

# **Data Structures**

# About the Author



**E Balagurusamy**, former Vice Chancellor, Anna University, Chennai and Member, Union Public Service Commission, New Delhi, is currently the Chairman of EBG Foundation, Coimbatore. He is a teacher, trainer, and consultant in the fields of Information Technology and Management. He holds an ME (Hons) in Electrical Engineering and PhD in Systems Engineering from the Indian Institute of Technology, Roorkee. His areas of interest include *Object-Oriented Software Engineering, E-Governance, Technology Management, Business Process Re-engineering, and Total Quality Management*.

A prolific writer, Dr Balagurusamy has authored a large number of research papers and several books.

A recipient of numerous honors and awards, Dr Balagurusamy has been listed in the Directory of Who's Who of Intellectuals and in the Directory of Distinguished Leaders in Education.

# Data Structures

**E Balagurusamy**

*Chairman  
EBG Foundation  
Coimbatore*



**McGraw Hill Education (India) Private Limited**  
**CHENNAI**

---

*McGraw Hill Education Offices*

**Chennai** 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



## McGraw Hill Education (India) Private Limited

Published by McGraw Hill Education (India) Private Limited  
444/1, Sri Ekambara Naicker Industrial Estate, Alapakkam, Porur, Chennai 600 116

### Data Structures

Copyright © 2019 by McGraw Hill Education (India) Private 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,  
McGraw Hill Education (India) Private Limited.

2 3 4 5 6 7 8 9 D103074 22 21 20 19  18

Printed and bound in India.

#### Print-Book Edition

ISBN (13): 978-93-5316-182-8  
ISBN (10): 93-5316-182-7

#### E-Book Edition

ISBN (13): 978-93-5316-183-5  
ISBN (10): 93-5316-183-5

Director—Science & Engineering Portfolio: *Vibha Mahajan*  
Senior Portfolio Manager—Science & Engineering: *Hemant K Jha*  
Associate Portfolio Manager—Science & Engineering: *Tushar Mishra*

Production Head: *Satinder S Baveja*  
Copy Editor: *Taranpreet Kaur*

General Manager—Production: *Rajender P Ghansela*  
Manager—Production: *Reji Kumar*

Information contained in this work has been obtained by McGraw Hill Education (India), from sources believed to be reliable. However, neither McGraw Hill Education (India) nor its authors guarantee the accuracy or completeness of any information published herein, and neither McGraw Hill Education (India) 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 McGraw Hill Education (India) 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 APS Compugraphics, 4G, PKT 2, Mayur Vihar Phase-III, Delhi 96, and printed at

Cover Designer: APS Compugraphics  
Cover Image Source: Shutterstock

Cover Printer:

Visit us at: [www.mheducation.co.in](http://www.mheducation.co.in)  
Write to us at: [info.india@mheducation.com](mailto:info.india@mheducation.com)  
CIN: U22200TN1970PTC111531  
Toll Free Number: 1800 103 5875

# Preface

## About the Book

Data Structure is the way of storing data in a computer system. It allows an application to fetch and store data in the computer's memory in an efficient manner. It is very important to choose the correct type of data structure while developing a software application. C is one of the first programming languages that students of computer science get familiar with. It is also the language of choice while facilitating the learning of programming concepts such as data structures.

The strength of *Data Structures* lies in its simple and lucid presentation of the subject which will help beginners in better understanding of the concepts. It adopts a student-friendly approach to the subject matter with many solved and unsolved examples, illustrations and well-structured C programs.

This book will prove to be a stepping stone in understanding the data structure concepts in an efficient and organized manner, and also for revisiting the fundamentals of data structure.

## Salient Features of the Book

- In-depth coverage of all important topics like *Arrays, Linked lists, Stacks, Queues, Trees, Graphs, Sorting, and Searching*
- Dedicated chapter on Real Life Applications of Data Structures
- Explains run-time complexity of all algorithms
- Multiple-Choice Questions for university exams and interviews
- Innovative chapter features includes *pedagogical aids like illustrations, programs, important commands in programs, output and program analysis, note, checkpoint, key terms, solved problems, and review questions.*

## What Sets This Book Apart

### Chapter Opening Features

At the opening of each chapter, the outline lists the major headings, followed by an introduction to the chapter. This will help students organize their study priorities.

### In-chapter Features

Features like algorithms, pseudocodes, flowcharts and programs emphasize on a point or help teach a concept. Commands in bold draw students' attention to a particular section in the program.

### Other Significant Features

Notes, Tips and Checkpoints are designed to provide extra information or alternative views or results or interesting snippets of information related to the content of the chapter.

### Chapter-end Features

Summary reviews the concepts while a list of key terms helps identify the vocabulary students need to understand the concepts presented in the chapter. Students can assess their knowledge by answering the basic review questions, programming exercises and multiple-choice questions.

## **Chapter Organization**

This book is organized into 8 chapters, which explain concepts like Arrays, Stacks, Queues, Linked Lists, Trees and Graphs.

**Chapter 1** introduces algorithm and its related concepts. It also provides a brief introduction to the different types of data structures. **Chapter 2** discusses one of the commonly used derived data types, i.e., array and explains how it is used as a data structure in different programming situations. **Chapter 3** explains the concept of linked list along with its different variants. **Chapters 4 and 5** elucidates the restricted data structures, stacks and queues. These data structures are of great importance in programming situations because of the specific restrictions that they apply on insertion and deletion of data elements. **Chapters 6 and 7** explain the non-linear data structures trees and graphs and their related operations. These chapters also explain the various algorithms that are used to traverse these data structures. **Chapter 8** introduces two of the most common computing operations, i.e., searching and sorting. It explains various searching and sorting techniques along with their related advantages and disadvantages.

## **Acknowledgements**

I would like to thank the following reviewers for their suggestions in improving the script:

<b>Dr. K. Sasi Kala Rani</b>	<i>Hindusthan Institute of Technology, Coimbatore</i>
<b>Shashank Dwivedi</b>	<i>UCER, Allahabad, Uttar Pradesh</i>
<b>Rajiv Pandey</b>	<i>Amity University Lucknow Campus, Lucknow, Uttar Pradesh</i>
<b>Mahua Banerjee</b>	<i>Xavier Institute of Social Service, Ranchi, Jharkhand</i>
<b>Sameer Bhave</b>	<i>Indore Professional Studies Academy, Indore, Madhya Pradesh</i>
<b>D Lakshmi, Adithya</b>	<i>Institute of Technology, Coimbatore, Tamil Nadu</i>
<b>A Sharada</b>	<i>G Narayanaamma Institute of Technology and Science, Hyderabad, Andhra Pradesh</i>

Sincere thanks to the editorial team of McGraw Hill Education (India) for their support and cooperation.

## **Publisher's Note**

Remember to write to us. We look forward to receiving your feedback, comments, and ideas to enhance the quality of this book. You can reach us at [info.india@mheducation.com](mailto:info.india@mheducation.com). Please mention the title and authors' name as the subject. In case you spot piracy of this book, please do let us know.

# Contents

*Preface*

*Roadmap to the Syllabus*

v

xiii

## **UNIT-I LINEAR DATA STRUCTURES – LIST**

---

<b>1. Introduction to Algorithm and Data Structures</b>	<b>1.1</b>
1.1 Introduction	1.2
1.2 Algorithms	1.2
1.2.1 Characteristics of an Algorithm	1.3
1.2.2 Representation of an Algorithm	1.3
1.2.3 Efficiency of an Algorithm	1.5
1.3 Asymptotic Notation	1.6
1.3.1 Big-Oh Notation	1.6
1.3.2 Omega Notation	1.7
1.3.3 Theta Notation	1.9
1.4 Introduction to Data Structures	1.10
1.4.1 Characteristics of Data Structure	1.11
1.5 Types of Data Structures	1.11
1.5.1 Arrays	1.12
1.5.2 Linked Lists	1.12
1.5.3 Stacks	1.13
1.5.4 Queues	1.13
1.5.5 Trees	1.14
1.5.6 Graphs	1.14
1.6 Data Structure Operations	1.15
1.6.1 Data Structure Efficiency	1.15
Summary	1.15
Key Terms	1.16
Multiple-Choice Questions	1.16
Review Questions	1.17
<b>2. Arrays</b>	<b>2.1</b>
2.1 Introduction	2.2
2.2 Types of Arrays	2.2
2.3 Representation of One-Dimensional Array in Memory	2.3

2.4	Array Traversal	2.3
2.5	Insertion and Deletion	2.5
2.5.1	Insertion	2.5
2.5.2	Deletion	2.8
2.6	Sorting and Searching	2.10
2.6.1	Sorting	2.10
2.6.2	Searching	2.13
2.8	Realizing Matrices using Two-Dimensional Arrays	2.16
2.9	Matrix Operations	2.18
2.9.1	Addition	2.18
2.9.2	Subtraction	2.21
2.9.3	Multiplication	2.21
2.9.4	Transpose	2.24
<i>Solved Problems</i>		2.26
<i>Summary</i>		2.27
<i>Key Terms</i>		2.27
<i>Multiple-Choice Questions</i>		2.28
<i>Review Questions</i>		2.29
<i>Programming Exercises</i>		2.29

### 3. Linked Lists

3.1

3.1	Introduction	3.2
3.2	Linked Lists—Basic Concept	3.2
3.2.1	Representation of Linked Lists	3.2
3.2.2	Advantages of Linked Lists	3.3
3.2.3	Disadvantages of Linked Lists	3.3
3.3	Linked List Implementation	3.3
3.3.1	Linked List Node Declaration	3.3
3.3.2	Linked List Operations	3.4
3.3.3	Linked List Implementation	3.7
3.4	Types of Linked Lists	3.15
3.5	Circular Linked List	3.15
3.5.1	Circular Linked List Operations	3.16
3.5.2	Circular Linked List Implementation	3.17
3.6	Doubly Linked List	3.24
3.6.1	Doubly Linked List Node Declaration	3.25
3.6.2	Doubly Linked List Operations	3.25
3.6.3	Doubly Linked List Implementation	3.27
<i>Solved Problems</i>		3.32
<i>Summary</i>		3.33
<i>Key Terms</i>		3.33

*Multiple-Choice Questions* 3.34

*Review Questions* 3.35

*Programming Exercises* 3.35

## **UNIT-II LINEAR DATA STRUCTURES – STACKS, QUEUES**

---

### **4. Stacks**

**4.1**

4.1 Introduction 4.2

4.2 Stacks 4.2

    4.2.1 Stack Representation in Memory 4.2

    4.2.2 Arrays vs. Stacks 4.3

4.3 Stack Operations 4.3

    4.3.1 Push 4.4

    4.3.2 Pop 4.4

    4.3.3 An Example of Stack Operations 4.5

4.4 Stack Implementation 4.5

    4.4.1 Array Implementation of Stacks 4.5

    4.4.2 Linked Implementation of Stacks 4.11

*Solved Problems* 4.16

*Summary* 4.19

*Key Terms* 4.19

*Multiple-Choice Questions* 4.20

*Review Questions* 4.21

*Programming Exercises* 4.22

### **5. Queues**

**5.1**

5.1 Introduction 5.2

5.2 Queues—Basic Concept 5.2

    5.2.1 Logical Representation of Queues 5.3

5.3 Queue Operations 5.4

5.4 Queue Implementation 5.6

    5.4.1 Array Implementation of Queues 5.6

    5.4.2 Linked Implementation of Queues 5.12

5.5 Circular Queues 5.17

5.6 Priority Queues 5.25

5.7 Double-Ended Queues 5.31

*Solved Problems* 5.34

*Summary* 5.37

*Key Terms* 5.37

*Multiple-Choice Questions* 5.38

*Review Questions* 5.39

*Programming Exercises* 5.39

## **UNIT-III NON LINEAR DATA STRUCTURES - TREES**

---

### **6. Trees**

**6.1**

6.1 Introduction 6.2

6.2 Basic Concept 6.2

    6.2.1 Tree Terminology 6.2

6.3 Binary Tree 6.3

    6.3.1 Binary Tree Concepts 6.4

6.4 Binary Tree Representation 6.5

    6.4.1 Array Representation 6.5

    6.4.2 Linked Representation 6.6

6.5 Binary Tree traversal 6.10

6.6 Binary Search Tree 6.17

6.7 Tree Variants 6.24

    6.7.1 Expression Trees 6.25

    6.7.2 Threaded Binary Trees 6.26

    6.7.3 Balanced Trees 6.27

    6.7.4 Splay Trees 6.30

    6.7.5 m-way Trees 6.31

*Summary* 6.33

*Multiple-Choice Questions* 6.34

*Key Terms* 6.34

*Review Questions* 6.35

*Programming Exercises* 6.35

## **UNIT-IV NON LINEAR DATA STRUCTURES - GRAPHS**

---

### **7. Graphs**

**7.1**

7.1 Introduction 7.2

7.2 Basic Concept 7.2

7.3 Graph Terminology 7.3

7.4 Graph Implementation 7.4

    7.4.1 Implementing Graphs using Adjacency Matrix 7.4

    7.4.2 Implementing Graphs using Path Matrix 7.6

    7.4.3 Implementing Graphs using Adjacency List 7.8

7.5	Shortest Path Algorithm	7.11
7.6	Graph Traversal	7.15
7.6.1	Breadth First Search	7.15
7.6.2	Depth First Search	7.16
	<i>Summary</i>	7.17
	<i>Key Terms</i>	7.17
	<i>Multiple-Choice Questions</i>	7.17
	<i>Review Questions</i>	7.18
	<i>Programming Exercises</i>	7.18

## **UNIT-V SEARCHING, SORTING AND HASHING TECHNIQUES**

---

<b>8.</b>	<b>Sorting and Searching</b>	<b>8.1</b>
8.1	Introduction	8.2
8.2	Sorting Techniques	8.2
8.2.1	Selection Sort	8.3
8.2.2	Insertion Sort	8.7
8.2.3	Bubble Sort	8.10
8.2.4	Quick Sort	8.14
8.2.5	Merge Sort	8.19
8.2.6	Bucket Sort	8.23
8.3	Searching Techniques	8.28
8.3.1	Linear Search	8.28
8.3.2	Binary Search	8.31
8.3.3	Hashing	8.34
	<i>Solved Problems</i>	8.37
	<i>Summary</i>	8.38
	<i>Key Terms</i>	8.39
	<i>Multiple-Choice Questions</i>	8.39
	<i>Review Questions</i>	8.40
	<i>Programming Exercises</i>	8.40



# Roadmap to the Syllabus

## Data Structures Semester III

### Unit-I: LINEAR DATA STRUCTURES – LIST

Abstract Data Types (ADTs) – List ADT – array-based implementation – linked list implementation – singly linked lists- circularly linked lists- doubly-linked lists – applications of lists –Polynomial Manipulation – All operations (Insertion, Deletion, Merge, Traversal)

 GO TO

- Chapter 1: Introduction to Algorithm and Data Structures**
- Chapter 2: Arrays**
- Chapter 3: Linked Lists**

### Unit-II: LINEAR DATA STRUCTURES – STACKS, QUEUES

Stack ADT – Operations - Applications - Evaluating arithmetic expressions- Conversion of Infix to postfix expression - Queue ADT – Operations - Circular Queue – Priority Queue - deQueue – applications of queues

 GO TO

- Chapter 4: Stacks**
- Chapter 5: Queues**

### Unit-III: NON LINEAR DATA STRUCTURES – TREES

Tree ADT – tree traversals - Binary Tree ADT – expression trees – applications of trees – binary search tree ADT –Threaded Binary Trees- AVL Trees – B-Tree - B+ Tree - Heap – Applications of heap.

 GO TO

- Chapter 6: Trees**

## **Unit-IV: NON LINEAR DATA STRUCTURES – GRAPHS**

Definition – Representation of Graph – Types of graph – Breadth-first traversal – Depth-first traversal – Topological Sort – Bi-connectivity – Cut vertex – Euler circuits – Applications of graphs.

**GO TO**

**Chapter 7: Graphs**

## **Unit-V: SEARCHING, SORTING AND HASHING TECHNIQUES**

Searching- Linear Search – Binary Search. Sorting – Bubble sort – Selection sort – Insertion sort – Shell sort – Radix sort. Hashing – Hash Functions – Separate Chaining – Open Addressing – Rehashing – Extendible Hashing

**GO TO**

**Chapter 8: Sorting and Searching**

# **UNIT-I**

---

# **Linear Data Structures – List**

## **CHAPTERS**

Chapter 1: Introduction to Algorithm and Data Structures

Chapter 2: Arrays

Chapter 3: Linked Lists



# 1

## INTRODUCTION TO ALGORITHM AND DATA STRUCTURES

Chapter outline

- 1.1 Introduction
- 1.2 Algorithms
  - 1.2.1 Characteristics of an Algorithm
  - 1.2.2 Representation of an Algorithm
  - 1.2.3 Efficiency of an Algorithm
- 1.3 Asymptotic Notations
  - 1.3.1 Big-Oh Notation
  - 1.3.2 Omega Notation
  - 1.3.3 Theta Notation
- 1.4 Introduction to Data Structures
  - 1.4.1 Characteristics of Data Structures
- 1.5 Types of Data Structures
  - 1.5.1 Arrays
  - 1.5.2 Linked Lists
  - 1.5.3 Stacks
  - 1.5.4 Queues
  - 1.5.5 Trees
  - 1.5.6 Graphs
- 1.6 Data Structure Operations
  - 1.6.1 Data Structure Efficiency

*Summary*

*Key Terms*

*Multiple-Choice Questions*

*Review Questions*

*Answers to Multiple-Choice Questions*



## 1.1 INTRODUCTION

In the last two chapters, we learnt the basics of C programming language and its various programming constructs. In this chapter, we will focus on one of the main starting points to developing programming applications, i.e., algorithms. An algorithm is a set of instructions that defines the complete solution to a given problem. It uses simple English language for writing the solution steps. It is quite possible to have multiple algorithmic solutions for the same problem. In such cases, performance becomes the sole criterion for choosing a specific solution. We can use various asymptotic notations, such as big-oh and omega for assessing the performance or running time of an algorithm.

In this chapter, we will also get a brief introduction to data structures. Data structure is a collection of data and the associated operations. Some of the commonly used data structures are arrays, stacks, queues, linked lists, etc.

## 1.2 ALGORITHMS

An algorithm can be defined as a step by step procedure that provides solution to a given problem. It comprises of a well-defined set of finite number of steps or rules that are executed sequentially to obtain the desired solution.

To understand algorithms in a better way, let us consider a simple problem of identifying the smallest number from a given list of numbers. Following is the algorithm for this problem:

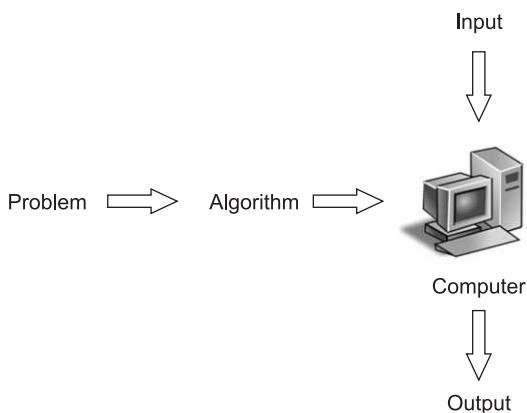
**Select the first number in the list and tag it as the smallest-so-far element.**

1. **For each subsequent element in the list.**
2. **Replace the smallest-so-far number with the list element if the latter is smaller.**
3. **Once all the numbers have been compared, the smallest-so-far number is considered as the smallest number in the list.**

In computing terms, an algorithm is described a little differently. It is defined as a hierarchy of steps used for computational procedures, which usually starts with an input value and generates the desired output. While defining an algorithm, you must consider two primary factors, the time it requires to solve the problem and the required memory space. For instance, if an algorithm takes hours to solve a problem, then it is of no use. Similarly, if an algorithm requires gigabytes of computer memory, then also it is not considered an ideal algorithm.

Figure 1.1 shows a simple illustration of how algorithms are used for solving computational problems.

An algorithm solves only a single problem at a time. However, the same problem can be solved using multiple algorithms. The benefit of



**Fig. 1.1 Use of algorithms for solving computational problems**

using multiple algorithms to solve the same problem is purely situational. One algorithm could be more efficient for a particular set of inputs or for a specific variation of the problem while another algorithm could be more efficient for some different set of inputs or for some different variation of the problem. The use of multiple algorithms is more evident while solving sorting problems. One sorting algorithm could be efficient for sorting a large collection of integers while another sorting algorithm could be more efficient for sorting a large collection of strings. Thus, in this case the choice of a particular algorithm solely depends on the type of input values.

### 1.2.1 Characteristics of an Algorithm

There are certain key characteristics that an algorithm must possess. These characteristics are:

1. An algorithm must comprise of a **finite** number of steps.
2. It should have zero or more valid and clearly defined **input** values.
3. It should be able to generate at least a single valid **output** based on a valid input.
4. It must be **definite**, i.e., each instruction in the algorithm should be defined clearly.
5. It should be **correct**, i.e., it should be able to perform the desired task of generating correct output from the given input.
6. There should be no **ambiguity** regarding the order of execution of algorithm steps.
7. It should be able to **terminate** on its own, i.e., it should not go into an infinite loop.

### 1.2.2 Representation of an Algorithm

You can represent an algorithm in a number of ways, right from normal English language phrases to graphical representation using flow charts. However, such representations are mainly useful when the algorithm is simple and small.

Another way of representing an algorithm is the pseudocode. Pseudocode is an informal representation of the algorithm that provides a complete outline of a program so that the programmers can easily understand it and transform it into a program using the programming language of their choice. The structure and syntax of pseudocode is quite similar to typical programming language constructs, thus it is easy to transform it into a program. Since there are no tight syntactical constraints associated with developing a pseudocoded algorithm, the programmer has the liberty to focus only on getting the solution logic right.

While representing an algorithm in pseudocode form, you must use certain conventions consistently throughout the algorithm. This helps in easy understanding of the algorithm. Following are some of the general conventions that are followed while writing pseudocode:

1. Provide a valid name for the algorithm written using pseudocode.
2. For each line of instruction, specify a line number.
3. Always begin an identifier name with English alphabet.
4. It is not necessary to explicitly specify the data type of the variables.
5. Always indent the statements present inside a block structure appropriately.
6. Use **read** and **write** instructions to specify input and output operations respectively.
7. Use **if** or **if else** constructs for conditional statements. You must end an **if** statement with the corresponding **end if** statement. Further, each **if** construct should be vertically aligned, depicted as follows:

```
If (conditional expression)
    Statement
end-if
```

Or

```
If (conditional expression)
    Statement
else
    Statement
end-if
```

8. For looping or iterative statements, you can use **for** or **while** looping constructs. A **for** loop must end with an **end for** statement while a **while** loop must end with an **end while** statement, as depicted below:

```
for i = 1 to 10 do
{
    Statement 1
    .
    .
    Statement n
}
end-for

while (conditional expression) do
{
    Statement 1
    .
    .
    Statement n
}
end-while
```

9. Use logical and relational operators whenever logical or relational operations are to be performed.  
For example,

```
i = j
i < j
i ≥ j
```

10. Represent an array or list element by specifying the name of the array followed by its index within square brackets. For instance, A[i] will represent the  $i^{\text{th}}$  element of the array A.

Let us now go through some examples of algorithms created using pseudocode.

**Example 1.1** Write an algorithm to interchange two numbers.

#### Interchange (X, Y)

Step 1: Begin

Step 2: Set X = X + Y

```
Step 3: Set Y = X - Y  
Step 4: Set X = X - Y  
Step 5: Write (X, Y)  
Step 6: End
```

**Example 1.2** Write an algorithm to calculate the average of 15 numbers.

```
Average (avg, sum)  
Step 1: Begin  
Step 2: Set avg = 0.0 and sum = 0  
Step 3: for i = 1 to 15 do  
Step 4: Read (a)  
Step 5: sum = sum + a  
Step 6: end-for  
Step 7: avg = sum/15  
Step 8: Write (avg)  
Step 9: End
```

**Example 1.3** Write an algorithm to sort **n** numbers.

```
Sort (a, n)  
Step 1: Begin  
Step 2: Read (n)  
Step 3: for i = n to 2 do  
Step 4: for j = 1 to i-1 do  
Step 5: if a[j]>a[j+1] then  
Step 6: Interchange a[j] and a[j+1]  
Step 7: end-if  
Step 8: end-for  
Step 9: end-for  
Step 10: End
```

### 1.2.3 Efficiency of an Algorithm

Whenever we refer the term efficiency in the context of algorithms, it points at two aspects: one, whether the algorithm runs faster; and two, whether it uses lesser amount of memory space. Thus, an efficient algorithm will always create the best possible tradeoff between its running time and memory space consumption.

The function that derives the running time of an algorithm and its memory space requirements for a given set of inputs is referred as algorithm complexity. Time complexity is the measure of the running time of an algorithm for a given set of inputs. Space complexity is the measure of the amount of memory space required by an algorithm for its complete execution, for a given set of inputs.

Time complexity is typically measured by counting the number of primitive or elementary steps performed by the algorithm for its complete execution. These steps are machine independent and their count is directly dependent on the size of input data set. The representation or expression of time complexity is done asymptotically, as we shall see in the subsequent section.

## 1.3 ASYMPTOTIC NOTATION

Asymptotic notation is the most simple and easiest way of describing the running time of an algorithm. It represents the efficiency and performance of an algorithm in a systematic and meaningful manner. Asymptotic notations describe time complexity in terms of three common measures, best case (or ‘fastest possible’), worst case (or ‘slowest possible’), and average case (or ‘average time’).

The three most important asymptotic notations are:

1. Big-Oh notation
2. Omega notation
3. Theta notation

### 1.3.1 Big-Oh Notation

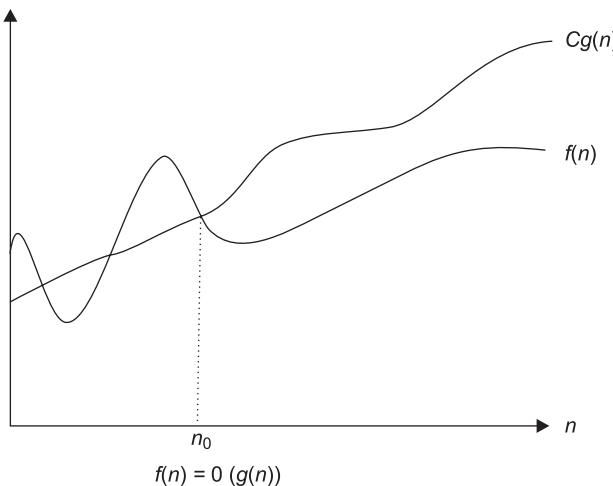
The big-oh notation is a method that is used to express the upper bound of the running time of an algorithm. It is denoted by ‘O’. Using this notation, we can compute the maximum possible amount of time that an algorithm will take for its completion.

**Definition** Consider  $f(n)$  and  $g(n)$  to be two positive functions of  $n$ , where  $n$  is the size of the input data. Then,  $f(n)$  is big-oh of  $g(n)$ , if and only if there exists a positive constant  $C$  and an integer  $n_0$ , such that

$$f(n) \leq Cg(n) \text{ and } n > n_0$$

Here,  $f(n) = O(g(n))$

Figure 1.2 shows the graphical representation of big-oh notation.



**Fig. 1.2** Graphical representation of big-oh notation

Some of the typical complexities (computing time) represented by big-oh notation are:

1.  $O(1) \rightarrow$  Constant
2.  $O(n) \rightarrow$  Linear
3.  $O(n^2) \rightarrow$  Quadratic

4.  $O(n^3) \rightarrow$  Cubic
5.  $O(2^n) \rightarrow$  Exponential
6.  $O(\log n) \rightarrow$  Logarithmic

**Example 1.4** Derive the big-oh notation, if  $f(n) = 8n + 7$  and  $g(n) = n$ .

**Solution** To show  $f(n)$  is  $O(g(n))$ , we must consider positive constants  $C$  and integer  $n_0$ , such that  $f(n) \leq Cg(n)$  for all  $n > n_0$   
or  $8n + 7 \leq Cn$  for all  $n > n_0$

Let  $C = 15$ .

Now, we must show that  $8n + 7 \leq 15n$ .

or  $7 \leq 7n$

or  $1 \leq n$

Therefore,  $f(n) = 8n + 7 \leq 15n$  for all  $n \geq 1$ , where  $C = 15$  and  $n_0 = 1$ .

Hence,  $f(n) = O(g(n))$ .

**Example 1.5** Derive the big-oh notation, if  $f(n) = 2n + 2$  and  $g(n) = n^2$ .

**Solution** Given,  $f(n) = 2n + 2$  and  $g(n) = n^2$ .

For  $n = 1$ ,

$$\begin{aligned}f(n) &= 2(1) + 2 \\&= 4\end{aligned}$$

$$\begin{aligned}g(n) &= (1)^2 \\&= 1\end{aligned}$$

i.e.,  $f(n) > g(n)$

For  $n = 2$ ,

$$\begin{aligned}f(n) &= 2(2) + 2 \\&= 6\end{aligned}$$

$$\begin{aligned}g(n) &= (2)^2 \\&= 4\end{aligned}$$

i.e.,  $f(n) > g(n)$

For  $n = 3$ ,

$$\begin{aligned}f(n) &= 2(3) + 2 \\&= 8\end{aligned}$$

$$\begin{aligned}g(n) &= (3)^2 \\&= 9\end{aligned}$$

i.e.,  $f(n) < g(n)$

Therefore,  $f(n) \leq Cg(n)$  is true if  $n > 2$ .

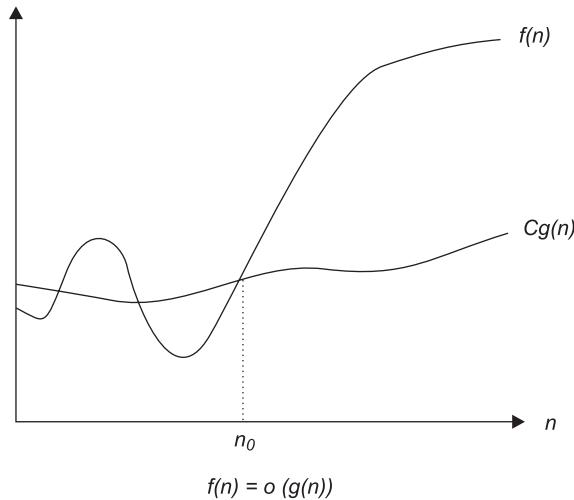
### 1.3.2 Omega Notation

The omega notation is a method that is used to express the lower bound of the running time of an algorithm. Omega notation is denoted by ' $\Omega$ '. Using this notation, you can compute the minimum amount of time that an algorithm will take for its completion.

**Definition** Consider  $f(n)$  and  $g(n)$  to be two positive functions of  $n$ , where  $n$  is the size of the input data. Then,  $f(n)$  is omega of  $g(n)$ , if and only if there exists a positive constant  $C$  and an integer  $n_0$ , such that  $f(n) \geq Cg(n)$  and  $n > n_0$

Here,  $f(n) = \Omega(g(n))$

Figure 1.3 shows the graphical representation of omega notation.



