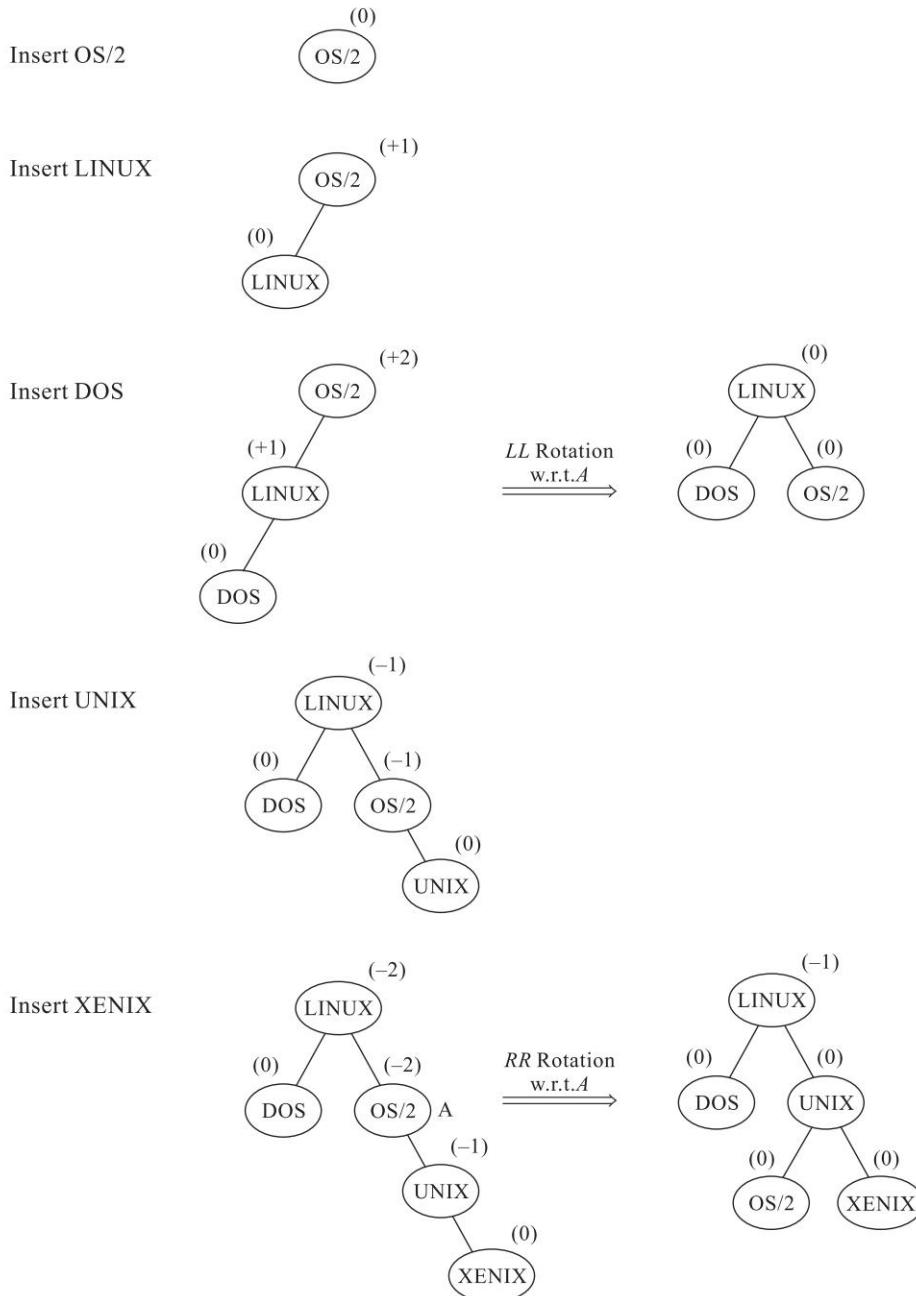
(iii)

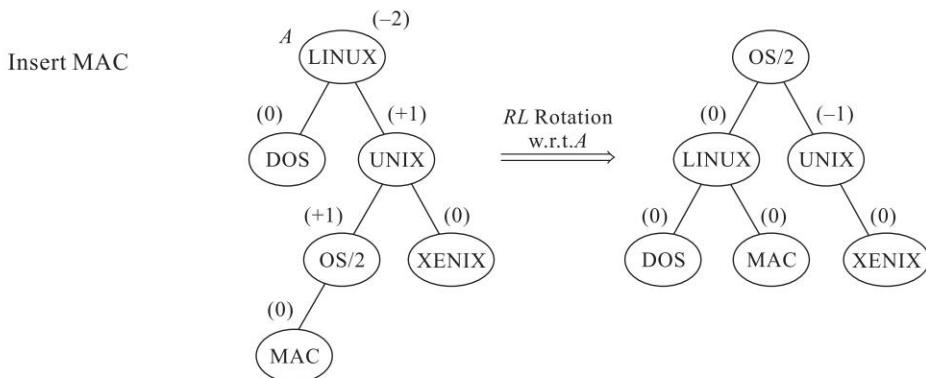
**Solution:** Tree (iii)

**Problem 10.6** Construct an AVL search tree using the following data. Perform the appropriate rotations to rebalance the tree.

OS/2, LINUX, DOS, UNIX, XENIX, MAC

**Solution:** The construction of the AVL search tree is as shown below:

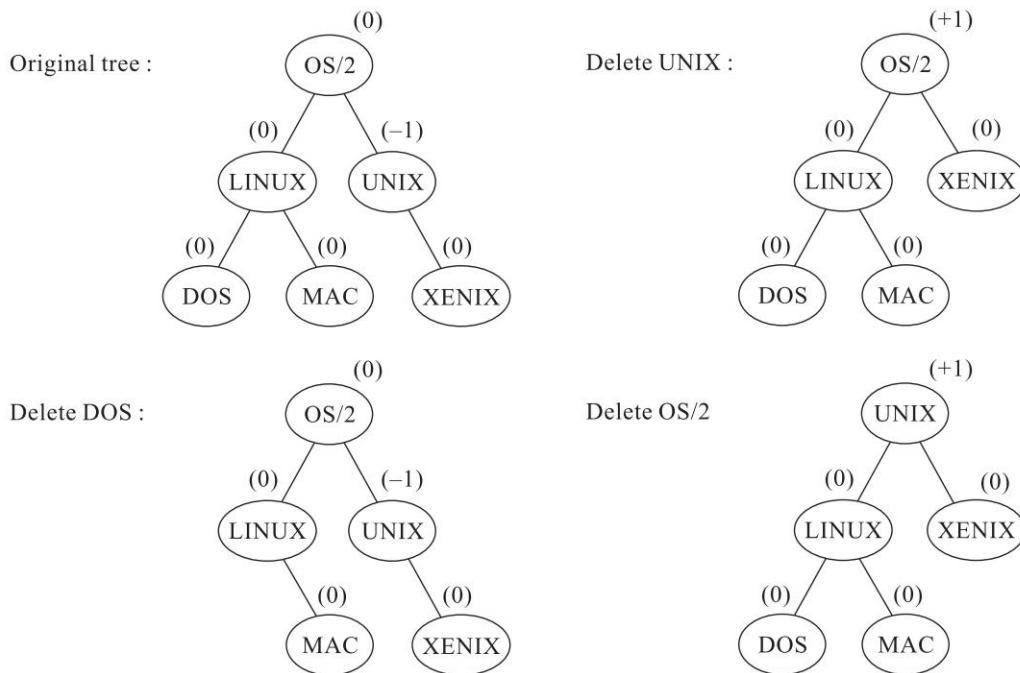




**Problem 10.7** For the AVL search tree constructed in Illustrative Problem 10.6, perform the following operations using the original tree for each operation:

Delete DOS, Delete UNIX, Delete OS/2

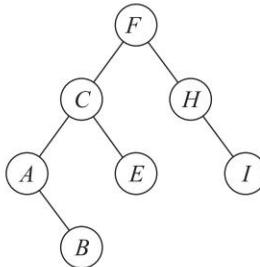
**Solution:** Each of the delete operations as performed on the original tree is illustrated below:



Observe that each of the three cases are examples of deletion of a leaf node, a node with a single subtree and a node with both subtrees respectively. Also the deletion operations are rotation free.

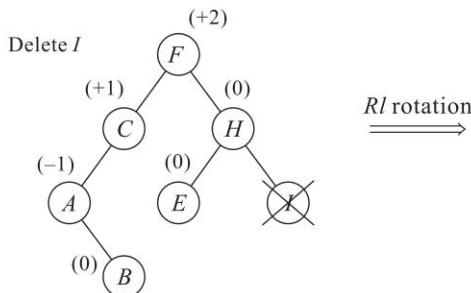
**Problem 10.8** For the following AVL search tree undertake the following deletions in the sequence shown:

Delete I, Delete B, Delete H

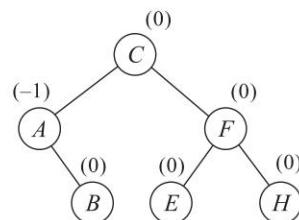


**Solution:** The balanced tree after the execution of the three operations in a sequence is shown below:

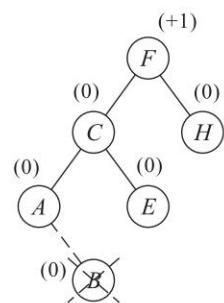
Delete I



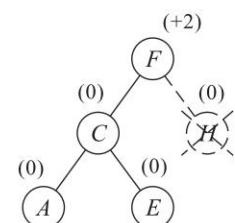
Rl rotation



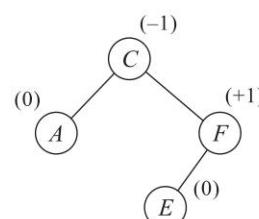
Delete B



Delete H

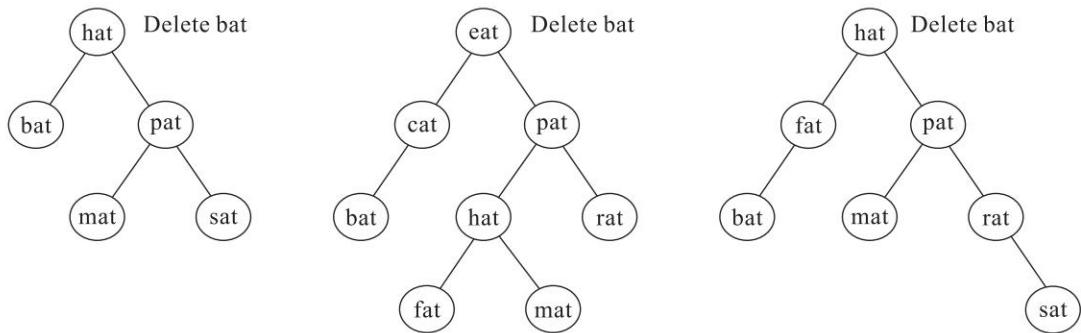


Ro rotation

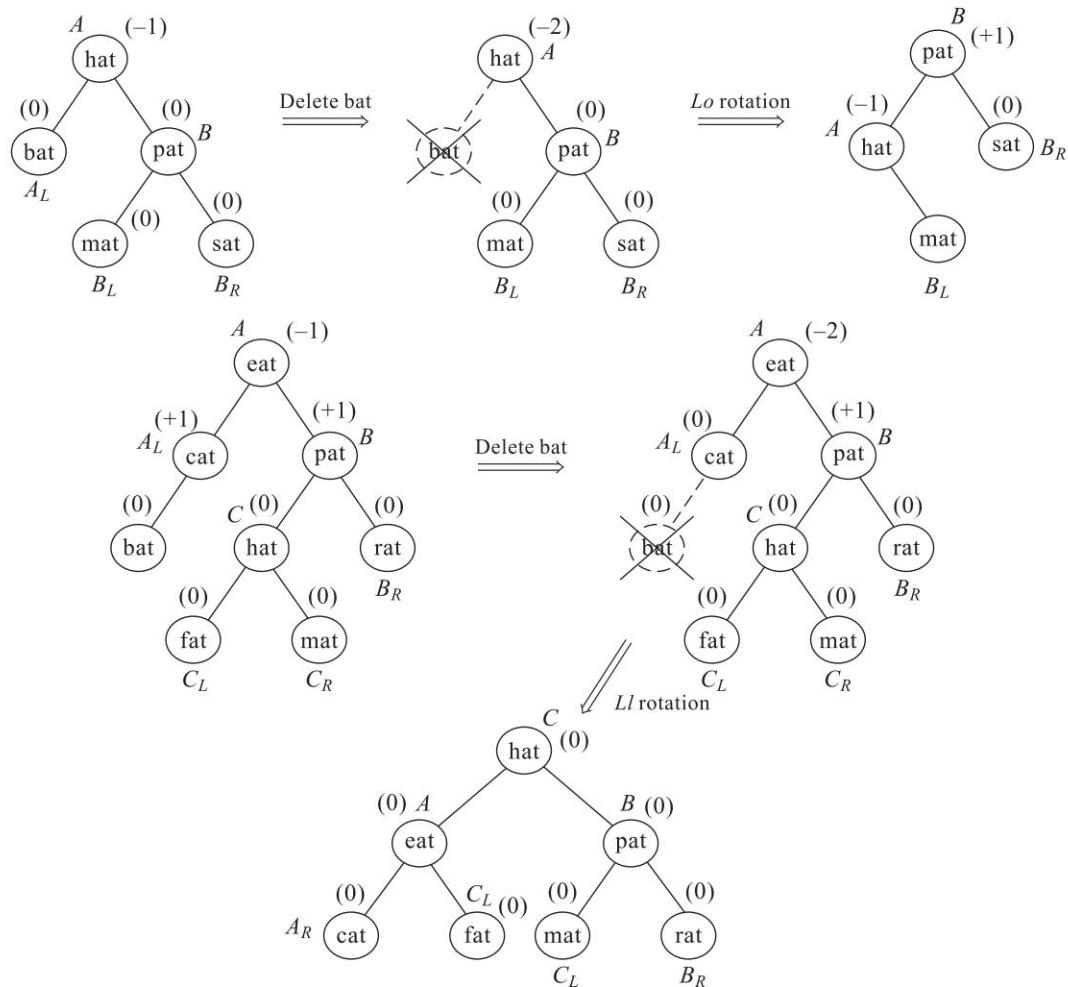


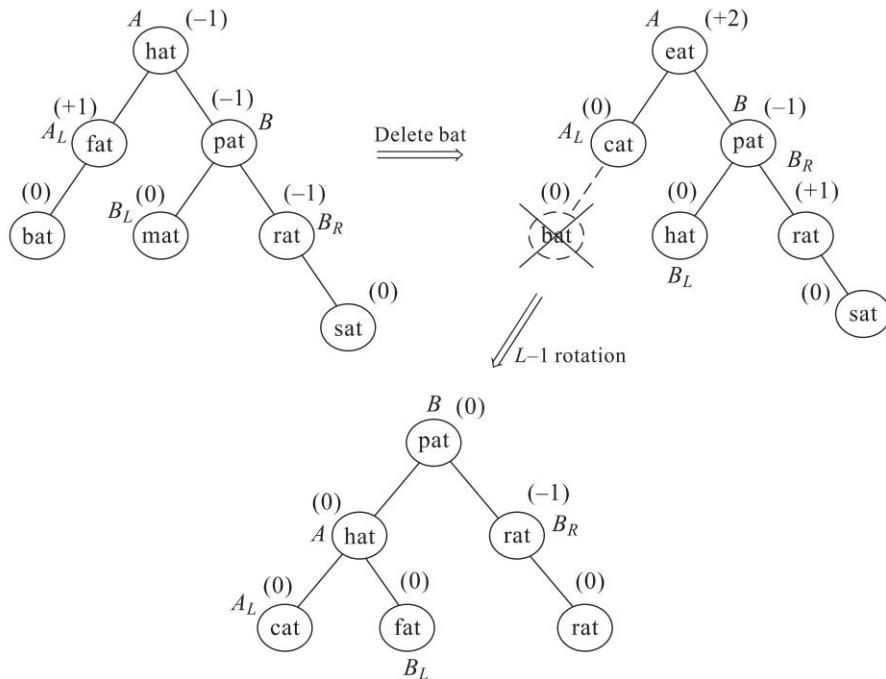
Here, while Delete I and Delete H called for R1 and R0 rotations respectively, Delete B did not call for any rotations.

**Problem 10.9** Perform the following deletions on the given AVL search trees:

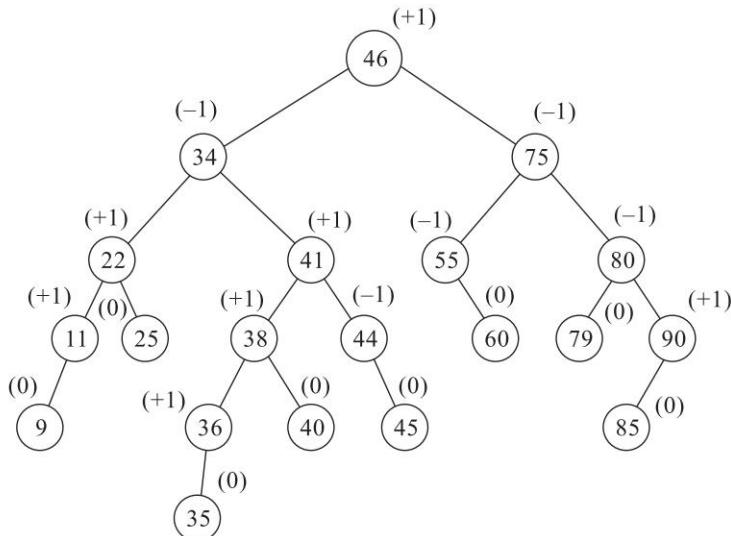


**Solution:** This problem is an illustration of L category rotations. Hence the notations  $A, B, C, A_L, A_R, B_R, C_L, C_R$  and employed in the generic representations of the L rotations (Sec. 10.3) have been made use of in the tree for ease of understanding.

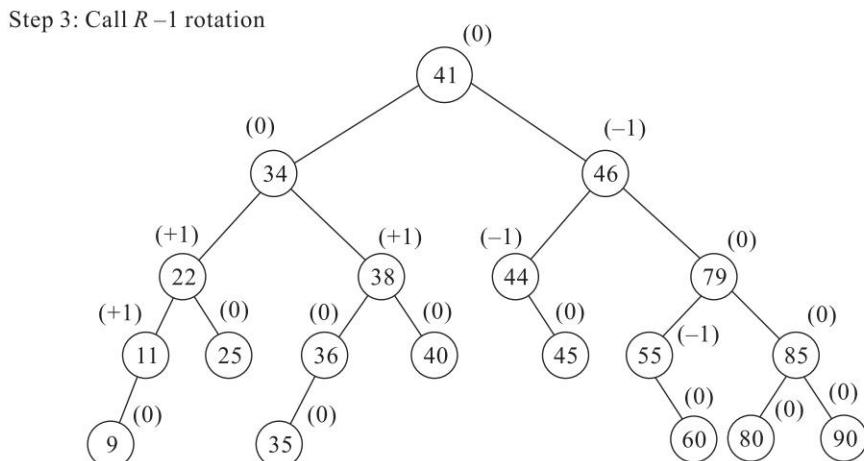
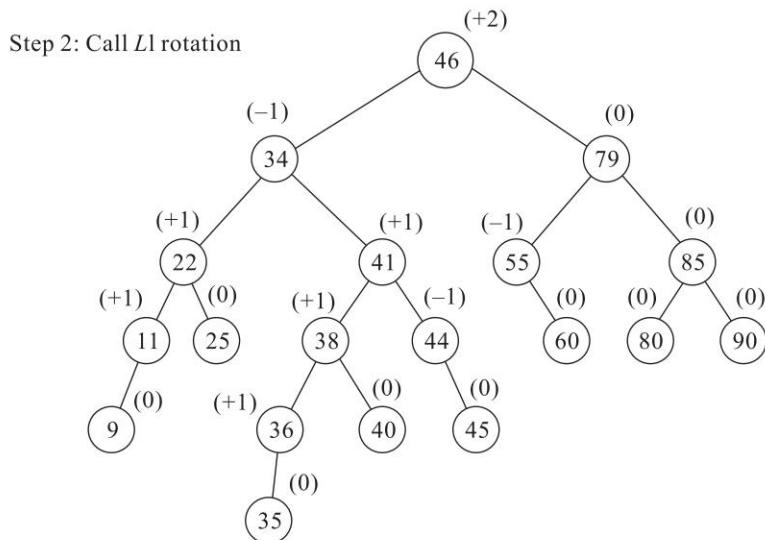
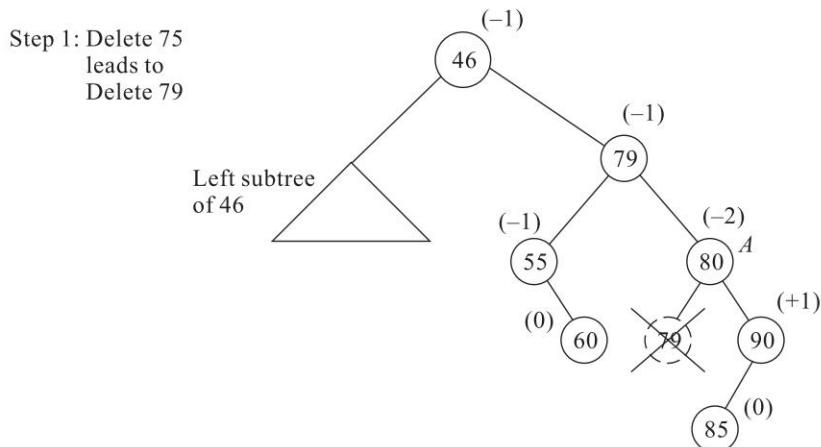




**Problem 10.10** Delete 75 from the following AVL search tree:



**Solution:** The deletion of 75 leads to deletion of 79 and calls for two rotations viz., L1 and R-1 to set right the imbalance. The various steps during the deletion are shown below:





## Review Questions

1. Which among the following norms is not satisfied by a binary search tree  $T$ ?
  - (i) all keys of the binary search tree need not be distinct
  - (ii) all keys in the left subtree of  $T$  are less than the root element
  - (iii) all keys in the right subtree of  $T$  are greater than the root element and
  - (iv) the left and right subtrees of  $T$  are also binary search trees.

(a) (i)                      (b) (ii)                      (c) (iii)                      (d) (iv)
2. State whether true or false:  
In the case of a binary search tree with  $n$  nodes,
  - (i) the number of external nodes is  $(n - 1)$
  - (ii) an inorder traversal yields the keys of the nodes in the ascending order
  - (a) (i) true (ii) true (b) (i) true (ii) false (c) (i) false (ii) true (d) (i) false (ii) false
3. Which among the following deteriorates the performance of a binary search tree  $T$  with  $n$  nodes?
  - (i) when there are large number of deletions to  $T$
  - (ii) when the number of external nodes becomes  $(n+1)$
  - (iii) when the height of  $T$  becomes  $n$
  - (iv) when the height of  $T$  becomes  $\log_2 n$

(a) (i)                      (b) (ii)                      (c) (iii)                      (d) (iv)
4. The balance factor of any node in an AVL tree cannot be
  - (a) 0                              (b) +1                              (c) -1                              (d) +3
5. In an AVL tree, during the deletion of node  $t$  which is a leaf node and whose parent node is node  $p$ , which among the following does not happen during the sequence of operations?
  - (i) physically delete node  $t$  and make the child link of its parent, viz., node  $p$  null.
  - (ii) make the child link of node  $p$  point to the child node of node  $t$  and then physically delete node  $t$
  - (iii) update the balance factor of node  $p$  based on whether the deletion occurred to its right or left
  - (iv) update the balance factors of the ancestor nodes of node  $p$

(a) (i)                              (b) (ii)                              (c) (iii)                              (d) (iv)
6. If a key  $u$  to be deleted from a binary search tree has only a left subtree and if it were the right child of its parent node, then
  - (i) set both the links of its parent node to NIL
  - (ii) allow the right link of its parent node alone to be NIL
  - (iii) allow the left link of its parent node to point to the left subtree of key  $u$
  - (iv) allow the right link of its parent node to point to the left subtree of key  $u$

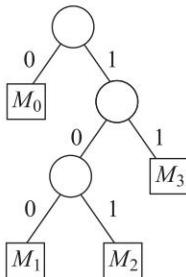
(a) (i)                              (b) (ii)                              (c) (iii)                              (d) (iv)
7. How are binary search tree representation of lists advantageous over their sequential list representations?
8. How is the deletion of a node that has both left and right subtrees, undertaken in a binary search tree?
9. What is the need for an AVL tree?
10. How is the rotation free deletion of a node having both the subtrees, done in an AVL search tree?

11. Outline the generic representation of an *LL* type imbalance in an AVL search tree and the corresponding rotation.
12. For the following list of data construct a binary search tree:  
LINUX, OS2, DOS, XENIX, SOLARIS, WINDOWS, VISTA, XP, UNIX, CPM,  
Undertake the following operations on the binary search tree:  
(i) Insert MAC    (ii) Delete WINDOWS    (iii) Delete UNIX
13. Represent the data list shown in Review Questions 10.12 as a sequential list. Tabulate the number of comparisons undertaken for retrieving the following keys:  
(i) LINUX    (ii) XENIX    (iii) DOS    (iv) UNIX    (v) CPM
14. For the data list {AND, BEGIN, CASE, DO, END, FOR, GOTO, IF, IN, LET, NOT, OR, QUIT, READ, REPEAT, RESET, THEN, UNTIL, WHILE, XOR}  
(i) Construct a binary search tree. What are your observations?  
(ii) Construct an AVL search tree.
15. For the AVL search tree constructed in Review Questions 10.14 delete the following keys in the order given:  
XOR, READ, END, AND



## Programming Assignments

1. Implement a menu-driven program to perform the following operations on a binary search tree:  
(i) Construct a binary search tree /\* construction begins from an empty tree\*/  
(ii) Insert element(s), into a non empty binary search tree  
(iii) Delete element(s) from a non empty binary search tree  
(iv) Search for an element in a binary search tree
2. Write a function to retrieve the elements of a binary search tree in the sorted order.
3. Execute a programming project to illustrate the construction of an AVL search tree. Demonstrate the construction, insertion and deletion operations using graphics and animation
4. [Huffman Coding] D. Huffman applied binary trees with minimal external path length to obtain an optimal set of codes for messages  $M_j$ ,  $0 \leq j \leq n$ . Each message is encoded using a binary string for transmission. At the receiving end, the messages are decoded using a binary tree in which external nodes represent messages and the external path to the node represents the binary string encoding of the message. Thus if 0 labels a left branch and 1 a right branch, then a sample decode tree given below can decode messages  $M_0, M_1, M_2, M_3$  represented using codes {0, 100, 101, 11}. These codes are called Huffman codes. The expected decode time is given by  $\sum_{0 \leq i \leq n} q_i \cdot d_i$  where  $d_i$  is the distance of the external node representing message  $M_i$  from the root node and  $q_i$  is the relative frequency with which the message  $M_i$  is transmitted. It is obvious that the cost of decoding a message is dependent on the number of bits in the binary code of the message and hence on the external path to the node representing the message. Therefore the problem is to look for a decode tree with minimal weighted external path length to speed up decoding.



- (i) Investigate the algorithm given by D. Huffman to obtain a decode tree with minimal weighted external path length.
- (ii) Implement the algorithm given a sample set of weights  $q_i$ .
5. Implement an algorithm which given an AVL search tree  $T$  and a data item  $X$  will split the AVL search tree into two AVL search trees  $T_1, T_2$  such that all the keys in tree  $T_1$  are less than or equal to  $X$  and all the keys in tree  $T_2$  are greater than  $X$  respectively.



# B TREES AND TRIES

# 11

## Introduction

## 11.1

In this chapter, we discuss data structures pertaining to *multi-way trees*. Multi-way trees are tree data structures with more than two branches at a node. The data structures of *m-way search trees*, B trees and Tries belong to this category of tree structures.

AVL search trees which are height-balanced versions of binary search trees no doubt promote efficient retrievals and storage operations. The complexity of insert, delete and search operations on AVL search trees is  $O(\log n)$ . However, while considering applications such as File indexing where the entries in an index may be very large, maintaining the index as *m-way search trees* provides a better option than AVL search trees which are but only balanced binary search trees. While binary search trees are two-way search trees, *m-way search trees* which are extended binary search trees are multi-way search trees and hence favor more efficient retrievals.

B trees are height balanced versions of *m-way search trees* and hence command their own merits. In the case of all the search trees which deal with key based storage and retrievals, it is essential that the keys are of fixed size for efficient storage management. In other words, these data structures do not recommend representation of keys with varying sizes. Tries are tree based data structures that support keys with varying sizes. Also while other search trees indulge in searching based on the whole key, tries are based on searching only a portion of the key before the whole key is retrieved or stored.

The definition and operations of the three data structures viz., *m-way search trees*, B trees and Tries are detailed. Finally, the application of the data structures to file indexing and spell checking are discussed.

## *m-way search trees: Definition and Operations*

## 11.2

*m-way search trees* are extensions of binary search trees. Binary search trees indulge in at most two way branching at every node with the left subtree of the node representing elements less than the key value of the node and the right subtree representing elements which are greater than the key value of the node. On the other hand, each node of an *m-way search tree* can hold at most  $m$  branches. Thus, *m-way search trees* adopt multi way branching extending the above mentioned characteristic of binary search trees.

11.1 *Introduction*

11.2 *m-way search trees: Definition and Operations*

11.3 *B Trees: Definition and Operations*

11.4 *Tries: Definition and Operations*

11.5 *Applications*

## Definition

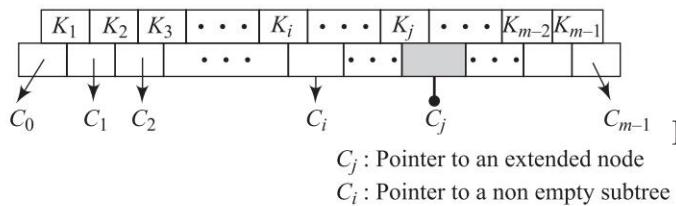
An  $m$ -way search tree  $T$  may be an empty tree. If  $T$  is non-empty then the following properties must hold good:

- (i) For some integer  $m$ , known as the order of the tree, each node has at most  $m$  child nodes. In other words, each node in  $T$  is of degree at most  $m$ . Thus a node of degree  $m$  will be represented by  $C_0, (K_1, C_1), (K_2, C_2), (K_3, C_3) \dots (K_{m-1}, C_{m-1})$  where  $K_i, 1 \leq i \leq m-1$  are the keys and  $C_j, 0 \leq j \leq m-1$  are pointers to the root nodes of the  $m$  subtrees of the node.
- (ii) If a node has  $k$  child nodes,  $k \leq m$ , then the node has exactly  $(k-1)$  keys  $K_1, K_2, K_3, \dots K_{k-1}$  where  $K_i < K_{i+1}$  and each of the keys  $K_i$  partitions the keys in the subtrees into  $k$  subsets.
- (iii) For a node  $C_0, (K_1, C_1), (K_2, C_2), (K_3, C_3) \dots (K_{m-1}, C_{m-1})$  all key values in the subtree pointed to by  $C_i$  are less than the key  $K_{i+1}, 0 \leq i \leq m-2$  and the key values in the subtree pointed to by  $C_{m-1}$  are greater than  $K_{m-1}$ .
- (iv) The subtrees pointed to by  $C_i, 0 \leq i \leq m-1$  are also  $m$ -way search trees.

## Node structure and representation

A  $m$ -way search tree is conceived to be an extended tree with its null pointers represented by external nodes. Although this method of representation is useful for its definition and discussion of its operations, the external nodes as in the case of any general tree, are only fictitious and not physically represented.

Figure 11.1 illustrates a general structure of a node in an  $m$ -way search tree. The node has  $(m-1)$  key elements and hence exactly  $m$  child pointers to the root nodes of the  $m$  subtrees. Those pointers to subtrees which are empty are indicated by external nodes represented as circles.



**Fig. 11.1 Structure of a node in an  $m$ -way search tree**

**Example 11.1** An example 4-way search tree is shown in Fig. 11.2. Observe how each node has at most 4 child nodes some of which are external nodes. Also every node has exactly  $(p-1)$  keys if the number of child nodes it has is  $p$ . The root node for example has four child pointers and hence four subtrees. The number of keys in the root node is therefore three viz., [34, 56, 84]. The first subtree of the root node contains keys which are less than 34, the second subtree which is to contain keys greater than 34 and less than 56 is empty, the third subtree contains keys greater than 56 and less than 84 and finally the last subtree contains keys which are greater than 84. A similar concept is extended to every other node in the subtrees.

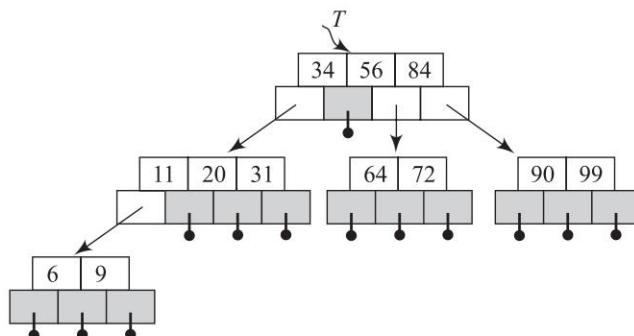


Fig. 11.2 An example 4-way search tree

### Searching an $m$ -way search tree

Searching for a key  $K$  in an  $m$ -way search tree  $T$ , is an extension of the method by which the key would have been searched for in a binary search tree.  $K$  is first sequentially searched against the key elements of the root node  $[K_1, K_{i+1}, \dots, K_t]$ . If  $K = K_j$  then the search is done. If  $K > K_j$  and  $K < K_{j+1}$  for some  $j$ , then the search moves down to the root node of the corresponding subtree  $T_j$ . The search progresses in a similar fashion until the key is obtained in which case the search is termed successful otherwise unsuccessful.

Consider the sample 4-way search tree shown in Fig. 11.2. Searching for key 6 calls for moving down the first subtree of the root  $[34, 56, 84]$ , since  $6 < 34$ . The search further moves down the first subtree of the node  $[11, 20, 31]$  since  $6 < 11$ . Now the node  $[6, 9]$  is reached. A mere sequential search of the key in the list of elements reports the presence of 6. The path traced by the search for the successful search of 6 is shown in Fig. 11.3. Let us now search for key 66 in the tree shown in Fig. 11.2. Since  $66 > 56$  and  $66 < 84$  the search moves down the third subtree of the root node. Here the node  $[64, 72]$  is encountered and it is sequentially searched for 66. Since  $66 > 64$  and  $66 < 72$ , the search moves down the second subtree of the node  $[64, 72]$  which is an external node. Here the search terminates and the search is termed unsuccessful since the element 66 is not found in the tree. The path traced by the search is shown in Fig. 11.3.

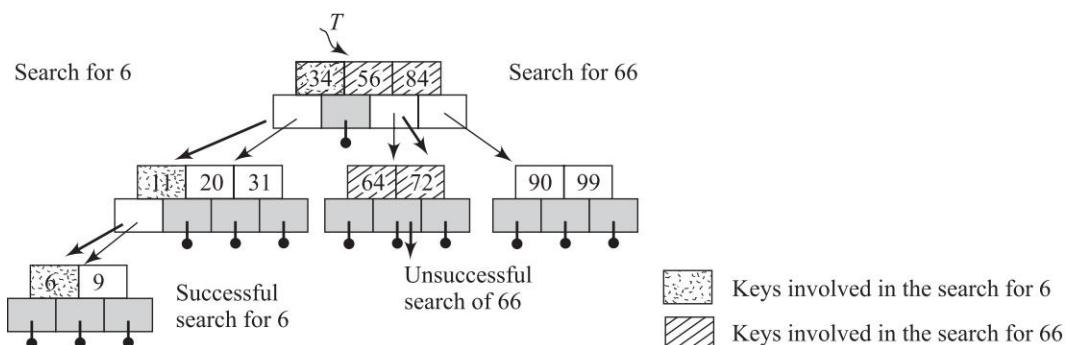
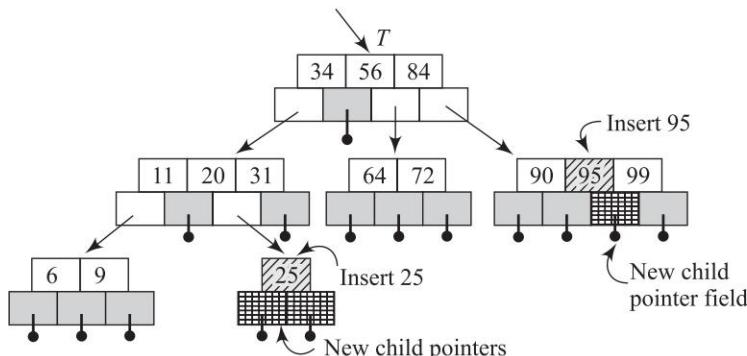


Fig. 11.3 Search for keys 6 and 66 from the 4-way search tree shown in Fig. 11.2

## Inserting into an $m$ -way search tree

The insertion of a key into a  $m$ -way search tree proceeds as one would search for the key. The search is bound to fall off at some node in the tree. At that position, the key may be either inserted as an element, if the node can accommodate the key or may be inserted as a new node in the next level.

Consider the insertion of key 95 in Fig. 11.2. Searching for 95 results in falling off at the node [90, 99]. Since the tree is a 4-way search tree every node in the tree can accommodate at most 3 keys. Therefore we merely insert 95 in the node [90, 99] to obtain [90, 95, 99]. Accordingly observe the change in the pointer fields of the node. Let us now insert 25 into the same tree. In this case searching for 25 results in falling off at an external node belonging to [11, 20, 31]. Since the node is already full with three elements, we insert 25 as a new node in the subtree of [11, 20, 31]. Figure 11.4 illustrates the insertion of keys 95 and 25 in the given 4-way search tree.



**Fig. 11.4** Insertion of keys 95 and 25 into the 4-way search tree shown in Fig. 11.2

## Deleting from an $m$ -way search tree

The delete operation as always, is complicated when compared to its insert operation counterpart. To delete a key we proceed as usual to find the key in the tree. Now let us suppose the key  $K$  is found in a node with its left subtree pointer as  $C_i$  and its right subtree pointer as  $C_j$ . Based on the following cases (**Dm** in the cases indicates Deletion in an  $m$ -way search tree) the deletion of  $K$  is undertaken:

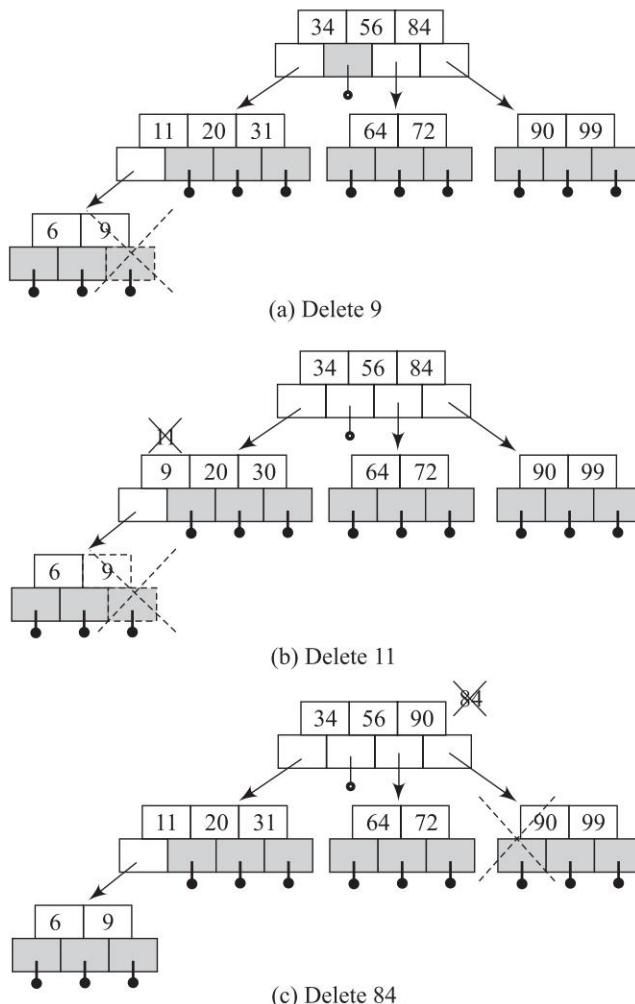
**Case Dm. 1**  $C_i = C_j = \text{NIL}$ . If the left and right subtrees of key  $K$  are NIL, then we simply delete the key  $K$  and adjust the number of pointer fields in the node.

**Case Dm. 2**  $C_i = \text{NIL}$  and  $C_j \neq \text{NIL}$ . If the left subtree of key  $K$  is empty and the right subtree is not, choose the smallest key  $K'$  from the right subtree of  $K$  and replace  $K$  with  $K'$ . This in turn may recursively call for the appropriate deletion of  $K'$  from the tree following one or more of the four cases.

**Case Dm. 3**  $C_i \neq \text{NIL}$  and  $C_j = \text{NIL}$ . If the right subtree of key  $K$  is empty and the left subtree is not, choose the largest key  $K''$  from the left subtree of  $K$  and replace  $K$  with  $K''$ . This in turn may recursively call for the appropriate deletion of  $K''$  from the tree following one or more of the four cases.

**Case Dm. 4**  $C_i \neq \text{NIL}$  and  $C_j \neq \text{NIL}$ . If the left subtree and the right subtree of key  $K$  are non empty, then choose either the largest key from the left subtree or the smallest key from the right subtree. Call it  $K'''$ . Replace key  $K$  with the same and as before undertake appropriate steps to delete  $K'''$  from the tree.

Let us delete 9 from the 4-way search tree shown in Fig. 11.2. The deletion belongs to Case Dm. 1 where both the left and right subtrees of key 9 are empty. We simply delete 9 from the node. The operation delete 11 belongs to Case Dm. 3 where the left subtree of key 11 is non empty and its right subtree is empty. We replace 11 with the largest key from its left subtree viz., 9 and delete 9 following Case Dm. 1. Finally, delete 84 illustrates Case Dm. 4 where both its left and right subtrees are not empty. In such a case, we choose to replace 84 with the smallest key from the right subtree of 84, viz., 90. The deletion of 90 in turn follows Case Dm. 1. The 4-way search tree after the deletion of 9, 11 and 84 are illustrated in Fig. 11.5 (a-c).



**Fig. 11.5** Deletion (independent) of keys 9, 11 and 84 from the 4-way search tree shown in Fig. 11.2

**Example 11.2** Consider a 5-way search tree shown in Fig. 11.6. Let us insert  $B$ ,  $Y$  and  $L$  in the sequence given into the original tree. Fig. 11.7 illustrates the insertions. While  $B$  and  $L$  call for insertion of the keys in the existing nodes,  $Y$  needs a new node to be opened since the node  $[S, T, X, Z]$  is full.

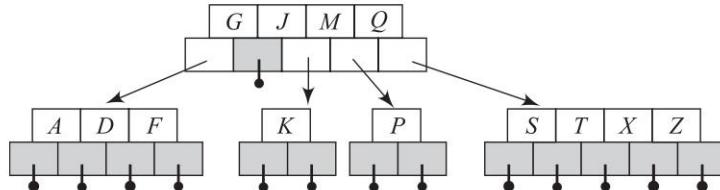


Fig. 11.6 An example 5-way search tree

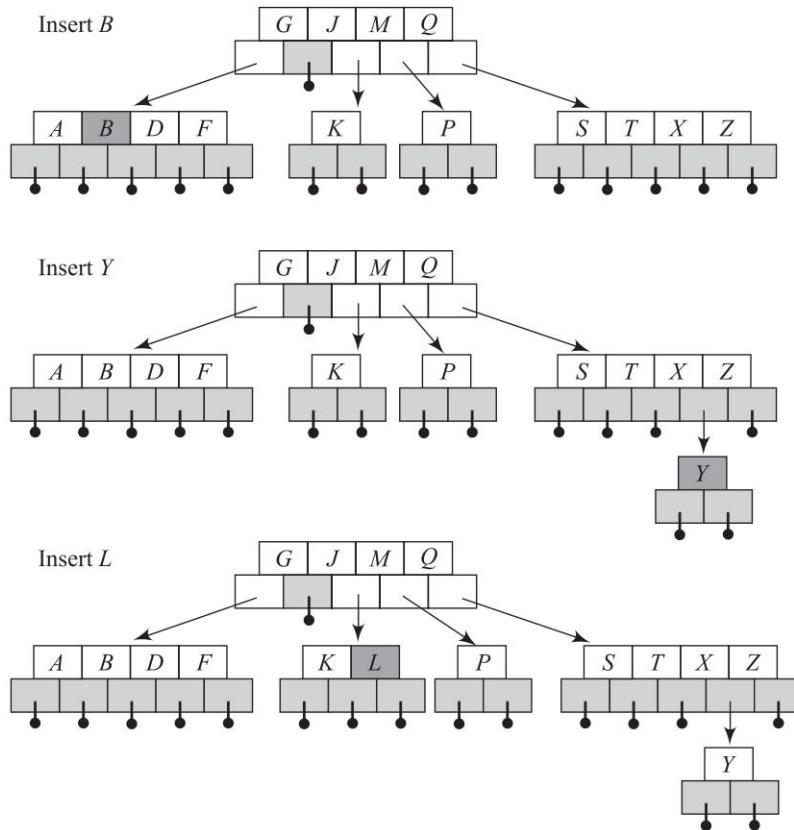
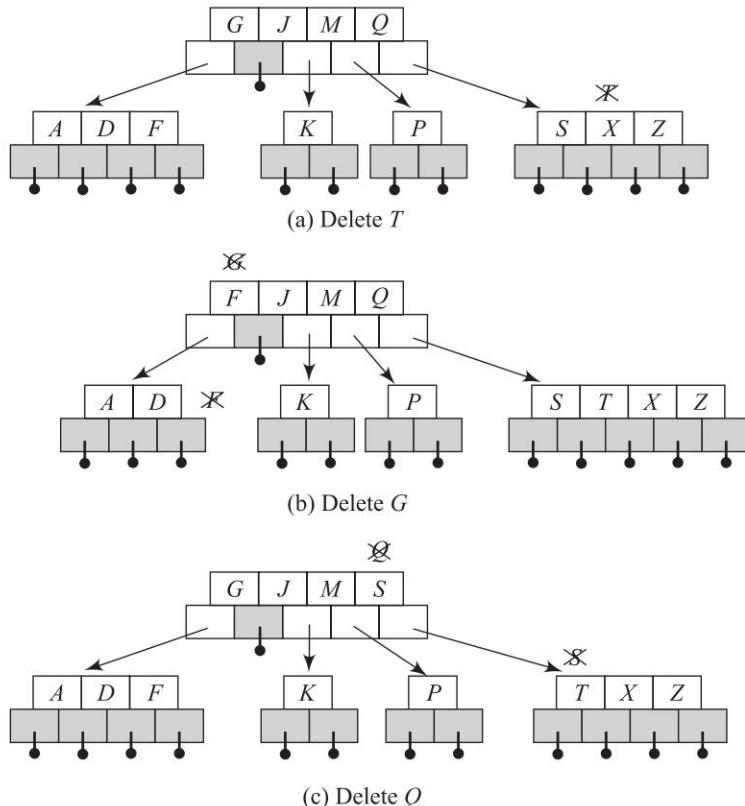


Fig. 11.7 Insertion of  $B$ ,  $Y$ ,  $L$  into the 5-way search tree shown in Fig. 11.6

**Example 11.3** In the 5-way search tree shown in Fig. 11.6, let us delete  $T$ ,  $G$ ,  $Q$  with each deletion independent of the other and performed on the original tree.

Delete  $T$  follows Case Dm. 1 of Sec. 11.2. We simply delete  $T$  and change the pointer fields accordingly. Delete  $G$  follows Case Dm. 3. We replace  $G$  with the largest key value chosen from

its left subtree viz.,  $F$ .  $F$  is deleted from its original position following Case Dm. 1. Delete  $Q$  follows Case Dm. 4.  $Q$  is replaced by the smallest element in the right subtree of  $Q$  viz.,  $S$ .  $S$  is deleted from its original position following Case Dm. 1. Figure 11.8 illustrates each of the three deletions.



**Fig. 11.8** Deletion (independent) of  $T$ ,  $G$  and  $Q$  from the 5-way search tree shown in Fig. 11.7

### Drawbacks of $m$ -way search trees

It is evident that the complexity of a search, insert and delete operation on an  $m$ -way search tree of height  $h$  (excluding external nodes) is given by  $O(h)$ . An  $m$ -way search tree of height  $h$  can have elements whose number lies between a minimum of  $h$  and a maximum of  $m^h - 1$ . A minimum of  $h$  elements would mean having one node with one element per node at each level. The maximum would be possible when each level has  $m$  child nodes and each node on each level has  $(m-1)$  elements. This would imply the maximum number of nodes in any level  $i$  to be given by  $m^{i-1}$ . The total number of nodes in an  $m$ -way search tree of height  $h$  would be given by

$$\sum_{i=1}^h m^{i-1} = \frac{(m^h - 1)}{(m - 1)}$$

Hence the maximum number of elements in the  $m$ -way search tree of height  $h$  would be given by  $\frac{(m^h - 1)}{(m - 1)} \cdot (m - 1) = (m^h - 1)$ .

Since the number of elements in an  $m$ -way search tree of height  $h$  varies from a minimum of  $h$  to a maximum of  $m^h - 1$ , if the tree represents  $n$  elements then the height varies from a minimum of  $\log_m(n + 1)$  to a maximum of  $n$ . Thus in the worst case the height of the  $m$ -way search tree representing  $n$  elements may be  $O(n)$  resulting in poor performance. Hence it is essential that even  $m$ -way search trees are maintained with balanced heights. B trees are height balanced  $m$ -way search trees and this data structure is detailed in the next section.

## B Trees: Definition and Operations

11.3

As pointed out in Sec. 11.2, if the growth of the  $m$ -way search trees are left unchecked for, then they may result in trees which yield a complexity of  $O(n)$  in the worst case thereby deteriorating its performance. Hence the need for balanced  $m$ -way search trees which are known as B trees of order  $m$ . B trees assure a complexity of  $O(\log n)$  for their search, insert and delete operations.

### Definition

A *B tree of order  $m$*  is an  $m$ -way search tree and hence may be empty. If non empty, then the following properties are satisfied on its extended tree representation:

- (i) The root node must have *at least two child nodes* and at most  $m$  child nodes
- (ii) All internal nodes other than the root node must have *at least*  $\left\lceil \frac{m}{2} \right\rceil$  non empty child nodes and *at most  $m$*  non empty child nodes
- (iii) The number of keys in each internal node is one less than its number of child nodes and these keys partition the keys of the tree into subtrees in a manner similar to that of  $m$ -way search trees
- (iv) All external nodes are at the same level

The node structure and representation of a B tree of order  $m$  is similar to that of an  $m$ -way search tree (Sec. 11.2). Figure 11.9 illustrates a B tree of order 5. The properties of B trees may be easily verified on the example tree.

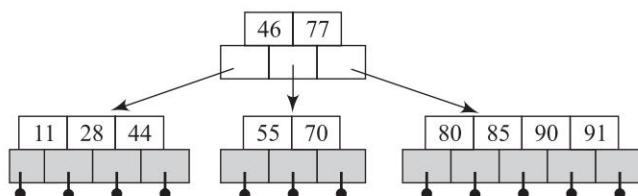
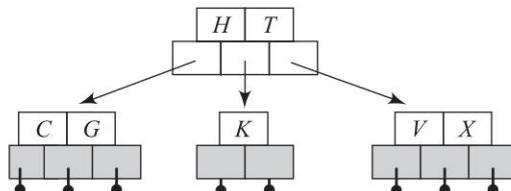


Fig. 11.9 A B tree of order 5

All internal nodes of the tree except the root have *at least*  $\left\lceil \frac{m}{2} \right\rceil = \left\lceil \frac{5}{2} \right\rceil = 3$  child nodes and hence have *at least two keys* in their respective nodes. The root node [46, 77] has *at least two child nodes*. All the external nodes indicated by circles are on the same level. The keys in each of the nodes partition the tree into subtrees following the principle of 5-way search trees.

B trees of order 3 are known as *2-3 trees* since each of their internal nodes have only two or three child nodes. Figure 11.10 illustrates a B tree of order 3. B trees of order 4 are called *2-4 trees* or *2-3-4 trees*. Rudolf Bayer (1972) who first discussed about a 2-4 tree called it a *symmetric binary B tree*.



**Fig. 11.10** A B tree of order 3 ( 2-3 tree)

### Searching a B tree of order $m$

The search procedure for a B tree of order  $m$  is same as the one applied on  $m$ -way search trees. The complexity of a search procedure is given by  $O(h)$  where  $h$  is the height of the B tree of order  $m$ . (See Sec. 11.3).

### Inserting into a B tree of order $m$

Inserting a key into a B tree of order  $m$  proceeds as one would to search for the key. However at the point where the search falls off the tree, the key is inserted based on the following norms (IB in the cases indicates Insertion in a B tree):

**Case IB. 1** If the node  $X$  of the B tree of order  $m$ , where the key  $K$  is to be inserted, can accommodate  $K$ , then it is inserted in the node and the number of child pointer fields are appropriately upgraded.

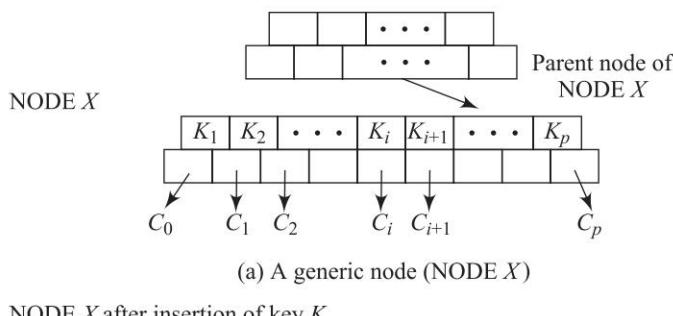
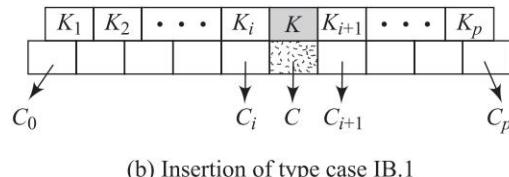
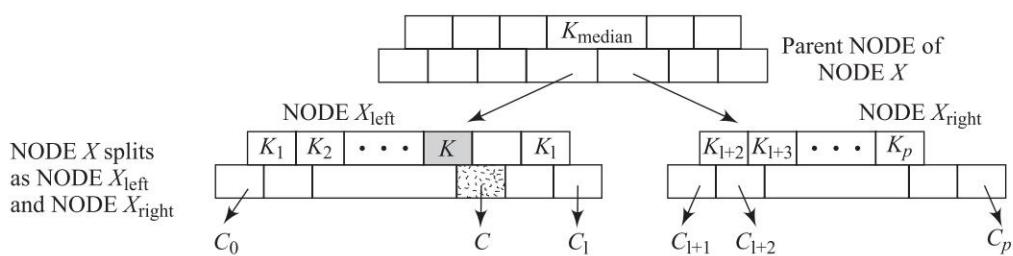
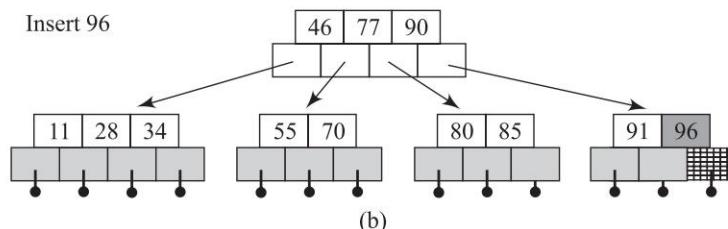
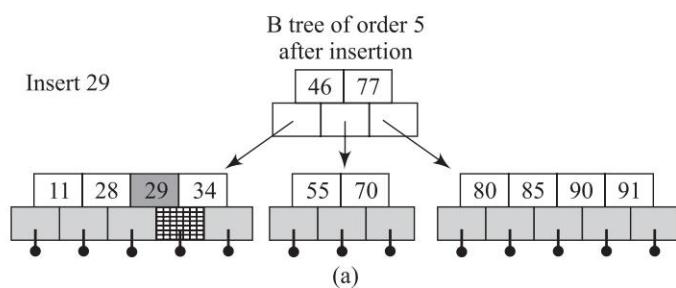
Thus if node  $X$  is given by,  $[C_0, (K_1, C_1), (K_2, C_2) \dots (K_i, C_i), (K_{i+1}, C_{i+1}) \dots (K_p, C_p)]$   $p < (m - 1)$  and  $K$  is such that  $K_i < K < K_{i+1}$  then we merely insert  $(K, C)$  where  $C$  is the child pointer field, at the appropriate position in the node  $X$ . The updated node  $X$  is given by  $[C_0, (K_1, C_1), (K_2, C_2) \dots (K_i, C_i), (K, C), (K_{i+1}, C_{i+1}) \dots (K_p, C_p)]$ .

Figure 11.11(b) illustrates Case IB. 1 type of insertion in the generic node shown in Fig. 11.11(a).

**Case IB. 2** If the node  $X$  where the key  $K$  is to be inserted is full, then we apparently insert  $K$  into the list of elements and split the list into two at its median  $K_{median}$ . The keys which are less than  $K_{median}$  form a node  $X_{left}$  and those greater than  $K_{median}$  form another node  $X_{right}$ . The median element  $K_{median}$  is pulled up to be inserted in the parent node of  $X$ . This insertion may in turn call for Case IB. 1 or Case IB. 2 depending on whether the parent node can accommodate  $K_{median}$  or not.

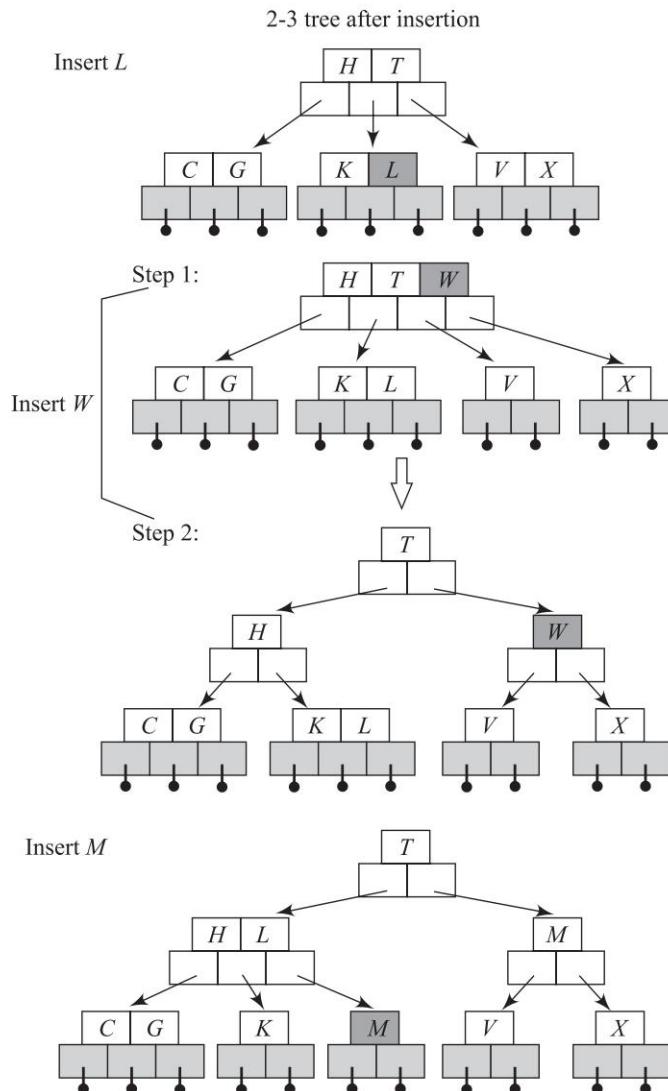
Figure 11.11(c) illustrates Case IB. 2 type of insertion in the generic node shown in Fig. 11.11(a).

Let us insert 29 into the B tree of order 5 shown in Fig. 11.9. The search for 29 falls off the tree at the node [11, 28, 34]. Since the node can accommodate an element more, 29 is inserted into the node with appropriate changes made in the number of child pointers. Figure 11.12(a) illustrates the B tree after insertion of 29. On the other hand, the insertion of 96 into the B tree results in the search falling off the node [80, 85, 90, 91]. However, the node is full. We now apparently insert

NODE  $X$  before insertion of key  $K$ NODE  $X$  after insertion of key  $K$ NODE  $X$  after insertion of key  $K$ **Fig. 11.11** Insertion of a key  $K$  in a B tree of order  $m$ **Fig. 11.12** Insertion of 29 and 96 in the B tree of order 5 shown in Fig. 11.9

96 into the list to obtain [80, 85, 90, 91, 96] and split the list into two at its median viz., 90. While [80, 85] form one node, [91, 96] form another node and the median element 90 is pulled up to be inserted in the parent node [46, 77]. Since the parent which is also the root can accommodate up to two more elements, 90 is inserted into the node to obtain the new root [46, 77, 90]. Figure 11.12(b) illustrates the insertion of 96 in the B tree.

**Example 11.4** For the 2-3 tree shown in Fig. 11.10 let us perform the following operations in the sequence given: Insert *L*, Insert *W* and Insert *M*.



**Fig. 11.13** Insertion of *L*, *W* and *M* for the 2-3 tree shown in Fig. 11.10

Insert  $L$  is a Case IB. 1 type of insertion and is accommodated straightaway in node  $[K]$ . Insert  $W$  on the other hand is more involved. It calls for the application of Case IB. 2 twice.  $W$  is virtually inserted into the node  $[V, X]$  to obtain the list  $[V, W, X]$ . The list  $[V, W, X]$  splits into two nodes  $[V]$  and  $[X]$  pulling  $W$  up to be inserted in the node  $[H, T]$ . This again triggers a split of the list  $[H, T, W]$  into two nodes  $[H]$  and  $[W]$  with  $[T]$  further pulled up to act as the new root. Insert  $M$  triggers Case IB.2 to obtain the nodes  $[K]$  and  $[M]$  with  $L$  accommodated in its parent as  $[H, L]$ .

### Deletion from a B tree of order $m$

The deletion of a key  $K$  from a B tree of order  $m$  may trigger various cases. It may be as simple as the cases DB. 1-2 or as complicated as cases DB. 3-4. (here **DB** indicates Deletion from a **B** tree)

**Case DB. 1** Key  $K$  belongs to a leaf node and its deletion does not result in the node having less than its minimum number of elements. This deletion is the simplest of the cases. In such a case we merely delete the element from the leaf node and adjust the child pointers accordingly.

**Case DB. 2** Key  $K$  belongs to a non leaf node. In such a case replace  $K$  with the largest key ( $K'$ ) in the left subtree of  $K$  or the smallest key ( $K''$ ) from the right subtree of  $K$  and follow steps to delete  $K'$  or  $K''$  from the node.  $K'$  or  $K''$  is bound to occur in a leaf node (why?) and hence triggers Case DB. 1 for their deletion.

Consider the B tree of order 5 shown in Fig. 11.9. Let us delete 80 and 77 both undertaken independently on the original B tree. Note that deletion of 80 follows Case DB.1. We merely delete 80 and adjust the child pointers. Deletion of 77 on the other hand follows Case DB. 2. We replace 77 with 80 the smallest key in its right subtree. The deletion of 80 now follows Case DB.1. Figure 11.14 shows the deletion of 80 and 77 from the B tree of order 5 (Fig. 11.9).

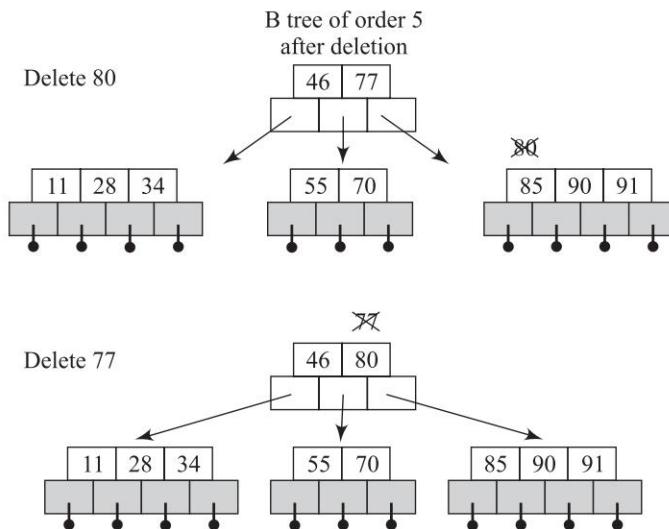


Fig. 11.14 Deletion of 80 and 77 from the B tree of order 5 shown in Fig. 11.9

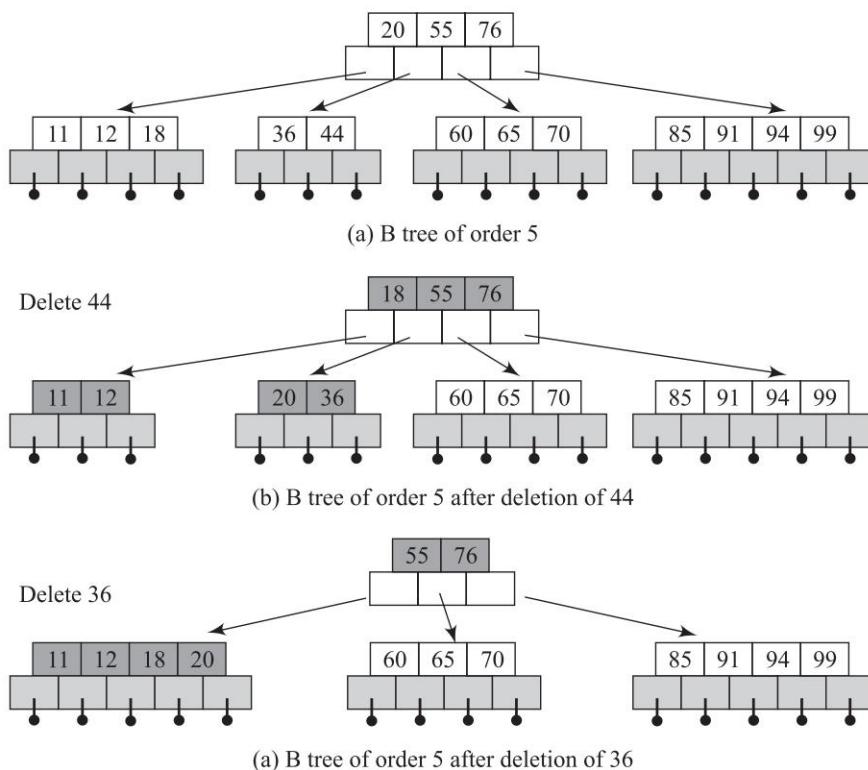
Deletions may turn out to be complicated when they leave less than the minimum number of elements in the nodes concerned. Cases DB. 3-4 illustrate these instances.

**Case DB. 3** When the deletion of a key  $K$  from a node  $X$  leaves it with less than its minimum number of elements, elements are borrowed from one of its left or right sibling nodes. Thus if the left sibling node has elements to spare, move the largest key  $K'$  in the left sibling node to the parent node. The intervening element  $P$  in the parent node is moved down to set right the vacancy created by the deletion of  $K$  in node  $X$ .

If the left sibling node has no element to spare it would be a waste of time to move to the right sibling node to check if there is an element to spare. What if after the check we were to realize that there were no elements to be spared by the right sibling node as well? In such a case we proceed to Case DB. 4 which covers the case when either of the sibling nodes have no elements to offer.

**Case DB. 4** When the deletion of a key  $K$  from a node  $X$  leaves its elements to be less than the stipulated minimum number and if the first tested sibling node (left or right) or both the sibling nodes are unable to spare an element, node  $X$  is merged with one of the sibling nodes along with the intervening element  $P$  in the parent node. We shall choose to test for the availability of element from the left sibling node first. If there is no element available to be spared, then the elements of the left sibling node are merged with those of node  $X$  and the intervening parent element  $P$  to create a new node. This in turn calls for the deletion of element  $P$  which may trigger one or more of the cases discussed above.

Consider the deletion of 44 from the B tree of order 5 shown in Fig. 11.15(a). This is a direct illustration of Case DB. 3. The operation would leave the node [36, 44] with less than its minimum number of keys and therefore we borrow 18 from the left sibling node which has an element to



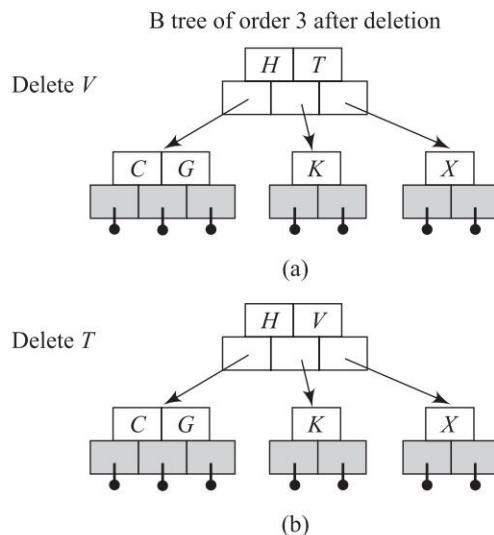
**Fig. 11.15** Deletion of 44 and 36 from a B tree of order 5

spare. Now 18 replaces the intervening parent element 20 and 20 moves down to fill the space created by the deletion of 44 in the node. Figure 11.15(b) illustrates the deletion of 44.

Let us proceed to delete 36 from the resulting B tree of order 5 shown in Fig. 11.15(b). Note this is a direct illustration of Case DB.4. Since the left sibling node has no elements to spare, we merge the left sibling node with the intervening parent element 18 and the node containing the only element after deletion of 36, viz., [20]. The new node [11, 12, 18, 20] is now a prospective child node. To decide on its appropriate parent we proceed to delete 18 from the parent node. Again the deletion of 18 from its node is problem free since the parent node can afford to do the same. Thus we obtain [55, 76] to be the updated parent node. The new node [11, 12, 18, 20] joins the root as its first subtree.

**Example 11.5** Consider the B tree of order 3 shown in Fig. 11.10. Let us delete  $V$  and  $T$  both undertaken independently on the original tree.