**Fig. 1.3** Graphical representation of Omega notation

**Example 1.6** Deduce the omega notation if  $f(n) = 2n^2 + 4$  and  $g(n) = 6n$ .

Given,  $f(n) = 2n^2 + 4$  and  $g(n) = 6n$ .

For  $n = 0$ ,

$$\begin{aligned} f(n) &= 2(0)^2 + 4 \\ &= 4 \end{aligned}$$

$$\begin{aligned} g(n) &= 6(0) \\ &= 0 \end{aligned}$$

i.e.,  $f(n) > g(n)$

For  $n = 1$ ,

$$\begin{aligned} f(n) &= 2(1)^2 + 4 \\ &= 2 + 4 \\ &= 6 \end{aligned}$$

$$\begin{aligned} g(n) &= 6(1) \\ &= 6 \end{aligned}$$

i.e.,  $f(n) = g(n)$

For  $n = 2$ ,

$$f(n) = 2(2)^2 + 4$$

$$= 8 + 4$$

$$= 12$$

$$g(n) = 6(2)$$

$$= 12$$

i.e.,  $f(n) = g(n)$

For  $n = 3$ ,

$$f(n) = 2(3)^2 + 4$$

$$= 18 + 4$$

$$= 22$$

$$g(n) = 6(3)$$

$$= 18$$

i.e.,  $f(n) > g(n)$

Therefore, we can say that  $f(n) > Cg(n)$ , if  $n > 2$ .

**Example 1.7** Deduce the omega notation if  $f(n) = 2n + 6$  and  $g(n) = 2n$ .

Given,  $f(n) = 2n + 6$  and  $g(n) = 2n$ .

For  $n = 0$ ,

$$f(n) = 2(0) + 6$$

$$= 6$$

$$g(n) = 2(0)$$

$$= 0$$

i.e.,  $f(n) > g(n)$

For  $n = 1$ ,

$$f(n) = 2(1) + 6$$

$$= 2 + 6$$

$$= 8$$

$$g(n) = 2(1)$$

$$= 2$$

i.e.,  $f(n) > g(n)$

Therefore, we can say that  $f(n) > Cg(n)$ , for  $n > 1$ .

### 1.3.3 Theta Notation

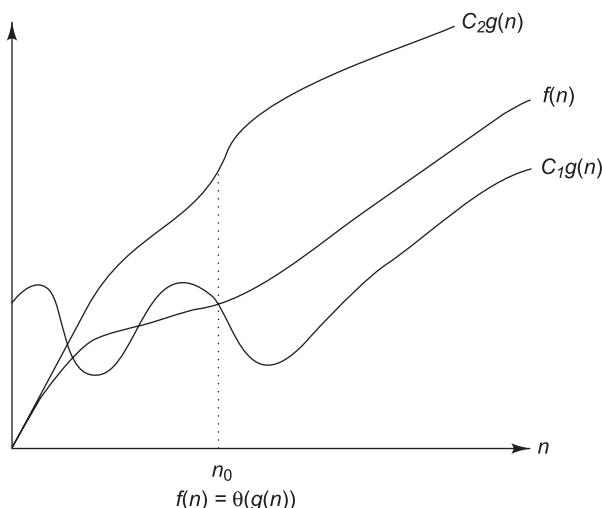
The theta notation is a method that is used to express the running time of an algorithm between the lower and upper bounds. Theta notation is denoted by ' $\theta$ '. Using this notation, we can compute the average time that an algorithm will take for its completion.

**Definition** Consider  $f(n)$  and  $g(n)$  to be two positive functions of  $n$ , where  $n$  is the size of the input data. Then,  $f(n)$  is theta of  $g(n)$ , if and only if there exists two positive constants  $C_1$  and  $C_2$ , such that,

$$C_1 g(n) \leq f(n) \leq C_2 g(n)$$

Here,  $f(n) = \theta(g(n))$ .

Figure 1.4 shows the graphical representation of theta notation.



**Fig. 1.4** Graphical representation of Theta notation

**Example 1.8** Deduce the theta notation if  $f(n) = 2n + 8$ .

Let  $f(n) = 2n + 8 > 5n$  where  $n \geq 2$

Similarly,  $f(n) = 2n + 8 > 6n$  where  $n \geq 2$

and  $f(n) = 2n + 8 < 7n$  where  $n \geq 2$

Thus,  $5n < 2n + 8 < 7n$ , for  $n \geq 2$ ,

Here  $C_1 = 5$  and  $C_2 = 7$

Hence,  $f(n) = 2n + 8 = \theta(n)$

## 1.4 INTRODUCTION TO DATA STRUCTURES

In simple terms, data structure can be defined as a representation of data along with its associated operations. It is the way of organizing and storing data in a computer system so that it can be used efficiently. This organization can be in the form of a group of data elements stored under one name. Here, the data elements are referred as members of the data structure.

Depending on the type of data structures, the members can be of different types and lengths. Some of the examples of data structures include arrays, linked lists, binary trees, stacks, etc. Algorithms are used to manipulate the data structures in a number of different ways, like sorting the data elements or searching a particular data item.

The design and implementation of a typical data structure is associated with the definition of the operations that can be performed on the data structure. The specification of these data structure operations is done with the help of algorithms.

### 1.4.1 Characteristics of Data Structure

Data structures help in storing, organizing, and analyzing the data in a logical manner. The following points highlight the need of data structures in computer science:

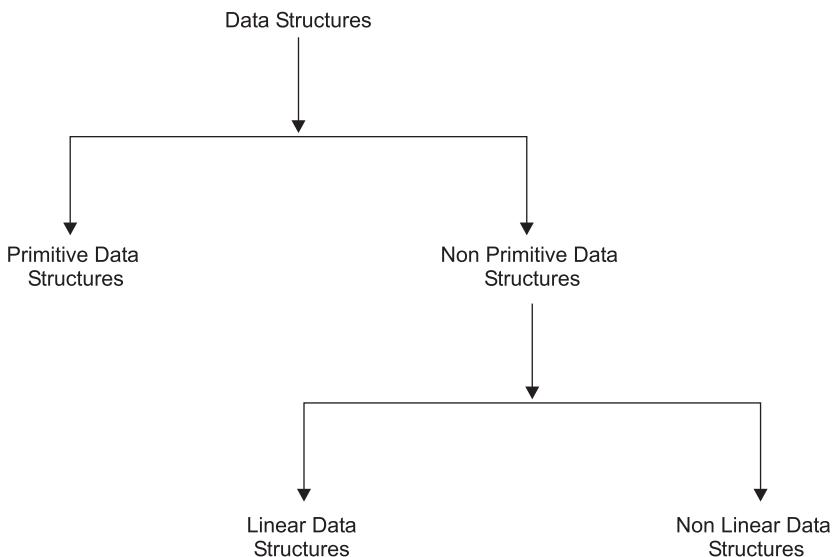
1. It depicts the logical representation of data in computer memory.
2. It represents the logical relationship between the various data elements.
3. It helps in efficient manipulation of stored data elements.
4. It allows the programs to process the data in an efficient manner.

## 1.5 TYPES OF DATA STRUCTURES

Data structures are primarily divided into two classes, primitive and non-primitive. Primitive data structures include all the fundamental data structures that can be directly manipulated by machine-level instructions. Some of the common primitive data structures include integer, character, real, boolean, etc. Non-primitive data structures, on the other hand, refer to all those data structures that are derived from one or more primitive data structures. The objective of creating non-primitive data structures is to form sets of homogeneous or heterogeneous data elements.

Non-primitive data structures are further categorized into two types: linear and non-linear. In linear data structures, all the data elements are arranged in a linear or sequential fashion. Examples of linear data structures include arrays, stacks, queues, linked lists, etc. In non-linear data structures, there is no definite order or sequence in which data elements are arranged. For instance, a non-linear data structure could arrange data elements in a hierarchical fashion. Examples of non-linear data structures are trees and graphs.

Figure 1.5 shows the classification of different types of data structures.



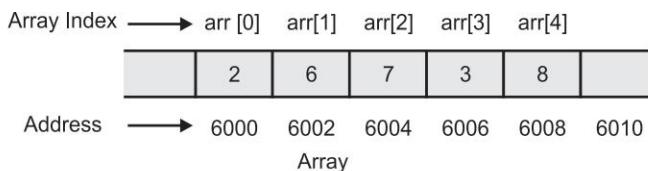
**Fig. 1.5 Types of data structures**

The subsequent sections give a brief overview of some of the important data structures. Each of these data structures will be covered in detail in later chapters.

### 1.5.1 Arrays

An array is a collection of similar type data elements stored at consecutive locations in the memory. Typical examples of arrays include list of integers, group of names, etc. The group of array elements is referred with a common name called *array name*. Access to individual array elements is provided with the help of an index identifier. In C language, array index starts with 0. For example, list[5] refers to the 6<sup>th</sup> element of the array ‘list’.

Figure 1.6 shows the logical representation of arrays.



**Fig. 1.6** Logical representation of arrays

**Application of Arrays** Arrays are particularly used in programs that require storing large collection of similar type data elements.



#### Note

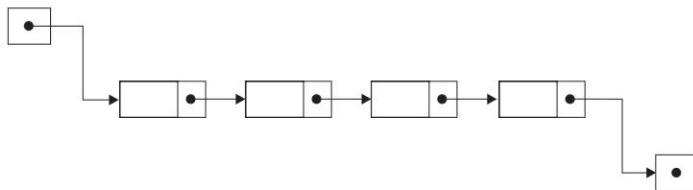
For more information on arrays, refer to Chapter 4.

### 1.5.2 Linked Lists

Linked list is a data structure used for storing data in the form of a list. It comprises of multiple nodes connected to each other through pointers. Each node comprises of two parts. One part contains the data value while the other part contains a pointer to the next node in the list.

Linked lists eliminate one of the main disadvantages associated with arrays, that is inefficient utilization of memory space. A linked list blocks only that much amount of memory space as is required for storing its constituent data elements. Every time a new element is to be inserted into the linked list, a corresponding new node is created. This is in contrast to arrays, which block a fixed amount of memory space irrespective of their precise requirement.

Figure 1.7 shows the logical representation of a linked list.



**Fig. 1.7** Logical representation of linked lists

**Application of Linked Lists** Linked lists are used in situations where there is a need for dynamic memory allocation. For instance, a number of data structures like stacks, queues, trees, etc., are implemented with the help of linked lists.

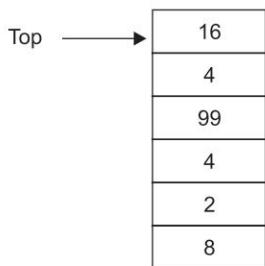


**Note** For more information on linked lists, refer to Chapter 5.

### 1.5.3 Stacks

Stack is a linear data structure that maintains a list of elements in such a manner that elements can be inserted or deleted only from one end of the list. This end is referred as top of the stack. Stack is based on the Last-In-First-Out (LIFO) principle, which means the element that is last added to the stack is the one that is first removed from the stack. A stack of books can be considered similar to a stack data structure as it allows the books to be added or removed only from the top end of the stack and not from the middle.

Figure 1.8 shows the logical representation of a stack.



**Fig. 1.8** Logical representation of stacks

**Application of Stacks** Stacks are typically used in the implementation of system processes, such as compilation and program control.

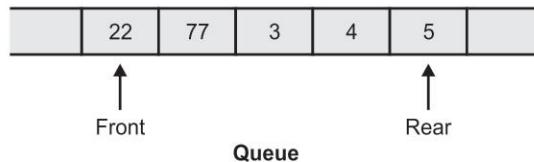


**Note** For more information on stacks, refer to Chapter 6.

### 1.5.4 Queues

Queue is a linear data structure that maintains a list of elements in such a manner that elements are inserted from one end of the queue (called *rear*) and deleted from the other end (called *front*). Queue is based on the First-In-First-Out (FIFO) principle, which means the element that is first added to the queue is also the one that is first removed from the queue. A queue of people standing at a bus stop can be considered similar to a queue data structure as people join the queue at the back and leave the queue from the front.

Figure 1.9 shows the logical representation of a queue.



**Fig. 1.9** Logical representation of queues

**Application of Queues** Queues are typically used in the implementation of key system processes such as CPU scheduling, resource sharing, etc.



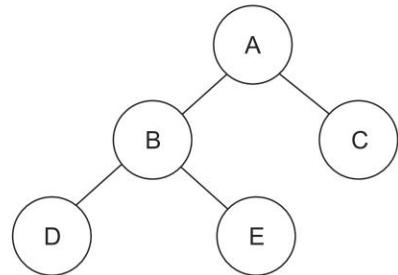
**Note** For more information on queues, refer to Chapter 7.

### 1.5.5 Trees

Tree is a linked data structure that arranges its nodes in the form of a hierarchical tree structure. Each node comprises of zero or more child nodes. The node present at the top of the tree structure is referred as root node. Data is accessed from the tree data structure through various tree traversal methods.

Figure 1.10 shows the representation of a tree.

**Application of Trees** Tree data structure is typically used for storing hierarchical data, implementing search trees, and maintaining sorted data.



**Fig. 1.10** Tree

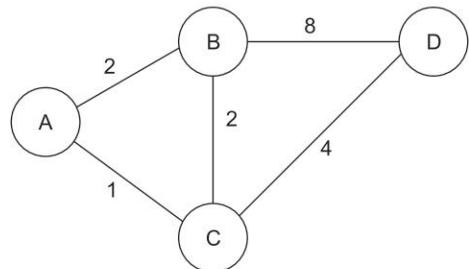


**Note** For more information on trees, refer to Chapter 8.

### 1.5.6 Graphs

Graph is a linked data structure that comprises of a group of vertices called *nodes* and a group of edges. An edge is nothing but a pair of vertices. Graph data structure realizes the mathematical concept of graphs. The edges of a graph are typically associated with certain values also called weights. This helps to compute the cost of traversing the graph through a certain path.

Figure 1.11 shows the representation of a graph.



**Fig. 1.11** Graph

**Application of Graphs** One of the typical application areas of graphs is in the modelling of networking systems. It helps to compute the cost of transmitting data from a particular network path.



**Note** For more information on trees, refer to Chapter 9.

## 1.6 DATA STRUCTURE OPERATIONS

There are several common operations associated with data structures that are used for manipulating the stored data. While defining a data structure, you also need to define these associated operations. The following operations are most frequently performed on any data structure type:

1. **Traversing** It is the process of accessing each record of a data structure exactly once.
2. **Searching** It is the process of identifying the location of a record that contains a specific key value.
3. **Inserting** It is the process of adding a new record in to a data structure.
4. **Deleting** It is the process of removing an existing record from a data structure.

Apart from these typical data structure operations, there are some other operations associated with data structures, such as

1. **Sorting** It is the process of arranging the records of a data structure in a specific order, such as alphabetical, ascending, or descending.
2. **Merging** It is the process of combining the records of two different sorted data structures to produce a single sorted data set.

### 1.6.1 Data Structure Efficiency

Table 1.1 lists the efficiency of various data structure types for different operations.

**Table 1.1** *Efficiency of various data structure types*

Data Structure	Insert	Search	Delete
Array	$O(n)$	$O(n)$	$O(n)$
Linked List	$O(1)$	$O(n)$	$O(n)$
Stack	$O(1)$	–	$O(1)$
Queue	$O(1)$	–	$O(1)$
Tree (Sorted)	$O(\log n)$	$O(\log n)$	$O(\log n)$

## »»» Summary



- ◆ An algorithm is a well-defined set of finite number of steps or rules that are executed sequentially to obtain the desired solution.
- ◆ Algorithms can be represented in the form of pseudocode or flowcharts.
- ◆ The function that derives the running time of an algorithm and its memory space requirements for a given set of inputs is referred as algorithm complexity.
- ◆ Asymptotic notations describe time complexity in terms of three common measures, best case, worst case, and average case.

- ◆ The various types of asymptotic notations are big-oh, omega, and theta.
- ◆ Data structure is the way of organizing and storing data in a computer system.
- ◆ Data structures are primarily divided into two classes, primitive and non-primitive.
- ◆ Primitive data structures include all the fundamental data structures such as integer, character, real, etc.
- ◆ Non-primitive data structures are the ones that are derived from one or more primitive data structures. Examples of non-primitive data structures include arrays, linked lists, stacks, queues, etc.
- ◆ An array is a collection of similar type data elements stored at consecutive locations in the memory.
- ◆ Linked list is a collection of nodes connected to each other through pointers.
- ◆ Stack is a linear data structure that stores the data elements based on Last-In-First-Out (LIFO) principle, which means the element that is last added to the stack is the one that is first removed from the stack.
- ◆ Queue is a linear data structure that stores the data elements based on First-In-First-Out (FIFO) principle, which means the element that is first added to the stack is also the one that is first removed from the stack.
- ◆ Tree is a linked data structure that arranges its nodes in the form of a hierarchical tree structure.
- ◆ Graph is a linked data structure that comprises of a group of vertices called nodes and a group of edges.

## ▶▶ Key Terms

- ◆ **Pseudocode** It is an informal representation of the algorithm that provides a complete outline of a program.
- ◆ **Time complexity** It is the measure of the running time of an algorithm.
- ◆ **Space complexity** It is the measure of the amount of memory space required by the algorithm for its complete execution.
- ◆ **Big-oh** It is an asymptotic notation that expresses the upper bound of the running time of an algorithm.
- ◆ **Omega** It is an asymptotic notation that expresses the lower bound of the running time of an algorithm.
- ◆ **Theta** It is an asymptotic notation that expresses the running time of an algorithm between the lower and upper bounds.
- ◆ **LIFO** It stands for Last-In-First-Out i.e., the principle on which stacks are based.
- ◆ **FIFO** It stands for First-In-First-Out i.e., the principle on which queues are based.
- ◆ **Root node** It is the node present at the top of a tree data structure.

## Multiple-Choice Questions

- 1.1** Which of the following is not true for algorithms?
- It is a set of instructions that defines the solution for a given problem.
  - It can be represented in the form of pseudocode or flowchart.

- (c) It should have at least one valid input value.  
(d) It is possible to have multiple algorithms for the same problem.

**1.2** Efficiency of an algorithm is a tradeoff between which of the following factors?  
(a) Time and Space  
(b) Input and Output  
(c) Compilation time and Running time  
(d) None of the above

**1.3** Big-oh notation is a method that is used to express the \_\_\_\_\_ of the running time of an algorithm.  
(a) Lower bound  
(b) Upper bound  
(c) Lower and upper bound  
(d) None of the above

**1.4** \_\_\_\_\_ notation is a method that is used to express the running time of an algorithm between the lower and upper bounds.  
(a) Big-oh  
(b) Beta  
(c) Theta  
(d) Omega

**1.5** Which of the following is the correct representation of the Omega notation?  
(a)  $f(n) \geq Cg(n)$   
(b)  $f(n) \leq Cg(n)$   
(c)  $C_1 g(n) \leq f(n) \leq C_2 g(n)$   
(d) None of the above

**1.6** Which of the following is an example of primitive data structure?  
(a) Integer  
(b) Array  
(c) Character  
(d) Stack

**1.7** Which of the following data structure is based on FIFO principle?  
(a) Tree  
(b) Graph  
(c) Stack  
(d) Queue

**1.8** A linked list blocks only that much amount of memory space as is required for storing its constituent data elements.  
(a) True  
(b) False

**1.9** Which of the following data structure arranges its nodes in the form of a hierarchical structure?  
(a) Stac  
(b) Graph  
(c) Linked List  
(d) Tree

**1.10** Which of the following is a typical data structure operation?  
(a) Insert  
(b) Delete  
(c) Search  
(d) All of the above

## Review Questions

- 1.1 What is an algorithm? Explain with the help of an example.
  - 1.2 List the characteristics of an algorithm.
  - 1.3 How are algorithms represented? Explain with the help of an example.
  - 1.4 What is algorithm complexity?
  - 1.5 What is an asymptotic notation? Explain the various types of asymptotic notations.
  - 1.6 What is a data structure? What are its various characteristics?
  - 1.7 What are the different types of data structures?

- 1.8** Explain any two non-primitive data structures.  
**1.9** What is a tree? Why is it used?  
**1.10** List the typical operations associated with derived data structure types.

### Answers to Multiple-Choice Questions

- |                 |         |         |         |          |
|-----------------|---------|---------|---------|----------|
| 1.1 (c)         | 1.2 (a) | 1.3 (b) | 1.4 (c) | 1.5 (a)  |
| 1.6 (a) and (c) | 1.7 (d) | 1.8 (a) | 1.9 (d) | 1.10 (d) |

# 2

# ARRAYS

Chapter outline

- 2.1 Introduction
- 2.2 Types of Arrays
- 2.3 Representation of One-Dimensional Array in Memory
- 2.4 Array Traversal
- 2.5 Insertion and Deletion
  - 2.5.1 Insertion
  - 2.5.2 Deletion
- 2.6 Sorting and Searching
  - 2.6.1 Sorting
  - 2.6.2 Searching
- 2.7 Representation of Multi-Dimensional Array in Memory
- 2.8 Realizing Matrices Using Two-Dimensional Arrays
- 2.9 Matrix Operations
  - 2.9.1 Addition
  - 2.9.2 Subtraction
  - 2.9.3 Multiplication
  - 2.9.4 Transpose

*Solved Problems*

*Summary*

*Key Terms*

*Multiple-Choice Questions*

*Review Questions*

*Programming Exercises*

*Answers to Multiple-Choice Questions*



## 2.1 INTRODUCTION

In Chapter 2, we briefly introduced one of the most important and commonly used derived data types, called *array*. In this chapter, we will observe how array is used as a data structure in different programming situations. We will also get familiar with the logical representation of arrays in memory.

An array is regarded as one of the most fundamental entities for storing logical groups of data. It also forms the basis for implementing some advanced data structures, like stacks and queues, as we shall see in the later chapters.

Typically, an array is defined as a collection of same type elements. That means, it can store a group of integers, characters, strings, structures, and so on. However, an array cannot store different type elements like a group of integers and fractions, or a group of strings and integers. Following are some of the characteristics associated with arrays:

1. It uses a single name for referencing all the array elements. The differentiation between any two elements is made on the basis of index value.
2. It stores the different elements at consecutive memory locations.
3. Its name can be used as a pointer to the first array element.
4. Its size is always a constant expression and not a variable.
5. It does not perform bound checking on its own. It has to be implemented programmatically.

Before we delve further into exploring array as a data structure, let us first identify the different types of arrays.

## 2.2 TYPES OF ARRAYS

As already explained, an array is a logical grouping of same type data elements. Now, it is quite possible that each of these elements is itself an array. Further, each of the elements of the sub array could also be an array. This forms the basis of characterizing an array into different types, as depicted in Table 2.1.

**Table 2.1** *Types of arrays*

Array Type	Description	C Representation	Example
One-dimensional array	It is a group of same type data elements, such as integers, floats, or characters.	array[ ]	A group of integers. {2, 5, 6, 1, 9}
Multi-dimensional array	It is a group of data elements, where each element is itself an array.	array[ ][ ].[]	A group of strings. {"Ajay", "Vikas", "Amit", "Sumit"}

The various instances of multi-dimensional arrays are two-dimensional (2D) array, three-dimensional (3D) array, four-dimensional (4D) array, and so on. The choice of a particular multi-dimensional array depends on the programming situation at hand. For instance, if we are required to realize a  $3 \times 3$  matrix programmatically, then we can do so by declaring a two dimensional array, say  $M[3][3]$ . Here, each dimension of the array  $M$  signifies the row and column of the matrix respectively. Multi-dimensional arrays are covered in greater detail later in this chapter. For now, let us focus on implementing and manipulating one-dimensional array.

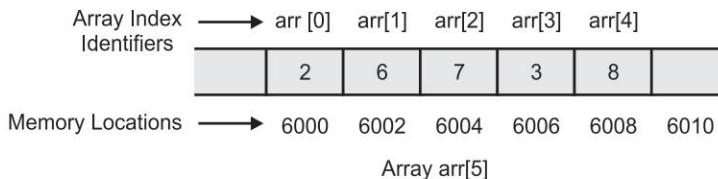
## 2.3 REPRESENTATION OF ONE-DIMENSIONAL ARRAY IN MEMORY

The elements of a one-dimensional array are stored at consecutive locations in memory. Each of the locations is accessed with the help of array index identifier to retrieve the corresponding element.

Consider the following integer array:

`arr[5] = {2, 6, 7, 3, 8}`

Here, `arr` is a five-element integer array. Figure 2.1 shows the representation of array `arr` in memory:



**Fig. 2.1** Array representation in memory

As shown in Fig. 2.1, each array element is stored at consecutive memory locations, i.e., 6000, 6002, 6004, and so on. The location of the first element, i.e., 6000 is also referred as the base address of the array.

If we know the base address of an array, then we can find the location of its individual elements by using a simple formula, which is

$$\text{Address of } A[k] = B + W * k$$

Here,

`A[ ]` is the array.

`B` is the base address, i.e., the address of the first element.

`W` is the word size or the size of an array element.

`k` is the index identifier.

For instance, the address of the third element of array `arr` stored at index location 2 would be

$$\begin{aligned}\text{Address of } \text{arr}[2] &= 6000 + 2 * 2 \\ &= 6000 + 4 \\ &= 6004\end{aligned}$$



**Note** The word size of a data type is decided by the programming language being used and the hardware specifications.

## 2.4 ARRAY TRAVERSAL

While working with arrays, it is often required to access the array elements; that is, reading values from the array. This is achieved with the help of array traversal. It involves visiting the array elements and storing or retrieving values from it. Some of the typical situations where array traversal may be required are:



### Check Point

#### 1. What is a base address?

**Ans.** It is the memory address of the first element of an array.

#### 2. How are array elements stored in memory?

**Ans.** The elements of an array are stored at consecutive locations in memory.

Printing array elements,  
Searching an element in the array,  
Sorting an array, and so on

### Algorithm

**Example 2.1** Write an algorithm to sequentially traverse an array.

```
traverse(arr[], size)
Step 1: Start
Step 2: Set i = 0
Step 3: Repeat Steps 4-5 while i < size
Step 4: Access arr[i]
Step 5: Set i = i + 1
Step 6: Stop
```

### Program

**Example 2.2** Write a C program to traverse each element of an array and print its value on the console.

Program 2.1 performs array traversal and prints the array elements as output. It uses the algorithm depicted in Example 2.1.

**Program 2.1** C program to traverse each element of an array and print its value

```
#include <stdio.h>
#include <conio.h>

void traverse(int*, int); /*Function prototype for array traversal*/
void main()
{
    int arr[5] = {2, 6, 7, 3, 8}; ←
    int N=5;
    clrscr();

    printf("Press any key to perform array traversal and display its elements:
\n\n");
    getch();

    traverse(arr,N); /*Calling traverse function*/

    getch();
}

void traverse(int *array, int size)
{
    int i;
    for(i=0;i<size;i++)
        printf("arr[%d] = %d\n", i, array[i]); /*Accessing array element and
printing it*/
}
```

*Specifying array values at the time  
of writing a program is referred as  
compile-time initialization.*

## Output

Press any key to perform array traversal and display its elements:

```
arr[0] = 2  
arr[1] = 6  
arr[2] = 7  
arr[3] = 3  
arr[4] = 8
```

## Program analysis

Key Statement	Purpose
<b>void traverse(int*, int);</b>	Declares the prototype for the <i>traverse()</i> function for traversing an array
<b>traverse(arr,N);</b>	Calls the <i>traverse()</i> function for traversing the array <i>arr</i> containing <i>N</i> elements
<b>for(i=0;i&lt;size;i++)</b>	Uses the <b>for</b> loop to access the array elements with each iteration



**Tip** While traversing an array, the index identifier should be updated carefully so that array out of bound situation does not arise. In this situation, the program tries to access a location outside of the reserved memory block, which is an illegal operation.

## 2.5 INSERTION AND DELETION

Insertion is the task of adding an element into an existing array while deletion is the task of removing an element from the array. The point of insertion or deletion that is the position where an element is to be inserted or deleted holds vital importance, as we shall see in the subsequent sections.

### 2.5.1 Insertion

If an element is to be inserted at the end of the array, then it can be simply achieved by storing the new element one position to the right of the last element. However, the array must have vacant positions at the end for this to be feasible. Alternatively, if an element is required to be inserted at the middle, then this will require all the subsequent elements to be moved one place to the right. Figure 2.2 depicts the insertion of an element into an array.



### Check Point

#### 1. What is array traversal?

**Ans.** It is the task of visiting the array elements and storing or retrieving values from it.

#### 2. What is the need for array traversal?

**Ans.** It is required in almost all array related operations, such as sorting, searching, printing, etc.

	-1	3	5	22	77			
--	----	---	---	----	----	--	--	--

A [0] A [1] A [2] A [3] A [4] A [5] A [6]

**Initial Array A**

	-1	3	5	22	77	4		
--	----	---	---	----	----	---	--	--

A [0] A [1] A [2] A [3] A [4] A [5] A [6]

**Array contents after element 4 is inserted at the end**

	-1	3	4	5	22	77		
--	----	---	---	---	----	----	--	--

A [0] A [1] A [2] A [3] A [4] A [5] A [6]

**Array contents after element 4 is inserted at index location 2**

**Fig. 2.2 Array insertion**

## Algorithm

**Example 2.3** Write an algorithm to perform array insertion.

The following algorithm inserts an element P at index location k in the array A[N], where k<=N.

```
insert(A[N], k, P)
Step 1: Start
Step 2: Set i = N
Step 3: Repeat Steps 4-5 while i >=k
Step 4: Set A[i+1] = A[i]
Step 5: Set i = i - 1
Step 6: Set A[k] = P
Step 7: Set N = N + 1
Step 8: Stop
```

## Program

**Example 2.4** Write a C program to perform array insertion.

Program 2.2 performs array insertion and prints the updated array elements as output. It uses the algorithm depicted in Example 2.3.

**Program 2.2 Array insertion**

```
#include <stdio.h>
#include <conio.h>

void main()
{
    int array[10] = {-1, 3, 5, 22, 77};
```

```

int i, k, N, P;
clrscr();
N = 5;

printf("The contents of the array are:\n");
for(i=0;i<N;i++)
    printf("array[%d] = %d\n",i,array[i]); /*Printing array elements*/

printf("\nEnter the element to be inserted:\n");
scanf("%d",&P);

printf("\nEnter the index location where %d is to be inserted:\n", P);
scanf("%d",&k);

for(i=N;i>=k;i--)
    array[i+1]=array[i];
array[k] = P;
N = N + 1;

printf("\nThe contents of the array after array insertion are:\n");
for(i=0;i<N;i++)
    printf("array[%d] = %d\n",i,array[i]); /*Printing array elements after
array insertion*/
getch();
}

```

*The existing array elements need  
to be moved to make space for the  
new element.*

## Output

```

The contents of the array are:
array[0] = -1
array[1] = 3
array[2] = 5
array[3] = 22
array[4] = 77

Enter the element to be inserted:
19

Enter the index location where 19 is to be inserted:
1

The contents of the array after array insertion are:
array[0] = -1
array[1] = 19
array[2] = 3
array[3] = 5
array[4] = 22
array[5] = 77

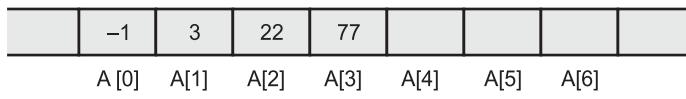
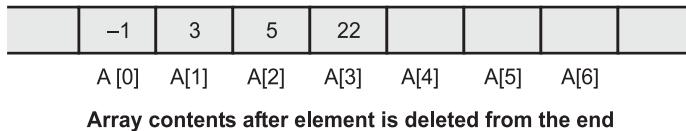
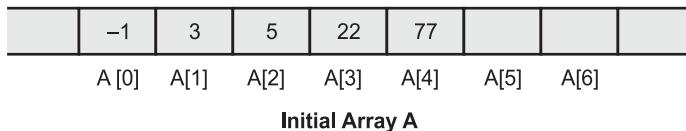
```

## Program analysis

Key Statement	Purpose
<b>for(i=N;i&gt;=k;i--)</b> <b>array[i+1]=array[i];</b>	Shuffles the array elements to the right to create space for inserting a new element.
<b>array[k] = P;</b>	Inserts a new element at the point of insertion.
<b>N = N + 1;</b>	Increments the total number of array elements by 1.

## 2.5.2 Deletion

The deletion of elements follows a similar procedure as insertion. The deletion of element from the end is quite simple and can be achieved by mere updation of index identifier. However, to remove an element from the middle, one must move all the elements present to the right of the point of deletion, one position to the left. Figure 2.3 depicts the deletion of an element from an array.



**Array contents after an element is deleted from index location 2**

**Fig. 2.3 Array deletion**

## Algorithm

**Example 2.5** Write an algorithm to perform array deletion.

The following algorithm deletes the element at index location  $k$  in the array  $A[N]$ , where  $k \leq N$ .

```
delete (A[N], k)
Step 1: Start
Step 2: Set D = A[k]
Step 3: Set i = k
Step 4: Repeat Steps 5-6 while i <= N-1
Step 5: Set A[i] = A[i+1]
Step 6: Set i = i + 1
```

Step 7: Set N = N - 1  
Step 8: Stop

## Program

**Example 2.6** Write a C program to perform array deletion.

Program 2.3 performs array deletion and prints the updated array elements as output. It uses the algorithm depicted in Example 2.5.

### Program 2.3 Array deletion

```
#include <stdio.h>
#include <conio.h>

void main()
{
    int array[10] = {-1, 3, 5, 22, 77};
    int i, k, N, D;
    clrscr();

    N = 5;

    printf("The contents of the array are:\n");
    for(i=0;i<N;i++)
        printf("array[%d] = %d\n",i,array[i]); /*Printing array elements*/

    printf("\nEnter the index location from where element is to be deleted:\n");
    scanf("%d",&k);

    D = array[k];

    for(i=k;i<=N-2;i++)
        array[i]=array[i+1];
    N = N - 1;

    printf("\n%d element deleted from index location %d\n",D,k);

    printf("\nThe contents of the array after array deletion are:\n");
    for(i=0;i<N;i++)
        printf("array[%d] = %d\n",i,array[i]); /*Printing array elements after
array deletion*/

    getch();
}
```

The existing array elements need to be moved to fill the vacant space created by the deleted element.

## Output

```
The contents of the array are:
```

```
array[0] = -1  
array[1] = 3  
array[2] = 5  
array[3] = 22  
array[4] = 77
```

```
Enter the index location from where element is to be deleted:
```

```
3
```

```
22 element deleted from index location 3
```

```
The contents of the array after array deletion are:
```

```
array[0] = -1  
array[1] = 3  
array[2] = 5  
array[3] = 77
```

## Program analysis

Key Statement	Purpose
<b>D = array[k];</b>	Retrieves the element value that is to be deleted.
<b>for(i=k;i&lt;=N-2;i++) array[i]=array[i+1];</b>	Shuffles the array elements to the left to fill the vacant space created after deleting the array element.
<b>N = N - 1;</b>	Decrement the total number of array elements by 1.



**Note** For large-sized arrays, inserting an element at the middle could be a considerable programming overhead as it would require the other elements to be moved from their current positions.

## 2.6 SORTING AND SEARCHING

Sorting and searching are two of the most common operations performed on arrays. The sorting operation arranges the elements of an array in a specific order or sequence. Searching, on the other hand, locates a specific element in the array.

### 2.6.1 Sorting

Sorting involves comparing the array elements with each other and shuffling them until all the elements are sorted. There are a



### Check Point

#### 1. What is array insertion and deletion?

**Ans.** The task of adding an element into an existing array is called array insertion while the task of deleting an existing element from the array is called array deletion.

#### 2. What is a point of insertion?

**Ans.** It is the location in the array where a new element is to be inserted.

number of sorting techniques that are applied to sort an array in an efficient manner. We shall explore these sorting techniques in Chapter 10. Here, let us focus on one of the most basic sorting techniques called *bubble sort*.

Bubble sort operates on an array in such a manner that the largest element is brought to the end of the array with each successive iteration. It repeatedly compares two consecutive elements and moves the largest amongst them to the right. This process is repeated for each pair of elements until the current iteration moves the largest element to the end.

Consider the following integer array:

```
arr[5] = {5, 4, 3, 2, 1}
```

Here, **arr** is a five-element integer array. It will take four iterations or passes to sort this five-element array. Each pass will bring the largest element to the end of the array. Here's a snapshot of the array contents after each of the four passes:

Initial Array -	arr[5] = {5, 4, 3, 2, 1}
Pass 1	- arr[5] = {4, 3, 2, 1, 5}
Pass 2	- arr[5] = {3, 2, 1, 4, 5}
Pass 3	- arr[5] = {2, 1, 3, 4, 5}
Pass 4	- arr[5] = {1, 2, 3, 4, 5}

As shown above, the fourth pass produces the sorted array.

## Algorithm

Refer to Section 10.2.3 for the algorithm on applying bubble sorting technique to sort an array.

## Program

**Example 2.7** Write a C program to sort an array of five elements.

Program 2.4 implements bubble sorting technique to sort an array of five elements.

### Program 2.4 Bubble sorting technique

```
#include <stdio.h>
#include <conio.h>

void main()
{
    int array[5] = {5, 4, 3, 2, 1};
    int i, k, j, temp;
    clrscr();

    printf("\nThe initial array elements are:\n");
    for(i=0;i<5;i++)
        printf("array[%d] = %d\n", i, array[i]); /*Printing initial array*/

    for(i=5;i>1;i--) /*Outer loop for controlling the number of passes*/
        for(j=0;j<i-1;j++) /*Inner loop for controlling the number of comparisons
made in a pass*/
            if (array[j]>array[j+1])
                {
                    temp = array[j];
                    array[j] = array[j+1];
                    array[j+1] = temp;
                }
}
```

```

{
    /*Swapping adjacent elements*/
    temp = array[j+1];
    array[j+1] = array[j];
    array[j] = temp;
}

printf("\nThe sorted elements are:\n");
for(i=0;i<5;i++)
    printf("array[%d] = %d\n",i,array[i]); /*Printing sorted array*/

getch();
}

```

If the swap operation moves the larger element towards the right then the array is sorted in ascending order; otherwise it is sorted in descending order.

## Output

The initial array elements are:

```

array[0] = 5
array[1] = 4
array[2] = 3
array[3] = 2
array[4] = 1

```

The sorted elements are:

```

array[0] = 1
array[1] = 2
array[2] = 3
array[3] = 4
array[4] = 5

```

## Program analysis

Key Statement	Purpose
<b>for(i=5;i&gt;1;i--)</b>	Uses <i>for</i> loop to control the number of passes of bubble sort algorithm
<b>for(j=0;j&lt;i-1;j++)</b>	Uses <i>for</i> loop to compare the array elements in each pass of bubble sort algorithm
<b>temp = array[j+1];</b>	
<b>array[j+1] = array[j];</b>	
<b>array[j] = temp;</b>	Swaps two array elements



### Note

Just like an integer array, sorting can also be applied to an array of floats, characters, structures, and so on.

## 2.6.2 Searching

Searching is the process of traversing an array to find out if a specific element is present in the array or not. If the search is successful, the index location of the element is returned. There are various searching mechanisms that can be employed to search an element in an array. We shall explore these searching techniques in Chapter 10. Here, let us focus on one of the most basic searching techniques called *linear search*.

The linear search technique traverses an array sequentially to search the desired element. It starts the search with the first element and moves towards the end in a step-by-step fashion. At each step, it matches the element to be searched with the array element, and if there is a match, the location of the array element is returned.

Consider the following integer array:

```
arr[5] = {22, 19, 4, 8, 10}
```

Here, **arr** is a five element integer array. If we apply linear search to the array arr to search element 4, then the search will be successful as element 4 is present in the array. The search module will return index location 2 as the search result because element 4 is the third element in the array.

### Algorithm

Refer to Section 10.3.1 for the algorithm on applying linear search on an array.

### Program

**Example 2.8** Write a C program to perform linear search on an array.

Program 2.5 applies linear searching technique on an array of five elements.

#### Program 2.5 *Performing linear search on an array*

```
#include <stdio.h>
#include <conio.h>

void main()
{
    int array[5] = {22, 19, 4, 8, 10};
    int i, flag, k, index;
    clrscr();

    flag = 0;

    printf("The contents of the array are:\n");
    for(i=0;i<5;i++)
        printf("array[%d] = %d\n", i, array[i]); /*Printing array elements*/
```



### Check Point

#### 1. What is sorting?

**Ans.** It is the task of arranging the elements of an array in a sequence.

#### 2. How many passes does bubble sorting technique require to sort an array of $n$ elements?

**Ans.**  $n-1$ .

```

printf("\nEnter the element to be searched:\n");
scanf("%d",&k);

for(i=0;i<5;i++) /*Scanning array elements one by one*/
{
    if(k==array[i])
    {
        flag = 1; /*Successful Search*/
        index = i;
        break;
    }
    else
    ;

if(flag==1)
    printf("Search is successful! Element %d is present at index location
%d in the array",k,index);
else /*Successful Search*/
    printf("Unsuccessful Search!");

getch();
}

```

*The flag variable is updated to signal  
successful search.*

## Output

The contents of the array are:

```

array[0] = 22
array[1] = 19
array[2] = 4
array[3] = 8
array[4] = 10

```

Enter the element to be searched:

4

Search is successful! Element 4 is present at index location 2 in the array

## Program analysis

Key Statement	Purpose
<b>scanf("%d",&amp;k);</b>	Reads the key value that needs to be searched in the array.
<b>if(k==array[i])</b>	Compares the key value with each array element.
<b>break;</b>	Takes the program control out of the <b>for</b> loop as soon as the search is successful.

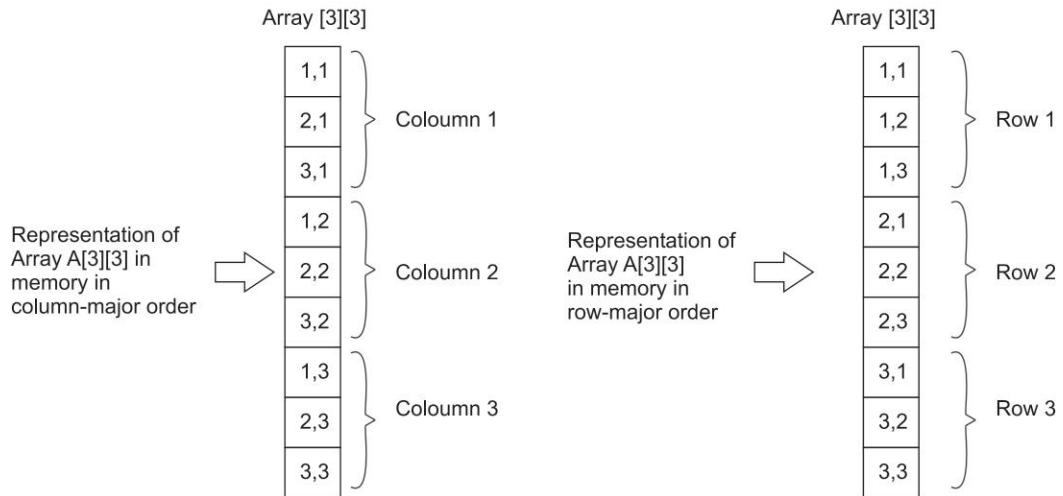


**Note** Just like an integer array, searching can also be performed on an array of floats, characters, structures, and so on.

## 2.7 REPRESENTATION OF MULTI-DIMENSIONAL ARRAY IN MEMORY

Let us recall that a multi-dimensional array is an array of arrays. Unlike one-dimensional arrays which have only one subscript, a multidimensional array has multiple subscripts. For example, a two-dimensional array, one of the most widely used instances of multi-dimensional arrays, has two subscripts. It is used to programmatically realize a matrix with its first subscript representing the row and the second subscript representing the column of a matrix.

The representation of a two-dimensional array in memory is not like the grid-like structure of a matrix. Instead, it is same as the one-dimensional array representation in memory. It either stores the array elements row by row (*row major order*) or column by column (*column major order*). Figure 2.4 illustrates these representations:



**Fig. 2.4** Representation of two-dimensional array in memory

As shown in Fig. 2.4, the elements of a two-dimensional array are stored at consecutive memory locations. The only difference is in the order in which these elements are stored in memory. In column-major order, the elements are stored column-by-column while in row-major order the elements are stored row-by-row. Both these memory representations are intrinsic to a programming language and the programmer does not have a choice of selecting a particular representation format for storing array elements.



### Check Point

#### 1. What is searching?

**Ans.** Searching is the process of determining if a specific element is present in the array or not.

#### 2. At a maximum, how many elements would the searching technique require to traverse in an $n$ -element array?

**Ans.**  $n$

The formula for computing the address location of a multi-dimensional array element in row major implementation is given below:

$$\text{Address of } A[i, j] = B + W (n (i - LBR) + (j - LBC))$$

Here,

1. **A[ ][ ]** is the multidimensional array.
2. **B** is the base address.
3. **W** is the word size or the size of an array element.
4. **n** is the number of columns.
5. **i, j** are the index identifiers.
6. **LBR** is the lower bound of row index.
7. **LBC** is the lower bound of column index.

Similarly, the formula for computing the address location of a multi-dimensional array element in column major implementation is given below:

$$\text{Address of } A[i, j] = B + W (m (j - LBC) + (i - LBR))$$

Here, **m** represents the number of rows.

**Example 2.9** A  $10 \times 12$  matrix is implemented using array  $A[10][12]$ . If the base address of the array is 200 and the word size is 2 then compute the address of the element  $A[4, 7]$  in:

- (a) Row major order
- (b) Column major order

Assume that the lower bound of both row and column indices is 1.

**Solution** (a) Row major order

$$\text{Address of } A[i, j] = B + W (n (i - LBR) + (j - LBC))$$

$$\begin{aligned}\text{Address of } A[4, 7] &= 200 + 2 (12 (4 - 1) + (7 - 1)) \\ &= 200 + 2 (42) \\ &= 284\end{aligned}$$

(b) Column major order

$$\text{Address of } A[i, j] = B + W (m (j - LBC) + (i - LBR))$$

$$\begin{aligned}\text{Address of } A[4, 7] &= 200 + 2 (10 (7 - 1) + (4 - 1)) \\ &= 200 + 2 (63) \\ &= 326\end{aligned}$$

## 2.8 REALIZING MATRICES USING TWO-DIMENSIONAL ARRAYS

Two-dimensional arrays are most commonly used for realizing matrices. The first subscript signifies the rows of a matrix while the second subscript signifies the columns. Operation on these array-represented matrices can be performed through simple programming.

Figure 2.5 depicts the realization of a matrix through a two-dimensional array:



### Check Point

#### 1. What is row-major order?

**Ans.** It is the memory representation of a two-dimensional array in row-by-row fashion.

#### 2. What is column-major order?

**Ans.** It is the memory representation of a two-dimensional array in column-by-column fashion.



### Mind Jog

#### What is a square matrix?

*It is the matrix that has equal number of rows and columns.*

$$M[3][4] \rightarrow \left\{ \begin{array}{cccc} 0,0 & 0,1 & 0,2 & 0,3 \\ 1,0 & 1,1 & 1,2 & 1,3 \\ 2,0 & 2,1 & 2,2 & 2,3 \end{array} \right\}$$

**Fig. 2.5** Matrix represented by two-dimensional array

Figure 2.5 shows the subscript values for each of the elements of the matrix M[3][4].

### Program

**Example 2.10** Write a C program to realize a  $3 \times 3$  matrix.

Program 2.6 realizes a  $3 \times 3$  matrix using a two-dimensional array.

**Program 2.6**  $3 \times 3$  matrix using two-dimensional array

```
#include <stdio.h>
#include <conio.h>

void main()
{
    int i,j,M[3][3];
    clrscr();

    /*Reading matrix elements*/
    printf("Enter the elements of the 3 x 3 matrix:\n");
    for(i=0;i<3;i++)
        for(j=0;j<3;j++)
    {
        printf("M[%d] [%d] = ",i,j);
        scanf("%d",&M[i][j]);
    }

    /*Printing matrix elements*/
    printf("The matrix represented by the 3 x 3 2D array is:\n");
    for(i=0;i<3;i++)
    {
        printf("\n\t\t");
        for(j=0;j<3;j++)
            printf("%d\t",M[i][j]);
    }
    getch();
}
```

The indentation and display of the two-dimensional array elements is done in such a manner so to represent a real matrix.

### Output

```
Enter the elements of the 3 x 3 matrix:
M[0][0] = 1
```

```

M[0][1] = 2
M[0][2] = 3
M[1][0] = 4
M[1][1] = 5
M[1][2] = 6
M[2][0] = 7
M[2][1] = 8
M[2][2] = 9

```

The matrix represented by the  $3 \times 3$  2D array is:

1	2	3
4	5	6
7	8	9

### Program analysis

Key Statement	Purpose
<b>int i,j,M[3][3];</b>	Declares a two-dimensional array to represent a $3 \times 3$ matrix
<b>for(i=0;i&lt;3;i++)</b>	Uses <i>for</i> loop to iterate through each row of the matrix
<b>for(j=0;j&lt;3;j++)</b>	Uses <i>for</i> loop to iterate through each column of the matrix
<b>scanf("%d",&amp;M[i][j]);</b>	Reads the matrix elements

## 2.9 MATRIX OPERATIONS

The various operations associated with matrices are:

1. Addition
2. Subtraction
3. Multiplication
4. Transpose

### 2.9.1 Addition

Addition is the task of adding individual elements of two matrices. For instance,

If matrix A =	a1      a2      a3
	a4      a5      a6
	a7      a8      a9
And, matrix B =	b1      b2      b3
	b4      b5      b6
	b7      b8      b9
Then, A + B =	a1+b1    a2+b2    a3+b3
	a4+b4    a5+b5    a6+b6
	a7+b7    a8+b8    a9+b9

## Program

**Example 2.11** Write a C program to perform addition on two  $3 \times 3$  matrices.

**Program 2.7 Adding on two  $3 \times 3$  matrices**

```
#include <stdio.h>
#include <conio.h>

void main()
{
    int i,j,A[3][3],B[3][3],C[3][3];
    clrscr();

    printf("Enter the elements of 3 x 3 matrix A:\n");
    for(i=0;i<3;i++)
    {
        for(j=0;j<3;j++)
        {
            printf("A[%d] [%d] = ",i,j);
            scanf("%d",&A[i][j]);/*Reading the elements of 1st matrix*/
        }
    }

    printf("Enter the elements of 3 x 3 matrix B:\n");
    for(i=0;i<3;i++)
    {
        for(j=0;j<3;j++)
        {
            printf("B[%d] [%d] = ",i,j);
            scanf("%d",&B[i][j]);/*Reading the elements of 2nd matrix*/
        }
    }

    printf("\nThe entered matrices are: \n");
    for(i=0;i<3;i++)
    {
        printf("\n");
        for(j=0;j<3;j++)
            printf("%d    ,A[i][j]);/*Displaying the elements of matrix A*/
        printf("\t\t");
        for(j=0;j<3;j++)
            printf(" %d    ,B[i][j]);/*Displaying the elements of matrix B*/
    }

    for(i=0;i<3;i++)
        for(j=0;j<3;j++)
            C[i][j] = A[i][j]+B[i][j];/*Computing the sum of two matrices*/

    printf("\n\nSum of A and B is shown below: \n");
}
```

```

for(i=0;i<3;i++)
{
    printf("\n");
    for(j=0;j<3;j++)
        printf("%d   ",C[i][j]);/*Displaying the result*/
}

getch();
}

```

## Output

Enter the elements of  $3 \times 3$  matrix A:

```

A[0][0] = 1
A[0][1] = 1
A[0][2] = 1
A[1][0] = 1
A[1][1] = 1
A[1][2] = 1
A[2][0] = 1
A[2][1] = 1
A[2][2] = 1

```

Enter the elements of  $3 \times 3$  matrix B:

```

B[0][0] = 2
B[0][1] = 2
B[0][2] = 2
B[1][0] = 2
B[1][1] = 2
B[1][2] = 2
B[2][0] = 2
B[2][1] = 2
B[2][2] = 2

```

The entered matrices are:

1 1 1	2 2 2
1 1 1	2 2 2
1 1 1	2 2 2

Sum of A and B is shown below:

```

3 3 3
3 3 3
3 3 3

```

## Program analysis

Key Statement	Purpose
$C[i][j] = A[i][j] + B[i][j];$	Adds the elements of A and B matrices and stores the result at corresponding positions of the resultant matrix C

## 2.9.2 Subtraction

Subtraction is the task of subtracting individual elements of two matrices. For instance,

If matrix A = 
$$\begin{array}{ccc} a1 & a2 & a3 \\ a4 & a5 & a6 \\ a7 & a8 & a9 \end{array}$$

And, matrix B = 
$$\begin{array}{ccc} b1 & b2 & b3 \\ b4 & b5 & b6 \\ b7 & b8 & b9 \end{array}$$

Then,  $A - B = \begin{array}{ccc} a1-b1 & a2-b2 & a3-b3 \\ a4-b4 & a5-b5 & a6-b6 \\ a7-b7 & a8-b8 & a9-b9 \end{array}$

A C program to perform matrix subtraction will be same as matrix addition (see Example 2.11). We just need to replace the + sign with a - sign.

## 2.9.3 Multiplication

Matrix multiplication is not as simple as matrix addition or subtraction. It uses a certain formula to generate multiplication result. For instance,

If matrix A = 
$$\begin{array}{ccc} a1 & a2 & a3 \\ a4 & a5 & a6 \\ a7 & a8 & a9 \end{array}$$

And, matrix B = 
$$\begin{array}{ccc} b1 & b2 & b3 \\ b4 & b5 & b6 \\ b7 & b8 & b9 \end{array}$$

Then,  $A \times B = \begin{array}{ccc} a1b1+a2b4+a3b7 & a1b2+a2b5+a3b8 & a1b3+a2b6+a3b9 \\ a4b1+a5b4+a6b7 & a4b2+a5b5+a6b8 & a4b3+a5b6+a6b9 \\ a7b1+a8b4+a9b7 & a7b2+a8b5+a9b8 & a7b3+a8b6+a9b9 \end{array}$

For two non-square matrices, multiplication is feasible only if the number of columns in the left matrix is equal to the number of rows in the right matrix. Thus, if a  $M \times N$  matrix is multiplied with a  $N \times P$  matrix, then the resultant matrix would be a  $M \times P$  matrix.

### Program

**Example 2.12** Write a C program to perform multiplication on two  $3 \times 3$  matrices.

**Program 2.8** *Multiplying on two  $3 \times 3$  matrices*

```
#include <stdio.h>
#include <conio.h>
```

```

void main()
{
    int i,j,k,A[3][3],B[3][3],C[3][3];
    clrscr();
    printf("Enter the 3 x 3 matrix A:\n");
    for(i=0;i<3;i++)
    {
        for(j=0;j<3;j++)
        {
            printf("A[%d] [%d] = ",i,j);
            scanf("%d",&A[i][j]);/*Reading the elements of the 1st matrix*/
        }
    }

    printf("Enter the 3 x 3 matrix B:\n");
    for(i=0;i<3;i++)
    {
        for(j=0;j<3;j++)
        {
            printf("B[%d] [%d] = ",i,j);
            scanf("%d",&B[i][j]);/*Reading the elements of the 2nd matrix*/
        }
    }
    printf("\nThe entered matrices are: \n");

    for(i=0;i<3;i++)
    {
        printf("\n");
        for(j=0;j<3;j++)
        {
            printf("%d\t",A[i][j]);/*Displaying the elements of matrix A*/
        }
        printf(" \t \t");
        for(j=0;j<3;j++)
        {
            printf("%d\t",B[i][j]);/*Displaying the elements of the matrix B*/
        }
    }
    /*Multiplying the two matrices*/
    for(i=0;i<3;i++)
        for(j=0;j<3;j++)
    {
        C[i][j]=0;
        for(k=0;k<3;k++)
            C[i][j]=C[i][j]+A[i][k]*B[k][j];
    }
    printf("\n\nThe product of the two matrices A x B is shown below: \n");
}

```

```

for(i=0;i<3;i++)
{
    printf("\n");
    for(j=0;j<3;j++)
    {
        printf("%d\t",C[i][j]); /*Displaying the result*/
    }
}

getch();
}

```

## Output

```

Enter the elements of 3 × 3 matrix A:
A[0][0] = 1
A[0][1] = 2
A[0][2] = 3
A[1][0] = 4
A[1][1] = 5
A[1][2] = 6
A[2][0] = 7
A[2][1] = 8
A[2][2] = 9
Enter the elements of 3 × 3 matrix B:
B[0][0] = 9
B[0][1] = 8
B[0][2] = 7
B[1][0] = 6
B[1][1] = 5
B[1][2] = 4
B[2][0] = 3
B[2][1] = 2
B[2][2] = 1

```

The entered matrices are:

1	2	3		9	8	7
4	5	6		6	5	4
7	8	9		3	2	1

The product of the two matrices A × B is shown below:

30	24	18
84	69	54
138	114	90

## Program analysis

Key Statement	Purpose
$C[i][j] = C[i][j] + A[i][k] * B[k][j];$	Multiplies the elements of $A$ and $B$ matrices and stores the result at corresponding positions of the resultant matrix $C$

## 2.9.4 Transpose

In simple words, transposing a matrix refers to the task of changing the rows into columns and columns into rows. For instance,

If matrix A =	a1      a2      a3
	a4      a5      a6
	a7      a8      a9
	a1      a4      a7
Then, transpose(A) =	a2      a5      a8
	a3      a6      a9

## Program

**Example 2.13** Write a C program to transpose a given  $3 \times 3$  matrix.

**Program 2.9** C program to transpose  $3 \times 3$  matrix

```
#include <stdio.h>
#include <conio.h>

void main()
{
    int i,j,A[3][3],T[3][3];
    clrscr();
    printf("Enter a 3 x 3 matrix:\n");
    for(i=0;i<3;i++)
    {
        for(j=0;j<3;j++)
        {
            printf("A[%d] [%d] = ",i,j);
            scanf("%d",&A[i][j]); /*Reading the elements of the 3X3 matrix*/
        }
    }

    printf("\nThe entered matrix is: \n");
    for(i=0;i<3;i++)
    {
        printf("\n");
        for(j=0;j<3;j++)
        {
            printf("%d\t",A[i][j]); /*Displaying the matrix*/
        }
    }
}
```

```

for(i=0;i<3;i++)
{
    for(j=0;j<3;j++)
        T[i][j]=A[j][i]; /*Computing matrix transpose*/
}
printf("\n\nThe transpose of the matrix is: \n");

for(i=0;i<3;i++)
{
    printf("\n");
    for(j=0;j<3;j++)
    {
        printf("%d\t",T[i][j]); /*Displaying the resultant transposed matrix*/
    }
}

getch();
}

```

## Output

Enter a  $3 \times 3$  matrix:

```

A[0][0] = 1
A[0][1] = 2
A[0][2] = 3
A[1][0] = 4
A[1][1] = 5
A[1][2] = 6
A[2][0] = 7
A[2][1] = 8
A[2][2] = 9

```

The entered matrix is:

```

1      2      3
4      5      6
7      8      9

```

The transpose of the matrix is:

```

1      4      7
2      5      8
3      6      9

```

## Program analysis

Key Statement	Purpose
T[i][j]=A[j][i];	Transposes each element of the matrix $A$ and stores the result in the matrix $T$

**Note**

*Adequate checks must be included in a program to ensure that non-compatible matrices are not operated.*

**Check Point****1. What is matrix addition?**

**Ans.** It is the task of adding the relative elements of two matrices.

**2. What is matrix transpose?**

**Ans.** It is the task of changing the rows into columns and columns into rows.

## Solved Problems

**Problem 2.1** Consider the following array of integers:

35 18 7 12 5 23 16 3 1

Create a snapshot of the above array for the following operations:

Inserting element 99 at index location 2.

Deleting the first element of the array.

**Solution**

Array contents after insertion: 35 18 99 7 12 5 23 16 3 1

Array contents after deletion: 18 7 12 5 23 16 3 1

**Problem 2.2** Consider the following array of integers:

74 39 35 32 97 84

Create a snapshot of the above array after the sorting operation is performed on it.

**Solution**

*Initial array*                    74 39 35 32 97 84

*Sorted array*                    32 35 39 74 84 97

**Problem 2.3** Consider the following array of integers:

74 39 35 32 97 84

How many elements would need to be traversed before search operation is completed on the following items:

32

83

**Solution**

4

6

**Problem 2.4** Consider the following array of integers::

35 54 12 18 23 15 45 38

Deduce the address of the 4<sup>th</sup> element (index location 3), if the base address is 3000. Assume that the word size is 2.

$$\begin{aligned}\text{Solution Address of arr[3]} &= 3000 + 2 * 3 \\ &= 3000 + 6 \\ &= 3006\end{aligned}$$

**Problem 2.5** A two-dimensional array A[5][10] is implemented in row order manner in the memory. Deduce the address of the A[3][5] element, if the base address of the array is 3000 and the word size is 2. Assume the lower bound of row and column indices to be 1.

$$\begin{aligned}\text{Solution Address of } A[i,j] &= B + W(n(i - LBR) + (j - LBC)) \\ \text{Address of } A[3,5] &= 3000 + 2(10(3 - 1) + (5 - 1)) \\ &= 3000 + 2(24) \\ &= 3048\end{aligned}$$

**Problem 2.6** Solve Problem 5 in case of column order implementation.

$$\begin{aligned}\text{Solution Address of } A[i,j] &= B + W(m(j - LBC) + (i - LBR)) \\ \text{Address of } A[3,5] &= 3000 + 2(5(3 - 1) + (5 - 1)) \\ &= 3000 + 2(14) \\ &= 3028\end{aligned}$$

## »»» Summary

- ◆ Arrays are characterized as one-dimensional and multi-dimensional arrays.
- ◆ One-dimensional arrays are stored at consecutive locations in memory.
- ◆ Array traversal involves visiting the array elements and storing or retrieving values from it.
- ◆ Insertion is the task of adding an element into an existing array while deletion is the task of removing an element from the array.
- ◆ Sorting involves arranging the elements of an array in a specific order or sequence.
- ◆ Searching involves locating a specific element in an array.
- ◆ Multi-dimensional array either stores the array elements in row major order or column major order.
- ◆ Two-dimensional arrays are most commonly used for realizing matrices.
- ◆ Common operations performed on matrices are: addition, subtraction, multiplication, transpose.