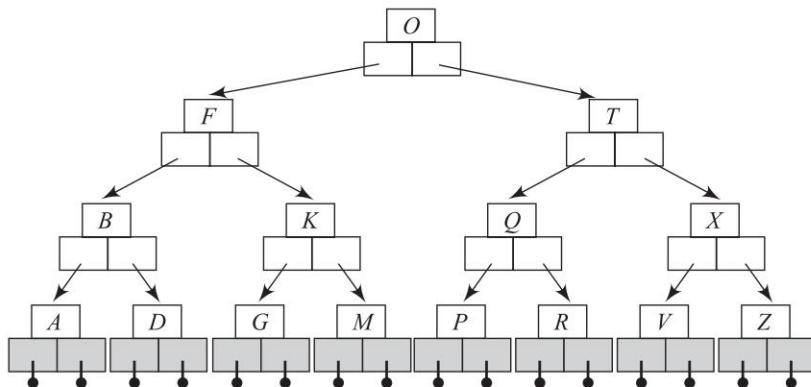
Deletion of  $V$  is a direct illustration of Case DB. 1. We merely delete  $V$  and adjust the child pointers of the node. Figure 11.16(a) shows the B tree after deletion of  $V$ . Deletion of  $T$  illustrates Case DB. 2 and hence replaces  $T$  with  $V$ , the largest key in the right subtree. Deletion of  $V$  from its original position in the node follows Case DB. 1. Figure 11.16(b) illustrates the B tree after deletion of  $T$ .



**Fig. 11.16** Deletion of  $V$  and  $T$  from the B tree of order 3 shown in Fig. 11.10

**Example 11.6** Given the B tree of order 3 shown in Fig. 11.17, let us delete  $M$ . To avoid clutter, the snapshots of the B trees during the delete process are shown without pointer fields. Observe how the deletion triggers Case DB.4 repeatedly before the tree gets balanced.  $M$  is a single key in a leaf node and its deletion would leave the node with zero elements. To borrow from the node  $[G]$  is futile and hence we undertake a merge operation as discussed in Case DB. 4 and this yields  $[G, K]$ . This leaves the parent node concerned with no elements and hence once again triggers Case DB.4. The new parent node is  $[B, F]$ . Observe the rearrangement of the child

pointers of the node. Case DB. 4 is once again triggered with regard to the empty parent of [B, F]. Finally the tree balances itself by settling on [O, T] as its root.



(a) B tree of order 3

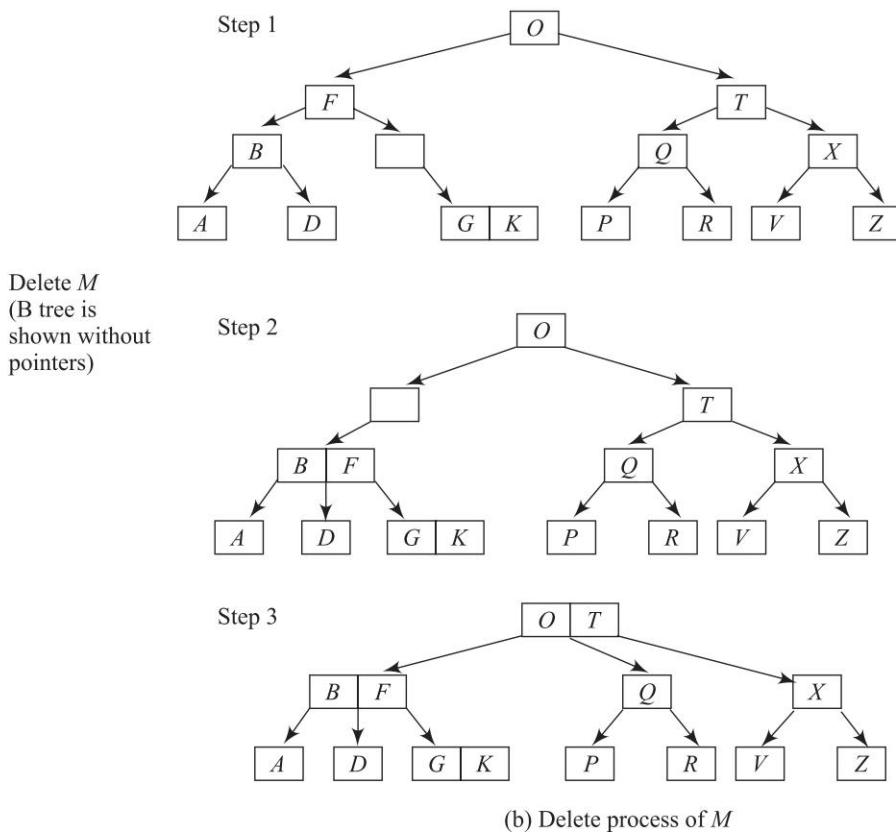


Fig. 11.17 Deletion of M from a B tree of order 3

## Height of a B tree of order $m$

If a B tree of order  $m$  and height  $h$  has  $n$  elements then  $n$  satisfies  $n \leq m^h - 1$ . This is true since a B tree of order  $m$  is basically an  $m$ -way search tree (Sec. 11.2). Now having determined the upper bound of  $n$ , what is its lower bound? In other words what is the minimum number of elements that a B tree of order  $m$  and height  $h$  can hold? To obtain this let us find out what are the minimum number of nodes in levels  $1, 2, \dots, (h+1)$ . Here  $(h+1)$  is the level at which the external

nodes reside. Since each internal node other than the root have a minimum of  $\left\lceil \frac{m}{2} \right\rceil$  child nodes and the root has just one node, the minimum number of nodes in each level beginning from 1

and ending at  $(h+1)$  in the sequential order would be  $1, 2, 2 \cdot \left\lceil \frac{m}{2} \right\rceil, 2 \cdot \left\lceil \frac{m}{2} \right\rceil^2, \dots, 2 \cdot \left\lceil \frac{m}{2} \right\rceil^{h-1}$

respectively. Thus the number of external nodes on level  $(h + 1)$  would be  $2 \cdot \left\lceil \frac{m}{2} \right\rceil^{h-1}$ . Since the

number of elements in the B tree is one less than the number of external nodes, the lower bound of  $n$  is given by  $n \geq 2 \cdot \left\lceil \frac{m}{2} \right\rceil^{h-1} - 1$ . Hence we have  $2 \cdot \left\lceil \frac{m}{2} \right\rceil^{h-1} - 1 \leq n \leq m^h - 1$ . From this we can

easily infer that  $\log_m (n + 1) \leq h \leq \log_{\left\lceil \frac{m}{2} \right\rceil} \left( \frac{n + 1}{2} \right) + 1$ . This determines the best case and worst case

complexities of a search, insert and delete operation on B trees which is generally given by  $O(h)$ , the height of the B tree.

## Tries: Definition and Operations

11.4

Search trees in general favor keys which are of fixed size since this leads to efficient storage management. However in the case of applications which are retrieval based and which call for keys of varying length, tries provide better options. Search trees indulge in multi-way branching based on the whole key and hence searching is done based on key comparisons. In contrast, though tries are also multi-way branched trees, searching is based only on a portion of a key and not on the whole, before it is completely retrieved or stored.

Tries are also called as *Lexicographic search trees*. The name *trie* (pronounced as “try”) originated from the word “retrieval”.

### Definition and representation

A trie of order  $m$  may be empty. If non empty, then it consists of an ordered sequence of exactly  $m$  tries of order  $m$ . The branching at any level of the trie is determined only by a portion and not by the whole key.

Alphabetical keys require a trie of order 27 (26 letters of the alphabet + a blank (“ ”)) for their storage and retrieval. Each branch of the trie partitions the keys into groups beginning with the specific alphabet.

Thus tries have two category of node structures, viz., *branch node* and *information node*. A branch node is merely a collection of LINK fields each pointing either to a branch node or to an

information node. An information node holds the key that is to be stored in the trie. For example, in the case of alphabetical keys, each branch node has 27 LINK fields, one for each of the 26 alphabet characters and one for a blank (" "). The keys are stored in the information nodes. To access an information node containing a key, we need to move down a branch node or a series of branch nodes following the appropriate branch based on the alphabetical characters composing the key. All LINK fields that neither point to a branch node nor to an information node are represented using null pointers. To avoid clutter, null pointers have not been represented using any special notations.

Figure 11.18 illustrates an example trie for alphabetical keys. The trie stores the keys CAR, CARRIAGE, CARAVAN, BIKE, BUS, TRAIN, BICYCLE, AEROPLANE. The information nodes wholly store the keys. To access these information nodes, we follow a path beginning from a branch node moving down each level depending on the characters forming the key, until the appropriate information node holding the key is reached. Thus the depth of an information node in a trie depends on the similarity of its first few characters (*prefix*) with its fellow keys. Here, while AEROPLANE and TRAIN occupy shallow levels (level 1 branch node) in the trie, CAR, CARRIAGE, CARAVAN have moved down by four levels of branch nodes due to their uniform prefixes "CAR". Observe how we move down each level of the branch node with the help of the characters forming the key. The role played by the blank field in the branch node is evident when we move down the trie to access CAR. While the information node pertaining to CAR positions itself under the blank field, those of CARAVAN and CARRIAGE attach themselves to pointers from A and R respectively of the same branch node.

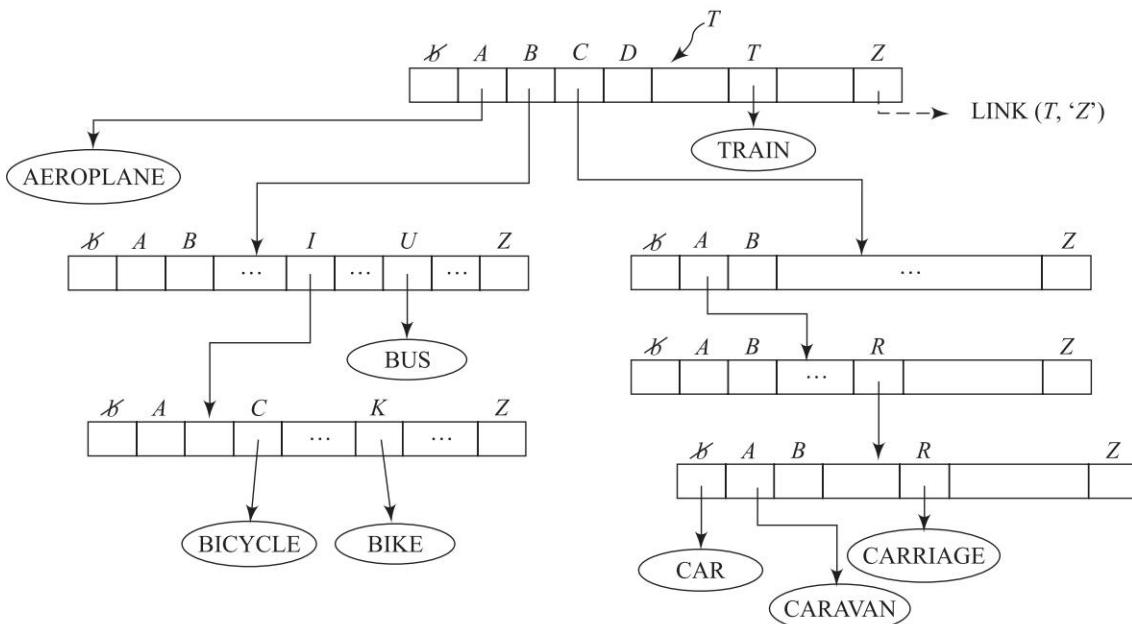


Fig. 11.18 An example trie

## Searching a trie

To search for a key  $K$  in a trie  $T$ , we begin at the root which is a branch node. Let us suppose the key  $K$  is made up of characters  $k_1 k_2 k_3 \dots k_n$ . The first character of the key  $K$  viz.,  $k_1$  is extracted and the LINK field corresponding to the letter  $k_1$  in the root branch node is spotted. If  $\text{LINK}(T, k_1)$ , the LINK field of character  $k_1$  corresponding to the branch node  $T$ , is equal to NIL, then the search is unsuccessful, since no such key is found. If  $\text{LINK}(T, k_1)$  is not equal to NIL, then the LINK field may either point to an information node or a branch node. If the information node holds  $K$  then the search is done. The key  $K$  has been successfully retrieved. Otherwise, it implies the presence of key(s) with a similar prefix. We extract the next character,  $k_2$  of key  $K$  and move down the LINK field corresponding to  $k_2$  in the branch node encountered at level 2 and so on until the key is found in an information node or the search is unsuccessful. The deeper the search, the more there are keys with similar but longer prefixes.

**Example 11.7** Consider the trie  $T$  shown in Fig. 11.18. Let us search for the keys TRAIN and CARAVAN. To search for the key TRAIN, we extract the first character 'T' and move down the LINK field corresponding to 'T' in the root branch node. The retrieval is successful since the information node corresponding to the LINK holds the key TRAIN. Let us proceed to retrieve CARAVAN. The first character C urges the search process to move down  $\text{LINK}(T, 'C')$  in the first branch node. The second character A again leads one to move down to the next level and so does R. At level four, the LINK field corresponding to the fourth character viz., 'A' leads to an information node holding the key CARAVAN. The path can be easily traced on the trie shown.

## Insertion into a trie

To insert a key  $K$  into a trie we begin as we would to search for the key  $K$ , possibly moving down the trie, following the appropriate LINK fields of the branch nodes, corresponding to the characters of the key. At the point where the LINK field of the branch node leads to NIL, the key  $K$  is inserted as an information node.

**Example 11.8** Consider the trie shown in Fig. 11.18. Let us insert SHIP and TRAM into the tree. Insertion of SHIP is simple and straight forward. The LINK field corresponding to the first character S in the root node is NIL and hence we insert SHIP as an information node in the appropriate place of the root branch node. In the case of TRAM, the LINK field corresponding to 'T' in the root branch node points to an information node holding TRAIN. This implies that there is already a key with a uniform prefix available in the trie. We now remove TRAIN and instead open a branch node to accommodate both TRAIN and TRAM. And lo! the second character of the two keys matches and so does the third! Since the matching prefixes of TRAIN and TRAM ("TRA") is of length 3, the situation now calls for opening three levels of branch nodes other than the root node. It is only at level 4 that TRAIN and TRAM can be inserted as information nodes corresponding to the LINK fields of I and M respectively. Figure 11.19 illustrates the insertion of SHIP and TRAM in the trie shown in Fig. 11.18.

## Deletion from a trie

The deletion of a key  $K$  from a trie proceeds as one would to search for the key. On reaching the information node (NODE  $I$ ) holding  $K$ , the same is deleted. But deletion does not merely stop

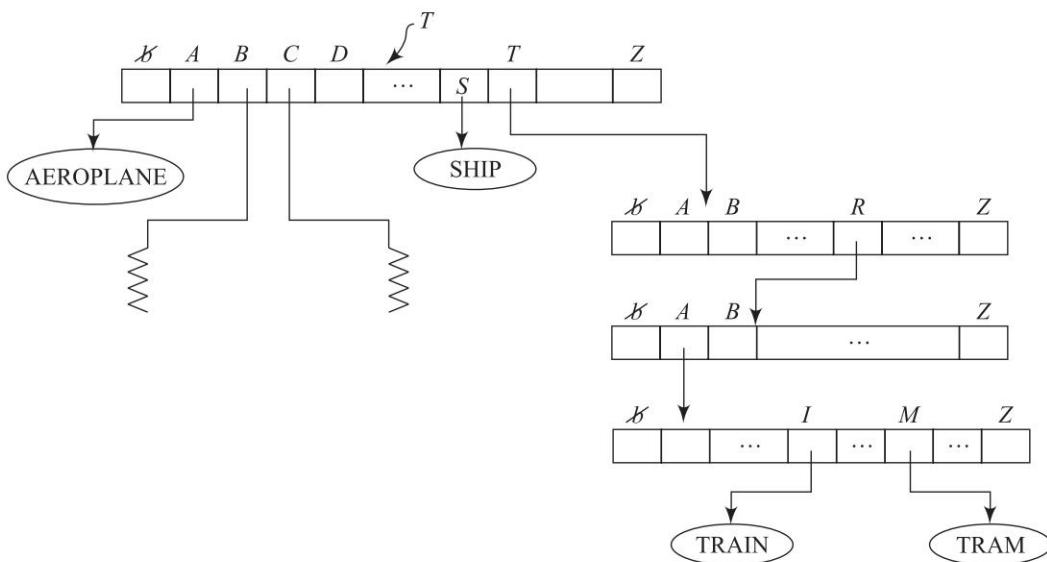
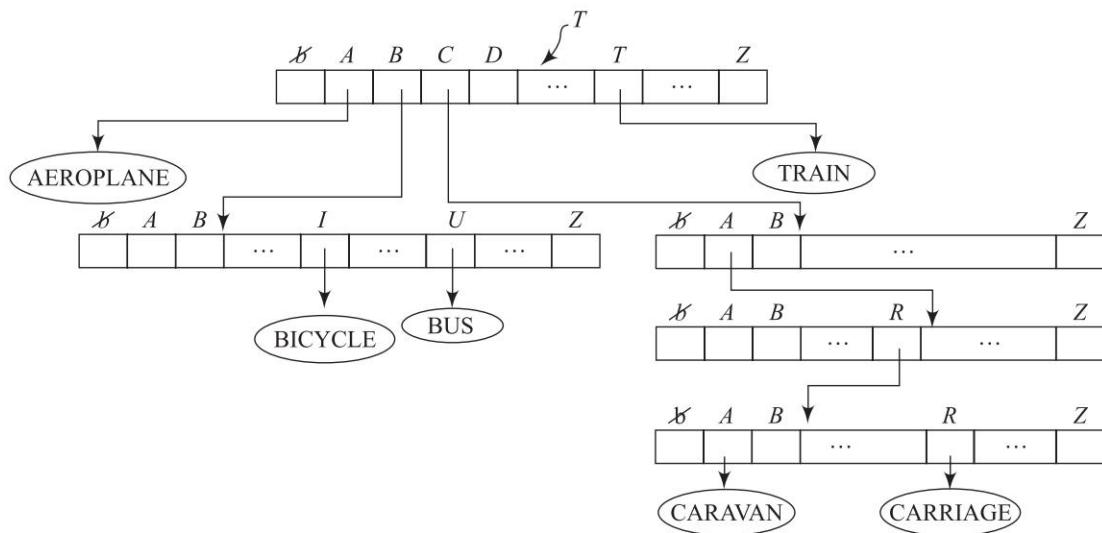


Fig. 11.19 Insertion of SHIP and TRAM into the trie shown in Fig. 11.18

with this. It needs to be ensured whether the branch node to which NODE  $I$  is linked accommodates other information nodes as well! If there are more than one information nodes linked to the branch node concerned or if there is at least one LINK field to another branch node or both, then the deletion is done. We merely delete the information node holding the key. On the other hand, if after deletion of NODE  $I$ , it leaves the branch node with just one more key (information node NODE  $I'$ ) then there is no reason why the branch node should be retained at all. We delete the branch node and push NODE  $I'$  to a level higher. If the situation leads to NODE  $I'$  being the only non empty node in the current branch node, once again we delete the branch node and push NODE  $I'$  higher until it finds a position in a branch node that makes the best use of its LINK fields. Since the deletions are sensitive to the number of keys (information nodes) that are present in a branch node it would be prudent to include a COUNT field in each branch node recording the number of information nodes that are attached to the branch node.

**Example 11.9** Let us delete CAR and BIKE from the trie shown in Fig. 11.18. To delete CAR, we search for it moving down four levels of branch nodes and spot it at an information node before deleting the same. This leaves us with the specific branch node holding two more keys viz., CARAVAN and CARRIAGE. Therefore there is nothing that can be done to the branch node and the deletion of CAR is deemed complete.

In the case of BIKE we proceed as before and delete the information node holding the key. But note this leaves the branch node with a single key BICYCLE. We therefore delete the branch node and proceed to accommodate BICYCLE in the branch node that is a level higher. The current branch node holds the key BUS in the LINK field corresponding to 'U'. The key BICYCLE is attached to it corresponding to the LINK field of I. Figure 11.20 shows the deletion of CAR and BIKE from the trie shown in Fig. 11.18.



**Fig. 11.20** Deletion of CAR and BIKE from the trie shown in Fig. 11.18

### Some remarks on tries

The performance of search trees is determined by the number of keys that form the tree. Recall that the complexities of the search, delete and insert operations were given by  $O(h)$  where the height  $h$  is dependent on the number of keys represented in the search tree. In contrast, the performance of the trie is dependent on the length of the key- the number of characters forming the key- rather than the number of keys itself. Thus for example, if the length of the keys of a trie are equal to 7 then the trie can represent  $(26)^7 = 8031810176$  combinations of keys and with the maximum length of uniform prefixes within the keys being 6 can retrieve keys in at most 7 comparisons. In contrast, a search tree such as a binary search tree would need approximately  $\log_2((26)^7) \approx 33$  comparisons for the same!

In general however, most applications involve keys of large lengths and the number of keys to be represented in the trie may be sparse when compared to its capacity. In such a case, tries may be expected to perform less better than their search tree counterparts. It is therefore recommended that tries which are multi-way trees are used in judicious combinations with other search trees.

## Applications

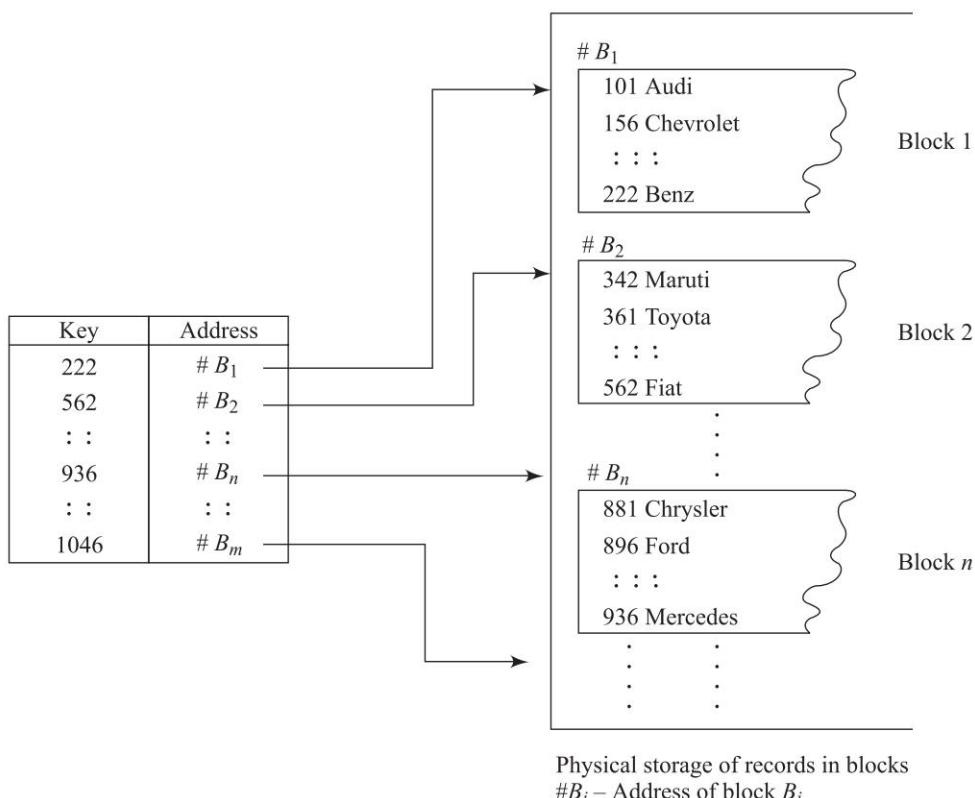
## 11.5

Most of the data structures including search trees such as binary search trees and AVL trees are suitable for *internal searching* (i.e.) searching related to small files that can be accommodated in the internal memory of the computer. In the case of applications that call for very large files or data bases with voluminous records, the files cannot be accommodated in the internal memory of the computer. Hence these need to be stored in *external memory* or what are called *auxiliary storage* or *external storage devices* such as hard disks, drums etc. While internal memory access is very fast, the problem with these external devices are that accesses are time consuming. For

example, to retrieve a record residing in a hard disk, the block in which the record resides is to be first accessed, next the entire block of records needs to be read and finally the required record is to be retrieved. Hence it is essential that files stored in the external memory resort to strategies and techniques resulting in their efficient retrieval and storage. Chapter 14 details methods of file organization. It is in this context that multi-way trees such as  $m$ -way search trees, B trees and Tries find their application.

## File indexing

Retrieval of records from large files or data bases stored in external memory is time consuming. To promote efficient retrievals, file indexes are maintained. An *index* is a  $\langle \text{key}, \text{address} \rangle$  pair. The purpose of indexing is to expedite the search process or retrieval of a record. Though there are more than one file management techniques which employ indexing, Indexed Sequential Access Method (ISAM) based files have been the foremost in using indexing for efficient retrievals (see Sec. 14.7). The records of the file are sequentially stored and for each block of records, the largest key and the block address is stored in an index. To retrieve a record whose key is  $K$ , the index is first searched to obtain the address of the block and thereafter a sequential search of the block should yield the desired record. Figure 11.21 illustrates an ISAM file structure. However if the file is too large, then index over indexes may have to be built.



**Fig. 11.21** An ISAM file structure

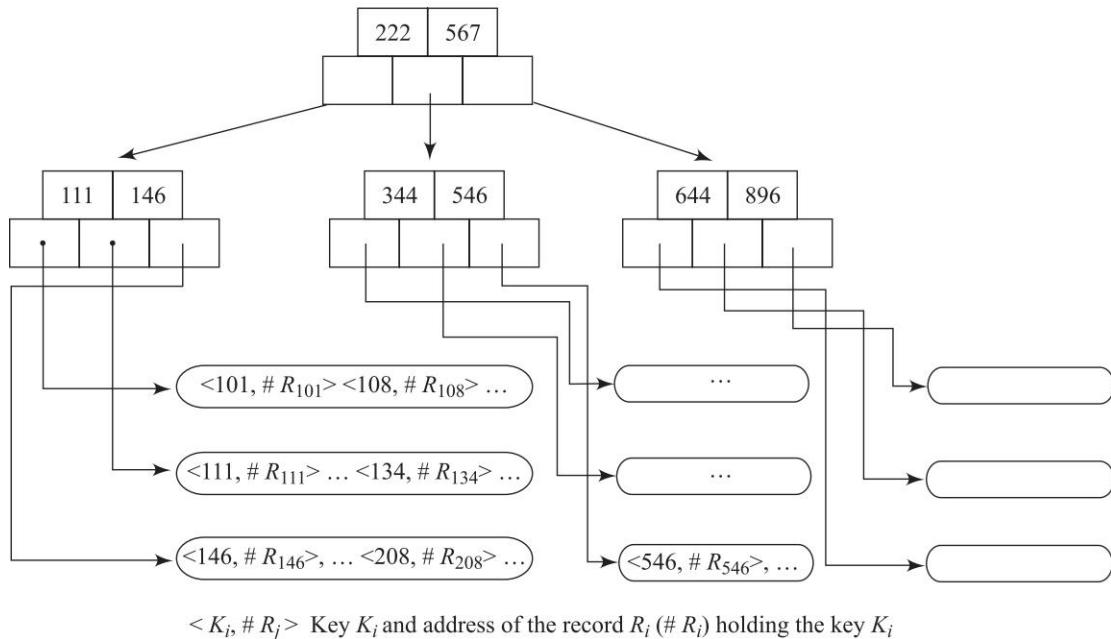
From the above it is clear that efficient retrievals now are dependent on the indexes. Though indexes are basically look up tables, it is essential that they are represented using efficient data structures to expedite retrievals. It is here that one finds the application of multi-way trees such as  $m$ -way search trees, B trees and tries.

## B trees as file indexes

B trees are ideally suited for storing file indexes. Each internal node of the B tree stores the  $\langle \text{key}, \text{address} \rangle$  pair. Their balanced heights call for fewer node accesses during the retrievals. Once the key is found the address of the record is also accessed along with it thereby speeding up the retrieval process.

## $B^+$ trees as file indexes

$B^+$  trees are descendants of B trees. They satisfy all the properties of B trees but for a modification in the structure of the leaf nodes. While leaf nodes in B trees hold null pointers (external nodes in fact), the leaf nodes of  $B^+$  trees point to storage areas which contain records having the appropriate key values or pointers to each of these records. Therefore in a  $B^+$  tree to retrieve a record given its key, it is essential that the search traverses down to a leaf node to retrieve its address. The non leaf nodes only serve to help the search process traverse downwards towards the appropriate leaf node. Figure 11.22 illustrates a file index stored as a  $B^+$  tree.



**Fig. 11.22**  $B^+$  tree representation of a file index

In comparison to  $B^+$  trees, B trees are more efficient for the following reasons:

- (i) The  $\langle \text{key}, \text{address} \rangle$  pair of the records are stored directly in the internal nodes of the B trees only once thereby saving on storage. In contrast,  $B^+$  trees store the keys in duplicate, once in the internal nodes and the next in the storage areas pointed to by the leaf node pointers.
- (ii) Unlike  $B^+$  trees, to access the  $\langle \text{key}, \text{address} \rangle$  pair in a B tree, there is no need to traverse down the whole tree to reach the leaf node. The  $\langle \text{key}, \text{address} \rangle$  pair may be found directly in the respective internal nodes. Therefore the keys may be accessed in fewer accesses when compared to  $B^+$  trees.

## Spell checker

Most word processing software embed spell checking which is provided online. Any incorrectly spelt word is automatically highlighted. Tries find an application in this problem. The words of a dictionary are stored as a trie and remembered in the memory of the computer. Every time a word is typed, the word is searched for in the trie and anything which does not lead to an information node is highlighted as an incorrectly spelt word. However, practical considerations call for curtailing the size and storage requirements of the trie since the whole trie needs to be present in the memory at the time of spell checking.



## Summary

- $m$ -way search trees, B trees of order  $m$  and tries are examples of multi-way search trees.
- $m$ -way search trees are extensions of binary search trees. Searching for a key in a  $m$ -way search tree is similar to that in a binary search tree. Insertion of a key in a node is directly done if the node has less than its maximum number of elements ( $m-1$ ). On the other hand if the node is full then we insert the key as a new node in the next level at the appropriate position.
- To delete a key from a  $m$ -way search tree, if the key has both its left and right subtrees to be empty then merely delete the key. If any one of the subtrees are non empty or both are non empty, then we replace the key to be deleted by either the smallest key in its left subtree or the largest key in its right subtree as the case may be.
- Since the height of a  $m$ -way search tree of  $n$  elements varies from a minimum of to a maximum of  $n$ , the worst case performance of the tree may yield  $O(n)$ . Hence the need for height balanced  $m$ -way search trees.
- B trees of order  $m$  are height balanced  $m$ -way search trees. The insertion of an element in a B tree is direct if the node is partially full. If the node is full, the key is virtually inserted into the list of keys in the node and the same is split into two nodes at its median element. The median element is pushed up to be accommodated in the parent node. This in turn may trigger further adjustments amongst the key elements of the parent node.

- The deletion of a key from a leaf node in a B tree is direct if it does not leave the node with less than its minimum number of elements. If the deletion belongs to a non leaf node then replace it with either the largest key in its left subtree or the smallest in its right subtree. If the deletion leaves a node with less than its minimum number of elements, then borrow an element either from the left sibling node or the right sibling node provided they have an element to spare. Otherwise, merge the node with one of its sibling nodes along with the intervening parent element to form a new node. This calls for the deletion of the intervening parent element concerned.
- Tries are search trees based on searching with portions of keys rather than the whole keys themselves. The search, insert and delete operation proceed down the trie along the branch nodes to reach the information node where the appropriate operation is carried out.
- The application of B trees to file indexing and Tries to spell checking have been discussed.

## Illustrative Problems

**Problem 11.1** For the 5-way search tree shown in Fig. I 11.1 perform the following operations in the sequence shown:

Insert  $u$ , Delete  $z$ , Insert  $b$ , Delete  $p$

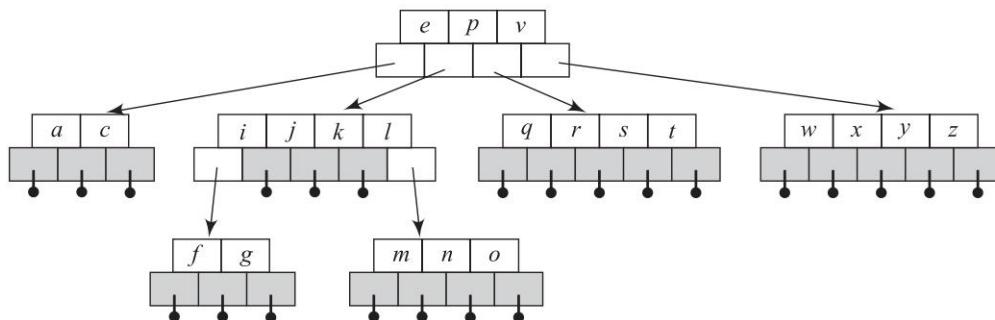
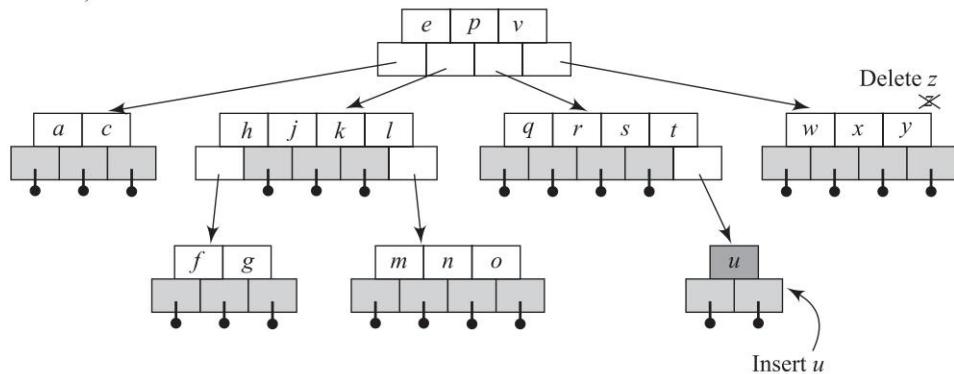
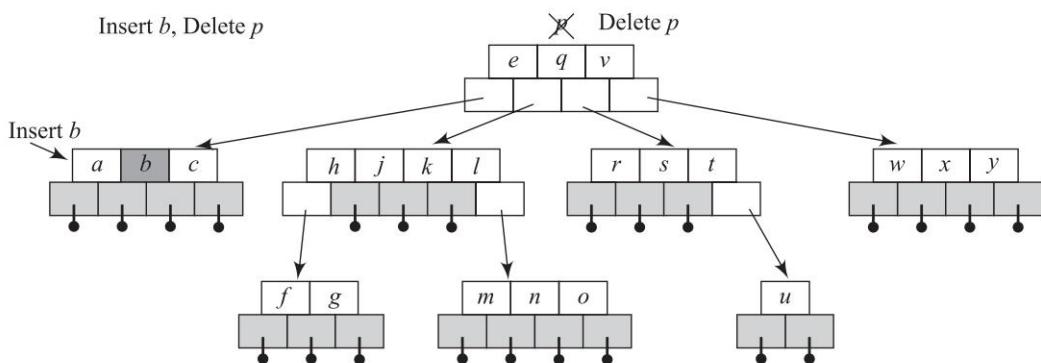


Fig. I 11.1

**Solution:** For insertion of  $u$ , the node  $[q, r, s, t]$  is full and hence cannot accommodate  $u$ . Therefore we create a new node  $[u]$ . For delete  $z$  operation, since the left and right subtree pointers of  $z$  are NIL, we can easily delete  $z$  from the node  $[w, x, y, z]$ . Insert  $b$  accommodates  $b$  into the node  $[a, c]$  to form the node  $[a, b, c]$ . Lastly, delete  $p$  calls for the case where the left and right subtrees of the key to be deleted are not NIL. We choose the smallest key of its right subtree, viz.,  $q$  and replace  $p$  with  $q$ . In turn  $q$  is deleted from its node directly since its left and right subtree pointers are NIL. The snapshots of the 5-way search tree for the given operations are shown below:

Insert  $u$ , Delete  $z$ Insert  $b$ , Delete  $p$ 

**Problem 11.2** What is the maximum and minimum number of elements a 100-way search tree of height 3 can hold? What is its maximum and minimum height if  $(100)^2$  elements are represented in the tree?

**Solution:** Following the discussion of Sec. 11.2, the maximum number of elements a 100-way search tree of height 3 can hold is  $(100)^3 - 1$  while the minimum number it can hold is equal to its height which is given to be 3. If the tree were to represent  $(100)^2$  elements, then the maximum height of the search tree would be  $(100)^2$  and the minimum height would be  $\log_{100}((100)^2 + 1) \approx 2$ .

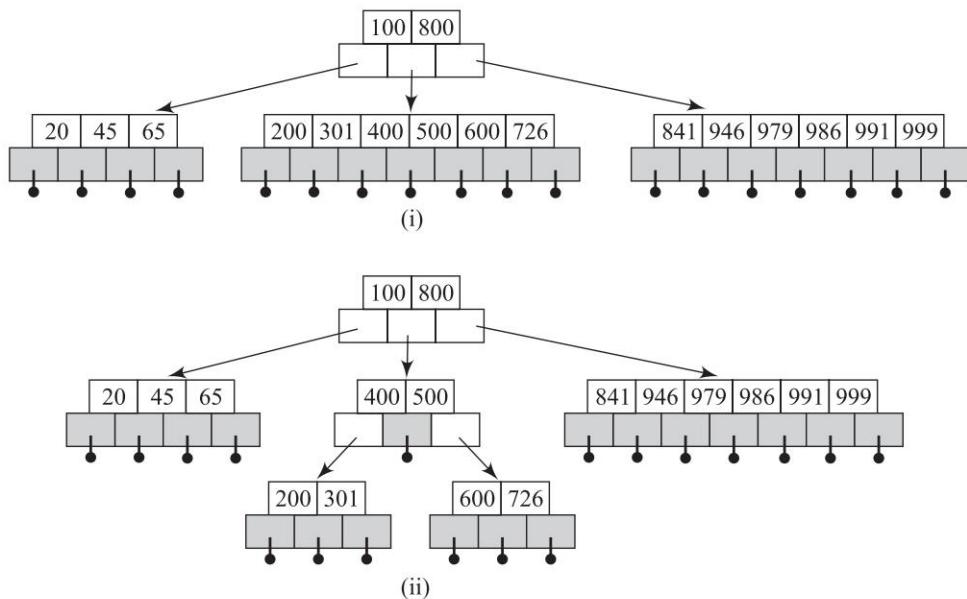
**Problem 11.3** For the 5-way search tree shown in Fig. I 11.1 how many comparisons are needed to search for the elements  $n$  and  $z$ ?

**Solution:** Searching for  $n$  requires 8 comparisons and searching for  $z$  requires 7 comparisons.

**Problem 11.4** Which of the following is a B tree of order 7? If so, why?

**Solution:** The tree shown in (i) is a B tree of order 7, since the following properties appropriate to the tree are satisfied:

- (i) The root node must have at least two child nodes and that of the given tree has 3 child nodes



(ii) All internal nodes other than the root have at least  $\lceil \frac{7}{2} \rceil = 4$  child nodes

(iii) Each node of  $p$  child nodes has  $(p-1)$  key elements

(iv) All external nodes are at the same level.

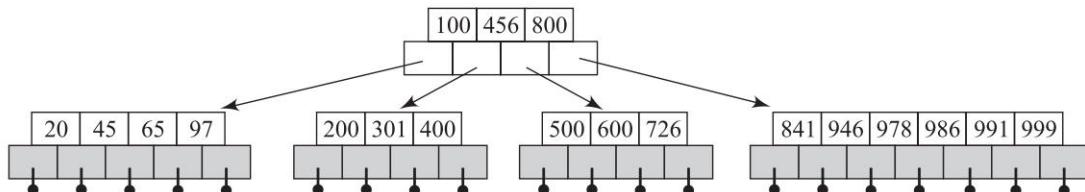
The tree shown in (ii) is not a B tree of order 7 (but possibly a 7-way search tree) since all the external nodes are not on the same level and some of the internal nodes have less than  $\lceil \frac{7}{2} \rceil = 4$  child nodes.

**Problem 11.5** In the tree shown in figure (i) of Illustrative Problem 11.4 undertake the following operations:

Insert 456, Insert 97

**Solution:** To insert 456 into the B tree of order 7 shown in the figure (i) of Illustrive Problem 11.4, we arrive at the node [200, 301, 400, 500, 600, 726] while searching for it. Since the node is full we virtually insert 456 into the node and split the node into two at its median which is 456. While 456 is absorbed in the root which can accommodate it, the split nodes are [200, 301, 400] and [500, 600, 726].

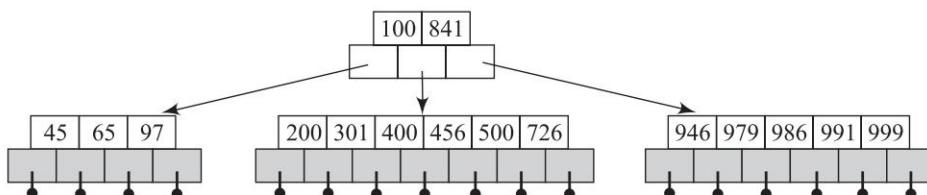
The insertion of 97 is simple and straight. It is accommodated in the node [20, 45, 65]. The B tree after the two insertions is shown below:



**Problem 11.6** In the B tree of order 7 obtained in Illustrative Problem Fig. 11.5, perform the following operations in the sequence given:

delete 600, delete 800 and delete 20

**Solution:** Delete 600 leaves the node [500, 600, 726] with fewer than its minimum number of elements. To borrow an element from its sibling (left sibling) is futile, since it would leave the sibling node concerned with less than its minimum number of elements. Therefore the left sibling node [200, 301, 400] is merged with [500, 726] along with its intervening parent element 456. Deletion of 800 in the resulting tree is done by merely replacing 800 by the smallest key in its right subtree viz., 841. Deletion of 20 is simple and direct. The key 20 is merely removed from its node. The B tree, after the three deletions that have been carried out is shown below:



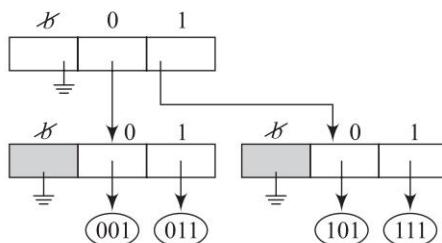
**Problem 11.7** The *preorder traversal* of a B tree of order  $m$  is undertaken by visiting all entries of the root node first, followed by traversing each of the subtrees from left to right in preorder. A *postorder traversal* of a B tree of order  $m$  is undertaken by first traversing all its subtrees from left to right in post order and finally visiting all the entries in the root node.

For the final snapshot of the 2-3 tree shown in Fig. 11.17(b) in the text, undertake preorder and post order traversals.

**Solution:** The preorder traversal of the 2-3 tree yields: O T B F A D G K Q P R X V Z  
The postorder traversal of the 2-3 tree yields: A D G K B F P R Q V Z X O T

**Problem 11.8** Construct a trie for the binary keys 011, 111, 101, 001

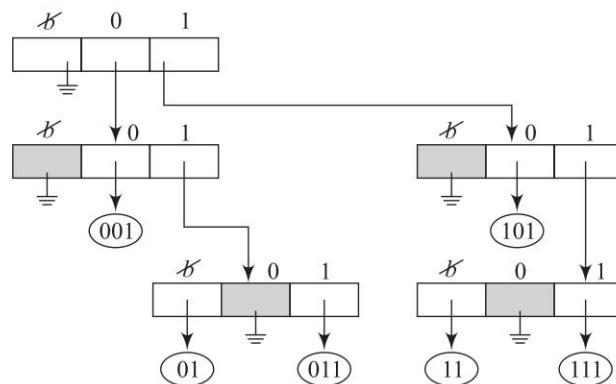
**Solution:** The trie for the binary keys is as given below:



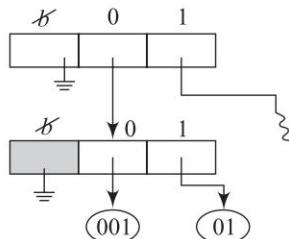
**Problem 11.9** Perform the following operations on the trie constructed in Illustrative Problem I 11.8:

Insert 01, Insert 11, Delete 011, Delete 001

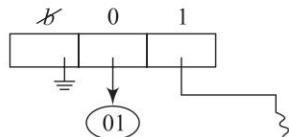
**Solution:** The trie after the insert operations is as follows:



The relevant portion of the trie after the deletion of 011 is as given below:



The relevant portion of the trie after the deletion of 001 is as given below:

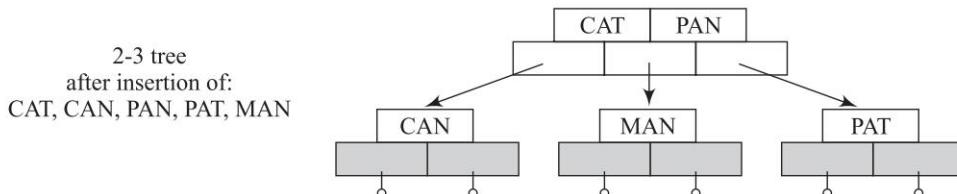


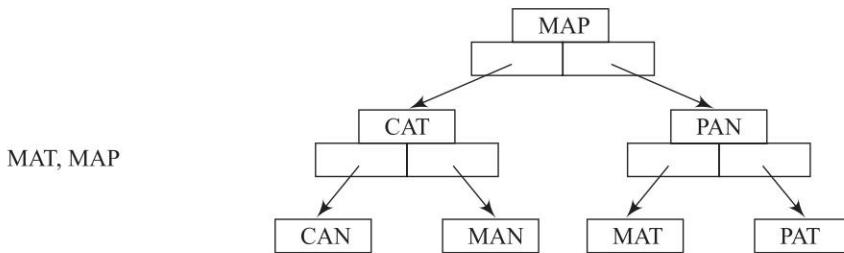
**Problem 11.10** Construct (i) 2-3 tree and (ii) trie for the following keys in the order of their appearance:

CAT, CAN, PAN, PAT, MAN, MAT, MAP

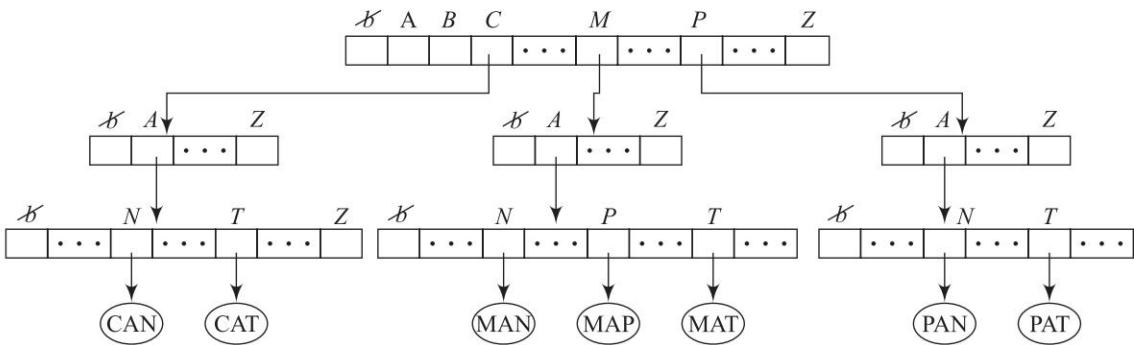
**Solution:**

(i) The snapshots of the 2-3 tree during its construction are shown below:





(ii) The trie after insertion of the key elements is shown below:



## Review Questions

- Which among the following properties does not hold good for an  $m$ -way search tree?
  - each node has at least  $m$  child nodes.
  - if a node has  $k$  child nodes,  $k \leq m$ , then the node has exactly  $(k-1)$  keys and each of the keys  $K_i$  partitions the keys in the subtrees into  $k$  subsets.
  - for a node  $C_0$ ,  $(K_1, C_1)$ ,  $(K_2, C_2)$ ,  $(K_3, C_3)$ , ...,  $(K_{m-1}, C_{m-1})$  all key values in the sub tree pointed to by  $C_i$  are less than the key  $K_{i+1}$ ,  $0 \leq i \leq m-2$  and the key values in the sub tree pointed to by  $C_{m-1}$  are greater than  $K_{m-1}$ .
  - the subtrees pointed to by  $C_i$ ,  $0 \leq i \leq m-1$  are also  $m$ -way search trees.
    - (i)
    - (ii)
    - (iii)
    - (iv)
- To delete key  $K$  found in a node of an  $m$ -way search tree, with its left subtree empty and right subtree non empty,
  - simply delete the key  $K$
  - choose the smallest key  $K'$  from the right subtree of  $K$  and replace  $K$  with  $K'$ , which in turn may recursively call for the appropriate deletion of  $K'$  from the tree
  - choose the largest key  $K''$  from the left subtree of  $K$  and replace  $K$  with  $K''$  which in turn may recursively call for the appropriate deletion of  $K''$  from the tree.

- (iv) choose either the largest key from the left subtree or the smallest key from the right subtree ( $K'''$ ) and replace key  $K$  with  $K'''$  which in turn may recursively call for the appropriate deletion of  $K'''$  from the tree.
- (a) (i) (b) (ii) (c) (iii) (d) (iv)
3. Which among the following properties is not satisfied by a B tree of order  $m$ ?
- (i) The root node must have *at least*  $m$  child nodes,  $m \geq 1$ .
- (ii) All internal nodes other than the root node must have *at least*  $\left\lceil \frac{m}{2} \right\rceil$  non empty child nodes and at most  $m$  non empty child nodes.
- (iii) The number of keys in each internal node is one less than its number of child nodes and these keys partition the keys of the tree into subtrees in a manner similar to that of  $m$ -way search trees.
- (iv) All external nodes are at the same level.
- (a) (i) (b) (ii) (c) (iii) (d) (iv)
4. In the context of insertion of a key  $K$  into a B tree of order  $m$ , state whether true or false:
- (i) If the node  $X$  of the B tree of order  $m$ , where the key  $K$  is to be inserted, can accommodate  $K$ , then it is inserted in the node and the number of child pointer fields are appropriately upgraded.
- (ii) If the node  $X$  where the key  $K$  is to be inserted is full, then we apparently insert  $K$  into the list of elements and split the list into two at its median  $K_{median}$ .
- (a) (i) true (ii) true (b) (i) true (ii) false  
(c) (i) false (ii) true (d) (i) false (ii) false
5. Which among the following properties is not satisfied by a B tree of order  $m$ ?
- (i) keys are stored in the information nodes.
- (ii) the depth of an information node in a trie always depends on the length of the key.
- (iii) to access an information node containing a key, we need to move down a branch node or a series of branch nodes following the appropriate branch based on the alphabetical characters composing the key.
- (iv) a branch node is merely a collection of pointers to either a branch node or an information node.
- (a) (i) (b) (ii) (c) (iii) (d) (iv)
6. What are the merits of  $m$ -way search trees over AVL search trees?
7. What are the demerits of  $m$ -way search trees?
8. What is the need for B trees?
9. Distinguish between 2-3 trees and 2-4 trees.
10. What is the height of a B tree of order  $m$ ?
11. What is the need for tries?
12. When do tries perform less better than search trees?
13. Insert the following elements in the order given into an empty B tree of order (i) 3 (ii) 4 and (iii) 7  
 $Z R T A D F H Q W C V B S E O P L J K M N U T X$   
 Undertake the following operations on the B trees:  
 (i) Delete Q (ii) Delete A (iii) Delete M (iv) Delete S
14. For the data list shown in Review Question 13 (Chapter 11) construct a 3-way search tree.  
 Insert G and delete J K and Z from the search tree.

15. For the following data list construct a trie:

ANT ANTELOPE BEAR BUG ELEPHANT ZEBRA BEATLE TIGER ANTEATER BISON  
MONKEY ORANGUTANG CHIMPANZEE KOALA KOEL.

Perform the following operations on the trie:

- (i) Delete CHIMPANZEE
- (ii) Delete ANTELOPE
- (iii) Insert RHINOCEROS
- (iv) Insert MONGREL
- (v) Delete ANTEATER



## Programming Assignments

1. Implement a menu driven program to
  - (i) construct an  $m$ -way search tree for a specific order  $m$ ,
  - (ii) insert elements into the  $m$ -way search tree and
  - (iii) delete elements from the  $m$ -way search tree.
2. Implement a menu driven demonstration of all the functions pertaining to insert, delete and search operation of B trees of order  $m$ .
3. Implement a function to delete a key  $K$  from a trie  $T$ . Assume that each of the branch nodes have a COUNT field which records the number of information nodes in the sub trie for which it is the root.
4. Implement functions to traverse a B tree of order  $m$  by inorder, preorder and post order traversals.
5. Execute a function to gather all the information nodes beginning with a specific alphabet from a trie representing alphabetical keys.

## CHAPTER



# RED-BLACK TREES AND SPLAY TREES

# 12

The data structures of Red-Black trees and Splay trees are discussed in this chapter. Red-Black trees which are special forms of binary search trees have their origins in B trees of order 4 but are more efficient than the latter, by way of performance and storage considerations. Splay trees are binary search trees with a self-adjusting mechanism that renders a better performance with regard to what is known as amortized analysis. They are more efficient when compared to their binary search tree or AVL tree counterparts.

12.1 *Red-Black Trees*

12.2 *Splay Trees*

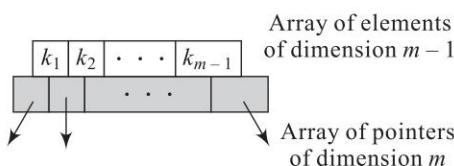
12.3 *Applications*

## Red-Black Trees

12.1

### Introduction to red-black trees

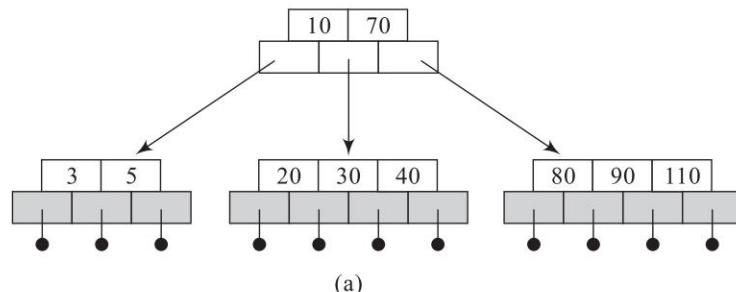
B trees of order  $m$  were discussed in Chapter 11. It was shown how B trees are balanced trees and are efficient for use in applications such as file indexing since they serve to reduce disk accesses. To recall, the node structure of the B trees appears as shown in Fig. 12.1. A simple implementation of the node may call for using sequential data structures such as arrays to hold the keys and the pointers to the child nodes. Thus a B tree of order  $m$  may have each of their nodes to be represented using two arrays of maximum dimension  $m$  and  $m-1$  corresponding to the child pointers and keys respectively. This does entail wastage of space in the worst case. Also to search for or insert a key demands sequentially searching for the element in the nodes, to determine the child pointers of the nodes for the search to move down the tree, before it eventually reaches the key or inserts the key, as the case may be. If the order of the B tree is small, then an effective solution would be to maintain the keys of each of the nodes in the B tree as a binary search tree. However it needs to be ensured that the branches linking the nodes of the binary search tree are distinguished from the same linking the nodes of the B tree.



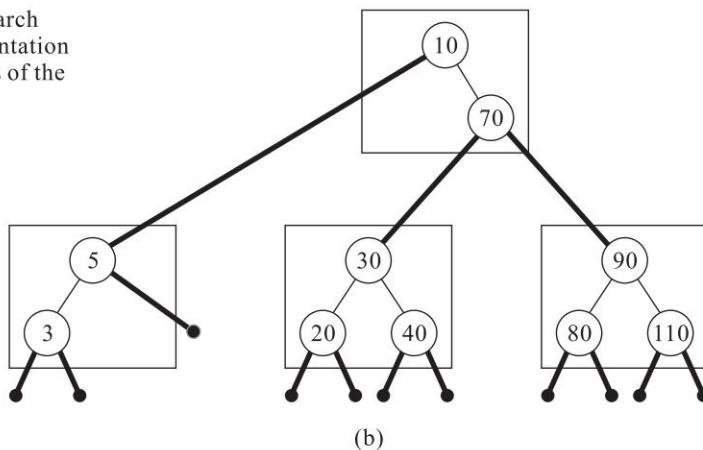
**Fig. 12.1** Node structure of a B tree of order  $m$  and its representation

With regard to B trees of order 4 also referred to as 2-4 trees or 2-3-4 trees, it can be shown that a binary search tree representation of each of its nodes, yields a concept that forms the basis for a special kind of binary search tree called *red-black tree*. Consider the 2-4 tree shown in Fig. 12.2(a), a binary search tree representation of each of the nodes is shown in Fig. 12.2(b). Here, thin lines indicate branches which link the nodes of the binary search tree and thick lines same between the nodes of the original B tree. Now if the thin lines and the nodes hanging from them were shaded light (red) and the thick lines and the nodes hanging from them were shaded grey (black) the resulting tree would be as shown in Fig. 12.2(c). Note that the root node is always shaded black. Such a tree is what is known as the red-black tree.

A 2-4 tree :



A binary search tree representation of the nodes of the 2-4 tree :



Red-black tree :

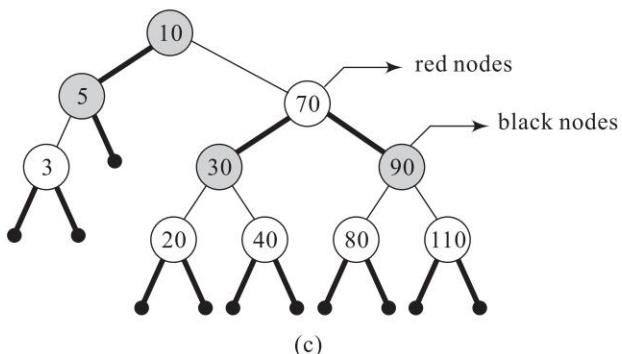
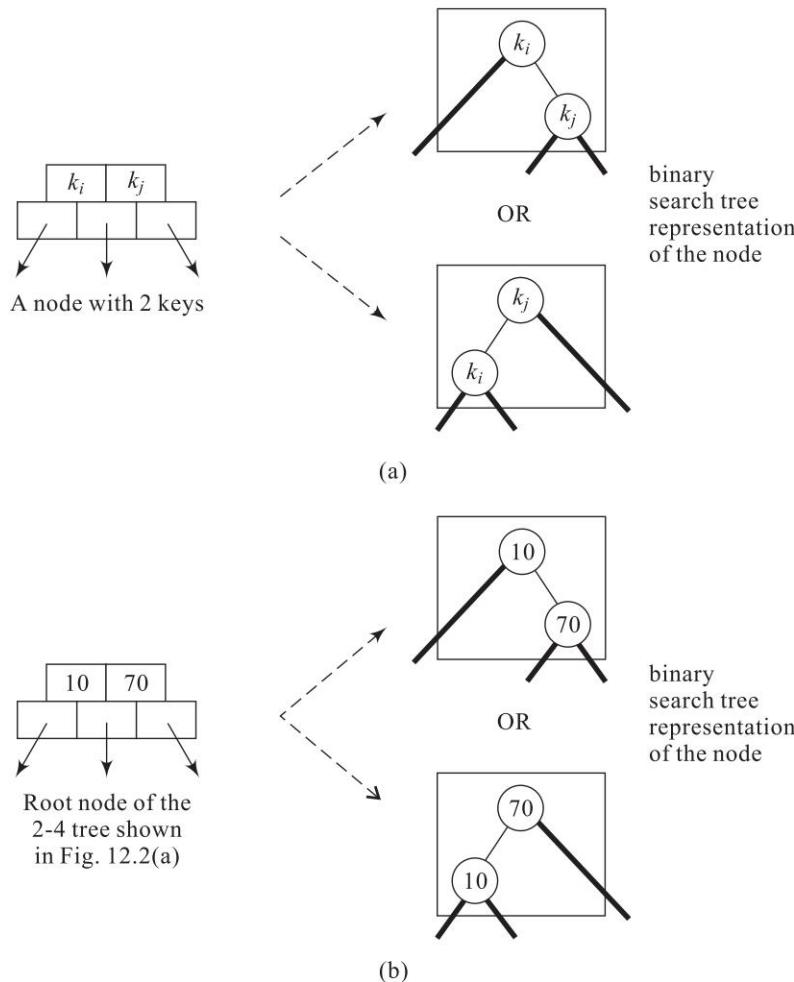


Fig. 12.2 Evolving a red-black tree from a 2-4 tree

A node with two keys in the 2-4 tree may be represented as a binary search tree in either of the two ways as shown in Fig. 12.3(a). For example, the root node of the 2-4 tree [10, 70] may be represented in any one of the two ways as shown in Fig. 12.3(b).



**Fig. 12.3 Possible binary search tree representations of a node with two keys**

## Definition

A *red-black tree* is an extended binary search tree in which the nodes and the edges from which these nodes emanate are either *red* or *black* and satisfy the following properties:

- The root node and the external nodes are always black nodes.
- [*Red Condition*] No two red nodes can occur consecutively on the path from the root node to an external node.
- [*Black Condition*] The number of black nodes on the path from the root node to an external node must be the same for all external nodes.

Since the colour of the node is same as the colour of the edge from which the node emanates, the Red and Black Conditions may be expressed in terms of the edges as well. The Red Condition could be alternatively defined as no two red pointers or edges can occur consecutively on a path from the root node to an external node. The Black Condition could be redefined as all paths from the root node to external nodes must have the same number of black pointers. Besides the above mentioned conditions, all pointers linking internal nodes with the external nodes must be black.

The number of black nodes or edges on the path from a node to an external node is called the *rank* of the node. The rank of all external nodes is 0.

In this chapter, all red nodes and edges will be represented using empty circles and thin lines and all black nodes and edges will be represented using shaded circles and thick lines respectively.

**Example 12.1** Figure 12.4 illustrates a red-black tree. Observe the tree to be an extended binary search tree with its root and external nodes to be black. The Red Condition where no two consecutive red nodes can occur is satisfied on all the paths from the root to the external nodes. Also the Black Condition where all root-to-external node paths must contain the same number of black nodes is also true. Every such path in the given tree contains exactly 2 black nodes.

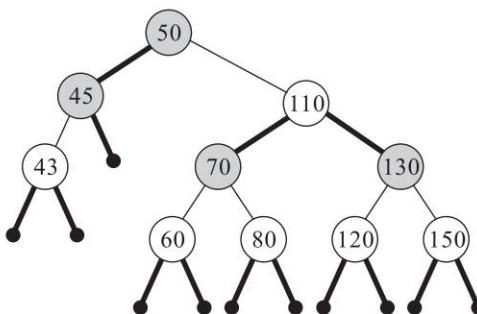


Fig. 12.4 An example red-black tree

## Representation of a red-black tree

Since a red-black tree is an extended binary search tree, the kind of node representation used for a binary search tree may be employed for the tree as well. However since the colour of a node plays a dominant role in the definition of the red-black tree it is essential that the colour is also recorded in the node structure as a field (COLOUR). Another scheme could be to record the colour of the two pointers emanating from the node.

The insert/delete operations quite often unbalance a red-black tree. The rebalancing of the tree may call for moving up and down the tree during the node adjustments. To facilitate this upward movement the node structure of a red-black tree may have a provision for a PARENT field. PARENT fields of nodes are pointers to their respective parent nodes.

## Searching a red-black tree

Searching a red-black tree for a key is in no way different from the procedure used to search for a key in a binary search tree. Algorithm 10.1 discussed in Chapter 10 and which retrieves a key ITEM from a binary search tree can be employed for the same.

## Inserting into a red-black tree

Inserting a key  $K$  into a red-black tree follows a procedure exactly similar to the one employed for binary search trees. The only concern now is to determine the colour to which the node must be set to. If the node is set to black, then the path from the root node to the external node passing through the node would have one more black node. This results in the violation of the Black Condition of a red-black tree. Hence the other alternative is to set the node to red. Now, if doing so leads to the violation of the Red Condition, then the red-black tree is said to be unbalanced. To set right the imbalance we need to undertake rotations.

Let us suppose  $u$  is the newly inserted red node and  $parent\_u$  its consecutive red node which is also the parent of node  $u$ . Now,  $u$  must have a grand parent,  $grandparent\_u$  which is a black node. Based on the position of node  $u$  in relation to  $parent\_u$  and  $grandparent\_u$ , and the colour of the other child of  $grandparent\_u$ , the imbalances are classified as  $LLb$ ,  $LLr$ ,  $RLb$ ,  $RLr$ ,  $LRb$ ,  $LRr$ ,  $RRb$  and  $RRr$ . Thus if  $u$  is inserted as the Left child of  $parent\_u(L)$  which in turn is the Left child of  $grandparent\_u$  ( $L$ ) and the other child of  $grandparent\_u$  is black ( $b$ ) -the child may in fact be an external node which is black- then the rotation undertaken is  $LLb$ . Again if  $u$  were to be inserted as the Right child of  $parent\_u$  ( $R$ ) which in turn is the Left child of  $grandparent\_u$  ( $L$ ) whose other child is red ( $r$ ) then the rotation to be undertaken is  $LRr$  and so on.

Imbalances of the type  $LLr$ ,  $RLr$ ,  $LRr$ , and  $RRr$  with ' $r$ ' as its suffix only call for a colour change of the nodes to set right the imbalance. On the other hand imbalances of the type  $LLb$ ,  $LRb$ ,  $RRb$  and  $RLb$ , with ' $b$ ' as its suffix, call for rotations to set right the imbalance.

### LLr, LRr, RRr, and RLr imbalances

Figure 12.5 illustrates a generic representation of the  $LLr$ ,  $LRr$ ,  $RRr$ , and  $RLr$  imbalances and the colour changes that need to be undertaken to set right the imbalance. The notations **L**, **R** and **r** inscribed on the edges of the red-black trees illustrate the classification of the imbalance.

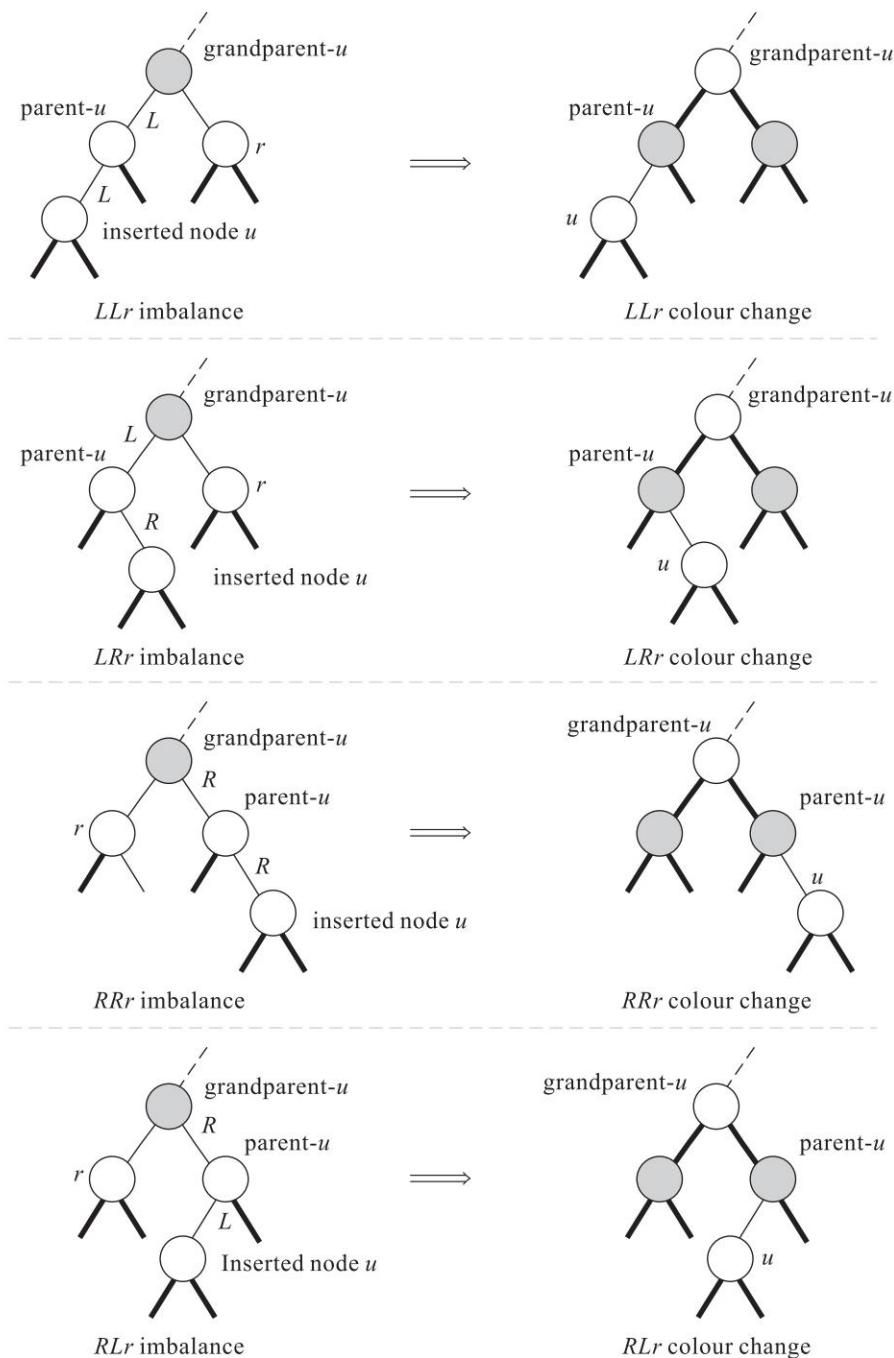
**Example 12.2** Consider the red-black tree shown in Fig. 12.6 (a), the insertion of 60 results in an  $LLr$  imbalance and the tree after colour change is shown in Fig. 12.6(b). For the red-black tree shown in Fig. 12.7(a), inserting 184 yields an  $RRr$  imbalance. The balanced tree after colour change is shown in Fig. 12.7(b).

Note how the colour of the  $grandparent\_u$  node changes from black to red in the generic representations shown in Fig. 12.5. This is so provided the  $grandparent\_u$  is not the root. If  $grandparent\_u$  turns out to be the root, then no colour change on it is done. This would therefore increase the number of black nodes on all paths from the root ( $grandparent\_u$ ) to the external nodes by 1.

Also, if changing the colour of  $grandparent\_u$  to red causes further imbalance up the tree, then we identify the category of imbalance treating  $grandparent\_u$  as  $u$  and so on until the whole tree gets rebalanced after undertaking the appropriate rotations or colour change.

### LLb, LRb, RRb, and RLb imbalances

Figure 12.8 illustrates the generic representations of the  $LLb$ ,  $LRb$ ,  $RRb$ , and  $RLb$  imbalances and the respective rotations to rebalance the red-black tree. The notations **L**, **R** and **b** inscribed on the edges of the red-black trees illustrate the classification of the imbalance.