## »»» Key Terms

- ◆ **Array** An array is defined as a collection of same type elements, such as integers, characters, strings, structures, and so on.
- ◆ **One-dimensional array** It is a group of same type data elements, such as integers, floats, or characters.
- ◆ **Multi-dimensional array** It is a group of data elements, where each element is itself an array.
- ◆ **Array subscript** It is the index identifier used to identify individual array elements.

- ◆ **Base address** It is the memory address of the first element of an array.
- ◆ **Sorting** It involves arranging the elements of an array in a specific order or sequence.
- ◆ **Searching** It involves locating a specific element in an array.
- ◆ **Row major order** It is the memory representation of a two-dimensional array in row-by-row fashion.
- ◆ **Column major order** It is the memory representation of a two-dimensional array in column-by-column fashion.

## Multiple-Choice Questions

- 2.1** Which of the following is not true about arrays?
  - (a) It uses a single name for referencing all the array elements.
  - (b) Its name can be used as a pointer to the first array element.
  - (c) It performs automatic bound checking on its own.
  - (d) It stores the different elements at consecutive memory locations.
- 2.2** Which of the following is an incorrect array representation?
  - (a) {2, 5, 6, 1, 9}
  - (b) {2.5, 5.5, 6.8, 1.0, 9.7}
  - (c) {'S', 'J', 6, '4', 'P'}
  - (d) All of the above are correct
- 2.3** While performing array insertion, the elements to the right of the point of insertion are required to be moved in which direction?
  - (a) Right
  - (b) Left
  - (c) They are not required to be moved
  - (d) None of the above
- 2.4** While performing array deletion, the elements to the right of the point of deletion are required to be moved in which direction?
  - (a) Right
  - (b) Left
  - (c) They are not required to be moved
  - (d) None of the above
- 2.5** Which of the following is a representation of multi-dimensional array in memory?
  - (a) Row major order
  - (b) Column major order
  - (c) Sequential order
  - (d) Both (a) and (b)
- 2.6**  $Address\ of\ A[i,j] = B + W(m(j - LBC) + (i - LBR))$  is the formula for computation of memory addresses of which of the following array representations?
  - (a) Column major order
  - (b) Row major order
  - (c) Sequential order
  - (d) None of the above

- 2.7** A multi-dimensional array  $A[3][7]$  possesses how many number of elements?

  - (a) 10
  - (b) 21
  - (c) 17
  - (d) None of the above

**2.8** Transposing a matrix refers to

  - (a) converting rows into columns
  - (b) converting columns into rows
  - (c) Both (a) and (b)
  - (d) None of the above

## Review Questions

- 2.1** What is an array? What are its various types?
  - 2.2** Explain the representation of a one-dimensional array in memory with the help of an illustration.
  - 2.3** What is array traversal? Why is it used?
  - 2.4** What are the typical operations associated with arrays? Explain.
  - 2.5** Write an algorithm for deleting an element at index location  $k$  in the array  $A[N]$ .
  - 2.6** What is the difference between sorting and searching?
  - 2.7** Explain the representation of a two-dimensional array in memory with the help of an illustration.
  - 2.8** Explain with the help of an illustration how a  $2 \times 2$  matrix is stored in memory using column major order representation.
  - 2.9** What is matrix multiplication? Explain with the help of an example.
  - 2.10** What is the significance of transposing a matrix?

## Programming Exercises

- 2.1** Write a C program to find the smallest element in an integer array.
  - 2.2** Write a C program to read a value and insert it at the middle of an integer array.
  - 2.3** Write a C program to sort an array of 10 integers.
  - 2.4** Write a C program to demonstrate searching on an array of ten integers.
  - 2.5** Write a C program to show how matrices are realized using two-dimensional arrays.
  - 2.6** Write a C program to perform matrix subtraction.
  - 2.7** Write a C program to perform transpose of a matrix.

## Answers to Multiple-Choice Questions

- 2.1 (c)                  2.2 (c)                  2.3 (a)                  2.4 (b)                  2.5 (d)  
2.6 (a)                  2.7 (b)                  2.8 (c)



# 3

## Chapter outline

- 3.1 Introduction
- 3.2 Linked Lists – Basic Concept
  - 3.2.1 Representation of Linked Lists
  - 3.2.2 Advantages of Linked Lists
  - 3.2.3 Disadvantages of Linked Lists
- 3.3 Linked List Implementation
  - 3.3.1 Linked List Node Declaration
  - 3.3.2 Linked List Operations
  - 3.3.3 Linked List Implementation
- 3.4 Types of Linked Lists
- 3.5 Circular Linked List
  - 3.5.1 Circular Linked List Operations
  - 3.5.2 Circular Linked List Implementation
- 3.6 Doubly Linked List
  - 3.6.1 Doubly Linked List Node Declaration
  - 3.6.2 Doubly Linked List Operations
  - 3.6.3 Doubly Linked List Implementation

*Solved Problems*

*Summary*

*Key Terms*

*Multiple-Choice Questions*

*Review Questions*

*Programming Exercises*

*Answers to Multiple-Choice Questions*



### 3.1 INTRODUCTION

In the previous chapter, we learnt about arrays and how they are used for storing same type data elements in memory. While arrays are a good way of grouping same data together, they also have a key limitation associated with them. An array is allocated fixed amount of memory space before a program is executed. Thus, if there is a need at run time to store more data in the array than its actual capacity, then there is no way of doing this. This is where a linked list becomes more useful. It allows for dynamic allocation of memory space at run time. Thus, there is no need to block memory space at compile time.

Linked list is a collection of nodes or data elements logically connected to each other. Whenever there is a need to add a new element to the list, a new node is created and appended at the end of the list. In this chapter, we will learn how a linked list is implemented and how common operations like insertion and deletion are performed on it. We will also learn about linked list variants, that is circular linked list and doubly linked list.

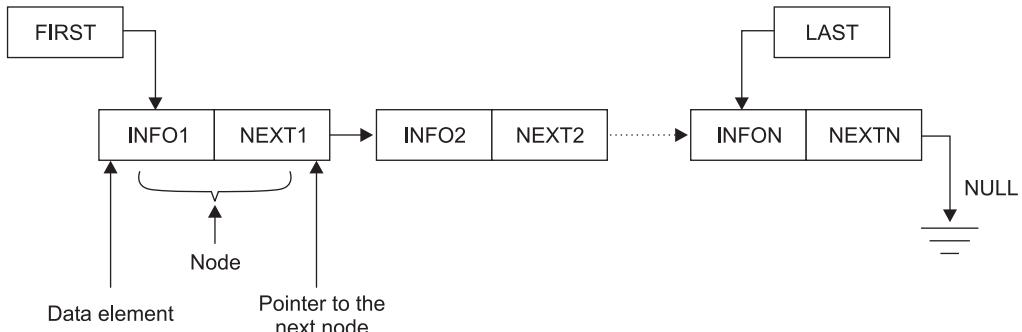
### 3.2 LINKED LISTS—BASIC CONCEPT

Linked list is a collection of data elements stored in such a manner that each element points at the next element in the list. The elements of a linked list are also referred as *nodes*. Each node has two parts: INFO and NEXT. The INFO part contains the data element while the NEXT part contains the address of the next node. The NEXT part of the last node of the list contains a NULL value indicating the end of the list. The beginning of the list is indicated with the help of a special pointer called FIRST. Similarly, the end of the list is indicated by a pointer called LAST.

#### 3.2.1 Representation of Linked Lists

Unlike arrays, the nodes of a linked list need not occupy contiguous locations in memory. Instead, they can be stored at discrete memory locations, logically connected with each other through node NEXT.

Figure 3.1 depicts the logical representation of a linked list.



**Fig. 3.1** Logical representation of a linked list

As shown in the above representation, the first and last nodes of the list are indicated by two distinct pointers, FIRST and LAST.

### 3.2.2 Advantages of Linked Lists

Some of the key advantages of linked lists are:

1. Linked lists facilitate dynamic memory management by allowing elements to be added or deleted at any time during program execution.
2. The use of linked lists ensures efficient utilization of memory space as only that much amount of memory space is reserved as is required for storing the list elements.
3. It is easy to insert or delete elements in a linked list, unlike arrays, which require shuffling of other elements with each insert and delete operation.

### 3.2.3 Disadvantages of Linked Lists

Apart from the advantages, linked lists also possess certain limitations, which are:

1. A linked list element requires more memory space in comparison to an array element because it has to also store the address of the next element in the list.
2. Accessing an element is a little more difficult in linked lists than arrays because unlike arrays, there is no index identifier associated with each list element. Thus, to access a linked list element, it is mandatory to traverse all the preceding elements.

## 3.3 LINKED LIST IMPLEMENTATION

The implementation of a linked list involves two tasks:

1. Declaring the list node
2. Defining the linked list operations

### 3.3.1 Linked List Node Declaration

Since a linked list node contains two parts, INFO and NEXT, a structure construct is best suited for its realization. The following structure declaration defines a linked list node:

```
struct node
{
    int INFO;
    struct node *NEXT;
};

typedef struct node NODE;
```

The above structure declaration defines a new data type called NODE that represents a linked list node. The node structure contains two members, INFO for storing integer data values and NEXT for storing address of the next node.

The statement, `struct node *NEXT`, indicates that the pointer NEXT points at same structure type i.e. node. Such structures that contain pointer references to their own types are called as self-referential structures.

### 3.3.2 Linked List Operations

The typical operations performed on a linked list are:

1. Insert
2. Delete
3. Search
4. Print

**1. Insert** The insert operation adds a new element to the linked list. The following tasks are performed while adding the new element:

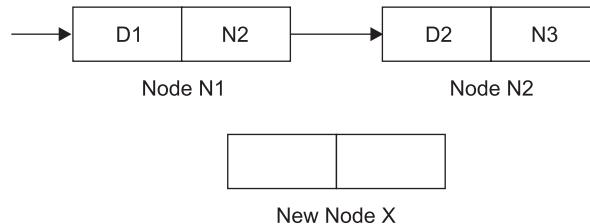
- (a) Memory space is reserved for the new node.
- (b) The element is stored in the INFO part of the new node.
- (c) The new node is connected to the existing nodes in the list.

Depending on the location where the new node is to be added, there are three scenarios possible, which are:

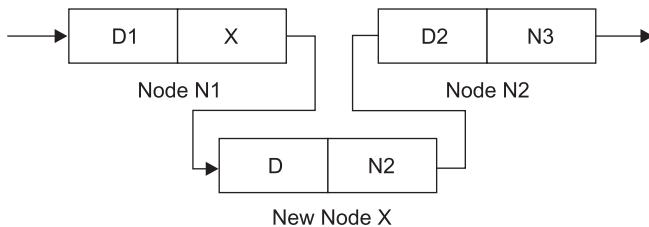
- (a) Inserting the new element at the beginning of the list
- (b) Inserting the new element at the end of the list
- (c) Inserting the new element somewhere at the middle of the list

Inserting a new element at the beginning or end of the list is easy as it only requires resetting the respective NEXT fields. However, if the new element is to be added somewhere at the middle of the list then a search operation is required to be performed to identify the point of insertion.

Figures 3.2 (a) and (b) show the insertion of a new element between two existing elements of a linked list.



**Fig 3.2 (a)** Creating a new element



**Fig 3.2 (b)** Inserting the newly created element

**Example 3.1** Write an algorithm to insert an element at the end of a linked list.

```
insert (value)
Step 1: Start
```

```

Step 2: Set PTR = addressof (New Node)
        //Allocate a new node and assign its address to the pointer PTR
Step 3: Set PTR->INFO = value;
        //Store the element value to be inserted in the INFO part of the new node
Step 4: If FIRST = NULL, then goto Step 5 else goto Step 7
        //Check whether the existing list is empty
Step 5: Set FIRST=PTR and LAST=PTR
        //Update the FIRST and LAST pointers
Step 6: Set PTR->NEXT = NULL and goto Step 8
Step 7: Set LAST->NEXT=PTR, PTR->NEXT=NULL and LAST=PTR
        //Link the newly created node at the end of the list
Step 8: Stop

```

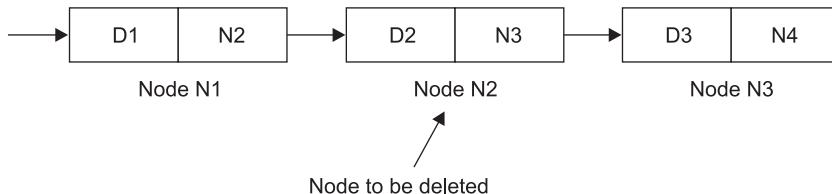
**2. Delete** The delete operation removes an existing element from the linked list. The following tasks are performed while deleting an existing element:

- (a) The location of the element is identified.
- (b) The element value is retrieved. In some cases, the element value is simply ignored.
- (c) The link pointer of the preceding node is reset.

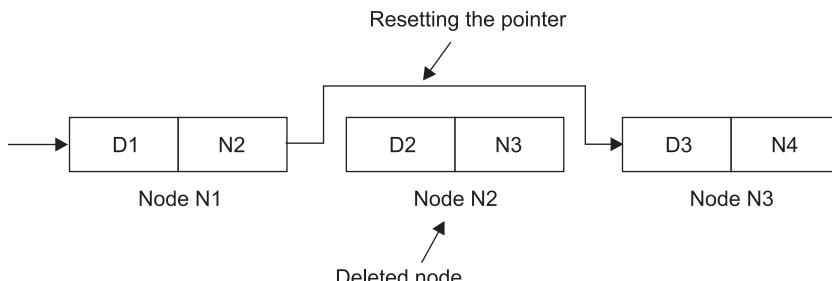
Depending on the location from where the element is to be deleted, there are three scenarios possible, which are:

- (a) Deleting an element from the beginning of the list.
- (b) Deleting an element from the end of the list.
- (c) Deleting an element somewhere from the middle of the list.

Deleting an element from the beginning or end of the list is easy as it only requires resetting the first and last pointers. However, if an element is to be deleted from within the list then a search operation is required to be performed for locating that element. Figures 3.3 (a) and (b) show the deletion of an element that is present between two existing elements of a linked list.



**Fig 3.3 (a)** Identifying the node to be deleted



**Fig 3.3 (b)** Deleting the node

**Example 3.2** Write an algorithm to delete a specific element from a linked list.

```
delete (value)
Step 1: Start
Step 2: Set LOC = search (value)
    //Call the search module to search the location of the node to be
deleted and assign it to LOC pointer
Step 3: If LOC=NULL goto Step 4 else goto Step 5
Step 4: Return ("Delete operation unsuccessful: Element not present") and
Stop
Step 5: If LOC=FIRST goto Step 6 else goto Step 10
    //Check if the element to be deleted is the first element in the list
Step 6: If FIRST=LAST goto Step 7 else goto Step 8
    //Check if there is only one element in the list
Step 7: Set FIRST=LAST=NULL and goto Step 9
Step 8: Set FIRST=FIRST->NEXT
Step 9: Return ("Delete operation successful") and Stop
Step 10: Set TEMP=LOC-1
    //Assign the location of the node present before LOC to temporary
pointer TEMP
Step 11: Set TEMP->NEXT=LOC->NEXT
    //Link the TEMP node with the node being currently pointed by LOC
Step 12: If LOC=LAST goto Step 13 else goto Step 14
    //Check if the element to be deleted is currently the last element in
the list
Step 13: Set LAST=TEMP
Step 14: Return ("Delete operation successful")
Step 15: Stop
```

**3. Search** The search operation helps to find an element in the linked list. The following tasks are performed while searching an element:

- (a) Traverse the list sequentially starting from the first node.
- (b) Return the location of the searched node as soon as a match is found.
- (c) Return a search failure notification if the entire list is traversed without any match.

The NEXT pointers help in traversing the linked list from start till end.

**Example 3.3** Write an algorithm to search a specific element in the linked list.

```
search (value)
Step 1: Start
Step 2: If FIRST=NULL goto Step 3 else goto Step 4
    //Check if the linked list is empty
Step 3: Return ("Search unsuccessful: Element not present") and Stop
Step 4: Set PTR=FIRST
Step 5: Repeat Steps 6-8 until PTR!=LAST
    //Repeat Steps 6-8 until PTR is not equal to LAST
Step 6: If PTR->INFO=value goto Step 7 else goto Step 8
Step 7: Return ("Search successful", PTR) and Stop
Step 8: Set PTR=PTR->NEXT
```

```

Step 9: If LAST->INFO=value goto Step 10 else goto Step 11
    //Check if the element to be searched is the last element in the list
Step 10: Return ("Search successful", LAST) and Stop
Step 11: Return ("Search unsuccessful: Element not present")
Step 12: Stop

```

**4. Print** The print operation prints or displays the linked list elements on the screen. To print the elements, the linked list is traversed from start till end using NEXT pointers.

**Example 3.4** Write an algorithm to print all the linked list elements.

```

print ()
Step 1: Start
Step 2: If FIRST=NULL goto Step 3 else goto Step 4
    //Check if the linked list is empty
Step 3: Display ("Empty List") and Stop
Step 4: If FIRST=LAST goto Step 5 else goto Step 6
    //Check if the list has only one element
Step 5: Display (FIRST->INFO) and Stop
Step 6: Set PTR=FIRST
Step 7: Repeat Steps 8-9 until PTR!=LAST
    //Repeat Steps 8-9 until PTR is not equal to LAST
Step 8: Display (PTR->INFO)
    //Displaying list elements
Step 9: Set PTR=PTR->NEXT
Step 10: Display (LAST->INFO)
    //Displaying last element
Step 11: Stop

```

### 3.3.3 Linked List Implementation

Linked list implementation involves declaring its structure and defining its operations. The following example shows how a linked list is implemented using C language.

**Example 3.5** Write a program to implement a linked list and perform its common operations.

Program 3.1 implements a linked list in C. It uses the insert (Example 3.1), delete (Example 3.2), search (Example 3.3) and print (Example 3.4) algorithms for realizing the common linked list operations.

**Program 3.1** *Implementation of linked list*

```

#include<stdio.h>
#include<conio.h>

/*Linked list declaration*/
struct node
{
    int INFO;
    struct node *NEXT;
};

```

Here, the structure declaration of the linked list node has been done globally so as to enable all the functions in the program to create its instances.

```

/*Declaring pointers to first and last node of the linked list*/
struct node *FIRST = NULL;
struct node *LAST = NULL;

/*Declaring function prototypes for linked list operations*/
void insert(int);
int delete(int);
void print(void);
struct node *search (int);

void main()
{
    int num1, num2, choice;
    struct node *location;

    /*Displaying a menu of choices for performing linked list operations*/
    while(1)
    {
        clrscr();
        printf("\n\nSelect an option\n");
        printf("\n1 - Insert");
        printf("\n2 - Delete");
        printf("\n3 - Search");
        printf("\n4 - Print");
        printf("\n5 - Exit");

        printf("\n\nEnter your choice: ");
        scanf("%d", &choice);

        switch(choice)
        {
            case 1:
            {
                printf("\nEnter the element to be inserted into the linked list: ");
                scanf("%d", &num1);
                insert(num1); /*Calling the insert() function*/
                printf("\n%d successfully inserted into the linked list!", num1);
                getch();
                break;
            }

            case 2:
            {
                printf("\nEnter the element to be deleted from the linked list: ");
                scanf("%d", &num1);
                num2=delete(num1); /*Calling the delete() function */
                if(num2==9999)
                    printf("\n\t%d is not present in the linked list\n\t", num1);
                else

```

```

printf("\n\tElement %d successfully deleted from the linked list\n\t",num2);
getch();
break;
}

case 3:
{
printf("\nEnter the element to be searched: ");
scanf("%d",&num1);
location=search(num1); /*Calling the search() function*/
if(location==NULL)
printf("\n\t%d is not present in the linked list\n\t",num1);
else
{
if(location==LAST)
printf("\n\tElement %d is the last element in the list",num1);
else
printf("\n\tElement %d is present before element %d in the linked list\n\t",num1,(location->NEXT)->INFO);
}
getch();
break;
}

case 4:
{
print(); /*Printing the linked list elements*/
getch();
break;
}

case 5:
{
exit(1);
break;
}

default:
{
printf("\nIncorrect choice. Please try again.");
getch();
break;
}
}
}

/*Insert function*/

```

*If an incorrect choice is entered, an error prompt is generated.*

```

void insert(int value)
{
    /*Creating a new node*/
    struct node *PTR = (struct node*)malloc(sizeof(struct node));

    /*Storing the element to be inserted in the new node*/
    PTR->INFO = value;

    /*Linking the new node to the linked list*/
    if(FIRST==NULL)
    {
        FIRST = LAST = PTR;
        PTR->NEXT=NULL;
    }
    else
    {
        LAST->NEXT = PTR;
        PTR->NEXT = NULL;
        LAST = PTR;
    }
}

/*Delete function*/
int delete(int value)
{
    struct node *LOC,*TEMP;
    int i;
    i=value;

    LOC=search(i); /*Calling the search() function*/

    if(LOC==NULL) /*Element not found*/
    return (-9999);

    if(LOC==FIRST)
    {
        if(FIRST==LAST)
        FIRST=LAST=NULL;
        else
        FIRST=FIRST->NEXT;
        return (value);
    }

    for(TEMP=FIRST; TEMP->NEXT!=LOC; TEMP=TEMP->NEXT)
    ; ←
    TEMP->NEXT=LOC->NEXT;
    if(LOC==LAST)
}

```

*Here, a single semi-colon indicates that the for loop is not executing any instructions; it is simply used to update the TEMP pointer through linked list traversal.*

### 3.10 Data Structures

```

LAST=TEMP;
return (LOC->INFO);
}

/*Search function*/
struct node *search (int value)
{
    struct node *PTR;

    if(FIRST==NULL) /*Checking for empty list*/
    return(NULL);

    /*Searching the linked list*/
    for(PTR=FIRST; PTR!=LAST; PTR=PTR->NEXT)
    if(PTR->INFO==value)
        return(PTR); /*Returning the location of the searched element*/

    if(LAST->INFO==value)
        return(LAST);
    else
        return(NULL); /*Returning NULL value indicating unsuccessful search*/
}

/*print function*/
void print()
{
    struct node *PTR;

    if(FIRST==NULL) /*Checking whether the list is empty*/
    {
        printf("\n\tEmpty List!!");
        return;
    }

    printf("\nLinked list elements:\n");
    if(FIRST==LAST) /*Checking if there is only one element in the list*/
    {
        printf("\t%d", FIRST->INFO);
        return;
    }

    /*Printing the list elements*/
    for(PTR=FIRST; PTR!=LAST; PTR=PTR->NEXT)
        printf("\t%d", PTR->INFO);
        printf("\t%d", LAST->INFO);
}

```

## Output

```
Select an option
```

```
1 - Insert  
2 - Delete  
3 - Search  
4 - Print  
5 - Exit
```

```
Enter your choice: 4
```

```
Empty List!!
```

```
Select an option
```

```
1 - Insert  
2 - Delete  
3 - Search  
4 - Print  
5 - Exit
```

```
Enter your choice: 1
```

```
Enter the element to be inserted into the linked list: 1
```

```
1 successfully inserted into the linked list!
```

```
Select an option
```

```
1 - Insert  
2 - Delete  
3 - Search  
4 - Print  
5 - Exit
```

```
Enter your choice: 1
```

```
Enter the element to be inserted into the linked list: 2
```

```
2 successfully inserted into the linked list!
```

```
Select an option
```

```
1 - Insert  
2 - Delete  
3 - Search  
4 - Print  
5 - Exit
```

```
Enter your choice: 1
```

```
Enter the element to be inserted into the linked list: 3
```

```
3 successfully inserted into the linked list!
```

```
Select an option
```

```
1 - Insert  
2 - Delete  
3 - Search  
4 - Print  
5 - Exit
```

```
Enter your choice: 3
```

```
Enter the element to be searched: 5
```

```
5 is not present in the linked list
```

```
Select an option
```

```
1 - Insert  
2 - Delete  
3 - Search  
4 - Print  
5 - Exit
```

```
Enter your choice: 3
```

```
Enter the element to be searched: 2
```

```
Element 2 is present before element 3 in the linked list
```

```
Select an option
```

```
1 - Insert  
2 - Delete  
3 - Search  
4 - Print  
5 - Exit
```

```
Enter your choice: 2
```

```
Enter the element to be deleted from the linked list: 2
```

```
Element 2 successfully deleted from the linked list
```

Select an option

```
1 - Insert  
2 - Delete  
3 - Search  
4 - Print  
5 - Exit
```

Enter your choice: 4

Linked list elements:

```
1      3
```

Select an option

```
1 - Insert  
2 - Delete  
3 - Search  
4 - Print  
5 - Exit
```

Enter your choice: 5

### Program analysis

Key Statement	Purpose
<code>void insert(int); int delete(int); void print(void); struct node *search (int);</code>	Declares the prototypes for the functions that perform linked list operations
<code>while(1)</code>	Initiates an infinite loop for displaying a menu of options; the loop terminates only when <code>exit()</code> function is called from the enclosing statement block
<code>switch(choice)</code>	Uses <code>switch</code> statement to select an appropriate <code>case</code> block as per user's choice
<code>insert(num1);</code>	Calls the <code>insert()</code> function for inserting an element into the linked list
<code>num2=delete(num1);</code>	Calls the <code>delete()</code> function for deleting an element from the linked list
<code>location=search(num1);</code>	Calls the <code>search()</code> function for searching an element in the linked list
<code>print();</code>	Calls the <code>print()</code> function for printing the linked list elements
<code>default:</code>	Refers to the <code>default</code> instruction block which is executed when the user enters an incorrect choice

Key Statement	Purpose
<b>PTR-&gt;INFO = value;</b>	Stores a <i>value</i> in the <i>INFO</i> part of the linked list node
<b>FIRST = LAST = PTR;</b>	Initializes the <i>FIRST</i> and <i>LAST</i> pointers of the linked list
<b>LAST-&gt;NEXT = PTR;</b>	Stores an address value in the <i>NEXT</i> part of the linked list node

## 3.4 TYPES OF LINKED LISTS

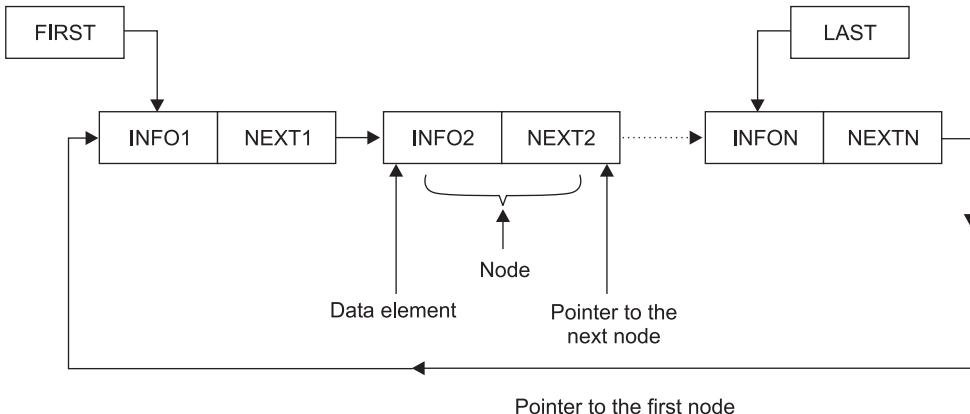
Depending on the manner in which its nodes are interconnected with each other, linked lists are categorized into the following types:

1. **Singly linked list** In this type of linked list, each node points at the successive node. Thus, the list can only be traversed in the forward direction. The linked list implementation that we saw in the previous section is an example of singly linked list.
2. **Circular list** In this type of linked list, the first and the last node are logically connected with each other, thus giving the impression of a circular list formation. Actually, the *NEXT* part of the last node contains the address of the *FIRST* node, thus connecting the rear of the list to its front.
3. **Doubly linked list** In this type of linked list, a node points at both its preceding as well as succeeding nodes. Thus, the list can be traversed in both forward as well as backward directions.

## 3.5 CIRCULAR LINKED LIST

The only difference between singly linked list and circular linked list is that the last node of singly linked list points at NULL while the last node of circular linked list points at the first list element. That means, the *NEXT* part of the last node of a circular linked list contains the address of its *FIRST* node. One of the main advantages of circular linked list is that it allows traversal of the complete list from any of its node, which is not possible with singly or doubly linked lists.

Figure 3.4 depicts the logical representation of a circular linked list.



**Fig. 3.4** Logical representation of a circular linked list

The implementation of a circular linked list involves two tasks:

1. Declaring the list node
2. Defining the list operations

The declaration of the circular linked list node is similar to the declaration of the singly linked list node. However, the definition of certain operations of a circular linked list is slightly different than that of the singly linked list.

### 3.5.1 Circular Linked List Operations

The typical operations performed on a circular linked list are:

1. Insert
2. Delete
3. Search
4. Print

**1. Insert** The insert operation in a circular list is performed in the same manner as a singly linked list. The only exception is when the element is inserted at the end of the list. In such a case, the NEXT pointer of the newly inserted node is assigned the address of the first element in the list, thus ensuring that the list stays circular.

**Example 3.6** Write an algorithm to insert an element at the end of a circular linked list.

```
insert (value)
Step 1: Start
Step 2: Set PTR = addressof (New Node)
        //Allocate a new node and assign its address to the pointer PTR
Step 3: Set PTR->INFO = value;
        //Store the element value to be inserted in the INFO part of the new node
Step 4: If FIRST = NULL, then goto Step 5 else goto Step 7
        //Check whether the existing list is empty
Step 5: Set FIRST=PTR and LAST=PTR
        //Update the FIRST and LAST pointers
Step 6: Set PTR->NEXT = FIRST and goto Step 8
        //Create a circular link
Step 7: Set LAST->NEXT=PTR, PTR->NEXT=FIRST and LAST=PTR
        //Add the newly created node at the end of the list and link it with
the first node
Step 8: Stop
```

**2. Delete** The delete operation in a circular list is performed in the same manner as a singly linked list. The only exception is when the element to be deleted is at the end of the list. In such a case, the NEXT pointer of the second last node in the list is assigned the address of the first element to ensure that the list stays circular.