**Fig. 12.5** Generic representations of LLr, LRr, RRr, and RLr imbalances and their colour change

Insert 60

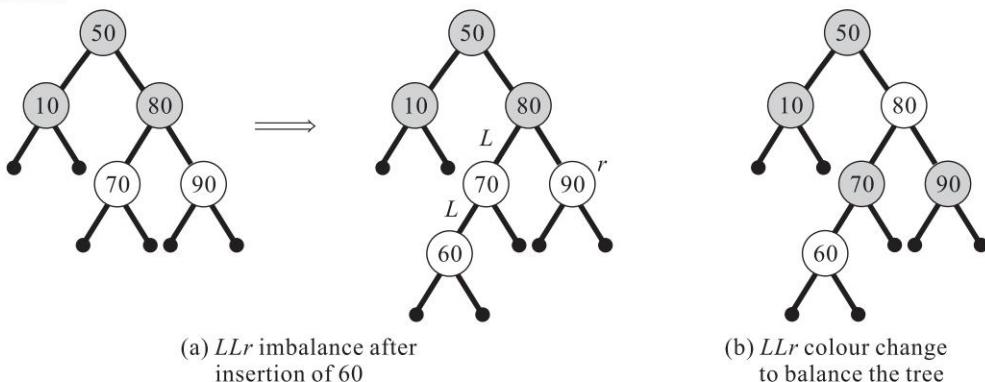


Fig. 12.6 An example LLr imbalance and its colour change

Insert 184 :

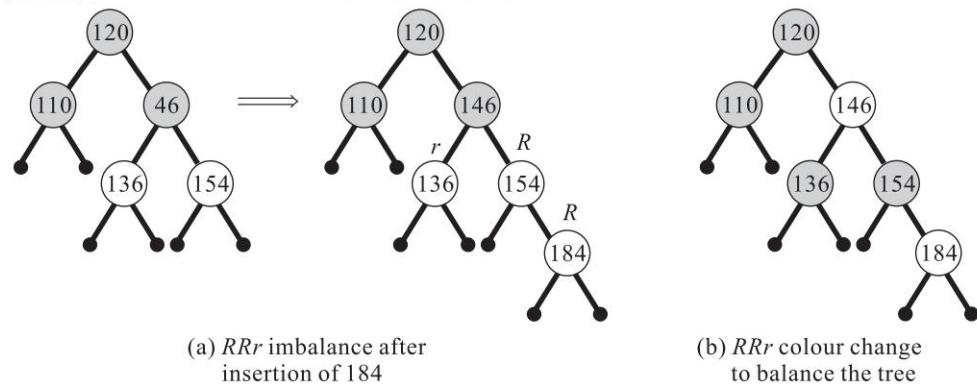
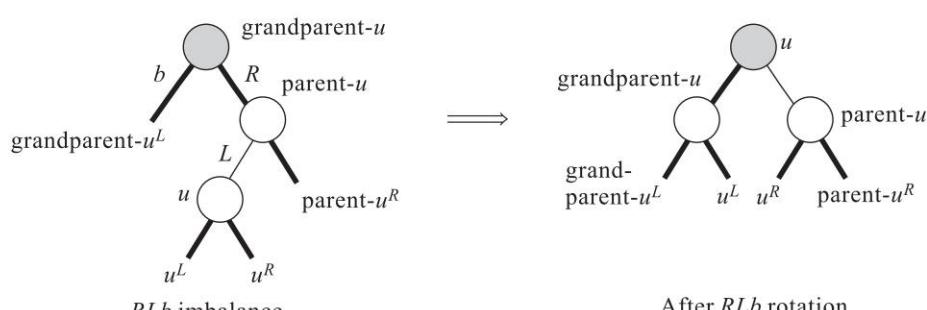
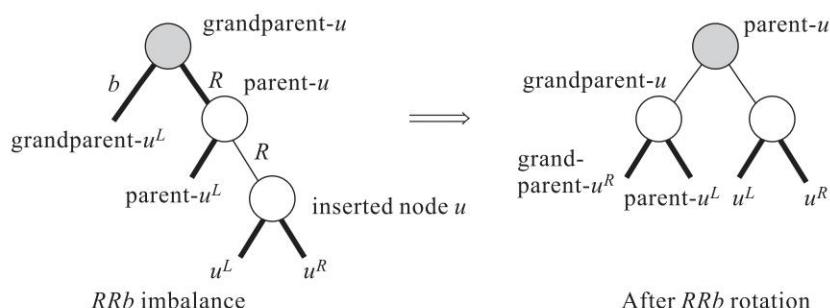
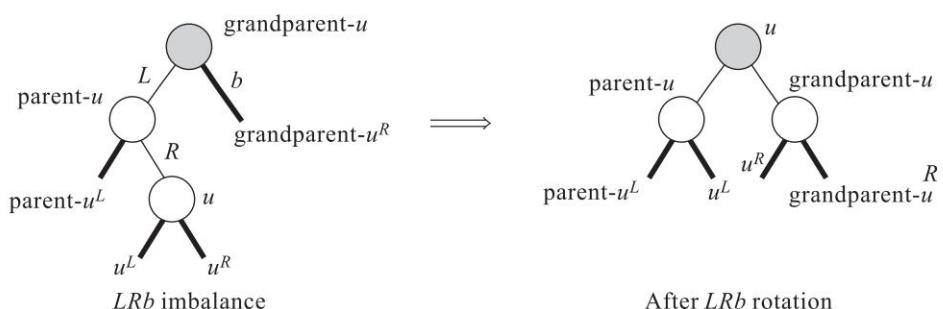
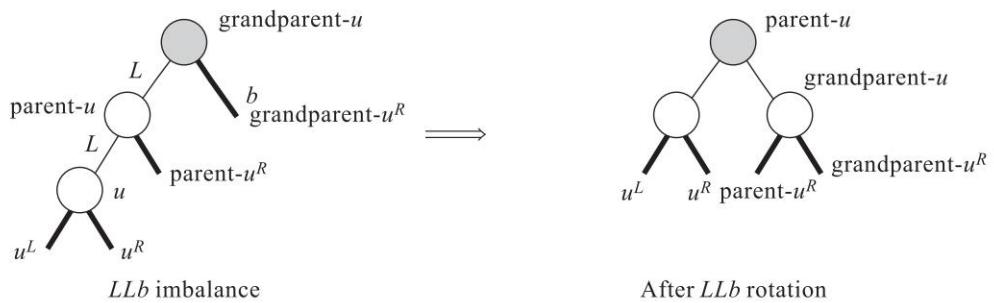
RRr imbalance  
after insertion of 184

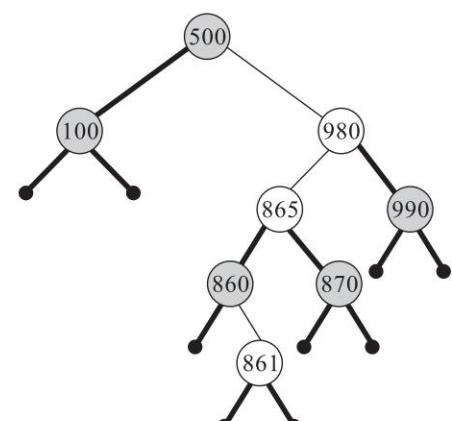
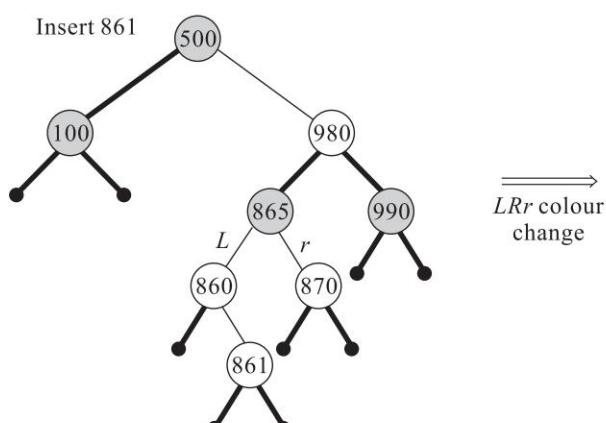
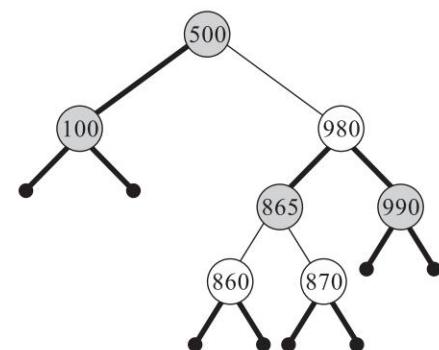
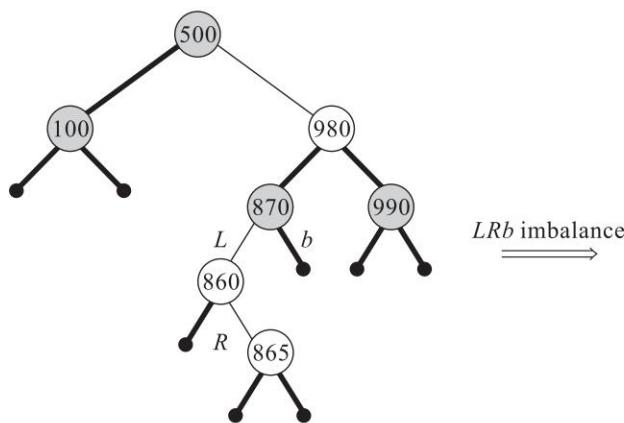
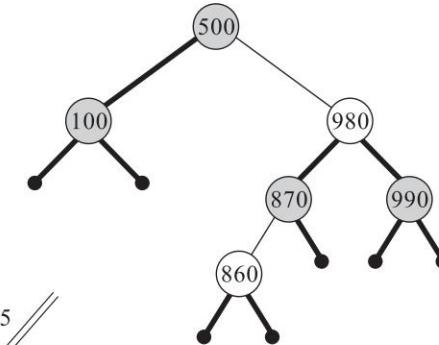
Fig. 12.7 An example RRr imbalance and its colour change

Here  $u$  is the node which is inserted into the tree as the left or right child of  $parent\_u$  which is the left or right child of  $grandparent\_u$ .  $u^L$ ,  $parent\_u^L$  and  $grandparent\_u^L$  indicate the left subtrees of  $u$ ,  $parent\_u$  and  $grandparent\_u$  respectively.  $u^R$ ,  $parent\_u^R$  and  $grandparent\_u^R$  indicate the right subtrees of  $u$ ,  $parent\_u$  and  $grandparent\_u$  respectively. It may be observed that the  $LLb$ ,  $LRb$ ,  $RRb$  and  $RLb$  rotations resemble the  $LL$ ,  $LR$ ,  $RR$  and  $RL$  rotations discussed earlier apart from the colour changes that are called for after the rotation.

**Example 12.3** Consider the red-black tree shown in Fig. 12.9(a). Let us insert 865 and 861 into the tree in the order given. Figure 12.9(b) shows the  $LRb$  imbalance of the tree after the insertion of 865. The  $LRb$  rotation rebalancing the tree is shown in Fig. 12.9(c). Insertion of 861 into the red-black tree shown in Fig. 12.9(c) yields an  $LRr$  imbalance (Fig. 12.9(d)) which is set right by a colour change shown in Fig. 12.9(e). But lo! this triggers a further imbalance of the type  $RLb$  with the nodes 865 and 980 turning out to be consecutive red nodes! Figure 12.9(f) shows the  $RLb$  imbalance in the tree. The  $RLb$  rotation rebalancing the tree is shown in Fig. 12.9(g).



**Fig. 12.8** Generic representations of LLb, LRb, RRb and RLb imbalances and their rotations



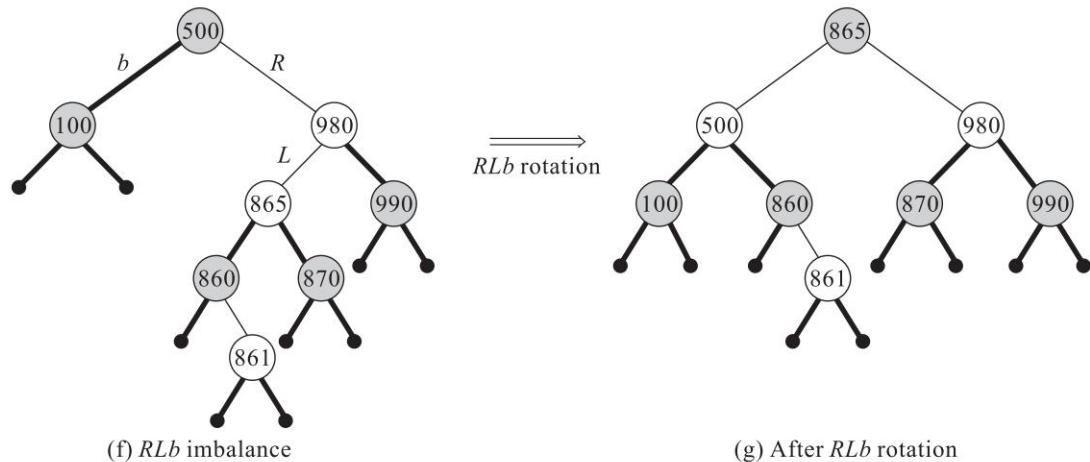
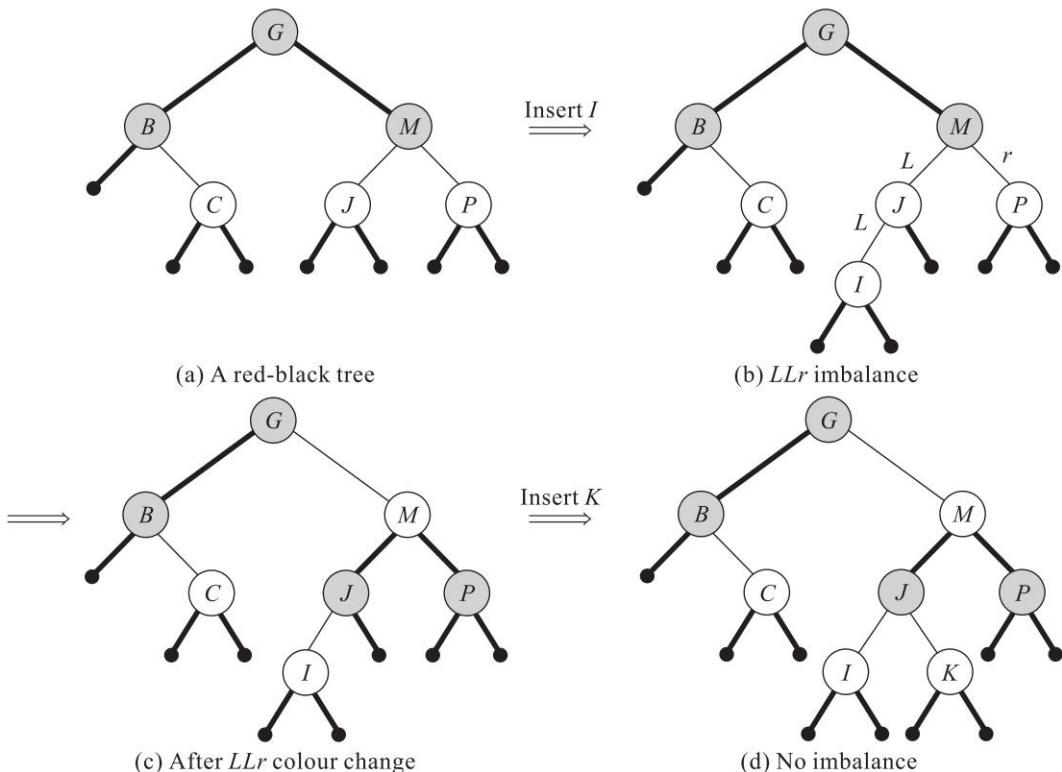


Fig. 12.9 An example LRb, RLb imbalance

**Example 12.4** In the red-black tree shown in Fig. 12.10(a), let us insert  $I$ ,  $K$ ,  $L$  in the order given. Insertion of  $I$  into the red-black tree results in an  $LLr$  imbalance which only calls for a colour change to rebalance the tree. Figure 12.10(b) shows the rebalanced tree after  $LLr$  colour change. Insertion of  $K$  does not unbalance the tree (Fig. 12.10(c)). However, insertion of  $L$  results in an  $RRr$  imbalance the rebalancing of which triggers an  $RLb$  imbalance. Figure 12.10(d) and Fig. 12.10(e) illustrate the  $RRr$  colour change and  $RLb$  rotation respectively.



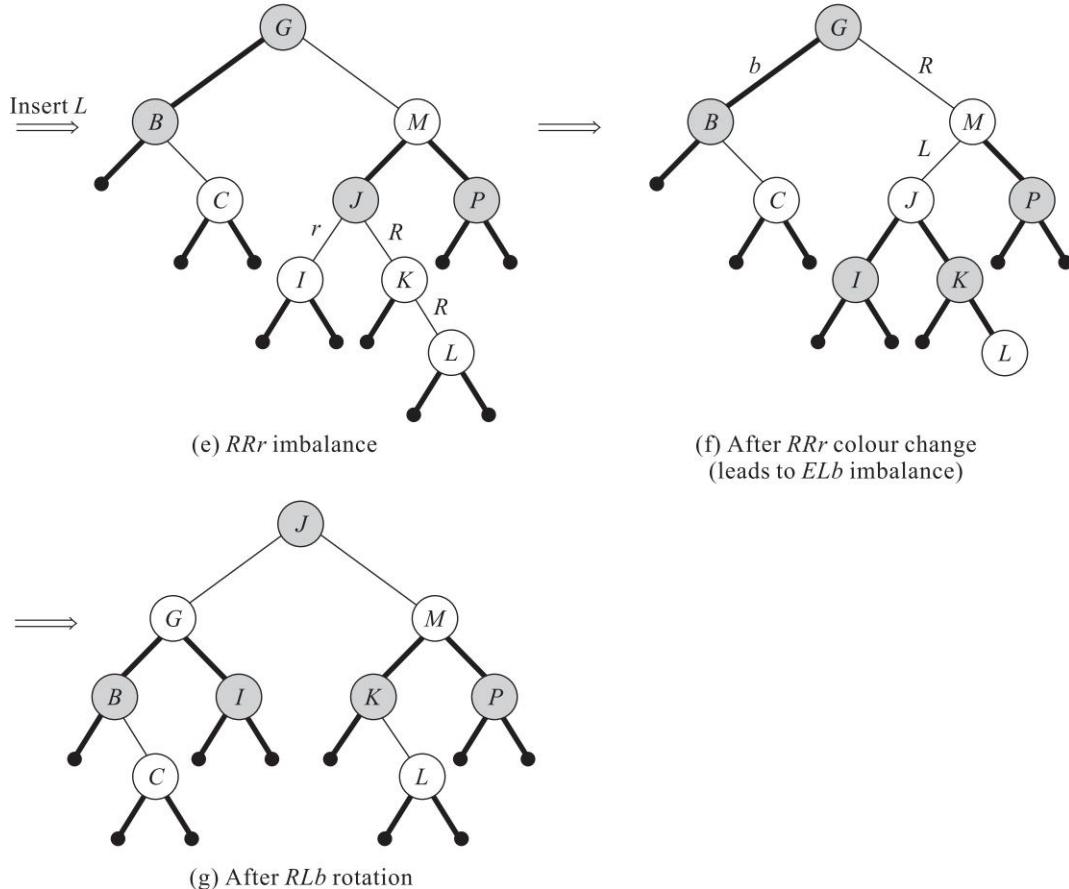


Fig. 12.10 Inserting into a red-black tree (Example 12.1)

## Deleting from a red-black tree

Deleting a key  $K$  from a red-black tree proceeds as one would to delete the same from a binary search tree. In this regard, the cases discussed in Chapter 10 in connection with the deletion of key  $K$  from a binary search tree, when  $K$  is a leaf node or  $K$  has a lone subtree (left subtree or right subtree only) or  $K$  has both left subtree and right subtree hold good here as well. However, if the deletion results in an imbalance in the tree then this may call for a colour change or a rotation if necessary.

If the deleted node were red, then there is no way that the Black Condition would be violated and hence no imbalance is possible. On the other hand if the deleted node were to be black then there is every possibility of violation of the Black Condition due to the shortage of a black node in a specific root-to-external node path. In such a case the tree is said to be *unbalanced*.

The imbalance is classified as *Left (L)* or *Right (R)* based on whether the deleted node  $v$ , occurs to the right or left of its parent node, *parent\_v*. Again if the sibling of node  $v$ , *sibling\_v* is a black node then the imbalance is further classified as *Lb* or *Rb*. If *sibling\_v* is a red node, then the imbalance is classified as *Lr* or *Rr*. Based on whether *sibling\_v* has 0 or 1 or 2 red children the *Lb*,

*Rb* imbalances are further sub classified as *Lb0*, *Lb1* and *Lb2*, and *Rb0*, *Rb1* and *Rb2* respectively. Similarly, the *Lr*, *Rr* imbalances are also sub classified as *Lr0*, *Lr1* and *Lr2*, and *Rr0*, *Rr1* and *Rr2* respectively. During rebalancing, *v* denotes the node that was deleted but physically replaced by another node which takes its place as called for by the delete process.

We deal with imbalances concerning *R* in the next section. Those pertaining to *L* have been demonstrated as Illustrative Problems 12.3 and 12.4. Nodes superscripted with *L* indicate their left subtrees and those with *R* indicate their right subtrees. The nodes shaded grey emanating from thick lined edges indicate black nodes and those shaded white emanating from thin lined edges indicate red nodes. Nodes that are hatched indicate either red or black nodes.

## Rb0, Rb1 and Rb2 imbalances

Figure 12.11 illustrates the generic representations of *Rb0*, *Rb1* and *Rb2* imbalances. The notations **R**, **b** and **0/1/2** inscribed on the edges of the red-black trees illustrate the classification of the imbalance.

In the case of *Rb0* imbalance the rebalancing only calls for a colour change of nodes. The two possibilities of *Rb0* imbalance are shown in the figure. *Rb1* imbalance is of two types indicated as *Rb1*(type 1) and *Rb2* (type 2). In these, the node *sibling\_v* has a single red child in either *sibling\_v<sup>L</sup>* or *w* respectively. The *Rb2* imbalance has *sibling\_v* holding two red children in *sibling\_v<sup>L</sup>* and *w*. Rotations as illustrated in the figure are performed for the *Rb1* and *Rb2* imbalances.

## Rr0, Rr1 and Rr2 imbalances

Figure 12.12 illustrates the generic representations of *Rr0*, *Rr1* and *Rr2* imbalances. The notations **R**, **r** and **0/1/2** inscribed on the edges of the red-black tree illustrate the classification of imbalance. Rotations are undertaken in all the three cases to rebalance the trees. *Rr1* imbalance is of two types indicated as *Rr1*(type 1) and *Rr1*(type 2).

**Example 12.5** A series of red-black trees, the *Rb0*, *Rb1* and *Rb2* imbalances and their rebalancing rotations are shown in Fig. 12.13. The **R**, **b**, **0/1/2** notations are inscribed on the tree to help classify the kind of imbalance.

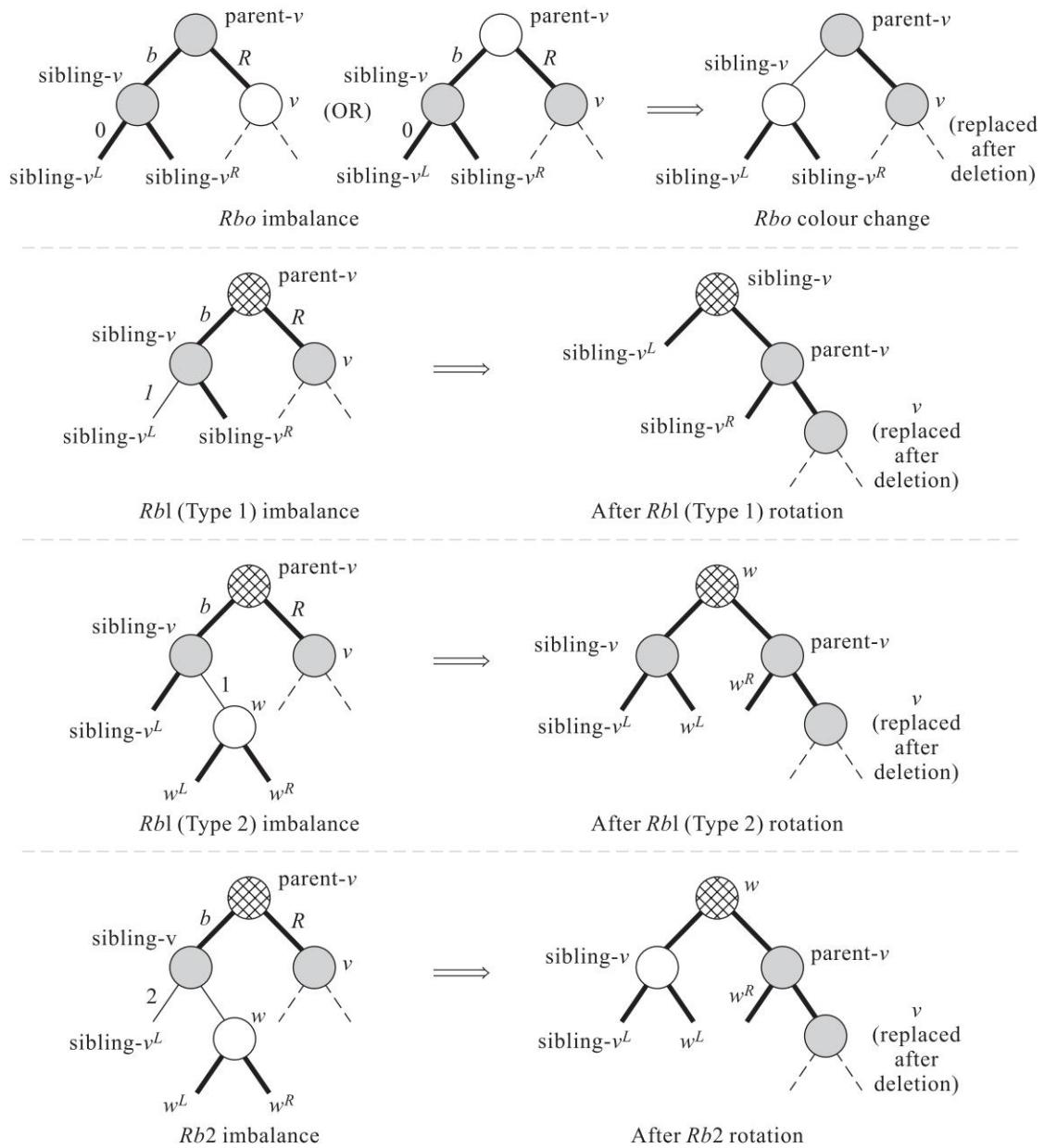
Deleting 36 from the red-black tree shown in Fig. 12.13(a) leaves the tree violating the Black Condition. The imbalance is classified as *Rb0*. The rebalancing calls for a mere colour change with 28 set as a red node.

Deleting 32 (Fig. 12.13(b)) leads to *Rb1* (type 1) imbalance with a violation of the Black Condition. The rebalancing rotation pushes 28 up as a red node and changes 26 and 30 to black nodes thereby ensuring the satisfaction of both Red and Black Conditions.

Deleting 59 (Fig. 12.13 (c)) is a case of *Rb1*(type 2) imbalance. During the rebalancing rotation, 48 moves up to become the root of the subtree. The Red and Black Conditions are satisfied after rotation.

Deleting 99 (Fig. 12.13(d)) is a case of *Rb2* imbalance and the rebalancing pushes 78 up as the root of the subtree. The Red and Black Conditions hold good after rebalancing.

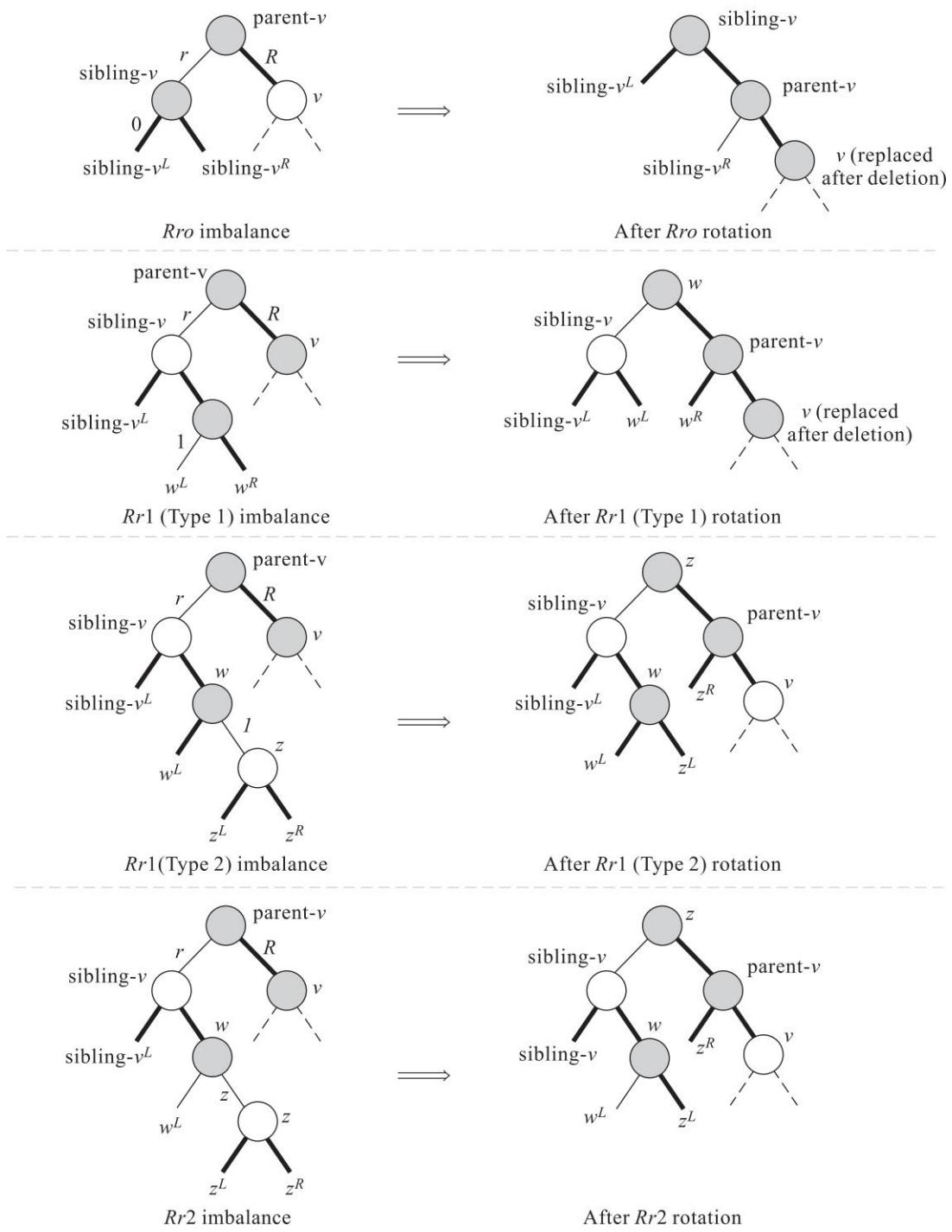
**Example 12.6** Figure 12.14 shows a series of red-black trees and the respective deletions which trigger the *Rr0*, *Rr1* and *Rr2* imbalances. The rotations illustrated are self explanatory.



**Fig. 12.11** Generic representations of Rb0, Rb1 and Rb2 imbalances and their rebalancing mechanisms

**Example 12.7** Consider the red-black tree shown in Fig. 12.15(a). Let us delete the following keys from the tree:

$$F, S, O, L, K.$$



**Fig. 12.12** Generic representations of Rr0, Rr1 and Rr2 imbalances and their rotations

## Red-Black Trees and Splay Trees

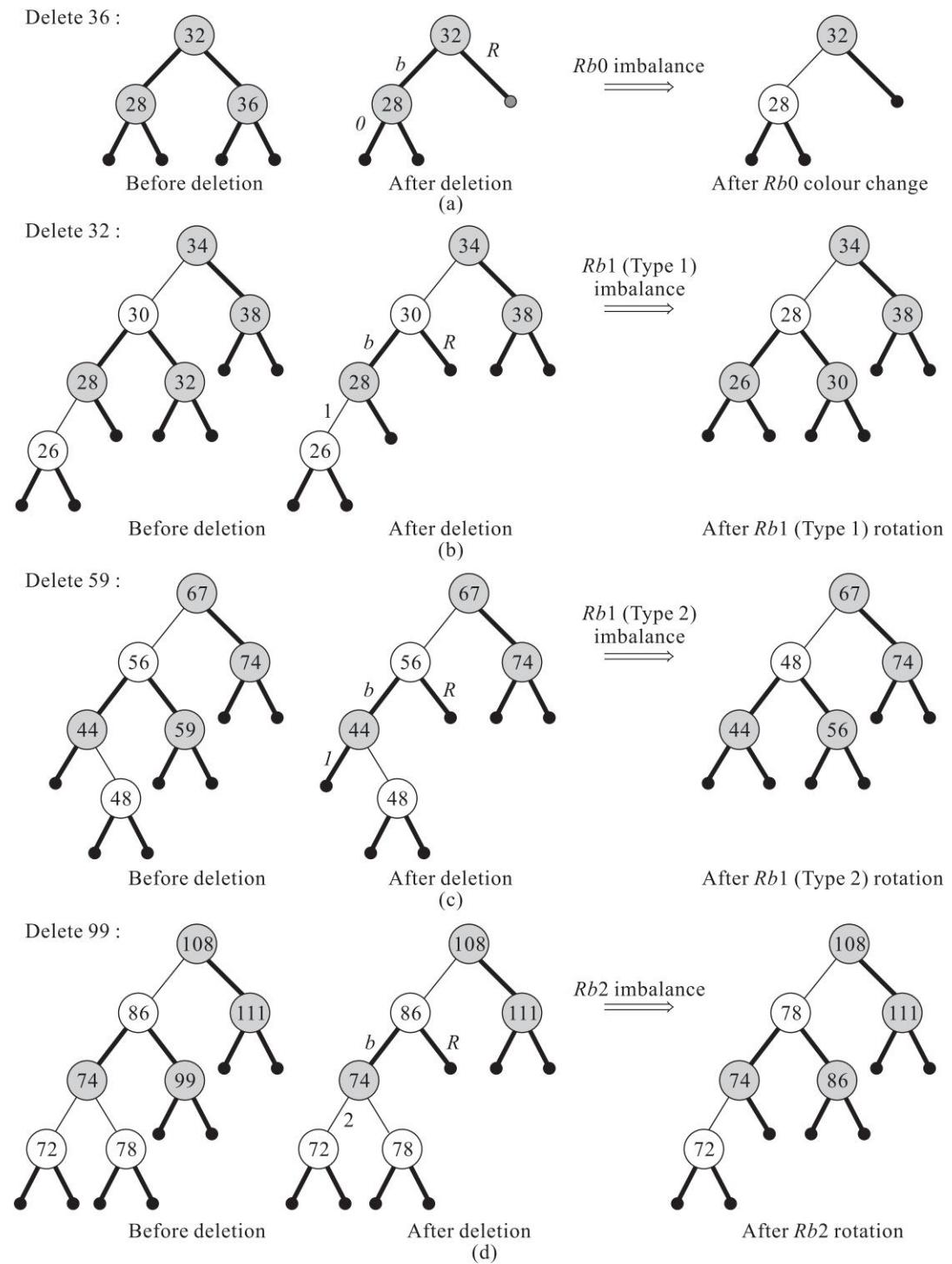
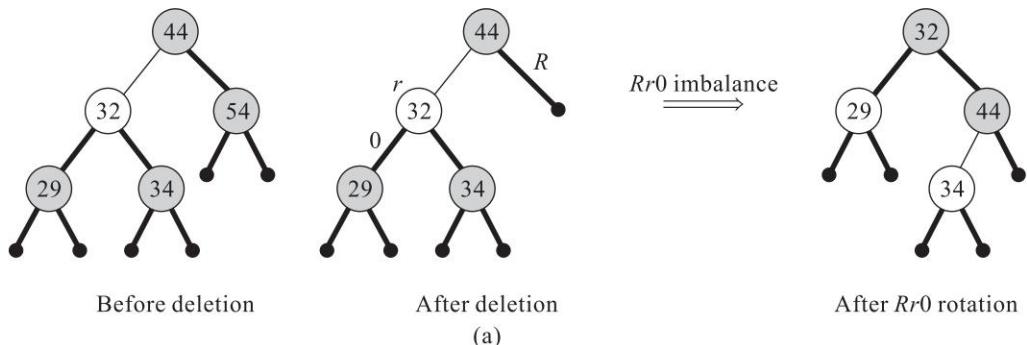
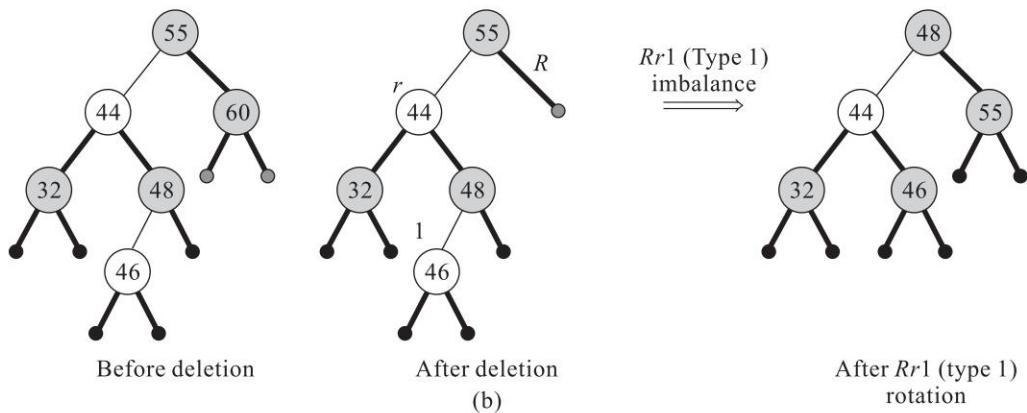


Fig. 12.13 Example Rb0, Rb1 and Rb2 imbalances

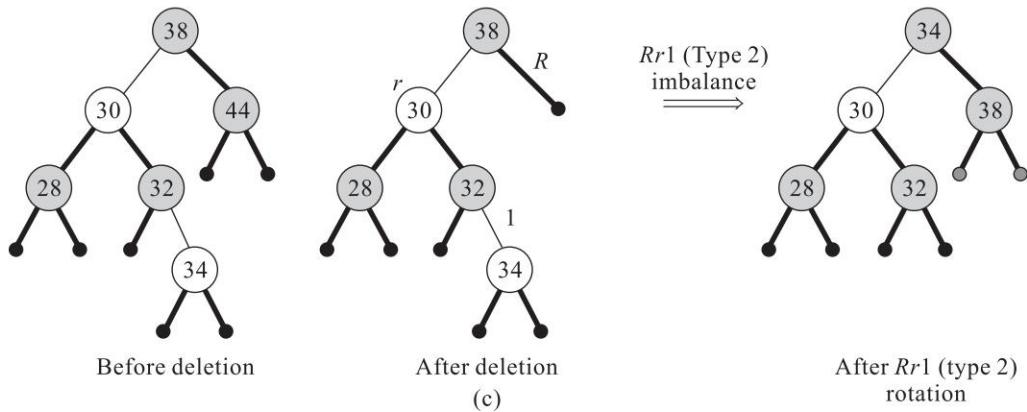
Delete 54 :



Delete 60 :



Delete 44 :



## Red-Black Trees and Splay Trees

Delete 41 :

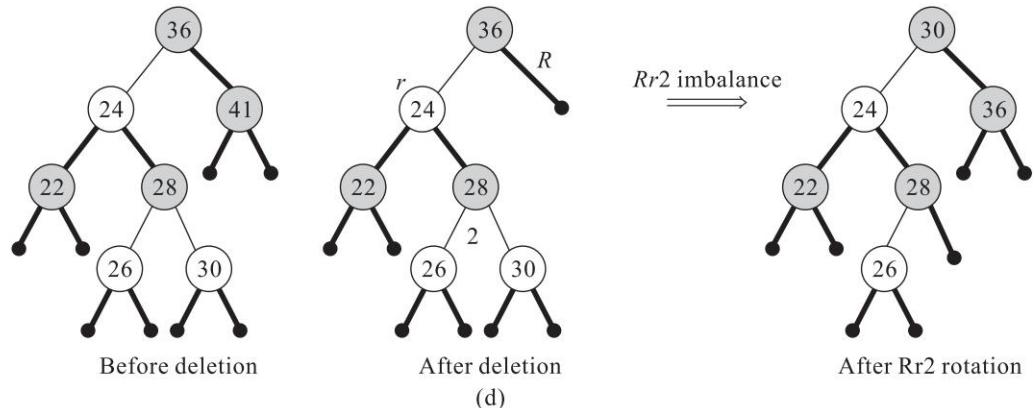


Fig. 12.14 Example Rr0, Rr1 and Rr2 rotations

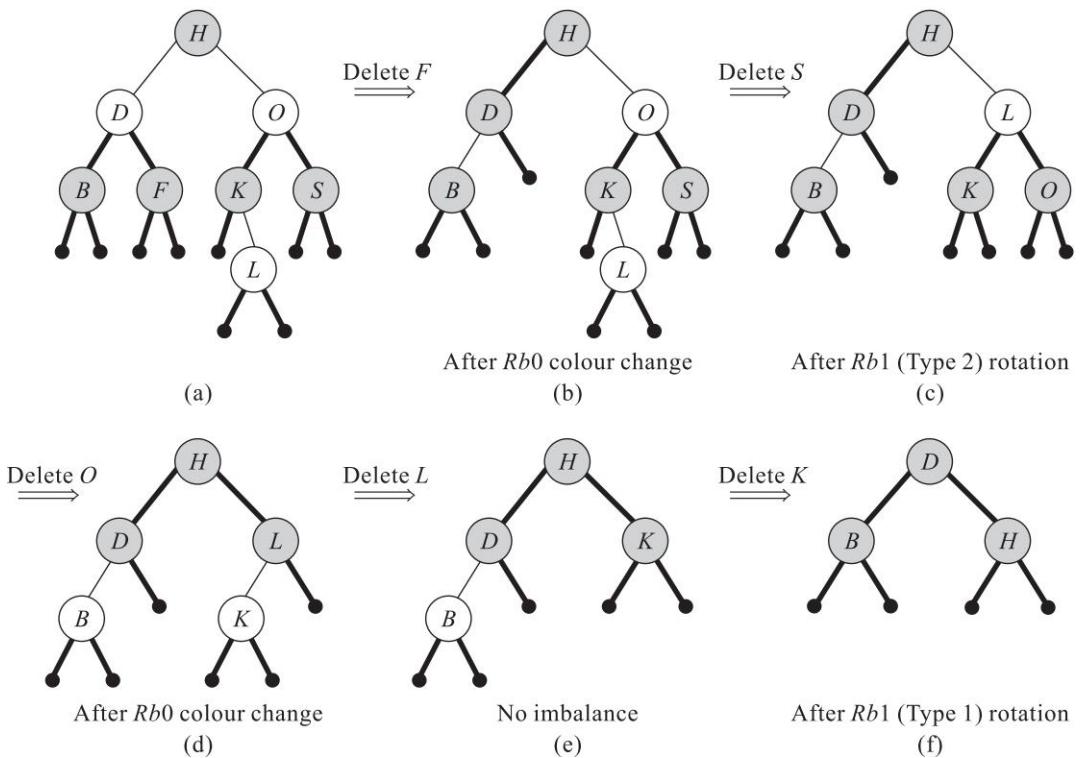


Fig. 12.15 Deletion operations on a red-black tree (Example 12.7)

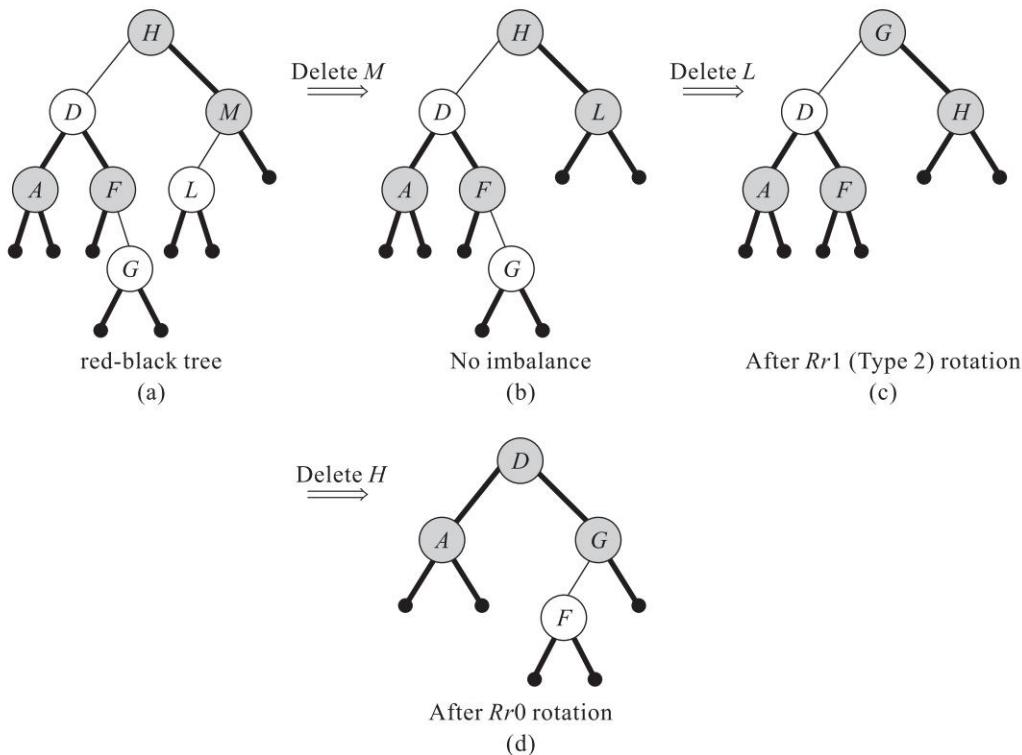
All deletions except that of  $L$  trigger a series of  $Rb0/1/2$  kind of imbalance. The snapshots of the tree after the respective rotations that were undertaken to rebalance the tree, are shown in Figs 12.15(b-f). Note that the deletion of  $L$  only triggers the deletion of a red node, after  $L$  has

been replaced by K. Therefore this deletion does not lead to an imbalance in the tree due to the non violation of the Black Condition.

**Example 12.8** Consider the red-black tree shown in Fig. 12.16(a). Let us delete the following keys from the tree:

$M, L, H$

While deletion of  $M$  does not result in an imbalance since it only causes the deletion of a red node, the rest of the deletions call for an  $Rr0/1/2$  kind of imbalance. Figures 12.16(b-d) illustrate the snapshots of the trees after the appropriate rotations to rebalance the tree have been performed.



**Fig. 12.16** Deletion operations on a red-black tree (Example 12.8)

### Time complexity of search, insert and delete operations on a red-black tree

Since the search operation on a red-black tree is similar to that on a binary search tree, the time complexity of the operation is  $O(\log n)$ . In the case of insertion or deletion, the operation may call for a colour change that can, in the worst case, propagate up to the root and also may call for a rotation to rebalance the tree. Though the colour change and rotation needs only a constant time ( $O(1)$ ), the overall time for an insert/delete operation in the worst case would be  $O(\log n)$ . It can be shown that the height of a red-black tree is at most  $2\log_2(n+1)$  and therefore all search, insert and delete operations that need  $O(h)$  time would have a time complexity of  $O(\log n)$ .

## Splay Trees

## 12.2

### Introduction to splay trees

In the case of binary search trees (Sec. 10.2) it was observed that the worst case time complexity of the tree is  $O(n)$ . Assume a case where a group of records is stored as a binary search tree. Now if a record were to be repeatedly accessed ( $m$  times), then the time complexity of the operation sequence would be  $O(m.n)$  in the worst case. In fact, studies have shown that an information or a node that is accessed once is likely to get accessed more often than not. What if there were to be a data structure, which, once a node is accessed, radically changes its shape to push the accessed node as the root? This adjustment though expensive the first time an access for a node is made, can make the repeated accesses to the node cheaper. Also, during the process of adjustment, where the nodes are moved around to make room for the new node, the other nodes which are deep down may move up making their accesses relatively cheaper as well.

Splay trees are such data structures which provide this mechanism. These are binary search trees with a self adjusting mechanism which renders them remarkably efficient over a sequence of accesses. Nodes which are frequently accessed are moved towards the root thereby rendering further retrievals of the same to be efficient. Thus every time a node is accessed either for search or insertion, the newly accessed node is pushed towards the root. This would dislodge the other nodes to a position away from the root and in course of time would have the inactive nodes moving farther and farther away from the root.

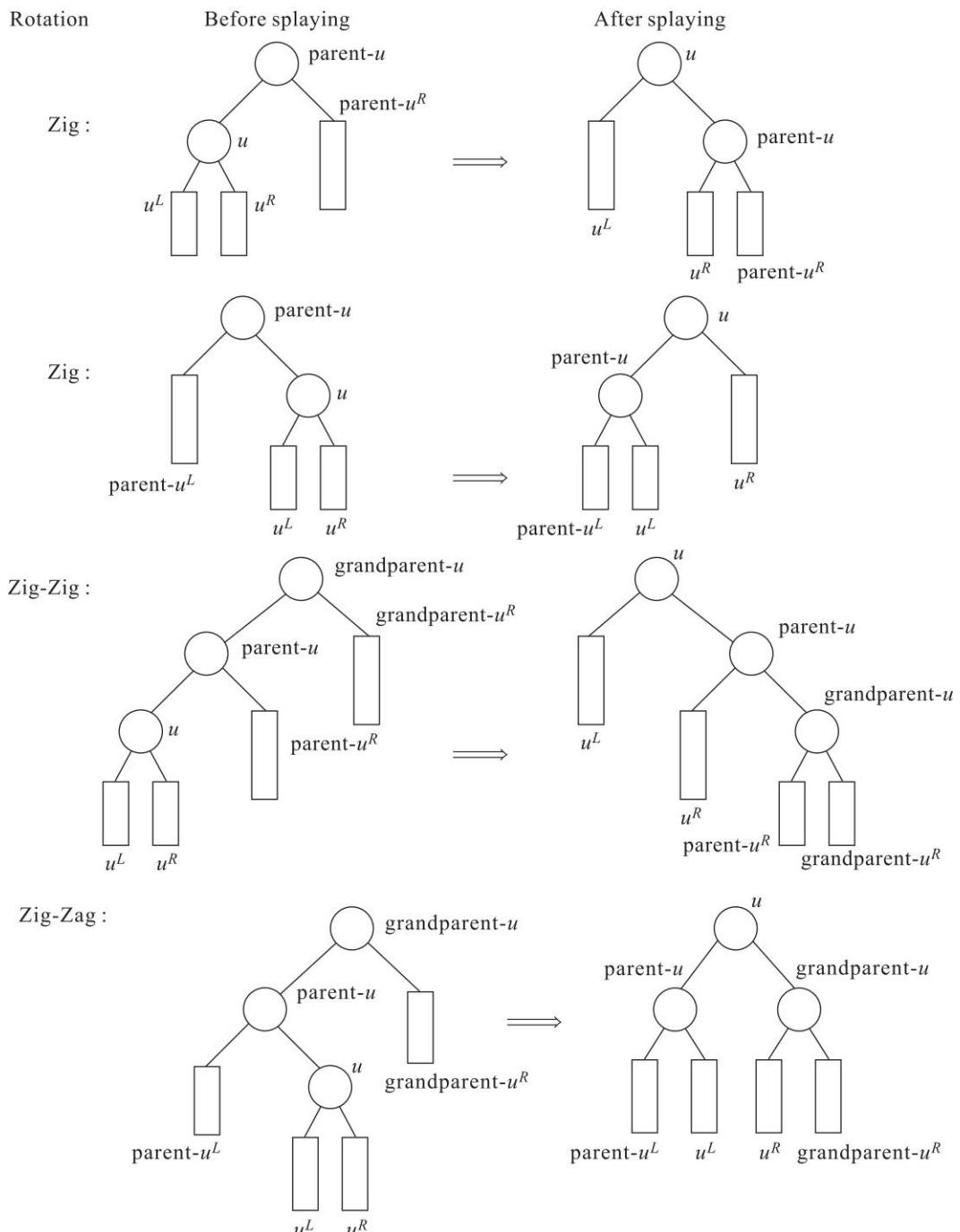
Unlike AVL search trees which are always height balanced, there is no guarantee that the splay tree would remain balanced always. In fact if the splay tree turns out to be unbalanced, then an access may turn out to be fairly expensive. However, over a long sequence of accesses, splay trees may prove to be even cheaper than AVL trees by way of the number of operations. Such an analysis which spreads over a sequence of operations and in which the expensive operations are averaged over the less expensive ones is what is called as *amortized analysis*. If the time complexity of a single access turns out to be  $O(n)$ , then the amortized analysis of the access in a splay tree for a sequence of  $m$  operations is  $O(m. \log n)$ .

### Splay rotations

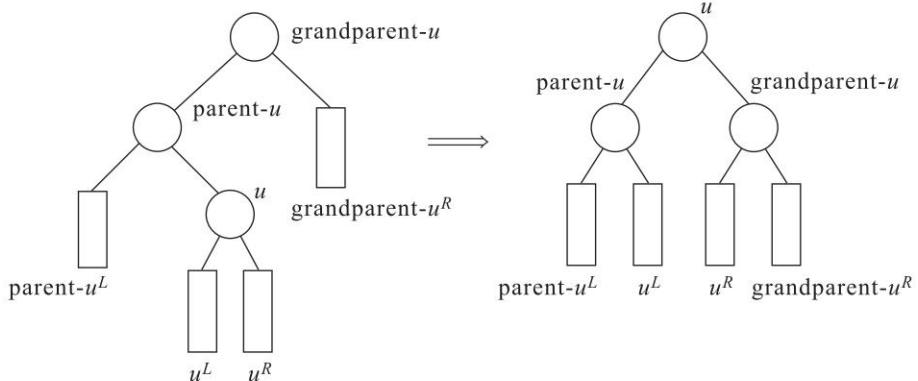
An insert or search operation on a splay tree proceeds as one would on a binary search tree. However, after the operation is over, the tree is splayed with regard to the specific node. This would mean pushing the node upwards towards the root methodically following what are known as *splay rotations*. Splay rotations are more or less similar to AVL tree rotations (Sec. 10.3.3) and proceed bottom up from the node towards the root.

The splay rotations are performed with regard to the specific node  $u$ , its parent  $parent\_u$  and grandparent  $grandparent\_u$ , until perhaps the root node becomes the parent of  $u$ . At that stage, which is the last step, the rotation involves only  $u$  and the root node. The aim of splaying is to move the accessed node  $u$  up by two levels at every step. To do this we track the path from the root to the accessed node  $u$ . Every time the path turns left we term it *zig* and every time it turns right we term it *zag*. Thus in the case of a single step down the tree, the path could be either a *zig* or *zag*. If two steps were to be considered, the path could be any one of *zig-zig*, *zig-zag*, *zag-zig* or *zag-zag*. Since the splaying proceeds bottom up, if the length of the path from the root to the accessed node  $u$  is even, then the rotations appropriate to the two step series viz., *zig-zig*, *zig-zag*, *zag-zig* or *zag-zag* are undertaken. On the other hand if the length of the path from the root

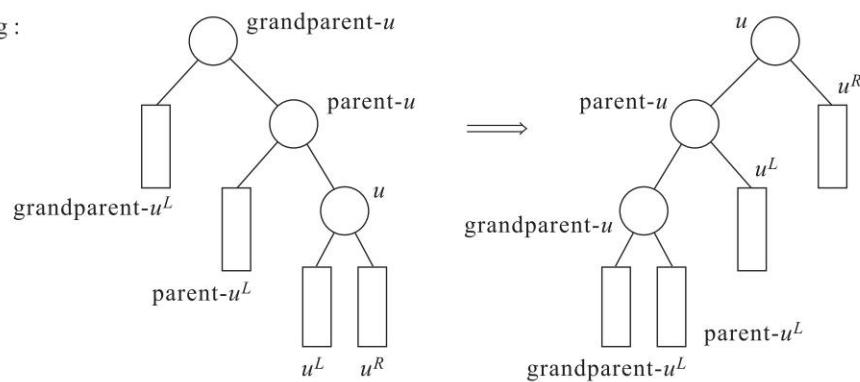
to the accessed node  $u$  is odd, then the final rotation may turn out to be the one corresponding to a single step series, either a *zig* or a *zag*. The rotations corresponding to the single step and two step series are shown in Fig. 12.17.



Zig-Zag :



Zag-Zag :



Zag-Zig :

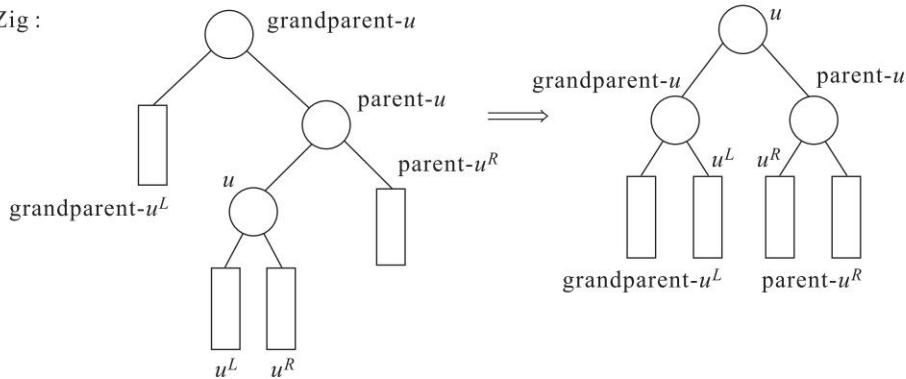


Fig. 12.17 Splay rotations

It can be seen that while zig and zag represent single rotations corresponding to those of AVL trees, zig-zag and zag-zig represent double rotations corresponding to the same. However, zig-zig and zag-zag are not the same as performing two single rotations. Figure 12.18 demonstrates the incorrect implementation of a zig-zig with two single rotations.

Zig-Zig :

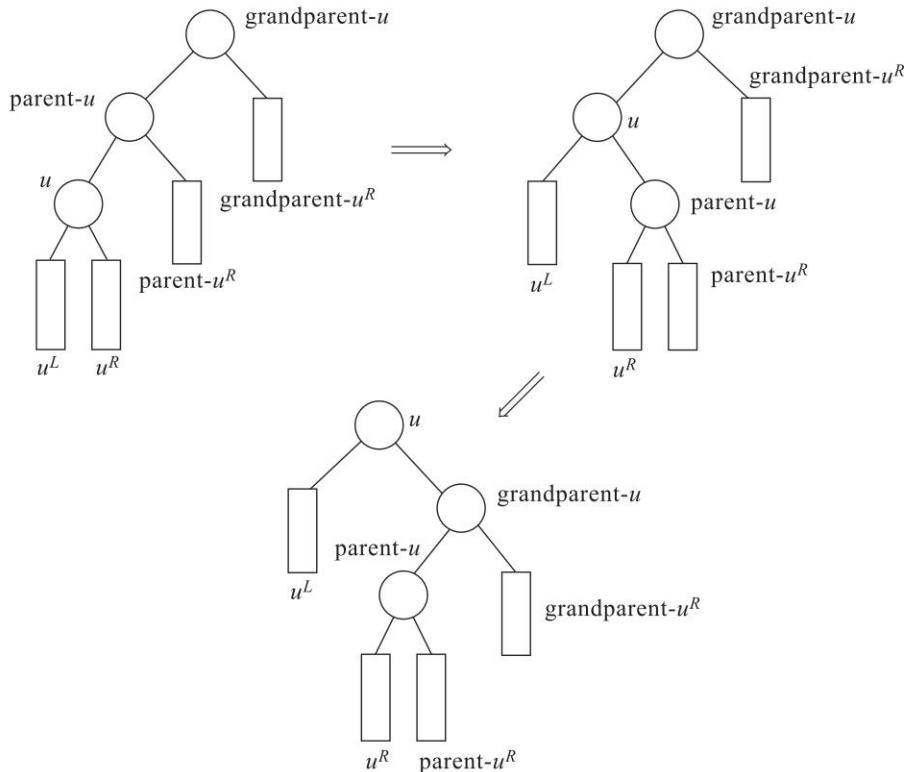


Fig. 12.18 Incorrect Zig-Zig using two single rotations

**Example 12.9** Consider the binary search tree shown in Fig. 12.19(a). Let us attempt splaying the tree at node 9. The path from the root to node 9 involves the path 24-12-10-9. Proceeding bottom-up, we perform a zig-zig on the triad, 9 (node  $u$ ), 10 ( node  $parent\_u$ ) and 12 (node  $grandparent\_u$ ). Fig. 12.19(b) shows the tree after the first step. Now the path from the root to node 9 involves only a single step, viz., zig. At the end of the zig rotation, the tree shown in Fig. 12.19(c) is obtained.

Let us continue to splay the tree shown in Fig. 12.19 (c) at 36. The path from the root to node 36 is given by 9-24-48-36. Proceeding bottom-up, the first step involves a zag-zig case. The rotation yields the tree shown in Fig. 12.19(d). Finally a zig case shows up which results in the splay tree shown in Fig. 12. 19(e).

**Example 12.10** Build a splay tree inserting the following elements in the sequence shown:  
 $H, Q, A, N, P, O$

The snapshots of the splay tree during the insertion of each of the elements is shown in Fig. 12. 20. Insertion of  $H$  determines the root of the splay tree (Fig. 12.20(a)). Insertion of  $Q$  calls for a zig rotation yielding the tree shown in Fig. 12.20(b). While insert  $A$  calls for a zig-zig case, insert  $N$  calls for two splay rotations viz., zag-zig and zig. Insert  $P$  calls for a zig-zig case and insert  $O$  calls for a zig-zag case.

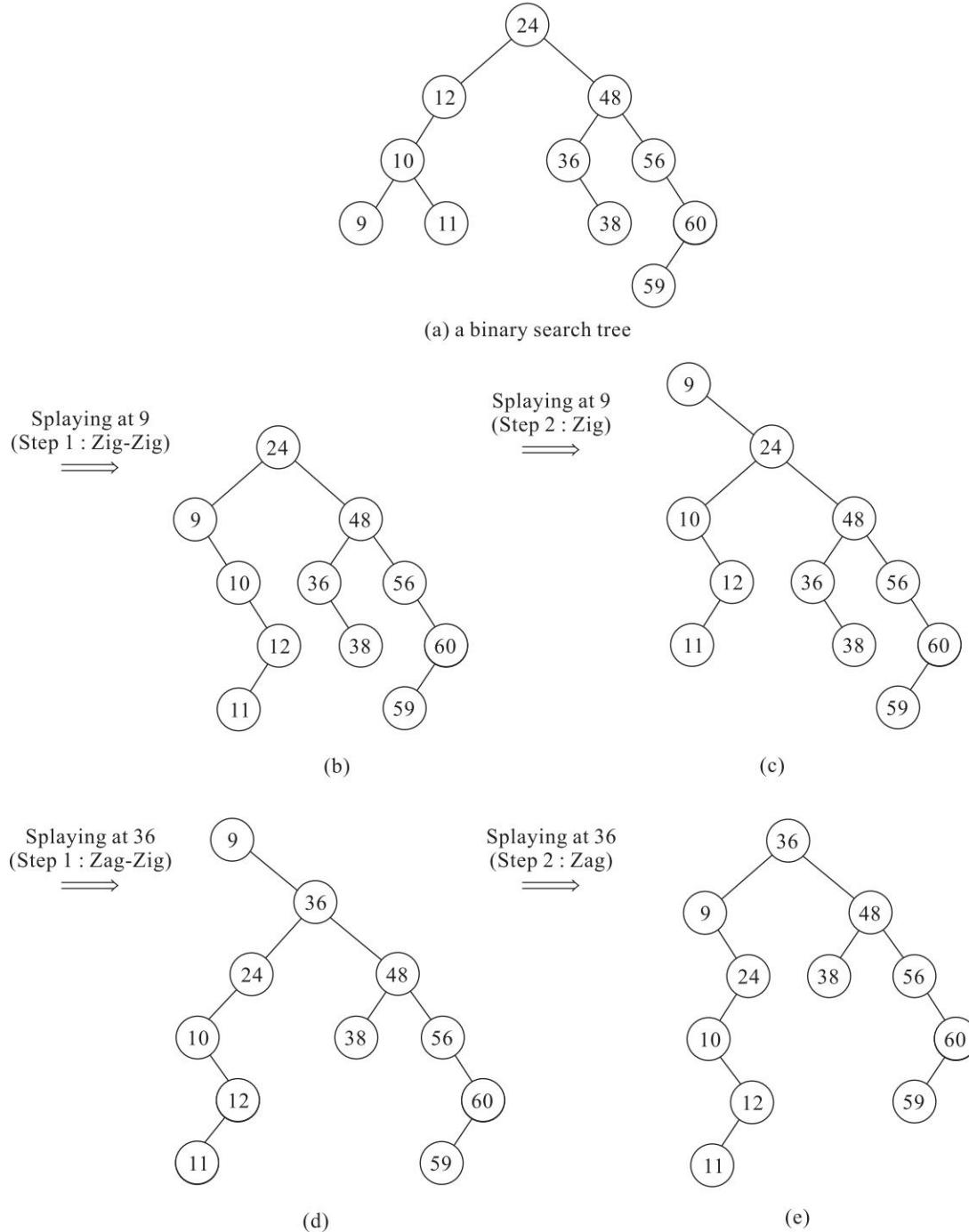
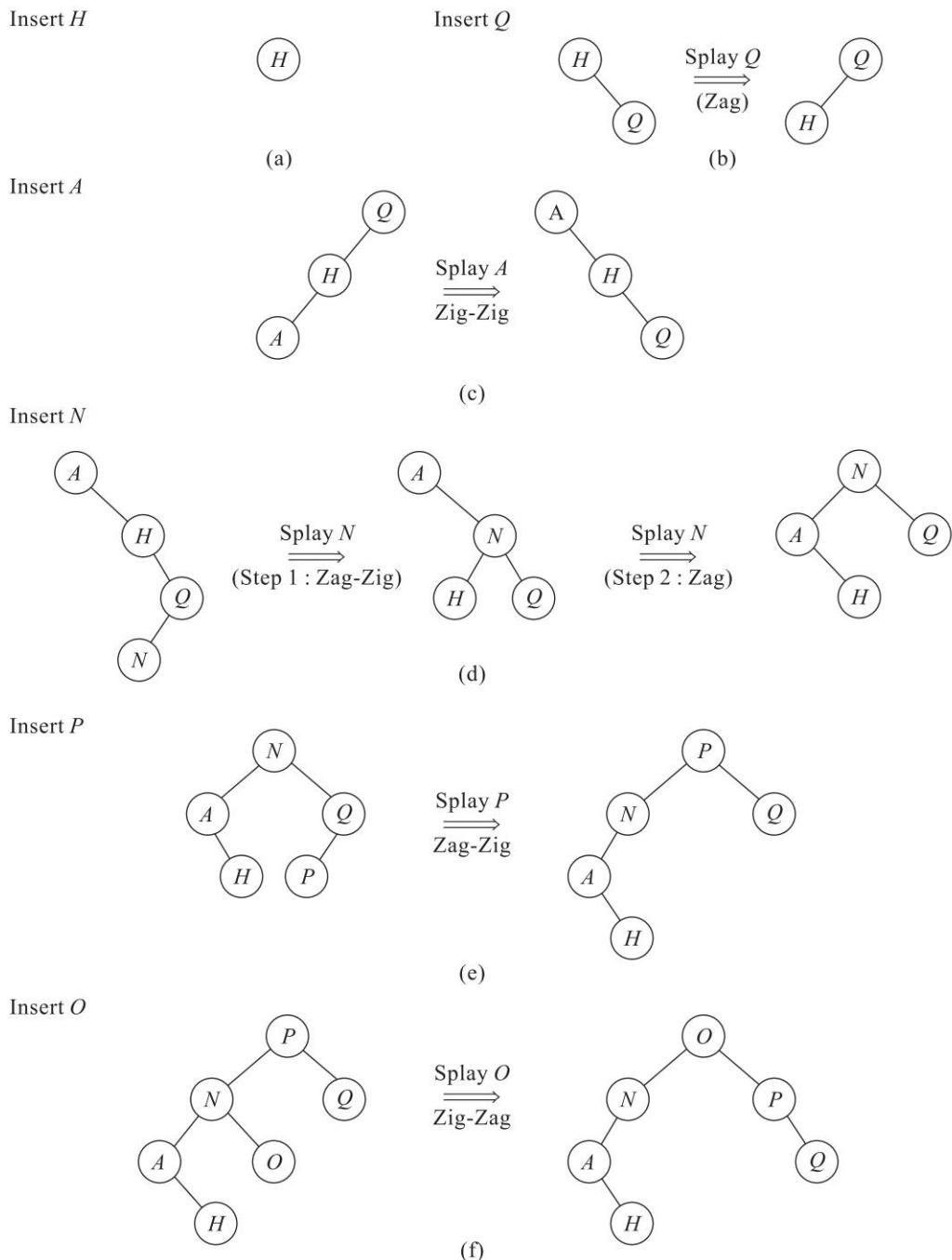


Fig. 12.19 Splaying a binary search tree



**Fig. 12.20** Snapshots of the splay tree (Example 12.10)

## Some remarks on amortized analysis of splay trees

Analysis of algorithms involves the computation of best, worst and average case time complexities based on the input instances to the algorithm. The analysis determines the work done by the algorithm over an instance or a specific class of instances.

Amortized analysis is different from these analyses in the sense that it estimates the work done by an algorithm over a long sequence of events rather than any single or a specific class of events in isolation. It is the worst case performance of an algorithm over a long sequence of events. In fact 'to amortize' itself means to extinguish a debt over a long period of regular installments.

It can be observed some times that one operation though expensive to perform for the first time, can lead to the same or other operations performed in a sequence thereafter, to get executed at a cheaper cost. Amortized analysis involves problems of such a nature. Thus amortized analysis is not average case analysis. While average case analysis deals with work done by the algorithm over a set of independent input instances, amortized analysis deals with the same over associated or related instances.

In the case of splay trees, it is observed that an insert splay or a search splay after its operation results in the specific key moving up to become the root. Let us suppose we were searching for a key  $K$  in the splay tree. Though the time complexity of such an operation is  $O(n)$  in the worst case, a subsequent execution of the same operation after splaying would only incur a time complexity that is definitely much less than  $O(n)$ ! In fact splay trees report an amortized time complexity of  $O(\log n)$ . Though an individual search operation for example, in a splay tree may not be  $O(\log n)$  the amortized time complexity of  $m$  operations on a splay tree would be  $O(m \cdot \log n)$ .

During the splaying process of a binary search tree  $T$ , let  $T_i(u)$  be the subtree of a node  $u$  which undergoes splaying in step  $i$ . Then the *rank* of the node  $u$  ( $r_i(u)$ ) in step  $i$  of the splaying process is defined to be

$$r_i(u) = \log_2 |T_i(u)|$$

where  $|T_i(u)|$  indicates the size of the subtree  $T_i(u)$ . In other words, if the subtree comprises  $s$  nodes, then the rank of node  $u$  is given by  $r(u) = \log_2 s$ . The *credit* of a node  $u$  is given by its rank  $r(u)$ . If  $u$  is a leaf node then  $r(u) = \log_2 1 = 0$  and if  $u$  is the root of the tree  $T$  with  $n$  nodes then  $r(u) = \log_2 n$ . Just as heights of trees act as a potential functions for the computation of their time complexities, ranks act as equally potential functions for the computation of amortized time complexities of splay trees. In fact while the heights of many nodes in the splay tree may get affected during a rotation operation, the ranks of the participating nodes, viz.,  $u$ ,  $parent\_u$  and  $grandparent\_u$  alone get affected during rotation.

The total credit balance for a tree is the sum of all the individual credits of its nodes. That is  $Cr_i = \sum_{u \in T_i} r_i(u)$  where  $Cr_i$  is the credit balance of the tree during the  $i$ th step of splaying the tree,

$r_i(u)$  is the rank of the node  $u$  in the  $i$ th step of splaying and  $T_i$  is the splayed tree in step  $i$ .

The amortized complexity  $A_i$  of a splay step  $i$  is given by

$$A_i = t_i + Cr_i - Cr_{i-1}$$

where  $t_i$  is the work done for the splay operation and  $Cr_i$  and  $Cr_{i-1}$  are the Credit balances of the tree before and after the splay operation.  $t_i$  is computed as the number of levels the target node rises during a splay operation. In the case of zig-zig, zig-zag, zag-zag and zag-zig splay operations,  $t_i$  is counted as 2 units whereas in the case of a simple zig or zag splay operation it

is counted as 1 unit.  $(Cr_i - Cr_{i-1})$  gives the change in the credit balance after the splay operation. Since the ranks of the participating nodes viz.,  $u$ ,  $parent\_u$  and  $grandparent\_u$  alone change during a splay operation it is obvious that the credit balance  $(Cr_i - Cr_{i-1})$  need be computed only with regard to these nodes. The rest of the summation with regard to the other nodes merely get cancelled.

The following results hold good with regard to the amortized analysis of splay trees:

- (i) The amortized complexity  $A_i$  of the splay tree, if step  $i$  of its splaying process initiates a zig-zig or a zag-zag step at the specific node  $u$ , satisfies the following relation:

$$A_i < 3 \cdot (r_i(u) - r_{i-1}(u)).$$

- (ii) The amortized complexity  $A_i$  of the splay tree, if step  $i$  of its splaying process initiates a zig-zag or a zag-zig step at the specific node  $u$ , satisfies the following relation:

$$A_i < 2 \cdot (r_i(u) - r_{i-1}(u)).$$

- (iii) The amortized complexity  $A_i$  of the splay tree, if step  $i$  of its splaying process initiates a zig or a zag step at the specific node  $u$ , satisfies the following relation:

$$A_i < 1 + (r_i(u) - r_{i-1}(u)).$$

- (iv) In a binary search tree with  $n$  nodes, the amortized cost  $C(n)$  of an insertion or search of a specific node with splaying does not exceed  $(1 + 3\log_2 n)$  upward moves from the specific node.
- (v) In a binary search tree with not more than  $n$  nodes, the total complexity of a sequence of  $w$  insertions or search operations with splaying, does not exceed  $w \cdot (1 + 3\log_2 n) + \log_2 n$ .

## Applications

## 12.3

Red-black trees which are derived from B trees of order 4 are only a variant of binary search trees. Hence any application that calls for binary search trees can also call for red-black trees.

On the other hand, splay trees which are typical binary search trees undergo splaying to favor retrievals which are efficient with regard to their amortized complexity. They are suitable for applications with the characteristic that information that is recently retrieved is highly likely to be retrieved in the near future. For example, in the case of a university information system, at the beginning of the admission season, those records pertaining to the newly admitted students are highly likely to be accessed over and over again in the first few weeks of their entry. In such a case, it would be a good move to store the records as a splay tree rather than a binary search tree. Since a splay tree pushes the recently retrieved records to stay closer to the root and in due course those records that were remotely used move farther and farther away from the root and occupy positions close to the fringe of the tree. Maintenance of patient records in a hospital information system, maintenance of records pertaining to seasonal items in a supermarket information system are some examples where splay trees find ideal applications.

Splay trees have also found applications in data compression, lexicographic search trees and dynamic Huffman coding.



## Summary

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



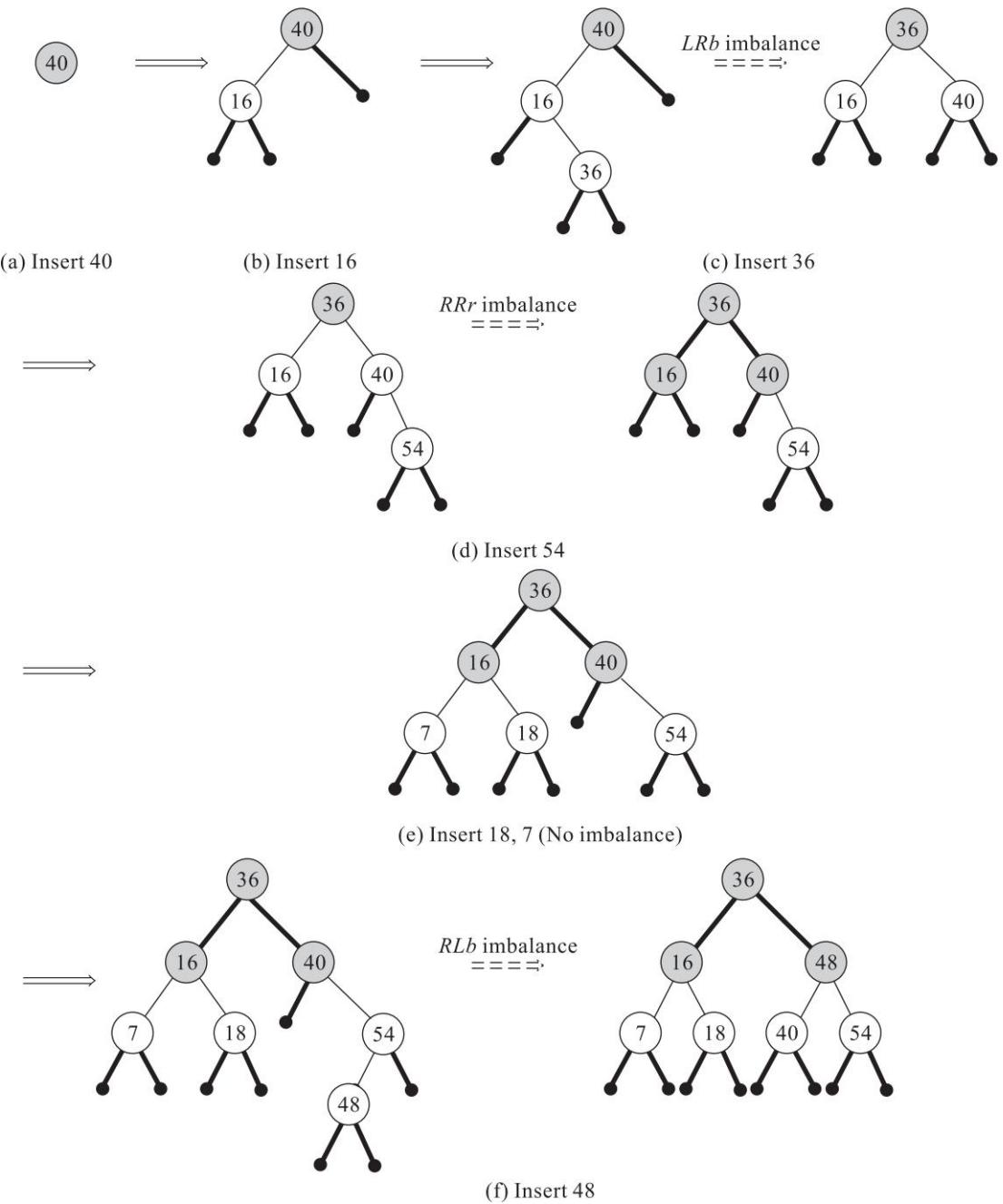
## Illustrative Problems

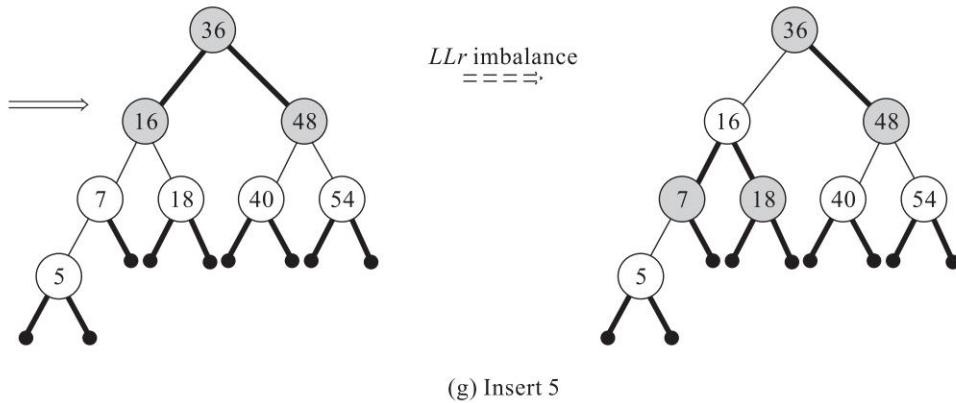
**Problem 12.1** Construct a red-black tree inserting the following keys into an empty tree, in the sequence given:

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

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

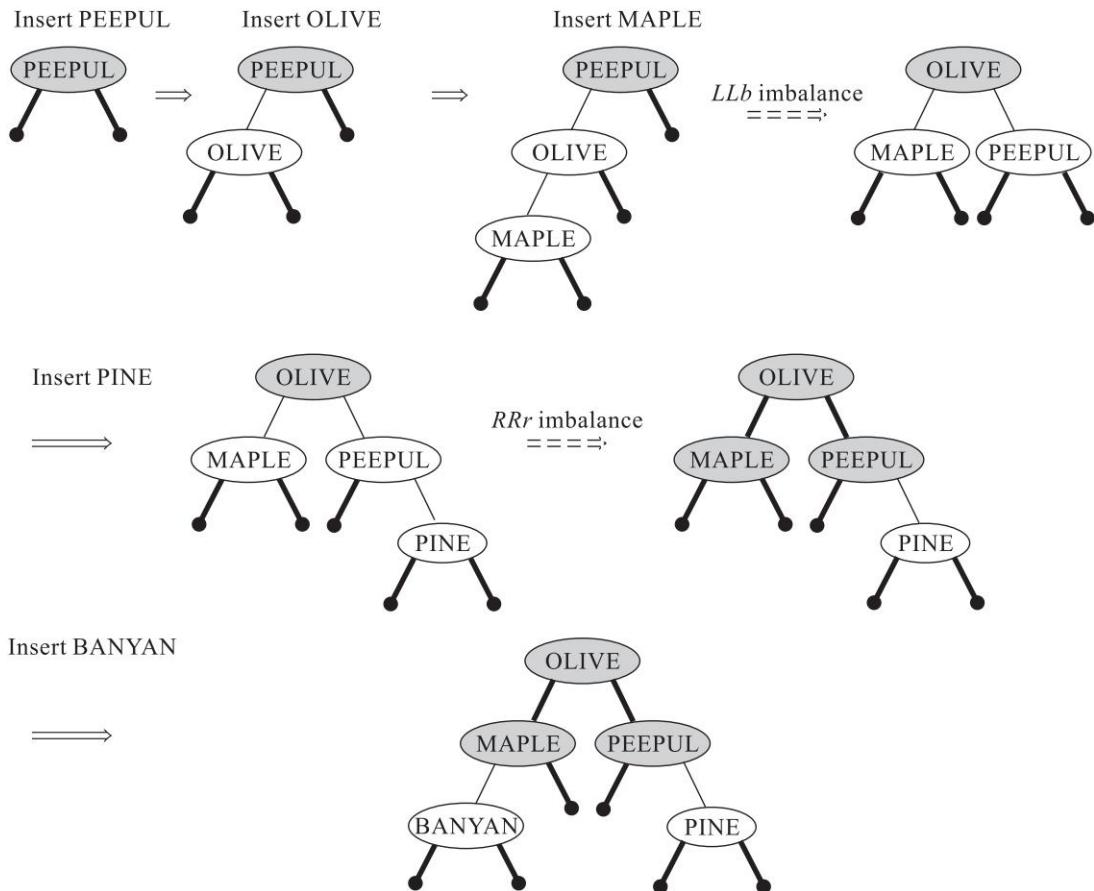
such that the number of black nodes in all the paths from the root to the external nodes increases by 1.



**Fig. I 12.1**

**Problem 12.2** Build a red-black tree using the keys given below:  
PEEPUL, OLIVE, MAPLE, PINE, BANYAN, CHESTNUT

**Solution:** Figure I 12.2 illustrates the snapshots of the red-black tree during its construction.



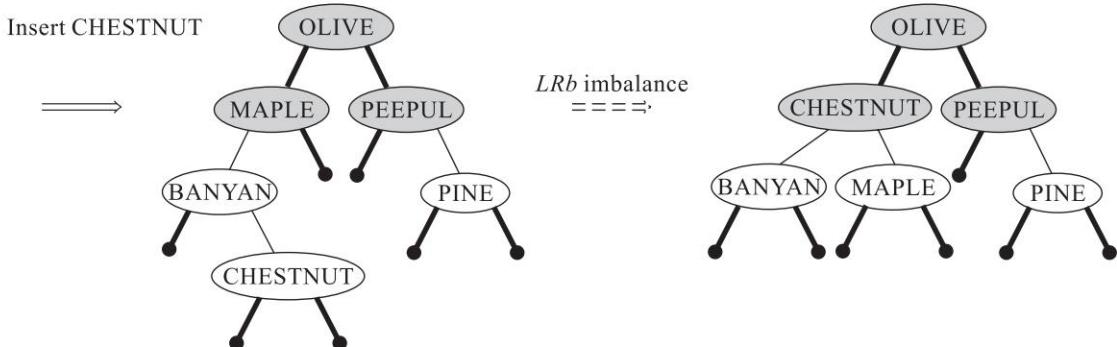
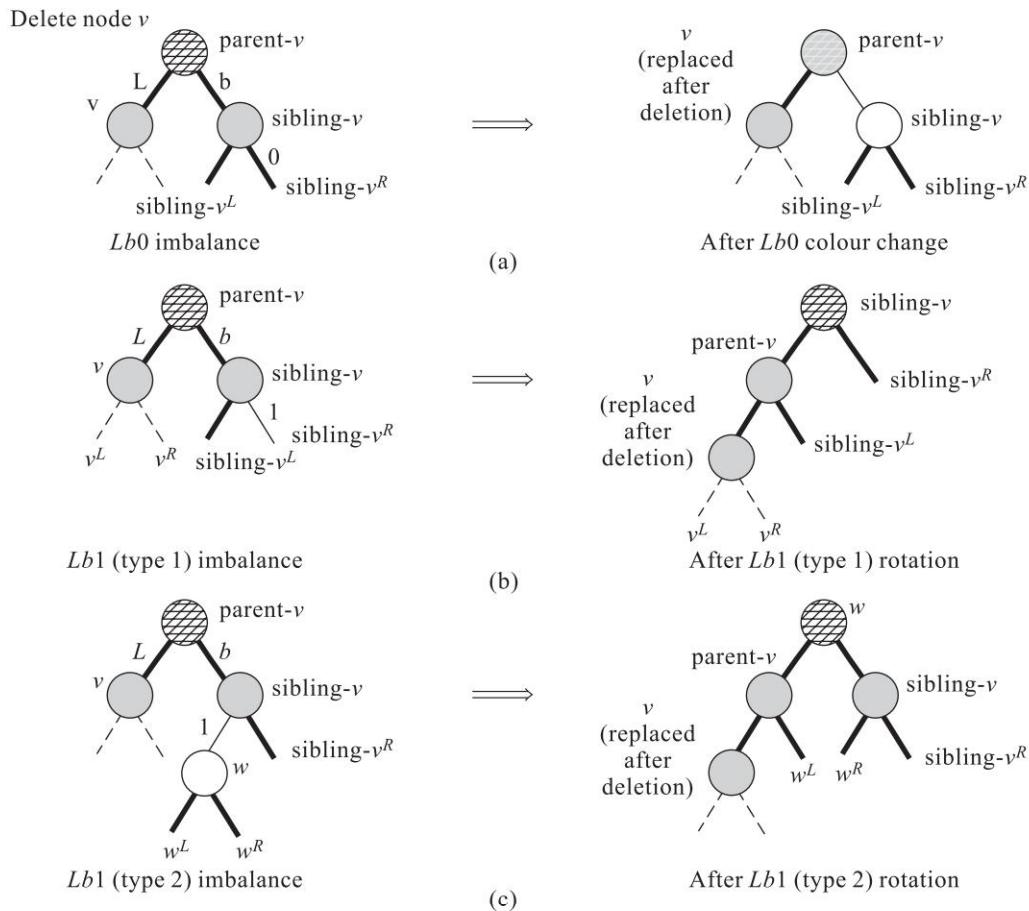
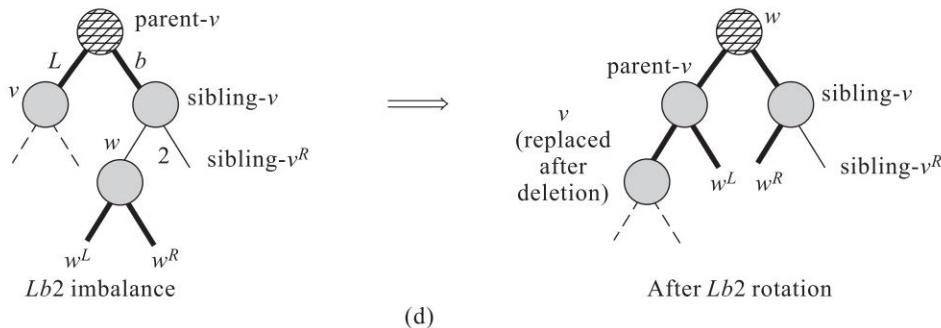


Fig. I 12.2

**Problem 12.3** Obtain generic representations for the  $Lb0$ ,  $Lb1$  and  $Lb2$  rotations on lines similar to that of their  $Rbx$  counterparts discussed in Sec. 12.5.

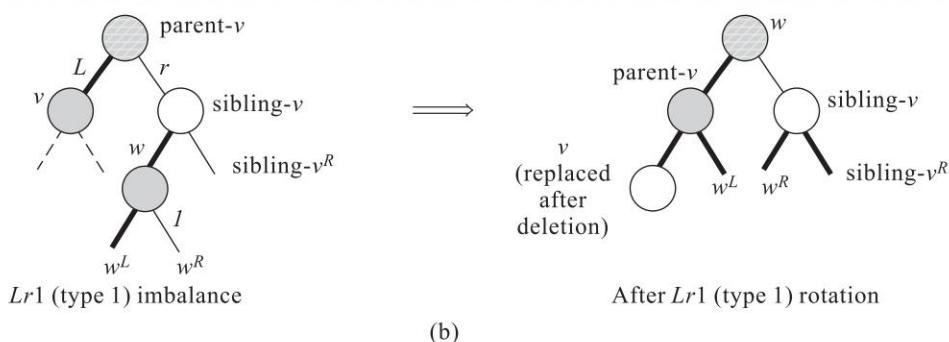
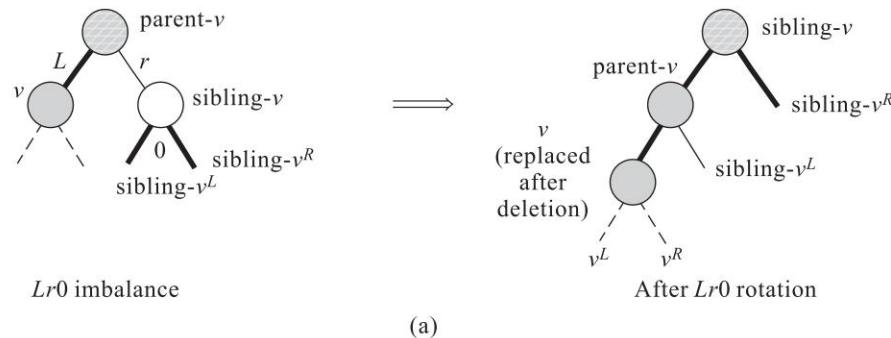
**Solution:** The generic representations of the rotations following similar notations and style of their  $Rbx$  counterparts are shown as follows:

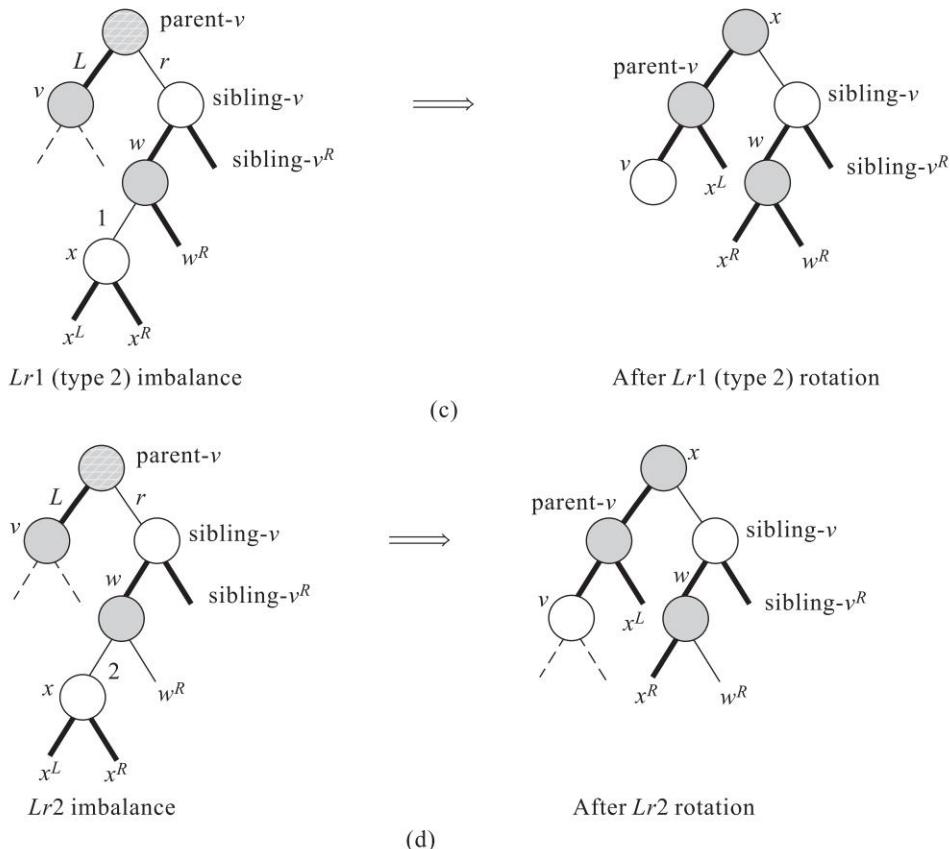




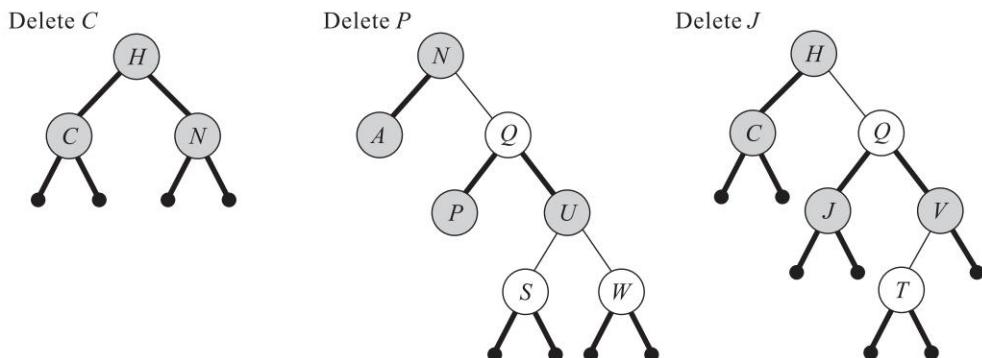
**Problem 12.4** Obtain generic representations for the *Lr0*, *Lr1* and *Lr2* rotations on lines similar to that of their *Rrx* counterparts discussed in the main text of this chapter.

**Solution:** The generic representations of the rotations following similar notations and style of their *Rrx* counterparts is shown as follows:



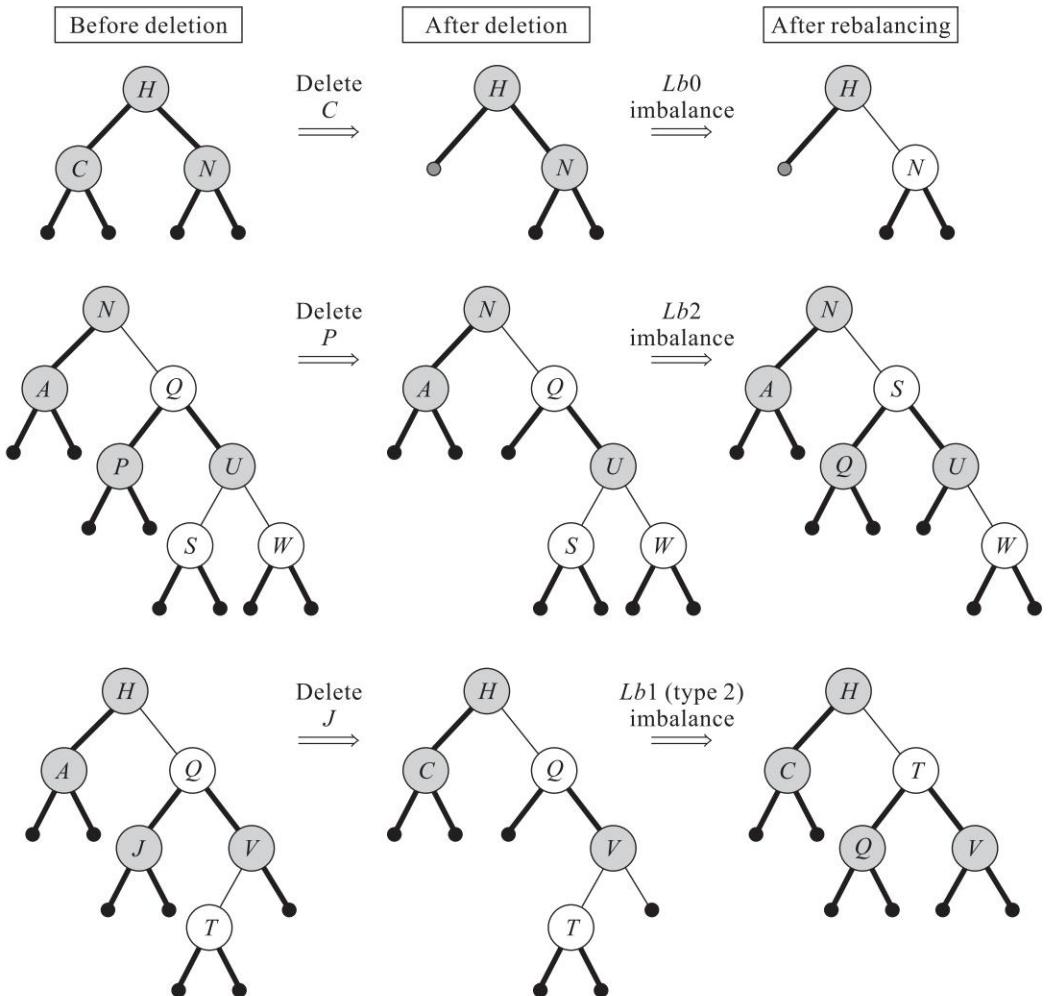


**Problem 12.5** Perform the corresponding operations on the red-black trees shown as follows:

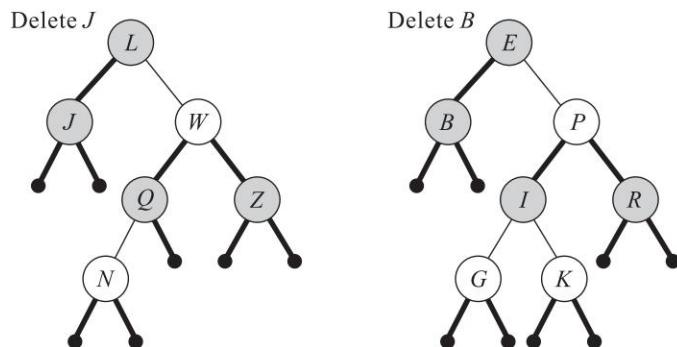


**Solution:** The snap shots of the red-black trees after the performance of the operations and after rebalancing are shown as follows:

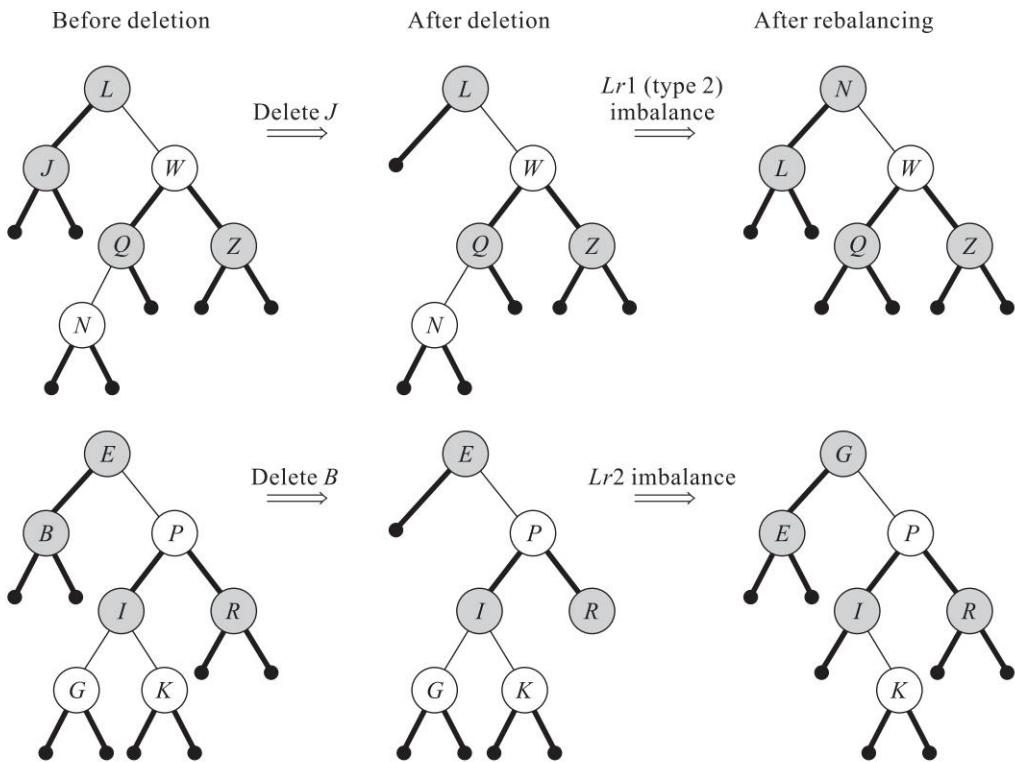
## Red-Black Trees and Splay Trees



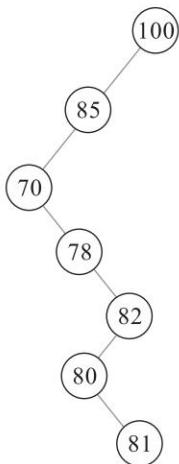
**Problem 12.6** Undertake the respective delete operations on the red-black trees shown as follows:



**Solution:** The state of the red-black trees after the delete operations and after rebalancing are shown as follows:

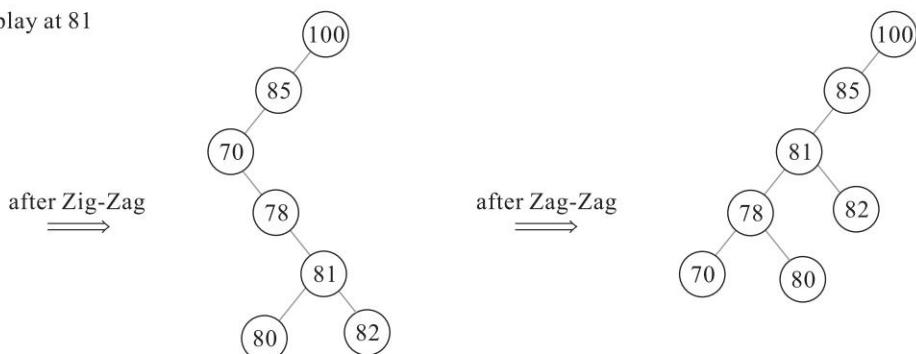


**Problem 12.7** Undertake splaying of the following binary search tree at key 81.



**Solution:** The snap shots of the tree during the splay rotations are shown as follows:

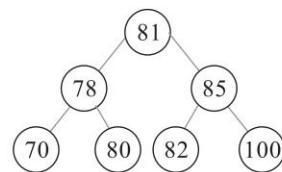
Splay at 81



after Zig-Zag  
====>

after Zag-Zag  
====>

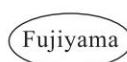
after Zig-Zig  
====>



**Problem 12.8** Build a splay tree inserting the following keys in the order shown:  
Fujiyama, Zao, Mt. Etna, Vesuvius, South Sister, Usu

**Solution:** The snapshots of the splay tree during its construction is shown below:

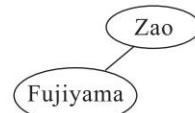
Insert Fujiyama:



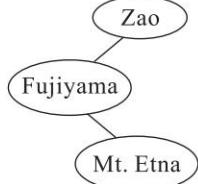
Insert Zao:



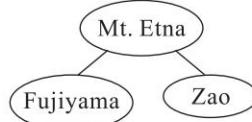
Zag  
====>



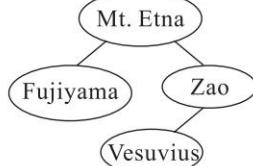
Insert Mt. Etna:



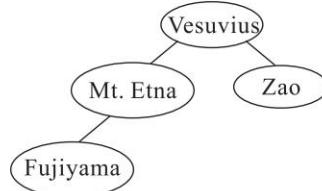
Zig-Zag  
====>

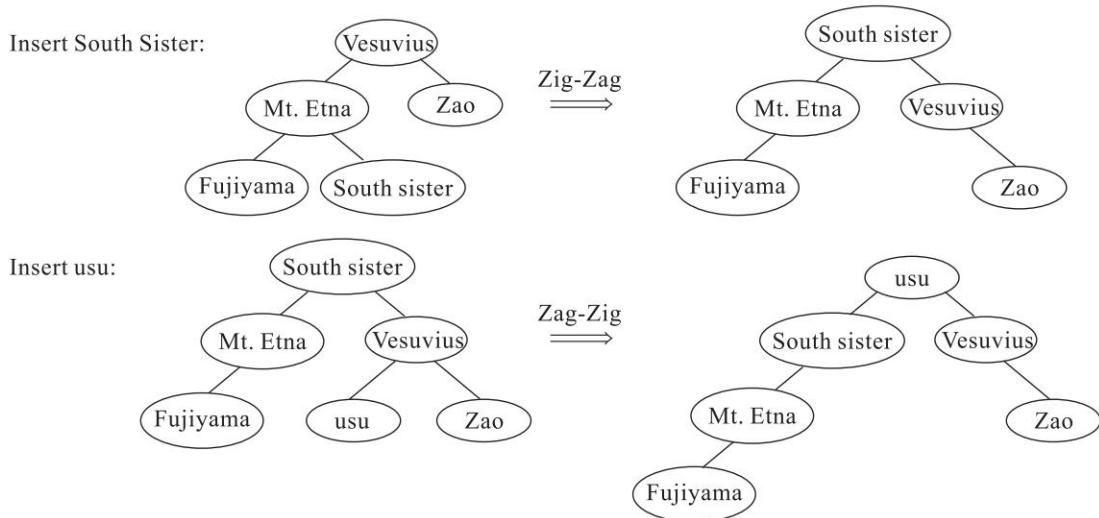


Insert Vesuvius:



Zag-Zig  
====>





**Problem 12.9** For the splay tree built in Illustrative Problem 12.7, trace the ranks of the involved nodes during the splaying steps.

**Solution:** The ranks of the participating nodes viz.,  $r(u)$ ,  $r(\text{parent}_u)$  and  $r(\text{grandparent}_u)$  are shown in Table I 12.9. The ranks of the other nodes during the splaying steps are also shown. Note how there is no change in their ranks though the heights of some of these nodes do undergo changes during the rotations.

**Table I 12.9**

| Operation                                  | Other nodes in the tree |             |             |             | Node $u$    | Node $\text{parent}_u$ | Node $\text{grandparent}_u$ |
|--------------------------------------------|-------------------------|-------------|-------------|-------------|-------------|------------------------|-----------------------------|
| (Step 1)                                   | 100                     | 85          | 78          | 70          | 81          | 80                     | 82                          |
| Rank of the nodes before zig-zag operation | $\log_2(7)$             | $\log_2(6)$ | $\log_2(4)$ | $\log_2(5)$ | $\log_2(1)$ | $\log_2(2)$            | $\log_2(3)$                 |
| Rank of the nodes after zig-zag operation  | $\log_2(7)$             | $\log_2(6)$ | $\log_2(4)$ | $\log_2(5)$ | $\log_2(3)$ | $\log_2(1)$            | $\log_2(1)$                 |
| Operation                                  | Other nodes in the tree |             |             |             | Node $u$    | Node $\text{parent}_u$ | Node $\text{grandparent}_u$ |
| (Step 2)                                   | 100                     | 85          | 82          | 80          | 81          | 78                     | 70                          |
| Rank of the nodes before zag-zag operation | $\log_2(7)$             | $\log_2(6)$ | $\log_2(1)$ | $\log_2(1)$ | $\log_2(3)$ | $\log_2(4)$            | $\log_2(5)$                 |
| Rank of the nodes after zag-zag operation  | $\log_2(7)$             | $\log_2(6)$ | $\log_2(1)$ | $\log_2(1)$ | $\log_2(5)$ | $\log_2(3)$            | $\log_2(1)$                 |
| Operation                                  | Other nodes in the tree |             |             |             | Node $u$    | Node $\text{parent}_u$ | Node $\text{grandparent}_u$ |
| (Step 3)                                   | 82                      | 80          | 78          | 70          | 81          | 85                     | 100                         |
| Rank of the nodes before zig-zig operation | $\log_2(1)$             | $\log_2(1)$ | $\log_2(3)$ | $\log_2(1)$ | $\log_2(5)$ | $\log_2(6)$            | $\log_2(7)$                 |
| Rank of the nodes after zig-zig operation  | $\log_2(1)$             | $\log_2(1)$ | $\log_2(3)$ | $\log_2(1)$ | $\log_2(7)$ | $\log_2(3)$            | $\log_2(1)$                 |

**Problem 12.10** Obtain the amortized complexity of each of the splay steps during the splaying of the tree at node 81 shown in Illustrative Problem 12.7. Make use of Table I 12.9 for ease of computation.

**Solution:** The amortized complexity of a splay step  $i$  is given by  $A_i = t_i + Cr_i - Cr_{i-1}$  where  $t_i$  the work done is 1 unit for a zig or a zag operation and 2 units for all other operations. The change in the credit balance ( $Cr_i - Cr_{i-1}$ ) is computed as the difference in sum of the ranks of the participating nodes.

The amortized complexities of the three steps involved in the splaying of 81 is given below:

**Step 1** zig-zag operation:  $A_1 = 2 + ((\log_2(3) + 0 + 0) - (0 + \log_2(2) + \log_2(3))) = 2 - \log_2 2 = 1$

It can be observed that the amortized time complexity  $A_i$  of the zig-zag operation satisfies the relation  $A_i < 2(r_i(u) - r_{i-1}(u))$ .

$$\text{(i.e.)} \quad A_1 = 1$$

$$< 2(r_i(81) - r_{i-1}(81)) = 2(\log_2 3 - \log_2 1) = 2 \log_2 3.$$

**Step 2** zag-zag operation:  $A_2 = 2 + ((\log_2 5 + \log_2 3 + 0) - (\log_2 3 + \log_2 4 + \log_2 5)) = 0$

It can be observed that the amortized time complexity  $A_i$  of the zag-zag operation satisfies the relation  $A_i < 3(r_i(u) - r_{i-1}(u))$

$$\text{(i.e.)} \quad A_2 = 0$$

$$< 3(r_i(81) - r_{i-1}(81)) = 3(\log_2 5 - \log_2 3)$$

**Step 3** zig-zig operation:  $A_3 = 2 + ((\log_2 7 + \log_2 3 + \log_2 1) - (\log_2 5 + \log_2 6 + \log_2 7)) = 1 - \log_2 5$

It can be observed that the amortized time complexity  $A_i$  of the zig-zig operation satisfies the relation  $A_i < 3(r_i(u) - r_{i-1}(u))$

$$\text{(i.e.)} \quad A_3 = 1 - \log_2 5$$

$$< 3(r_i(81) - r_{i-1}(81)) = 3(\log_2 7 - \log_2 5)$$



## Review Questions

- Which among the following properties does not hold good for a Red-Black tree?
  - the root node is always a black node.
  - all external nodes are black nodes.
  - two red nodes can occur consecutively on the path from the root node to an external node.
  - the number of black nodes on the path from the root node to an external node must be the same for all external nodes.
  - (i) (ii) (iii) (iv)
- Which among the following calls for a rotation to set right the imbalance?
  - $LRb$
  - $LrR$
  - $RLr$
  - $RRr$
  - (i) (ii) (iii) (iv)
- In the context of deletion of a node from a red-black tree, state whether true or false:
  - if the deleted node were red, then the Black Condition would be violated and hence the tree is unbalanced.
  - if the deleted node were to be black then the Black Condition is violated and hence the tree is unbalanced.
  - (i) true (ii) true (b) (i) true (ii) false
  - (c) (i) false (ii) true (d) (i) false (ii) false

4. Which among the following properties is not satisfied by a Splay tree?
  - (i) splay trees are binary search trees.
  - (ii) splay trees result in efficient repeated accesses.
  - (iii) splay trees like AVL trees are always height balanced.
  - (iv) splay trees have their frequently accessed nodes moving towards the root.

(a) (i)                   (b) (ii)                   (c) (iii)                   (d) (iv)
5. In the context of splay rotations, state whether true or false:
  - (i) if the length of the path from the root to the accessed node  $u$  is even, then the rotations undertaken are associated with zig-zig, zig-zag, zag-zig or zag-zag.
  - (ii) if the length of the path from the root to the accessed node  $u$  is odd, then the rotations undertaken are associated with either a zig or a zag.

(a) (i) true (ii) true                           (b) (i) true (ii) false  
                                                          (c) (i) false (ii) true                           (d) (i) false (ii) false
6. What are the merits of Red-Black trees over B-trees of order  $m$ ?
7. Outline the generic representation of an XY $r$  imbalance, where X, Y could be either an L or R.
8. What is the need for Splay trees?
9. How are splay rotations performed?
10. What is the amortized time complexity of a search operation on a splay tree?
11. For the following list of data construct a red black tree:  
 LINUX, OS2, DOS, XENIX, SOLARIS, WINDOWS, VISTA, XP, UNIX, CPM,  
 Undertake the following operations on the tree:
  - (i) Insert MAC                                   (ii) Delete WINDOWS                                   (iii) Delete UNIX.
12. Represent the data list shown in Review Questions 11 (Chapter 12) as a Splay tree. Tabulate the number of comparisons undertaken for retrieving the following keys:
  - (i) LINUX                                       (ii) XENIX                                           (iii) LINUX                                           (iv) LINUX
  - (v) LINUX.



## Programming Assignments

1. Implement a function RB\_IMBALANCE( $T$ ) which given a red-black tree  $T$  would test for the violation of Red and Black Conditions.
2. Execute a menu driven program to insert keys into an initially empty red-black tree. Make use of the function RB\_IMBALANCE( $T$ ) developed in Programming Assignments 1 (Chapter 12) to test for any imbalance. Display the tree after rebalancing it.
3. Implement a program to accept a non-empty red-black tree (make use of Programming Assignments 12 (Chapter 12)) as input and delete all its leaf nodes. Rebalance the tree after every deletion and display the rebalanced tree on the screen.
4. Execute a program with animations and graphics to demonstrate the splaying of a tree given a specific node  $u$  in the tree.
5. In the menu-driven program implemented in Programming Assignments 1 (Chapter 10) to perform the search, insert and delete operations on a binary search tree, introduce functions to splay the tree soon after every insert and search operation is executed.



# HASH TABLES

# 13

## Introduction

## 13.1

The data structures of binary search trees, AVL trees, B trees, tries, red-black trees, splay trees discussed so far in this part of the book, are tree based data structures. These are non-linear data structures and serve to capture the hierarchical relationship existing between the elements forming the data structure. However, there exist applications which deal with linear or tabular forms of data, devoid of any superior–subordinate relationship. In such cases employing these data structures would be superfluous. Hash tables are one among such data structures which favor efficient storage and retrieval of data elements which are linear in nature.

## Dictionaries

*Dictionary* is a collection of data elements uniquely identified by a field called *key*. A dictionary supports the operations of search, insert and delete. The ADT of a dictionary is defined as a set of elements with distinct keys supporting the operations of search, insert, delete and create (which creates an empty dictionary). While most dictionaries deal with distinct keyed elements, it is not uncommon to find applications calling for dictionaries with duplicate or repeated keys. In this case it is essential that the dictionary evolves rules to resolve the ambiguity that may arise while searching for or deleting data elements with duplicate keys.

A dictionary supports both *sequential* and *random access*. A sequential access is one in which the data elements of the dictionary are ordered and accessed according to the order of the keys (ascending or descending, for example). A random access is one in which the data elements of the dictionary are not accessed according to a particular order.

Hash tables are ideal data structures for dictionaries. In this chapter we introduce the concept of hashing and hash functions. The structure and operations of the hash tables are also discussed. The various methods of collision resolution viz., linear open addressing and chaining, and their performance analyses are detailed. Finally the application of hash tables in the fields of compiler design, relational database query processing and file organization are discussed.

13.1 *Introduction*

13.2 *Hash Table Structure*

13.3 *Hash Functions*

13.4 *Linear open addressing*

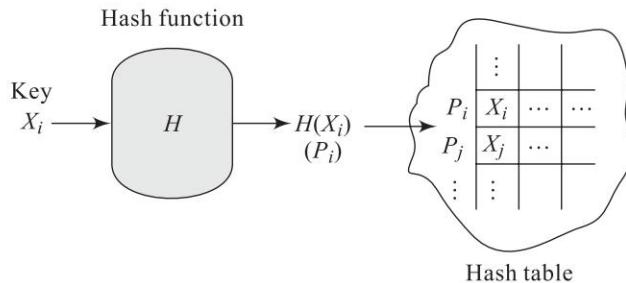
13.5 *Chaining*

13.6 *Applications*

## Hash Table Structure

## 13.2

A *hash function*  $H(X)$  is a mathematical function which given a key  $X$  of the dictionary  $D$ , maps it to a position  $P$  in a storage table termed *hash table*. The process of mapping the keys to their respective positions in the hash table is called *hashing*. Figure 13.1 illustrates a hash function.



**Fig. 13.1 Hashing a key**

When the data elements of the dictionary are to be stored in the hash table, each key  $X_i$  is mapped to a position  $P_i$  in the hash table as determined by the value of  $H(X_i)$ , (i.e.)  $P_i = H(X_i)$ . To search for a key  $X$  in the hash table all that one does is to determine the position  $P$  by computing  $P = H(X)$  and access the appropriate data element. In the case of insertion of a key  $X$  or its deletion, the position  $P$  in the hash table where the data element needs to be inserted or from where it is to be deleted respectively, is determined by computing  $P = H(X)$ .

If the hash table is implemented using a sequential data structure, for example arrays, then the hash function  $H(X)$  may be so chosen to yield a value that corresponds to the index of the array. In such a case, the hash function is a mere mapping of the keys to the array indices.

**Example 13.1** Consider a set of distinct keys { AB12, VP99, RK32, CG45, KL78, OW31, ST65, EX44 } to be represented as a hash table. Let us suppose the hash function  $H$  is defined as below:

$H(XYmn) = \text{ord}(X)$  where  $X, Y$  are the alphabetical characters,  
 $m, n$  are the numerical characters of the key and  $\text{ord}(X)$  is the  
ordinal number of the alphabet  $X$ .

The computation of the positions of the keys in the hash table is shown below:

| Key XYmn | $H(XYmn)$       | Position of the key in the hash table |
|----------|-----------------|---------------------------------------|
| AB12     | $\text{ord}(A)$ | 1                                     |
| VP99     | $\text{ord}(V)$ | 22                                    |
| RK32     | $\text{ord}(I)$ | 18                                    |
| CG45     | $\text{ord}(C)$ | 3                                     |
| KL78     | $\text{ord}(K)$ | 11                                    |
| OW31     | $\text{ord}(O)$ | 15                                    |
| ST65     | $\text{ord}(S)$ | 19                                    |
| EX44     | $\text{ord}(E)$ | 5                                     |

In the Example 13.1, it was assumed that the hash function yields distinct values for the individual keys. If this were to be followed as a criterion, then the situation may turn out of control since in the case of dictionaries with very large set of data elements, the hash table size can be too huge to be handled efficiently. Therefore it is convenient to choose hash functions which yield values lying within a limited range so as to restrict the length of the table. This would consequently imply that the hash functions may yield identical values for a set of keys. In other words, a set of keys could be mapped to the same position in the hash table. Let  $X_1, X_2, \dots, X_n$  be the  $n$  keys which are mapped to the same position  $P$  in the hash table. Then  $H(X_1) = H(X_2) = \dots = H(X_n) = P$ . In such a case,  $X_1, X_2, \dots, X_n$  are called as *synonyms*. The act of two or more synonyms vying for the same position in the hash table is known as *collision*. Naturally, this entails a modification in the structure of the hash table to accommodate the synonyms. The two important methods of linear open addressing and chaining to handle synonyms are presented in Sec. 13.4 and Sec. 13.5 respectively.

The hash table accommodating the data elements appears as shown below:

|     |      |       |
|-----|------|-------|
| 1   | AB12 | ..... |
| 2   | ...  |       |
| 3   | CG45 |       |
| 4   | ...  |       |
| 5   | EX44 | ..... |
| ... | ...  |       |
| 11  | KL78 |       |
| ... |      |       |
| 15  | OW31 | ..... |
| ... |      |       |
| 18  | RK32 |       |
| 19  | ST65 | ..... |
| ... |      |       |
| 22  | VP99 | ..... |
| ... | ...  | ..... |

## Hash Functions

## 13.3

The choice of the hash function plays a significant role in the structure and performance of the hash table. It is therefore essential that a hash function satisfies the following characteristics:

- (i) easy and quick to compute
- (ii) even distribution of keys across the hash table. In other words, a hash function must minimize collisions.

## Building hash functions

The following are some of the methods of obtaining hash functions:

**(i) Folding:** The key is first partitioned into two or three or more parts. Each of the individual parts are combined using any of the basic arithmetic operations such as addition or multiplication. The resultant number could be conveniently manipulated, for example truncated, to finally arrive at the index where the key is to be stored. Folding assures better spread of keys across the hash table.

**Example** Consider a six digit numerical key: 719532. We choose to partition the key into three parts of two digits each, (i.e.) 71 | 95 | 32, and merely add the numerical equivalent of each of the parts, (i.e.)  $71 + 95 + 32 = 198$ . Truncating the result yields 98 which is chosen as the index of the hash table where the key 719532 is to be accommodated.

**(ii) Truncation:** In this method the selective digits of the key are extracted to determine the index of the hash table where the key needs to be accommodated. In the case of alphabetical keys their numerical equivalents may be considered. Truncation though quick to compute, does not ensure even distribution of keys.

**Example** Consider a group of six digit numerical keys that need to be accommodated in a hash table with 100 locations. We choose to select digits in position 3 and 6 to determine the index where the key is to be stored. Thus key 719532 would be stored in location 92 of the hash table.

**(iii) Modular Arithmetic:** This is a popular method and the size of the hash table  $L$  is involved in the computation of the hash function. The function makes use of modulo arithmetic. Let  $k$  be the numerical key or the numerical equivalent if it is an alphabetical key. The hash function is given by

$$H(k) = k \bmod L$$

The hash function evidently returns a value that lies between 0 and  $L-1$ . Choosing  $L$  to be a prime number has a proven better performance by way of even distribution of keys.

**Example** Consider a group of six digit numerical keys that need to be stored in a hash table of size 111. For a key 145682,  $H(k) = 145682 \bmod 111 = 50$ . Hence the key is stored in location 50 of the hash table.

## Linear Open Addressing

13.4

Let us suppose a group of keys are to be inserted into a hash table  $HT$  of size  $L$ , making use of the modulo arithmetic function  $H(k) = k \bmod L$ . Since the range of the hash table index is limited to lie between 0 and  $L-1$ , for a population of  $N$  ( $N > L$ ) keys collisions are bound to occur. Hence a provision needs to be made in the hash table to accommodate the data elements that are synonyms.

We choose to adopt a sequential data structure to accommodate the hash table. Let  $HT[0:L-1]$  be the hash table. Here the  $L$  locations of the hash table are termed as *buckets*. Every bucket provides accommodation for the data elements. However to accommodate synonyms (i.e.) keys which map to the same bucket, it is essential that a provision be made in the buckets. We therefore

partition buckets into what are called *slots* to accommodate synonyms. Thus if a bucket  $b$  has  $s$  slots, then  $s$  synonyms can be accommodated in the bucket  $b$ . In the case of an array implementation of a hash table, the rows of the array indicate buckets and the columns the slots. In such a case, the hash table is represented as  $HT[0:L-1, 0:s-1]$ . The choice of number of slots in a bucket needs to be decided based on the application. Figure 13.2 illustrates a general hash table implemented using a sequential data structure.

**Example 13.2** Let us consider a set of keys  $\{45, 98, 12, 55, 46, 89, 65, 88, 36, 21\}$  to be represented as a hash table as shown in Fig. 13.2. Let us suppose the hash function  $H$  is defined as  $H(X) = X \bmod 11$ . The hash table therefore has 11 buckets. We propose 3 slots per bucket. Table 13.1 shows the hash function values of the keys and Fig. 13.3 shows the structure of the hash table.

**Table 13.1** Hash function values of the keys (Example 13.2)

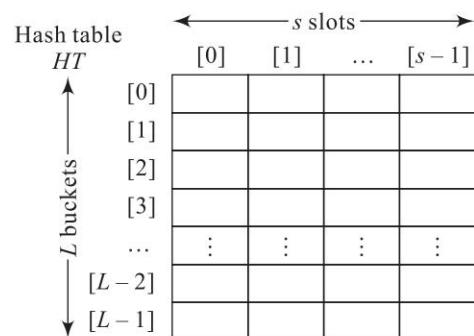
| Key X | 45 | 98 | 12 | 55 | 46 | 89 | 65 | 88 | 36 | 21 |
|-------|----|----|----|----|----|----|----|----|----|----|
| H(X)  | 1  | 10 | 1  | 0  | 2  | 1  | 10 | 0  | 3  | 10 |

Observe how keys  $\{45, 12, 89\}$ ,  $\{98, 65, 21\}$  and  $\{55, 88\}$  are synonyms mapping to the same bucket 1, 10 and 0 respectively. The provision of 3 slots per bucket makes it possible to accommodate synonyms.

Now what happens if a synonym is unable to find a slot in the bucket? In other words, if the bucket is full, then where do we find place for the synonyms? In such a case an *overflow* is said to have occurred. All collisions need not result in overflows. But in the case of a hash table with single slot buckets, collisions mean overflows.

The bucket to which the key is mapped by the hash function is known as the *home bucket*. To tackle overflows we move further down, beginning from the home bucket and look for the closest slot that is empty and place the key in it. Such a method of handling overflows is known as *Linear probing* or *Linear open addressing* or *closed hashing*.

**Example 13.3** Let us proceed to insert the keys  $\{77, 34, 43\}$  in the hash table discussed in Example 13.2. The hash function values of the keys are  $\{0, 1, 10\}$ . When we proceed to insert 77 in its home bucket 0, we find a slot is available and hence the insertion is done. In the case of 34, its home bucket 1 is full and hence there is overflow. By linear probing, we look for the closest slot that is vacant and find one in the second slot of bucket 2. While inserting 43, we find bucket



**Fig. 13.2** Hash table implemented using a sequential data structure

| Bucket | [0] | [1] | [2] |
|--------|-----|-----|-----|
| [0]    | 55  | 88  |     |
| [1]    | 45  | 12  | 89  |
| [2]    | 46  |     |     |
| [3]    | 36  |     |     |
| [4]    |     |     |     |
| [5]    |     |     |     |
| [6]    |     |     |     |
| [7]    |     |     |     |
| [8]    |     |     |     |
| [9]    |     |     |     |
| [10]   | 98  | 65  | 21  |

**Fig. 13.3** Hash table (Example 13.2)

10 to be full. The search for the closest empty slot proceeds by moving downwards in a circular fashion until it finds a vacant place in slot 3 of bucket 2. Note the circular movement of searching the hash table while looking for an empty slot. Figure 13.3 illustrates the linear probing method undertaken for the listed keys. The keys which have been accommodated in places other than their home buckets are shown over a grey background.

## Operations on linear open addressed hash tables

**Search:** Searching for a key in a linear open addressed hash table proceeds on lines similar to that of insertion. However, if the searched key is available in the home bucket then the search is done. The time complexity in such a case is  $O(1)$ . However, if there had been overflows while inserting the key, then a sequential search has to be called for, which searches through each slot of the buckets following the home bucket, until either (i) the key is found or (ii) an empty slot is encountered in which case the search terminates or (iii) the search path has curled back to the home bucket. In the case of (i) the search is said to be successful. In the case of (ii) and (iii) it is said to be unsuccessful.

**Example 13.4** Consider the snapshot of the hash table shown in Fig. 13.5, which represents keys whose first character lies between 'A' and 'I', both inclusive. The hash function used is  $H(X) = \text{ord}(C) \bmod 10$  where C is the first character of the alphabetical key X. The search for keys F18 and G64 are straightforward since they are present in their home buckets viz., 6 and 7 respectively. The search for keys A91 and F78 for example, are a trifle involved in the sense that, though they are available in their respective home buckets, they are accessed only after a sequential search for them is done in the slots corresponding to their buckets. On the other hand, the search for I99 fails to find it in its home bucket viz., 9. This therefore triggers a sequential search of every slot following the home bucket until the key is found, in which case the search is successful or until an empty slot is encountered in which case the search is a failure. I99 is indeed found in slot 2 of bucket 2. Observe how the search path curls back to the top of the hash table from the home bucket of key I99. Let us now search for the key G93. The search proceeds to look into its home bucket (7) before a sequential search for the same is undertaken in the slots following the home bucket. The search stops due to its encountering an empty slot and therefore the search is deemed unsuccessful.

| HT   | [0] | [1] | [2] |
|------|-----|-----|-----|
| [0]  | 55  | 48  | 77  |
| [1]  | 45  | 12  | 89  |
| [2]  | 46  | 34  | 43  |
| [3]  | 36  |     |     |
| [4]  |     |     |     |
| [5]  |     |     |     |
| [6]  |     |     |     |
| [7]  |     |     |     |
| [8]  |     |     |     |
| [9]  |     |     |     |
| [10] | 98  | 65  | 21  |

**Fig. 13.4** Linear Open Addressing (Example 13.3)

| HT  | [0] | [2] |
|-----|-----|-----|
| [0] | I81 | I90 |
| [1] | A12 | A91 |
| [2] | B47 | I99 |
| [3] |     |     |
| [4] | D36 |     |
| [5] |     |     |
| [6] | F18 | F78 |
| [7] | G64 | F73 |
| [8] | H11 | F99 |
| [9] | I54 | I75 |
|     | :   | :   |

**Fig. 13.5** Illustration of search in a hash table

Algorithm 13.1 illustrates the search algorithm for a linear open addressed hash table.

**Algorithm 13.1:** Procedure to search for a key X in a linear open addressed hash table

```

procedure LOP_HASH_SEARCH(HT, b, s X)
    /* HT[0:b-1, 0:s-1] is the hash table implemented as a two
       dimensional array. Here b is the number of buckets and s is
       the number of slots. X is the key to be searched in the hash
       table. In case of unsuccessful search, the procedure prints
       the message "KEY not found" otherwise prints "KEY found"*/
    h = H(X); /* H(X) is the hash function computed on X */
    i = h; j = 0; /* i, j are the indexes for the bucket and slot
                         respectively*/
    while (HT[i, j] ≠ 0 and HT[i, j] ≠ X) do
        j = j + 1; /* search for X in the slots*/
        if (j > (s-1)) then j = 0; /* reset slot index to 0 to continue
   searching in the next bucket*/
        if (j == 0) then { i = (i+1) mod b; /* continue searching in
   the next bucket in a
   circular manner*/
        if (i == h) then print ("Key not found"); exit(); }
    endwhile
    if (HT[i, j]== X) then print (" KEY found");
    if (HT[i, j] = 0) then print (" KEY not found");
end LOP_HASH_SEARCH.

```

**Insert:** The insertion of data elements in a linear open addressed hash table is executed as explained in the previous section. The hash function that is quite often modulo arithmetic based, determines the bucket *b* and thereafter slot *s* in which the data element is to be inserted. In the case of overflow, we search for the closest empty slot beginning from the home bucket and accommodate the key in the slot. Algorithm 13.1 could be modified to execute the insert operation. The line

**if** (*HT[i, j] = 0*) **then** **print** (" KEY not found"); in the algorithm is replaced by  
**if** (*HT[i, j] = 0*) **then** *HT[i, j] = X*; /\* insert *X* in the empty slot\*/

**Delete:** The delete operation on a hash table can be clumsy. When a key is deleted it cannot be merely wiped off from its bucket (slot). A deletion leaves the slot vacant and if an empty slot is chosen as a signal to terminate a search then many of the elements following the empty slot and displaced from their home buckets may go unnoticed. To tackle this it is essential that the keys following the empty slot are moved up. This can make the whole operation clumsy.

An alternative could be, to write a special element in the slot every time a delete operation is done. This special element not only serves to camouflage the empty space 'available' in the deleted slot when a search is under progress, but also serves to accommodate an insertion when an appropriate element assigned to the slot turns up.

However, it is generally recommended that deletions in a hash table are avoided as much as possible due to their clumsy implementation.

## Performance analysis

The complexity of the linear open addressed hash table is dependent on the number of buckets. In the case of hash functions that follow modular arithmetic, the number of buckets is given by the divisor  $L$ . The best case time complexity of searching for a key in a hash table is given by  $O(1)$  and the worst case time complexity is given by  $O(n)$ , where  $n$  is the number of data elements stored in the hash table. A worst case occurs when all the  $n$  data elements map to the same bucket. The time complexities when compared to that of their linear list counterparts is not in any way less. The best and worst case complexity of searching for an element in a linear list of  $n$  elements is respectively,  $O(1)$  and  $O(n)$ . However, on an average the performance of the hash table is much more efficient than that of the linear lists. It has been shown that the average case performance of a linear open addressed hash table for successful and unsuccessful search is given by

$$U_n \sim \frac{1}{2} \left( 1 + \frac{1}{(1-\alpha)^2} \right) \quad \text{and}$$

$$S_n \sim \frac{1}{2} \left( 1 + \frac{1}{(1-\alpha)^2} \right),$$

where  $U_n$  and  $S_n$  are the number of buckets examined on an average during an unsuccessful and successful search respectively. The average is considered over all possible sequences of the  $n$  keys

$X_1, X_2, \dots, X_n$ .  $\alpha$  is the *loading factor* of the hash table and is given by  $\alpha = \frac{n}{b}$  where  $b$  is the number of buckets. Smaller the loading factor better is the average case performance of the hash table in comparison to that of linear lists.

## Other collision resolution techniques with open addressing

The drawbacks of linear probing or linear open addressing could be overcome to an extent by employing one or more of the following strategies:

(i) *Rehashing* A major drawback of linear probing is *clustering* or *primary clustering* wherein the hash table gives rise to long sequences of records with gaps in between the sequences. This leads to longer sequential searches especially when an empty slot needs to be found out. The problem could be resolved to an extent by resorting to what is known as *rehashing*. In this, a second hash function is used to determine the slot where the key is to be accommodated. If the slot is not empty, then another function is called for and so on.

Thus rehashing makes use of at least two functions  $H, H'$  where  $H(X), H'(X)$  map keys  $X$  to any one of the  $b$  buckets. To insert a key,  $H(X)$  is computed and the key  $X$  is accommodated in the bucket if it is empty. In the case of a collision, the second hash function  $H'(X)$  is computed and the search sequence for an empty slot proceeds by computing,

$$h_i = (H(X) + i \cdot H'(X)) \bmod b, \quad i=1, 2, \dots$$

Here  $h_1, h_2, \dots$  is the search sequence before an empty slot is found to accommodate the key. It needs to be ensured that  $H'(X)$  does not evaluate to 0, since there is no way this would be of help. A good choice for  $H'(X)$  is given by  $M - (X \bmod M)$  where  $M$  is chosen to be a prime smaller than the hash table size (see Illustrative Problem 13.6).

(ii) **Quadratic probing** This is another method that can substantially reduce clustering. In this method when a collision occurs at address  $h$ , unlike linear probing which probes buckets in locations  $h + 1, h + 2 \dots$  etc., the technique probes buckets at locations  $h + 1, h + 4, h + 9, \dots$  etc. In other words, the method probes buckets at locations  $(h + i^2) \bmod b$ ,  $i = 1, 2, \dots$  where  $h$  is the home bucket and  $b$  is the number of buckets. However, there is no guarantee that the method gives a fair chance to probe all locations in the hash table. Though quadratic probing reduces primary clustering, it may result in probing the same set of alternate cells. Such a case known as *secondary clustering* occurs especially when the hash table size is not prime.

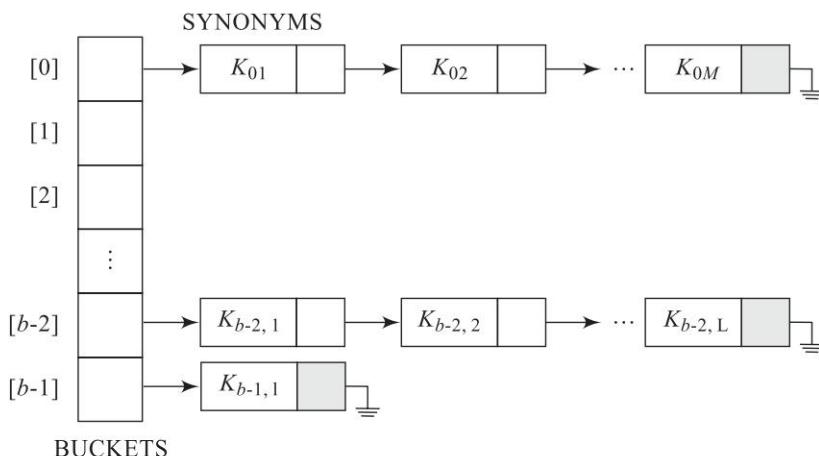
If  $b$  is a prime number then quadratic probing probes exactly half the number of locations in the hash table. In this case, the method is guaranteed to find an empty slot if the hash table is at least half empty (see Illustrative Problems 13.4, 13.5).

(iii) **Random probing** Unlike quadratic probing where the increment during probing was definite, random probing makes use of a random number generator to obtain the increment and hence the next bucket to be probed. However, it is essential that the random number generator function generates the same sequence. Though this method reduces clustering, it can be a little slow when compared to others.

## Chaining

## 13.5

In the case of linear open addressing, the solution of accommodating synonyms in the closest empty slot may contribute to a deterioration in performance. For example, the search for a synonym key may involve sequentially going through every slot occurring after its home bucket before it is either found or unfound. Also, the implementation of the hash table using a sequential data structure such as arrays, limits its capacity ( $b \times s$  slots). While increasing the number of slots to minimize overflows may lead to wastage of memory, containing the number of slots to the bare minimum may lead to severe overflows hampering the performance of the hash table. An alternative to overcome this malady is to keep all synonyms that are mapped to the same bucket chained to it. In other words, every bucket is maintained as a singly linked list with synonyms represented as nodes. The buckets continue to be represented as a sequential data structure as before and to favor the hash function computation. Such a method of handling overflows is called *chaining* or *open hashing* or *separate chaining*. Figure 13.6 illustrates a chained hash table.

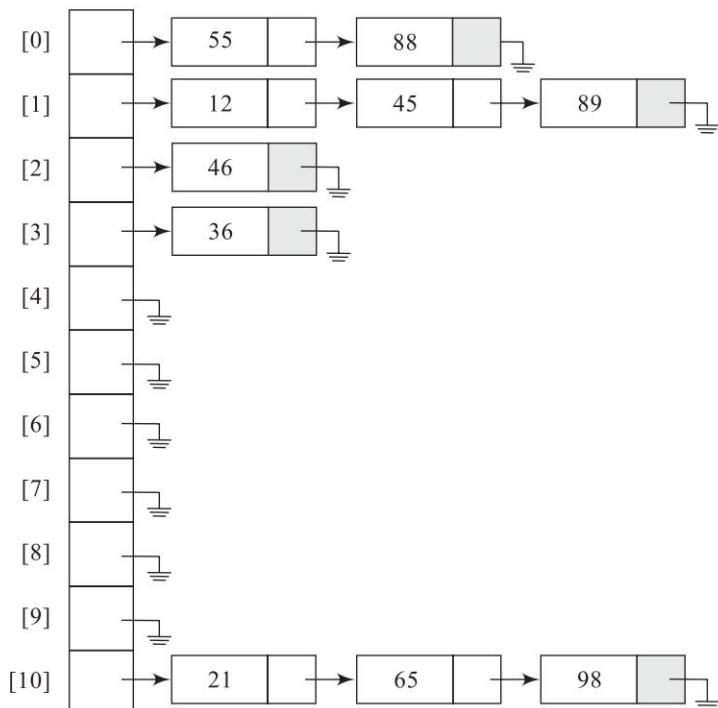


**Fig. 13.6** A chained hash table

In the Fig. 13.6, observe how the buckets have been represented sequentially and each of the buckets is linked to a chain of nodes which are synonyms mapping to the same bucket.