**Example 3.7** Write an algorithm to delete an element from a circular linked list.

```
delete (value)
Step 1: Start
Step 2: Set LOC = search (value)
```

```

    //Call the search module to search the location of the node to be
    deleted and assign it to LOC pointer
    Step 3: If LOC=NULL goto Step 4 else goto Step 5
    Step 4: Return ("Delete operation unsuccessful: Element not present")
    and Stop
    Step 5: If LOC=FIRST goto Step 6 else goto Step 11
        //Check if the element to be deleted is the first element in the list
    Step 6: If FIRST=LAST goto Step 7 else goto Step 8
        //Check if there is only one element in the list
    Step 7: Set FIRST=LAST=NULL and goto Step 10
    Step 8: Set FIRST=FIRST->NEXT
        //Reset the FIRST pointer
    Step 9: Set Last->NEXT=FIRST
        //Link the last node with the updated FIRST pointer
    Step 10: Return ("Delete operation successful") and Stop
    Step 11: Set TEMP=LOC-1
        //Assign the location of the node present before LOC to temporary
pointer TEMP
    Step 11: Set TEMP->NEXT=LOC->NEXT
        //Link the TEMP node with the node being currently pointed by LOC
    Step 12: If LOC=LAST goto Step 13 else goto Step 15
        //Check if the element to be deleted is currently the last element in
the list
    Step 13: Set LAST=TEMP
    Step 14: Set TEMP->NEXT=FIRST
        //Create circular link
    Step 15: Return ("Delete operation successful")
    Step 16: Stop

```

**3. Search** The search operation in a circular linked list is performed in the same manner as a singly linked list. The circular list also provides the additional flexibility of starting the search from anywhere in the list. An unsuccessful search is signified when the same node is reached from where the search was started.

**4. Print** The print operation in a circular list is performed in the same manner as a singly linked list. The circular nature of the list allows us to start the print operation from anywhere in the list.

### 3.5.2 Circular Linked List Implementation

The implementation of circular linked list involves declaring its structure and defining its operations. The following example shows how a circular linked list is implemented in C.

**Example 3.8** Write a program to implement a circular linked list and perform its common operations.

Program 3.2 implements a circular linked list in C. It uses the insert (Example 3.6) and delete (Example 3.7) algorithms for realizing the insert and delete operations on the circular linked list. For performing the search and print operations, the same algorithms (Example 3.3 and Example 3.4) have been used that were earlier used for implementing a singly linked list.

### **Program 3.2** Implementation of a circular linked list

```
#include<stdio.h>
#include<conio.h>

/*Circular linked list declaration*/
struct cl_node
{
    int INFO;
    struct cl_node *NEXT;
};

/*Declaring pointers to first and last node of the list*/
struct cl_node *FIRST = NULL;
struct cl_node *LAST = NULL;

/*Declaring function prototypes for list operations*/
void insert(int);
int delete(int);
void print(void);
struct cl_node *search (int);

void main()
{
    int num1, num2, choice;
    struct cl_node *location;

    /*Displaying a menu of choices for performing list operations*/
    while(1)
    {
        clrscr();
        printf("\n\nSelect an option\n");
        printf("\n1 - Insert");
        printf("\n2 - Delete");
        printf("\n3 - Search");
        printf("\n4 - Print");
        printf("\n5 - Exit");

        printf("\n\nEnter your choice: ");
        scanf("%d", &choice);

        switch(choice)
        {
        case 1:
        {
            printf("\nEnter the element to be inserted into the circular linked list: ");
            scanf("%d", &num1);
            insert(num1); /*Calling the insert() function*/
            printf("\n%d successfully inserted into the linked list!",num1);
        }
    }
}
```

```

getch();
break;
}

case 2:
{
printf("\nEnter the element to be deleted from the circular linked list: ");
scanf("%d",&num1);
num2=delete(num1); /*Calling the delete() function */
if(num2== -9999)
printf("\n\t%d is not present in the list\n\t",num1);
else
printf("\n\tElement %d successfully deleted from the list\n\t",num2);
getch();
break;
}

case 3:
{
printf("\nEnter the element to be searched: ");
scanf("%d",&num1);
location=search(num1); /*Calling the search() function*/
if(location==NULL)
printf("\n\t%d is not present in the list\n\t",num1);
else
printf("\n\tElement %d is present before element %d in the circular linked
list\n\t",num1,(location->NEXT)->INFO);
getch();
break;
}

case 4:
{
print(); /*Printing the list elements*/
getch();
break;
}

case 5:
{
exit(1);
break;
}

default:
{
printf("\nIncorrect choice. Please try again.");
getch();
break;
}

```

```

}
}
}
}

/*Insert function*/
void insert(int value)
{
    /*Creating a new node*/
    struct cl_node *PTR = (struct cl_node*)malloc(sizeof(struct cl_node));

    /*Storing the element to be inserted in the new node*/
    PTR->INFO = value;

    /*Linking the new node to the circular linked list*/
    if(FIRST==NULL)
    {
        FIRST = LAST = PTR;
        PTR->NEXT=FIRST;
    }
    else
    {
        LAST->NEXT = PTR;
        PTR->NEXT = FIRST;
        LAST = PTR;
    }
}

/*Delete function*/
int delete(int value)
{
    struct cl_node *LOC,*TEMP;
    int i;
    i=value;

    LOC=search(i); /*Calling the search() function*/

    if(LOC==NULL) /*Element not found*/
    return(-9999);

    if(LOC==FIRST)
    {
        if(FIRST==LAST)
        FIRST=LAST=NULL;
        else
        {
            FIRST=FIRST->NEXT;
            LAST->NEXT=FIRST;
        }
    }
}

```

*The instruction PTR->NEXT=FIRST links the newly added node with the first node in the list, thus depicting a circular arrangement.*

```

}

return(value);
}

for (TEMP=FIRST; TEMP->NEXT!=LOC; TEMP=TEMP->NEXT)
;
if (LOC==LAST)
{
LAST=TEMP;
TEMP->NEXT=FIRST;
}
else
TEMP->NEXT=LOC->NEXT;
return (LOC->INFO);
}

/*Search function*/
struct cl_node *search (int value)
{
 struct cl_node *PTR;

if(FIRST==NULL) /*Checking for empty list*/
return(NULL);

if(FIRST==LAST && FIRST->INFO==value) /*Checking if there is only one
element in the list*/
return(FIRST);

/*Searching the linked list*/
for(PTR=FIRST; PTR!=LAST; PTR=PTR->NEXT)
if(PTR->INFO==value)
return(PTR); /*Returning the location of the searched element*/

if(LAST->INFO==value)
return(LAST);
else
return(NULL); /*Returning NULL value indicating unsuccessful search*/
}

/*print function*/
void print()
{
 struct cl_node *PTR;

if(FIRST==NULL) /*Checking whether the list is empty*/
{

```

```
printf("\n\tEmpty List!!");
return;
}

printf("\nCircular linked list elements:\n");
if(FIRST==LAST) /*Checking if there is only one element in the list*/
{
printf("\t%d",FIRST->INFO);
return;
}

/*Printing the list elements*/
for(PTR=FIRST;PTR!=LAST;PTR=PTR->NEXT)
printf("\t%d",PTR->INFO);
printf("\t%d",LAST->INFO);
}
```

## Output

Select an option

1 - Insert  
2 - Delete  
3 - Search  
4 - Print  
5 - Exit

Enter your choice: 4

Empty List!!

Select an option

1 - Insert  
2 - Delete  
3 - Search  
4 - Print  
5 - Exit

Enter your choice: 1

Enter the element to be inserted into the circular linked list: 1

1 successfully inserted into the linked list!

Select an option

1 - Insert  
2 - Delete  
3 - Search

```
4 - Print  
5 - Exit
```

Enter your choice: 1

Enter the element to be inserted into the circular linked list: 2

2 successfully inserted into the linked list!

Select an option

```
1 - Insert  
2 - Delete  
3 - Search  
4 - Print  
5 - Exit
```

Enter your choice: 1

Enter the element to be inserted into the circular linked list: 3

3 successfully inserted into the linked list!

Select an option

```
1 - Insert  
2 - Delete  
3 - Search  
4 - Print  
5 - Exit
```

Enter your choice: 3

Enter the element to be searched: 2

Element 2 is present before element 3 in the circular linked list

Select an option

```
1 - Insert  
2 - Delete  
3 - Search  
4 - Print  
5 - Exit
```

Enter your choice: 3

```
Enter the element to be searched: 3
```

```
Element 3 is present before element 1 in the circular linked list
```

```
1      3
```

The presence of element 3 before element 1 confirms the circular nature of the list.

```
Select an option
```

- 1 - Insert
- 2 - Delete
- 3 - Search
- 4 - Print
- 5 - Exit

```
Enter your choice: 5
```

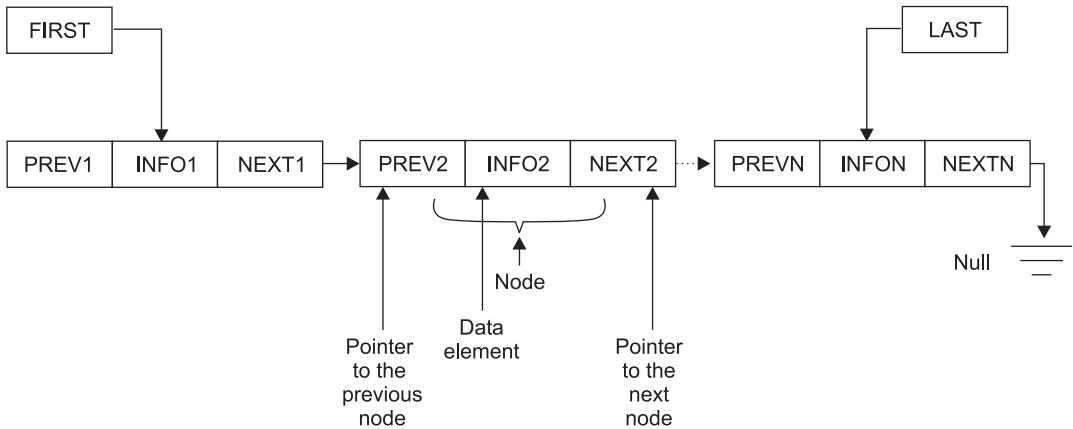
### Program analysis

Key Statement	Purpose
<code>struct cl_node *FIRST = NULL;</code> <code>struct cl_node *LAST = NULL;</code>	Declares pointers to the first and last nodes of the circular linked list
<code>void insert(int);</code> <code>int delete(int);</code> <code>void print(void);</code> <code>struct cl_node *search (int);</code>	Declares the prototypes for the functions that perform operations on the circular linked list
<code>insert(num1);</code>	Calls the <i>insert()</i> function for inserting an element into the circular linked list
<code>num2=delete(num1);</code>	Calls the <i>delete()</i> function for deleting an element from the circular linked list
<code>location=search(num1);</code>	Calls the <i>search()</i> function for searching an element in the circular linked list
<code>print();</code>	Calls the <i>print()</i> function for printing the elements of the circular linked list

## 3.6 DOUBLY LINKED LIST

Each node of a doubly linked list has three parts: INFO, NEXT, and PREVIOUS. The INFO part contains the data element while the NEXT and PREVIOUS parts contain the address of the next and previous nodes respectively. The NEXT part of the last node of the list contains a NULL value indicating the end of the list. The beginning of the list is indicated with the help of a special pointer called FIRST.

The main advantage of a doubly linked list is that it allows both forward and backward traversal. Figure 3.5 depicts the logical representation of a doubly linked list:



**Fig. 3.5** Logical representation of a doubly linked list

The implementation of a doubly linked list involves two tasks:

1. Declaring the list node
2. Defining the list operations

### 3.6.1 Doubly Linked List Node Declaration

The following structure declaration defines the node of a doubly linked list:

```
struct node
{
    int INFO;
    struct node *NEXT;
    struct node *PREVIOUS;
};

typedef struct node NODE;
```

The above structure declaration defines a new data type called NODE that represents a doubly linked list node. The node structure contains three members, INFO for storing integer data values, NEXT for storing address of the next node, and PREVIOUS for storing the address of the previous node.

### 3.6.2 Doubly Linked List Operations

The typical operations performed on a doubly linked list are:

1. Insert
  2. Delete
  3. Search
  4. Print
- 1. Insert** The insert operation in a doubly linked list is performed in the same manner as a singly linked list. The only exception is that the additional node pointer PREVIOUS is also required to be updated for the new node at the time of insertion.

**Example 3.9** Write an algorithm to insert an element at the end of a doubly linked list.

```
insert (value)
Step 1: Start
Step 2: Set PTR = addressof (New Node)
        //Allocate a new node and assign its address to the pointer PTR
Step 3: Set PTR->INFO = value;
        //Store the element value to be inserted in the INFO part of the new
node
Step 4: If FIRST = NULL, then goto Step 5 else goto Step 7
        //Check whether the existing list is empty
Step 5: Set FIRST=PTR and LAST=PTR
        //Update the FIRST and LAST pointers
Step 6: Set PTR->NEXT = PTR -> PREVIOUS = NULL and goto Step 8
Step 7: Set LAST->NEXT=PTR, PTR->PREVIOUS = LAST, PTR->NEXT=NULL, and
LAST=PTR
        //Link the newly created node at the end of the list
Step 8: Stop
```

**2. Delete** The delete operation in a doubly linked list is performed in the same manner as a singly linked list. The only exception is that the additional node pointer PREVIOUS of the adjacent node is also required to be updated at the time of deletion.

**Example 3.10** Write an algorithm to delete an element from a doubly linked list.

```
delete (value)
Step 1: Start
Step 2: Set LOC = search (value)
        //Call the search module to search the location of the node to be
deleted and assign it to LOC pointer
Step 3: If LOC=NULL goto Step 4 else goto Step 5
Step 4: Return ("Delete operation unsuccessful: Element not present")
and Stop
Step 5: If LOC=FIRST goto Step 6 else goto Step 10
        //Check if the element to be deleted is the first element in the list
Step 6: If FIRST=LAST goto Step 7 else goto Step 8
        //Check if there is only one element in the list
Step 7: Set FIRST=LAST=NULL and goto Step 9
Step 8: Set FIRST->NEXT->PREVIOUS=NULL and FIRST=FIRST->NEXT
Step 9: Return ("Delete operation successful") and Stop
Step 10: Set TEMP=LOC-1
        //Assign the location of the node present before LOC to temporary
pointer TEMP
Step 11: If LOC=LAST goto Step 12 else goto Step 13
Step 12: Set LAST=TEMP, TEMP->NEXT=NULL and goto Step 15
Step 13: Set TEMP->NEXT=LOC->NEXT
Step 14: Set LOC->NEXT->PREVIOUS=TEMP
        //Delete the LOC node and set the adjacent NEXT and PREVIOUS pointers
Step 15: Return ("Delete operation successful")
Step 16: Stop
```

3. **Search** The search operation in a doubly linked list is performed in the same manner as a singly linked list. The doubly linked list also provides the additional flexibility of starting the search from the end and moving backwards towards the front.
4. **Print** The print operation in a doubly linked list is performed in the same manner as a singly linked list. The doubly linked list also allows you to print the list elements in reverse order by starting from the end and moving backwards towards the front.

### 3.6.3 Doubly Linked List Implementation

The implementation of doubly linked list involves declaring its structure and defining its operations. The following example shows how a doubly linked list is implemented in C.

**Example 3.11** Write a program to implement a doubly linked list and perform its common operations. Program 3.3 implements a doubly linked list in C. It uses the insert (Example 3.9) and delete (Example 3.10) algorithms for realizing the insert and delete operations on the doubly linked list. For performing the search and print operations, the same algorithms (Example 3.3 and Example 3.4) have been used that were earlier used for implementing a singly linked list.

**Program 3.3** *Implementation of a doubly linked list*

```
#include<stdio.h>
#include<conio.h>

/*Doubly linked list declaration*/
struct dl_node
{
    int INFO;
    struct dl_node *NEXT;
    struct dl_node *PREVIOUS;
};

/*Declaring pointers to first and last node of the doubly linked list*/
struct dl_node *FIRST = NULL;
struct dl_node *LAST = NULL;

/*Declaring function prototypes for list operations*/
void insert(int);
int delete(int);
void print(void);
struct dl_node *search (int);

void main()
{
    int num1, num2, choice;
    struct dl_node *location;

    /*Displaying a menu of choices for performing list operations*/
    while(1)
```

```

{
clrscr();
printf("\n\nSelect an option\n");
printf("\n1 - Insert");
printf("\n2 - Delete");
printf("\n3 - Search");
printf("\n4 - Print");
printf("\n5 - Exit");

printf("\n\nEnter your choice: ");
scanf("%d", &choice);

switch(choice)
{
case 1:
{
printf("\nEnter the element to be inserted into the doubly linked list: ");
scanf("%d", &num1);
insert(num1); /*Calling the insert() function*/
printf("\n%d successfully inserted into the linked list!", num1);
getch();
break;
}

case 2:
{
printf("\nEnter the element to be deleted from the doubly linked list: ");
scanf("%d", &num1);
num2=delete(num1); /*Calling the delete() function */
if(num2==9999)
printf("\n\t%d is not present in the doubly linked list\n\t", num1);
else
printf("\n\tElement %d successfully deleted from the doubly linked list\n\t", num2);
getch();
break;
}

case 3:
{
printf("\nEnter the element to be searched: ");
scanf("%d", &num1);
location=search(num1); /*Calling the search()*/
if(location==NULL)
printf("\n\t%d is not present in the list\n\t", num1);
else
{
if(location==LAST)
printf("\n\tElement %d is the last element in the list", num1);
}
}
}
}

```

```

    else
        printf("\n\tElement %d is present before element %d in the doubly linked
list\n\t",num1,(location->NEXT)->INFO);
    }
    getch();
    break;
}

case 4:
{
print(); /*Printing the list elements*/
getch();
break;
}

case 5:
{
exit(1);
break;
}

default:
{
printf("\nIncorrect choice. Please try again.");
getch();
break;
}
}
}
}
}

/*Insert function*/
void insert(int value)
{
/*Creating a new node*/
struct dl_node *PTR = (struct dl_node*)malloc(sizeof(struct dl_node));

/*Storing the element to be inserted in the new node*/
PTR->INFO = value;

/*Linking the new node to the doubly linked list*/
if(FIRST==NULL)
{
FIRST = LAST = PTR;
PTR->NEXT=NULL;
PTR->PREVIOUS=NULL;
}
else
{

```

```

LAST->NEXT = PTR;
PTR->NEXT = NULL;
PTR->PREVIOUS = LAST;
LAST = PTR;
}

}

/*Delete function*/
int delete(int value)
{
    struct dl_node *LOC,*TEMP;
    int i;
    i=value;

    LOC=search(i); /*Calling the search() function*/

    if(LOC==NULL) /*Element not found*/
        return(-9999);

    if(LOC==FIRST)
    {
        if(FIRST==LAST)
            FIRST=LAST=NULL;
        else
        {
            FIRST->NEXT->PREVIOUS=NULL;
            FIRST=FIRST->NEXT;
        }
        return(value);
    }

    for(TEMP=FIRST;TEMP->NEXT!=LOC;TEMP=TEMP->NEXT)
    ;
    if(LOC==LAST)
    {
        LAST=TEMP;
        TEMP->NEXT=NULL;
    }
    else
    {
        TEMP->NEXT=LOC->NEXT;
        LOC->NEXT->PREVIOUS=TEMP;
    }
    return(LOC->INFO);
}

```

*A doubly linked list requires two pointers to be updated, NEXT and PREVIOUS.*

```

/*Search function*/
struct dl_node *search (int value)
{
    struct dl_node *PTR;

    if(FIRST==NULL) /*Checking for empty list*/
        return(NULL);

    if(FIRST==LAST && FIRST->INFO==value) /*Checking if there is only one
element in the list*/
        return(FIRST);

    /*Searching the linked list*/
    for(PTR=FIRST;PTR!=LAST;PTR=PTR->NEXT)
        if(PTR->INFO==value)
            return(PTR); /*Returning the location of the searched element*/

    if(LAST->INFO==value)
        return(LAST);
    else
        return(NULL); /*Returning NULL value indicating unsuccessful search*/
}

/*print function*/
void print()
{
    struct dl_node *PTR;

    if(FIRST==NULL) /*Checking whether the list is empty*/
    {
        printf("\n\tEmpty List!!");
        return;
    }

    printf("\nDoubly linked list elements:\n");
    if(FIRST==LAST) /*Checking if there is only one element in the list*/
    {
        printf("\t%d",FIRST->INFO);
        return;
    }

    /*Printing the list elements*/
    for(PTR=FIRST;PTR!=LAST;PTR=PTR->NEXT)
        printf("\t%d",PTR->INFO);
        printf("\t%d",LAST->INFO);
}

```

## Output

The output of this program is same as Example 3.5 (singly linked list implementation).

## Program analysis

Key Statement	Purpose
<b>struct dl_node *FIRST = NULL;</b> <b>struct dl_node *LAST = NULL;</b>	Declares pointers to the first and last nodes of the doubly linked list
<b>void insert(int);</b> <b>int delete(int);</b> <b>void print(void);</b> <b>struct dl_node *search (int);</b>	Declares the prototypes for the functions that perform operations on the doubly linked list
<b>insert(num1);</b>	Calls the <i>insert()</i> function for inserting an element into the doubly linked list
<b>num2=delete(num1);</b>	Calls the <i>delete()</i> function for deleting an element from the doubly linked list
<b>location=search(num1);</b>	Calls the <i>search()</i> function for searching an element in the doubly linked list
<b>print();</b>	Calls the <i>print()</i> function for printing the elements of the doubly linked list
<b>struct dl_node *PTR = (struct dl_node*) malloc(sizeof(struct dl_node));</b>	Creates a new node of the doubly linked list using dynamic memory allocation
<b>PTR-&gt;NEXT = NULL;</b> <b>PTR-&gt;PREVIOUS = LAST;</b>	Updates both the <i>NEXT</i> and <i>PREVIOUS</i> pointers of the node of a doubly linked list

## Solved Problems

**Problem 3.1** Write the code snippet for declaring the node of a singly linked list that stores students-related data.

### Solution

```
struct student
{
    char name[30];
    int rollno;
    float percentage;
};

struct node
{
    struct student S;
    struct node *NEXT;
};
typedef struct node NODE;
```

**Problem 3.2** Write a C function to print the elements of a doubly linked list in reverse order.

### Solution

```
/*print function*/
void print()
{
    struct node *PTR;

    if(FIRST==NULL) /*Checking whether the list is empty*/
    {
        printf("\n\tEmpty List!!");
        return;
    }

    printf("\nLinked list elements:\n");
    if(FIRST==LAST) /*Checking if there is only one element in the list*/
    {
        printf("\t%d", FIRST->INFO);
        return;
    }

    /*Printing the list elements in reverse order*/
    for(PTR=LAST; PTR!=FIRST; PTR=PTR->PREVIOUS)
        printf("\t%d", PTR->INFO);
    printf("\t%d", FIRST->INFO);
}
```

## »»» Summary



- ◆ Linked list is a collection of nodes or data elements logically connected to each other.
- ◆ Each node of a linked list has two parts: INFO and NEXT. The INFO part contains the data element while the NEXT part contains the address of the next node in the list.
- ◆ The implementation of a linked list involves declaring the list node and defining the list operations.
- ◆ The typical operations performed on a linked list are: insert, delete, search and print.
- ◆ The various types of linked lists are: singly linked list, doubly linked list, and circular linked list.
- ◆ In a circular linked list, the first and last nodes are logically connected with each other through the NEXT pointer.
- ◆ In a doubly linked list, a node points at both its preceding as well as succeeding nodes.

## »»» Key Terms



- ◆ **Singly linked list** Is a type of a linked list, in which each node points at the successive node.
- ◆ **Circular list** Is a type of a linked list, in which the last element points at the first element in the list, thus, giving the impression of a circular list formation.

- ◆ **Doubly linked list** Is a type of a linked list, in which a node points at both its preceding as well as succeeding nodes.
- ◆ **INFO** Is a part of a linked list node that stores the element value.
- ◆ **NEXT** Is a part of a linked list node that stores the address of the next node.
- ◆ **PREVIOUS** Is a part of a linked list node that stores the address of the previous node.
- ◆ **FIRST** Is a pointer to the first node of a linked list.
- ◆ **LAST** Is a pointer to the last node of a linked list.
- ◆ **Insert** Inserts an element into a linked list.
- ◆ **Delete** Deletes an element from a linked list.
- ◆ **Search** Search the linked list for a specific element.
- ◆ **Print** Prints the elements of a linked list.

## Multiple-Choice Questions

- 3.1** Which of the following is not true about linked lists?
  - (a) It is a collection of linked nodes.
  - (b) It helps in dynamic allocation of memory space.
  - (c) It allows direct access to any of the nodes.
  - (d) It requires more memory space in comparison to an array.
- 3.2** Which node pointers should be updated if a new node B is to be inserted in the middle of A and C nodes of a singly linked list?
  - (a) NEXT pointer of A and NEXT pointer of C
  - (b) NEXT pointer of B and NEXT pointer of C
  - (c) NEXT pointer of B
  - (d) NEXT pointer of A and NEXT pointer of B
- 3.3** A circular linked list contains four nodes {A, B, C, D}. Which node pointers should be updated if a new node E is to be inserted at end of the list?
  - (a) NEXT pointer of D and NEXT pointer of E
  - (b) NEXT pointer of E
  - (c) NEXT pointer of E and NEXT pointer of A
  - (d) NEXT pointer of E and START POINTER
- 3.4** Which node pointers should be updated if a new node B is to be inserted in the middle of A and C nodes of a doubly linked list?
  - (a) NEXT pointer of A, PREVIOUS pointer of B, NEXT pointer of C, and PREVIOUS pointer of C
  - (b) NEXT pointer of A, PREVIOUS pointer of B, NEXT pointer of B, and PREVIOUS pointer of C
  - (c) NEXT pointer of A, PREVIOUS pointer of A, NEXT pointer of B, and PREVIOUS pointer of C
  - (d) None of the above
- 3.5** Which of the following statements is true about doubly linked list?
  - (a) It allows list traversal only in forward direction.
  - (b) It allows list traversal only in backward direction.
  - (c) It allows list traversal in both forward and backward direction.
  - (d) It allows complete list traversal starting from any of the nodes.

- 3.6** Which of the following statements is true about circular linked list?
- (a) It allows complete list traversal starting from any of the nodes.
  - (b) It allows complete list traversal only if we begin from the FIRST node.
  - (c) Like singly and doubly linked lists, the NEXT part of the last node of a circular linked list contains a NULL pointer indicating end of the list.
  - (d) None of the above
- 3.7** You are required to create a linked list for storing integer elements. Which of the following linked list implementations will require maximum amount of memory space?
- (a) Singly linked
  - (b) Doubly linked
  - (c) Circular
  - (d) All of the above will occupy same space in memory
- 3.8** Which of the following linked list types allows you to print the list elements in reverse order?
- (a) Doubly
  - (b) Singly
  - (c) Circular
  - (d) None of the above

## Review Questions

- 3.1** What is a linked list? What are its various types?
- 3.2** Explain the representation of a linked list in memory with the help of an illustration.
- 3.3** Explain the typical operations that are performed on a linked list.
- 3.4** Explain the key advantages and disadvantages of linked lists.
- 3.5** What is a circular linked list? How is it different from a normal linked list?
- 3.6** What is a doubly linked list? Why is it used?
- 3.7** Write the algorithm for searching an element in a singly linked list.
- 3.8** Write the algorithm for inserting an element in a circular linked list.

## Programming Exercises

- 3.1** Write a code snippet for declaring the node of a doubly linked list.
- 3.2** Write a C function to delete a node from a singly linked list.
- 3.3** Write a C function to insert a new node at the end of a circular linked list.
- 3.4** Write a C function to print the elements of a linked list.
- 3.5** Write a C function to print the elements of a doubly linked list in both forward and backward directions.

## Answers to Multiple-Choice Questions

- |         |         |         |         |         |
|---------|---------|---------|---------|---------|
| 3.1 (c) | 3.2 (d) | 3.3 (a) | 3.4 (b) | 3.5 (c) |
| 3.6 (a) | 3.7 (b) | 3.8 (a) |         |         |



## **UNIT-II**

---

# **Linear Data Structures – Stacks, Queues**

### **CHAPTERS**

Chapter 4: Stacks

Chapter 5: Queues



# 4

## STACKS

Chapter outline

- 4.1 Introduction
- 4.2 Stacks
  - 4.2.1 Stack Representation in Memory
  - 4.2.2 Arrays Vs Stacks
- 4.3 Stack Operations
  - 4.3.1 Push
  - 4.3.2 Pop
  - 4.3.3 An Example of Stack Operations
- 4.4 Stack Implementation
  - 4.4.1 Array Implementation of Stacks
  - 4.4.2 Linked Implementation of Stacks

*Solved Problems*

*Summary*

*Key Terms*

*Multiple-Choice Questions*

*Review Questions*

*Programming Exercises*

*Answers to Multiple-Choice Questions*



## 4.1 INTRODUCTION

In the previous chapters, we learnt how arrays are used for implementing linear data structures. Arrays provide the flexibility of adding or removing elements anywhere in the list. But there are certain linear data structures that permit the insertion and deletion operations only at the beginning or end of the list, but not in the middle. Such data structures have significant importance in systems processes such as compilation and program control.

Stack is one such data structure which is in fact one of the very first data structures that students get familiar with while studying this subject.

## 4.2 STACKS

Stack is a linear data structure in which items are added or removed only at one end, called *top of the stack*. Thus, there is no way to add or delete elements anywhere else in the stack. A stack is based on Last-In-First-Out (LIFO) principle that means the data item that is inserted last into the stack is the first one to be removed from the stack. We can relate a stack to certain real-life objects and situations, as shown in Figs. 4.1 (a) and (b).

As we can see in Fig. 4.1, one can add a new book to an existing stack of books only at its top and nowhere else. Similarly, a plate cannot be added at the middle of the plates stack; one has to first remove all the plates above the insertion point for the new plate to be added there. Another apt example of a stack is a set of bangles worn by Indian women on their arms. A bangle can only be worn from one side of the hand and to remove a bangle from the middle one has to first remove all the prior bangles.

The concept of stack in data structures follows the same analogy as the stack of books or the stack of plates. We may use a stack in data structures to store built-in or user-defined type elements depending upon our programming requirements. Irrespective of the type of elements stored, each stack implementation follows similar representation in memory, as explained next.

### 4.2.1 Stack Representation in Memory

Just like their real world counterparts, stacks appear as a group of elements stored at contiguous locations in memory. Each successive insert or delete operation adds or removes an item from the group. The top location of the stack or the point of addition or deletion is maintained by a pointer called *top*. Figure 4.2 shows the logical representation of stacks in memory.