Chained hash tables only acknowledge collisions. There are no overflows per se since any number of collisions can be handled provided there is enough memory to handle them!

**Example 13.5** Let us consider the set of keys {45, 98, 12, 55, 46, 89, 65, 88, 36, 21} listed in Example 13.2, to be represented as a chained hash table. The hash function  $H$  used is  $H(X) = X \bmod 11$ . The hash function values for the keys are as shown in Table 13.1. The structure of the chained hash table is as shown in Fig. 13.7.



**Fig. 13.7 Hash table (Example 13.5)**

Observe how each of the groups of synonyms viz., {45, 12, 89}, {98, 65, 21} and {55, 88} are represented as singly linked lists corresponding to the buckets 1, 10 and 0 respectively. In accordance to the norms pertaining to singly linked lists, the link field of the last synonym in each chain is a null pointer. Those buckets which are yet to accommodate keys are also marked null.

## Operations on chained hash tables

**Search:** The search for a key  $X$  in a chained hash table proceeds by computing the hash function value  $H(X)$ . The bucket corresponding to the value  $H(X)$  is accessed and a sequential search along the chain of nodes is undertaken. If the key is found then the search is termed successful otherwise unsuccessful. If the chain is too long, maintaining the chain in order (ascending or descending) helps in rendering the search efficient.

Algorithm 13.2 illustrates the procedure to undertake search in a chained hash table.

**Algorithm 13.2:** Procedure to search for a key  $X$  in a chained hash table

```

procedure CHAIN_HASH_SEARCH(HT, b, X)
    /* HT[0:b-1] is the hash table implemented as a one
       dimensional array of pointers to buckets. Here b is the
       number of buckets. X is the key to be searched in the hash
       table. In case of unsuccessful search, the procedure prints
       the message "KEY not found" otherwise prints "KEY found"*/
h = H(X); /* H(X) is the hash function computed on X */
TEMP = HT[h]; /* TEMP is the pointer to the first node in the chain*/
while (DATA(TEMP) ≠ X and TEMP ≠ NIL) do /* search for the key
    down the chain*/
    TEMP = LINK(TEMP);
endwhile
if (DATA(TEMP) == X) then print (" KEY found");
if (TEMP == NIL) then print (" KEY not found");
end CHAIN_HASH_SEARCH.

```



**Insert:** To insert a key  $X$  into a hash table, we compute the hash function  $H(X)$  to determine the bucket. If the key is the first node to be linked to the bucket then all that it calls for, is a mere execution of a function to insert a node in an empty singly linked list. In the case of keys which are synonyms, the new key could be inserted either in the beginning or at the end of the chain leaving the list unordered. However, it would be prudent and less expensive too, to maintain each of the chains in the ascending or descending order of the keys. This would also render the search for a specific key amongst its synonyms to be efficiently carried out.

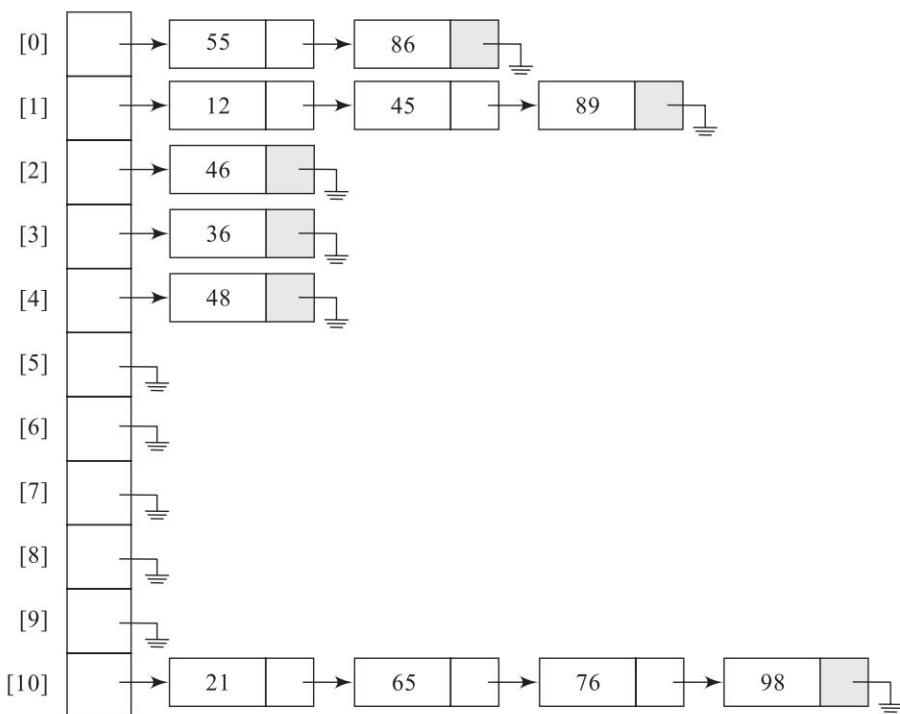
**Example 13.6** Let us insert keys {76, 48} into the chained hash table shown in Fig. 13.7. Since 76 has already three synonyms in its chain corresponding to bucket 10, we choose to insert it in order in the list. On the other hand 48 is the first key in its bucket viz., 4. Figure 13.8 illustrates the insertion.

Algorithm 13.2 could be modified to insert a key. It merely calls for the insertion of a node in a singly linked list that is unordered or ordered.

**Delete:** Unlike that of linear open addressed hash tables, the deletion of a key  $X$  in a chained hash table is elegantly done. All that it calls for, is a search for  $X$  in the corresponding chain and a deletion of the respective node.

## Performance Analysis

The complexity of the chained hash table is dependent on the length of the chain of nodes corresponding to the buckets. The best case complexity of a search is  $O(1)$ . A worst case occurs when all the  $n$  elements map to the same bucket and the length of the chain corresponding to that bucket is  $n$ , with the searched key turning out to be the last in the chain. The worst case complexity of the search in such a case is  $O(n)$ .



**Fig. 13.8** Inserting keys into a chained hash table

On an average, the complexity of the search operation on a chained hash table is given by

$$U_n \sim \frac{(1+\alpha)}{2}, \alpha \geq 1 \quad \text{and}$$

$$S_n \sim 1 + \frac{\alpha}{2},$$

where  $U_n$  and  $S_n$  are the number of nodes examined on an average during an unsuccessful and successful search respectively.  $\alpha$  is the loading factor of the hash table and is given by  $\alpha = \frac{n}{b}$  where  $b$  is the number of buckets.

The average case performance of the chained hash table is superior to that of linear open addressed hash table.

## Applications

## 13.6

In this section, we discuss the application of hash tables in the fields of compiler design, relational data base query processing and file organization.

### Representation of a keyword table in a compiler

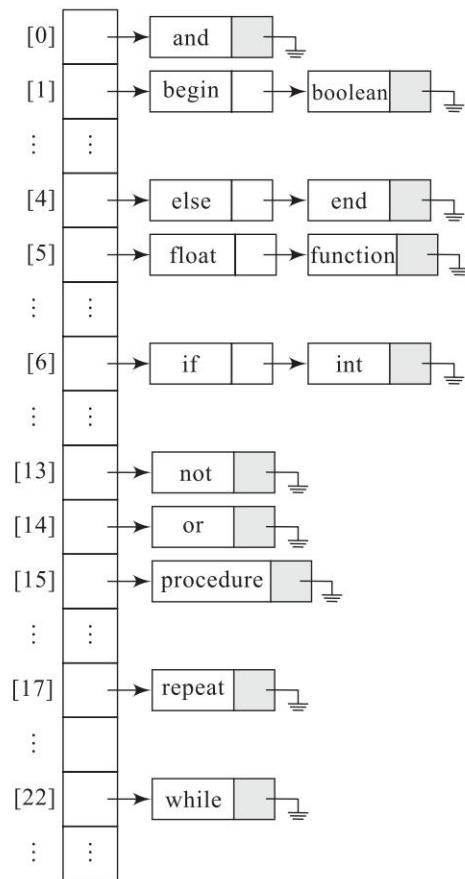
In Sec. 10.4, the application of binary search trees and AVL trees for the representation of symbol tables in a compiler were discussed. Hash tables find application in the same problem.

A keyword table which is a static symbol table is best represented by means of a hash table. Each time a compiler checks out on a string to be a keyword or a user-id, the string is searched against the keyword table. An appropriate hash function could be designed to minimize collisions amongst the keywords and yield the bucket where the keyword could be found. A successful search indicates that the string encountered is a keyword and an unsuccessful search indicates it is a user-id. Considering the significant fact that but for retrievals, no insertions or deletions are permissible on a keyword table, hash tables turn out to be one of the best propositions for the representation of symbol tables.

**Example 13.7** Consider a subset of a keyword set commonly used in programming languages, viz., {while, repeat, and, or, not, if, else, begin, end, function, procedure, int, float, Boolean}. For simplicity we make use of the hash function  $H(X) = ord(C) - 1$  where C is the first character of the keyword X. Figure 13.9 illustrates a linear open addressed hash table with two slots per bucket ( $HT[0..25, 0..1]$ ) and a chained hash table representation for the keyword set. Considering the efficient retrievals promoted by the hash table, the choice of the data structure for the symbol table representation contributes to the efficient performance of a compiler as well.

| Hash table<br>$HT[0..25, 0..1]$ |          |
|---------------------------------|----------|
| [0]                             | [1]      |
| and                             | ...      |
| begin                           | boolean  |
| :                               |          |
| else                            | end      |
| float                           | function |
| :                               | :        |
| if                              | int      |
| :                               | :        |
| not                             |          |
| or                              |          |
| procedure                       |          |
| :                               | :        |
| repeat                          |          |
| :                               | :        |
| while                           |          |
| :                               | :        |

(a) Linear open addressed hash table



(b) Chained hash table

**Fig. 13.9** Hash table representations for a keyword set

## Hash tables in the evaluation of a join operation on relational databases

Relational data bases support a selective set of operations viz., selection, projection, join (natural join, equi-join) and so on, which aid query processing. Of these, the *natural join* operation is most commonly used in relational data base management systems. Indicated by the notation  $\bowtie$ , the operation works on two relations (data bases) to combine them into a single relation. Given two relations  $R$  and  $S$  a natural join operation of the two data bases is indicated as  $R \bowtie S$ . The resulting relation is a combination of the two relations based on attributes common to the two relations.

**Example 13.8** Consider the two relations ITEM\_DESCRIPTION and VENDOR shown in Fig. 13.10(a). The ITEM\_DESCRIPTION relation describes the items and the VENDOR relation contains details about the vendors supplying the items. The relation ITEM\_DESCRIPTION contains the *attributes* ITEM\_CODE and ITEM\_NAME. The VENDOR relation contains the attributes ITEM\_CODE, VENDOR\_NAME, ADDRESS (city). A query pertaining to who the vendors are for a given item code calls for joining the two relations. The join of the two relations yields the relation shown in Fig. 13.10(b). Observe how the natural join operation combines the two relations on the basis of their common attribute ITEM\_CODE. Those *tuples* (rows) of the two relations having a common *attribute value* in the ITEM\_CODE field are “joined” together to form the output relation.

| Relation: ITEM_DESCRIPTION |                  | Relation: VENDOR |                     |            |
|----------------------------|------------------|------------------|---------------------|------------|
| ITEM_CODE                  | ITEM_NAME        | ITEM_CODE        | VENDOR_NAME         | ADDRESS    |
| P402                       | Pump.hp4-5-6     | P402             | Premier Electricals | Pune       |
| M636                       | Motor.621P       | M636             | Bharath Electronics | Coimbatore |
| S706                       | Stabilizer.VA500 | S706             | India Electricals   | Kolkata    |

(a)

| Relation: ITEM-DESCRIPTION $\bowtie$ VENDOR |                  |                     |            |
|---------------------------------------------|------------------|---------------------|------------|
| ITEM_CODE                                   | ITEM_NAME        | VENDOR_NAME         | ADDRESS    |
| P402                                        | Pump.hp4-5-6     | Premier Electricals | Pune       |
| M636                                        | Motor.621P       | Bharath Electronics | Coimbatore |
| S706                                        | Stabilizer.VA500 | India Electricals   | Kolkata    |

(b)

**Fig. 13.10** Natural join of two relations

One method of evaluating a join is to use the *hash method*. Let  $H(X)$  be the hash function where  $X$  is the attribute value of the relations. Here  $H(X)$  is the address of the bucket which contains the attribute value and a pointer to the appropriate tuple corresponding to the attribute value. The pointer to the tuple is known as *Tuple Identifier* (TID). TIDs in general, besides containing the physical address of the tuple of the relation, also hold identifiers unique to the relation. The hash tables are referred to as hash indexes in relational data base terminology.

A natural join of the two relations R and S over a common attribute ATTRIB, results in each bucket of the hash indexes recording the attribute values of ATTRIB along with the TIDs of the tuples in relations R and S whose R.ATTRIB = S.ATTRIB.

When a query associated with the natural join is to be answered all that it calls for is to access the hash indexes to retrieve the appropriate TIDs associated with the query. Retrieving the tuples using the TIDs satisfies the query.

**Example 13.9** Figure 13.11(a) shows a physical view of the two relations ITEM\_DESCRIPTION and VENDOR. Figure 13.11(b) shows the hash function values based on which the hash table (Fig. 13.11(c)) has been constructed. The hash function used is not discussed. Each bucket of the hash index records the TIDs of the attribute values mapped to the bucket. Thus TIDs corresponding to ITEM\_CODE = P402 of both the relations, are mapped to bucket 16 and so on.

| ITEM_DESCRIPTION |           |                  | VENDOR |           |                     |
|------------------|-----------|------------------|--------|-----------|---------------------|
| TID              | ITEM_CODE | ITEM_NAME        | TID    | ITEM_CODE | VENDOR_NAME         |
| 4001             | P402      | Pump.hp4-5-6     | 7001   | P402      | Premier Electricals |
| 4002             | M636      | Motor.621P       | 7002   | M636      | Bharath Electronics |
| 4003             | S706      | Stabilizer.VA500 | 7003   | S706      | India Electricals   |

(a) A physical view of the relations ITEM\_DESCRIPTION and VENDOR

|      |      |      |      |
|------|------|------|------|
| X    | P402 | M636 | S706 |
| H(X) | 16   | 5    | 14   |

(b) Hash function values of the ITEM\_CODE attribute values

| Buckets | Slots       |             |   |
|---------|-------------|-------------|---|
|         | [1]         |             |   |
| ⋮       | ⋮           | ⋮           | ⋮ |
| [5]     | M636 > 4002 | M636 > 702  |   |
| ⋮       | ⋮           | ⋮           | ⋮ |
| [14]    | S706 > 4003 | S706 } 7003 |   |
|         |             |             |   |
| [16]    | P402 } 4001 | P402 } 7001 |   |
|         |             |             |   |

(c) Hash table

**Fig. 13.11** Evaluation of natural join operation using hash indexes

Assume that a query “List the vendor(s) supplying the item P402” is to be processed. To process this request, we first compute  $H(\text{"P402"})$  which as shown in Fig. 13.11(b) yields the bucket address 16. Accessing bucket 16 we find the TID corresponding to the relation VENDOR is 7001. To answer the query, all that needs to be done is to retrieve the tuple whose TID is 7001.

A general query such as “List the vendors supplying each of the items” may call for sequentially searching each of the hash indexes corresponding to each attribute value of ITEM\_CODE.

## Hash tables in a direct file organization

*File organization* deals with methods and techniques to structure data in *external* or *auxiliary storage* devices such as tapes, disks, drums etc. A *file* is a collection of related data termed as *records*. Each record is uniquely identified by what is known as a *key*, which is a datum or a portion of data in the record. The major concern in all these methods is regarding the access time when records pertaining to the keys (primary or secondary) are to be retrieved from the storage devices to be updated, inserted or deleted. Some of the commonly used file organization schemes are sequential file organization, serial file organization, indexed sequential access file organization and direct file organization. Chapter 14 elaborately details on files and their methods of organization.

The direct file organization (see Sec. 14.8) which is a kind of file organization method, employs hash tables for the efficient storage and retrieval of records from the storage devices. Given a file of records,  $\{f_1, f_2, f_3, \dots, f_N\}$  with keys  $\{k_1, k_2, k_3, \dots, k_N\}$  a hash function  $H(k)$  where  $k$  is the record key, determines the storage address of each of the records in the storage device. Thus direct files undertake direct mapping of the keys to the storage locations of the records with the records of the file organized as a hash table.



## Summary

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

- A linked implementation of a hash table is known as chaining. In this all the synonyms are chained to their respective buckets as a singly linked list. On an average, a chained hash table is superior in performance when compared to that of a linear probed hash table
- Hash tables have found applications in the design of symbol tables in compiler design, query processing in relational database management systems and direct file organization.



## Illustrative Problems

**Problem 13.1** Insert the following data into a hash table implemented using linear open addressing. Assume the buckets to have 3 slots each. Make use of the hash function  $h(X) = X \bmod 9$ .

{ 17, 09, 34, 56, 11, 71, 86, 55, 22, 10, 4, 39, 49, 52, 82, 13, 40, 31, 35, 28, 44 }

**Solution:** The linear open addressed hash table is shown in Fig. I 13.1 Those keys not accommodated in their home buckets are shown in shaded background.

**Problem 13.2** For the set of keys listed in Illustrative Problem 13.1, trace a chained hash table making use of the same hash function.

**Solution:** The chained hash table is shown in Fig. I 13.2. The nodes in the chain are inserted in the ascending order.

| HT      | Slots |     |     |
|---------|-------|-----|-----|
|         | [0]   | [1] | [2] |
| Buckets | 9     | 44  |     |
|         | 55    | 10  | 82  |
|         | 56    | 11  | 28  |
|         | 39    |     |     |
|         | 22    | 4   | 49  |
|         | 86    | 13  | 40  |
|         | 31    |     |     |
|         | 34    | 52  |     |
|         | 17    | 71  | 35  |

Fig. I 13.1

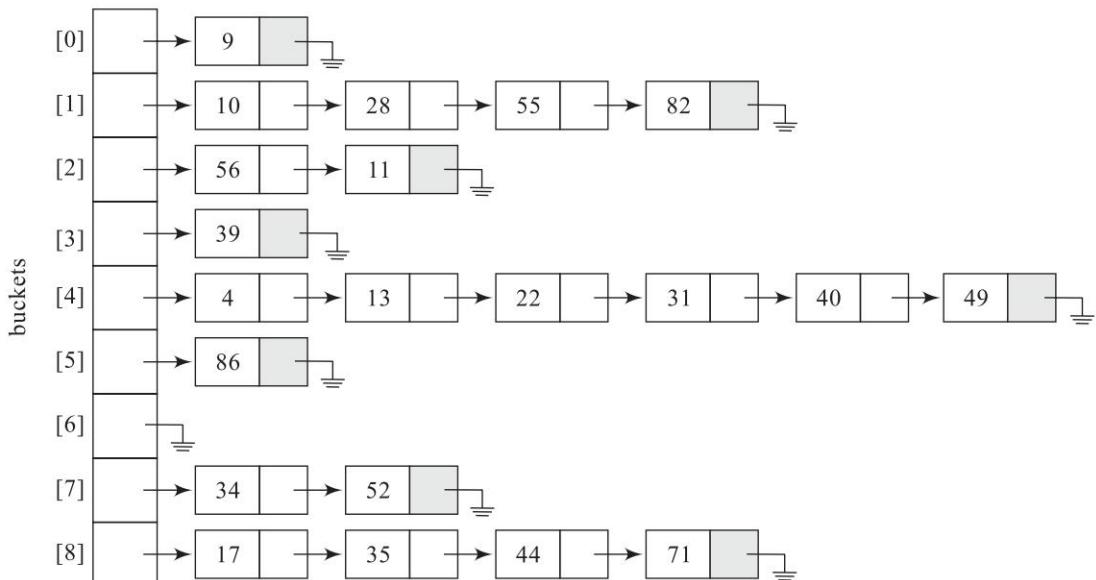


Fig. I 13.2

**Problem 13.3** Comment on the statement: "To minimize collisions in a linear open addressed hash table it is recommended that the ratio of the number of buckets in a hash table to the number of keys to be stored in the hash table is made bigger"

**Solution:** No, this is illogical since increasing the number of buckets will only lead to wastage of space.

**Problem 13.4** For the set of keys { 17, 9, 34, 56, 11, 4, 71, 86, 55, 10, 39, 49, 52, 82, 31, 13, 22, 35, 44, 20, 60, 28} obtain a hash table following quadratic probing. Make use of the hash function  $H(X) = X \bmod 9$ . What are your observations?

**Solution:** Quadratic probing employs the function  $(h + i^2) \bmod n$ ,  $i = 1, 2, \dots$  where  $n = 9$ , to determine the empty slot during collisions. Here  $h$  is the address of the home bucket given by the hash function  $H(X)$ , where  $X$  is the key. The quadratic probed hash table is as shown in Fig. I 13.4.

Note how during the insertion of keys 13 and 22, their home buckets viz., 4 is full. To handle this collision, quadratic probing begins searching buckets  $4+1 \bmod 9, 4+2^2 \bmod 9, \dots$ . Since the first searched bucket 5 has empty slots the keys find accommodation there. However, in the case of key 44, to handle its collision with bucket 8, quadratic probing searches for an empty slot as ordered by the sequence,  $8+1 \bmod 9, 8+2^2 \bmod 9, \dots$ . The search for an empty slot is successful when the bucket  $8+1^2 \bmod 9$  is encountered. 44 is accommodated in slot 2 of the bucket 0.

The case of inserting key 28 is interesting, for, despite the hash table containing free slots, quadratic probing is unable to find an empty slot to accommodate the key. The sequence searched for is  $1+1 \bmod 9, 1+2^2 \bmod 9, 1+3^2 \bmod 9, \dots$

An important observation regarding quadratic probing is that there is no guarantee of finding an empty slot in a quadratic probed hash table if the hash table size is not prime. In this example the hash table size is not prime.

**Problem 13.5** For the set of keys listed in Illustrative Problem 13.4, obtain a hash table following quadratic probing and employing the hash function  $H(X) = X \bmod 11$ . What are your observations?

**Solution:** The quadratic probed hash table for the given set of keys using the hash function is shown in Fig. I 13.5.

An important observation regarding this example is that quadratic probing can always find an empty slot to insert a key if the hash table size is prime and the table is at least half empty.

**Problem 13.6** For the set of keys { 11, 55, 13, 35, 71, 52, 61, 9, 86, 31, 49, 85, 70} obtain the hash table which employs rehashing for collision resolution. Assume the hash function to be  $H(X) = X \bmod 9$  and the rehashing function to be  $H'(X) = 7 - (X \bmod 7)$ . The collision resolution function is given by  $h_i = (H(X) + i \cdot H'(X)) \bmod b$ ,  $i=1, 2, \dots$ .

**Solution:** The hash table for the problem is shown below:

Observe how during the insertion of key 49 a collision occurs and its bucket 4 ( $H(49)=49 \bmod 9 = 4$ ) is found to be full. Rehashing turns to the next hash function  $H'(49) = 7 - (49 \bmod 7)$  to

| HT  | [0] | [1] | [2] |
|-----|-----|-----|-----|
| [0] | 9   | 44  |     |
| [1] | 55  | 10  | 82  |
| [2] | 56  | 11  | 20  |
| [3] | 39  |     |     |
| [4] | 4   | 49  | 31  |
| [5] | 86  | 13  | 22  |
| [6] | 60  |     |     |
| [7] | 34  | 52  |     |
| [8] | 17  | 71  | 35  |

28 fails to find an empty slot

Fig. I 13.4

| HT   | [0] | [1] | [2] |
|------|-----|-----|-----|
| [0]  | 11  | 55  | 22  |
| [1]  | 34  | 56  | 44  |
| [2]  | 13  | 35  |     |
| [3]  |     |     |     |
| [4]  | 4   |     |     |
| [5]  | 71  | 49  | 82  |
| [6]  | 17  | 39  | 60  |
| [7]  | 28  |     |     |
| [8]  | 52  |     |     |
| [9]  | 9   | 86  | 31  |
| [10] | 10  | 20  |     |

Fig. I 13.5

## Hash Tables

help obtain the empty slot to accommodate the key. The slot searched is  $h_1 = (H(49) + 1, H(49)) \bmod 9 = 2$ . Since the bucket contains a vacant slot, key 49 is accommodated in the slot.

In the case of key 85 which once again collides with the keys in bucket 4, rehashing computes  $H(85) = 6$  and  $h_1 = (H(85) + 1, H(85)) \bmod 9 = 1$ . Key 85 is accommodated in bucket 1 slot 2. Finally, following similar lines, key 70 is accommodated in bucket 5.

**Problem 13.7** Assume a chained hash table in which each of the chain is implemented as a binary search tree rather than a singly linked list. Build such a hash table for the keys { 9, 10, 6, 20, 14, 16, 5, 40, 4, 2, 7, 3, 8} using the hash function  $H(X) = X \bmod 5$  where  $X$  is the key. What are the advantages of adopting this system?

**Solution:** A chained hash table with each of the chains implemented as a binary search tree is shown in Fig. I 13.7.

The advantage is that during the search operation, the binary search tree based chains would record  $O(\log n)$  performance. In contrast a linear chain would report  $O(n)$  complexity on an average.

| HT  | [0] | [1] |
|-----|-----|-----|
| [0] | 9   |     |
| [1] | 55  | 85  |
| [2] | 11  | 49  |
| [3] |     |     |
| [4] | 13  | 31  |
| [5] | 86  | 70  |
| [6] |     |     |
| [7] | 52  | 61  |
| [8] | 35  | 71  |

Fig. I 13.6

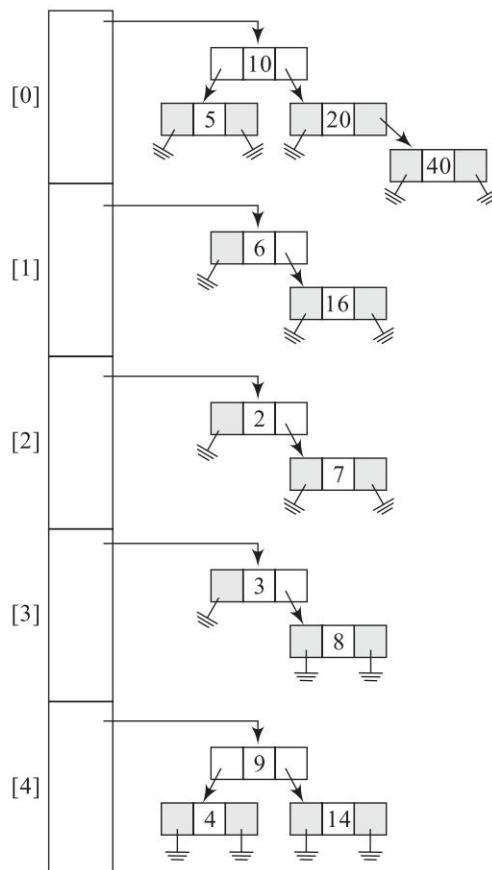


Fig. I 13.7

**Problem 13.8** Fill in Table I 13.8(a) with the number of comparisons made, when the elements shown in row 1 of the table (<{ 66, 100, 55, 3, 99, 144}) are either successfully or unsuccessfully searched over the list of elements { 66, 42, 96, 100, 3, 55, 99} when the latter is represented as (i) sequential list (ii) binary search tree and (iii) linear probing based hash table with single slot buckets using the hash function  $h(X) = X \bmod 7$ .

**Table I 13.8(a)**

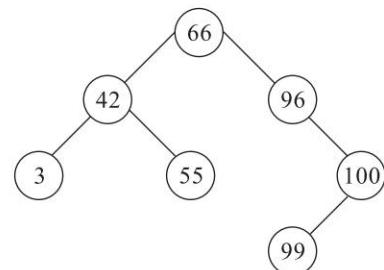
| Representation of data elements | Number of Comparisons |     |    |   |    |     |
|---------------------------------|-----------------------|-----|----|---|----|-----|
|                                 | 66                    | 100 | 55 | 3 | 99 | 144 |
| Sequential list                 |                       |     |    |   |    |     |
| Binary search tree              |                       |     |    |   |    |     |
| Hash table                      |                       |     |    |   |    |     |

**Solution:** Representing the elements of the list to be searched as a sequential list, yields {3, 42, 55, 66, 96, 99, 100}. The number of comparisons made for searching 66 is 4 and that for 144 which is an unsuccessful search is 7.

Representation of the elements in the list as a binary search tree is given in Fig. I 13.8. The number of comparisons made for the element 66 is 1 and that for 144 is 3.

Representation of the elements as a linear probed hash table with single slot bucket is shown below. The hash function used is  $h(X) = X \bmod 7$ . The data element displaced from the home bucket is shown over grey background. The number of comparisons made for the element 66 is 1 and that for 144 is 7.

|     |     |
|-----|-----|
| [0] | 42  |
| [1] | 99  |
| [2] | 100 |
| [3] | 66  |
| [4] | 3   |
| [5] | 96  |
| [6] | 55  |

**Fig. I 13.8**

The comparisons for the rest of the elements is shown in Table I 13.8(b).

**Table I 13.8(b)**

| Representation of data elements | Number of Comparisons |     |    |   |    |     |
|---------------------------------|-----------------------|-----|----|---|----|-----|
|                                 | 66                    | 100 | 55 | 3 | 99 | 144 |
| Sequential list                 | 4                     | 7   | 3  | 1 | 6  | 7   |
| Binary search tree              | 1                     | 3   | 3  | 3 | 4  | 3   |
| Hash table                      | 1                     | 1   | 1  | 2 | 1  | 7   |



## Review Questions

1. Hash tables are ideal data structures for \_\_\_\_\_  
 (a) dictionaries      (b) graphs      (c) trees      (d) none of these

2. State whether true or false:

In the case of linear open addressed hash table with multiple slots in a bucket,

(i) overflows always mean collisions, and

(ii) collisions always mean overflows

(a) (i) true (ii) true    (b) (i) true (ii) false    (c) (i) false (ii) true    (d) (i) false (ii) false

3. In the context of building hash functions, find the odd term out in the following list:

Folding, modular arithmetic, truncation, random probing

(a) folding      (b) modular arithmetic  
 (c) truncation      (d) random probing

4. In the case of a chained hash table of  $n$  elements with  $b$  buckets, assuming that a worst case resulted in all the  $n$  elements getting mapped to the same bucket, then the worst case time complexity of a search on the hash table would be given by

(a)  $O(1)$       (b)  $O(n/b)$       (c)  $O(n)$       (d)  $O(b)$

5. Match the following:

(A) rehashing      (i) collision resolution

(B) folding      (ii) hash function

(C) linear probing

(a) (A, (i) )      (B, (ii) )      (C, (ii) )

(b) (A, (ii) )      (B, (ii) )      (C, (i) )

(c) (A, (ii) )      (B, (i) )      (C, (i) )

(d) (A, (i) )      (B, (ii) )      (C, (i) )

6. What are the advantages of using modulo arithmetic for building hash functions?

7. How are collisions handled in linear probing?

8. How are insertions and deletions handled in a chained hash table?

9. Comment on the search operation for a key  $K$  in a list  $L$  represented as (i) sequential list (ii) a chained hash table and (iii) linear probed hash table

10. What is rehashing? How does it serve to overcome the drawbacks of linear probing?

11. The following is a list of keys. Making use of a hash function  $h(k) = k \bmod 11$ , represent the keys in a linear open addressed hash table with buckets containing (i) 3 slots and (ii) 4 slots.

090 890 678 654 234 123 245 678 900 111 453 231 112 679 238 876 009 122  
 233 344 566 677 899 909 512 612 723 823 956 221 331 441 551

12. For the problem in Review Questions 11 (Chapter 13), resolve collisions by means of (i) rehashing that makes use of an appropriate rehashing function and (ii) quadratic probing.

13. For the problem in Review Questions 11 (Chapter 13), implement a chained hash table.



## Programming Assignments

1. Implement a hash table using an array data structure. Design functions to handle overflows using (i) linear probing (ii) quadratic probing and (iii) rehashing. For a set of keys observe the performance when the methods listed above are executed.
2. Implement a hash table for a given set of keys using chaining method of handling overflows. Maintain the chains in the ascending order of the keys. Design a menu driven front end to perform the insert, delete and search operations on the hash table.
3. The following is a list of binary keys:  
0011, 1100, 1111, 1010, 0010, 1011, 0111, 0000, 0001, 0100, 1000, 1001, 0011.  
Design a hash function and an appropriate hash table to store and retrieve the keys efficiently. Compare the performance when the set is stored as a sequential list.
4. Store a dictionary of limited set of words as a hash table. Implement a spell check program which given an input text file will check for the spelling using the hash table based dictionary and in the case of misspelled words will correct the same.
5. Let TABLE\_A and TABLE\_B be two files implemented as a table. Design and implement a function JOIN (TABLE\_A, TABLE\_B) which will “natural join” the two files as discussed in Sec. 13.6. Make use of an appropriate hash function.



# FILE ORGANIZATIONS 14

## Introduction

## 14.1

One of the main components of a computer system is the *memory*, also referred to as the *main memory* or the *internal memory* of the computer. Memory is a storage repository of data that is used by the CPU during its processing.

When the CPU has to process voluminous data, the computer system has to take recourse to *external memory* or *external storage* to store the data, due to the limited capacity of the internal memory. The devices which provide support for external memory are known as *external storage devices* or *auxiliary storage devices*. Given that the main memory of the computer is the *primary memory*, the external memory is also referred to as *secondary memory* and the devices as *secondary storage devices*. Examples of secondary storage devices are magnetic tapes, magnetic disks, drums, floppies etc. While internal memory of a computer system is *volatile*, meaning that data may be lost when the power goes off, secondary memory is *nonvolatile*.

Each of the secondary storage devices have their distinct characteristics. Magnetic tapes, built on the principle of audio tape devices, are *sequential storage devices* that store data sequentially. On the other hand, magnetic disks, drums and floppy diskettes are *random access storage devices* that can store and retrieve data both sequentially and randomly. The random access storage devices are also known as *direct access storage devices*. Section 17.2 elaborately discusses the structure and configuration of magnetic tapes and disks.

The growing demands of information have called for the support of what are known as *tertiary storage devices*. Though these devices are capable of storing huge volumes of data running to terabytes, at lesser costs, are characterized by significantly higher read/write time when compared to that of secondary storage devices. Examples of tertiary storage devices are optical disk juke boxes, ad hoc tape storage and tape silos.

The organization of data in the internal memory calls for an application of both sequential and linked data structures. In the same vein, the organization of data in the secondary memory also calls for a number of strategies for their efficient storage and retrieval. The organization of data in the secondary memory is known as *files*.

14.1 *Introduction*

14.2 *Files*

14.3 *Keys*

14.4 *Basic file operations*

14.5 *Heap or Pile organization*

14.6 *Sequential file organization*

14.7 *Indexed sequential file organization*

14.8 *Direct file organization*

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

## Files

## 14.2

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

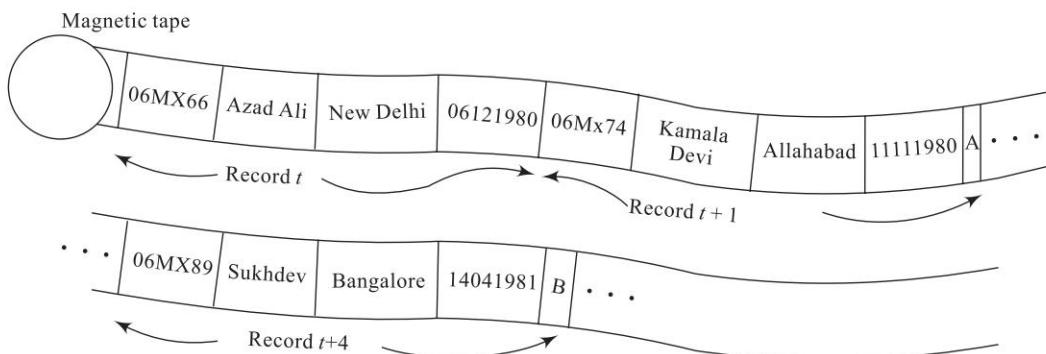
Let us consider the example of a student file. The file is a logical collection of student records. A student record is a collection of fields such as *roll number*, *name*, *city*, *date of birth*, *grade* etc. Each of these fields could be numeric, alphabetic or alphanumeric. A sample set of student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

### Student file

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

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

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



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

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

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

## Keys

## 14.3

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

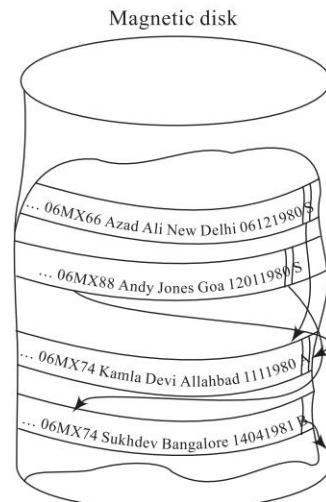
If additional fields were added to the primary key, the combination would still continue to uniquely identify the record. Such a combination of fields is referred to as a *super key*. For example, the combination of `roll number` and name would still continue to uniquely identify records in the student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

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

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

### Employee file

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



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

## Basic File Operations

14.4

Some of the basic file operations are

- (i) *open*, which prepares the files concerned, for reading or writing. Commonly a file pointer is opened and set at the beginning of the file that is to be read or written.
- (ii) *read*, when the contents of the record pointed to by the file pointer, is read.
- (iii) *insert*, when new records are added to the file.
- (iv) *delete*, when existing records are removed from the file.
- (v) *update*, when data in one or more fields in the existing records of the files are modified.
- (vi) *reset*, when the file pointer is set to the beginning of the file.
- (vii) *close*, when the files that were opened for operations are closed for access.

Commercial implementations of programming languages provide a variety of other file operations. However, from the data structure stand point the operations of insert, delete and update are considered significant and therefore we shall restrict our discussion to these operations alone.

In the case of deletion, the operation could be executed logically or physically as determined by the application. In the case of *physical deletion* the records are physically removed from the file. On the other hand, in the case of *logical deletion*, the record is either ‘flagged’ or ‘marked’ to indicate that it is not in use. Every record has a bit or a byte called the *deletion marker* which is set to some value indicating deletion of the record. Though the records are physically available in the file, they are logically excluded from consideration during file processing. The logical deletion also facilitates restoration of the deleted records which could be done by “unflagging” or “unmarking” the records.

In the case of the student file, addition of details pertaining to new students could call for insertion of appropriate student records. Students opting to drop out of the programme could have their records ‘logically’ or ‘physically’ deleted. Again, a change of address or a change in grades after revaluation, could call for updating the relevant fields of the record.

## Heap Or Pile Organization

14.5

The heap or pile organization is one of the simplest of the file organizations. These are non-keyed sequential files. The records are maintained in no particular order. The insert, delete and update operations are undertaken as described below. This unordered file organization is basically suitable for instances where records are to be collected and stored for future use.

### Insert, delete and update operations

**Insert:** To insert records into the heap or pile, the records are merely appended to the file.

**Delete:** To delete records, either a physical deletion or logical deletion is done. In the case of physical deletion, either the record is physically deleted or the last record is brought forward to replace the deleted one. This indeed calls for several accesses.

**Update:** To retrieve a record for updating it, entails a linear search of the file which in the worst case, could call for a search from beginning to end of the file.

## Sequential File Organisation

14.6

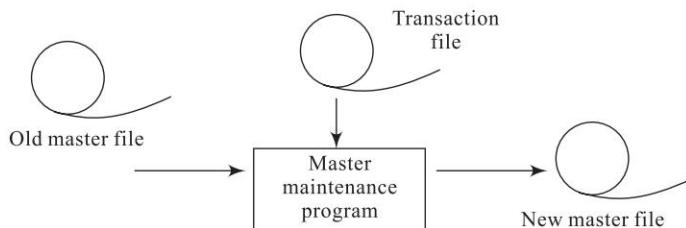
Sequential files are ordered files maintained in a logical sequence of primary keys. The organization was primarily meant to satisfy the characteristics of magnetic tapes which are sequential devices.

### Insert, delete and update operations

A sequential file is stored in the same logical sequence of its records, on the tape. Thus the physical and logical organization of sequential files are one and the same. Since random access is difficult on a tape, the handling of insert, delete and update operations could turn out to be expensive if they are handled on an individual basis. Therefore a batched mode of these operations is undertaken.

For a sequential file  $S$ , those records which are to be inserted, deleted or updated are written on to a separate tape as a sequential file  $T$ . The file  $T$ , known as the *transaction file*, is ordered according to its primary keys. Here  $S$  is referred to as the *master file*. With both  $S$  and  $T$  ordered according to their primary keys, a maintenance program reads the two files and while undertaking a "merge" operation executes the relevant operation( insert / delete / update), in parallel. The updated file is available on an output tape.

During the merge operation, in the case of insert operation, the new records merely get copied on to the output tape in the primary key ordering sequence. In the case of delete operation, the corresponding records are stopped from getting copied on to the output tape and are just skipped. For update operation, the appropriate fields are updated and the modified records are copied on to the output tape. Figure 14.3 illustrates the master maintenance procedure.



**Fig. 14.3 Master file maintenance**

The new file that is available on the output tape is referred to as the *new master file*  $S^{\text{new}}$ . The advantage of this method is that it leaves a back up of the master file before it is updated. The file  $S$  at this point gets referred to as the *old master file*. In fact it is common to retain an ancestry of back up files depending on the needs of the application. In such a case, while the new master file would be referred to as the *son file*, the old master file would be referred to as the *father file* and the older master file as the *grandfather file* and so on.

### Making use of overflow blocks

Since the updating of the file calls for a creation of a new file each time, an alternative could be to store the records in blocks with 'holes' in them. The 'holes' are vacant spaces at the tail end of the blocks.

Insertions are accommodated in these 'holes'. If there is no space to accommodate insertions in the appropriate blocks, the records are accommodated in special blocks called *overflow* blocks.

Although this method renders insert operations to be efficient, retrievals could call for a linear search of the whole file. In the case of deletions it would be prudent to adopt logical deletions. However, when the number of logical deletions increase or when the over flow blocks are fast filling up, it is advisable to reorganize the entire file.

## Indexed Sequential File Organization

14.7

While sequential file organizations provide efficient sequential access to data, random access to records are quite cumbersome. Indexed sequential file organizations are hybrid organizations which provide efficient sequential as well as random access to data. The method of storage and retrieval, known as *Indexed Sequential Access Method* (ISAM) makes use of *indexes* to facilitate random access of data, while the data themselves are maintained in a *sequential* order. The files following the ISAM method of storage and retrieval are also known as *ISAM files*.

### Structure of the ISAM files

An ISAM file consists of

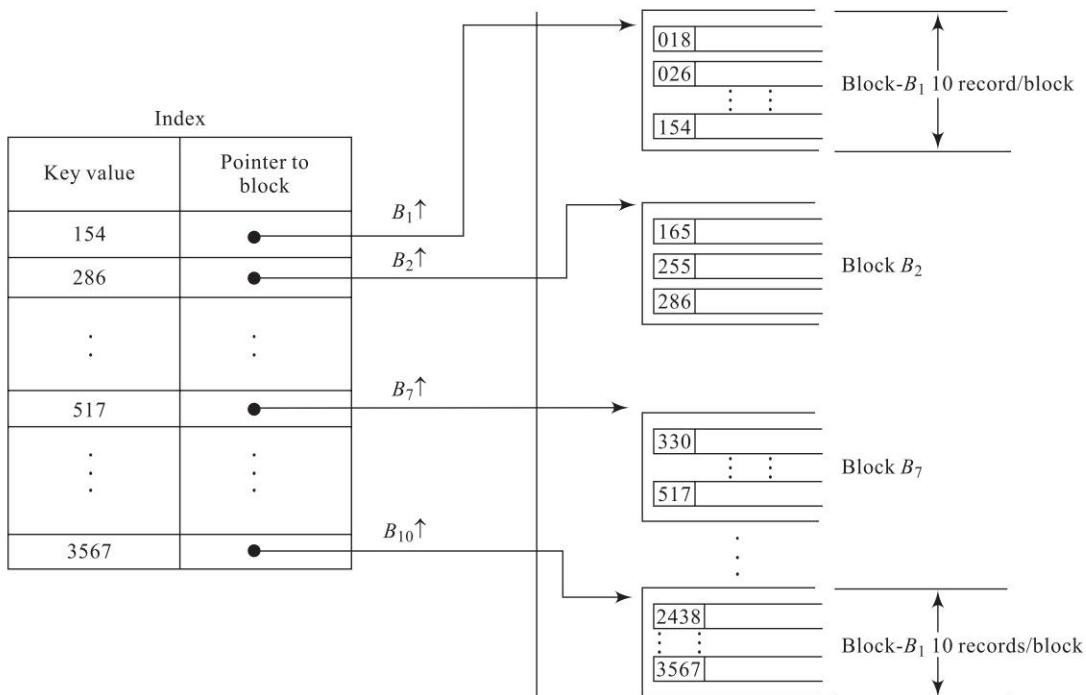
- (i) a *primary storage area*, where the data records of the file are sequentially stored,
- (ii) a *hierarchy of indexes*, where an *index* is a directory of information pertaining to the physical location of the records and
- (iii) *overflow area(s)* or *block(s)*, where new records to be added to the file and which could not be accommodated in the primary storage area, are stored.

Though ISAM files provide efficient retrieval of records, the operations of insertion and deletion can get quite involved and need to be efficiently handled. There are many methods to maintain indexes and efficiently handle insertions and deletions in the primary storage as well as overflow areas.

The primary storage area is divided into *blocks* where each block can store a definite number of records. The data records of the ISAM file are distributed on to the blocks in the logical order of their sequence, one block after the other. Thus all records stored in a block  $B_t$  have their keys to be greater than or equal to those of the records stored in the previous block  $B_{t-1}$ .

The index is a two dimensional table with each entry indicative of the physical location of the records. An index entry is commonly a key - address pair,  $(K, B \uparrow)$  where  $K$  is the key of the record and  $B \uparrow$  is a pointer to the block containing the record or sometimes a pointer to the record itself. An index is always maintained in the sorted order of the keys  $K$ . If the index maintains an entry for each record of the file then the index is an  $N \times 2$  table where  $N$  is the size of the file. In such a case the index is said to be a *dense index*. In the case of large files, the processing time of dense indexes can be large due to the huge amount of entries in them. To reduce the processing time, one could devise strategies so that only one entry per block is made in the index. In such a case the index is known as a *sparse index*. Commonly the entry could be pertaining to a special record in the block known as the *block anchor*. The block anchor could be either the first record (smallest key) or the last record (largest key) of the block. If the file occupies  $b$  blocks the size of the index would be  $b \times 2$ .

**Example 14.1** Figure 14.4 illustrates a schematic diagram of a simple ISAM file. The records of the file are stored sequentially in the ascending order of their primary keys. The file occupies 10 blocks each comprising 100 records. The last record of each block is chosen as the block anchor. Observe how the index maintains entries for each of the block anchors alone. The entries in the index are sorted according to the key values.



**Fig. 14.4** Schematic diagram of a naïve ISAM file

### Insert, Delete and Update operations for a simple ISAM file

The insert, delete and update operations for a simple ISAM file are introduced here. However it needs to be recollected that a variety of methods exist for maintaining indexes, each of which command their exclusive methods of operations.

**Insert** To insert records, the records are to be first inserted in the primary storage area. The existing records in a block may have to be moved to make space for the new records. This in turn may call for a change in the index entries especially if the block anchors get shifted due to the insertions.

A simple solution would be to provide 'holes' in the blocks where new records could be inserted. However the possibility of blocks overflowing cannot be ruled out. In such a case the new records are pushed into the overflow area, which merely maintains the records as an unordered list. Another option would be to maintain an overflow area for each block, as a sorted linked list of records.

**Delete** The most convenient way to handle deletions is to undertake logical deletions making use of deletion markers.

**Update** A retrieval of record for update, is quite efficiently done in an ISAM file. To retrieve a record with key  $K'$ , we merely undertake a linear search (or even binary search) of the index table to find that entry  $(K, B \uparrow)$  such that  $K' \leq K$ . Following the pointer  $B \uparrow$ , we linearly search the block of records to retrieve the desired record. However in the case of the record being available in the over flow blocks, the search procedure can turn out to be a bit more complex.

For example, in the ISAM file structure shown in Fig. 14.4, to retrieve the record with key 255, we merely search the index to find the appropriate entry  $(286, B_2 \uparrow)$ , where  $B_2 \uparrow$  is the pointer to the block  $B_2$ . A linear search of the key 255 in block  $B_2$  retrieves the relevant record.

## Types of indexing

There are many methods of indexing files. Commonly, all methods of indexes make use of a single key field based on which the index entries are maintained. The key field is known as *indexing field*. A few of the indexing techniques are detailed here.

**Primary indexing** This is one of the most common forms of indexing. The file is sorted according to its primary key. The *primary index* is a sparse index that contains the pair  $(K, B \uparrow)$  as its entries, where  $K$  is the primary key of the block anchor and  $B \uparrow$  is the pointer to the block concerned. The indexing method used in the ISAM file illustrated in Example 14.1, is in fact primary indexing. The general operations of insert, delete and update discussed in Sec. 14.7 hold good for primary indexing based ISAM files.

## Multilevel indexing

In the case of voluminous files, despite employing sparse indexes, searching through the index can itself become an overhead due to the large amount of entries in the index. In such a case to cut down the search time, a hierarchy of indexes also known as *multilevel indexes*, is constructed. Multilevel indexes are but index over indexes.

Example 14.2 discusses an ISAM file based on multilevel indexing. It can be seen that while the lowest level index points to the physical location of the blocks, the rest of the indexes point to their lower level indexes. To retrieve a record with key  $K$ , we begin from the highest level index and work our way down the indexes until the appropriate block is reached. A linear search of the block yields the record.

**Example 14.2** Figure 14.5 illustrates an ISAM file with multilevel indexing. The file has 10,000 records and is stored in a sequential fashion. 400 blocks, each holding 25 records make up for the primary storage area. The file organization shows three levels of indexing. Observe how each of the higher level indexes are indexes over the lower level indexes. To search for a key  $K$  we begin from the highest level index and follow the pointers to the lower level indexes. At the lowest level index we obtain the block address from which the record could be searched out.

**Cluster indexing** Typically, ISAM files have their records ordered sequentially according to the primary key values. It is possible that the records are ordered sequentially according to some non-key field that can carry duplicate values. In such a case the indexing field which is the non-key field is called as the *clustering field* and the index is called as the *cluster index*.

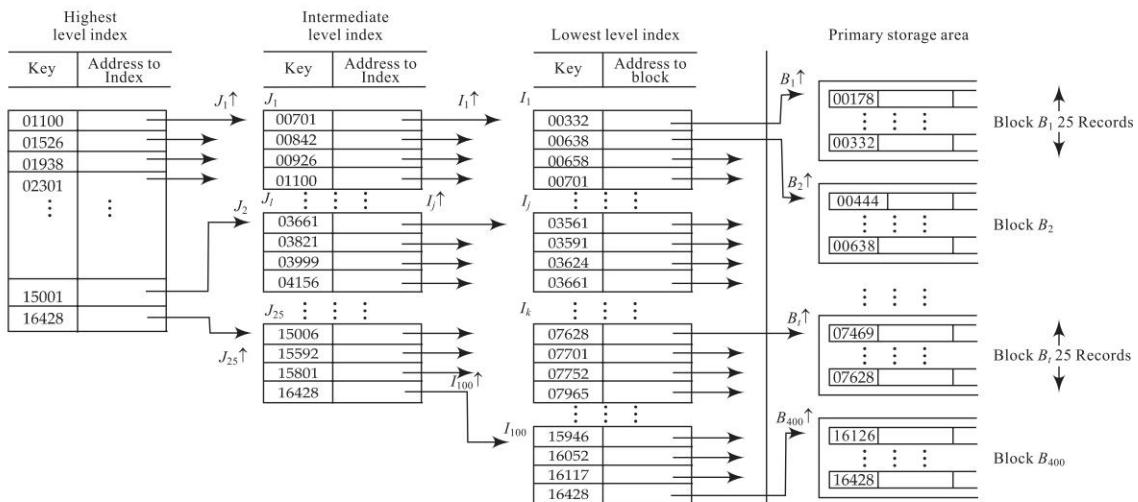


Fig. 14.5 An ISAM file based on multilevel indexing

Cluster index is a sparse index. As all other sparse indexes, a cluster index is also made up of entries of the form  $(I, B\uparrow)$ , where  $I$  is the clustering field value and  $B\uparrow$  is the block address of the appropriate record. For a group of records with the same value for  $I$ ,  $B\uparrow$  indicates the block address of the first record in the group. The rest of the records in the group may be easily accessed by making a forward search in the block concerned, since the records in the primary storage area are already sorted according to the non-key field.

However, in the case of cluster indexing, the insert / delete operations as before can become complex, since the data records are physically ordered. A straight forward strategy for efficient handling of insert / delete operations would be to maintain the block or cluster of blocks in such a way that all records holding the same value for their clustering field are stored in the same blocks or cluster of blocks.

**Example 14.3** An ISAM file based on cluster indexing is illustrated in Fig. 14.6. We consider the record structure of the employee file discussed in Sec. 14.3. department is used as the clustering field. Observe the duplicate values of the clustering field in the records. The blocks are maintained in such a way that records holding the same value for the clustering field are stored in the same block or cluster of blocks.

**Secondary indexing** Secondary indexes are built on files for which some primary access already exists. In other words, the data records in the prime storage area are already sorted according to the primary key and available in blocks. The secondary indexing may be on a field that may be a candidate key( distinct values) or on a non-key field (duplicate values).

In the case of secondary key field having distinct values, the index is a dense index with one entry  $(K, B\uparrow)$  where  $K$  is the secondary key value and  $B\uparrow$  is the block address, for every record. The  $(K, B\uparrow)$  entries are however ordered on the key  $K$ , to facilitate binary search on the index during retrievals of data. To retrieve the record with secondary key value  $K$ , the entire block of records pointed to by  $B\uparrow$  is transferred to the internal memory and a linear search is done to retrieve the record.

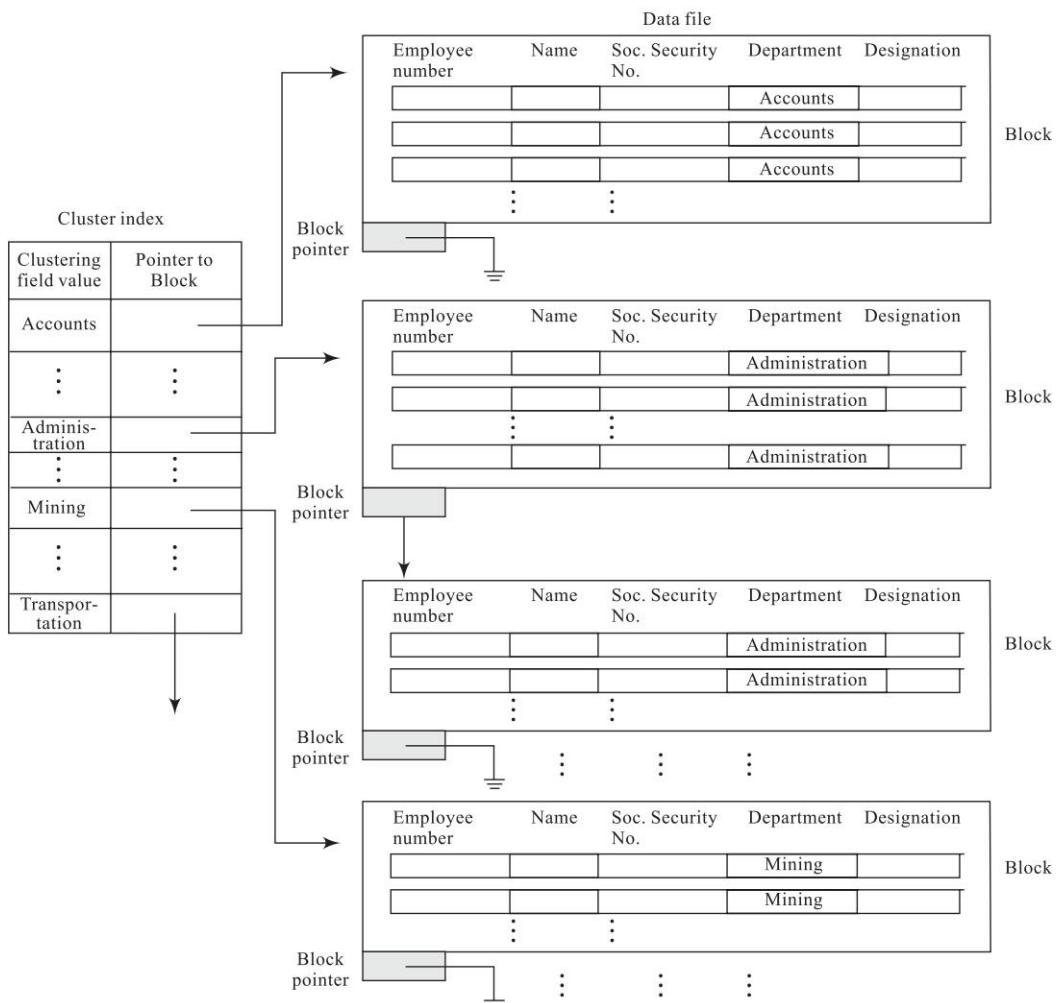


Fig. 14.6 An ISAM file based on cluster indexing

In the case of secondary key field having duplicate values, there are various options available to construct a secondary index. The first is to maintain a dense index of  $(K, B\uparrow)$  pairs where  $K$  is the secondary key value and  $B\uparrow$  is the block address, for every record. In such a case the index could carry several entries of  $(K, B\uparrow)$  pairs, for the same value of  $K$ . The second option would be to maintain the index as consisting of variable length entries. Each index entry would be of the form  $(K, B_1\uparrow, B_2\uparrow, B_3\uparrow, \dots, B_f\uparrow)$  where  $B_i\uparrow$ 's are block addresses of the various records holding the same value for the secondary key  $K$ . A third option is a modification of the second where  $B_i\uparrow$ 's are maintained as a linked list of block pointers and the index entry is just  $(K, T\uparrow)$  where  $T\uparrow$  is a pointer to the linked list.

A file could have several secondary indexes defined on it. Secondary indexes find significant applications in query based interfaces to data bases.

**Example 14.4** Figure 14.7 illustrates a secondary indexing based file. The secondary index implements its entries as a tuple comprising the secondary key value and the list of block addresses of the records with the particular key value. The sequential file available in the primary storage area is already sorted on its primary key.

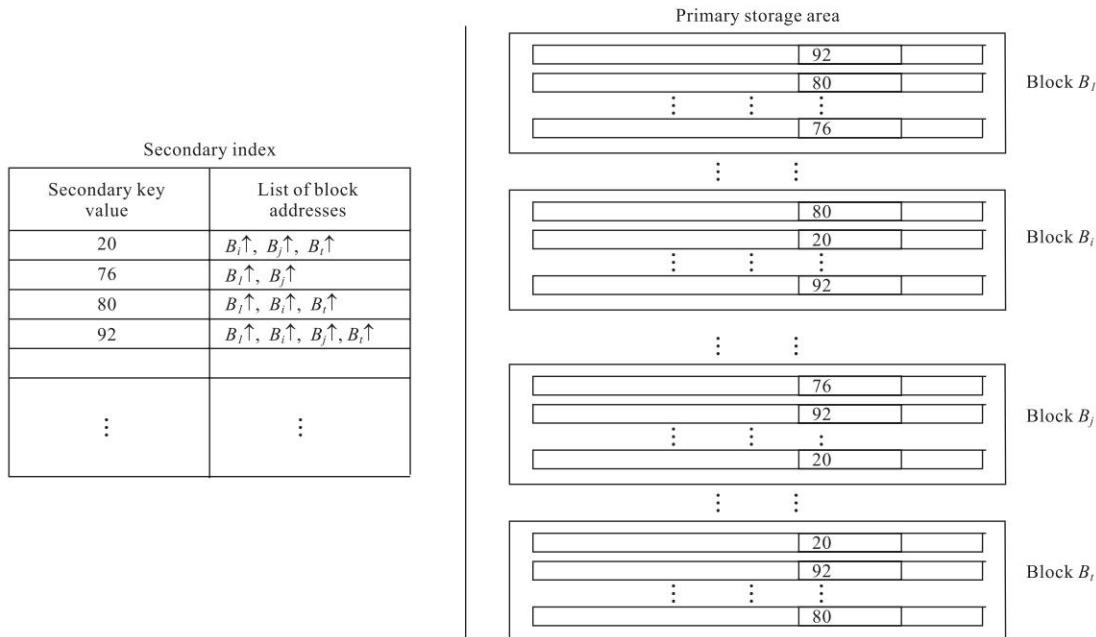


Fig. 14.7 Secondary indexing of a file

## Direct File Organization

## 14.8

*Direct file organizations* make use of techniques to directly map the key values of their records to their respective physical location addresses. Commonly, the techniques employ some kind of arithmetic or mathematical functions to bring about the mapping. The direct mapping of the keys with their addresses paves way for efficient retrievals and storage.

Hash functions are prominent candidates used by direct files for bringing about the mapping between the keys and the addresses. Hash functions and hashing were elaborately discussed in Chapter 13. The application of hashing for the storage of files in the external memory is known as *external hashing*.

Given a file of records,  $\{R_1, R_2, R_3, \dots, R_N\}$  with keys  $\{k_1, k_2, k_3, \dots, k_N\}$  a hash function  $H$  is employed to determine the storage address of each of the records in the storage device. Given a key  $k$ ,  $H(k)$  yields the storage address. Unlike other file organizations where indexes are maintained to track the storage address area, direct files undertake direct mapping of the keys to the storage locations of the records. In practice the hash function  $H(k)$  yields a bucket number which is then mapped to the absolute block address in the disk.

Buckets are designed to handle collisions amongst keys. Thus a group of synonyms share the same bucket. In the case of overflow of a bucket, a common solution employed is to maintain overflow buckets with links to their original buckets. Severe overflows to the same bucket may call for multiple overflow buckets each linked to the other. This may however deteriorate the performance of the file organization during a retrieval operation. If a deletion leaves an overflow bucket empty, then the bucket is removed and perhaps could be inserted into a linked list of empty overflow buckets for future use.

**Example 14.5** Figure 14.8 illustrates the overall structure of a direct file organization. Each bucket records the synonym keys and the pointers to their storage locations. The storage area is divided into blocks which hold a group of records. Note the overflow buckets which take care of synonyms that overflowed from their respective buckets.

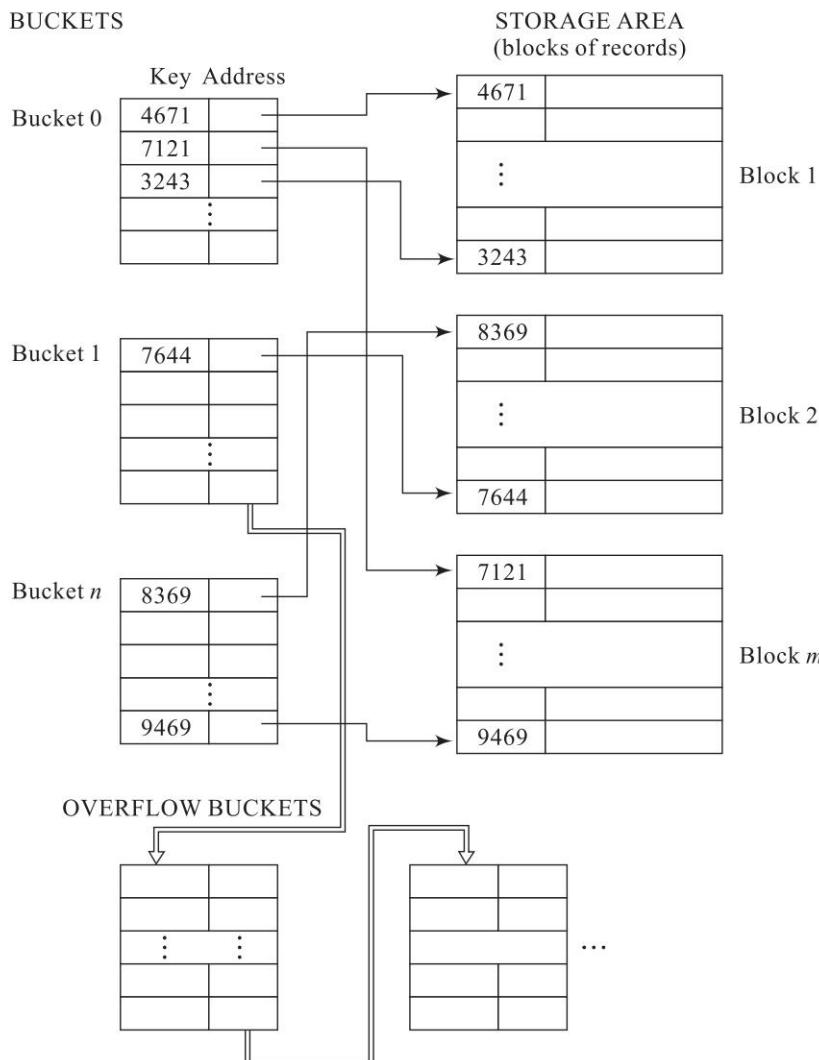


Fig. 14.8 A direct file organization



## Summary

- The internal memory or the primary memory of a computer is limited in capacity. To handle voluminous data, a computer system takes resort to external memory or secondary memory. Magnetic tapes, disks, drums are examples of secondary storage devices.
- A file is a collection of records and a record is a collection of fields. File organizations are methods or strategies for the efficient storage and retrieval of data. While the organization of records in a file refers to its logical organization, the storage of the records on the secondary storage device refers to its physical organization.
- A primary key or a key, is a field or a collection of fields that uniquely identifies a record. Candidate keys, super keys, secondary keys and alternate keys are other terms associated with the keys of a file.
- Files support a variety of operations such as open, close, read, insert, delete, update and reset.
- A heap or pile organization is a non-keyed file where records are not maintained in any particular order.
- A sequential file organization maintains its records in the order of its primary keys. The insert, delete and update operations are carried out in a batched mode, leading to the creation of transaction and new master files. The operations could also be handled by making use of overflow blocks.
- Indexed Sequential files offer efficient sequential and random access to its data records. The random access is made possible by making use of indexes. A variety of indexing based file organizations are possible by employing various types of indexing. Primary indexing, multilevel indexing, cluster indexing and secondary indexing are some of the important types of indexing.
- Direct file organizations make use of techniques to map their keys to the physical storage addresses of the records. Hash functions are a popular choice to bring about this mapping.



## Illustrative Problems

**Problem 14.1** The primary keys of a sample set of records are listed below. Assuming that the primary storage area accommodates 7 records / block and that the first record of the block is chosen as the block anchor, outline a schematic diagram for an ISAM file organization of the records built on primary indexing.

007 024 116 244 356 359 386 451 484 496 525 584 591 614 622 646 678 785 981  
991 999 1122 1466 2468 3469 4567 8907

**Solution:** The schematic diagram of the ISAM file organization based on primary indexing is shown in Fig. I 14.1.

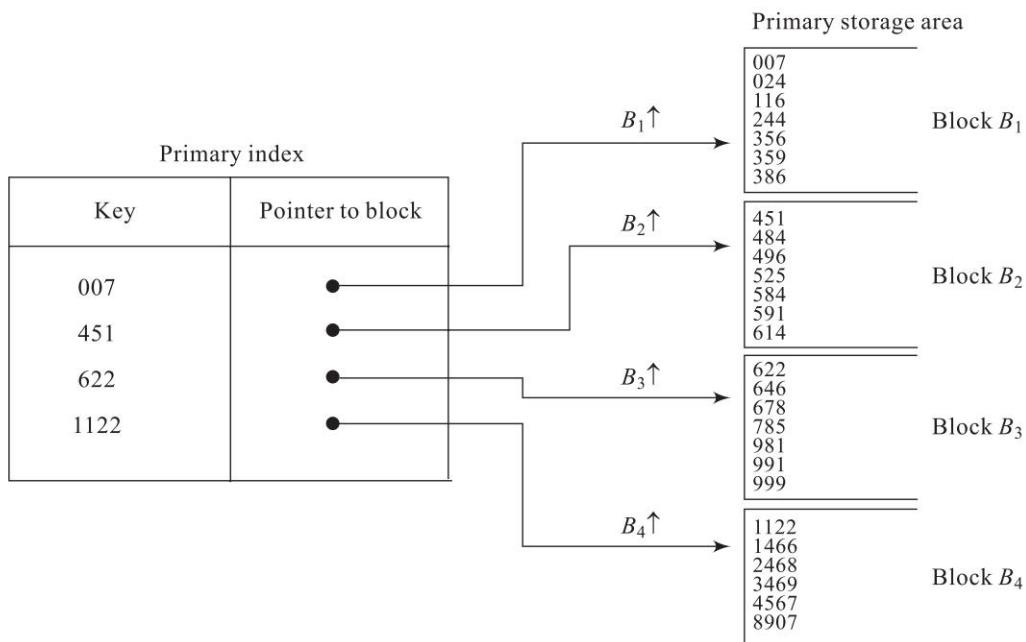


Fig. I 14.1

**Problem 14.2** For the sample set of records shown in Illustrative Problem I 14.1, design an ISAM file organization based on multilevel indexing for two levels of indexes. Assume a block size of 4 in the primary storage area and the first record of the block as the block anchor.

**Solution:** The schematic diagram of the ISAM file organization based on multilevel indexing for two levels of indexes is shown in Fig. I 14.2.