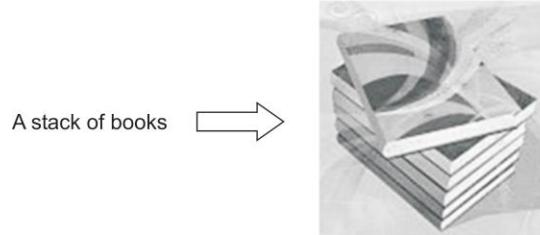


Fig. 4.1(a) Stack of books

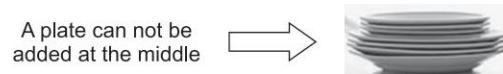


Fig. 4.1(b) Stack of plates



### Mind Jog

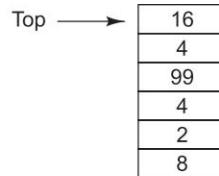
#### Who discovered stacks?

Stack was first proposed in 1957 by a German computer scientist Friedrich L. Bauer.

As we can see in Fig. 4.2, there are six elements in the stack with element 16 being at the top of the stack.



**Note** *The logical representation of stacks showing stack elements stored at contiguous memory location might be true in case of their array implementation but the same might not be true in case of their linked implementation, as we shall study later in this chapter.*



**Fig. 4.2** Logical representation of stacks

## 4.2.2 Arrays vs. Stacks

While both arrays and stacks may look to be similar in their logical representation, they are different in several aspects, as explained in Table 4.1.

**Table 4.1** Arrays vs. Stacks

Arrays	Stacks
Arrays provide the flexibility of adding or removing data elements anywhere in the list, i.e., at the beginning, end or anywhere in the middle. While this flexibility may seem to be a boon in certain situations, the same may not be true in situations where frequent insertions or deletions are required. This is because; each insertion or deletion in arrays requires the adjoining elements to be shifted to new locations, which is an overhead.	Stacks restrict the insertion or deletion of elements to only one place in the list i.e. the top of the stack. Thus, there are no associated overheads of shifting other elements to new locations.
By using arrays, a programmer can realize common scenarios where grouping of records is required, for example inventory management, employee records management, etc.	Stacks find their usage as vital in solutions to advanced systems problems such as recursion control, expression evaluation, etc.



### Check Point

#### 1. What is a stack?

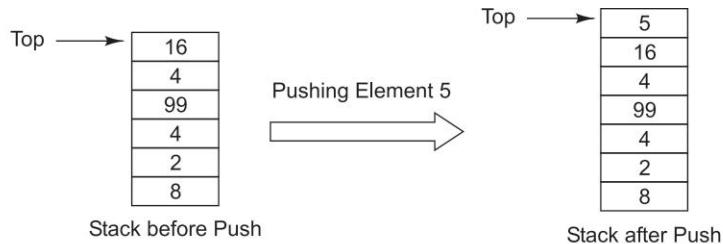
**Ans.** Stack is a linear data structure in which items are added or removed only at one end, called top.

#### 2. What is LIFO?

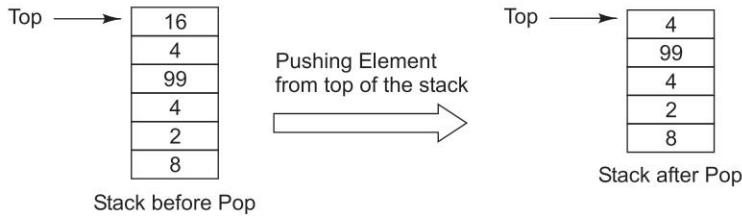
**Ans.** Last-In-First-Out (LIFO) principle specifies that the data item that is inserted last into the stack is the first one to be removed from the stack.

## 4.3 STACK OPERATIONS

There are two key operations associated with the stack data structure: push and pop. Adding an element to the stack is referred as push operation while reading or deleting an element from the stack is referred as pop operation. Figures 4.3 (a) and (b) depict the push and pop operations on a stack.



**Fig. 4.3(a)** Push operation



**Fig. 4.3(b)** Pop operation



### Note

*Top is the cornerstone of the stack data structure as it points at the entry/exit gateway of the stack.*

#### 4.3.1 Push

As we can see in Fig. 4.3 (a), the push operation involves the following subtasks:

1. Receiving the element to be inserted
2. Incrementing the stack pointer, top
3. Storing the received element at new location of top

Thus, the programmatic realization of the push operation requires implementation of the above mentioned subtasks, as we shall see later in this chapter.



### Tip

*What happens if the stack is full and there is no more room to push any new element? Such a condition is referred as stack overflow. It is always advisable to implement appropriate overflow handling mechanisms in a program to counter any unexpected results.*

#### 4.3.2 Pop

As we can see in Fig. 4.3 (b), the pop operation involves the following subtasks:

Retrieving or removing the element at the top of the stack.

Decrementing the stack pointer, top.

Thus, the programmatic realization of the pop operation requires implementation of the above mentioned subtasks, as we shall see later in this chapter.

Some pop implementations require the popped element to be returned back to the calling function while others may simply focus on updating the stack pointer and ignore the popped element all together. The choice of a particular type of implementation depends solely on the programming situation at hand.



### Tip

*What happens if the stack is empty and we want to perform the pop operation? Such a condition is referred as stack underflow. It is always advisable to implement appropriate underflow handling mechanisms in a program to counter any unexpected results.*

### 4.3.3 An Example of Stack Operations

Figure 4.4 shows how the stack contents change after a series of push and pop operations.

We can see in this figure how stack contents change by the push/pop operations occurring at one end of the stack, i.e., its top.

## 4.4 STACK IMPLEMENTATION

Stack implementation involves choosing the data storage mechanism for storing stack elements and implementing methods for performing the two stack operations, push and pop. A typical implementation of the push operation checks if there is any room left in the stack, and if there is any, it increments the stack counter by one and inserts the received item at the top of the stack. Similarly, the implementation of the pop operation checks whether or not the stack is already empty, if it is not, it removes the top element of the stack and decrements the stack counter by one.



### Mind Jog

**Is stack a restricted data structure?**

*Yes, because limited operations can be performed on it.*



### Mind Jog

**Which programming language provide built-in support for stacks?**

*LISP and Python*



### Check Point

#### 1. What is a push operation?

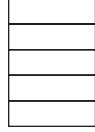
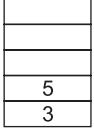
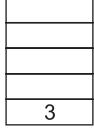
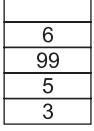
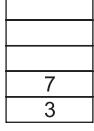
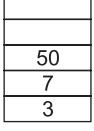
**Ans.** Adding an element into the stack is referred as push operation.

#### 2. What is a pop operation?

**Ans.** Reading or deleting an element from the stack is referred as pop operation.

### 4.4.1 Array Implementation of Stacks

The array implementation of stacks involves allocation of fixed size array in the memory. Both stack operations (push and pop) are made on this array with a constant check being made to ensure that the array does not go out of bounds.

<b>Initial Empty Stack</b>	Top → Null	 Empty Stack
<b>Push (3), Push (5)</b>	Top →	 Stack Contents
<b>Pop ()</b>	Top →	 Stack Contents
<b>Push(7), Push (99), Push (6)</b>	Top →	 Stack Contents
<b>Pop (), Pop ()</b>	Top →	 Stack Contents
<b>Push (50)</b>	Top →	 Stack Contents
<b>Pop (), Pop (), Pop ()</b>	Top → Null	 Empty Stack

**Fig 4.4** Stack operations

**Push Operation** The push operation involves checking whether or not the stack pointer is pointing at the upper bound of the array. If it is not, the stack pointer is incremented by 1 and the new item is pushed (inserted) at the top of the stack.

**Example 4.1** Write an algorithm to implement the push operation under array representation of stacks.

```
push(stack[MAX], element)
Step 1: Start
Step 2: If top = MAX-1 goto Step 3 else goto Step 4
Step 3: Display message "Stack Full" and exit
Step 4: top = top + 1
Step 5: stack[top] = element
Step 6: Stop
```

The above algorithm inserts an element at the top of a stack of size MAX.

**Pop Operation** The pop operation involves checking whether or not the stack pointer is already pointing at NULL (empty stack). If it is not, the item that is being currently pointed is popped (removed) from the stack (array) and the stack pointer is decremented by 1.

**Example 4.2** Write an algorithm to implement the pop operation under array representation of stacks.

```
pop(stack[MAX], element)
Step 1: Start
Step 2: If top = -1 goto Step 3 else goto Step 4
Step 3: Display message "Stack Empty" and exit
Step 4: Return stack[top] and set top = top - 1
Step 5: Stop
```

The above algorithm removes the element at the top of the stack.

### Implementation

**Example 4.3** Write a program to implement a stack using arrays and perform its common operations.

Program 4.1 implements a stack using arrays in C. It uses the push (Example 4.1) and pop (Example 4.2) algorithms for realizing the common stack operations.

**Program 4.1** *Implementing a stack using arrays*

```
/*Program for demonstrating implementation of stacks using arrays*/
#include <stdio.h>
#include <conio.h>

int stack[100]; /*Declaring a 100 element stack array*/
int top=-1; /*Declaring and initializing the stack pointer*/
```

If we do not initialize the top variable then it may continue to store garbage value which may lead to erroneous results

```
void push(int); /*Declaring a function prototype for inserting an element into the stack*/
```

```

int pop(); /*Declaring a function prototype for removing an element from
the stack*/
void display(); /*Declaring a function prototype for displaying the
elements of a stack*/

void main()
{
    int choice;
    int num1=0, num2=0;
    while(1) ←
    {
        clrscr();
        /*Creating an interactive interface for performing stack operations*/
        printf("Select a choice from the following:");
        printf("\n[1] Push an element into the stack");
        printf("\n[2] Pop out an element from the stack");
        printf("\n[3] Display the stack elements");
        printf("\n[4] Exit\n");
        printf("\n\tYour choice: ");
        scanf("%d", &choice);

        switch(choice)
        {
            case 1:
            {
                printf("\n\tEnter the element to be pushed into the stack: ");
                scanf("%d", &num1);
                push(num1); /*Inserting an element*/
                break;
            }

            case 2:
            {
                num2=pop(); /*Removing an element*/
                printf("\n\t%d element popped out of the stack\n\t", num2);
                getch();
                break;
            }

            case 3:
            {
                display(); /*Displaying stack elements*/
                getch();
                break;
            }

            case 4:
            exit(1);
        }
    }
}

```

*Here, while (1) signifies an infinite looping condition that'll continue to execute the statements within until a jump statement is encountered*

```

break;

default:←
printf("\nInvalid choice!\n");
break;
}
}

/*Push function*/
void push(int element)
{
    if(top==99) /*Checking whether the stack is full*/
    {
        printf("Stack is Full.\n");
        getch();
        exit(1);
    }
    top=top+1; /*Incrementing stack pointer*/
    stack[top]=element; /*Inserting the new element*/
}

/*Pop function*/
int pop()
{
    if(top==-1) /*Checking whether the stack is empty*/
    {
        printf("\n\tStack is Empty.\n");
        getch();
        exit(1);
    }
    return(stack[top--]); /*Returning the top element and decrementing the
stack pointer*/
}

void display()
{
    int i;
    printf("\n\tThe various stack elements are:\n");
    for(i=top;i>=0;i--)
        printf("\t%d\n",stack[i]); /*Printing stack elements*/
}

```

*Default blocks are always advisable in switch-case constructs as it allows handling of incorrect input values*

*The upper bound of 99 shows that this stack can store a maximum of 100 elements*

*Here, NULL value is represented by -1*



*The above code shows storage of an integer type element into the stack. However, we may store other built-in or user-defined type elements in the stack as per our own requirements.*

## Output

```
Select a choice from the following:  
[1] Push an element into the stack  
[2] Pop out an element from the stack  
[3] Display the stack elements  
[4] Exit  
  
Your choice: 1  
  
Enter the element to be pushed into the stack: 1  
  
Select a choice from the following:  
[1] Push an element into the stack  
[2] Pop out an element from the stack  
[3] Display the stack elements  
[4] Exit  
  
Your choice: 1  
  
Enter the element to be pushed into the stack: 2  
  
Select a choice from the following:  
[1] Push an element into the stack  
[2] Pop out an element from the stack  
[3] Display the stack elements  
[4] Exit  
  
Your choice: 1  
  
Enter the element to be pushed into the stack: 3  
  
Select a choice from the following:  
[1] Push an element into the stack  
[2] Pop out an element from the stack  
[3] Display the stack elements  
[4] Exit  
  
Your choice: 3  
The various stack elements are:  
3  
2  
1  
  
Select a choice from the following:  
[1] Push an element into the stack  
[2] Pop out an element from the stack  
[3] Display the stack elements  
[4] Exit
```

```
Your choice: 2
```

```
3 element popped out of the stack
```

```
Select a choice from the following:
```

- [1] Push an element into the stack
- [2] Pop out an element from the stack
- [3] Display the stack elements
- [4] Exit

```
Your choice: 3
```

```
The various stack elements are:
```

```
2  
1
```

```
Select a choice from the following:
```

- [1] Push an element into the stack
- [2] Pop out an element from the stack
- [3] Display the stack elements
- [4] Exit

```
Your choice: 4
```

### Program analysis

Key Statement	Purpose
<code>int stack[100];</code>	Declares an array to represent a stack
<code>void push(int);</code> <code>int pop();</code> <code>void display();</code>	Declares prototypes for the functions that perform stack operations
<code>push(num1);</code>	Calls the <i>push()</i> function for inserting an element into the stack
<code>num2=pop();</code>	Calls the <i>pop()</i> function for deleting an element from the stack
<code>display();</code>	Calls the <i>display()</i> function for displaying the stack elements
<code>top=top+1;</code> <code>stack[top]=element;</code>	Inserts an element at the top of the stack and updates the stack pointer

### 4.4.2 Linked Implementation of Stacks

The linked implementation of stacks involves dynamically allocating memory space at run time while performing stack operations. Since, the allocation of memory space is dynamic, the stack consumes only that much amount of space as is required for holding its data elements. This is contrary to array-

implemented stacks which continue to occupy a fixed memory space even if there are no elements present. Thus, linked implementation of stacks based on dynamic memory allocation technique prevents wastage of memory space.



**Note** *The linked implementation of stacks is based on dynamic memory management techniques, which allow allocation and deallocation of memory space at runtime.*

**Push Operation** The push operation under linked implementation of stacks involves the following tasks:

1. Reserving memory space of the size of a stack element in memory
2. Storing the pushed (inserted) value at the new location
3. Linking the new element with existing stack
4. Updating the stack pointer

**Example 4.4** Write an algorithm to implement the push operation under linked representation of stacks.

```
push(structure stack, element, next, value)
```

Step 1: Start

Step 2: Set `ptr=(struct stack*)malloc(sizeof(struct stack))`, to reserve a block of memory for the new stack node and assign its address to pointer `ptr`

Step 3: Set `ptr->element=value`, to copy the inserted value into the new node

Step 4: Set `ptr->next=top`, to link the new node to the current top node

Step 5: Set `top = ptr` to designate the new node as the top node

Step 6: Return

Step 7: Stop

The above algorithm inserts an element at the top of the stack.

**Pop Operation** The pop operation under linked implementation of stacks involves the following tasks:

1. Checking whether the stack is empty
2. Retrieving the top element of the stack
3. Updating the stack pointer
4. Returning the retrieved (popped) value

**Example 4.5** Write an algorithm in C to implement the pop operation under linked representation of stacks.

```
pop(structure stack, element, next)
```

Step 1: Start

Step 2: If `top = NULL` goto Step 3 else goto Step 4

Step 3: Display message "Stack Empty" and exit

Step 4: Set `temp=top->element`, to retrieve the element at top node of the stack

Step 5: Set `top=top->next`, to designate the next stack node as the top node

Step 6: Return `temp`

Step 7: Stop

The above algorithm removes the element at the top of the stack.

## Implementation

**Example 4.6** Write a program to implement a stack using linked lists and perform its common operations.

Program 4.2 implements a stack using linked lists in C. It uses the push (Example 4.4) and pop (Example 4.5) algorithms for realizing the common stack operations.

### Program 4.2 Implementation of stack using linked list

```
/*Program for demonstrating implementation of stacks using linked list*/
#include <stdio.h>
#include <conio.h>

struct stack /*Declaring the structure for stack elements*/
{
    int element; ←
    struct stack *next; /*Stack element pointing to another stack element*/
}*top;

void push(int); /*Declaring a function prototype for inserting an element into the stack*/
int pop(); /*Declaring a function prototype for removing an element from the stack*/
void display(); /*Declaring a function prototype for displaying the elements of a stack*/

void main()
{
    int num1, num2, choice;

    while(1)
    {
        clrscr();
        /*Creating an interactive interface for performing stack operations*/
        printf("Select a choice from the following:");
        printf("\n[1] Push an element into the stack");
        printf("\n[2] Pop out an element from the stack");
        printf("\n[3] Display the stack elements");
        printf("\n[4] Exit\n");
        printf("\n\tYour choice: ");
        scanf("%d", &choice);

        switch(choice)
        {
            case 1:
            {
                printf("\n\tEnter the element to be pushed into the stack: ");
                scanf("%d", &num1);
                Each stack element comprises of two fields, one for storing the stack element value and another for storing a pointer to the next element in the stack
            }
        }
    }
}
```

```

push(num1); /*Inserting an element*/
break;
}

case 2:
{
num2=pop(); /*Removing an element*/
printf("\n\t%d element popped out of the stack\n\t",num2);
getch();
break;
}

case 3:
{
display(); /*Displaying stack elements*/
getch();
break;
}

case 4:
exit(1);
break;

default:
printf("\nInvalid choice!\n");
break;
}
}

/*Push function*/
void push(int value)
{
    struct stack *ptr;
    ptr=(struct stack*)malloc(sizeof(struct stack)); /*Dynamically allocating
memory space to store stack element*/

    ptr->element=value; /*Assigning value to the newly allocated stack
element*/
}

/*Updating stack pointers*/
ptr->next=top;
top=ptr;
return;
}

/*Pop function*/
int pop()
{

```

*malloc function is used for dynamic or runtime reservation of space for new stack elements*

```

if(top==NULL) /*Checking whether the stack is empty*/
{
printf("\n\nSTACK is Empty.");
getch();
exit(1);
}
else
{
int temp=top->element; /* Retrieving the top element*/
top=top->next; /*Updating the stack pointer*/
return (temp); /*Returning the popped value*/
}

void display()
{
struct stack *ptr1=NULL;
ptr1=top;
printf("\n\nThe various stack elements are:\n");
while(ptr1!=NULL)
{
printf("%d\t",ptr1->element); /*Printing stack elements*/
ptr1=ptr1->next;
}
}

```

If the stack is empty then the stack pointer (top) will point at NULL

## Output

The output of the above program is same as the output of the program shown in Example 4.3.

## Program analysis

Key Statement	Purpose
<b>struct stack</b> { int element; struct stack *next; }*top;	Uses linked list to represent a stack and declares the stack pointer
<b>void push(int);</b> <b>int pop();</b> <b>void display();</b>	Declares prototypes for the functions that perform stack operations
<b>push(num1);</b>	Calls the <i>push()</i> function for inserting an element into the stack
<b>num2=pop();</b>	Calls the <i>pop()</i> function for removing an element from the stack
<b>display();</b>	Calls the <i>display()</i> function for displaying the stack elements
<b>ptr-&gt;element=value;</b> <b>ptr-&gt;next=top;</b> <b>top=ptr;</b>	Inserts an element at the top of the stack and updates the stack pointer

**Tip**

*It is advisable to check the overflow condition even with linked implementation of stacks, as in certain situations the available memory may also run out of space.*

**Tip**

*It is a good programming practice to release unused memory space so as to ensure efficient memory space utilization.*

**Check Point****1. What is array implementation of stacks?**

**Ans.** It involves allocation of fixed size array in the memory for storing stack elements. Both push and pop operations are performed on this array-implemented stack.

**2. What is linked implementation of stacks?**

**Ans.** It involves dynamic allocation of memory space at run time while performing stack operations.

## Solved Problems

**Problem 4.1** The contents of a stack S are as follows:

Stack (S)	99	2	44	8				
Index	0	1	2	3 ↑	4	5	6	7

The stack can store a maximum of eight elements and the top pointer currently points at index 3.

Show the stack contents and indicate the position of the top pointer after each of the following stack operations:

- (a) Push (S, 5)
- (b) Push (S, 7)
- (c) Pop (S)
- (d) Pop (S)
- (e) Pop (S)
- (f) Push (S, -1)

**Solution**

Push (S, 5)

*Step 1* Top = Top + 1 = 3 + 1 = 4

*Step 2* S [Top] = S [4] = 5

Stack contents

Stack (S):	99	2	44	8	5			
Index:	0	1	2	3	4 ↑	5	6	7

Push (S, 7)

Step 1 Top = Top + 1 = 4 + 1 = 5

Step 2 S [Top] = S [5] = 7

Stack contents

Stack (S):	99	2	44	8	5	7		
Index:	0	1	2	3	4	5↑	6	7

Pop (S)

Step 1 Item = S [Top] = S [5] = 7

Step 2 Top = Top - 1 = 5 - 1 = 4

Stack contents

Stack (S)	99	2	44	8	5			
Index	0	1	2	3	4↑	5	6	7

Pop (S)

Step 1 Item = S [Top] = S [4] = 5

Step 2 Top = Top - 1 = 4 - 1 = 3

Stack contents

Stack (S)	99	2	44	8				
Index	0	1	2	3↑	4	5	6	7

Pop (S)

Step 1 Item = S [Top] = S [3] = 8

Step 2 Top = Top - 1 = 3 - 1 = 2

Stack contents

Stack (S)	99	2	44					
Index	0	1	2↑	3	4	5	6	7

Push (S, -1)

Step 1 Top = Top + 1 = 2 + 1 = 3

Step 2 S [Top] = S [3] = -1

Stack contents

Stack (S)	99	2	44	-1				
Index	0	1	2	3↑	4	5	6	7

**Problem 4.2** Consider the following two states of a stack S:

State 1

Stack (S)	99	2	44	8	4			
Index	0	1	2	3	4↑	5	6	7

State 2

Stack (S)	99	5	-1	6				
Index	0	1	2	3↑	4	5	6	7

Write the series of push and pop operations that will transition the stack S from State 1 to State 2.

### Solution

- Step 1 Pop (S)
- Step 2 Pop (S)
- Step 3 Pop (S)
- Step 4 Pop (S)
- Step 5 Push (S, 5)
- Step 6 Push (S, -1)
- Step 7 Push (S, 6)

**Problem 4.3** Consider the following stack S:

Stack (S)	99	2	44	8		
Index	0	1	2	3↑	4	5

What will be result of the following statements?

```
i = 0;  
while (i != 5)  
{  
    push (S, i);  
    i = i + 1;  
}
```

### Solution

- Step 1 push (S, 0) → Top = 4, S [4] = 0
- Step 2 push (S, 1) → Top = 5, S [5] = 1
- Step 3 push (S, 2) → Top = 6 → Stack Overflow

**Problem 4.4** Consider the following stack S:

Stack (S)	99	2	44	8		
Index	0	1	2	3↑	4	5

What will be result of the following statements?

```
i = 0;  
while (i != 5)  
{  
    item = pop (S);  
    i = i + 1;  
}
```

### Solution

- Step 1 pop (S) item = 8, Top = 2
- Step 2 pop (S) item = 44, Top = 1
- Step 3 pop (S) item = 2, Top = 0
- Step 4 pop (S) → item = 99, Top = NULL
- Step 5 pop (S) → Stack Underflow

**Problem 4.5** The linked implementation of stacks eliminates the limitation of array implementation that restricts the number of stack elements below the array upper bound. So, is it true to say that a stack overflow condition can never occur with linked implementation of stacks? Justify your answer.

**Solution** No, it is not correct to say that a stack overflow condition can never occur with linked implementation of stacks. This is because; the system memory is also available till a certain extent. If we continue to push elements into a stack then a situation will arise when the system memory will run out of space causing the malloc function to return a NULL pointer. In this situation, the stack would be considered to be in an overflow state.

**Problem 4.6** Identify and correct the logical error in the following statement that performs pop operation on a stack S.

*item = S[--top];*

**Solution:** The statement,

*item = S[--top];*

contains the prefix operator --, which decrements the value of top by one. The new value now pointed by top is then popped and allocated to the variable item. However, this is not the top value of the stack. To pop out the top value of the stack we must use -- as the postfix operator, as shown below.

*item = S[top--];*

The above statement will first retrieve the top value of the stack and then decrement the top pointer by one.

## ➤➤➤ Summary

- ◆ A stack is a linear list in which elements are added and removed only from one end called top of the stack.
- ◆ Stacks are based on Last-In-First-Out or LIFO principle that means, the element added last into the list is the first one to be removed.
- ◆ Inserting an element into a stack is referred as push operation while removing an element from the stack is referred as pop operation.
- ◆ Stacks can be implemented through arrays or linked lists.
- ◆ The array implementation of stacks reserves a fixed amount of memory space in the form of an array for storing stack elements.
- ◆ The linked implementation of stacks uses dynamic memory management techniques for allocating the memory space for storing a new stack element at run time.
- ◆ Since linked implementation of stacks is based on dynamic memory allocation it is more efficient as compared to array-based implementation.
- ◆ The various application areas of stacks are expression evaluation, program control, recursion control, etc.

## ➤➤➤ Key Terms

- ◆ **Stack** It is a linear data structure in which items are added or removed only at one end.
- ◆ **Stack top** It is that end of the stack from where insertions and deletion of elements takes place.

- ◆ **LIFO** It stands for Last-In-First-Out i.e., the principle on which stacks are based.
- ◆ **Push** It refers to the task of inserting an element into the stack.
- ◆ **Pop** It refers to the task of deleting an element from the stack.
- ◆ **Array implementation** It refers to the realization of stack data structure using arrays.
- ◆ **Lined implementation** It refers to the realization of stack data structure using linked lists.

## Multiple-Choice Questions

- 4.1** Which of the following is not true for stacks?
- It is a linear data structure.
  - It allows insertion/deletion of elements only at one end
  - It is widely used by systems processes, such as compilation and program control
  - It is based on First-In-First-Out principle
- 4.2** Which of the following is not an example of a stack?
- Collection of tiles one over another
  - A set of bangles worn by a lady on her arm
  - A line up of people waiting for the bus at the bus stop
  - A pileup of boxes in a warehouse one over another
- 4.3** Tower of Hanoi can be regarded as a problem of which of the following data structures?
- Stack
  - Queue
  - Graph
  - Tree
- 4.4** Recursive function calls are executed using which of the following data structures?
- Stack
  - Queue
  - Graph
  - Tree
- 4.5** If 2, 1, 5, 8 are the stack contents with element 2 being at the top of the stack, then what will be the stack contents after following operations:
- Push (11)  
 Pop ()  
 Pop ()  
 Pop ()  
 Push(7)
- 11, 2, 1
  - 8, 11, 7
  - 7, 5, 8
  - 5, 8, 7
- 4.6** Which of the following is best suitable for storing a simple collection of employee records?
- Stack
  - Queue
  - Array
  - None of the above
- 4.7** If ‘top’ points at the top of the stack and ‘stack []’ is the array containing stack elements, then which of the following statements correctly reflect the push operation for inserting ‘item’ into the stack?
- $\text{top} = \text{top} + 1$ ;  $\text{stack}[\text{top}] = \text{item}$ ;
  - $\text{stack}[\text{top}] = \text{item}$ ;  $\text{top} = \text{top} + 1$ ;
  - $\text{stack}[\text{top}++] = \text{item}$ ;
  - Both (a) and (c) are correct

- 4.8** If ‘top’ points at the top of the stack and ‘stack []’ is the array containing stack elements, then which of the following statements correctly reflect the pop operation?
- (a)  $\text{top} = \text{top} - 1$ ;  $\text{item} = \text{stack}[\text{top}]$ ;
  - (b)  $\text{item} = \text{stack}[\text{top}]$ ;  $\text{top} = \text{top} - 1$ ;
  - (c)  $\text{item} = \text{stack}[-\text{top}]$ ;
  - (d) Both (b) and (c) are correct
- 4.9** If a pop operation is performed on an empty stack, then which of the following situations will occur?
- (a) Overflow
  - (b) Underflow
  - (c) Array out of bound
  - (d) None of the above
- 4.10** Which of the following is not a stack application?
- (a) Recursion control
  - (b) Expression evaluation
  - (c) Message queuing
  - (d) All of the above are stack applications

## Review Questions

- 4.1** What is a stack? Explain with examples.
- 4.2** Briefly describe the LIFO principle.
- 4.3** What is a top pointer? Explain its significance.
- 4.4** What are the different application areas of stack data structure?
- 4.5** Give any four real-life examples that principally resemble the stack data structure.
- 4.6** Explain the logical representation of stacks in memory with the help of an example.
- 4.7** Explain push and pop operations with the help of examples.
- 4.8** Deduce the contents of an empty stack after the execution of the following operations in sequence:  
Push (6)  
Push (8)  
Push (-1)  
Pop ()  
Push (7)  
Pop ()  
Pop ()
- 4.9** What will happen if we keep on pushing elements into a stack one after another?
- 4.10** What will happen if we continue to pop out elements from a stack one after another?
- 4.11** How are stacks implemented?
- 4.12** What is the advantage of linked implementation of stacks over array implementation?
- 4.13** What role does dynamic memory management techniques play in linked implementation of stacks?
- 4.14** Briefly explain the overflow and underflow conditions along with their remedies.
- 4.15** Can an overflow situation occur even with linked implementation of stacks that uses dynamic memory allocation techniques? Explain.

## Programming Exercises

- 4.1** Write a function in C to perform the push operation on an array-based stack that can store a maximum of 50 elements. Make sure that the overflow condition is adequately handled.
- 4.2** Write a function in C to perform the pop operation on a linked implementation of stack. Make sure that the underflow condition is adequately handled.
- 4.3** A stack contains  $N$  elements in it with  $TOP$  pointing at the top of the stack. It is required to reverse the order of occurrence of the  $N$  elements and store them in the same stack. Write a C program to achieve the same.
- 4.4** Modify the C program solution of Question 4.3 to store the  $N$  elements in sorted fashion with the largest element stored at the  $TOP$ .
- 4.5** A linked list implemented stack containing unknown number of elements is given. You are required to count the number of elements present in the stack. Write a function *count ()* in C that uses the pop operation to count the number of elements in the stack but does not actually remove the elements from the stack.
- 4.6** The Tower of Hanoi problem comprises of three towers with discs initially stacked on to the first tower. The requirement is to replicate the initial stack of discs into another tower while adhering to the following conditions:
- A larger disk can not be placed on a smaller disk
  - Only one disc can be shifted at a time
- Write a C program to find a solution to the above problem using stacks.
- 4.7** An input text string comprises the following:
- Letters
  - Digits
  - Special Characters

Write a program in C that accepts a text string from the user and stores its individual characters in three different stacks, i.e., L (for storing letters), D (for storing digits) and SC (for storing special characters). The program should terminate as soon as a ‘~’ symbol is encountered.

- 4.8** A stack is represented by the following structure declaration:

struct STACK

```
{  
    int ELEMENT[100];  
    int TOP;  
};
```

Write the push () and pop () functions in C for the above stack.