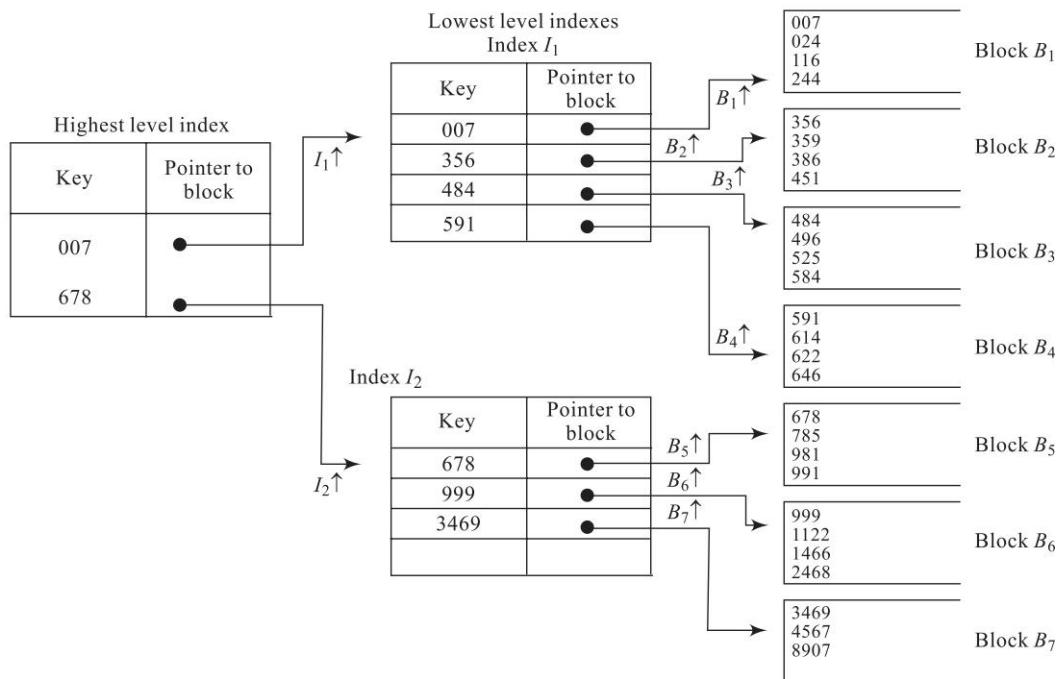
**Problem 14.3** For the following used car file with a record structure as shown below, design a secondary indexing based file organization making use of the sample set of records shown in Table I 14.3. Here, vehicle registration number is the primary key. Assume a block size of 2 records, in the primary storage area. Design secondary indexes using the fields (i) year of registration and (ii) colour.

Used car record structure:

|                             |                      |        |       |
|-----------------------------|----------------------|--------|-------|
| Vehicle registration number | year of registration | colour | model |
|-----------------------------|----------------------|--------|-------|

Table I 14.3

| Vehicle registration number | Year of registration | Colour        | Model    |
|-----------------------------|----------------------|---------------|----------|
| TN4117                      | 1990                 | Pearl white   | Prestige |
| TN4623                      | 1990                 | Silky silver  | Pride    |
| TN5724                      | 1991                 | Metallic blue | Pride    |
| TN6234                      | 1994                 | Silky silver  | Sarathi  |
| TN7146                      | 1994                 | Metallic blue | Sarathi  |
| TN7245                      | 1994                 | Pearl white   | Pride    |
| TN8436                      | 1995                 | Black         | Prestige |
| TN8538                      | 1996                 | Pearl white   | Flight   |

**Fig. I 14.2**

**Solution:** The schematic diagram for the secondary indexing of the used car file is shown in Fig. I 14.3. Both the indexes are shown in the same figure. Observe how the data records are ordered according to the primary key in the primary storage area.

**Problem 14.4** For the used car file discussed in Illustrative Problem I 14.3, design a cluster index based file organization on the non-key field year of registration. Assume that the blocks in the primary storage area can hold up to 2 records each.

**Solution:** Figure I 14.4 illustrates the schematic diagram for the cluster index based file organization for the used car file.

**Problem 14.5** Design a direct file organization using a hash function, to store an item file with item number as its primary key. The primary keys of a sample set of records of the item file are listed below. Assume that the buckets can hold 2 records each and the blocks in the primary storage area can accommodate a maximum of 4 records each. Make use of the hash function  $h(k) = k \bmod 8$ , where  $k$  represents the numerical value of the primary key (item number).

369 760 692 871 659 975 981 115 620 208 821 111 554 781 181 965

**Solution:** Figure I 14.5 represents the schematic diagram of the direct file organization of the item file using a hash function. Table I14.5 represents the hash function values of the primary keys.

| Secondary index           |                                                      | Primary storage area |
|---------------------------|------------------------------------------------------|----------------------|
| Secondary key<br>(colour) | Pointers of blocks                                   |                      |
| Black                     | $B_4 \uparrow$                                       | Block $B_1$          |
| Metallic blue             | $B_2 \uparrow \quad B_3 \uparrow$                    | Block $B_2$          |
| Pearl blue                | $B_1 \uparrow \quad B_3 \uparrow \quad B_4 \uparrow$ | Block $B_3$          |
| Silky silver              | $B_1 \uparrow \quad B_2 \uparrow$                    | Block $B_4$          |

| Secondary key<br>(year of registration) | Pointers of blocks                                   |  |
|-----------------------------------------|------------------------------------------------------|--|
| 1990                                    | $B_1 \uparrow \quad B_1 \uparrow$                    |  |
| 1991                                    | $B_2 \uparrow$                                       |  |
| 1994                                    | $B_2 \uparrow \quad B_3 \uparrow \quad B_3 \uparrow$ |  |
| 1995                                    | $B_4 \uparrow$                                       |  |
| 1996                                    | $B_4 \uparrow$                                       |  |

Fig. I 14.3

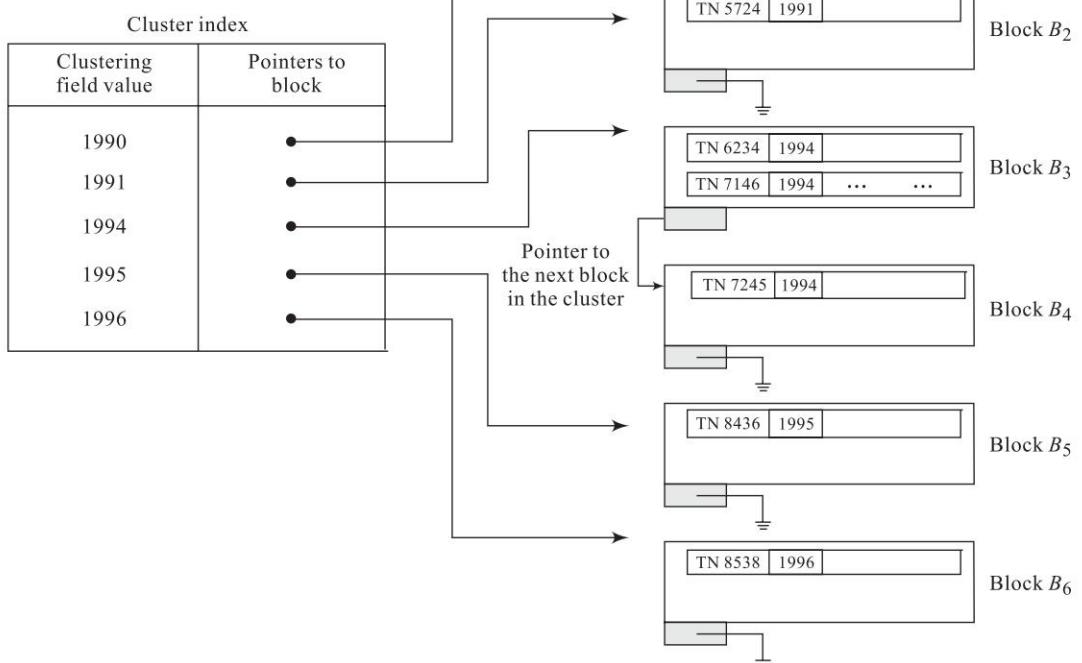
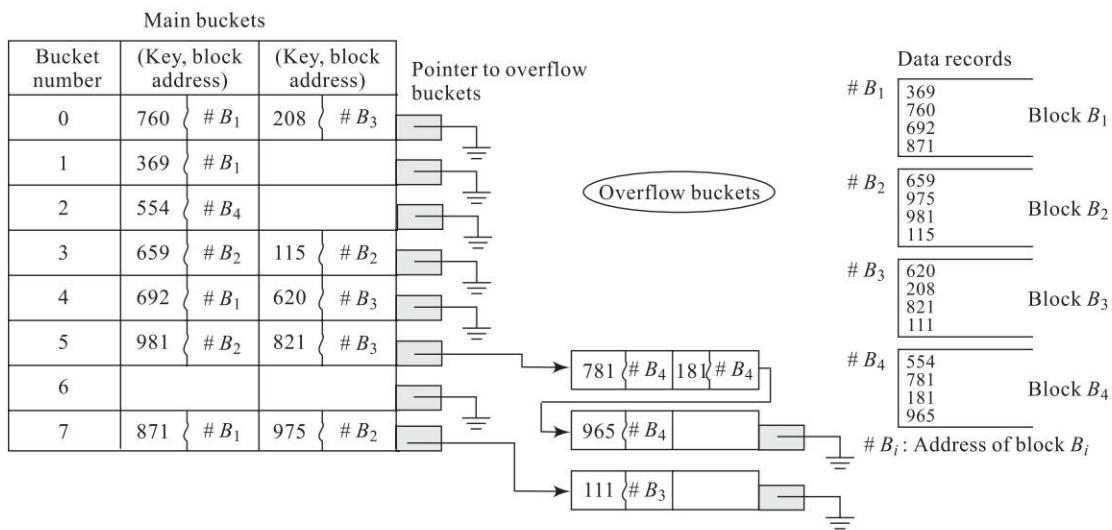


Fig. I 14.4

**Table I 14.5**

|                                |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
|--------------------------------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| Primary key ( $k$ )            | 369 | 760 | 692 | 871 | 659 | 975 | 981 | 115 | 620 | 208 | 821 | 111 | 554 | 781 | 181 | 965 |
| Hash function value ( $h(k)$ ) | 1   | 0   | 4   | 7   | 3   | 7   | 5   | 3   | 4   | 0   | 5   | 7   | 2   | 5   | 5   | 5   |

**Fig. I 14.5**

**Problem 14.6** For the direct file organization of the item file constructed in Illustrative Problem 14.5, undertake the following operations independently:

- (i) Insert item records with primary keys 441 and 805
- (ii) Delete the records with primary keys 369 and 111.

**Solution:** Figure I 14.6(a) illustrates the insertion of the keys 441 and 805 into the direct file. Figure I 14.6(b) illustrates the deletion of the keys 369 and 111. The delete operations are undertaken independent of the insert operations. The affected portions of the file alone are shown in the figures.

It can be observed how the deletion of the key 111 empties the overflow bucket, as a result of which the entire empty bucket gets removed.



## Review Questions

1. A minimal superkey is in fact a \_\_\_\_\_
  - (a) secondary key
  - (b) primary key
  - (c) non key
  - (d) none of these
2. State whether true or false:
  - (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

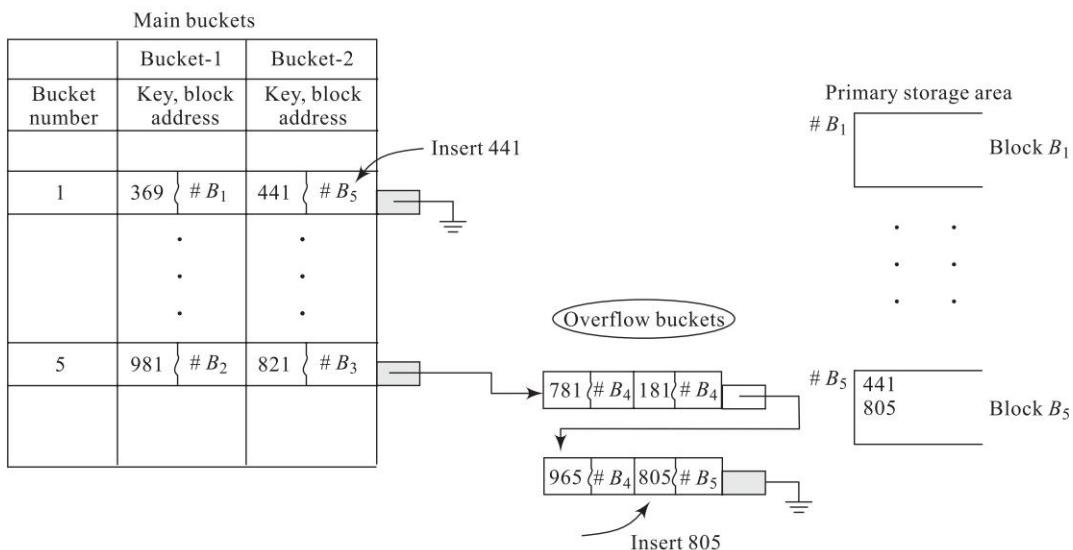


Fig. I 14.6(a)

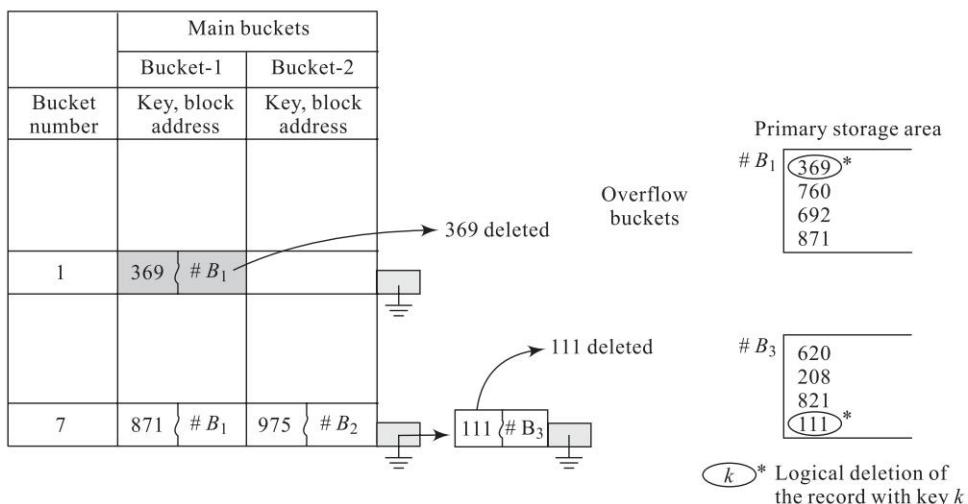


Fig. I 14.6(b)

3. An index consisting of variable length entries where each index entry would be of the form  $(K, B_1 \uparrow, B_2 \uparrow, B_3 \uparrow, \dots B_t \uparrow)$  where  $B_i \uparrow$ '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
4. Match the following:
 

|                                  |                      |
|----------------------------------|----------------------|
| (A) heap file organization       | (i) transaction file |
| (B) sequential file organization | (ii) non keyed       |
| (C) ISAM file organization       | (iii) hash function  |
| (D) direct file organization     | (iv) indexing        |

- |                 |            |             |            |
|-----------------|------------|-------------|------------|
| (a) (A, (i) )   | (B, (iv) ) | (C, (iii) ) | (D, (ii) ) |
| (b) (A, (ii) )  | (B, (iv) ) | (C, (iii) ) | (D, (i) )  |
| (c) (A, (ii) )  | (B, (i) )  | (C, (iv) )  | (D, (iii)) |
| (d) (A, (iii) ) | (B, (i) )  | (C, (ii) )  | (D, (iv) ) |

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



## Programming Assignments

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

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

Assume that the name of the movie is the primary key of the file. The field type refers to the type of the movie viz., drama, sci-fi, horror, crime thriller, comedy etc. Input a sample set of records of your choice into the movie file.

- (i) Implement a primary index based ISAM file organization.
  - (ii) Implement secondary indexes on director, type and production cost.
  - (iii) How could the secondary index based file organization in Programming Assignment 3 (Chapter 14) (ii) be used to answer a query such as "Who are the directors who have directed films of the type comedy or drama incurring the highest production cost?"
4. A company provides reimbursement of mobile phone subscription charges to its employees belonging to the managerial cadre and above. The following record structure captures the details. employee number which is designated as the primary key is a numerical 3-digit key. type refers to post paid or pre paid class of subscription to the mobile service. subscription charges refers to the charges incurred by the employee at the end of every month.

| employee number | designation | mobile number | type | subscription charges |
|-----------------|-------------|---------------|------|----------------------|
|-----------------|-------------|---------------|------|----------------------|

For a sample set of records implement the file as

- (a) an array of records (block size = 1), and
- (b) an array of pointers to records (assume that each pointer to record is a linked list of two nodes, each representing a record. In other words, each block is a linked list of two nodes (block size = 2)).

Make use of an appropriate hash function to design a direct file organization for the saidfile. Write a menu driven program, which

- (1) inserts new records, when recruitments or promotions to the managerial cadre are made,
  - (2) deletes records, when the employees concerned relinquish duties or terminate mobile usage due to various reasons and
  - (3) updates records regarding the subscription charges at the end of every month, changes if any, in type and designation fields etc.
5. Make use of a random number generator to generate a list of 500 three digit numbers. Create a sequential list FILE of the 500 numbers. Artificially implement storage blocks on the sequential list with every block containing a maximum of 10 numbers only. Open an index INDX over the sequential list FILE which records the highest key in each storage block and the address of the storage block. Implement Indexed Sequential search to look for keys K in FILE. Compare the number of comparisons made by the search with that of Sequential search for the same set of keys.
- Extend the implementation to include an index over the index INDX.

## CHAPTER



## SEARCHING

## 15

## Introduction

## 15.1

*Search* (or *Searching*) is a common place occurrence in every day life. Searching for a book in the library, searching for a subscriber's telephone number in the telephone directory, searching for one's name in the electoral rolls are some examples.

In the discipline of computer science, the problem of search has assumed enormous significance. It spans a variety of applications, rather disciplines, beginning from searching for a key in a list of data elements to searching for a solution to a problem in its *search space*. Innumerable problems exist where one searches for patterns – images, voice, text, hyper text, photographs etc., in a repository of data or patterns, for the solution of the problems concerned. A variety of search algorithms and procedures appropriate to the problem and the associated discipline exist in the literature.

In this chapter we enumerate search algorithms pertaining to the problem of looking for a key  $K$  in a list of data elements. When the list of data elements is represented as a linear list the search procedures of *linear search* or *sequential search*, *transpose sequential search*, *interpolation search*, *binary search* and *Fibonacci search* are applicable. When the list of data elements is represented using non linear data structures such as binary search trees or AVL trees or B trees etc., the appropriate *tree search* techniques unique to the data structure representation may be applied. Hash tables also promote efficient searching. Search techniques such as *breadth first search* and *depth first search* are applicable on graph data structures. In the case of data representing an index of a file or a group of ordered elements, *indexed sequential search* may be employed. This chapter discusses all the above mentioned search procedures.

## Linear Search

## 15.2

A *linear search* or *sequential search* is one where a key  $K$  is searched for, in a linear list  $L$  of data elements. The list  $L$  is commonly represented using a sequential data structure such as an array. If  $L$  is ordered then the search is said to be an *ordered linear search* and if  $L$  is unordered then it is said to be *unordered linear search*.

- 15.1 *Introduction*
- 15.2 *Linear Search*
- 15.3 *Transpose sequential search*
- 15.4 *Interpolation search*
- 15.5 *Binary search*
- 15.6 *Fibonacci search*
- 15.7 *Other search techniques*

## Ordered linear search

Let  $L = \{K_1, K_2, K_3, \dots, K_n\}$ ,  $K_1 < K_2 < \dots < K_n$  be the list of ordered elements. To search for a key  $K$  in the list  $L$ , we undertake a linear search comparing  $K$  with each of the  $K_i$ . So long as  $K > K_i$  comparing  $K$  with the data elements of the list  $L$  progresses. However, if  $K \leq K_i$ , then if  $K = K_i$  then the search is done, otherwise the search is unsuccessful implying  $K$  does not exist in the list  $L$ . It is easy to see how ordering the elements renders the search process to be efficient.

Algorithm 15.1 illustrates the working of ordered linear search.

### Algorithm 15.1: Procedure for ordered linear search

```
procedure LINEAR_SEARCH_ORDERED( $L, n, K$ )
    /*  $L[0:n-1]$  is a linear ordered list of data elements.  $K$ 
       is the key to be searched for in the list. In case of
       unsuccessful search, the procedure prints the message "KEY
       not found" otherwise prints "KEY found" and returns the
       index  $i$  */
     $i = 0;$ 
    while (( $i < n$ ) and ( $K > L[i]$ )) do          /* search for  $X$  down the list*/
         $i = i + 1;$ 
    endwhile
    if ( $K = L[i]$ ) then { print (" KEY found");
                           return ( $i$ ); } /* Key  $K$  found. Return index  $i$  */
    else
        print (" KEY not found");
    end LINEAR_SEARCH_ORDERED.
```



**Example 15.1** Consider an ordered list of elements  $L[0:5] = \{16, 18, 56, 78, 90, 100\}$ . Let us search for the key  $K = 78$ . Since  $K$  is greater than 16, 18, and 56, the search terminates at the fourth element when  $K \leq (L[3] = 78)$  is true. At this point, since  $K = L[3] = 78$ , the search is successfully done. However in the case of searching for key  $K = 67$ , the search progresses until the condition  $K \leq (L[3] = 78)$  is reached. At this point since  $K \neq L[3]$ , we deem the search to be unsuccessful.

Ordered linear search reports a time complexity of  $O(n)$  in the worst case and  $O(1)$  in the best case, in terms of key comparisons. However, it is essential that the elements are ordered before the search process is undertaken.

## Unordered linear search

In this search, a key  $K$  is looked for in an unordered linear list  $L = \{K_1, K_2, K_3, \dots, K_n\}$  of data elements. The method obviously of the 'brute force' kind, merely calls for a sequential search down the list looking for the key  $K$ .

Algorithm 15.2 illustrates the working of the unordered linear search.

### Algorithm 15.2: Procedure unordered linear search

```
procedure LINEAR_SEARCH_UNORDERED( $L, n, K$ )
    /*  $L[0:n-1]$  is a linear unordered list of data elements.
        $K$  is the key to be searched for in the list. In case
       of unsuccessful search, the procedure prints the message "KEY
       not found" otherwise prints "KEY found" and returns the
       index  $i$  */
```

```

i = 0;
while (( i < n) and ( L[i] ≠ K)) do      /* search for X down the list*/
i = i + 1;
endwhile

if ( L[i]= K) then { print (" KEY found");
                        return (i); }    /* Key K found. Return index i */

else
    print (" KEY not found");
end LINEAR_SEARCH_UNORDERED.

```



**Example 15.2** Consider an unordered list  $L[0:5] = \{ 23, 14, 98, 45, 67, 53 \}$  of data elements. Let us search for the key  $K = 53$ . Obviously the search progresses down the list comparing key  $K$  with each of the elements in the list until it finds it as the last element in the list. In the case of searching for the key  $K = 110$ , the search progresses but falls off the list thereby deeming it to be an unsuccessful search.

Unordered linear search reports a worst case complexity of  $O(n)$  and a best case complexity of  $O(1)$  in terms of key comparisons. However, its average case performance in terms of key comparisons can only be inferior to that of ordered linear search.

## Transpose Sequential Search

## 15.3

Also known as *Self organizing sequential search*, *Transpose sequential search* searches a list of data items for a key, checking itself against the data items one at a time in a sequence. If the key is found, then it is swapped with its predecessor and the search is termed successful. The swapping of the search key with its predecessor, once it is found, favours faster search when one repeatedly looks for the key. The more frequently one looks for a specific key in a list, the faster the retrievals take place in transpose sequential search, since the found key moves towards the beginning of the list with every retrieval operation. Thus transpose sequential search is most successful when a few data items are repeatedly looked for in a list.

Algorithm 15.3 illustrates the working of transpose sequential search.

### Algorithm 15.3: Procedure for transpose sequential search

```

procedure TRANSPOSE_SEQUENTIAL SEARCH(L, n, K)
    /* L[0:n-1] is a linear unordered list of data
       elements. K is the key to be searched for in the
       list. In case of unsuccessful search, the procedure
       prints the message "KEY not found" otherwise prints
       "KEY found" and swaps the key with its predecessor
       in the list */
i = 0;
while (( i < n) and (L[i] ≠ K)) do      /*search for X down the list*/
i = i + 1;
endwhile

```

```

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

```

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

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

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

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

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

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

## Interpolation Search

## 15.4

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

Interpolation search is based on this principle of attempting to look for a key in a list of elements, by comparing the key with specific elements at “calculated” positions and ensuring if the key occurs “before” it or “after” it until either the key is found or not found. The list of elements *must be ordered* and we assume that they are uniformly distributed with respect to requests.

Let us suppose we are searching for a key  $K$  in a list  $L = \{K_1, K_2, K_3, \dots, K_n\}$ ,  $K_1 < K_2 < \dots < K_n$  of numerical elements. When it is known that  $K$  lies between  $K_{low}$  and  $K_{high}$  (i.e.)  $K_{low} < K < K_{high}$ , then the next element that is to be probed for key comparison is chosen to be the one that lies

$\frac{(K - K_{low})}{(K_{high} - K_{low})}$  of the way between  $K_{low}$  and  $K_{high}$ . It is this consideration that has made the search to be termed interpolation search.

During the implementation of the search procedure, the next element to be probed in a sublist  $\{K_i, K_{i+1}, K_{i+2}, \dots, K_j\}$  for comparison against the key  $K$  is given by  $K_{mid}$  where  $mid$  is given by

$$mid = i + (j - i) \cdot \frac{(K - K_i)}{(K_j - K_i)}. \text{ The key comparison results in any one of the following cases:}$$

If  $(K = K_{mid})$  then the search is done.

If  $(K < K_{mid})$  then continue the search in the sublist  $\{K_i, K_{i+1}, K_{i+2}, \dots, K_{mid-1}\}$

If  $(K > K_{mid})$  then continue the search in the sublist  $\{K_{mid+1}, K_{mid+2}, K_{mid+3}, \dots, K_j\}$

Algorithm 15.4 illustrates the interpolation search procedure.

#### Algorithm 15.4: Procedure for interpolation search

```

procedure INTERPOLATION_SEARCH(L, n, K)
    /* L[1:n] is a linear ordered list of data elements.
       K is the key to be searched for in the list. In case
       of unsuccessful search, the procedure prints the message
       "KEY not found" otherwise prints "KEY found". */
    i = 1;
    j = n;
    if ( K < L[i] ) or (K > L[j]) then { print("Key not found"); exit(); }
        /* if the key K does not lie within the
           list then found = false; print "key not found"*/
    while (( i ≤ j) and (found = false)) do
        mid = i + (j - i) ·  $\frac{(K - L[i])}{(L[j] - L[i])}$ ;
        case
            : K = L[mid]: { found = true; print ("Key found"); }
            : K < L[mid]: j = mid-1;
            : K > L[mid]: i = mid+1;
        endcase
    endwhile
    if ( found = false) then print (" Key not found");
        /* Key K not found in the list L*/
end INTERPOLATION_SEARCH.

```

**Example 15.4** Consider a list  $L = \{24, 56, 67, 78, 79, 89, 90, 95, 99\}$  of ordered numerical elements. Let us search for the set of keys  $\{67, 45\}$ . Table 15.2 illustrates the trace of Algorithm 15.4 for the set of keys. The algorithm proceeds with its search since both the keys lie within the list.

**Table 15.2** Trace of Algorithm 15.4 during the search for keys 67 and 45

| Search key $K$ | $i$ | $j$ | found | mid                                                                        | $K = L[mid]$         |
|----------------|-----|-----|-------|----------------------------------------------------------------------------|----------------------|
| 67             | 1   | 9   | false | $1 + (9 - 1) \cdot \frac{(67 - 24)}{(99 - 24)} = \lfloor 5.58 \rfloor = 5$ | 67 < ( $L[5] = 79$ ) |
|                | 1   | 4   | false | 3                                                                          | 67 = ( $L[3]=67$ )   |
| 45             | 1   | 9   | false | $1 + (9 - 1) \cdot \frac{(45 - 24)}{(99 - 24)} = \lfloor 3.24 \rfloor = 3$ | 45 < ( $L[3]=67$ )   |
|                | 1   | 2   | false | 1                                                                          | 45 > ( $L[1]=24$ )   |
|                | 2   | 2   | false | 2                                                                          | 45 < ( $L[2]=56$ )   |
|                | 2   | 1   | false | Key not found                                                              |                      |

In the case of key 67 the search was successful. However in the case of key 45, as can be observed in the last row of the table, the condition  $(i \leq j)$  (i.e.)  $(2 \leq 1)$  failed in the while loop of the algorithm and hence the search terminates signaling "key not found".

The worst case complexity of interpolation search is  $O(n)$ . However, on an average the search records a brilliant  $O(\log_2 \log_2 n)$  complexity.

## Binary Search

## 15.5

In the previous section we discussed interpolation search which works on ordered lists and reports an average case complexity of  $O(\log_2 \log_2 n)$ . Another efficient search technique that operates on ordered lists is the *binary search* also known as *logarithmic search* or *bisection*.

A binary search searches for a key  $K$  in an ordered list  $L = \{K_1, K_2, K_3, \dots, K_n\}$ ,  $K_1 < K_2 < \dots < K_n$  of data elements, by halving the search list with each comparison until the key is either found or not found. The key  $K$  is first compared with the median element of the list viz.,  $K_{mid}$ . For a sublist  $\{K_i, K_{i+1}, K_{i+2}, \dots, K_j\}$ ,  $K_{mid}$  is obtained as the key occurring at the position  $mid$  which is computed as  $mid = \left\lfloor \frac{(i+j)}{2} \right\rfloor$ . The comparison of  $K$  with  $K_{mid}$  yields the following cases:

If  $(K = K_{mid})$  then the binary search is done.

If  $(K < K_{mid})$  then continue binary search in the sub list  $\{K_i, K_{i+1}, K_{i+2}, \dots, K_{mid-1}\}$

If  $(K > K_{mid})$  then continue binary search in the sub list  $\{K_{mid+1}, K_{mid+2}, K_{mid+3}, \dots, K_j\}$

During the search process, each comparison of key  $K$  with  $K_{mid}$  of the respective sub lists results in the halving of the list. In other words with each comparison the search space is reduced to half its original length. It is this characteristic that renders the search process to be efficient. Contrast this with a sequential list where the entire list is involved in the search!

Binary search adopts the *Divide-and-Conquer* method of algorithm design. Divide-and-Conquer is an algorithm design technique where to solve a given problem, the problem is first recursively divided (*Divide*) into sub-problems (smaller problem instances). The sub-problems that are small enough are easily solved (*Conquer*) and the solutions combined to obtain the solution to the whole problem. Divide-and-Conquer has turned out to be a successful algorithm design technique with regard to many problems. In the case of binary search, the divide-and-conquer aspect of the technique breaks the list (problem) into two sub lists (sub-problems). However, the key is searched for only in one of the sublists hence with every division a portion of the list gets discounted.

Algorithm 15.5 illustrates a recursive procedure for binary search.

### Decision tree for binary search

The binary search for a key  $K$  in the ordered list  $L = \{K_1, K_2, K_3, \dots, K_n\}, K_1 < K_2 < \dots < K_n$  traces a binary decision tree. Figure 15.1 illustrates the decision tree for  $n = 15$ . The first element to be compared with  $K$  in the list  $L = \{K_1, K_2, K_3, \dots, K_{15}\}$  is  $K_8$  which becomes the root of the decision tree. If  $K < K_8$  then the next element to be compared is  $K_4$  which is the left child of the decision tree. For the other cases of comparisons it is easy to trace the tree by making use of the following characteristics:

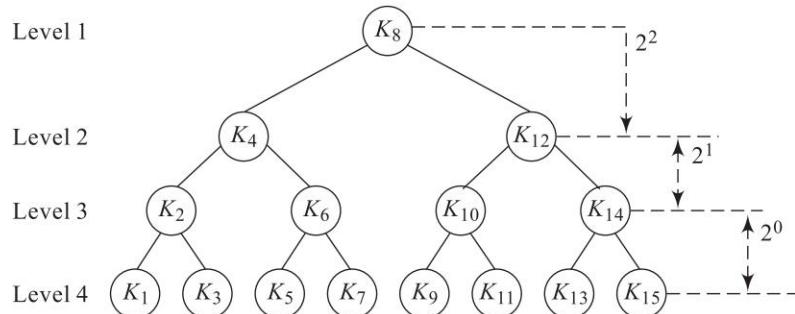
- the indexes of the left and the right child nodes differ by the same amount from that of the parent node.

For example, in the decision tree shown in Fig. 15.1 the left and right child nodes of the node  $K_{12}$ , viz.,  $K_{10}$  and  $K_{14}$  differ from their parent key index by the same amount.

This characteristic renders the search process to be *uniform* and therefore binary search is also termed as *uniform binary search*.

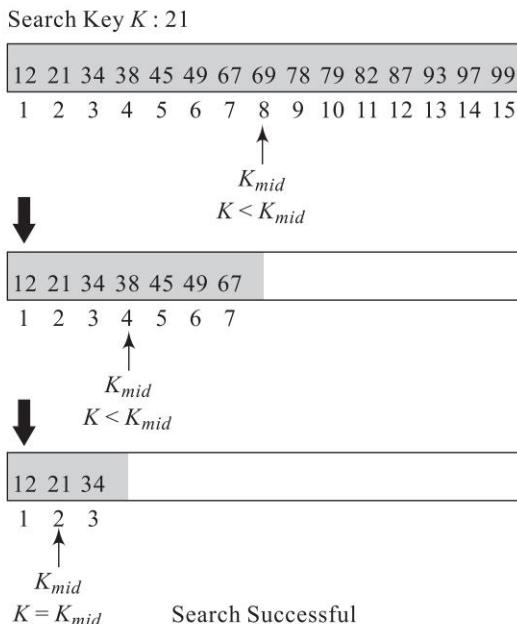
- for  $n$  elements where  $n = 2^t - 1$ , the difference in the indexes of a parent node and its child nodes follows the sequence  $2^0, 2^1, 2^2 \dots$  from the leaf upwards.

For example, in Fig. 15.1 where  $n = 15 = 2^4 - 1$ , the difference in index of all the leaf nodes from their respective parent nodes is  $2^0$ . The difference in index of all the nodes in level 3 from their respective parent nodes is  $2^1$  and so on.



**Fig. 15.1** Decision tree for binary search

**Example 15.5** Consider an ordered list  $L = \{K_1, K_2, K_3, \dots, K_{15}\} = \{12, 21, 34, 38, 45, 49, 67, 69, 78, 79, 82, 87, 93, 97, 99\}$ . Let us search for the key  $K = 21$  in the list  $L$ . The search process is illustrated in Fig. 15.2.  $K$  is first compared with  $K_{mid} = K_{\lfloor \frac{1+15}{2} \rfloor} = K_8 = 69$ . Since  $K < K_{mid}$ , the search continues in the sublist  $\{12, 21, 34, 38, 45, 49, 67\}$ . Now,  $K$  is compared with  $K_{mid} = K_{\lfloor \frac{1+7}{2} \rfloor} = K_4 = 38$ . Again  $K < K_{mid}$ , shrinks the search list to  $\{12, 21, 34\}$ . Now finally when  $K$  is compared with  $K_{mid} = K_{\lfloor \frac{1+3}{2} \rfloor} = K_2 = 21$ , the search is done. Thus in three comparisons we are able to search for the key  $K = 21$ .



**Fig. 15.2** Binary search process (Example 15.5)

Let us now search for the key  $K = 75$ . Proceeding in a similar manner,  $K$  is first compared with  $K_8 = 69$ . Since  $K > K_8$ , the search list is reduced to  $\{78, 79, 82, 87, 93, 97, 99\}$ . Now  $K < (K_{12} = 87)$ , hence the search list is reduced further to  $\{78, 79, 82\}$ . Comparing  $K$  with  $K_{10}$  reduces the search list to  $\{78\}$  which obviously yields the search to be unsuccessful. In the case of Algorithm 15.5, at this step of the search, the recursive call to BINARY\_SEARCH would have both  $low = high = 9$ , resulting in  $mid = 9$ . Comparing  $K$  with  $K_{mid}$  results in the call to `binary_search(L, 9, 8, K)`. Since  $(low > high)$  condition is satisfied, the algorithm terminates with the 'key not found' message.

**Algorithm 15.5:** Procedure for binary search

```
procedure binary_search( $L$ ,  $low$ ,  $high$ ,  $K$ )
    /*  $L[low:high]$  is a linear ordered sublist of data
       elements. Initially low is set to 1 and high to  $n$ .  $K$ 
       is the key to be searched in the list. */
```

```

if ( low > high) then {binary_search =0;
                        print("Key not found");
                        exit();}

else
{
                                /* key K not found*/
    mid=  $\left\lfloor \frac{low+high}{2} \right\rfloor$ ;

    case
        : K = L[mid]: { print ("Key found");
                         binary_search=mid;
                         return L[mid];}
        : K < L[mid]: binary_search = binary_search(L, low, mid-1, K);
        : K > L[mid]: binary_search = binary_search(L, mid+1, high, K);

    endcase
}
end binary_search.

```



Considering the decision tree associated with binary search, it is easy to see that in the worst case the number of comparisons needed to search for a key would be determined by the height of the decision tree and is therefore given by  $O(\log_2 n)$ .

## Fibonacci Search

15.6

The Fibonacci number sequence is given by { 0, 1, 1, 2, 3, 5, 8, 13, 21,...} and is generated by the following recurrence relation:

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_i &= F_{i-1} + F_{i-2} \end{aligned}$$

It is interesting to note that the Fibonacci sequence finds an application in a search technique termed *Fibonacci search*. While binary search selects the median of the sublist as its next element for comparison, the Fibonacci search determines the next element of comparison as dictated by the Fibonacci number sequence.

Fibonacci search works only on ordered lists and for convenience of description we assume that the number of elements in the list is one less than a Fibonacci number, (i.e.)  $n = F_k - 1$ . It is easy to follow Fibonacci search once the decision tree is traced, which otherwise may look mysterious!

### Decision tree for Fibonacci search

The decision tree for Fibonacci search satisfies the following characteristic:

If we consider a grandparent, parent and its child nodes and if the difference in index between the grandparent and the parent is  $F_k$  then

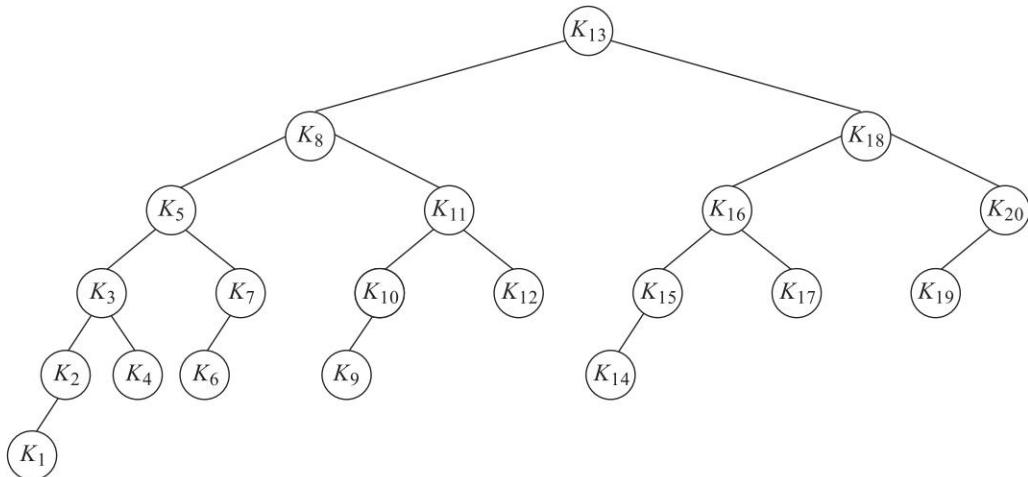
- (i) if the parent is a left child node then the difference in index between the parent and its child nodes is  $F_{k-1}$ , whereas

- (ii) if the parent is a right child node then the difference in index between the parent and the child nodes is  $F_{k-2}$ .

Let us consider an ordered list  $L = \{K_1, K_2, K_3, \dots, K_n\}, K_1 < K_2 < \dots < K_n$  where  $n = F_k - 1$ . The Fibonacci search decision tree for  $n = 20$  where  $20 = (F_8 - 1)$  is shown in Fig. 15.3.

The root of the decision tree which is the first element in the list to be compared with key  $K$  during the search is that key  $K_i$  whose index  $i$  is the closest Fibonacci sequence number to  $n$ . In the case of  $n = 20$ ,  $K_{13}$  is the root since the closest Fibonacci number to  $n = 20$  is 13.

If  $(K < K_{13})$  then the next key to be compared is  $K_8$ . If again  $(K < K_8)$  then it would be  $K_5$  and so on. Now it is easy to determine the other decision nodes making use of the characteristics mentioned above. Since child nodes differ from their parent by the same amount, it is easy to see that the right child of  $K_{13}$  should be  $K_{18}$  and that of  $K_8$  should be  $K_{11}$  and so on. Consider the grandparent-parent combination,  $K_8$  and  $K_{11}$  respectively, since  $K_{11}$  is the right child of its parent and the difference between  $K_8$  and  $K_{11}$  is  $F_4$ , the same between  $K_{11}$  and its two child nodes should be  $F_2$  which is 1. Hence the two child nodes of  $K_{11}$  are  $K_{10}$  and  $K_{12}$ . Similarly, considering the grandparent and parent combination of  $K_{18}$  and  $K_{16}$  where  $K_{16}$  is the left child of its parent and their difference is given by  $F_3$ , the two child nodes of  $K_{16}$  are given by  $K_{15}$  and  $K_{17}$  (difference is  $F_2$ ) respectively.



**Fig. 15.3** Decision tree for Fibonacci search

Algorithm 15.6 illustrates the procedure for Fibonacci search. Here  $n$ , the number of data elements is such that

- (i)  $F_{k+1} > (n+1)$  and
- (ii)  $F_k + m = (n+1)$  for some  $m \geq 0$ , where  $F_{k+1}$  and  $F_k$  are two consecutive Fibonacci numbers.

**Algorithm 15.6:** Procedure for Fibonacci search

```

procedure FIBONACCI_SEARCH( $L, n, K$ )
    /*  $L[1:n]$  is a linear ordered (non decreasing) list of data
       elements.  $n$  is such that  $F_{k+1} > (n+1)$ . Also  $F_k + m = (n+1)$ .
        $K$  is the key to be searched in the list. */
  
```

```

Obtain the largest Fibonacci number  $F_k$  closest to  $n+1$ ;
 $p = F_{k-1}$ ;
 $q = F_{k-2}$ ;
 $r = F_{k-3}$ ;
 $m = (n+1) - (p+q)$ ;
if ( $K > L[p]$ ) then  $p = p+m$ ;
found = false;
while (( $p \neq 0$ ) and (not found)) do
    case
        :  $K = L[p]$ : { print ("key found"); /* key found*/
            found = true;
        }
        :  $K < L[p]$ : { if ( $r = 0$ ) then  $p = 0$ 
            else {  $p = p-r$ ;  $t = q$ ;  $q = r$ ;  $r = t-r$ ; }
        }
        :  $K > L[p]$ : { if ( $q = 1$ ) then  $p = 0$ 
            else {  $p = p+r$ ;  $q = q-r$ ;  $r = r-q$  }
        }
    endcase
endwhile
if (found = false) then print ("key not found");
end FIBONACCI_SEARCH.

```

**Example 15.6** Let us search for the key  $K = 434$  in the ordered list  $L = \{2, 4, 8, 9, 17, 36, 44, 55, 81, 84, 94, 116, 221, 256, 302, 356, 396, 401, 434, 536\}$ . Here  $n$  ( $n = 20$ ) the number of elements is such that (i)  $F_9 > (n+1)$  and (ii)  $F_8 + m = (n+1)$  where  $m=0$  and  $n=20$ .

The algorithm for Fibonacci search first obtains the largest Fibonacci number closest to  $n+1$ , (i.e.),  $F_8$  in this case. It compares  $K = 434$  with the data element with index  $F_7$  (i.e.)  $L[13] = 221$ . Since  $K > L[13]$ , the search list is reduced to  $L[14: 20] = \{256, 302, 356, 396, 401, 434, 536\}$ . Now  $K$  compares itself with  $L[18] = 401$ . Since  $K > L[18]$  the search list is further reduced to  $L[19:20] = \{434, 536\}$ . Now  $K$  is compared with  $L[20] = 536$ . Since  $K < L[20]$  is true it results in the search list  $\{434\}$  which when searched yields the search key. The key is successfully found.

Following a similar procedure, searching for 66 in the list yields an unsuccessful search.

The detailed trace of Algorithm 15.6 for the search keys 434 and 66 is shown in Table 15.3.

**Table 15.3 Trace of Algorithm 15.6 for the search keys 434 and 66**

| Search key $K$ | $<$<br>$K = L[p]$<br>$>$ | $t$ | $p$ | $q$ | $r$ | Remarks                                        |
|----------------|--------------------------|-----|-----|-----|-----|------------------------------------------------|
| 434            |                          |     | 13  | 8   | 5   | $n = 20$<br>$m = 0$ since<br>$F_8 + 0 = n + 1$ |
|                | $K > L[13] = 221$        |     | 13  | 8   | 5   | Since $K > L[p]$ ,<br>$p = p+m$                |
|                | $K > L[13] = 221$        |     | 18  | 3   | 2   |                                                |

(Contd.)

(Contd.)

|    |                                                             |   |          |        |        |                                                                |
|----|-------------------------------------------------------------|---|----------|--------|--------|----------------------------------------------------------------|
|    | $K > L[18] = 401$<br>$K < L[20] = 536$<br>$K = L[19] = 434$ | 1 | 20<br>19 | 1<br>1 | 1<br>0 | <b>Key is found</b>                                            |
| 66 | $K < L[13] = 221$                                           | 8 | 13<br>8  | 8<br>5 | 5<br>3 | $n = 20$<br>$m = 0$                                            |
|    | $K > L[8] = 55$                                             |   | 11       | 2      | 1      |                                                                |
|    | $K < L[11] = 94$                                            | 2 | 10       | 1      | 1      |                                                                |
|    | $K < L[10] = 84$                                            | 1 | 9        | 1      | 0      |                                                                |
|    | $K < L[9] = 81$                                             |   |          |        |        | Since ( $r = 0$ ), $p$ is set to 0.<br><b>Key is not found</b> |

An advantage of Fibonacci search over binary search is that while binary search involves division which is computationally expensive, during the selection of the next element for key comparison, Fibonacci search involves only addition and subtraction.

## Other Search Techniques

15.7

### Tree search

The tree data structures of AVL trees (Sec. 10.3),  $m$ -way trees, B trees and tries (Chapter 11), Red-Black trees (Sec. 12.2) etc., are also candidates for the solution of search related problems. The inherent search operation that each of these data structures support can be employed for the problem of searching.

The techniques of sequential search, interpolation search, binary search and Fibonacci search are primarily employed for files or group of records or data elements that can be accommodated within the high speed internal memory of the computer. Hence these techniques are commonly referred to as *internal searching* methods. On the other hand when the file size is too large to be accommodated within the memory of the computer one has to take recourse to external storage devices such as disks or drums to store the file (see Chapter 17). In such cases when a search operation for a key needs to be undertaken, the process involves searching through blocks of storage spanning across storage areas. Adopting internal searching methods for these cases would be grossly inefficient. The search techniques as emphasized by  $m$ -way trees, B trees, tries and so on are suitable for such a scenario. Hence these search techniques are referred to as *external searching* methods.

### Graph search

The graph data structure and its traversal techniques of Breadth first traversal and Depth first traversal (Sec. 9.4) can also be employed for search related problems. If the search space is represented as a graph and the problem involves searching for a key  $K$  which is a node in the

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

## Indexed sequential search

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

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



## Summary

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



## Illustrative Problems

**Problem 15.1** For the list CHANNELS={ AAXN, ZZEE, CCBC, CNNN, DDDN, HHBO, GGOD, FFAS, NNDT, SSON, CCAF, NNGE, BBCB, PPRO} trace transpose sequential search for the elements in the list SELECT\_CHANL= {DDDN, NNDT, DDDN, PPRO, DDDN, NNDT}. Obtain the number of comparisons made during the search for each of the elements in the list SELECT\_CHANL.

**Solution:** The trace of transpose sequential search for the search of elements in the list SELECT\_CHANL over the list CHANNELS is presented in the following table:

| Search key | List L before search                                                                  | Number of element comparisons made during the search | List L after search                                                                   |
|------------|---------------------------------------------------------------------------------------|------------------------------------------------------|---------------------------------------------------------------------------------------|
| DDDN       | { AAXN, ZZEE, CCBC, CNNN, DDDN, HHBO, GGOD, FFAS, NNDT, SSON, CCAF, NNGE, BBCB, PPRO} | 5                                                    | { AAXN, ZZEE, CCBC, DDDN, CNNN, HHBO, GGOD, FFAS, NNDT, SSON, CCAF, NNGE, BBCB, PPRO} |
| NNDT       | { AAXN, ZZEE, CCBC, DDDN, CNNN, HHBO, GGOD, FFAS, NNDT, SSON, CCAF, NNGE, BBCB, PPRO} | 9                                                    | { AAXN, ZZEE, CCBC, DDDN, CNNN, HHBO, GGOD, NNDT, FFAS, SSON, CCAF, NNGE, BBCB, PPRO} |
| DDDN       | { AAXN, ZZEE, CCBC, DDDN, CNNN, HHBO, GGOD, NNDT, FFAS, SSON, CCAF, NNGE, BBCB, PPRO} | 4                                                    | { AAXN, ZZEE, DDDN, CCBC, CNNN, HHBO, GGOD, NNDT, FFAS, SSON, CCAF, NNGE, BBCB, PPRO} |
| PPRO       | { AAXN, ZZEE, DDDN, CCBC, CNNN, HHBO, GGOD, NNDT, FFAS, SSON, CCAF, NNGE, BBCB, PPRO} | 14                                                   | { AAXN, ZZEE, DDDN, CCBC, CNNN, HHBO, GGOD, NNDT, FFAS, SSON, CCAF, NNGE, PPRO, BBCB} |
| DDDN       | { AAXN, ZZEE, DDDN, CCBC, CNNN, HHBO, GGOD, NNDT, FFAS, SSON, CCAF, NNGE, PPRO, BBCB} | 3                                                    | { AAXN, DDDN, ZZEE, CCBC, CNNN, HHBO, GGOD, NNDT, FFAS, SSON, CCAF, NNGE, PPRO, BBCB} |

(Contd.)

(Contd.)

|      |                                                                                       |   |                                                                                       |
|------|---------------------------------------------------------------------------------------|---|---------------------------------------------------------------------------------------|
| NNDT | { AAXN, DDDN, ZZEE, CCBC, CCNN, HHBO, GGOD, NNDT, FFAS, SSON, CCAF, NNGE, PPRO, BBC } | 8 | { AAXN, DDDN, ZZEE, CCBC, CCNN, HHBO, NNDT, GGOD, FFAS, SSON, CCAF, NNGE, PPRO, BBC } |
|------|---------------------------------------------------------------------------------------|---|---------------------------------------------------------------------------------------|

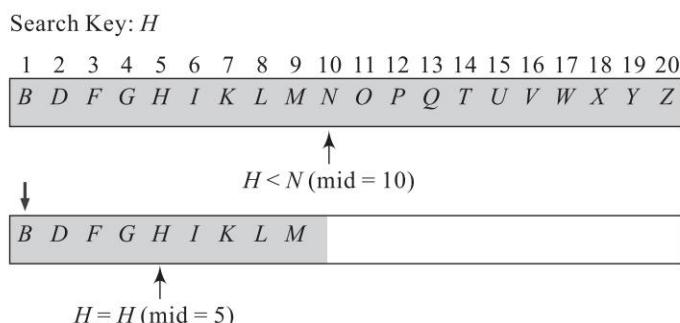
**Problem 15.2** For the ordered list  $L = \{B, D, F, G, H, I, K, L, M, N, O, P, Q, T, U, V, W, X, Y, Z\}$  undertake interpolation search (trace of Algorithm 15.4) for keys  $H$  and  $Y$ . Make use of the respective alphabetical sequence number for the keys, during the computation of the interpolation function.

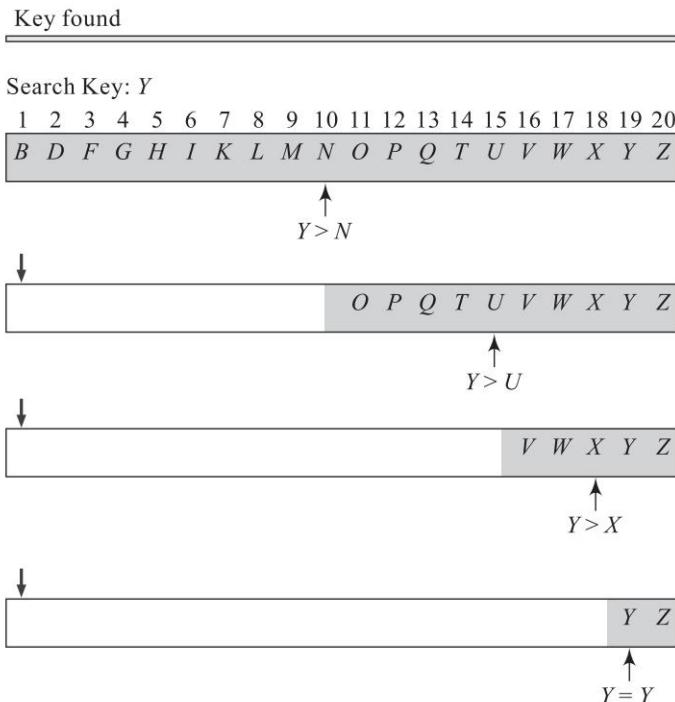
**Solution:** The table given below illustrates the trace of the algorithm during the search for keys  $H$  and  $Y$ .

| Search key $K$ | $i$ | $j$ | mid                                                                   | $\begin{array}{l} < \\ K = L[mid] \\ > \end{array}$ |
|----------------|-----|-----|-----------------------------------------------------------------------|-----------------------------------------------------|
| $H$            | 1   | 20  | $1 + (20-1) \cdot \frac{(8-2)}{(26-2)} = \lfloor 5.75 \rfloor = 5$    | $H = (L[5] = H)$<br><b>Key found</b>                |
| $Y$            | 1   | 20  | $1 + (20-1) \cdot \frac{(25-2)}{(26-2)} = \lfloor 19.20 \rfloor = 19$ | $Y = (L[19] = Y)$<br><b>Key found</b>               |

**Problem 15.3** For the ordered list  $L$  and the search keys given in Illustrative Problem 15.2 trace the steps of binary search during the search process.

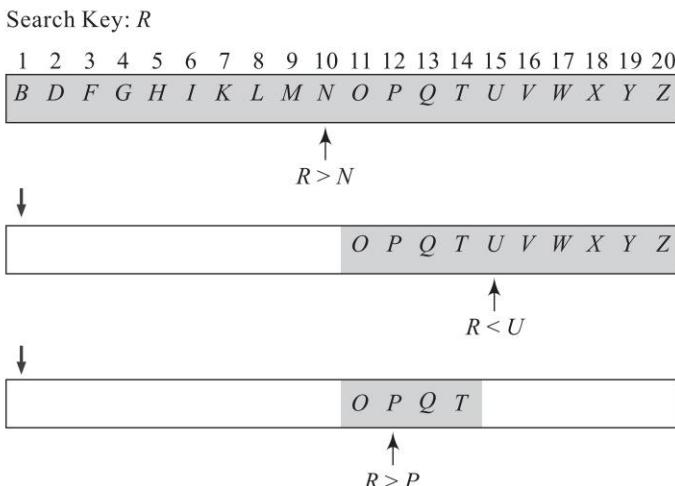
**Solution:** The binary search processes for the search keys  $H$  and  $Y$  over the list  $L$  are shown in Fig. I 15.3. The median of the list ( $mid$ ) during each step of the search process and the key comparisons made are also shown. While  $H$  calls for only 2 key comparisons,  $Y$  calls for 4 key comparisons.



**Fig. I 15.3**

**Problem 15.4** For the ordered list  $L$  shown in Illustrative Problem 15.2 trace the steps of binary search for the search key  $R$ .

**Solution:** The steps of the binary search process for the search key  $R$  is shown in Fig. I 15.4. The median of the list during each step and the key comparisons made are shown in the figure. The search is deemed unsuccessful.



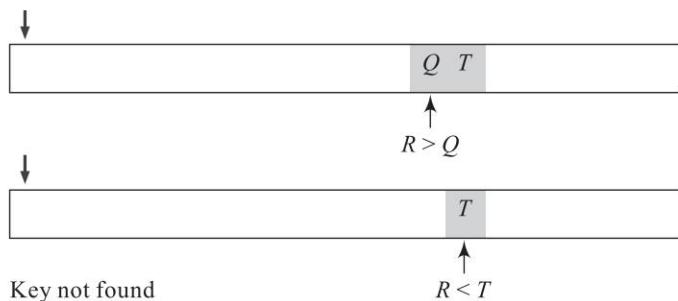


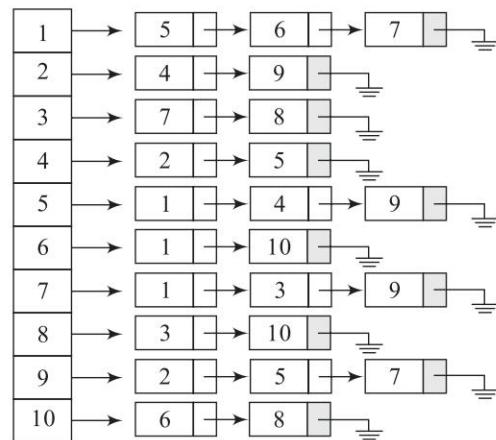
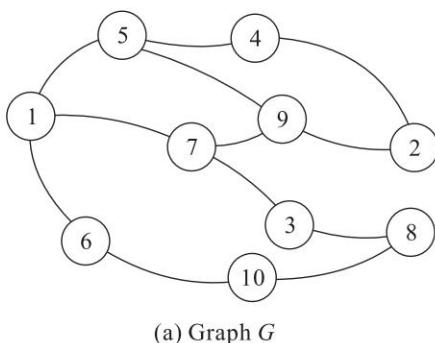
Fig. I 15.4

**Problem 15.5** Given the ordered list  $L = \{2, 4, 8, 9, 17, 36, 44, 55, 65, 100\}$  trace the steps of the Fibonacci search algorithm (Algorithm 15.6) for the search key 100.

**Solution:** The number of data elements in the list  $L$  is  $n = 10$  and  $n$  is such that,  $F_7 > (n+1)$  and  $F_6 + m = (n+1)$ . Here,  $F_7$  and  $F_6$  are the two consecutive Fibonacci numbers between which  $n$  lies and  $m$  is obviously 3. The trace of the Fibonacci search is shown below:

| Search key $K$ | $<$<br>$K = L[p]$<br>$>$ | $t$ | $p$     | $q$ | $r$ | Remarks                         |
|----------------|--------------------------|-----|---------|-----|-----|---------------------------------|
| 100            |                          |     | 5       | 3   | 2   | $n = 10$<br>$m = 3$             |
|                | $K > L[5] = 17$          |     | $5+3=8$ | 3   | 2   | Since $K > L[p]$ ,<br>$p = p+m$ |
|                | $K > L[8] = 55$          |     | 10      | 1   | 1   |                                 |
|                | $K = L[10] = 100$        |     |         |     |     | <b>Key found</b>                |

**Problem 15.6** For the undirected graph  $G$  (Fig. 9.26) shown in Example 9.1 and reproduced here for convenience, undertake Breadth first search for the key 9, by refining the Breadth first traversal algorithm (Algorithm 9.1).



**Solution:** The Breadth first search procedure is derived from Algorithm 9.1 (procedure  $BFT(S)$  where  $S$  is the start node of the graph) by replacing the procedure parameters as procedure  $BFT(S, K)$  where  $K$  is the search key. Also the statement `print (S)` is replaced by

```
if (s = K) then { print("key found"); exit(); }.
```

Unsuccessful searches may be trapped by including the statement

```
if EMPTY_QUEUE(Q) print("key not found");
```

soon after the while loop in procedure  $BFT(S, K)$ .

The trace of the breadth first search procedure for the search key 9 is shown below:

| Search key $K$ | Current vertex   | Queue $Q$ | Status of the visited flag (0/1) of the vertices (1-10) of graph $G$ |
|----------------|------------------|-----------|----------------------------------------------------------------------|
| 9              | 1 (start vertex) | 1         | 1 2 3 4 5 6 7 8 9 10<br>[1 0 0 0 0 0 0 0 0 0]                        |
|                | 1                | 5 6 7     | 1 2 3 4 5 6 7 8 9 10<br>[1 0 0 0 1 1 1 0 0 0]                        |
|                | 5                | 6 7 4 9   | 1 2 3 4 5 6 7 8 9 10<br>[1 0 0 1 1 1 1 0 1 0]                        |
|                | 6                | 7 4 9 10  | 1 2 3 4 5 6 7 8 9 10<br>[1 0 0 1 1 1 1 0 1 1]                        |
|                | 7                | 4 9 10 3  | 1 2 3 4 5 6 7 8 9 10<br>[1 0 1 1 1 1 1 0 1 1]                        |
|                | 4                | 9 10 3 2  | 1 2 3 4 5 6 7 8 9 10<br>[1 1 1 1 1 1 1 0 1 1]                        |
|                | 9                | 10 3 2    | <b>Key found</b>                                                     |

During the expansion of the current vertex, the algorithm sets the visited flag of the vertices visited to 1 before they are enqueued into the queue  $Q$ . Column 4 of the table illustrates the status of the visited flags. Once the current vertex reaches vertex 9, the key is found and the search is deemed successful.

**Problem 15.7** For the undirected graph  $G$  (Fig. 9.26) shown in Example 9.1, and reproduced in Illustrative Problem 15.6 for convenience, undertake Depth first search for the key 9, by refining the Depth first traversal algorithm (Algorithm 9.2). Trace the tree of recursive calls.

**Solution:** The recursive depth first search procedure can be derived from procedure  $DFT(S)$  (Algorithm 9.2), where  $S$  is the start node of the graph by replacing the procedure parameters as procedure  $DFT(S, K)$  where  $K$  is the search key. Also the statement `print (S)` is replaced by

```
if (s = K) then { print("key found"); exit(); }.
```

The tree of recursive calls for the depth first search of key 9 is shown in Fig. I 15.7

Each solid rectangular box indicates a call to the procedure  $DFT(S, K)$ . In the case of depth first search, as soon as a vertex is visited it is checked against the search key  $K$ . If the search key is found, the recursive procedure terminates with the message "key found".

The broken line rectangular box indicates a "pending" call to the procedure  $DFT(S, K)$ . For example, during the call  $DFT(5, 9)$ , vertex 5 has two adjacent unvisited nodes viz., 4 and 9. Since depth first search proceeds with the processing of vertex 4, vertex 9 is kept waiting.

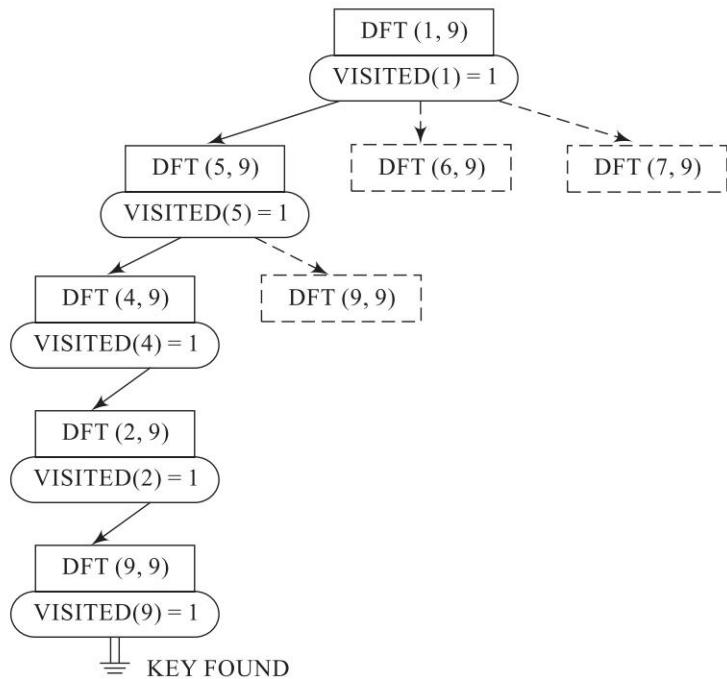


Fig. I 15.7

During the call to the procedure  $\text{DFT}(9, 9)$ , the search key is found in the graph. An unsuccessful search is signaled when all the visited flags of the vertices have been set to 1 and the search key is nowhere in sight.



## Review Questions

1. Binary search is termed uniform binary search since its decision tree holds the following characteristic:
  - the indexes of the left and the right child nodes differ by the same amount from that of the parent node
  - the list get exactly halved in each phase of the search
  - the height of the decision tree is  $\log_2 n$ .
  - each parent node of the decision tree has two child nodes.
2. In the context of binary search, state whether true or false:
  - the difference in index of all the leaf nodes from their respective parent nodes is  $2^0$ .
  - the height of the decision tree is  $n$ .
    - (i) true (ii) true
    - (i) true (ii) false
    - (i) false (ii) true
    - (i) false (ii) false
3. For a list  $L = \{K_1, K_2, K_3, \dots, K_{33}\}$ ,  $K_1 < K_2 < \dots < K_{33}$ , undertaking Fibonacci search for a key  $K$  would yield a decision tree whose root node is given by
 

|              |              |           |              |
|--------------|--------------|-----------|--------------|
| (a) $K_{16}$ | (b) $K_{17}$ | (c) $K_1$ | (d) $K_{21}$ |
|--------------|--------------|-----------|--------------|

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

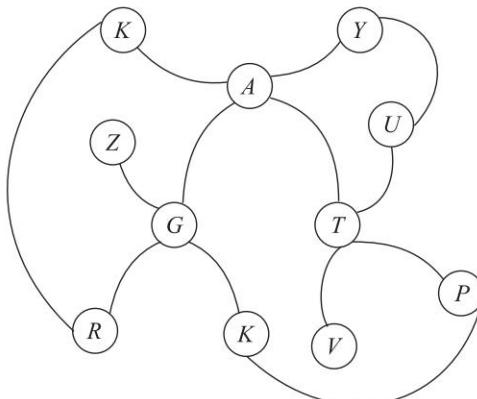


Fig. R 15.19



## Programming Assignments

1. Implement binary search and Fibonacci search algorithms ( Algorithms 15.5 and 15.6) on an ordered list. For the list  $L = \{ 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20 \}$  undertake search for the elements in the list  $\{ 3, 18, 1, 25 \}$ . Compare the number of key comparisons made during the searches.
2. Execute an online dictionary ( with a limited list of words) which makes use of interpolation search to search through the dictionary given a word. Refine the program to correct any misspelled word with the nearest and/or the correct word from the dictionary.
3.  $L$  is a linear list of data elements. Implement the list as
  - (i) a linear open addressed hash table using an appropriate hash function of your choice, and
  - (ii) an ordered list.

Search for a list of keys on the representations (i) and (ii) using (a) hashing and (b) binary search, respectively. Compare the performance of the two methods over the list  $L$ .

4. Implement a procedure to undertake search for keys  $L$  and  $M$  in the graph shown in Fig. P 15.4

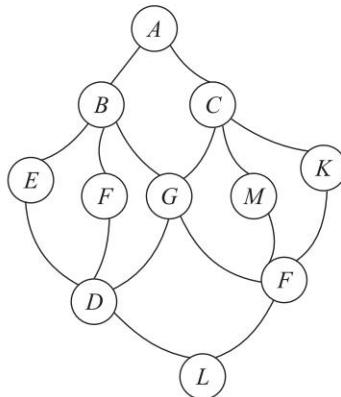


Fig. P 15.4



# INTERNAL SORTING

# 16

## Introduction

## 16.1

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

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

Examples of lists before and after sorting are shown below:

### Unsorted lists

|                                              |                                              |
|----------------------------------------------|----------------------------------------------|
| { 34, 12, 78, 65, 90, 11, 45}                | { 11, 12, 34, 45, 65, 78, 90}                |
| { tea, coffee, cocoa, milk, malt, chocolate} | { chocolate, cocoa, coffee, malt, milk, tea} |
| {n12x, m34b, n24x, a78h, g56v, m12k, k34d}   | {a78h, g56v, k34d, m12k, m34b, n12x, n24x}   |

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

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

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

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

16.1 Introduction

16.2 Bubble Sort

16.3 Insertion Sort

16.4 Selection Sort

16.5 Merge Sort

16.6 Shell Sort

16.7 Quick Sort

16.8 Heap Sort

16.9 Radix Sort

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

## Bubble Sort

## 16.2

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

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

Algorithm 16.1 illustrates the working of bubble sort.

**Algorithm 16.1:** Procedure for Bubble sort

```

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

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

```

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

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

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

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

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

## Stability and performance analysis

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

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

$$\begin{aligned} \text{Pass 1: } & \{ 7^1, 7^2, 6, 7^3 \} \\ \text{Pass 2: } & \{ 7^1, 6, 7^2, 7^3 \} \\ \text{Pass 3: } & \{ 6, 7^1, 7^2, 7^3 \} \end{aligned}$$

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

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

## Insertion Sort

## 16.3

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

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

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

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

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

|        |                     |                                      |
|--------|---------------------|--------------------------------------|
| Pass 1 | <b>(Insert 36)</b>  | { [16] 36, 4, 22, 100, 1, 54}        |
| After  | Pass 1              | { [16 36] 4, 22, 100, 1, 54}         |
| Pass 2 | <b>(Insert 4)</b>   | { [16 36] <b>4</b> , 22, 100, 1, 54} |
| After  | Pass 2              | { [4 16 36] 22, 100, 1, 54}          |
| Pass 3 | <b>(Insert 22)</b>  | { [4 16 36] <b>22</b> , 100, 1, 54}  |
| After  | Pass 3              | { [4 16 22 36] 100, 1, 54}           |
| Pass 4 | <b>(Insert 100)</b> | { [4 16 22 36] <b>100</b> , 1, 54}   |
| After  | Pass 4              | { [4 16 22 36 100] 1, 54}            |
| Pass 5 | <b>(Insert 1)</b>   | { [4 16 22 36 100] <b>1</b> , 54}    |
| After  | Pass 5              | { [1 4 16 22 36 100] 54}             |
| Pass 6 | <b>(Insert 54)</b>  | { [1 4 16 22 36 100] <b>54</b> }     |
| After  | Pass 6              | { [1 4 16 22 36 54 100]}             |

#### Algorithm 16.2: Procedure for Insertion sort

```

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

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

```

```

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

```



## Stability and performance analysis

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

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

The passes of the insertion sort are shown below:

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

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

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

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

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

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

## Selection Sort

## 16.4

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

The steps involved in the sorting process are listed below:

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

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

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

### Algorithm 16.3: Procedure for Selection sort

```

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

    for i = 1 to n-1 do
        minimum_index = FIND_MINIMUM(L, i, n);           /* find minimum element
   of the list L[i:n] and store the position index of
   the element in minimum_index*/
        swap(L[i], L[minimum_index]);
    end
end SELECTION_SORT

```

```

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

```

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

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

## Stability and performance analysis

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

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

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

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

## Merge Sort

## 16.5

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

### Two-way Merging

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

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

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

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

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

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

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

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

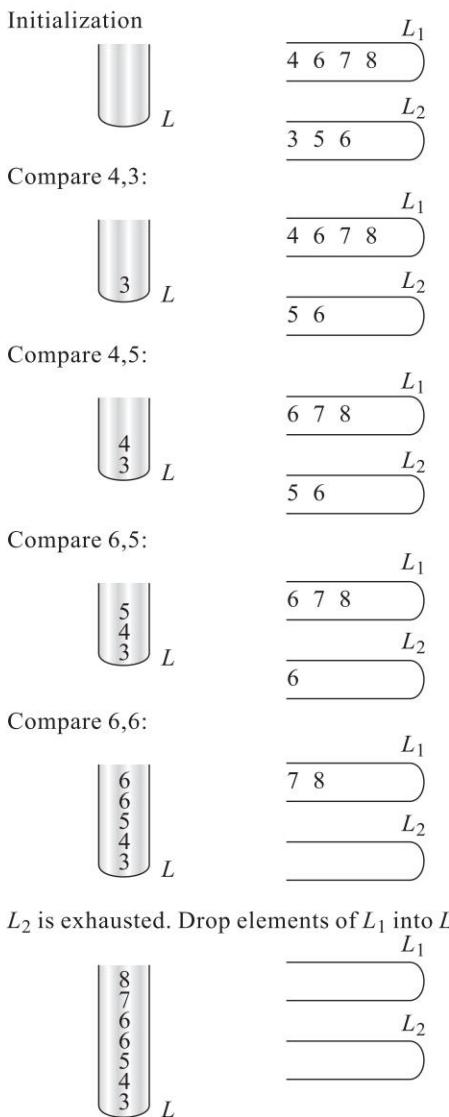


Fig. 16.1 Two-way merge

## Performance analysis

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

## *k-way merging*

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

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

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

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

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

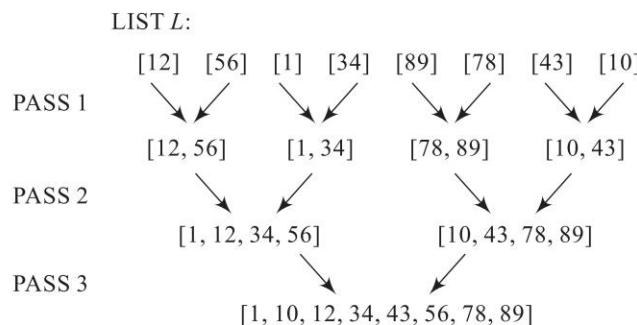
## Non recursive merge sort procedure

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

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

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

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



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

**Performance analysis** Merge sort proceeds by running several passes over the list that is to be sorted. In pass 1 sublists of size 1 are merged, in pass 2 sublists of size 2 are merged and in the  $i^{\text{th}}$  pass sublists of size  $2^{(i-1)}$  are merged. Thus one could expect a total of  $\lceil \log_2 n \rceil$  passes over the list. With the merge operation commanding  $O(n)$  time complexity, each pass of merge sort takes  $O(n)$  time. The time complexity of merge sort therefore turns out to be  $O(n \log_2 n)$ .

**Stability** Merge sort is a stable sort since the original relative orders of occurrence of repeating keys are maintained in the sorted list. Illustrative Problem 16.4 demonstrates the stability of the sort over a list.

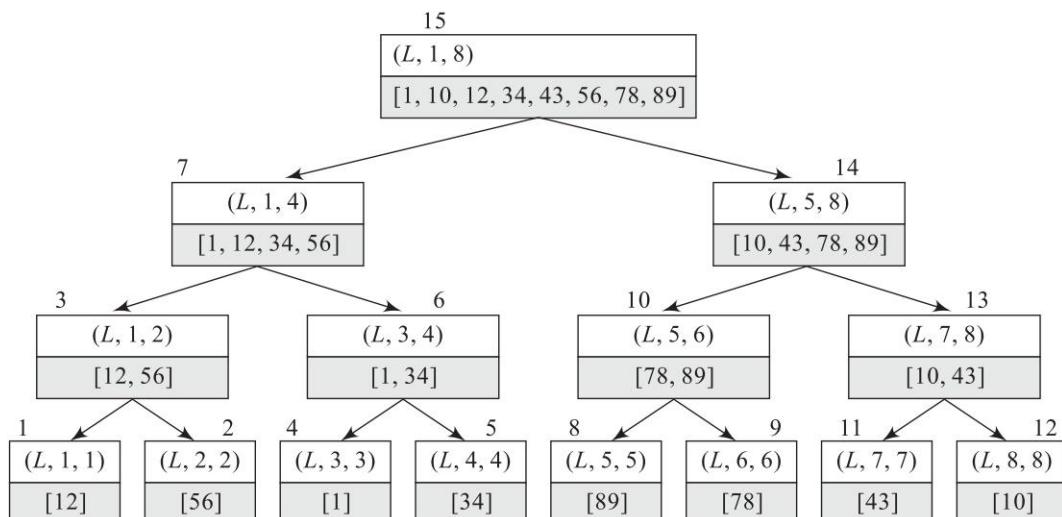
### Recursive merge sort procedure

The recursive merge sort procedure is built on the design principle of Divide and Conquer. Here, the original unordered list of elements is recursively divided roughly into two sublists until the sublists are small enough where a merge operation is done before they are combined to yield the final sorted list.

Algorithm 16.5 illustrates the recursive merge sort procedure. The procedure makes use of MERGE (Algorithm 16.4) for its merging operation.

**Example 16.9** Let us merge sort the list  $L = \{12, 56, 1, 34, 89, 78, 43, 10\}$  using Algorithm 16.5.

The tree of recursive calls demonstrating the working of the procedure on the list  $L$  is shown in Fig. 16.3. The list is recursively divided into two sublists to be merge sorted before they are merged to obtain the final sorted list. Each rectangular node of the tree indicates a procedure call to MERGE\_SORT with the parameters to the call inscribed inside the box. Beneath the parameter list is shown the output sublist obtained at the end of the execution of the procedure call.



**Fig. 16.3** Tree of recursive calls illustrating recursive merge sort of list  $L = \{12, 56, 1, 34, 89, 78, 43, 10\}$  (Example 16.9)

The invocation of `MERGE_SORT (L, 1, 8)` generates two other calls viz., `MERGE_SORT (L, 1, 4)` and `MERGE_SORT (L, 5, 8)` and so on leading to the construction of the tree. Down the tree, the procedure calls `MERGE_SORT (L, 1, 1)` and `MERGE_SORT (L, 2, 2)` in that order, are the first to terminate releasing the lists [12] and [56] respectively. This triggers the `MERGE (L, 1, 1, 2)` procedure yielding the sublist [12, 56] as the output of the procedure call `MERGE_SORT (L, 1, 2)`. Observe [12, 56] inscribed in the rectangular box 3 which corresponds to the procedure call `MERGE_SORT (L, 1, 2)`. Proceeding in a similar fashion, it is easy to build the tree and obtain the sorted sublists resulting out of each of the calls. The number marked over each rectangular node indicates the order of execution of the recursive procedure calls to `MERGE_SORT`.

With `MERGE_SORT (L, 1, 4)` yielding the sorted sublist [1, 12, 34, 56] and `MERGE_SORT (L, 5, 8)` yielding [10, 43, 78, 89], the execution of the call `MERGE (L, 1, 4, 8)` terminates the call to `MERGE_SORT (L, 1, 8)` resulting in the sorted list [1, 10, 12, 34, 43, 56, 78, 89].

**Performance analysis** Recursive merge sort follows a Divide and Conquer principle of algorithm design. Let  $T(n)$  be the time complexity of `MERGE_SORT` where  $n$  is the size of the list. The recurrence relation for the time complexity of the algorithm is given by

$$\begin{aligned} T(n) &= 2.T\left(\frac{n}{2}\right) + O(n), \quad n \geq 2 \\ &= d \end{aligned}$$

Here  $T\left(\frac{n}{2}\right)$  is the time complexity for each of the two recursive calls to `MERGE_SORT` over a list of size  $n/2$  and  $d$  is a constant.  $O(n)$  is the time complexity of merge. Framing the recurrence relation as

$$\begin{aligned} T(n) &= 2.T\left(\frac{n}{2}\right) + c.n, \quad n \geq 2 \\ &= d \end{aligned}$$

where  $c$  is a constant and solving the relation yields the time complexity  $T(n) = O(n \log_2 n)$  ( see Illustrative Problem 16.5).

## Shell Sort

## 16.6

Insertion sort (Sec. 16.3) moves items only one position at a time and therefore reports a time complexity of  $O(n^2)$  on an average. Shell sort is a substantial improvement over insertion sort in the sense that elements move in long strides rather than single steps, thereby yielding a comparatively short sub file or a comparatively well ordered sub file which quickens the sorting process.

The shell sort procedure was proposed by Donald L Shell in 1959. The general idea behind the method is to choose an increment  $h_t$  and divide a list of unordered keys  $L = \{K_1, K_2, K_3, \dots, K_j, \dots, K_n\}$  into sub lists of keys that are  $h_t$  units apart. Each of the sub lists are individually sorted (preferably insertion sorted) and gathered to form a list. This is known as a *pass*. Now we repeat the pass for any sequence of increments  $\{h_{t-1}, h_{t-2}, \dots, h_2, h_1, h_0\}$  where  $h_0$  must equal 1. The increments are kept in the diminishing order and therefore shell sort is also referred to as *diminishing increment sort*.

**Algorithm 16.4:** Procedure for Merge

```

procedure MERGE (x, first, mid, last )
    /* x[first:mid] and x[mid+1:last] are ordered lists of
       data elements to be merged into a single ordered list
       x[first:last] */

first1 = first;
last1 = mid;
first2 = mid + 1;
last2 = last;      /* set the beginning and the ending indexes of the two
                      lists into the appropriate variables*/
i = first;         /* i is the index variable for the temporary output list
                      temp*/
                      /* begin pair wise comparisons of elements from the two
                         lists*/

while (first1 ≤ last1) and (first2 ≤ last2) do
    case
        : x [first1] < x[first2]: { temp[i]= x[first1];
                                      first1 = first1 + 1;
                                      i = i + 1;
                                    }
        : x [first1] > x[first2]: { temp[i]= x[first2];
                                      first2 = first2 + 1;
                                      i = i + 1;
                                    }
        : x [first1] = x[first2]: { temp[i]= x[first1];
                                      temp[i + 1] = x[first2];
                                      first1 = first1 + 1;
                                      first2 = first2 + 1;
                                      i = i + 2;
                                    }
    end      /*end case*/
end          /* end while*/
                      /* the first list gets exhausted*/
while (first2 ≤ last2) do
    temp[i]= x[first2];
    first2 = first2 + 1;
    i = i + 1;
end          /* the second list gets exhausted*/
while (first1 ≤ last1) do
    temp[i]= x[first1];
    first1 = first1 + 1;
    i = i + 1;
end          /* copy list temp to list x*/
for j = first to last do
    x[j] = temp[j]
end
end MERGE.

```

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

```

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

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

```

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

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

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

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

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

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

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

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

**Unordered list  $L$ :**

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

**Pass 1** (increment  $h_3 = 8$ )

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

After insertion sort:

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

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

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

**Pass 2** (increment  $h_2 = 4$ )

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

After insertion sort:

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

**List  $L$  after Pass 2:**

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

**Pass 3** (increment  $h_1 = 2$ )

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

After insertion sort:

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

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

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

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

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

After insertion sort:

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

**Sorted List  $L$** 

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

**Fig. 16.4** Shell sorting of  $L = \{24, 37, 46, 11, 85, 47, 33, 66, 22, 84, 95, 55, 14, 09, 76, 35\}$  for the increment sequence  $\{8, 4, 2, 1\}$

**Unordered list  $L$ :**

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

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

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

After insertion sort:

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

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

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

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

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

After insertion sort:

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

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

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

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

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

After insertion sort:

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

**List  $L$  after Pass 2**

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

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

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

After insertion sort:

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

**Sorted List  $L$** 

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

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

**Algorithm 16.6:** Procedure for Shell Sort

```

procedure SHELL_SORT(L, n, H)
    /* L[1:n] is the unordered list of keys to be shell sorted.
       (L = {K1, K2, K3, ..., Kn}) H = {ht, ht-1, ht-2, ..., h2, h1, h0} is the sequence
       of increments */
    for each hj ∈ H do
        Insertion sort the sublist of elements in L[1:n]
        which are hj units apart, such that
        L[i] ≤ L[i + hj], for 1 ≤ i ≤ N-hj
    end
    print (L)
end SHELL_SORT.

```

**Analysis of shell sort**

The analysis of shell sort is dependent on a given choice of increments. Since there is no best possible sequence of increments that has been formulated, especially for large values of *n* (the size of the list *L*), the time complexity of shell sort is not completely resolved. In fact it has led to some interesting mathematical problems! An interested reader is referred to Sec. 5.2.1 of Donald Knuth's book (*Art of Computer Programming vol. III : Sorting and Searching, Second edition, Pearson Education, 2002*) for discussions on these results.

**Quick Sort****16.7**

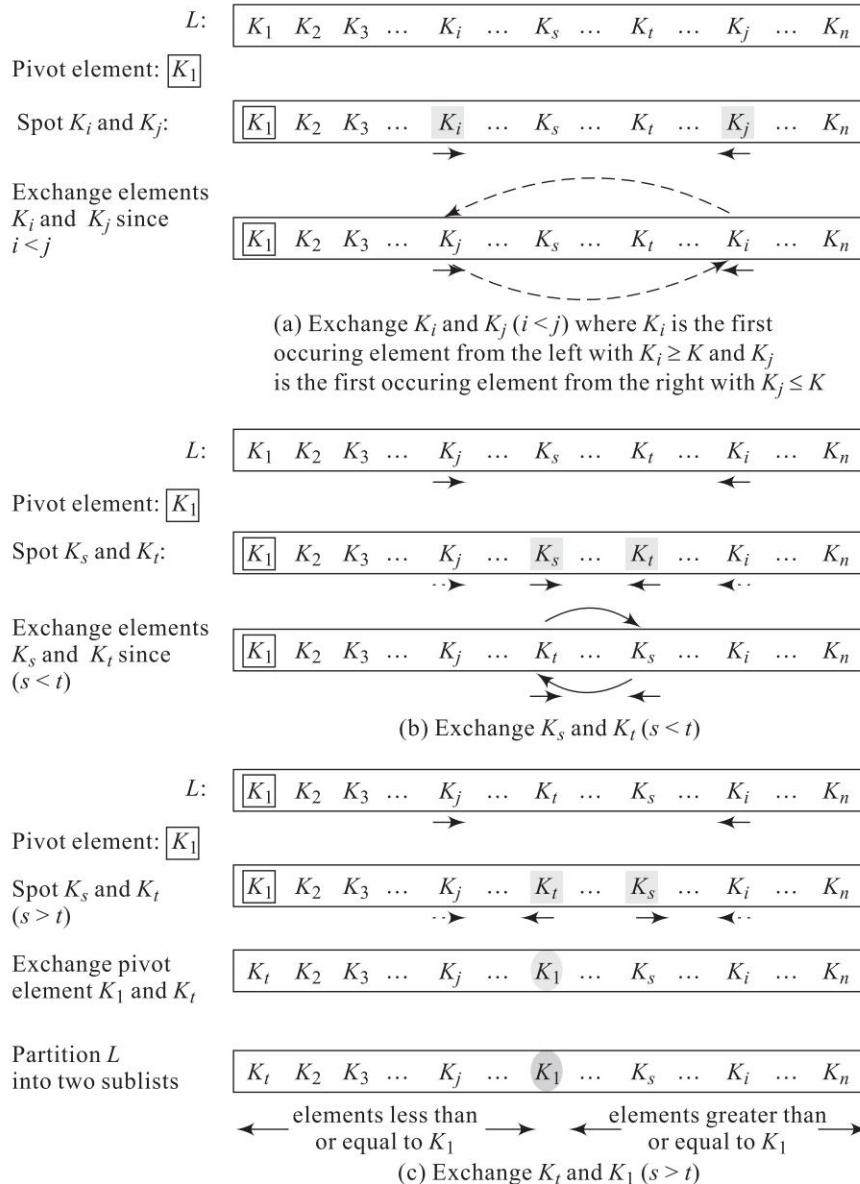
*Quick sort* procedure formulated by C.A.R. Hoare belongs to the family of *sorting by exchange or transposition* where elements that are out of order are exchanged amongst themselves to obtain the sorted list.

The procedure works on the principle of *partitioning* the unordered list into two sublists at every stage of the sorting process based on what is called a *pivot element*. The two sublists occur to the left and right of the pivot element. The pivot element determines its appropriate position in the sorted list and is therefore freed of its participation in the subsequent stages of the sorting process. Again each of the sublists are partitioned against their respective pivot elements until no more partitioning can be called for. At this stage all the elements would have determined their appropriate positions in the sorted list and quick sort is done.

**Partitioning**

Consider an unordered list *L = {K<sub>1</sub>, K<sub>2</sub>, K<sub>3</sub>, ..., K<sub>n</sub>}*. How does partitioning occur? Let us choose *K<sub>1</sub>* to be the pivot element. Now *K<sub>1</sub>* compares itself with each of the keys on a left to right encounter looking for the first key *K<sub>s</sub>*, *K<sub>s</sub> ≥ K*. Again *K* compares itself with each of the keys on a right to left encounter looking for the first key *K<sub>t</sub>*, *K<sub>t</sub> ≤ K*. If *K<sub>i</sub>* and *K<sub>j</sub>* are such that *i < j*, then *K<sub>i</sub>* and *K<sub>j</sub>* are exchanged. Figure 16.6(a) illustrates the process of exchange.

Now *K* moves ahead from position index *i* on a left to right encounter looking for a key *K<sub>s'</sub>*, *K<sub>s'</sub> ≥ K*. Again as before, *K* moves on a right to left encounter beginning from position index *j* looking for a key *K<sub>t'</sub>*, *K<sub>t'</sub> ≤ K*. As before if *s' < t'*, then *K<sub>s'</sub>* and *K<sub>t'</sub>* are exchanged and the process repeats

**Fig. 16.6 Partitioning in Quick Sort**

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

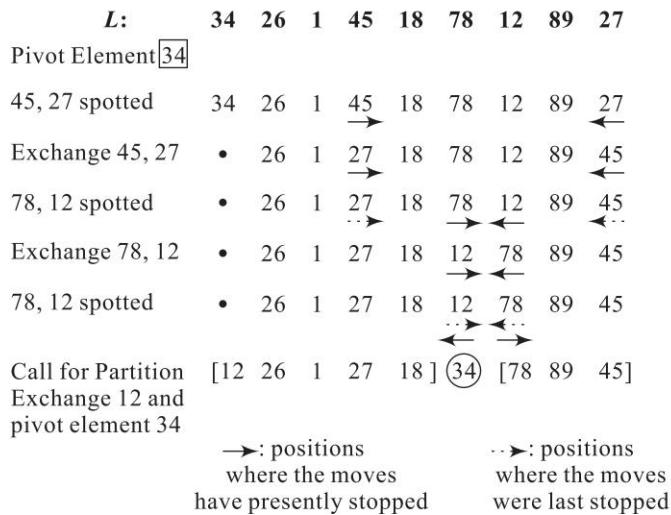
- The sublist occurring to the left of the pivot element  $K$  (now at position  $t$ ) has all its elements less than or equal to  $K$  and the sublist occurring to the right of the pivot element  $K$  has all its elements greater than or equal to  $K$ .

- (ii) The pivot element has settled down to its appropriate position which would turn out to be its rank in the sorted list.

**Example 16.12** Let  $L = \{34, 26, 1, 45, 18, 78, 12, 89, 27\}$  be an unordered list of elements. We now demonstrate the process of partitioning on the above list. Let us choose 34 as the pivot element. Figure 16.7 illustrates the snap shots of partitioning the list. Here 34 moves left to right looking for the first element that is greater than or equal to it and spots 45. Again moving from right to left looking for the first element less than or equal to 34, it spots 27. Since the position index of 45 is less than that of 27 (arrows face each other), they are exchanged.

Proceeding from the points where the moves were last stopped, 34 encounters 78 during its left to right move and encounters 12 during its right to left move. As before the arrows face each other resulting in an exchange of 78 and 12. In the next lap of the move we notice the elements 78 and 12 are spotted again but this time note that the arrows have crossed each other. This implies that the position index of 78 is greater than that of 12 calling for a partition. 34 exchanges position with 12 and the list is partitioned into two as shown.

It may be seen that all elements less than or equal to 34 have accumulated to its left and those greater than or equal to 34 have accumulated to its right. Again the pivot element 34 has settled down at position index 6 which is its rank in the sorted list.



**Fig. 16.7** Partitioning a list (Example 16.12)

## Quick sort procedure

Once the method behind partitioning is known, quick sort is nothing but repeated partitioning until every pivot element settles down to its appropriate position thereby sorting the list.

Algorithm 16.8 illustrates the quick sort procedure. The algorithm employs the Divide and Conquer principle by exploiting procedure PARTITION (Algorithm 16.7) to partition the list into two sublists and recursively calling procedure QUICK\_SORT to sort the two sublists.

Procedure PARTITION partitions the list  $L[\text{first}:\text{last}]$  at the position loc where the pivot element settles down.

**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  $\{1, 5, 15, 15, 26, 34, 76\}$ .

**Algorithm 16.8:** Procedure for Quick Sort

```

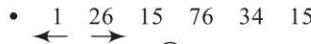
procedure QUICK_SORT(L, first, last)
    /* L[first:last] is the unordered list of elements to be
       quick sorted. The call to the procedure to sort the
       list L[1:n] would be QUICK_SORT(L, 1, n)*/
if (first < last) then
    { PARTITION(L, first, last, loc) ; /* partition the list into two
   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.

$L: \{5, 1, 26, 15, 76, 34, 15\}$

**Phase 1:** Pivot element  $\boxed{5}$

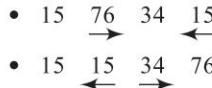


List  $L$  after partition  $(1) \boxed{5} [26 15 76 34 15]$

**Phase 2:** List  $[1]$  needs no quick sort.

Quick sort list  $[26 15 76 34 15]$

Pivot element  $\boxed{26}$



List  $L$  after partition  $(1) \boxed{5} [15 15] \boxed{26} [34 76]$

**Phase 3:** Quick sort list  $[15, 15]$

Pivot element:  $\boxed{15}$



List  $L$  after partition  $(1) \boxed{5} [15] \boxed{15} \boxed{26} [34 76]$

**Phase 4:** List  $[15]$  needs no quick sort

Quick Sort  $[34, 76]$

Pivot element  $\boxed{34}$



List  $L$  after partition  $(1) \boxed{5} \boxed{15} \boxed{15} \boxed{15} \boxed{26} \boxed{34} [76]$

The final sorted list:  $\{1, 5, 15, 15, 26, 34, 76\}$

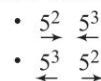
**Fig. 16.8** Snapshots of the quick sort process (Example 16.13)

**Example 16.14** Let us quick sort the list  $L = \{5^1, 5^2, 5^3\}$  where the superscripts indicate the relative orders of their occurrence in the list. Figure 16.9 illustrates the sorting process. It can be easily seen that quick sort is not stable.

Quick sort reports a worst case performance when the list is already sorted in its ascending order (see Illustrative Problem 16.6). The worst case time complexity of the algorithm is given by  $O(n^2)$ . However, quick sort reports a good average case complexity of  $O(n \log n)$ .

$L: \{5^1 5^2 5^3\}$

**Phase 1:** Pivot element  $\boxed{5^1}$



List  $L$  after portion:  $[5^3] \boxed{5^1} [5^2]$

The final sorted list  $L = \{5^3 5^1 5^2\}$   
Quick sort is unstable

**Fig. 16.9** Stability of Quick Sort

## Heap Sort

## 16.8

Heap sort is a sorting procedure belonging to the family of *sorting by selection*. This class of sorting algorithms is based on the principle of repeated selection of either the smallest or the largest key from the remaining elements of the unordered list and their inclusion in an output list. At

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

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

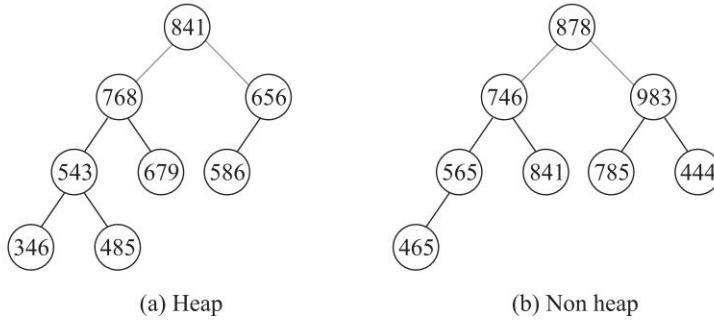
- (i) construction of a heap where the unordered list of elements to be sorted are converted into a heap, and
- (ii) repeated selection and inclusion of the root node key of the heap into the output list after reconstructing the remaining tree into a heap.

## Heap

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

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

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



**Fig. 16.10** An example heap and non heap

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

## Construction of heap

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

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

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

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

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

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

INSERT ELEMENT

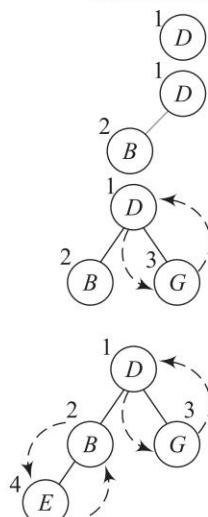
$D$

$B$

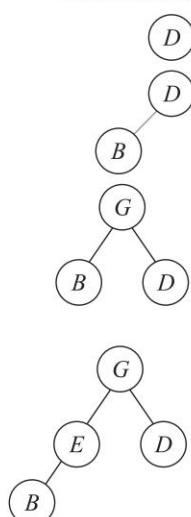
$G$

$E$

BEFORE HEAP RECONSTRUCTION



AFTER HEAP RECONSTRUCTION



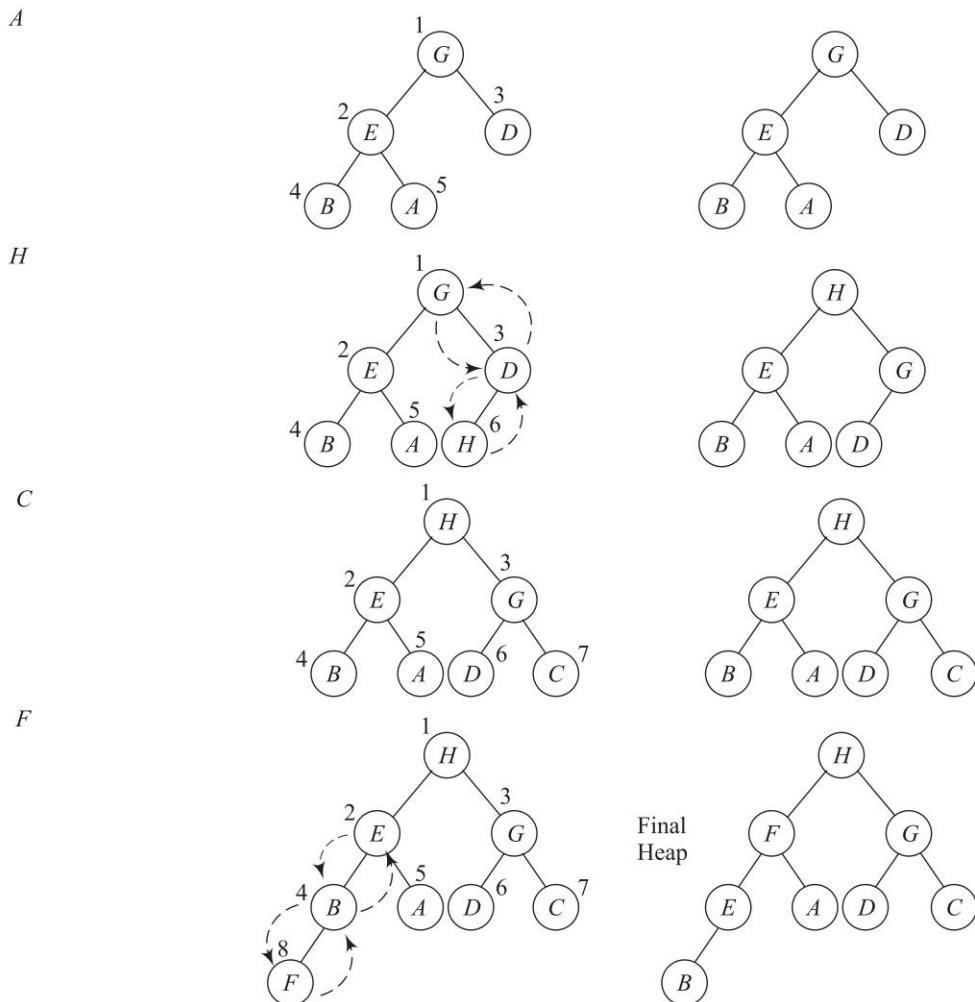


Fig. 16.11 Construction of heap (Example 16.16)

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




|                                                                |         |
|----------------------------------------------------------------|---------|
| List $L$<br>as an array $L[1 : 8]$<br>before heap construction | $L$<br> |
| List $L$<br>as an array $L[1 : 8]$<br>after heap construction  | $L$<br> |

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

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

```

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

```



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

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

```

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

```


**Heap sort procedure**

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

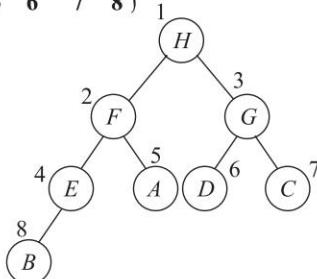
next largest numbered node in the tree which is disabled thereafter, yields a tree with  $(n-2)$  active nodes and so on. This process of heap reconstruction and outputting the root node to the output list continues until the tree is left with no active nodes. At this stage heap sort is done and the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

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

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

List  $L$ :  $\{D \ 2 \ B \ 3 \ G \ 4 \ E \ 5 \ A \ 6 \ H \ 7 \ C \ 8 \ F \ 1\}$

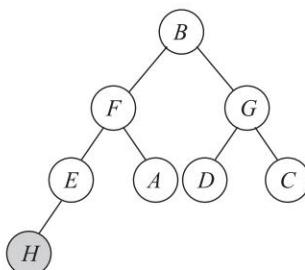
Initial heap



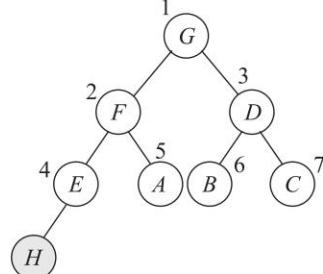
OUTPUT

H

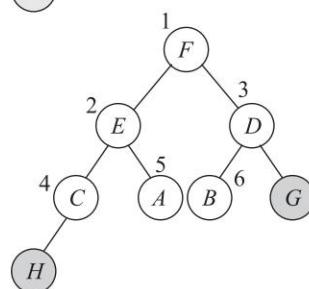
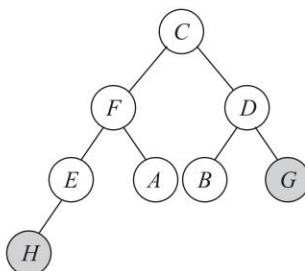
Before reconstruction  
of heap

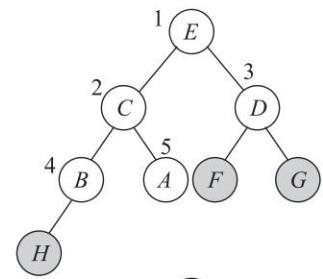
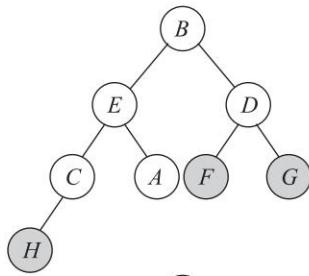
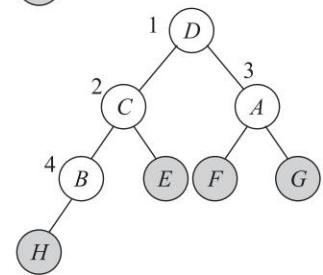
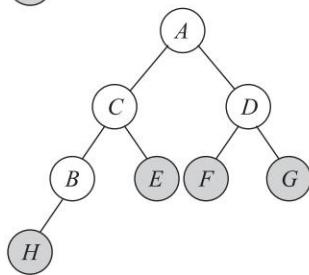
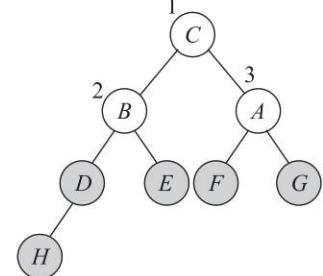
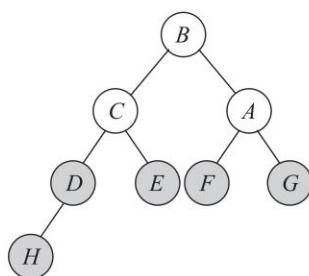
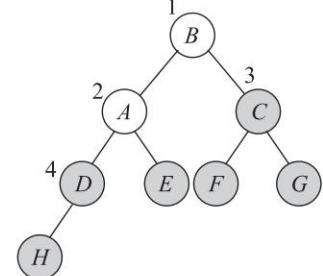
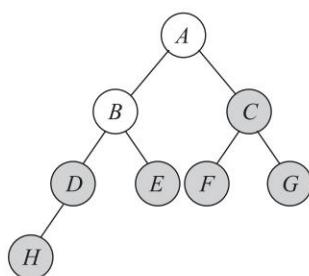
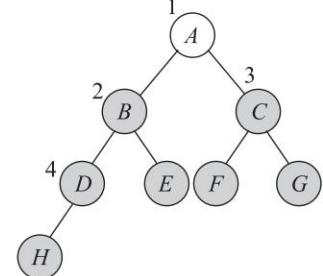
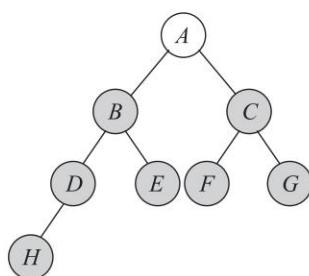


After reconstruction  
of heap

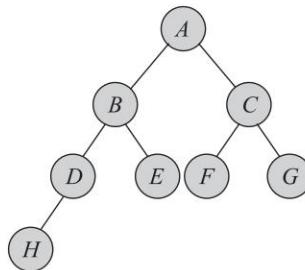


G



*F**E**D**C**B*

A



Heap sort complete  
 $L = \{A B C D E F G H\}$

**Fig. 16.13** Heap sorting of the list  $L = \{D, B, G, E, A, H, C, F\}$  (Example 16.13)

Algorithm 16.11 illustrates the heap sort procedure. The procedure CONSTRUCT\_HEAP builds the initial heap out of the list  $L$  given as input. RECONSTRUCT\_HEAP reconstructs the heap after the root node and the largest numbered node have been swapped. Procedure HEAP\_SORT accepts the list  $L[1 : n]$  as input and returns the output sorted list in  $L$  itself.

**Algorithm 16.11:** Procedures for Heap Sort

```

procedure HEAP_SORT( $L, n$ )
    /*  $L[1:n]$  is the unordered list to be sorted. The output list
       is returned in  $L$  itself*/
    CONSTRUCT_HEAP( $L, n$ ); /* construct the initial heap out of  $L[1:n]$ */
    BUILD_TREE( $L, n$ );      /* output root node and reconstruct heap*/
end HEAP_SORT.

procedure BUILD_TREE( $L, n$ )
for end_node_index =  $n$  to 2 step -1 do
{
    swap(  $L[1], L[end\_node\_index]$  ); /* swap root node with the largest
   numbered node (end node)*/
    RECONSTRUCT_HEAP( $L, end\_node\_index$ ); /*procedure for
   reconstructing a heap*/
}
end BUILD_TREE.

procedure RECONSTRUCT_HEAP( $L, end\_node\_index$  )
heap = false;
parent_index = 1;
child_index = parent_index * 2;
while (not heap) and (child_index < end_node_index) do
    right_child_index = child_index + 1;
    if (right_child_index < end_node_index) /* choose which of
  the child nodes are greater
  than or equal to the parent*/
    then
        if ( $L[right\_child\_index] > L[child\_index]$ )
        then    child_index = right_child_index;
        if ( $L[child\_index] > L[parent\_index]$ )
  
```

```

        then      {swap( L[child_index], L[parent_index]);
                    parent_index = child_index;
                    child_index = parent_index * 2;
                }
        else heap = true;
    end
end RECONSTRUCT_HEAP.

```

## Stability and performance comparison

Heap sort is an unstable sort (see Illustrative Problem 16.9). The time complexity of heap sort is  $O(n \log_2 n)$ .

## Radix Sort

## 16.9

Radix Sort belongs to the family of sorting by distribution where keys are repeatedly distributed into groups or classes based on the digits or the characters forming the key until the entire list at the end of a distribution phase gets sorted. For a long time this sorting procedure was used to sort punched cards. Radix sort is also known as *bin sort* or *bucket sort* or *digital sort*.

### Radix sort method

Given a list  $L$  of  $n$  number of keys where each key  $K$  is made up of  $l$  digits,  $K = \{K_1 K_2 K_3 \dots K_l\}$ , radix sort undertakes sorting by distributing the keys based on the digits forming the key. If the distribution proceeds from the least significant digit (LSD) onwards and progresses left digit after digit, then it is termed *LSD first sort*. We illustrate LSD first sort in this section.

Let us consider the case of LSD first sort of the list  $L$  of  $n$  keys each comprising  $l$  digits (i.e.)  $K = \{K_1 K_2 K_3 \dots K_l\}$  where each  $k_i$  is such that  $0 \leq k_i < r$ . Here  $r$  is termed as the *radix* of the key representation and hence the name radix sort. Thus if  $L$  were to deal with decimal keys then the radix would be 10. If the keys were to be octal the radix would be 8 and if they were to be hexadecimal it would be 16 and so on.

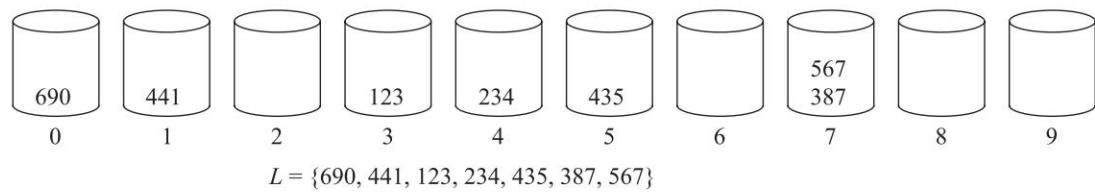
In order to understand the distribution passes of the LSD first sort procedure, we assume that  $r$  bins corresponding to the radix of the keys are present. In the first pass of the sort, all the keys of the list  $L$ , based on the value of their last digit, viz.,  $k_l$  are thrown into their respective bins. At the end of the distribution, the keys are collected in order from each of the bins. At this stage the keys are said to have been sorted based on their LSD. In the second pass we undertake a similar distribution of the keys throwing them into the bins based on their next digit,  $k_{l-1}$ . Collecting them in order from the bins yields the keys sorted according to their last but one digit. The distribution continues for  $l$  passes at the end of which the entire list  $L$  is obtained sorted.

**Example 16.18** Consider a list  $L = \{387, 690, 234, 435, 567, 123, 441\}$ . Here, the number of elements  $n = 7$ , the number of digits  $l = 3$  and radix  $r = 10$ . This means that radix sort would require 10 bins and would complete the sorting in 3 passes.

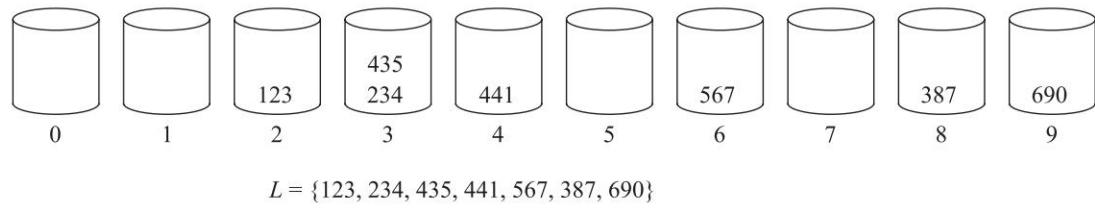
Figure 16.14 illustrates the passes of radix sort over the list. It is assumed that each key is thrown into the bin face down. At the end of each pass, when the keys are collected from each bin in order, the list of keys in each bin are turned upside down to be appended to the output list.

$$L = \{387, 690, 234, 435, 567, 123, 441\}$$

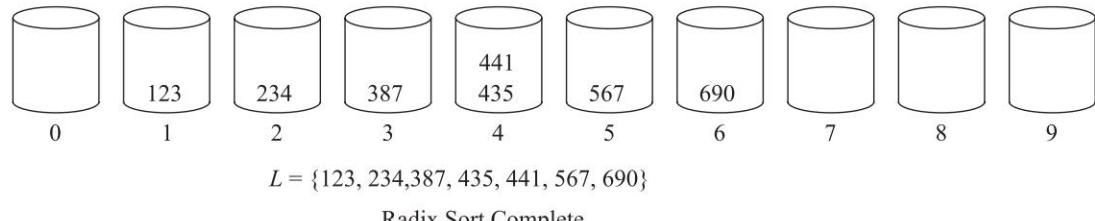
Pass 1



Pass 2

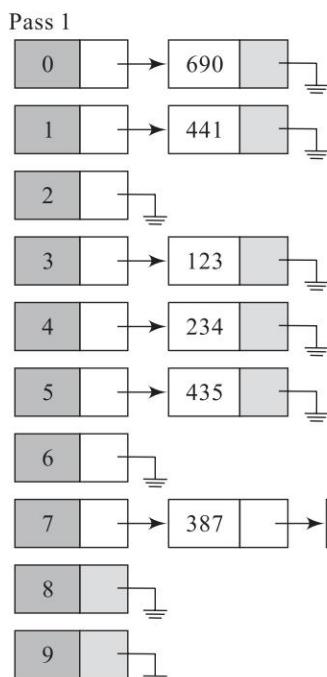


Pass 3

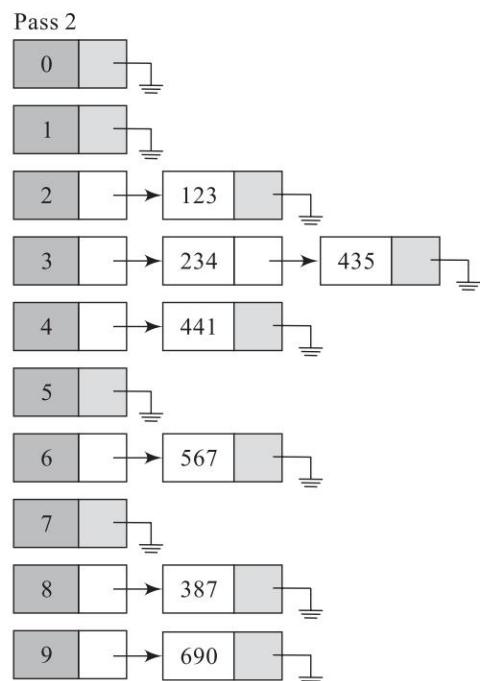


**Fig. 16.14 Radix Sort (Example 16.18)**

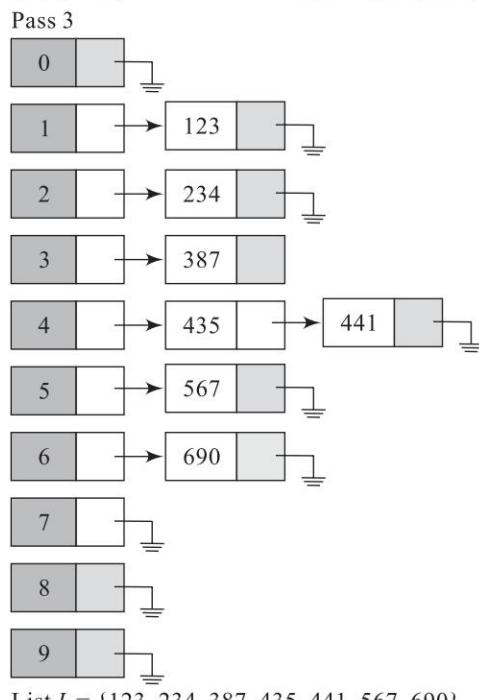
During the implementation of the radix sort procedure in the computer, it is convenient to make use of linked lists for the representation of the bins. The linked list implementation of the sort for the list shown in Example 16.18, is illustrated in Fig. 16.15. Here the bins are implemented as an array of head nodes (shaded in grey). Each of the headed linked lists representing the bins



List  $L = \{690, 441, 123, 234, 435, 387, 567\}$



List  $L = \{123, 234, 435, 441, 567, 387, 690\}$



List  $L = \{123, 234, 387, 435, 441, 567, 690\}$

**Fig. 16.15** Linked list implementation of radix sort (Example 16.18)

could be implemented as a linked queue with two pointers front and rear each pointing to the first and last node of the singly linked list respectively. At the end of each pass, the elements from each list could be appended to the output list by undertaking deletions in each of the linear queues representing the bins until they are empty.

Algorithm 16.12 illustrates the skeletal procedure for the LSD first radix sort.

**Algorithm 16.12:** Procedure for radix sort

```

procedure RADIX_SORT(L, n, r, d)
    /* radix sort sorts a list L of n keys, each comprising
       d digits with radix r*/

    Initialize each of the  $Q[0:r-1]$  linked queues representing the bins to be
    empty;