## Answers to Multiple-Choice Questions

- |         |         |         |         |          |
|---------|---------|---------|---------|----------|
| 4.1 (d) | 4.2 (c) | 4.3 (a) | 4.4 (a) | 4.5 (c)  |
| 4.6 (c) | 4.7 (a) | 4.8 (b) | 4.9 (b) | 4.10 (c) |

# 5

# QUEUES

Chapter outline

- 5.1 Introduction
- 5.2 Queues—Basic Concept
  - 5.2.1 Logical Representation of Queues
- 5.3 Queue Operations
  - 5.3.1 Insert
  - 5.3.2 Delete
  - 5.3.3 An Example of Queue Operations
- 5.4 Queue Implementation
  - 5.4.1 Array Implementation of Queues
  - 5.4.2 Linked Implementation of Queues

- 5.5 Circular Queues
- 5.6 Priority Queues
- 5.7 Double-Ended Queues

*Solved Problems*

*Summary*

*Key Terms*

*Multiple-Choice Questions*

*Review Questions*

*Programming Exercises*

*Answers to Multiple-Choice Questions*



## 5.1 INTRODUCTION

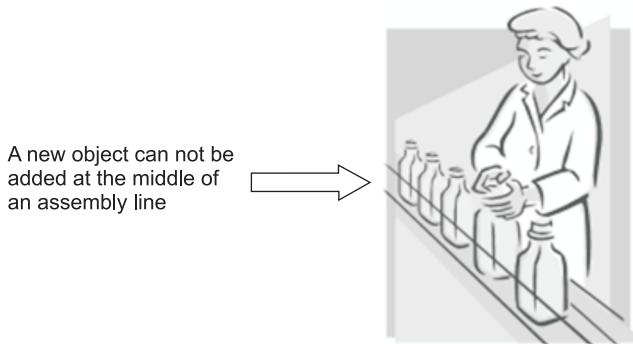
In Chapter 6, we learnt how stacks are different from arrays and how they store the data in memory. In this chapter, we will learn about another linear data structure called queues. While stacks allow insertion and deletion of data only at one end, queues restrict the insertion and deletion of data at two distinct ends. Just like stacks, queues also hold great significance in the implementation of key system processes such as CPU scheduling, resource sharing, etc.

## 5.2 QUEUES—BASIC CONCEPT

Queue is a linear data structure in which items are inserted at one end called ‘Rear’ and deleted from the other end called ‘Front’. Queues are based on the First-In-First-Out (FIFO) principle that means the data item that is inserted first in the queue is also the first one to be removed from the queue. We can relate queues to certain real-life objects and situations, as shown in Figs. 5.1 (a) and (b).



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

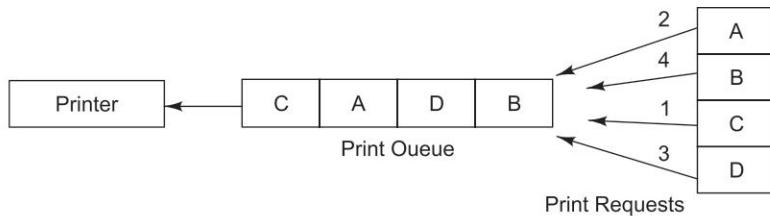


**Fig. 5.1(b)** An assembly line

As we can see in Fig. 5.1(a), a person can join a queue of waiting people only at its tail end while the person who joined the queue first becomes the first one to leave the queue. Likewise, the objects in an assembly line (Fig. 5.1(b)) also follow the same analogy. Another example of queue is the line up of vehicles at the toll booth. The vehicle that comes first to the toll booth leaves the booth first while

the vehicle that comes last leaves at the last; thus, observing FIFO principle. The concept of queue in data structures follows the same analogy as the queue of people or the queue of vehicles at toll booth.

An instance of queue implementation is a system of networked computers and resources where there are multiple users sharing one common printer amongst them. When a user on the network sends a print request, the request is added to the print queue. When the request reaches at the front it gets executed and is removed from the print queue. This ensures orderly execution of users' print requests. Figure 5.2 depicts this scenario.



**Fig. 5.2** Queue implementation

As we can see in the Fig. 5.2, four users, A, B, C and D share a single printer on the network. When a user sends a print request, it gets added to the print queue. User C sends the first print request, thus it gets added at the front of the print queue. Similarly, print requests from other users are also added in the queue as per their request order. Now, based on FIFO analogy, the printer will first process print request of user C, followed by A, D and user B at the last.



### Mind Jog

*What is the meaning of enqueue and dequeue?*

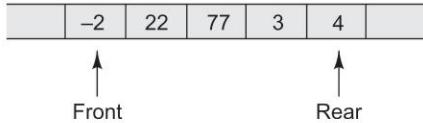
*All queue insertions are termed as enqueue while all queue deletions are termed as dequeue.*

## 5.2.1 Logical Representation of Queues

Just like their real world counterparts, queues appear as a group of elements stored at contiguous locations in memory. Each successive insert operation adds an element at the rear end of the queue while each delete operation removes an element from the front end of the queue. The location of the front and rear ends are marked by two distinct pointers called *front* and *rear*.

Figure 5.3 shows the logical representation of queues in memory.

As we can see in the above figure, there are five elements in the queue with -2 at the front and 4 at the rear.



**Fig. 5.3** Logical representation of queues



**Note** The logical representation of queues showing queue elements stored at contiguous memory location might be true in case of their array implementation but the same might not be true in case of their linked implementation, as we shall study later in this chapter.

## 5.3 QUEUE OPERATIONS

There are two key operations associated with the queue data structure: insert and delete. The insert operation adds an element at the rear end of the queue while the delete operation removes an element from the front end of the queue. Figures 5.4 (a) and (b) depict the insert and delete operations on a queue.



### Check Point

#### 1. What is a queue?

**Ans:** Queue is a linear data structure in which items are inserted at one end called ‘Rear’ and deleted from the other end called ‘Front’.

#### 2. What is FIFO?

**Ans:** First-In-First-Out (FIFO) principle specifies that the data item that is inserted first in the queue is also the first one to be removed from the queue.

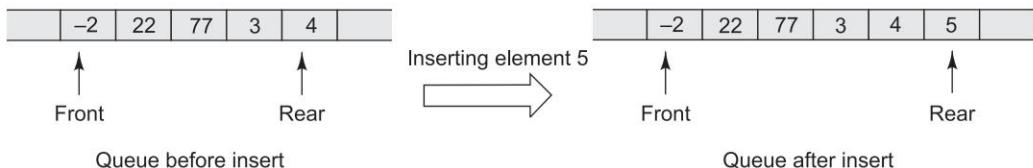


Fig. 5.4(a) Insert operation

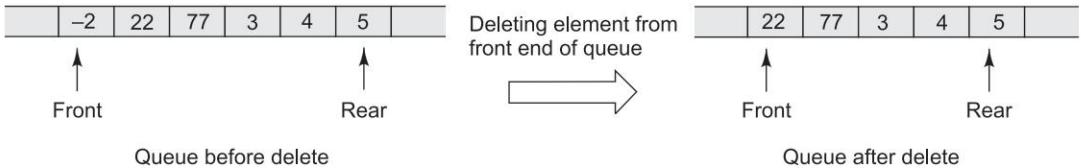


Fig. 5.4(b) Delete operation



**Note** The front and rear indicators are quite significant in queue’s context as they point at entry and exit gateways of the queue.

#### 1. Insert

As we can see in Fig. 5.4(a), the insert operation involves the following subtasks:

- Receiving the element to be inserted.
- Incrementing the queue pointer, *rear*.
- Storing the received element at new location of rear.

Thus, the programmatic realization of the insert operation requires implementation of the above mentioned subtasks, as we shall see later in this chapter.



**Tip** Before inserting a new element, it needs to be checked whether the queue is already full. If the queue is already full then a new element cannot be added at its rear end. Such a situation is termed as queue overflow.

**2. Delete** As we can see in Fig. 5.4 (b), the delete operation involves the following subtasks:  
Retrieving or removing the element from the front end of the queue.

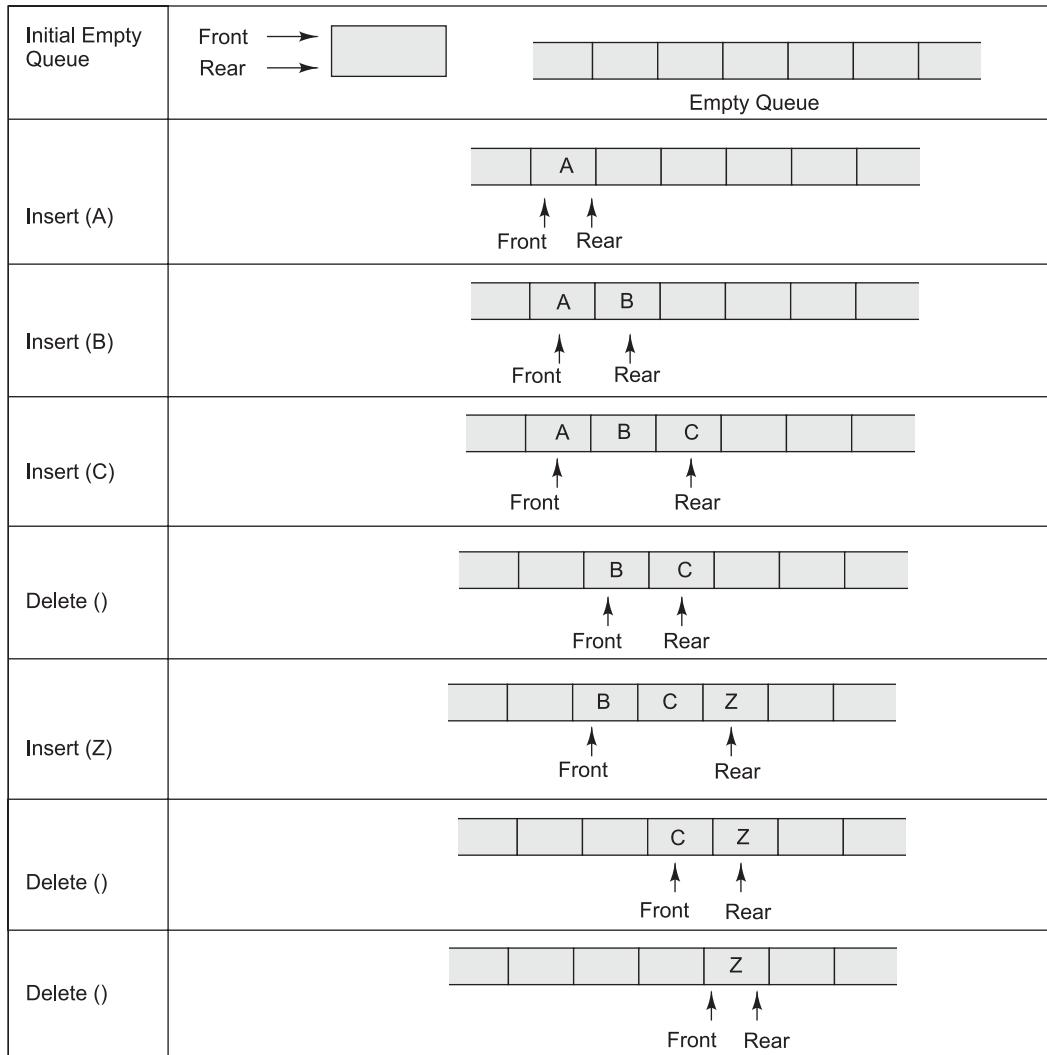
Incrementing the queue pointer, *front*, to make it point to the next element in the queue.

Thus, the programmatic realization of the delete operation requires implementation of the above mentioned subtasks, as we shall see later in this chapter.

Some queue implementations require the deleted element to be returned back to the calling function while others may simply focus on updating the front pointer and ignore the deleted element all together. The choice of a particular type of implementation depends solely on the programming situation at hand.

### An example of queue operations

Figure 5.5 shows how the queue contents change after a series of insert and delete operations.



**Fig. 5.5 Queue operations**

We can see in Fig. 5.5 how queue contents change with insert and delete operations occurring at rear and front ends respectively.



**Tip** Before deleting an element, it needs to be checked whether the queue is already empty. If the queue is already empty then there is nothing to be deleted. Such a situation is termed as queue underflow.

## 5.4 QUEUE IMPLEMENTATION

Queue implementation involves choosing the data storage mechanism for storing queue elements and implementing methods for performing the two queue operations, insert and delete. Like stacks, we can implement queues by using arrays or linked lists. The advantages or disadvantages of array or linked implementations of queues are the same that are associated with such types of data structures. However, both types of implementation have their own usage in specific situations.

### 5.4.1 Array Implementation of Queues

The array implementation of queues involves allocation of fixed size array in the memory. Both queue operations (insert and delete) are performed on this array with a constant check being made to ensure that the array does not go out of bounds.

**Insert Operation** The insert operation involves checking whether or not the queue pointer *rear* is pointing at the upper bound of the array. If it is not, *rear* is incremented by 1 and the new item is added at the end of the queue.

**Example 5.1** Write an algorithm to realize the insert operation under array implementation of queues.

```
insert(queue[MAX], element, front, rear)
Step 1: Start
Step 2: If front = NULL goto Step 3 else goto Step 6
Step 3: front = rear = 0
Step 4: queue[front]=element
Step 5: Goto Step 10
Step 6: if rear = MAX-1 goto Step 7 else goto Step 8
Step 7: Display the message, "Queue is Full" and goto Step 10
Step 8: rear = rear +1
Step 9: queue[rear] = element
Step 10: Stop
```



### Mind Jog

#### What is a bounded queue?

*It is a queue restricted to a fixed number of elements.*



### Check Point

#### 1. What is a queue insert operation?

**Ans.** The queue insert operation adds an element at the rear end of the queue.

#### 2. What is a queue delete operation?

**Ans.** The queue delete operation removes an element from the front end of the queue.



### Tip

The above code shows insertion of an integer type element into the queue. However, we may store other built-in or user-defined type elements in the queue as per our own requirements.

**Delete Operation** The delete operation involves checking whether or not the queue pointer *front* is already pointing at NULL (empty queue). If it is not, the item that is being currently pointed is removed from the queue (array) and the *front* pointer is incremented by 1.

**Example 5.2** Write an algorithm to realize the delete operation under array implementation of queues.

```
delete(queue[MAX], front, rear)
Step 1: Start
Step 2: If front = NULL and rear = NULL goto Step 3 else goto Step 4
Step 3: Display the message, "Queue is Empty" and goto Step 10
Step 4: if front != NULL and front = rear goto Step 5 else goto Step 8
Step 5: Set i = queue[front]
Step 6: Set front = rear = -1
Step 7: Return the deleted element i and goto Step 10
Step 8: Set i = queue[front]
Step 9: Return the deleted element i
Step 10: Stop
```

### Implementation

**Example 5.3** Write a program to implement a queue using arrays and perform its common operations.

Program 5.1 implements a queue using arrays in C. It uses the insert (Example 5.1) and delete (Example 5.2) functions for realizing the common queue operations.

#### Program 5.1 Implementation of queue

```
/*Program for demonstrating implementation of queues using arrays*/
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>

int queue[100]; /*Declaring a 100 element queue array*/
int front=-1; /*Declaring and initializing the front pointer*/
int rear=-1; /*Declaring and initializing the rear pointer*/
```

If we do not initialize the front and rear variables then they may continue to store garbage value which may lead to erroneous results

```
void insert(int); /*Declaring a function prototype for inserting an element into the queue*/
int del(); /*Declaring a function prototype for removing an element from the queue*/
void display(); /*Declaring a function prototype for displaying the queue elements*/

void main()
```

```

{
int choice;
int num1=0,num2=0;
while(1) ←————— Here, while (1) signifies an infinite looping condition that'll continue to execute the statements within until a jump statement is encountered
{
/*Creating an interactive interface for performing queue operations*/
printf("\nSelect a choice from the following:");
printf("\n[1] Add an element into the queue");
printf("\n[2] Remove an element from the queue");
printf("\n[3] Display the queue elements");
printf("\n[4] Exit\n");
printf("\n\tYour choice: ");
scanf("%d",&choice);

switch(choice)
{
case 1:
{
printf("\n\tEnter the element to be added to the queue: ");
scanf("%d",&num1);
insert(num1); /*Adding an element*/
break;
}

case 2:
{
num2=del(); /*Removing an element*/
if(num2== -9999)
;
else
    printf("\n\t%d element removed from the queue\n\t",num2);
getch();
break;
}

case 3:
{
display(); /*Displaying queue elements*/
getch();
break;
}

case 4:
exit(1);
break;
}

default: ←————— Default blocks are always advisable in switch-case constructs as it allows handling of incorrect input values
printf("\nInvalid choice!\n");
break;
}

```

```

    }
}

}

/*Insert function*/
void insert(int element)
{
if(front===-1) /*Adding element in an empty queue*/
{
front = rear = front+1;
queue[front] = element;
return;
}

if(rear==99) /*Checking whether the queue is full*/
{
printf("Queue is Full.\n");
getch();
return;
}
rear=rear+1; /*Incrementing rear pointer*/
queue[rear]=element; /*Inserting the new element*/
}

/*Delete function*/
int del()
{
int i;
if(front===-1 && rear===-1) /*Checking whether the queue is empty*/
{
printf("\n\tQueue is Empty.\n");
getch();
return (-9999);
}
if(front!=-1 && front==rear) /*Checking whether the queue has only one
element left*/
{
i=queue[front];
front=-1;
rear=-1;
return(i);
}
return(queue[front++]); /*Returning the front most element and incrementing
the front pointer*/
}

/*Display function*/
void display()
{

```

*The upper bound of 99 shows that this queue can store a maximum of 100 elements*

*Here, NULL value is represented by -1*

```
int i;
if(front== -1)
{
    printf("\n\tQueue is Empty!\n");
    return;
}

printf("\n\tThe various queue elements are:\n");
for(i=front;i<=rear;i++)
    printf("\t%d",queue[i]); /*Printing queue elements*/
}
```

## Output

Select a choice from the following:

- [1] Add an element into the queue
- [2] Remove an element from the queue
- [3] Display the queue elements
- [4] Exit

Your choice: 3

Queue is Empty!

Select a choice from the following:

- [1] Add an element into the queue
- [2] Remove an element from the queue
- [3] Display the queue elements
- [4] Exit

Your choice: 1

Enter the element to be added to the queue: 1

Select a choice from the following:

- [1] Add an element into the queue
- [2] Remove an element from the queue
- [3] Display the queue elements
- [4] Exit

Your choice: 1

Enter the element to be added to the queue: 2

Select a choice from the following:

- [1] Add an element into the queue
- [2] Remove an element from the queue
- [3] Display the queue elements
- [4] Exit

```
Your choice: 1
```

```
Enter the element to be added to the queue: 3
```

```
Select a choice from the following:
```

- [1] Add an element into the queue
- [2] Remove an element from the queue
- [3] Display the queue elements
- [4] Exit

```
Your choice: 3
```

```
The various queue elements are:
```

```
1 2 3
```

```
Select a choice from the following:
```

- [1] Add an element into the queue
- [2] Remove an element from the queue
- [3] Display the queue elements
- [4] Exit

```
Your choice: 2
```

```
1 element removed from the queue
```

```
Select a choice from the following:
```

- [1] Add an element into the queue
- [2] Remove an element from the queue
- [3] Display the queue elements
- [4] Exit

```
Your choice: 2
```

```
2 element removed from the queue
```

```
Select a choice from the following:
```

- [1] Add an element into the queue
- [2] Remove an element from the queue
- [3] Display the queue elements
- [4] Exit

```
Your choice: 2
```

```
3 element removed from the queue
```

```
Select a choice from the following:
```

- [1] Add an element into the queue
- [2] Remove an element from the queue
- [3] Display the queue elements

```
[4] Exit
Your choice: 3
Queue is Empty!
```

### Program analysis

Key Statement	Purpose
<b>int queue[100];</b>	Declares an array to represent a queue
<b>void insert(int);</b> <b>int del();</b> <b>void display();</b>	Declares prototypes for the functions that perform queue operations
<b>insert(num1);</b>	Calls the <i>insert()</i> function for inserting an element into the queue
<b>num2=del();</b>	Calls the <i>del()</i> function for deleting an element from the queue
<b>display();</b>	Calls the <i>display()</i> function for displaying the queue elements
<b>rear=rear+1;</b> <b>queue[rear]=element;</b>	Inserts an element at the end of the queue and updates the rear pointer
<b>if(front== -1)</b>	Checks whether or not the queue is empty

### 5.4.2 Linked Implementation of Queues

The linked implementation of queues involves dynamically allocating memory space at run time while performing queue operations. Since, the allocation of memory space is dynamic, the queue consumes only that much amount of space as is required for holding its data elements. This is contrary to array-implemented queues which continue to occupy a fixed memory space even if there are no elements present. Thus, linked implementation of queues based on dynamic memory allocation technique prevents wastage of memory space.



**Note** *The linked implementation of queues is based on dynamic memory management techniques, which allow allocation and de-allocation of memory space at runtime.*

**Insert Operation** The insert operation under linked implementation of queues involves the following tasks:

1. Reserving memory space of the size of a queue element in memory
2. Storing the added (inserted) value at the new location
3. Linking the new element with existing queue
4. Updating the *rear* pointer

**Example 5.4** Write an algorithm to realize the insert operation under linked implementation of queues.

```

insert(structure queue, value, front, rear)
Step 1: Start
Step 2: Set ptr=(struct queue*)malloc(sizeof(struct queue)), to reserve a
block of memory for the new queue node and assign its address to pointer ptr
Step 3: Set ptr->element=value, to copy the inserted value into the new node
Step 4: if front = NULL goto Step 5 else goto Step 7
Step 5: Set front = rear = ptr
Step 6: Set ptr->next=NULL and goto Step 10
Step 7: Set rear->next=ptr
Step 8: Set ptr->next=NULL
Step 9: Set rear = ptr
Step 10: Stop

```

**Delete Operation** The delete operation under linked implementation of queues involves the following tasks:

1. Checking whether the queue is empty.
2. Retrieving the front most element of the queue.
3. Updating the *front* pointer.
4. Returning the retrieved (removed) value

**Example 5.5** Write an algorithm to realize the delete operation under linked implementation of queues.

```

delete(structure queue, front, rear)
Step 1: Start
Step 2: if front = NULL goto Step 3 else goto Step 4
Step 3: Display message, "Queue is Empty" and goto Step 7
Step 4: Set i = front->element
Step 5: Set front = front->next
Step 6: Return the deleted element i
Step 7: Stop

```



**Tip** *It is a good programming practice to release unused memory space so as to ensure efficient memory space utilization.*

## Implementation

**Example 5.6** Write a program to implement a queue using linked lists and perform its common operations.

Program 5.2 implements a queue using linked lists in C. It uses the insert (Example 5.4) and delete (Example 5.5) functions for realizing the common queue operations.

### Program 5.2 Implementation of queue

```

/*Program for implementing queue using linked list*/
#include<stdio.h>
#include<conio.h>

```

```

#include<stdlib.h>

struct queue /*Declaring the structure for queue elements*/
{
    int element;
    struct queue *next; /*Queue element pointing to another queue element*/
};

struct queue *front=NULL;
struct queue *rear = NULL;

void insert(int); /*Declaring a function prototype for adding an element into the queue*/
int del(); /*Declaring a function prototype for removing an element from the queue*/
void display(void); /*Declaring a function prototype for displaying the elements of the queue*/

void main()
{
    int num1, num2, choice;
    while(1)
    {
        /*Creating an interactive interface for performing queue operations*/
        printf("\n\nSelect an option\n");
        printf("\n1 - Insert an element into the Queue");
        printf("\n2 - Remove an element from the Queue ");
        printf("\n3 - Display all the elements in the Queue");
        printf("\n4 - Exit");

        printf("\n\nEnter your choice: ");
        scanf("%d", &choice);

        switch(choice)
        {
            case 1:
            {
                printf("\nEnter the element to be inserted into the queue ");
                scanf("%d", &num1);
                insert(num1); /*Adding an element*/
                break;
            }

            case 2:
            {
                num2=del(); /*Removing an element*/
                if(num2==9999)

```

*Each queue element comprises of two fields, one for storing the stack element value and another for storing a pointer to the next element in the queue*

```

printf("\n\tQueue is empty!!");
else
printf("\n\t%d element removed from the queue\n\t", num2);
getch();
break;
}

case 3:
{
display(); /*Displaying queue elements*/
getch();
break;
}

case 4:
{
exit(1);
break;
}

default:
{
printf("\nInvalid choice.");
getch();
break;
}
}
}
}

/*Insert function*/
void insert(int value)
{
    struct queue *ptr = (struct queue*)malloc(sizeof(struct queue)); /*Dynamically
declaring a queue element*/

    ptr->element = value; /*Assigning value to the newly allocated queue
element*/

    if(front==NULL) /*Adding element in an empty queue*/
    {
        front = rear = ptr;
        ptr->next=NULL;
    }

    /*Updating queue pointers*/
    else

```

*malloc function is used for dynamic or runtime reservation of space for new queue elements*

```

{
rear->next = ptr;
ptr->next = NULL;
rear = ptr;
}
}

/*Delete function*/
int del()
{
int i;

if(front==NULL) /*Checking whether the queue is empty*/
return(-9999);

else
{
i=front->element; /*removing element from the start*/
front = front->next;
return(i);
}
}

/*Display function*/
void display()
{
struct queue *ptr=front;
if(front==NULL)
{
printf("\n\tQueue is Empty!!");
return;
}

else
{
printf("\nElements present in the Queue are:\n");
/*Printing queue elements*/
while(ptr!=rear)
{
printf(<<\t%d>>,ptr->element);
ptr=ptr->next;
}
printf("\t%d",rear->element);
}
}
}

```

*If the queue is empty then the stack pointer (front) will point at NULL*

## Output

The output of the above program is same as the output of the program shown in Example 5.3.

## Program analysis

Key Statement	Purpose
<code>struct queue { int element; struct queue *next; };</code>	Uses linked list to represent a queue
<code>struct queue *front=NULL; struct queue *rear = NULL;</code>	Declares queue pointers
<code>void insert(int); int del(); void display(void);</code>	Declares prototypes for the functions that perform queue operations
<code>insert(num1);</code>	Calls the <i>insert()</i> function for inserting an element into the queue
<code>num2=del();</code>	Calls the <i>del()</i> function for deleting an element from the queue
<code>display();</code>	Calls the <i>display()</i> function for displaying the queue elements
<code>rear-&gt;next = ptr; ptr-&gt;next = NULL; rear = ptr;</code>	Inserts an element at the end of the queue and updates the rear pointer
<code>if(front==NULL)</code>	Checks whether or not the queue is empty

## 5.5 CIRCULAR QUEUES

A circular queue is a queue whose start and end locations are logically connected with each other. That means, the start location comes after the end location. If we continue to add elements in a circular queue till its end location, then after the end location has been filled, the next element will be added at the beginning of the queue. Circular queues remove one of the main disadvantages of array implemented queues in which a lot of memory space is wasted due to inefficient utilization.

Figure 5.6 shows the logical representation of a circular queue.

As we can see in Fig. 5.6, the start location of the queue comes after its end location. Thus, if the queue is filled till its capacity, i.e., the end location, then the start location will be checked for space, and if it is empty, the new element will be added there. Figure 5.7 shows the different states of a circular queue during insert and delete operations.



### Check Point

#### 1. What is array implementation of queues?

**Ans.** It involves allocation of fixed size array in the memory for storing queue elements. Both insert and delete operations are performed on this array.

#### 2. What is linked implementation of queues?

**Ans.** It involves dynamic allocation of memory space at run time while performing queue operations.

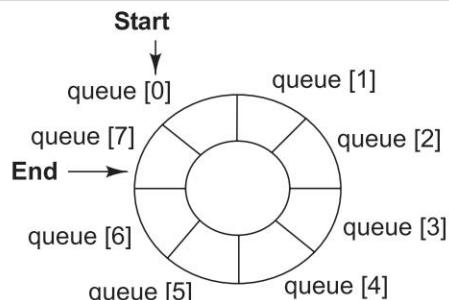
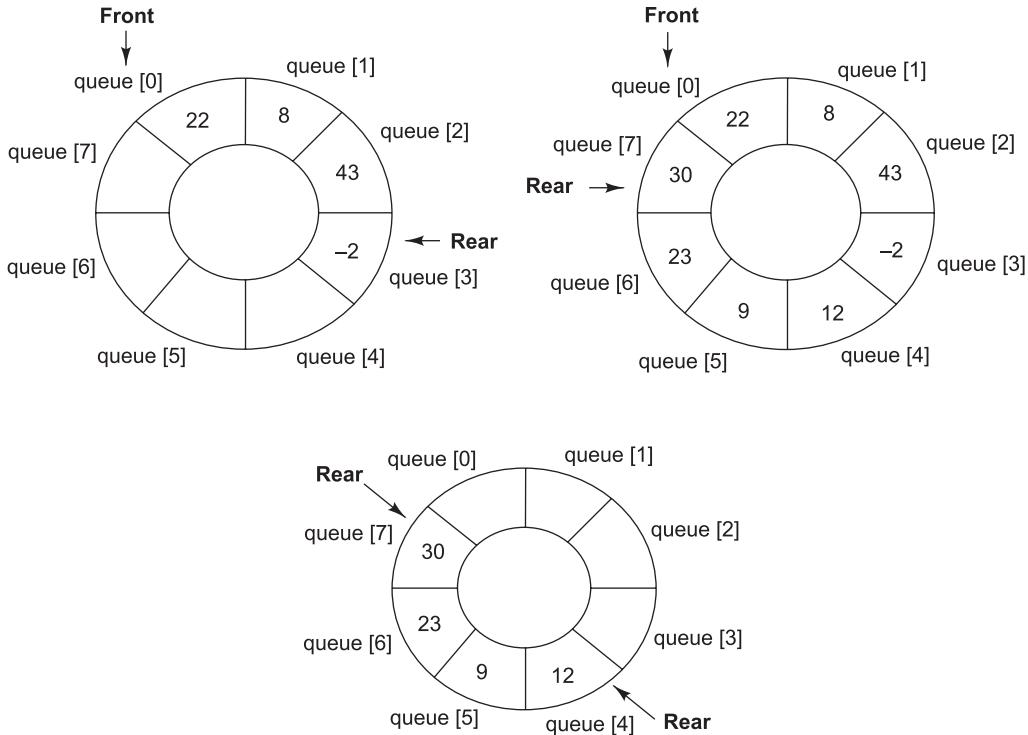


Fig. 5.6 Circular queue



**Fig. 5.7** Inserting and deleting elements in a circular queue

**Insert Operation** The insert operation for array implemented circular queues involves the following tasks:

1. Checking whether the queue is already full.
2. Updating the *rear* pointer.
  - (a) If the queue is empty, set *front* and *rear* to point to the first location in the queue.
  - (b) If *rear* is pointing at the last location of the queue, set *rear* to point to the first location in the queue.
  - (c) If none of the above situations exist, simply increment the *rear* pointer by 1.
3. Inserting the new element at the *rear* location.