    for i = d to 1 step -1      /* for each of the d passes over the list*/
        sort the list L of n keys  $K_i = k_1 k_2 k_3 \dots k_d$  based on the digit i,
        inserting each of the keys K into the linked queue  $Q[k_i]$ ,
         $0 \leq k_i < r$ ;           /* distribute the keys into  $Q[0:(r-1)]$  based
                               on the radix value of the digits*/
        Delete the keys from the queues  $Q[0:r-1]$  in order, and append
        the elements to the output list L;
    end
    return (L);
end RADIX_SORT.

```

**Most Significant Digit first sort** Radix sort can also be undertaken by considering the most significant digits of the key first. The distribution proceeds from the most significant digit (MSD) of the key onwards and progresses right digit after digit. In such a case, the sort is termed *MSD first sort*.

MSD first sort is similar to what happens in a post office during the sorting of letters. Using the pin code, the letters are first sorted into zones, for a zone into its appropriate states, for a state into its districts and so on until they are easy enough for efficient delivery to the respective neighborhoods. Similarly, MSD first sort, distributes the keys to the appropriate bins based on the MSD. If the sub pile in each bin is small enough then it is prudent to use a non radix sort method to sort each of the sub pile and gather them together. On the other hand, if the sub pile in each bin is not small enough, then each of the sub pile is once again radix sorted based on the second digit and so on until the entire list of keys gets sorted.

## Performance analysis

The performance of the radix sort algorithm is given by  $O(d.(n+r))$  where *d* is the number of passes made over the list of keys of size *n* and radix *r*. Each pass reports a time complexity of  $O(n+r)$  and therefore for *d* passes the time complexity is given by  $O(d.(n+r))$ .



## Summary

- Sorting deals with the problem of arranging elements in a list according to the ascending or descending order.
- Internal sort refers to sorting of lists or files that can be accommodated in the internal memory of the computer. On the other hand, external sorting deals with sorting of files or lists that are too huge 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 internal sorting methods of bubble sort, insertion sort, selection sort merge sort, quick sort, shell sort, heap sort and radix sort are discussed.
- Bubble sort belongs to the family of sorting by exchange or transposition. In each pass, the elements are compared pair wise until the largest key amongst the participating elements bubbles to the end of the list.
- Insertion sort belongs to the family of sorting by insertion. The sorting method is based on the principle that a new key  $K$  is inserted at its appropriate position in an already sorted sub list.
- Selection sort is built on the principle of selecting the minimum element of the list and exchanging it with the element in the first position of the list, and so on until the whole list gets sorted.
- Merge sort belonging to the family of sorting by merge makes use of the principle of merge to sort an unordered list. The sorted sublists of the original lists are merged to obtain the final sorted list.
- Shell sort divides the list into sub lists of elements making use of a sequence of increments and insertion sorts each sub list, before gathering them for the subsequent pass. The passes are repeated for each of the increment until the entire list gets sorted.
- Quick sort belongs to the family of sorting by exchange. The procedure works on the principle of partitioning the unordered list into two sublists at every stage of the sorting process based on the pivot element and recursively quick sorting the sublists.
- Heap sort belongs to the family of sorting by selection. It is based on the principle of repeated selection of either the smallest or the largest key from the remaining elements of the unordered list constructed as a heap, for inclusion in the output list.
- Radix sort belongs to the family of sorting by distribution and is classified as LSD first sort and MSD first sort. The sorting of the keys is undertaken digit wise in  $d$  passes where  $d$  is the number of digits in the keys over a radix  $r$ .



## Illustrative Problems

**Problem 16.1** Trace bubble sort algorithm on the list  $L = \{K, Q, A, N, C, A, P, T, V, B\}$ . Verify stability of bubble sort over  $L$ .

**Solution:** The partially sorted lists at the end of the respective passes of the search is shown below. Repeated elements in the list have been superscripted with indexes.

|               |                                        |
|---------------|----------------------------------------|
| Unsorted list | $\{K, Q, A^1, N, C, A^2, P, T, V, B\}$ |
| Pass 1        | $\{K, A^1, N, C, A^2, P, Q, T, B, V\}$ |
| Pass 2        | $\{A^1, K, C, A^2, N, P, Q, B, T, V\}$ |
| Pass 3        | $\{A^1, C, A^2, K, N, P, B, Q, T, V\}$ |
| Pass 4        | $\{A^1, A^2, C, K, N, B, P, Q, T, V\}$ |
| Pass 4        | $\{A^1, A^2, C, K, B, N, P, Q, T, V\}$ |
| Pass 5        | $\{A^1, A^2, C, B, K, N, P, Q, T, V\}$ |
| Pass 6        | $\{A^1, A^2, B, C, K, N, P, Q, T, V\}$ |
| Pass 7        | $\{A^1, A^2, B, C, K, N, P, Q, T, V\}$ |
| Pass 8        | $\{A^1, A^2, B, C, K, N, P, Q, T, V\}$ |
| Pass 9        | $\{A^1, A^2, B, C, K, N, P, Q, T, V\}$ |

Since the relative order of positions of equal keys remain unaffected even after the sort, bubble sort is stable over  $L$ .

**Problem 16.2** Trace the passes of insertion sort on the following lists:

- (i)  $\{H, K, M, N, P\}$
- (ii)  $\{P, N, M, K, H\}$

Compare their performance in terms of the comparisons made.

**Solution:** The lists at the end of each pass of insertion sort are shown in Table I 16.2. It may be observed that while list (i) is already in its ascending order, list (ii) is in its descending order. The sorted sublists are shown in brackets. The number of comparisons made in each of the passes is shown in bold. While list (i) needs to make a total of 4 comparisons, list (ii) needs to make a total of 10 comparisons to sort themselves using insertion sort.

**Table I 16.2**

| Pass | Insertion sort of { H, K, M, N, P } | Number of Comparisons | Insertion sort of { P, N, M, K, H } | Number of comparisons |
|------|-------------------------------------|-----------------------|-------------------------------------|-----------------------|
| 1    | { [H K] M N P }                     | <b>1</b>              | { [N P] M K H }                     | <b>1</b>              |
| 2    | { [H K M] N P }                     | <b>1</b>              | { [M N P] K H }                     | <b>2</b>              |
| 3    | { [H K M N] P }                     | <b>1</b>              | { [K M N P] H }                     | <b>3</b>              |
| 4    | { [H K M N P] }                     | <b>1</b>              | { [H K M N P] }                     | <b>4</b>              |

**Problem 16.3** Undertake 3-way merge for the lists shown below:

$$L_1 = \{F, J, L\}, \quad L_2 = \{F, H, M, N, P\}, \quad L_3 = \{G, M\}$$

**Solution:** The snapshots of the 3-way merge of the lists into the list  $L$  is shown in Fig. I 16.3. At every stage three elements from each of the lists are compared and the smallest of them is dropped into the list. At the end of step 5,  $L_1$  gets exhausted and at the end of step 6,  $L_3$  also gets exhausted. In the last step, the remaining elements in list  $L_2$  are merely flushed into list  $L$ .

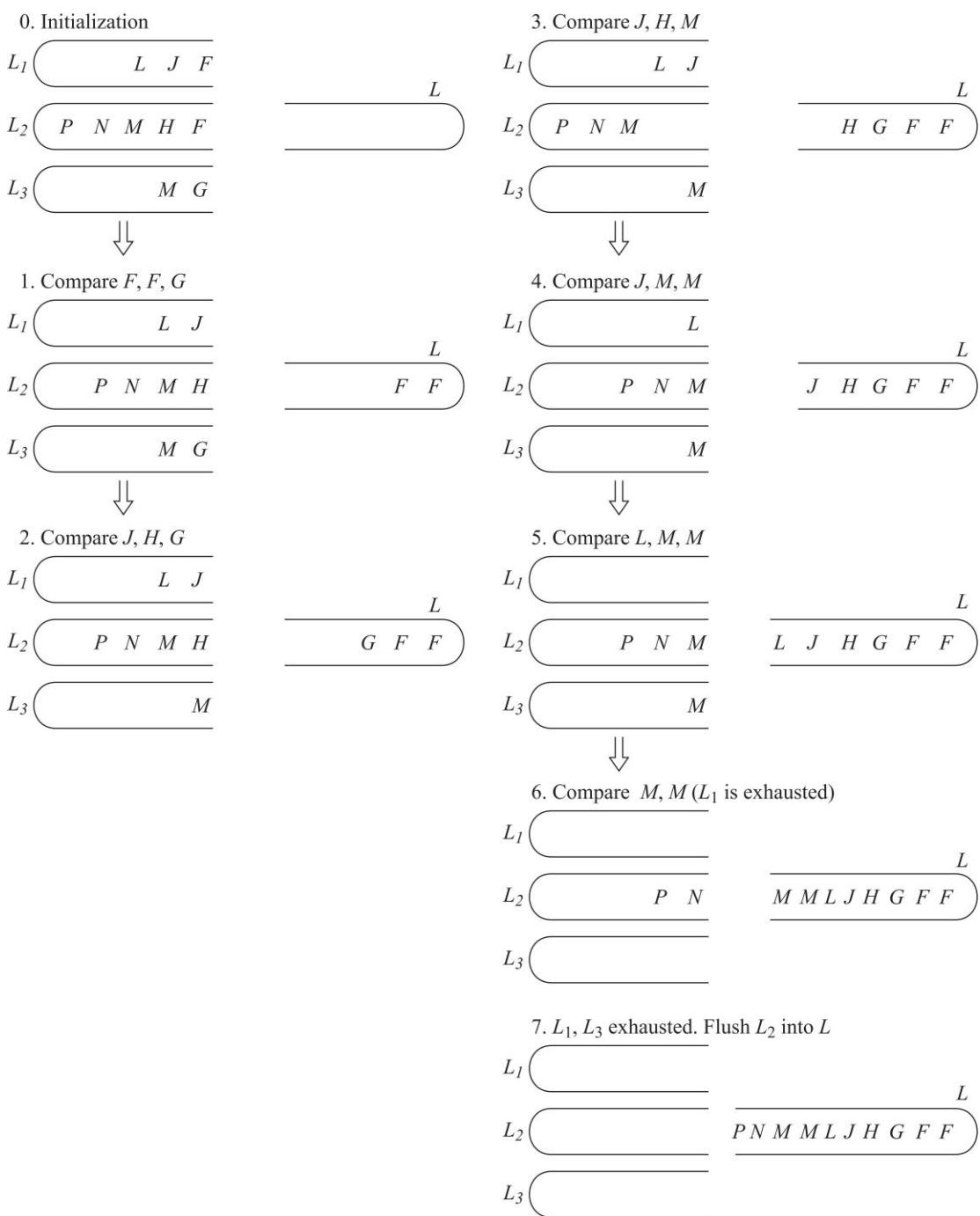


Fig. I 16.3

**Problem 16.4** Undertake non recursive merge sort for the list  $L = \{78, 78, 78, 1\}$  and check for the stability of the sort.

**Solution:** We undertake the non recursive formulation of merge sort procedure for the list  $L = \{78^1, 78^2, 78^3, 1\}$ . The repeated keys in the list  $L$  are superscripted to track their orders of occurrence in the list. Figure I16.4 shows the passes over the list  $L$ . The final sorted list verifies that merge sort is stable.

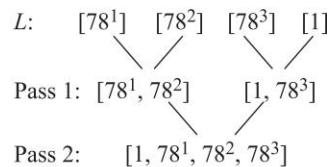


Fig. I 16.4

**Problem 16.5** Solve the recurrence relation for the time complexity of merge sort given in Sec. 16.5 assuming the size of the list  $n = 2^k$ .

**Solution:** The recurrence relation is given by

$$\begin{aligned} T(n) &= 2.T\left(\frac{n}{2}\right) + c.n, \quad n \geq 2 \\ &= d \end{aligned}$$

Solving the relation results in the following steps:

$$T(n) = 2.T\left(\frac{n}{2}\right) + c.n \quad (i)$$

$$\begin{aligned} &= 2\left(2.T\left(\frac{n}{4}\right) + c.\frac{n}{2}\right) + c.n \\ &= 2^2.T\left(\frac{n}{4}\right) + 2.c.n \quad (ii) \end{aligned}$$

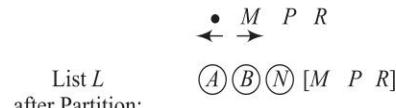
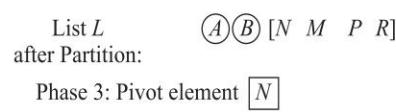
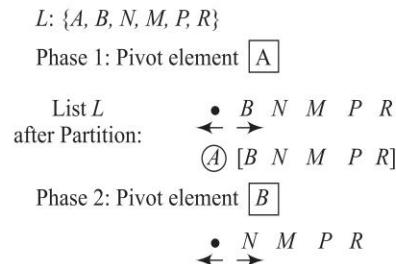
$$= 2^3.T\left(\frac{n}{8}\right) + 3.c.n \quad (iii)$$

In step (k)  $T(n)$  is obtained as,

$$\begin{aligned} T(n) &= 2^k.T\left(\frac{n}{2^k}\right) + k.c.n \\ &= n.T(1) + \log_2 n.c.n \quad (\because n = 2^k, \quad k = \log_2 n) \\ &= n.d + c.n.\log_2 n \\ &= O(n.\log_2 n) \end{aligned}$$

**Problem 16.6** Quick sort the list  $L=\{A, B, N, M, P, R\}$ . What are your observations? How can the observations help you in determining the worst case complexity of quick sort?

**Solution:** The quick sort process is demonstrated in Fig. I 16.6. Since the list is already in its ascending order, during each phase of the sort, the elements in the order given get thrown out one by one during the subsequent



In phases 4, 5 and 6 elements  $M, P$  and  $R$  get freed due to partitioning yielding the sorted list  $[A \ B \ N \ M \ P \ R]$

Fig. I 16.6

partitions. In other words with each partition the size of the list decrements by 1. The quick sort procedure is therefore recursively called, for lists of sizes  $n, (n-1), (n-2) \dots 3 \ 2 \ 1$ . Hence the worst case time complexity is given by  $O( n + (n-1) + (n-2) + \dots + 3 + 2 + 1 ) = O(n^2)$ .

**Problem 16.7** Discuss a procedure to obtain the rank of an element  $K$  in an array  $\text{LIST}[1 : n]$ . How can procedure `PARTITION` (Algorithm 16.7) be effectively used for the same problem? What are the time complexities of the methods discussed?

**Solution:** A direct method to obtain the rank of an element in an array  $\text{LIST}[1:n]$  is to sort the list and search for the element  $K$  in the sorted list. The time complexity of the procedure in such a case would be  $O(n \log_2 n)$  since the best sorting algorithm reports a time complexity of  $O(n \log_2 n)$ .

In the case of employing procedure `PARTITION` for the problem,  $K$  is first compared with the pivot element ( $P$ ) that gets dropped off the list during the first partition. If  $K = P$  then the problem is done. The index of  $P$  in the list  $\text{LIST}$  would be the rank of the element  $K$ . On the other hand, if  $(K < P)$  or  $(K > P)$  then it would only call for searching for the rank of  $K$  in any one of the sublists occurring to the left or right of  $P$  by repeatedly invoking procedure `PARTITION`. Hence in this case, the time complexity would be  $O(n)$  in the worst case.

**Problem 16.8** Trace procedure `CONSTRUCT_HEAP` (Algorithm 16.10) over the list  $L[1:5] = \{ 12, 45, 21, 67, 34 \}$ .

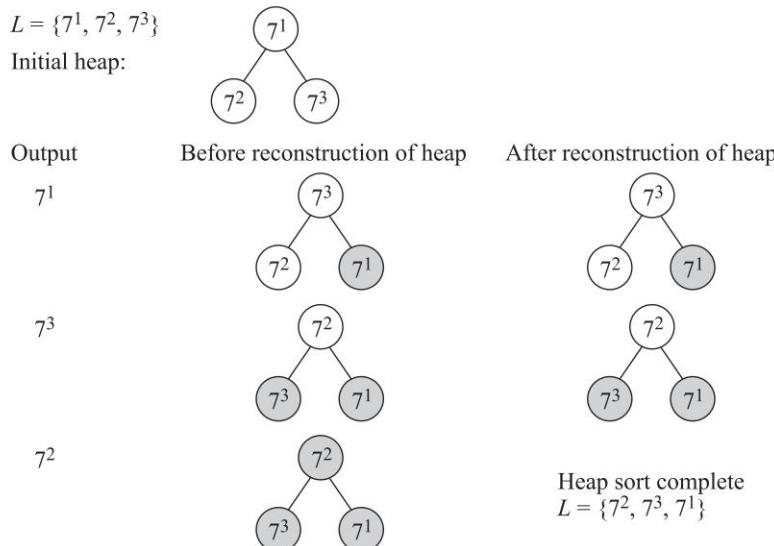
**Solution:** Table I 16.8 illustrates the trace of the procedure `CONSTRUCT_HEAP` which invokes procedure `INSERT_HEAP` repeatedly for the elements belonging to  $L[2:5] = \{ 45, 21, 67, 34 \}$ . At the end of the execution, procedure `CONSTRUCT_HEAP` yields the list  $L[1:5] = \{ 67, 45, 21, 12, 34 \}$  which is the heap.

**Table I 16.8**

| Call to <code>INSERT_HEAP</code> | <code>child_index</code> | <code>parent_index</code> | $L[1:5]$             | Remarks                                   |
|----------------------------------|--------------------------|---------------------------|----------------------|-------------------------------------------|
|                                  |                          |                           | {12, 45, 21, 67, 34} | Initialization                            |
| <code>INSERT_HEAP (L, 2)</code>  | 2                        | 1                         | {45, 12, 21, 67, 34} | $L[1] < L[2]$ swap ( $L[1], L[2]$ ) done. |
|                                  | 1                        | 0                         | {45, 12, 21, 67, 34} |                                           |
| <code>INSERT_HEAP (L, 3)</code>  | 3                        | 1                         | {45, 12, 21, 67, 34} | $L[1] > L[3]$ no swap                     |
| <code>INSERT_HEAP (L, 4)</code>  | 4                        | 2                         | {45, 67, 21, 12, 34} | $L[2] < L[4]$ swap ( $L[2], L[4]$ ) done. |
|                                  | 2                        | 1                         | {67, 45, 21, 12, 34} | $L[1] < L[2]$ swap ( $L[1], L[2]$ ) done. |
|                                  | 1                        | 0                         | {67, 45, 21, 12, 34} |                                           |
| <code>INSERT_HEAP (L, 5)</code>  | 5                        | 2                         | {67, 45, 21, 12, 34} | $L[2] > L[5]$ no swap                     |

**Problem 16.9** Test for the stability of heap sort on the list  $L = \{7^1, 7^2, 7^3\}$ .

**Solution:** Figure I 16.9 demonstrates the heap sort process on  $L$ . The final sorted list  $L = \{7^3, 7^2, 7^1\}$ . This verifies that heap sort is unstable.



**Fig. I 16.9**

**Problem 16.10** Radix sort the list  $L = \{001, 101, 010, 000, 111, 110, 011, 100\}$ .

**Solution:** The list  $L$  commands the following parameters:  $n = 8$ ,  $d = 3$  and  $r = 2$ . The radix sort process is shown in Fig. I 16.10.

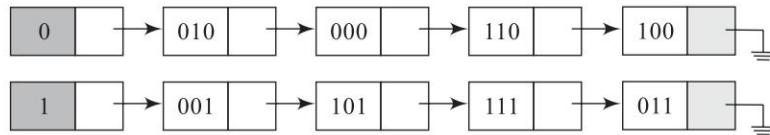
**Problem 16.11** Selection sort the list  $L = \{H, V, A, T, L, M, K\}$ .

**Solution:** The sorting steps are shown below. The minimum element in each pass is shown in bold. The arrows indicate the swap of the minimum element with that in the first position of the sub list considered for the pass. The elements in gray are indicative of the exclusion of elements from the pass.

| Pass                                          | List $L$ (During Pass)                                                                                                             | List $L$ (After Pass)     |
|-----------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------|---------------------------|
| 1                                             | $\{H, \underset{\swarrow}{V}, \underset{\nearrow}{A}, T, P, M, K\}$                                                                | $\{A, V, H, T, P, M, K\}$ |
| 2                                             | $\{A, \underset{\swarrow}{V}, \underset{\nearrow}{H}, T, P, M, K\}$                                                                | $\{A, H, V, T, P, M, K\}$ |
| 3                                             | $\{A, \underset{\swarrow}{H}, \underset{\nearrow}{V}, T, P, M, \underset{\nearrow}{K}\}$                                           | $\{A, H, K, T, P, M, V\}$ |
| 4                                             | $\{A, \underset{\swarrow}{H}, \underset{\nearrow}{K}, \underset{\swarrow}{T}, \underset{\nearrow}{P}, \underset{\nearrow}{M}, V\}$ | $\{A, H, K, M, P, T, V\}$ |
| 5                                             | $\{A, \underset{\swarrow}{H}, \underset{\nearrow}{K}, M, \underset{\swarrow}{P}, \underset{\nearrow}{T}, V\}$                      | $\{A, H, K, M, P, T, V\}$ |
| 6                                             | $\{A, \underset{\swarrow}{H}, \underset{\nearrow}{K}, M, P, \underset{\swarrow}{T}, \underset{\nearrow}{V}\}$                      | $\{A, H, K, M, P, T, V\}$ |
| <b>Sorted list:</b> $\{A, H, K, M, P, T, V\}$ |                                                                                                                                    |                           |

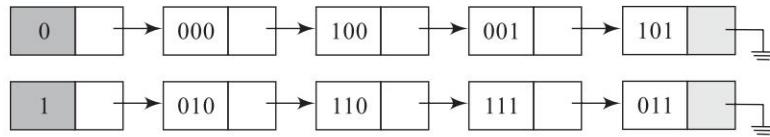
$$L = \{001, 101, 010, 000, 111, 110, 011, 100\}$$

Phase 1:



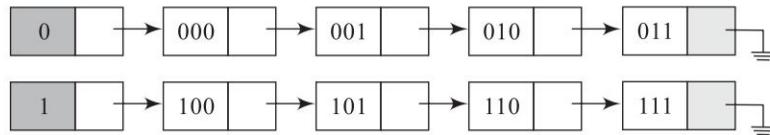
$$L = \{010, 000, 110, 100, 001, 101, 111, 011\}$$

Phase 2:



$$L = \{000, 100, 001, 101, 010, 110, 111, 011\}$$

Phase 3:



$$L = \{000, 001, 010, 011, 100, 101, 110, 111\}$$

The final radix sorted list

$$L = \{000, 001, 010, 011, 100, 101, 110, 111\}$$

**Fig. I 16.11**

**Problem 16.12** Test whether shell sort is stable on the list  $L = \{7, 5^1, 5^2, 5^3, 5^4, 5^5, 5^6, 5^7, 5^8, 5^9\}$  for a sequence of increments  $\{4, 2, 1\}$ . The repeated occurrences of element 5 have been superscripted with their orders of occurrence.

**Solution:** The trace of shell sort on the list  $L$  is shown in Fig. I 16.12. It is unstable.

| Unordered list $L$ :          |       |                                |       |                   |       |                   |       |       |          |
|-------------------------------|-------|--------------------------------|-------|-------------------|-------|-------------------|-------|-------|----------|
| $K_1$                         | $K_2$ | $K_3$                          | $K_4$ | $K_5$             | $K_6$ | $K_7$             | $K_8$ | $K_9$ | $K_{10}$ |
| 7                             | $5^1$ | $5^2$                          | $5^3$ | $5^4$             | $5^5$ | $5^6$             | $5^7$ | $5^8$ | $5^9$    |
| Pass 1 (increment $h_2 = 4$ ) |       |                                |       |                   |       |                   |       |       |          |
| $(K_1 \quad K_5 \quad K_9)$   |       | $(K_2 \quad K_6 \quad K_{10})$ |       | $(K_3 \quad K_7)$ |       | $(K_4 \quad K_8)$ |       |       |          |
| $(7 \quad 5^4 \quad 5^8)$     |       | $(5^1 \quad 5^5 \quad 5^9)$    |       | $(5^2 \quad 5^6)$ |       | $(5^3 \quad 5^7)$ |       |       |          |
| After insertion sort:         |       |                                |       |                   |       |                   |       |       |          |
| $(5^4 \quad 5^8 \quad 7)$     |       | $(5^1 \quad 5^5 \quad 5^9)$    |       | $(5^2 \quad 5^6)$ |       | $(5^3 \quad 5^7)$ |       |       |          |
| List $L$ after Pass 2         |       |                                |       |                   |       |                   |       |       |          |
| $K_1$                         | $K_2$ | $K_3$                          | $K_4$ | $K_5$             | $K_6$ | $K_7$             | $K_8$ | $K_9$ | $K_{10}$ |
| $5^4$                         | $5^1$ | $5^2$                          | $5^3$ | $5^8$             | $5^5$ | $5^6$             | $5^7$ | 7     | $5^9$    |

| <b>Pass 2 (increment <math>h_1 = 2</math>)</b> |       |       |       |        |        |       |       |       |           |
|------------------------------------------------|-------|-------|-------|--------|--------|-------|-------|-------|-----------|
| $(K_1$                                         | $K_3$ | $K_5$ | $K_7$ | $K_9)$ | $(K_2$ | $K_4$ | $K_6$ | $K_8$ | $K_{10})$ |
| $(5^4$                                         | $5^2$ | $5^8$ | $5^6$ | $7)$   | $(5^1$ | $5^3$ | $5^5$ | $5^7$ | $5^9)$    |
| After insertion sort:                          |       |       |       |        |        |       |       |       |           |
| $(5^4$                                         | $5^2$ | $5^8$ | $5^6$ | $7)$   | $(5^1$ | $5^3$ | $5^5$ | $5^7$ | $5^9)$    |
| List L after Pass 3:                           |       |       |       |        |        |       |       |       |           |
| $K_1$                                          | $K_2$ | $K_3$ | $K_4$ | $K_5$  | $K_6$  | $K_7$ | $K_8$ | $K_9$ | $K_{10}$  |
| $5^4$                                          | $5^1$ | $5^2$ | $5^3$ | $5^8$  | $5^5$  | $5^6$ | $5^7$ | $7$   | $5^9$     |
| <b>Pass 3 (increment <math>h_0 = 1</math>)</b> |       |       |       |        |        |       |       |       |           |
| $K_1$                                          | $K_2$ | $K_3$ | $K_4$ | $K_5$  | $K_6$  | $K_7$ | $K_8$ | $K_9$ | $K_{10}$  |
| $5^4$                                          | $5^1$ | $5^2$ | $5^3$ | $5^8$  | $5^5$  | $5^6$ | $5^7$ | $5^9$ | $7$       |
| After insertion sort:                          |       |       |       |        |        |       |       |       |           |
| $5^4$                                          | $5^1$ | $5^2$ | $5^3$ | $5^8$  | $5^5$  | $5^6$ | $5^7$ | $5^9$ | $7$       |
| <b>Sorted List L</b>                           |       |       |       |        |        |       |       |       |           |
| $5^4$                                          | $5^1$ | $5^2$ | $5^3$ | $5^8$  | $5^5$  | $5^6$ | $5^7$ | $5^9$ | $7$       |

Fig. I 16.12



## Review Questions

- Which of the following is unstable sort?  
 (a) Quick sort      (b) Insertion sort      (c) Bubble sort      (d) Merge sort
- The worst case time complexity of quick sort is  
 (a)  $O(n)$       (b)  $O(n^2)$       (c)  $O(n \log n)$       (d)  $O(n^3)$
- Which among the following belongs to the family of sorting by selection?  
 (a) merge sort      (b) quick sort      (c) heap sort      (d) shell sort
- Which among the following actions does not occur during the 2 – way merge of two lists  $L_1$  and  $L_2$  into the output list  $L$ ?  
 (a) If both  $L_1$  and  $L_2$  get exhausted, then the merge is done  
 (b) If  $L_1$  gets exhausted before  $L_2$ , then simply output the remaining elements of  $L_2$  into  $L$  and the merge is done.  
 (c) If  $L_2$  gets exhausted before  $L_1$ , then simply output the remaining elements of  $L_1$  into  $L$  and the merge is done.  
 (d) If one of the two lists ( $L_1$  or  $L_2$ ) gets exhausted, with the other still containing elements, then the merge is done.
- For a list  $L = \{ 7, 3, 9, 1, 8 \}$  the output list at the end of Pass 1 of bubble sort would yield  
 (a)  $\{ 3, 7, 1, 9, 8 \}$       (b)  $\{ 3, 7, 1, 8, 9 \}$       (c)  $\{ 3, 1, 7, 9, 8 \}$       (d)  $\{ 1, 3, 7, 8, 9 \}$
- Distinguish between internal sorting and external sorting.
- When is a sorting process said to be stable?
- Why is bubble sort stable?
- What is k-way merging?
- What is the time complexity of selection sort?
- Distinguish between a heap and a binary search tree. Give an example.
- What is the principle behind Shell sort?

13. When is radix sort termed LSD first sort?
14. What is the principle behind the Quick sort procedure?
15. What is the time complexity of merge sort?
16. Can bubble sort ever perform better than quick sort? Is so, list a case.
17. Trace (i) bubble sort (ii) insertion sort and (iii) selection sort on the list  $L = \{H, V, A, X, G, Y, S\}$
18. Demonstrate 3-way merging on the lists:  
 $L_1 = \{123, 678, 345, 225, 890, 345, 111\}$ ,  $L_2 = \{345, 123, 654, 789, 912, 144, 267, 909, 111, 324\}$  and  $L_3 = \{567, 222, 111, 900, 545, 897\}$
19. Trace Quick sort on the list  $L = \{11, 34, 67, 78, 78, 78, 99\}$ . What are your observations?
20. Trace Radix sort on the following list:  
 $L = \{5678, 2341, 90, 3219, 7676, 8704, 4561, 5000\}$
21. Undertake heap sort for the list  $L$  shown in Review Questions 17 (Chapter 16).



## Programming Assignments

1. Implement (i) Bubble sort and (ii) Insertion sort in a language of your choice. Test for the performance of the two algorithms on input files with elements already sorted in (i) descending order and (ii) ascending order. Record the number of comparisons. What are your observations?
2. Implement Quick sort algorithm. Enhance the algorithm to test for its stability.
3. Implement the non recursive version of merge sort. Enhance the implementation to test for its stability.
4. Implement the LSD first and MSD first version of Radix sort for alphabetical keys.
5. Implement Heap sort with the assumption that the smallest element of the list floats to the root during the construction of heap.
6. Implement shell sort for a given sequence of increments. Display the output list at the end of each pass.



# EXTERNAL SORTING

# 17

## Introduction

## 17.1

Internal sorting deals with the ordering of records (or keys) of a file (or list) in the ascending or descending order when the whole file or list is compact enough to be accommodated in the internal memory of the computer. Chapter 16 detailed internal sorting techniques such as Bubble Sort, Insertion Sort, Selection sort, Merge Sort, Shell sort, Quick Sort, Heap Sort and Radix Sort.

However, in many applications and problems it is quite common to encounter huge files comprising millions of records which need to be sorted for their effective use in the application concerned. The application domains of e-governance, digital library, search engines, on-line telephone directory and electoral system, to list a few, deal with voluminous files of records.

Majority of the internal sorting techniques that we learned are virtually incapable of sorting large files since they require the whole file in the internal memory of the computer, which is impossible. Hence the need for external sorting methods which are exclusive strategies to sort huge files.

## The principle behind external sorting

Due to their large volume, the files are stored in external storage devices such as tapes, disks or drums. The external sorting strategies therefore need to take into consideration the kind of medium on which the files reside, since these influence their work strategy.

The files residing on these external storage devices are read ‘piece meal’ since only that many records that can be accommodated in the internal memory of the computer, can be read at a time. These batches of records are sorted making use of any efficient internal sorting method. Each of the sorted batches of records are referred to as *runs*. The file is now viewed as a collection of runs. The runs, as and when they are generated, are written out onto the external storage devices. The variety in the external sorting methods for a particular storage device, is brought about only by the ways in which these runs are gathered and processed, before the final sorted file is obtained. However, majority of the popular external sorting methods make use of *merge sort* for gathering and processing the runs.

- 17.1 *Introduction*
- 17.2 *External Storage devices*
- 17.3 *Sorting with tapes: Balanced merge*
- 17.4 *Sorting with disks: Balanced merge*
- 17.5 *Polyphase merge Sort*
- 17.6 *Cascade merge Sort*

A common principle behind most popular external sorting methods is outlined below:

- (i) Internally sort batches of records from the source file to generate runs. Write out the runs as and when they are generated, onto the external storage device(s).
- (ii) Merge the runs generated in the earlier phase, to obtain larger but fewer runs, and write them out onto the external storage devices.
- (iii) Repeat the run generation and merge, until in the final phase only one run gets generated, on which the sorting of the file is done.

Since external storage devices play an imminent role in external sorting, we discuss sorting methods as applicable to two popular storage devices, viz., *magnetic tapes* and *magnetic disks*, the latter commonly referred to as *hard disks*. The reason for the choice is that these devices are representative of two different genres and display different characteristics. While magnetic tapes are undoubtedly obsolete these days, it is worthwhile to go through the external sorting methods applicable on these devices, considering the amount of research efforts and innovation that had gone into them, during their 'hey days'!

The following section briefly discusses the external storage devices of magnetic tapes and disks. The external sorting method of balanced merge applicable to files stored on both tapes and disks is elaborately discussed. A crisp description of polyphase merge and cascade merge sort procedures is presented finally.

## External Storage Devices

## 17.2

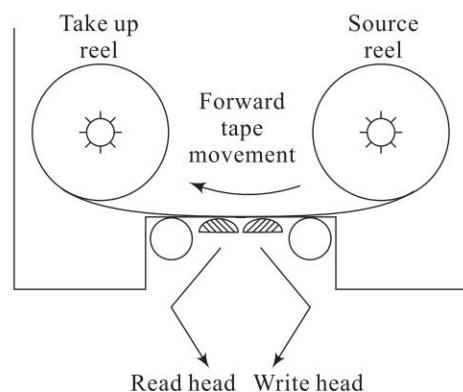
In this section we briefly explain the characteristics of magnetic tapes and magnetic disks.

### Magnetic tapes

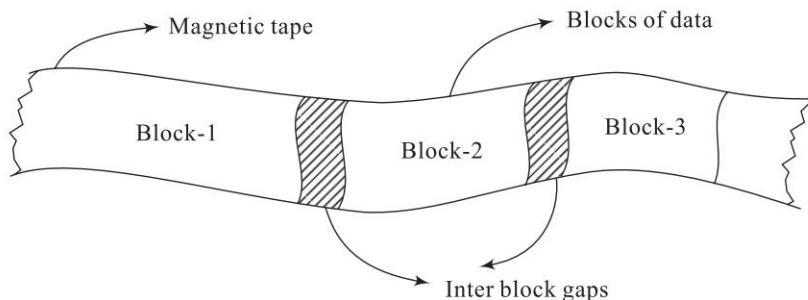
*Magnetic tape* is a *sequential device* whose principle is similar to that of an audio tape /cassette device. It consists of a reel of magnetic tape, approximately  $\frac{1}{2}$ " wide and wound round a *spool*. Data is stored on the tape using the principle of magnetization. Each tape has about 7 or 9 *tracks* running lengthwise. A spot on the tape represents a 0 or 1 bit depending on the direction of magnetization. A combination of bits on the tracks, at any point along the length of the tape, represents a character. The number of bits per inch that can be written on the tape is known as *tape density* and is expressed as *bpi* (bits per inch). Magnetic tapes with densities of 800 bpi and 1600 bpi were in common use during the earlier days.

The magnetic tape device consists of two *spindles*. While one spindle holds the *source reel*, the other holds the *take up reel*. During a forward read/write operation, the tape moves from the source reel to the take up reel. Fig. 17.1 illustrates a schematic diagram of the magnetic tape drive.

The data to be stored on a tape is written on to it in *blocks*. These blocks may be of fixed or variable size. A gap of  $\frac{3}{4}$ " is left between the blocks and is referred to as *Inter Block Gap* (IBG). The IBG is long enough to permit the tape accelerate from rest to reach its normal speed before it begins to read the next block. Figure 17.2 shows the IBG of a tape.



**Fig. 17.1** Schematic diagram of a magnetic tape drive



**Fig. 17.2 Inter Block Gap of a tape**

Magnetic tape is a sequential device since having read a block of data, if one desires to read another block that is several feet down the tape, then it is essential to *fast forward* the tape until the correct block is reached. Again if we desire to read blocks of data that occur towards the beginning of the tape, then it is essential that the tape is *rewound* and the reading starts from the beginning onwards. In these aspects the characteristic of tapes is similar to that of audio cassettes.

## Magnetic disks

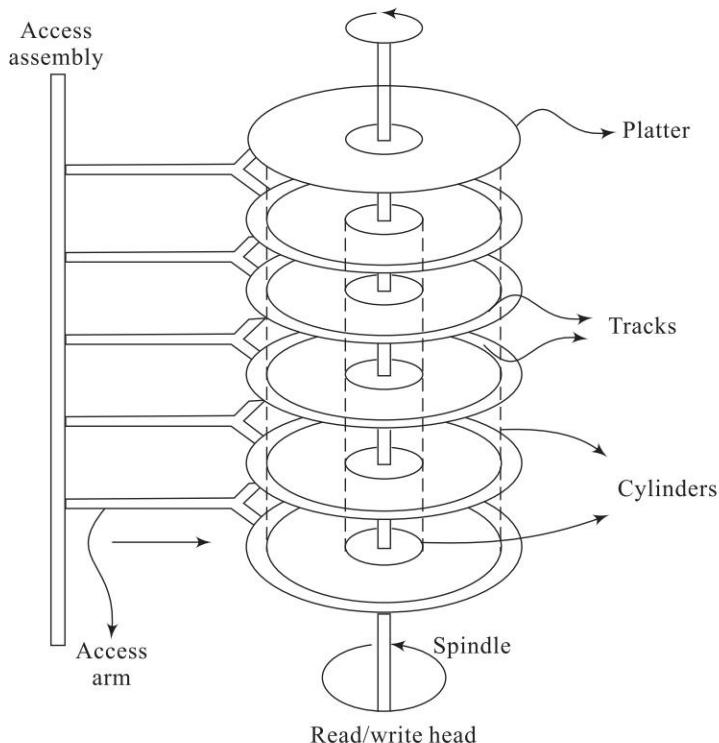
*Magnetic disks* are still in vogue and are commonly referred to as *hard disks*, these days. Hard disks are random access storage devices. This means that hard disks store data in such a manner that they permit both sequential access as well as random or direct access of data.

A *disk pack* is mountable on a *disk drive* and comprises of *platters* which are similar to phonograph records. The number of platters in a disk pack varies according to its capacity. Figure 17.3 shows a schematic diagram of a disk pack comprising 6 platters.

Recording of data is done on all *surfaces* of the platters except the outer surfaces of the first and last platter. Thus for a 6-platter disk pack, of the 12 surfaces available, data recording is done only on 10 of the surfaces. Each surface is accessed by a *read/write head*. The *access assembly* comprises of an assembly of *access arms* ending in the read/write head. The access assembly moves in and out together with the access arms, so that all the read/write heads at any point of time are stationed at the same position on the surface. During a read/write operation, the read/write head is held stationary over the appropriate position on the surface, while the disk rotates at high speed to enable the read/write operation. Disk speeds ranging from 3000 rpm to 4900 rpm are common these days.

Each surface of the platter, like a phonograph record, is made up of concentric circles of *tracks* of decreasing radii, on which the data is recorded. Modern versions of the hard disk contain tens of thousands of tracks per surface. The tracks are numbered from 0 beginning from the outer edge of the platter. The collection of tracks of the same radii, occurring on all the surfaces of the disk pack, is referred to as a *cylinder* (Refer Fig. 17.3). Thus a disk pack is virtually viewed as a collection of cylinders of decreasing radii. Each track is divided into *sectors* which is the smallest addressable segment of a track. Typically a sector can hold 512 bytes of data approximately. The early disk packs had all tracks holding the same number of sectors. The modern versions have however rid themselves off this feature to increase the storage capacity of the disk.

To access information on a disk, it is essential to first specify the cylinder number, followed by the track number and the sector number. A multilevel index based ISAM file organization



**Fig. 17.3 Schematic diagram of a disk pack**

(see Sec. 15.7) is adopted for obtaining the physical locations of records stored in the disk. The *cylinder index* records the highest key in each cylinder and the cylinder number. The *surface index* or the *track index* stores the highest key in each track and the track number. Finally the *sector index* records the highest key in each sector and the sector number. In practice, each of the index entries also contain other spatial information to help locate the records efficiently. Thus the cylinder, track and sector indexes form a hierarchy of indexes which help identify the physical location of the record.

The read/write head moves across the cylinders to position itself on the right cylinder. The time taken to position the read/write head on the correct cylinder is known as *seek time*. Once the read/write head has positioned itself on the correct track of the cylinder, it has to wait for the right sector in the track to appear under the corresponding read/write head. The time taken for the right sector to appear under the read/write head is known as *latency time* or *rotational delay*. Once the sector is reached, the corresponding data are read or written on to the disk. The time taken for the transfer of data to and from the disk is known as *data transmission time*.

## Sorting with Tapes: Balanced Merge

## 17.3

Balanced merge sort makes use of an internal sorting technique to generate the runs and employs merging to gather the runs for the next phase of the sorting. The repeated run generation and merging continue until a single run generated in the final phase delivers the sorted file. In this

section we discuss balanced merge when the file resides on a tape. Besides the input tape, the sorting method has to make use of a few more work tapes to hold the runs that are generated from time to time and to perform the merging of the runs as well. Example 17.1 illustrates balanced merge sort on tapes. The sorting method makes use of 2-way merge to gather the runs.

**Example 17.1** Let us suppose we had to sort a file of 50,000 records ( $R_1, R_2, R_3, \dots, R_{50000}$ ) which is available on a tape (Tape  $T_0$ ) using balanced 2-way merge sort. Assume that the internal memory can hold only 10,000 records. Also let us suppose that there are 4 work tapes ( $T_1, T_2, T_3, T_4$ ) available to assist in the sorting process.  $R_{ij}$  indicates the  $j$ th run in the  $i$ th phase of the sorting. ↑ indicates the read/write head position on the tape. The steps in the sorting process are listed below:

**Step 1:** Rewind all tapes and mount tapes  $T_0, T_1$ , and  $T_2$  onto the tape drive.

**Step 2:** *Phase 1:* Read blocks of 10,000 records each from tape  $T_0$  and internally sort them to generate runs. Let  $R_{11} (R_1 \dots R_{10000})$ ,  $R_{12} (R_{10001} \dots R_{20000})$ ,  $R_{13} (R_{20001} \dots R_{30000})$ ,  $R_{14} (R_{30001} \dots R_{40000})$  and  $R_{15} (R_{40001} \dots R_{50000})$  be the five runs that are to be generated. Distribute the runs alternately onto tapes  $T_1$  and  $T_2$ . The distribution of runs on the tapes  $T_1$  and  $T_2$  are as shown below:

|            |                             |                             |                             |
|------------|-----------------------------|-----------------------------|-----------------------------|
| Tape $T_1$ | $R_1 \dots R_{10000}$       | $R_{20001} \dots R_{30000}$ | $R_{40001} \dots R_{50000}$ |
| Tape $T_2$ | $R_{10001} \dots R_{20000}$ | $R_{30001} \dots R_{40000}$ |                             |

**Step 3:** Dismount tape  $T_0$  and rewind tapes  $T_1$  and  $T_2$ . Mount tapes  $T_1, T_2, T_3, T_4$  onto the drives. Here  $T_1, T_2$  are the input tapes and  $T_3, T_4$  are the output tapes.

**Step 4:** *Phase 2:* Merge runs on tapes  $T_1$  and  $T_2$  using a 2-way merge to obtain longer runs  $R_{21} (R_1 \dots R_{20000})$ ,  $R_{22} (R_{20001} \dots R_{40000})$  and  $R_{23} (R_{40001} \dots R_{50000})$ . The distribution of runs on the output tapes  $T_3, T_4$  are shown below. Note that run  $R_{23}$  is simply copied onto tape  $T_3$  and is just a dummy run.

|            |                             |                             |  |
|------------|-----------------------------|-----------------------------|--|
| Tape $T_3$ | $R_1 \dots R_{20000}$       | $R_{40001} \dots R_{50000}$ |  |
| Tape $T_4$ | $R_{20001} \dots R_{40000}$ |                             |  |

**Step 5:** Rewind all tapes  $T_1, T_2, T_3, T_4$ . Mount  $T_3, T_4$  as the input tapes and  $T_1, T_2$  as the output tapes.

**Step 6:** *Phase 3:* Merge runs on tapes  $T_3$  and  $T_4$  using a 2-way merge to obtain runs  $R_{31} (R_1 \dots R_{40000})$  and  $R_{32} (R_{40001} \dots R_{50000})$ . The distribution of runs on the output tapes  $T_1, T_2$  are shown below. Note that run  $R_{32}$  is simply copied onto tape  $T_2$  and is just a dummy run.

|            |                             |  |
|------------|-----------------------------|--|
| Tape $T_1$ | $R_1 \dots R_{40000}$       |  |
| Tape $T_2$ | $R_{40001} \dots R_{50000}$ |  |

**Step 7:** Rewind all tapes  $T_1, T_2, T_3, T_4$ . Mount  $T_1, T_2$  as the input tapes and  $T_3$  as the output tape.

**Step 8:** *Phase 4:* Merge runs on tapes  $T_1$  and  $T_2$  using a 2-way merge to obtain the final run  $R_{41}(R_1 \dots R_{50000})$ . The final run is written onto tape  $T_3$ .

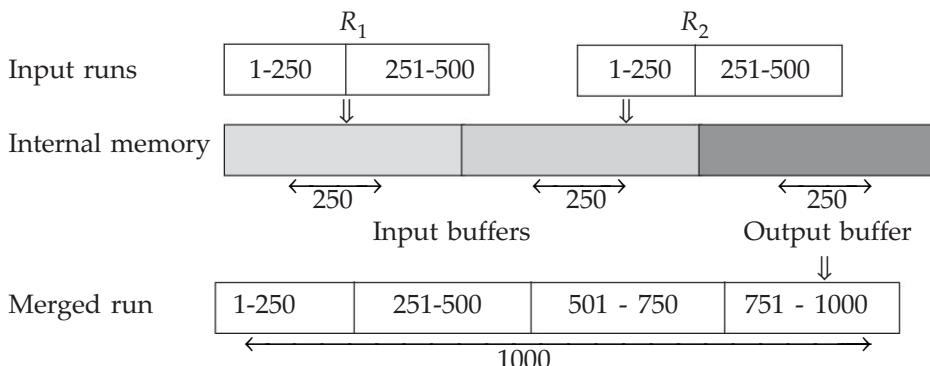


## Buffer handling

While merging runs in the balanced merge sort procedure, it needs to be observed that due to the limited capacity of the internal memory of the computer, it is not always possible to completely accommodate the runs and the merged list in it. In fact the problem gets severe as the phases in the sort procedure progress, since the runs get longer and longer.

To tackle this problem, in the case of 2-way merge let us say, we trifurcate the internal memory into blocks known as *buffers*. Two of these blocks will be used as *input buffers* and the third as the *output buffer*. During the merge of two runs  $R_1$  and  $R_2$ , for example, as many records as can be accommodated in the two input buffers are read from the runs  $R_1$  and  $R_2$  respectively. The merged records are sent to the output buffer. Once the output buffer is full, the records are written on to the disk. If during the merging process, any of the input buffers gets empty, it is once again filled with the rest of the records from the runs.

**Example 17.2** Let us consider the merge of two runs  $R_1$  and  $R_2$  each of which holds 500 records. The output run would contain 1000 records after merging. Let us suppose the internal memory of the computer can hold only 750 records. To undertake merging we divide the internal memory into two input buffers and an output buffer, each of which can hold 250 records each. The merge process is shown below:



The input buffers read in 250 records each, from the two runs  $R_1$  and  $R_2$  respectively. The merging which yields 500 records is emptied by the output buffer into the disk as two blocks of 250 records each. The final merged run which contains 1000 records is in fact a collection of 4 blocks of merged records each containing 250 records.

Example 17.2 presented a simple view of buffer handling. In reality, issues such as proper choice of buffer lengths, efficient utilization of program buffers to enable maximum overlapping of input/output and CPU processing times need to be attended to.

## Balanced P – way merging on tapes

In the case of a balanced 2-way merge, if  $M$  runs were produced in the internal sorting phase and if  $2^{k-1} < M \leq 2^k$  then the sort procedure makes  $k = \lceil \log_2 M \rceil$  merging passes over the data records.

Now balanced merging can easily be generalized to the inclusion of  $T$  tapes,  $T \geq 3$ . We divide the tapes  $T$  into two groups, with  $P$  tapes on the one side and  $(T-P)$  tapes on the other, where  $1 \leq P < T$ . The initial runs generated after internal sorting are evenly distributed on to the  $P$  tapes in the first group. A  $P$ -way merge is undertaken and the resulting runs are evenly distributed on to the next group containing  $(T-P)$  tapes. This is followed by a  $(T-P)$  merge of the runs available on the  $(T-P)$  tapes, with the output runs getting evenly distributed on to the  $P$  tapes of the first group and so on. However, it has been proved that  $P = \left\lceil \frac{T}{2} \right\rceil$  is the best choice. Illustrative

Problem 17.4 discusses an example. Though balanced merging can be quite simple in its implementation, it needs to be seen if better merging patterns which save on time and resource can be evolved for the specific cases in hand. Illustrative Problems 17.5 and 17.6 discuss cases.

## Sorting with Disks: Balanced Merge

17.4

Tapes being sequential access devices, the balanced merge sort methods had to employ sizable resources for the efficient distribution of runs besides spending time for mounting, dismounting and rewinding tapes. In the case of disks which are random access storage devices, we are spared of this burden. The seek time and latency time to access blocks of data from a disk is comparatively negligible, to the time taken to access blocks of data on tapes.

The balanced merge sort procedure for disk files, though similar in principle to that of tape files, is a lot simpler. The runs generated by the internal sorting methods are repeatedly merged until a single run emerges with the entire file sorted in the final pass. Example 17.3 demonstrates balanced merge sort on a disk file.

**Example 17.3** Let us suppose a file comprising 4500 records ( $R_1, R_2, R_3, \dots, R_{4500}$ ) is available on a disk. The internal memory of the computer can accommodate only 750 records. Another disk is available as a scratch pad. The input disk is not to be written on. Making use of buffer handling, we presume that during internal sorting as well as merging, blocks of data comprising 250 records each are read/written.  $R_{ij}$  indicates the  $j^{\text{th}}$  run generated in the  $i^{\text{th}}$  pass. The steps involved in undertaking balanced 2-way merge for sorting the file are shown below:

- Step 1:** Read three blocks of data (totally 750 records) at a time from the file residing on the disk. Internally sort the blocks in the internal memory of the computer to generate 6 runs viz.,  $R_{01}, R_{02}, R_{03}, R_{04}, R_{05}, R_{06}$ . Write the runs onto the scratch disk.
- Step 2:** Trifurcate the internal memory into two input buffers and a single output buffer each capable of holding 250 records.
- Step 3:** Read runs from the disk and merge them pair wise, appropriately making use of buffer handling during the merging process and write the output runs onto the scratch disk.
- Step 4:** Repeat step 3 until a single run emerges, holding the entire sorted file. The merging passes are schematically shown in Fig. 17.4.

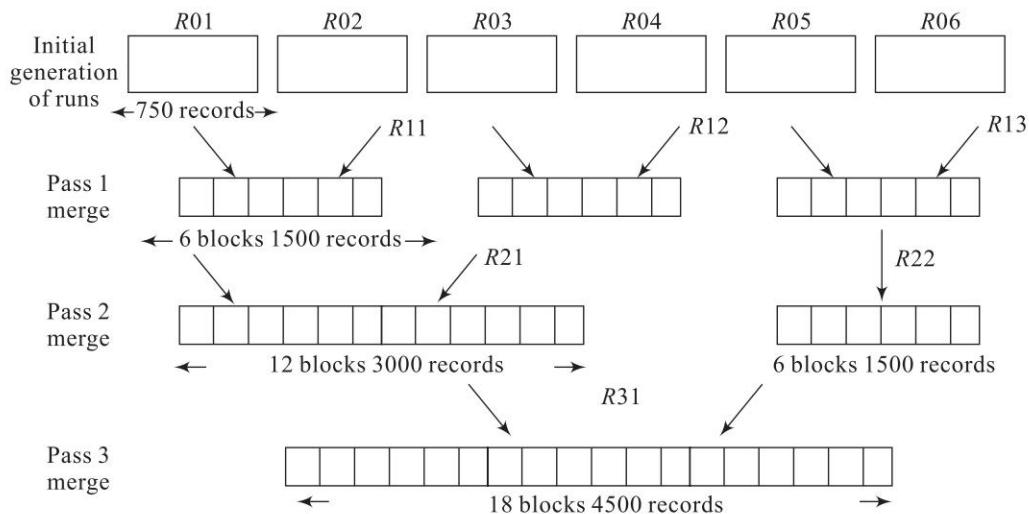


Fig. 17.4 Balanced merge sort: Merging the runs (Example 17.3)

### Balanced $k$ -way merging on disks

As discussed in Sec. 17.3 balanced 2-way merge sort can be generalized to  $k$ -way merging. For a 2-way merge, as can be deduced from Fig. 17.4, the number of passes over data is given by  $\lceil \log_2 M \rceil$  where  $M$  is the number of runs in the first level of the merge tree. A higher order merge can serve to reduce the number of passes over data. Thus in the case of  $k$ -way merge,  $k \geq 2$ , the number of passes is given by  $\lceil \log_k M \rceil$ , where  $M$  is the number of runs. Figure 17.5 shows the merge tree for  $k = 4$ , for an initial generation of 16 runs in a specific case.

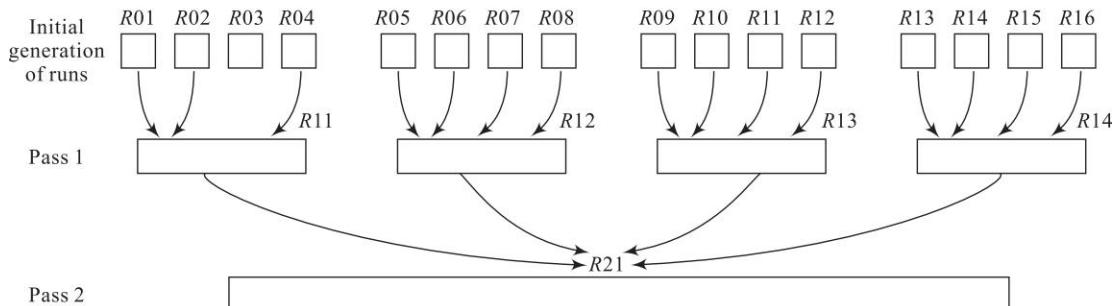


Fig. 17.5 Balanced  $k$ -way merge sort: Merging the runs for  $k = 4$

Though  $k$ -way merge can significantly reduce input / output due to the reduction in the number of passes, it is not without its ill effects. Let us suppose  $R_1, R_2, R_3, \dots, R_k$  are the  $k$  runs generated initially with size  $r_i$ ,  $1 \leq i \leq k$ . During a  $k$ -way merge the next record which is to be output is the one with the smallest key. A direct method to find the smallest key would call for  $(k-1)$  comparisons. The computing time to merge the  $k$  runs would be given by  $O((k-1) \cdot \sum_{i=1}^k r_i)$ .

Since  $\lceil \log_2 M \rceil$  passes are being made, the total number of key comparisons is given by  $n(k-1)\log_2 M$ , where  $n$  is the total number of records in the source file. We have

$n(k-1)\log_2 M = n(k-1) \frac{\log_2 M}{\log_2 k}$ . In other words for a  $k$ -way merge sort, the number of key

comparisons increases by a factor of  $\frac{(k-1)}{\log_2 k}$ . Thus for large  $k$  ( $k \geq 6$ ) the CPU time needed to

perform the  $k$ -way merge will outweigh the reduction achieved in input/output time due to the reduction in the number of passes. A significant reduction in the number of comparisons to find the smallest key can be achieved by using what is known as a *selection tree*.

## Selection tree

A *selection tree* is a complete binary tree which serves to obtain the smallest key from among a set of keys. Each internal node represents the smaller of its two children and external nodes represent the keys from which the selection of the smallest key needs to be made. The root node represents the smallest key that was selected.

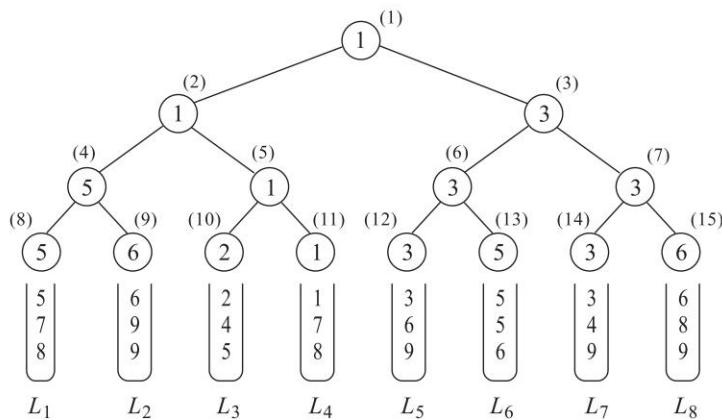
Figure 17.6 (a) represents a selection tree for an 8-way merge. The eight lists to be merged are  $L_1(5, 7, 8)$ ,  $L_2(6, 9, 9)$ ,  $L_3(2, 4, 5)$ ,  $L_4(1, 7, 8)$ ,  $L_5(3, 6, 9)$ ,  $L_6(5, 5, 6)$ ,  $L_7(3, 4, 9)$ ,  $L_8(6, 8, 9)$ . The external nodes represent the first set of 8 keys that were selected from the lists. Progressing from the bottom up, each of the internal node represents the smaller key of its two children until at the root node the smallest key gets automatically represented. The construction of the selection tree can be compared to a tournament being played with each of the internal nodes recording the winners of the individual matches. The final winner is registered by the root node. A selection tree therefore, is also referred to as a *tree of winners*.

In this case, the smallest key viz., 1 is dropped into the output list. Now the next key from  $L_4$  viz., 7 enters the external node. It is now essential to restructure the tree to determine the next winner. Observe how it is now sufficient to restructure only that portion of the tree occurring along the path from the node numbered 11 to the root node. The revised key values of the internal nodes are shown in Fig. 17.6(b). Note how in this case, the keys compared and revised along the path are (2, 7), (5,2), (2,3). The root node now represents 2 which is the next smallest key.

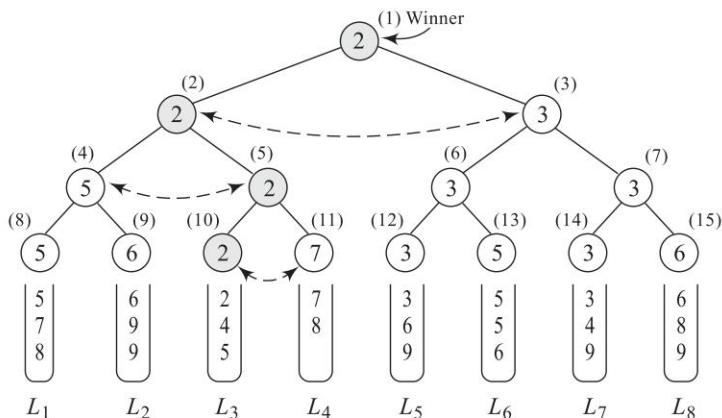
In practice, the external nodes of the selection tree are represented by the records and the internal nodes are only pointers to the records which are winners. (For ease of understanding the internal nodes in Fig. 17.6 were represented using the keys themselves, though in reality they are only pointers to the winning records)

Despite its merits a selection tree can result in increased overheads associated with maintaining the tree. This happens especially when the restructuring of the tree takes place to determine the next winner. It can be seen that the when the next key walks into the tree, tournaments have to be played between sibling nodes who were losers earlier.

Note how in the case of 7 entering the tree, the tournaments played were between (2, 7), (5,2) and (2,3), where 2, 5 and 3 were losers in the earlier case. It would therefore be prudent if the internal nodes could represent the losers rather than the winners. A tournament tree in which each internal node retains a pointer to the loser is called a *tree of losers*.



(a) The smallest key (key 1) is the winner

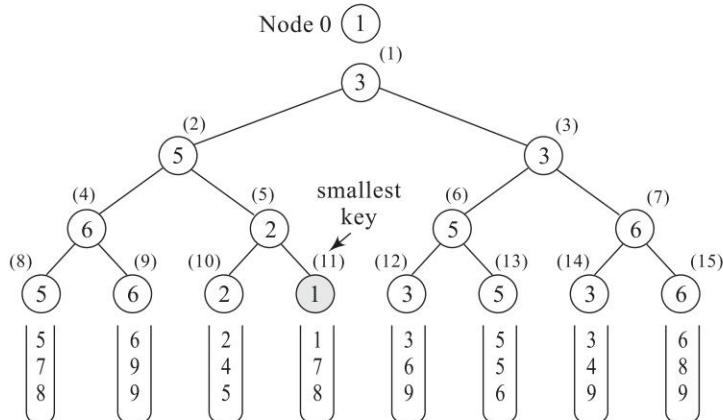


(b) Restructuring the tree to determine the next winner (key 2)

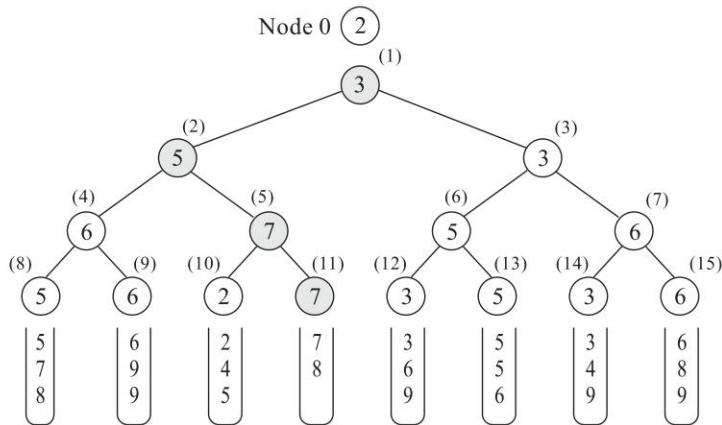
**Fig. 17.6** Selection tree for an 8-way merge

Figure 17.7 (a) illustrates the tree of losers for the selection tree discussed in Fig. 17.6. Node 0 is a special node which shows the winner. As said earlier, each of the internal nodes is shown carrying the key when in reality they represent only pointers to the loser records. To determine the smallest key, as before, a tournament is played between pairs of external nodes. Though the winners are ‘remembered’, it is the losers that the internal nodes are made to point to. Thus nodes numbered (4), (5), (6), (7) record pointers to the losing external nodes viz., the ones with the key values of 6, 2, 5, 6 respectively. Now node numbered (2) conducts a tournament between the two winners of the earlier game viz., key values 5 and 1 and records the pointer to the loser which is 5. In a similar way, node numbered (3) records the pointer to the loser node with key value 3. Progressing in this way the tree of losers is constructed and node 0 outputs the winning key value which is the smallest.

Once the smallest key viz., 1 has been output and the next key 7 enters the tree, the restructuring is easier now, since the sibling nodes with which the tournaments are to be played are losers and these are directly pointed to by the internal nodes. The restructured tree is shown in Fig. 17.7(b).



(a) The smallest key (key 1) is output



(b) Restructured tree after the smallest key is output

**Fig. 17.7 Tree of losers for the 8-way merge**

## Polyphase Merge Sort

## 17.5

Balanced  $k$ -way merge sort on tapes calls for an even distribution of runs on the tapes and to enable efficient merging requires  $2k$  tapes to avoid wasteful passes over data. Thus while  $k$  tapes act as input devices holding the runs generated, the other  $k$  tapes act as output devices to receive the merged runs. The  $k$  tape groups swap roles in the successive passes until a single run emerges in one of the tapes, signaling the end of sort.

It is possible to avoid wasteful redistribution of runs on the tapes while using less than  $2k$  tapes by a wisely thought out run redistribution strategy. *Polyphase merge* is one such external sorting method that makes use of an intelligent redistribution of runs during merging, so much that a  $k$ -way merge requires only  $(k+1)$  tapes!

The central principle of the method is to ensure that in each pass (except the last of course!) during the merge, the runs are to be cleverly distributed so that one tape is always rendered empty while the other  $k$  tapes hold the input runs that are to be merged! The empty tape for the current pass acts as the output tape for the next pass and so on. Ultimately, as in balanced merge sort, the final pass delivers only one run in one of the tapes.

At this point of time we introduce a useful notation mentioned in the literature to enable a crisp presentation of run distribution. Runs that are initially generated by internal sorting are thought to be of length 1 (unit of measure). Thus if there are  $t$  runs that are initially generated then the notation would describe it as  $1^t$ . For example, if there were 34 runs that were initially generated then it would be represented as  $1^{34}$ . Similarly, if after a merge there were 14 runs of size 2, it would be represented as  $2^{14}$ . In general,  $t$  runs of size  $s$  would be represented as  $s^t$ .

Example 17.4 illustrates polyphase merge on 3 tapes.

**Example 17.4** Let us suppose a source file was initially sorted to generate 34 runs of size 1 ( $1^{34}$ ). We demonstrate polyphase merge on 3 tapes ( $T_1$ ,  $T_2$ ,  $T_3$ ) undertaking a 2-way merge during each phase. Table 17.1 shows the redistribution of runs on the tapes in each phase.

**Table 17.1 Polyphase merge on 3 tapes: redistribution of runs**

| Phase | Tape $T_1$ | Tape $T_2$ | Tape $T_3$ | Remarks                      |
|-------|------------|------------|------------|------------------------------|
| 1     | $1^{13}$   | $1^{21}$   | -          | Initial distribution of runs |
| 2     | -          | $1^8$      | $2^{13}$   | Merge to $T_3$               |
| 3     | $3^8$      | -          | $2^5$      | Merge to $T_1$               |
| 4     | $3^3$      | $5^5$      | -          | Merge to $T_2$               |
| 5     | -          | $5^2$      | $8^3$      | Merge to $T_3$               |
| 6     | $13^2$     | -          | $8^1$      | Merge to $T_1$               |
| 7     | $13^1$     | $21^1$     | -          | Merge to $T_2$               |
| 8     | -          | -          | $34^1$     | Merge to $T_3$               |

Note how in phase 8, polyphase merge successfully completes its sorting by creating the final run of sorted records. Also observe how in each phase one of the tapes is rendered empty while the other two are non empty. Now what is the trick behind this procedure?

Let us suppose that 'intuitively' we decided to distribute 13 runs of size 1 and 21 runs of size 1 onto tapes  $T_1$  and  $T_2$  respectively. In phase 2, because it is a 2-way merge and polyphase merge expects one tape to fall vacant in every phase, we use up all the 13 runs of size 1 in tape  $T_1$  for a merge operation with an equivalent number of runs in tape  $T_2$ . This yields 13 runs of double the size ( $2^{13}$ ) which is written on to the empty tape  $T_3$ . That leaves 8 runs of size 1 on tape  $T_2$  that could not be used up and renders tape  $T_1$  empty. Again in phase 3,  $1^8$  runs in tape  $T_2$  are merged with an equivalent amount of runs in tape  $T_3$  to obtain  $3^8$  which is written on to tape  $T_1$ . This leaves a balance of  $2^5$  runs on tape  $T_3$  and renders tape  $T_2$  empty. The phases continue until in phase 8 a single run  $34^1$  gets written on to tape  $T_3$ .

To determine how the initial distribution of  $1^{13}$  and  $1^{21}$  was conceived, we work backwards from the last phase. Let us suppose there were  $n$  phases for a 3-tape case. In the  $n^{\text{th}}$  phase, we

should arrive at exactly one run on a tape  $T_1$  (let us say) with tapes  $T_2$  and  $T_3$  totally empty. This implies that in phase  $(n-1)$  there should have been two runs of size 1 on tapes  $T_2$  and  $T_3$  which should have been merged as a single run on  $T_3$  in the  $n^{\text{th}}$  phase. Continuing in this fashion we obtain the initial distribution of runs to be  $1^{13}$  and  $1^{21}$  on the two tapes respectively. Table 17.2 lists the run distribution for a 3-tape polyphase merge.

**Table 17.2 Run distribution for a 3 - tape polyphase merge**

| Phase | Tape $T_1$ | Tape $T_2$ | Tape $T_3$ |
|-------|------------|------------|------------|
| $n$   | 0          | 0          | 1          |
| $n-1$ | 1          | 1          | 0          |
| $n-2$ | 2          | 0          | 1          |
| $n-3$ | 0          | 2          | 3          |
| $n-4$ | 3          | 5          | 0          |
| $n-5$ | 8          | 0          | 5          |
| $n-6$ | 0          | 8          | 13         |
| $n-7$ | 13         | 21         | 0          |
| $n-8$ | 34         | 0          | 21         |

It can be easily seen that the number of runs needed for an  $n$ -phase merge is given by  $F_n + F_{n-1}$  where  $F_i$  is the  $i^{\text{th}}$  Fibonacci number. Hence this method of redistribution of runs is known as *Fibonacci merge*. The method can be clearly generalized to  $k$ -way merging on  $(k+1)$  tapes using generalized Fibonacci numbers.

## Cascade Merge Sort

## 17.6

Cascade merge is another intelligent merge pattern that was discovered before polyphase merge. The merge pattern makes use of a perfectly devised initial distribution of runs on the tapes. While the polyphase merge sort employs a uniform merge pattern during the run generation, cascade merge sort makes use of a ‘cascading’ merge pattern in each of its passes. Thus for  $t$  tapes, while polyphase merge uniformly employs a  $(t-1)$  merge for the run generation, cascade sort employs  $(t-1)$  merge,  $(t-2)$  merge and so on in the same pass for its run generation.

Example 17.5 demonstrates cascade merge on 6 tapes for an initial generation of 55 runs of length 1. We make use of the run distribution notation introduced in Sec. 17.5.

**Example 17.5** There are 6 tapes ( $T_1, T_2, T_3, T_4, T_5, T_6$ ) using which 55 runs of length 1 ( $1^{55}$ ) are to be cascade merge sorted, to generate the final run ( $55^1$  ).

Table 17.3 illustrates the run distribution of cascade merge.

**Table 17.3 Run distribution on 6 tapes by cascade merge**

| Pass                 | $T_1$    | $T_2$    | $T_3$    | $T_4$ | $T_5$ | $T_6$  |
|----------------------|----------|----------|----------|-------|-------|--------|
| Initial distribution | $1^{15}$ | $1^{14}$ | $1^{12}$ | $1^9$ | $1^5$ | -      |
| 1                    | -        | $1^1$    | $2^2$    | $3^3$ | $4^4$ | $5^5$  |
| 2                    | $15^1$   | $14^1$   | $12^1$   | $9^1$ | $5^1$ | -      |
| 3                    | -        | -        | -        | -     | -     | $55^1$ |

As before, let us assume that the initial distribution of  $(1^{15} 1^{14} 1^{12} 1^9 1^5)$  runs on the tapes ( $T_1, T_2, T_3, T_4, T_5$ ), was devised through some ‘intuitive’ means.

In pass 1, we undertake a series of merges. A 5-way merge on  $(T_1, T_2, T_3, T_4, T_5)$  yields the run  $5^5$  that is put onto tape  $T_6$ . A 4-way merge on  $(T_1, T_2, T_3, T_4)$  yields  $4^4$  which is put on to tape  $T_5$ . A 3-way merge on  $(T_1, T_2, T_3)$  yields  $3^3$  which is distributed onto tape  $T_4$ . A 2-way merge on  $(T_1, T_2)$  yields  $2^2$  which is put onto tape  $T_3$ . Lastly, a 1-way merge (which is mere copying of the balance run) on  $T_1$  yields  $1^1$  which is copied on to tape  $T_2$ . Of course, one could do away with the 1-way merge which is a mere copying of the run and retain the run in the tape itself. In pass 1, Tape  $T_1$  falls empty.

In pass 2, we repeat the cascading merge wherein the 5-way merge on  $(T_2, T_3, T_4, T_5, T_6)$  yields the run  $15^1$ , a 4-way merge on  $(T_3, T_4, T_5, T_6)$  yields  $14^1$  and so on until at the end of pass 2, the distribution of runs on the tapes is as shown in the table. This is the penultimate pass and observe how the distribution records one run each on the tapes. In the final pass, as it always is, the 5-way merge releases a single run of size 55 which is the final sorted file.

Now how does one arrive at the perfect initial distribution? As was done for polyphase merge, this could be arrived at by working backwards from the goal state of  $(1, 0, 0, 0, 0)$  obtained during the  $n^{th}$  pass. Table 17.4 illustrates the run distribution by cascade merge on 5 tapes.

**Table 17.4 Run distribution on 5 tapes by cascade merge**

| Phase | $T_1$ | $T_2$ | $T_3$ | $T_4$ | $T_5$ |
|-------|-------|-------|-------|-------|-------|
| $n$   | 1     | 0     | 0     | 0     | 0     |
| $n-1$ | 1     | 1     | 1     | 1     | 1     |
| $n-2$ | 5     | 4     | 3     | 2     | 1     |
| $n-3$ | 15    | 14    | 12    | 9     | 5     |
| $n-4$ | 55    | 50    | 41    | 29    | 15    |
| $n-5$ | 190   | 175   | 146   | 105   | 55    |
| $n-6$ | 671   | 616   | 511   | 365   | 190   |

For an in depth analysis of merge patterns and other external sorting schemes, a motivated reader is referred to Donald Knuth, Art of Computer Programming, Vol. III, Second edition, 2002.



## Summary

- External sorting deals with sorting of files or lists that are too huge 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)          |

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

**Solution:**

- 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

**Table I 17.1 Specification for a typical disk storage system**

|                                       |                    |
|---------------------------------------|--------------------|
| Number of platters                    | 6                  |
| Number of cylinders                   | 800                |
| Number of tracks (surfaces) /cylinder | 10                 |
| Number of sectors/track               | 50                 |
| Number of bytes/sector                | 512                |
| Maximum seek time                     | 50 milliseconds    |
| Average seek time                     | 25 milliseconds    |
| Maximum Latency time                  | 16.66 milliseconds |
| Average latency time                  | 8.33 milliseconds  |
| Time to read/write a sector           | 0.33 milliseconds  |

$$\therefore \text{The total number of bytes needed to store the file in the disk} = 25,000 \times 512 \\ = 12800000 \text{ bytes} \\ = 12.2 \text{ megabytes}$$

- (b) Number of tracks needed to hold the whole employee file given that there are 50 sectors/track =  $25000 / 50 = 500$  tracks  
 $\therefore$  Number of cylinders needed to store the whole file given that there are 10 tracks/cylinder =  $500/10 = 50$  cylinders

**Problem 17.2** For the employee file discussed in Illustrative Problem 17.1, making use of Table 17.1, answer the questions given below:

Records from the employee file are to be read and making use of the basic pay, allowances and deductions, the total salary is to be computed for each employee. Assume that it takes 200 microseconds of CPU time to perform the computation for a single record. The updated records are to be written onto the disk.

- (a) What is the time taken to process a sector of records?
- (b) Having processed a sector of records, what is the time taken to process all records in the very next sector?
- (c) What is the time taken to process the records, in all sectors of a track, assuming that the sectors are continuously read?
- (d) What is the time taken to process all records in a cylinder?

**Solution:**

- (a) The time taken to process a sector full of records =  
 (1) Time taken to access the cylinder + (2) Time taken to access the sector + (3) time taken to read the records + (4) time taken to compute the net salary for the records + (5) time taken to access the sector to write back the records + (6) time taken to write the updated records onto the sector.

For (1) and (2) since the question pertains to an arbitrary sector, we choose to use the average seek time of 25 milliseconds and the average latency time of 8.33 milliseconds, respectively. For (3) and (6) the time taken is 0.33 milliseconds each. For (4) it is 0.8 milliseconds ( $200$  microseconds  $\times$  4 records).

The computation of (5) which is in fact the time taken for the sector to appear under the read/write head to perform the write operation, is a trifle involved. It is computed as, (the

maximum latency time (time taken for the track to make a full revolution) – time taken to read the sector – time taken to process the records by the CPU ).

This is given by  $(16.66 - 0.33 - 0.8) = 15.53$  milliseconds.

$\therefore$  the time taken to process all records in a sector =

$$25 + 8.33 + 0.33 + 0.8 + 15.53 + 0.33 = 50.32 \text{ milliseconds}$$

- (b) While the time taken to process records in the first sector (Question 17.2(a)) includes the time taken to access the cylinder and the sector, to process the very next sector, there is no need to include the cylinder and sector access time since the reading is continuously done.

$\therefore$  the time taken to process the records in the very next sector =

(3) time taken to read the records + (4) time taken to compute the net salary for the records + (5) time taken to access the sector to write back the records + (6) time taken to write the updated records onto the sector.

$$= 0.33 + 0.8 + 15.53 + 0.33 = 16.99 \text{ milliseconds.}$$

- (c) The time taken to process all records on a track =

(7) time taken to process records in the first sector of the track +

(8) time taken to process records in the next sector of the track  $\times 49$  sectors

Here (7) and (8) have been obtained in Questions 17.2(a) and (b) respectively and therefore the result is given as,

$$50.32 + 16.99 \times 49 = 882.83 \text{ milliseconds.}$$

- (d) The time taken to continuously process all records in a cylinder, calls for processing all records track after track. Once the records in the first occurring track have been processed, the rest of the tracks in the cylinder are instantaneously accessed.

$\therefore$  the time taken to process all records in a cylinder =

(9) time taken to process all records in the first track + (10) time taken to process all records in the next track of the cylinder  $\times 9$  tracks

While (9) is found in Question 17.2(c), to compute (10) we simply need to use the time computed in (8) for all the 50 sectors in the next track.

Therefore the result is given as  $882.83 + 16.99 \times 50 \times 9 = 8.528$  seconds.

**Problem 17.3** Illustrative Problem 17.2(d) computed the time taken to process all records of the employee file residing in a cylinder. Assume that the time taken for the read/write head to move from one cylinder to another is 10 milliseconds.

- (a) What is the time taken to process all records in the next cylinder?

- (b) What is the time taken to process the entire employee file of records in the disk?

**Solution:**

- (a) Having processed a cylinder of records, the time taken to move to the next cylinder is 10 milliseconds. The time taken to process all records in the next cylinder is a straightforward computation given by,

(8) Time taken to process all records on the next sector  $\times 50$  sectors  $\times 10$  tracks

Here (8) is obtained in Question 17.2(b).

$\therefore$  the total time taken to process all records in the next cylinder, moving from the current cylinder =  $10 + (16.99 \times 50 \times 10) = 8.505$  seconds

- (b) The entire employee file resides on 50 cylinders (Illustrative Problem 17.1(b)).

Therefore the time taken to process the entire file =

(11) Time taken to process records in the first cylinder + (12) time taken to process records in the next cylinder  $\times 49$

(11) is obtained in Illustrative Problem 17.2(d) and (12) is obtained in Illustrative Problem 17.3(a).

$\therefore$  the time taken to process the entire employee file =  $8.528 + 8.505 \times 49 = 7.088$  minutes

**Problem 17.4** Given a file of 50,000 records with an internal memory capacity of 10,000 records, trace the steps of a Balanced P-way merge sort for  $T = 6$  tapes ( $T_1, T_2, T_3, T_4, T_5, T_6$ ) and  $P = 3$ .

**Solution:** An internal sort of the file yields 5 runs of 10,000 records each. Since  $P = 3$ , we need to undertake a 3-way merge. We therefore divide the 6 tapes into two groups of 3 tapes each. The two groups alternate as the input and output tapes during the merge passes.

The initial distribution of runs on the tapes  $T_1, T_2$  and  $T_3$  after internal sorting, are as follows:

|              |                             |                             |
|--------------|-----------------------------|-----------------------------|
| Tape $T_1$ : | $R_1 \dots R_{10000}$       | $R_{30001} \dots R_{40000}$ |
| Tape $T_2$ : | $R_{10001} \dots R_{20000}$ | $R_{40001} \dots R_{50000}$ |
| Tape $T_3$ : | $R_{20001} \dots R_{30000}$ |                             |

Rewind the tapes  $T_1, T_2$  and  $T_3$ . In the next pass, the 3-way merge of runs in tapes  $T_1, T_2$  and  $T_3$  yield output runs on  $T_4, T_5, T_6$  as follows:

|              |                             |
|--------------|-----------------------------|
| Tape $T_4$ : | $R_1 \dots R_{30000}$       |
| Tape $T_5$ : | $R_{30001} \dots R_{50000}$ |
| Tape $T_6$ : | Empty                       |

Rewind tapes  $T_4$  and  $T_5$ . In the last pass a 3-way merge of runs in tapes  $T_4$  and  $T_5$  yield the final run on tape  $T_1$  as follows:

Tape  $T_1$ :  $R_1 \dots R_{50000}$

**Problem 17.5** For a file comprising 50,000 records with an internal memory capacity of 10,000 records, the initial distribution of runs on two tapes  $T_1$  and  $T_2$  are as shown below:

|              |                             |                             |                             |
|--------------|-----------------------------|-----------------------------|-----------------------------|
| Tape $T_1$ : | $R_1 \dots R_{10000}$       | $R_{20001} \dots R_{30000}$ | $R_{40001} \dots R_{50000}$ |
| Tape $T_2$ : | $R_{10001} \dots R_{20000}$ | $R_{30001} \dots R_{40000}$ |                             |

Two standby tapes viz.,  $T_3$  and  $T_4$  are available. The following two merge patterns were undertaken. Which of these is efficient and why?

| Merge Pattern A                                                                                                                                                          | Merge pattern B                                                                                                                                                                                            |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Pass 1 (2-way merge):<br>Tape $T_3$ : $R_1 \dots R_{20000} R_{40001} \dots R_{50000}$<br>Tape $T_4$ : $R_{20001} \dots R_{40000}$<br>Rewind tapes $T_1, T_2, T_3, T_4$ . | Pass 1 (2-way merge):<br>Tape $T_3$ : $R_1 \dots R_{20000}$<br>Tape $T_4$ : $R_{20001} \dots R_{40000}$<br>Tape $T_1$ : $R_1 \dots R_{10000} R_{20001} \dots R_{30000} \uparrow R_{40001} \dots R_{50000}$ |

(Contd.)

(Contd.)

|                                                                                                                                                   |                                                                                                                                                                                                                         |
|---------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                                                                                                                                                   | Rewind tapes $T_2$ , $T_3$ , $T_4$ only. Tape $T_1$ retains the run $R_{40001} \dots R_{50000}$ . The ↑ indicates the position of the read / write head from which point onwards $T_1$ would be read for the next pass. |
| Pass 2 (2-way merge):<br>Tape $T_1$ : $R_1 \dots R_{40000}$<br>Tape $T_2$ : $R_{40001} \dots R_{50000}$<br>Rewind tapes $T_1$ , $T_2$ and $T_3$ . | Pass 2 (3-way merge of tapes $T_1$ , $T_3$ and $T_4$ )<br>Tape $T_2$ : $R_1 \dots R_{50000}$                                                                                                                            |
| Pass 3 (2-way merge):<br>Tape $T_3$ : $R_1 \dots R_{50000}$                                                                                       |                                                                                                                                                                                                                         |

**Solution:** Merge pattern B is efficient since the total number of records that were read to obtain the final run on tape  $T_2$  was  $40,000 + 50,000 = 90,000$  records. This took place in 2 passes. On the other hand, Merge pattern A read  $50,000 + 50,000 + 50,000 = 150,000$  records in three passes over the data, to obtain the final run on tape  $T_3$ .

**Problem 17.6** There are 5 runs distributed on three tapes ( $T_1$ ,  $T_2$ ,  $T_3$ ) as shown below. A standby tape ( $T_4$ ) is available. The internal memory capacity is 10,000 records. Undertake a balanced  $P$ -way merge devising a smart merge pattern for some  $P$ .

$$\begin{array}{ll} \text{Tape } T_1: & R_1 \dots R_{10000} \quad R_{30001} \dots R_{40000} \\ \text{Tape } T_2: & R_{10001} \dots R_{20000} \quad R_{40001} \dots R_{50000} \\ \text{Tape } T_3: & R_{20001} \dots R_{30000} \end{array}$$

**Solution:** We first undertake a 3-way merge on the tapes  $T_1$ ,  $T_2$  and  $T_3$  for the first three runs on the tapes.  $T_4$  is used as the output tape. The configuration at the end of pass 1 are as shown below:

$$\begin{array}{ll} \text{Tape } T_4: & R_1 \dots R_{30000} \\ \text{Tape } T_1: & R_1 \dots R_{10000} \quad \uparrow R_{30001} \dots R_{40000} \\ \text{Tape } T_2: & R_{10001} \dots R_{20000} \quad \uparrow R_{40001} \dots R_{50000} \end{array}$$

Tapes  $T_3$  and  $T_4$  are alone rewound.

In the final pass, a 3-way merge is undertaken on tapes  $T_4$ ,  $T_1$ ,  $T_2$ . The output is delivered on tape  $T_3$  as shown below:

$$\text{Tape } T_3: \quad R_1 \dots R_{50000}$$

The merge pattern for the specific case is efficient since only  $30000 + 50000 = 80000$  records were read in the two passes put together for the final sort of the file.

**Problem 17.7** Let us suppose a source file was initially sorted to generate 55 runs of size 1 ( $1^{55}$ ). Trace polyphase merge on 3 tapes ( $T_1$ ,  $T_2$ ,  $T_3$ ) undertaking a 2-way merge during each phase.

**Solution:** Table I 17.7 shows the redistribution of runs on the tapes in each phase. Observe how the initial distribution of runs is taken after the Fibonacci number sequence. The polyphase merged file is available on tape  $T_1$  in the final phase.

**Table I 17.7** Polyphase merge on 3 tapes: redistribution of runs

| Phase | Tape $T_1$ | Tape $T_2$ | Tape $T_3$ | Remarks                      |
|-------|------------|------------|------------|------------------------------|
| 1     | $1^{21}$   | $1^{34}$   | -          | Initial distribution of runs |
| 2     | -          | $1^{13}$   | $2^{21}$   | Merge to $T_3$               |
| 3     | $3^{13}$   | -          | $2^8$      | Merge to $T_1$               |
| 4     | $3^5$      | $5^8$      | -          | Merge to $T_2$               |
| 5     | -          | $5^3$      | $8^5$      | Merge to $T_3$               |
| 6     | $13^3$     | -          | $8^2$      | Merge to $T_1$               |
| 7     | $13^1$     | $21^2$     | -          | Merge to $T_2$               |
| 8     | -          | $21^1$     | $34^1$     | Merge to $T_3$               |
| 9     | $55^1$     | -          | -          | Merge to $T_1$               |

**Problem 17.8** There are 6 tapes ( $T_1, T_2, T_3, T_4, T_5, T_6$ ) using which 190 runs of length 1 ( $1^{190}$ ) are to be cascade merge sorted, to generate the final run ( $190^1$ ). Trace the steps of the sorting process.

**Solution:** Table I 17.8 illustrates the run distribution of cascade merge.

**Table 17.8** Run distribution on 6 tapes by cascade merge

| Pass                 | $T_1$    | $T_2$    | $T_3$    | $T_4$    | $T_5$    | $T_6$    |
|----------------------|----------|----------|----------|----------|----------|----------|
| Initial distribution | $1^{55}$ | $1^{50}$ | $1^{41}$ | $1^{29}$ | $1^{15}$ | -        |
| 1                    | -        | $1^5$    | $2^9$    | $3^{12}$ | $4^{14}$ | $5^{15}$ |
| 2                    | $15^5$   | $14^4$   | $12^3$   | $9^2$    | $5^1$    | -        |
| 3                    | -        | $15^1$   | $29^1$   | $41^1$   | $50^1$   | $55^1$   |
| 4                    | $190^1$  |          |          |          |          |          |

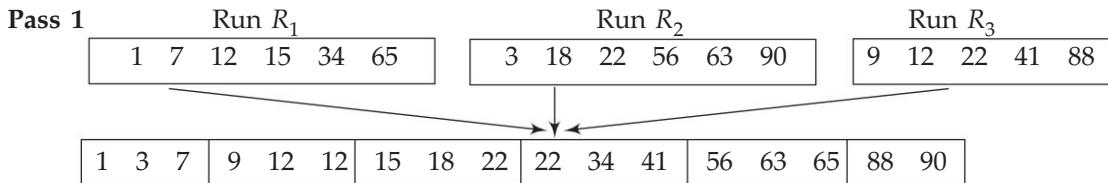
**Problem 17.9** Demonstrate balanced 3-way merge on the following “sample” list of keys available on a disk, with the internal memory capable of holding 6 keys:

12 1 65 7 34 15 90 22 63 56 18 3 9 22 12 88 41

**Solution:** The internal sort of the list yields three runs as follows:

R1: 1 7 12 15 34 65  
R2: 3 18 22 56 63 90  
R3: 9 12 22 41 88

We divide the internal memory into three input buffers and an output buffer to undertake the 3-way merge. Thus while the input buffers can hold one key each, we shall allow the output buffer to hold a maximum of 3 keys. Thus the input data will be read in blocks of one key each. During the merge, the output buffer releases blocks containing 3 keys each which are written onto the run. The merging passes are shown below:



At the end of pass 1 the entire list is sorted.



## Review Questions

- (i) Cascade merge sort adopts uniform merge patterns in its passes  
 (ii) The distribution of runs in the last pass of cascade merge sort is given by a pattern such as (1, 0, 0, ...0)  
 (a) (i) true (ii) true    (b) (i) true (ii) false    (c) (i) false (ii) false    (d) (i) false (ii) true
- Polyphase merge sort for a  $k$ -way merge on tapes requires \_\_\_\_\_ tapes  
 (a)  $2k$                          (b)  $(k-2)$                          (c)  $(k+1)$                          (d)  $k$
- The time taken for the right sector to appear under the read / write head is known as  
 (a) seek time                         (b) latency time                         (c) transmission time (d) data read time
- In the case of a balanced 2-way merge, if  $M$  runs were produced in the internal sorting phase and if  $2^{k-1} < M \leq 2^k$  then the sort procedure makes \_\_\_\_\_ merging passes over the data records.  
 (a)  $M$                                  (b)  $\lceil \log_2 M \rceil$                          (c)  $\lceil \log_M 2 \rceil$                          (d)  $M^2$
- Match the following:  
 W. Magnetic tape      A. tree of winners  
 X. Magnetic disks      B. Fibonacci merge  
 Y. Polyphase merge      C. Inter Block Gap  
 Z.  $k$ -way merge      D. platters  
 (a) (W A) (X B) (Y D) (Z C)                         (b) (W C) (X D) (Y B) (Z A)  
 (c) (W C) (X D) (Y A) (Z B)                             (d) (W A) (X B) (Y C) (Z D)
- What is the general principle behind external sorting?
- How is a selection tree useful in a  $k$ -way merge?
- What are the advantages of Polyphase merge sort over balanced  $k$ -way merge sort?
- What is the principle behind the distribution of runs in a cascade merge sort?
- How is data organized in a magnetic disk?
- An inventory record contains the following fields: ITEM NUMBER (8 bytes), NAME (20 bytes), DESCRIPTION (20 bytes), TOTAL STOCK(10 bytes), PRICE(10 bytes) TOTAL PRICE (14 bytes).  
 A record comprising the data on Item number, name, description and total stock is to be read and based on the current price which is input, the total price is to be computed and updated in the fields. There are 25, 000 records to be processed. Assuming the disk characteristics given in Table I 17.1,  
 (i) How much storage space is required to store the entire file in the disk (in terms of bytes/KB/MB)?

- (ii) How much storage space is required to store the entire file in the disk in terms of cylinders?
  - (iii) What is the time required to read, process and write back a given sector of records into the disk, assuming that it takes 100 microseconds to process a record?
  - (iv) What is the time required to read, process and write back an entire track of records if they were read sequentially sector after sector?
  - (v) What is the time required to read, process and write back an entire cylinder of records?
  - (vi) What is the time required to read, process and write back the records in the next (immediate) cylinder?
  - (vii) What is the time required to read, process and write back the entire file onto the disk?
12. A file comprising 500, 000 records is to be sorted. The internal memory has a capacity to hold only 50, 000 records. Trace the steps of a Balanced  $k$ -way merge for (i)  $k = 2$  and (ii)  $k = 4$ , when (a) the file is available on a tape and (ii) the file is available on a disk. Assume the availability of any number of tapes and a scratch disk for undertaking the appropriate sorting process.



## Programming Assignments

1. Implement a function to construct a tree of winners to obtain the smallest key from a list of keys representing its external nodes.
2. Implement a function to construct a tree of losers to obtain the smallest key from a list of keys representing its external nodes.
3. Making use of the function(s) developed in Programming Assignments 1 and 2 (Chapter 17), implement  $k$ -way merge algorithms for any given value of  $k$ .
4. Implement Balanced  $k$ -way merge sort for disk based files. Simulate the program for various sizes of files, internal memory capacity and choice of  $k$ . Graphically display the distribution of runs.



# 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

## Index

- multigraph 188
- path 190
- subgraph 190
- trees 192
- undirected 188
- Graph 188
  - search 384
- Growth of threaded binary trees 168
- Hard disks 436
- Hash function H 332
  - functions 333
  - table 332
- Hashing 332
- Head node 95
- Heap 356, 415
  - sort 414
- Height balanced trees 228
- Home bucket 335
- Huffman coding 260
- Incidence matrix 195
  - matrix representation 196
- Index 282
- Indexed sequential file organization 358
  - sequential search 385
- Infix, prefix and postfix expressions 45
- Information node 277
- Inorder traversal 158
- Input buffers 440
  - restricted deque 70
- Insertion and deletion in a singly linked list 88
  - into a binary search tree 222
  - into an AVL search tree 230
  - sort 396
- Internal memory 353
  - sorting 394, 435
- Interpolation search 376
- ISAM files 358
- Join operation 344
- k*-way merging 403
- Keys 355
- Keyword table 342
- Koenigsberg bridge problem 186
- L* category rotations 243
- Last In First Out 39
- Lb*0, *Lb*1 and *Lb*2 rotations 322
- Lexicographic search trees 277
- Limitations of linear queues 61
- Linear data structures 6
  - open addressed hash tables 336
  - open addressing 334
  - queues 59
  - search 373
- Linked list 86
- Linked queues 124
  - operations 124, 125
  - representation 6, 168
  - representation of graphs 198
  - stack 124
  - stack operations 125
  - LL* rotation 230
  - LLb*, *LRb*, *RRb* 297
  - LLr*, *LRr*, *RRr* 297
  - Loading factor 338
  - Logarithmic search 378
  - LR* rotation 232
  - Lr*0, *Lr*1 and *Lr*2 rotations 323
- m-way search trees 262
  - definition 263
  - deleting 265
  - drawbacks 268
  - inserting 265
  - node structure 263
  - representation 263
  - searching 264
- Magnetic disks 436, 437
  - tapes 436
- Master file 357
- Merge sort 401, 435
- Merging 401
- Merits of linked data structures 85
- Minimum cost spanning trees 206
- Modular arithmetic 334
- MSD first sort 425
- Multi-dimensional array 28
  - way trees 262
- Multilevel indexing 360
- Multiply linked list 87, 103
- N*-dimensional array 32
- Natural join 344
- Non-linear data structures 6
- Number of elements in an array 27
- One-dimensional array 27, 29

- Operations
  - circular queue 62
  - doubly linked lists 100
  - linked stacks and linked queues 124
  - queues 57
- Optimal binary search tree 246
- Ordered linear search 373, 374
  - lists 33
- Output buffer 440
  - restricted deque 70
- Overflow 335
- Partitioning 410
- Path matrix 195
  - matrix representation 197
- Pile organization 356
- Pivot element 410
- Polynomial representation 133
  - time complexities 12
- Polyphase merge sort 445
- Posteriori testing 8
- Postorder traversal 158, 162
- Preorder traversal 158, 162
- Primary indexing 360
  - keys 355
- Primitive operations on circularly linked lists 95
- Prims algorithm 206
- Priority queues 66
- Quadratic probing 339
- Queue 56
  - dequeuing 56
  - enqueueing 56
  - implementation 57
  - list 147
  - operations 57
- Quick sort 410
- R-1* rotation 242
- 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