**Example 5.7** Write an algorithm to realize the insert operation for array-implemented circular queues.

```

insert(queue[MAX], front, rear, element)
Step 1: Start
Step 2: if (front = 0 and rear = MAX-1) OR front = rear+1 goto Step 3 else
goto Step 4
Step 3: Display message, "Queue is Full" and goto Step 10
Step 4: if front = NULL goto Step 5 else goto Step 6
  
```

```

Step 5: Set front = rear = 0
Step 6: if rear = MAX-1 goto Step 7 else goto Step 8
Step 7: Set rear = 0
Step 8: Set rear = rear + 1
Step 9: Set queue[rear] = element
Step 10: Stop

```

**Delete Operation** The delete operation for array implemented circular queues involves the following tasks:

1. Checking whether the queue is already empty.
2. Retrieving the element at the front of the queue.
3. Updating the *front* pointer.
  - (a) If the queue has only one element left, set *front* and *rear* to point to NULL.
  - (b) If *front* is pointing at the last location of the queue, set *front* to point to the first location in the queue.
  - (c) If none of the above situations exist, simply increment the *front* pointer by 1.
4. Returning the element retrieved from the *front* location.

**Example 5.8** Write an algorithm to realize the delete operation for array-implemented circular queues.

```

delete(queue[MAX], front, rear)
Step 1: Start
Step 2: if front = NULL goto Step 3 else goto Step 4
Step 3: Display message, "Queue is Empty" and goto Step 13
Step 4: Set i = queue[front]
Step 5: if front = rear goto Step 6 else goto Step 8
Step 6: Set front = rear = NULL
Step 7: Return the deleted element i and go to Step 13
Step 8: if front = MAX-1 goto Step 9 else goto Step 11
Step 9: Set front = 0
Step 10: Return the deleted element i and go to Step 13
Step 11: Set front = front + 1
Step 12: Return the deleted element i
Step 13: Stop

```

## Implementation

**Example 5.9** Write a program to implement a circular queue using arrays and perform its common operations.

Program 5.3 implements a circular queue using arrays in C. It uses the insert (Example 5.7) and delete (Example 5.8) functions for realizing the common queue operations.

### Program 5.3 Implementation of a circular queue using arrays

```

/*Program for demonstrating implementation of circular queues using arrays*/
#include <stdio.h>
#include <conio.h>

```

```

#include <stdlib.h>

int queue[5]; /*Declaring a 5 element queue array*/
int front=-1; /*Declaring and initializing the front pointer*/
int rear=-1; /*Declaring and initializing the rear pointer*/

void insert(int); /*Declaring a function prototype for inserting an element
into the circular queue*/
int del(); /*Declaring a function prototype for removing an element from
the circular queue*/
void display(); /*Declaring a function prototype for displaying the queue
elements*/

void main()
{
    int choice;
    int num1=0,num2=0;
    while(1)
    {
        /*Creating an interactive interface for performing queue operations*/
        printf("\nSelect a choice from the following:");
        printf("\n[1] Add an element into the queue");
        printf("\n[2] Remove an element from the queue");
        printf("\n[3] Display the queue elements");
        printf("\n[4] Exit\n");
        printf("\n\tYour choice: ");
        scanf("%d",&choice);

        switch(choice)
        {
            case 1:
            {
                printf("\n\tEnter the element to be added to the queue: ");
                scanf("%d",&num1);
                insert(num1); /*Adding an element*/
                break;
            }

            case 2:
            {
                num2=del(); /*Removing an element*/
                if(num2== (-9999))
                ;
                else
                    printf("\n\t%d element removed from the queue\n\t",num2);
                getch();
                break;
            }
        }
    }
}

```

```

case 3:
{
display(); /*Displaying queue elements*/
getch();
break;
}

case 4:
exit(1);
break;

default:
printf("\nInvalid choice!\n");
break;
}
}

/*Insert function*/
void insert(int element)
{
if((front==0 && rear ==4) || front==rear+1)
{
    printf("\tQueue is Full. Element %d cannot be added into the queue\n",
n",element);
    getch();
    return;
}

if(front==-1) /*Adding element in an empty queue*/
{
    front=0;
    rear=0;
}
else if(rear==4)
    rear=0; /*Setting rear pointer to start of queue*/
else
    rear=rear+1; /*Incrementing rear pointer*/

queue[rear]=element; /*Inserting the new element*/
}

/*Delete function*/
int del()
{
    int i;
    if(front==-1) /*Checking whether the queue is empty*/
    {
        printf("\n\tQueue is Empty.\n");
    }
}

```

```

getch();
return (-9999);
}

i=queue[front]; /*Retrieving the element at the front of the queue*/

if(front==rear) /*Checking whether the queue has only one element left*/
{
front=-1;
rear=-1;
return(i);
}
else if(front==4)
{
front=0; /*Setting the front pointer to start of queue*/
return(i);
}
else
{
front=front+1; /*Incrementing the front pointer*/
return(i);
}

/*Display function*/
void display()
{
int i;
if(front== -1)
{
printf("\n\tQueue is Empty!\n");
return;
}

printf("\n\tThe various queue elements are:\n");
i=front;
while(i!=rear)
{
    printf("\t%d",queue[i]); /*Printing queue elements*/
    if(i==4)
        i=0;
    else
        i=i+1;
}
    printf("\t%d\n",queue[i]); /*Printing the last element in the queue*/
}

```

## Output

```
Select a choice from the following:  
[1] Add an element into the queue  
[2] Remove an element from the queue  
[3] Display the queue elements  
[4] Exit
```

```
Your choice: 1
```

```
Enter the element to be added to the queue: 1
```

```
Select a choice from the following:  
[1] Add an element into the queue  
[2] Remove an element from the queue  
[3] Display the queue elements  
[4] Exit
```

```
Your choice: 1
```

```
Enter the element to be added to the queue: 2
```

```
Select a choice from the following:  
[1] Add an element into the queue  
[2] Remove an element from the queue  
[3] Display the queue elements  
[4] Exit
```

```
Your choice: 1
```

```
Enter the element to be added to the queue: 3
```

```
Select a choice from the following:  
[1] Add an element into the queue  
[2] Remove an element from the queue  
[3] Display the queue elements  
[4] Exit
```

```
Your choice: 1
```

```
Enter the element to be added to the queue: 4
```

```
Select a choice from the following:  
[1] Add an element into the queue  
[2] Remove an element from the queue  
[3] Display the queue elements  
[4] Exit
```

```
Your choice: 1
```

```
Enter the element to be added to the queue: 5
```

```
Select a choice from the following:  
[1] Add an element into the queue  
[2] Remove an element from the queue  
[3] Display the queue elements  
[4] Exit
```

Your choice: 3

The various queue elements are:  
1 2 3 4 5

```
Select a choice from the following:  
[1] Add an element into the queue  
[2] Remove an element from the queue  
[3] Display the queue elements  
[4] Exit
```

Your choice: 1

Enter the element to be added to the queue: 6  
Queue is Full. Element 6 cannot be added into the queue

```
Select a choice from the following:  
[1] Add an element into the queue  
[2] Remove an element from the queue  
[3] Display the queue elements  
[4] Exit
```

Your choice: 2

1 element removed from the queue

```
Select a choice from the following:  
[1] Add an element into the queue  
[2] Remove an element from the queue  
[3] Display the queue elements  
[4] Exit
```

Your choice: 3

The various queue elements are:  
2 3 4 5

```
Select a choice from the following:  
[1] Add an element into the queue  
[2] Remove an element from the queue  
[3] Display the queue elements  
[4] Exit
```

Your choice: 1

Enter the element to be added to the queue: 6

Select a choice from the following:

- [1] Add an element into the queue
- [2] Remove an element from the queue
- [3] Display the queue elements
- [4] Exit

Your choice: 3

The various queue elements are:

2 3 4 5 6

We can observe in the above output that a circular queue makes the best utilization of available memory space by logically connecting the start and end locations.

### Program analysis

Key Statement	Purpose
<code>if((front==0 &amp;&amp; rear ==4)    front==rear+1)</code>	Checks whether the circular queue is full or not
<code>i=front; while(i!=rear)</code>	Traverses the elements of the circular queue

## 5.6 PRIORITY QUEUES

Priority queue is a type of queue in which each element is assigned certain priority such that the order of deletion of elements is decided by their associated priorities. The order of processing or deletion of elements in a priority queue is decided by the following rules:

1. An element with highest priority is deleted before all other elements of lower priority.
2. If two elements have the same priority then they are deleted as per the order in which they were added into the queue (i.e., First-In-First-Out).

The implementation of priority queues may follow different approaches. For instance, elements may be added arbitrarily into the queue and deleted as per their priority values or, the elements may be sorted as per their priorities at the time of their insertion itself, and deleted in a sequential fashion. We'll be following the later approach for implementing priority queues.

The structure of a priority queue needs to be defined in such a manner that each queue node is able to store both its value as well as its priority information. The following C structure defines the node of a priority queue:



### Check Point

#### 1. What is a circular queue?

**Ans.** A circular queue is a queue whose start and end locations are logically connected with each other.

#### 2. What is the advantage of circular queue?

**Ans.** The implementation of circular queues ensures efficient utilization of memory space in comparison to normal queues.

```

struct queue /*Node of a priority queue*/
{
    int element;
    int priority;
    struct queue *next; /*Pointer to the next queue node*/
};

```

## **Implementation**

**Example 5.10** Write a program to implement a priority queue using linked lists and perform its common operations.

Program 5.4 implements a priority queue using linked lists in C.

**Program 5.4** *Implementation of a priority queue using linked lists*

```

/*Program for implementing priority queue using linked list*/
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>

struct queue /*Declaring the structure for queue node*/
{
    int element;
    int priority;
    struct queue *next; /*Pointer to the next queue node*/
};

struct queue *front=NULL;

void insert(int,int); /*Declaring a function prototype for inserting an
element into the queue*/
int del(); /*Declaring a function prototype for deleting an element from
the queue*/
void display(void); /*Declaring a function prototype for displaying the
queue elements along with their priority values*/

void main()
{
    int num1, num2, pr, choice;
    while(1)
    {
        /*Creating an interactive interface for performing queue operations*/
        printf("\n\nSelect an option\n");
        printf("\n1 - Insert an element into the Queue");
        printf("\n2 - Remove an element from the Queue ");
        printf("\n3 - Display all the elements in the Queue");
        printf("\n4 - Exit");

```

```
printf("\n\nEnter your choice: ");
scanf("%d", &choice);

switch(choice)
{
case 1:
{
printf("\nEnter the element to be inserted into the queue ");
scanf("%d", &num1);
printf("\nEnter the priority of %d ", num1);
scanf("%d", &pr);
insert(num1, pr); /*Inserting an element*/
break;
}

case 2:
{
num2=del(); /*Deleting an element*/
if (num2== -9999)
printf("\n\tQueue is empty!!");
else
printf("\n\t%d element removed from the queue\n\t", num2);
getch();
break;
}

case 3:
{
display(); /*Displaying queue elements*/
getch();
break;
}

case 4:
{
exit(1);
break;
}

default:
{
printf("\nInvalid choice.");
getch();
break;
}
}
}
```

```

/*Insert Function*/
void insert(int value,int p)
{
    struct queue *temp;
    struct queue *ptr = (struct queue*)malloc(sizeof(struct queue)); /*Dynamically
declaring a queue element*/

    ptr->element = value; /*Assigning value to the newly allocated queue
element*/
    ptr->priority=p; /*Assigning priority to the newly allocated queue
element*/

    /*Checking if the newly allocated queue element needs to be inserted at
the front*/
    if(front==NULL||ptr->priority<front->priority)
    {
        ptr->next=front;
        front = ptr;
    }
    else
    {
        temp=front;

        /*Adding the newly allocated queue element as per priority*/
        while(temp->next!=NULL && temp->next->priority<=ptr->priority)
            temp=temp->next;
        ptr->next = temp->next;
        temp->next = ptr;
    }
}

/*Delete Function*/
int del()
{
    int i;

    if(front==NULL) /*Checking whether the queue is empty*/
    return(-9999);

    else
    {
        i=front->element; /*Removing elements as per priority*/
        front = front->next;
        return(i);
    }
}

/*Display Function*/
void display()

```

```
{  
    struct queue *ptr=front;  
    if(front==NULL)  
    {  
        printf("\n\tQueue is Empty!!");  
        return;  
    }  
  
    else  
    {  
        printf("\nElements present in the Queue are:\n");  
        printf("\n\tElement\tPriority\n");  
        /*Printing queue elements along with their priority*/  
        printf("Front->");  
        while(ptr!=NULL)  
        {  
            printf("\t %d\t %d\n",ptr->element,ptr->priority);  
            ptr=ptr->next;  
        }  
    }  
}
```

## Output

Select an option

- 1 - Insert an element into the Queue
- 2 - Remove an element from the Queue
- 3 - Display all the elements in the Queue
- 4 - Exit

Enter your choice: 1

Enter the element to be inserted into the queue 10

Enter the priority of 10 3

Select an option

- 1 - Insert an element into the Queue
- 2 - Remove an element from the Queue
- 3 - Display all the elements in the Queue
- 4 - Exit

Enter your choice: 1

Enter the element to be inserted into the queue 20

```
Enter the priority of 20 2
```

```
Select an option
```

- 1 - Insert an element into the Queue
- 2 - Remove an element from the Queue
- 3 - Display all the elements in the Queue
- 4 - Exit

```
Enter your choice: 1
```

```
Enter the element to be inserted into the queue 30
```

```
Enter the priority of 30 1
```

```
Select an option
```

- 1 - Insert an element into the Queue
- 2 - Remove an element from the Queue
- 3 - Display all the elements in the Queue
- 4 - Exit

```
Enter your choice: 3
```

```
Elements present in the Queue are:
```

```
Element Priority
Front-> 30 1
          20 2
          10 3
```

```
Select an option
```

- 1 - Insert an element into the Queue
- 2 - Remove an element from the Queue
- 3 - Display all the elements in the Queue
- 4 - Exit

```
Enter your choice: 2
```

```
30 element removed from the queue
```

```
Select an option
```

- 1 - Insert an element into the Queue

```

2 - Remove an element from the Queue
3 - Display all the elements in the Queue
4 - Exit

```

Enter your choice: 3

Elements present in the Queue are:

```

Element Priority
Front-> 20 2
    10 3

```

As we can see in the above output, irrespective of the order in which elements are added into the queue, they are placed inside the queue as per their priorities and removed in the same fashion.

### Program analysis

Key Statement	Purpose
sstruct queue { int element; int priority; struct queue *next; };	Declares a priority queue node using linked list representation
scanf("%d",&pr);	Reads the priority of the element being inserted into the queue
while(temp->next!=NULL && temp->next->priority<=ptr->priority) temp=temp->next;	Identifies the location where the new element is to be inserted as per priority

## 5.7 DOUBLE-ENDED QUEUES

A double-ended queue is a special type of queue that allows insertion and deletion of elements at both ends, i.e., front and rear. In simple words, a double-ended queue can be referred as a linear list of elements in which insertion and deletion of elements takes place at its two ends but not in the middle. This is the reason why it is termed as double-ended queue or deque.

Based on the type of restrictions imposed on insertion and deletion of elements, a double-ended queue is categorized into two types:



### Check Point

#### 1. What is a priority queue?

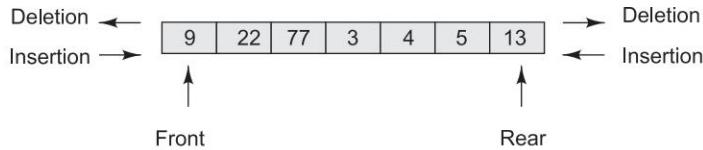
**Ans.** It is a type of queue in which each element is assigned certain priority such that the order of deletion of elements is decided by their associated priorities.

#### 2. What is order of deletion if two or more elements in a priority queue have same priorities?

**Ans.** If two or more elements have the same priority then they are deleted as per the order in which they were added into the queue (i.e. First-In-First-Out).

- Input-restricted deque** It allows deletion from both the ends but restricts the insertion at only one end.
- Output-restricted deque** It allows insertion at both the ends but restricts the deletion at only one end.

Figure 5.8 shows the logical representation of a deque.



**Fig. 5.8 Double-ended queue**

As shown in Fig. 5.8, insertion and deletion of elements is possible at both front and rear ends of the queue. As a result, the following four operations are possible for a double-ended queue:

- i\_front** Insertion at front end of the queue.
- d\_front** Deletion from front end of the queue.
- i\_rear** Insertion at rear end of the queue.
- d\_rear** Deletion from rear end of the queue.

**Example 5.11** Write C functions to realize the four possible insert and delete operations for array-implemented double-ended queues.

#### Program 5.5 i\_front() function

```
/*Insertion at front end/
/*queue[100], front and rear are global variables*/
void i_front(int element)
{
if(front== -1) /*Adding element in an empty queue*/
{
front = rear = front+1;
queue[front] = element;
return;
}

if(front==0) /*Checking whether the queue is full at the front end*/
{
printf("Queue is Full.\n");
getch();
return;
}

front=front-1; /*Decrementing rear pointer*/
queue[front]=element; /*Inserting the new element*/
}
```



#### Mind Jog

##### In which situation is a deque used?

A deque is used for implementing A-Steal job scheduling algorithm. This algorithm helps perform task scheduling for multiple processors.

### **Program 5.6 *d\_front()* function**

```
/*Deletion at front end*/
/*queue[100], front and rear are global variables*/
int d_front()
{
    int i;
    if(front===-1 && rear===-1) /*Checking whether the queue is empty*/
    {
        printf("\n\tQueue is Empty.\n");
        getch();
        return (-9999);
    }
    if(front==rear) /*Checking whether the queue has only one element left*/
    {
        i=queue[front];
        front=-1;
        rear=-1;
        return(i);
    }
    return(queue[front++]); /*Returning the front most element and incrementing
the front pointer*/
}
```

### **Program 5.7 *i\_rear()* function**

```
/*Insertion at rear end*/
/*queue[100], front and rear are global variables*/
void i_rear(int element)
{
    if(rear===-1) /*Adding element in an empty queue*/
    {
        front = rear = rear+1;
        queue[rear] = element;
        return;
    }

    if(rear==99) /*Checking whether the queue is full at the rear end*/
    {
        printf("Queue is Full.\n");
        getch();
        return;
    }

    rear=rear+1; /*Incrementing rear pointer*/
    queue[rear]=element; /*Inserting the new element*/
}
```

### **Program 5.8 *d\_rear()* function**

```
/*Deletion at rear end*/
/*queue[100], front and rear are global variables*/
int d_rear()
{
```

```

int i;
if(front===-1 && rear===-1) /*Checking whether the queue is empty*/
{
printf("\n\tQueue is Empty.\n");
getch();
return (-9999);
}
if(front==rear) /*Checking whether the queue has only one element left*/
{
i=queue[rear];
front=-1;
rear=-1;
return(i);
}
return(queue[rear-]); /*Returning the rear most element and decrementing
the rear pointer*/
}

```



## Check Points

---

### 1. What is a double-ended queue?

**Ans:** A double-ended queue is a special type of queue that allows insertion and deletion of elements at both ends, i.e., front and rear.

### 2. What are the different types of double-ended queues?

**Ans:** The two types of double-ended queues are input-restricted deque (insertion at one end, deletion at both ends) and output-restricted deque (insertion at both ends, deletion at one end).

## Solved Problems

### Problem 5.1

The contents of a queue Q are as follows:

Queue (Q)	4	5	-9	66				
Index	0	1	2	3	4	5	6	7

*F?*                           *R ?*

The queue can store a maximum of eight elements and the front (F) and rear (R) pointers currently point at index 0 and 3 respectively.

Show the queue contents and indicate the position of the front and rear pointers after each of the following queue operations:

- (a) Insert (Q, 16), (b) Delete (Q), (c) Delete (Q), (d) Insert (Q, 7), (e) Delete (Q),  
(f) Insert (Q, -2)

### Solution

- (a) Insert (Q, 16)

$$\text{Step 1} \quad R = R + 1 = 3 + 1 = 4$$

$$\text{Step 2} \quad Q[R] = Q[4] = 16$$

### Queue contents

<b>Queue (Q)</b>	4	5	-9	66	16			
<b>Index</b>	0	1	2	3	4	5	6	7

$F \uparrow$

$R \uparrow$

### (b) Delete (Q)

Step 1 Item =  $Q[F] = Q[0] = 4$

Step 2  $F = F + 1 = 0 + 1 = 1$

### Queue contents

<b>Queue (Q)</b>		5	-9	66	16			
<b>Index</b>	0	1	2	3	4	5	6	7

$F \uparrow$

$R \uparrow$

### (c) Delete (Q)

Step 1 Item =  $Q[F] = Q[1] = 5$

Step 2  $F = F + 1 = 1 + 1 = 2$

### Queue contents

<b>Queue (Q)</b>			-9	66	16			
<b>Index</b>	0	1	2	3	4	5	6	7

$F \uparrow$

$R \uparrow$

### (d) Insert (Q, 7)

Step 1  $R = R + 1 = 4 + 1 = 5$

Step 2  $Q[R] = Q[5] = 7$

### Queue contents

<b>Queue (Q)</b>			-9	66	16	7		
<b>Index</b>	0	1	2	3	4	5	6	7

$F \uparrow$

$R \uparrow$

### (e) Delete (Q)

Step 1 Item =  $Q[F] = Q[2] = -9$

Step 2  $F = F + 1 = 2 + 1 = 3$

### Queue contents

<b>Queue (Q)</b>				66	16	7		
<b>Index</b>	0	1	2	3	4	5	6	7

$F \uparrow$

$R \uparrow$

### (f) Insert (Q, -2)

Step 1  $R = R + 1 = 5 + 1 = 6$

Step 2  $Q[R] = Q[6] = -2$

### Queue contents

<b>Queue (Q)</b>			-9	66	16	7	-2	
<b>Index</b>	0	1	2	3	4	5	6	7

F↑

R ↑

**Problem 5.2** Consider the following two states of a queue Q:

### State 1

<b>Queue (Q)</b>		4	-1	8				
<b>Index</b>	0	1	2	3	4	5	6	7

F↑

R ↑

### State 2

<b>Queue (Q)</b>				3	11	22	33	
<b>Index</b>	0	1	2	3	4	5	6	7

F↑

R ?

Write the series of insert and delete operations that will transition the queue Q from State 1 to State 2.

### Solution

- Step 1      Delete (Q)
- Step 2      Delete (Q)
- Step 3      Insert (Q, 11)
- Step 4      Insert (Q, 22)
- Step 5      Insert (Q, 33)

**Problem 5.3** Is there any limitation associated with array implemented queues?

**Solution** One of the key limitations of array implemented queue is that it may lead to an overflow condition even when a number of its preceding locations are empty. Such a situation can be easily avoided by implementing the queue in a circular fashion, which logically connects its front and rear ends.

**Problem 5.4** Identify the error in the following structure declaration of a priority queue node:

```
struct queue /*Node of a priority queue*/
{
    int element;
    int priority;
}*next;
```

**Solution** In the linked implementation of a queue, the next pointer should be associated with each node of the queue. Hence, next should be declared inside the structure declaration and not outside, as shown below:

```
struct queue /*Node of a priority queue*/
{
    int element;
    int priority;
}*next; /*Pointer to the next queue node*/
}
```

## Summary

- ◆ Queue is a linear data structure in which items are inserted at one end called ‘Rear’ and deleted from the other end called ‘Front’.
- ◆ Queues are based on the First-In-First-Out (FIFO) principle that means the data item that is inserted first in the queue is also the first one to be removed from the queue.
- ◆ There are two key operations associated with the queue data structure: insert and delete.
- ◆ Queues can be implemented through arrays or linked lists.
- ◆ The array implementation of queues reserves a fixed amount of memory space in the form of an array for storing queue elements.
- ◆ The linked implementation of queues uses dynamic memory management techniques for allocating the memory space for storing a new queue element at run time.
- ◆ Since linked implementation of queues is based on dynamic memory allocation it is more efficient as compared to array-based implementation.
- ◆ A circular queue is one whose start and end locations are logically connected with each other.
- ◆ Circular queues remove one of the main disadvantages of array implemented queues in which a lot of memory space is wasted due to inefficient utilization.
- ◆ Priority queue is a type of queue in which each element is assigned certain priority such that the order of deletion of elements is decided by their associated priorities.
- ◆ A double-ended queue is a special type of queue that allows insertion and deletion of elements at both ends, i.e., front and rear.
- ◆ A double-ended queue can also be referred as a linear list of elements in which insertion and deletion of elements takes place at its two ends but not in the middle.
- ◆ A double-ended queue is categorized into two types: input-restricted deque and output-restricted deque.

## Key Terms

- ◆ **Queue** It is a linear data structure based on the First-In-First-Out (FIFO) principle that means the data item that is inserted first in the queue is also the first one to be removed from the queue.
- ◆ **Front** It represents the front end of the queue from where elements are deleted.
- ◆ **Rear** It represents the rear end of the queue where elements are added.
- ◆ **FIFO** It stands for First-In-First-Out i.e., the principle on which queues are based.
- ◆ **Insert** It refers to the task of inserting an element into the queue.
- ◆ **Delete** It refers to the task of retrieving or deleting an element from the queue.
- ◆ **Array implementation** It refers to the realization of queue data structure using arrays.
- ◆ **Lined implementation** It refers to the realization of queue data structure using linked lists.
- ◆ **Circular queue** It is a type of queue whose start and end locations are logically connected with each other.
- ◆ **Priority queue** It is a type of queue in which each element is assigned certain priority such that the order of deletion of elements is decided by their associated priorities.
- ◆ **Double-ended queue** It is a special type of queue that allows insertion and deletion of elements at both ends, i.e., front and rear.

## Multiple-Choice Questions

- 5.1** Which of the following statements is not true for queues?
- (a) It is a linear data structure.
  - (b) It allows insertion/deletion of elements only at one end.
  - (c) It has two ends front and rear.
  - (d) It is based on First-In-First-Out principle.
- 5.2** Which of the following statements is not an example of a queue?
- (a) Collection of tiles one over another.
  - (b) A queue of print jobs.
  - (c) A line up of people waiting for the bus at the bus stop.
  - (d) All of the above are queue examples.
- 5.3** CPU scheduler can be implemented by which of the following data structures?
- (a) Stack
  - (b) Queue
  - (c) Graph
  - (d) Tree
- 5.4** Which of the following is a type of a queue?
- (a) Circular queue
  - (b) Priority queue
  - (c) Double-ended queue
  - (d) All of the above
- 5.5** If 1, 2, 3, 4 are the queue contents with element 1 at the front and 4 at the rear, then what will be the queue contents after following operations:
- Insert (5)*  
*Delete ()*  
*Delete ()*  
*Delete ()*  
*Insert (6)*  
*Insert (-1)*  
*Delete ()*
- (a) 5, 6, -1
  - (b) 4, 5, 6, -1
  - (c) 1, 2, 6
  - (d) 1, 2, 6, -1
- 5.6** Which of the following is best suitable for implementing a print scheduler?
- (a) Stack
  - (b) Queue
  - (c) Array
  - (d) None of the above
- 5.7** If ‘front’ points at the front end of the queue, ‘rear’ points at the rear end of the queue and ‘queue []’ is the array containing queue elements, then which of the following statements correctly reflects the insert operation for inserting ‘item’ into the queue?
- (a) rear = rear + 1; queue [rear] = item;
  - (b) front = front + 1; queue [front] = item;
  - (c) queue [rear++] = item;
  - (d) Both (a) and (c) are correct
- 5.8** If ‘front’ points at the front end of the queue, ‘rear’ points at the rear end of the queue and ‘queue []’ is the array containing queue elements, then which of the following statements correctly reflects the delete operation for deleting an element from the queue?
- (a) item = queue [rear]; rear = rear + 1;
  - (b) item = queue [front]; front = front + 1;
  - (c) item = queue [++front];
  - (d) Both (b) and (c) are correct
- 5.9** If a delete operation is performed on an empty queue, then which of the following situations will occur?

- 5.10** Which of the following is not a queue application?

  - (a) Recursion control
  - (b) CPU scheduling
  - (c) Message queuing
  - (d) All of the above are queue applications

## Review Questions

- 5.1** What is a queue? Explain with examples.

**5.2** Briefly describe the FIFO principle.

**5.3** What are front and rear pointers? Explain their significance.

**5.4** What are the different application areas of queue data structure?

**5.5** Give any three real-life examples that principally resemble the queue data structure.

**5.6** Explain the logical representation of queue in memory with the help of an example.

**5.7** Explain insert and delete queue operations with the help of examples.

**5.8** Deduce the contents of an empty queue after the execution of the following operations in sequence:  
Insert (9)  
Insert (-7)  
Delete ()  
Insert (4)  
Delete ()  
Insert (18)  
Delete ()

**5.9** What is a priority queue? How is it different from a normal queue?

**5.10** Explain the significance of double-ended queue.

**5.11** How are queues implemented?

**5.12** What is the advantage of linked implementation of queues over array implementation?

**5.13** What is the objective of implementing a queue in circular fashion?

**5.14** List the differences between stack and queue data structures.

**5.15** How is a double-ended queue implemented?

## Programming Exercises

- 5.1** Write a C function to print the elements of a queue implemented using linked list.
  - 5.2** Write a C function to perform the delete operation on an array-implemented circular queue.
  - 5.3** Write a C function to insert an element into a priority queue as per priority.
  - 5.4** Write a C function to perform the delete operation at the end of a double-ended queue.
  - 5.5** Write a C function to remove elements from a queue and store them in a stack. Also, display the contents of the resultant stack.

## Answers to Multiple-Choice Questions

- 5.1 (b)                  5.2 (a)                  5.3 (b)                  5.4 (d)                  5.5 (a)  
5.6 (b)                  5.7 (d)                  5.8 (b)                  5.9 (b)                  5.10 (a)



## **UNIT-III**

---

# **Non Linear Data Structures – Trees**

### **CHAPTER**

Chapter 6: Trees



# 6

# TREES

## Chapter outline

- 6.1 Introduction
- 6.2 Basic Concept
  - 6.2.1 Tree Terminology
- 6.3 Binary Tree
  - 6.3.1 Binary Tree Concepts
- 6.4 Binary Tree Representation
  - 6.4.1 Array Representation
  - 6.4.2 Linked Representation
- 6.5 Binary Tree Traversal
- 6.6 Binary Search Tree
- 6.7 Tree Variants
  - 6.7.1 Expression Trees
  - 6.7.2 Threaded Binary Trees
  - 6.7.3 Balanced Trees
  - 6.7.4 Splay Trees
  - 6.7.5 m-way Trees

*Summary*

*Key Terms*

*Multiple-Choice Questions*

*Review Questions*

*Programming Exercises*

*Answers to Multiple-Choice Questions*



## 6.1 INTRODUCTION

Till now, we focussed only on linear data structures such as stacks, queues and linked lists. But, in real-world situations, data relationships are not always linear. Tree is one such non-linear data structure which stores the data elements in a hierarchical manner. Each node of the tree stores a data value, and is linked to other nodes in a hierarchical fashion.

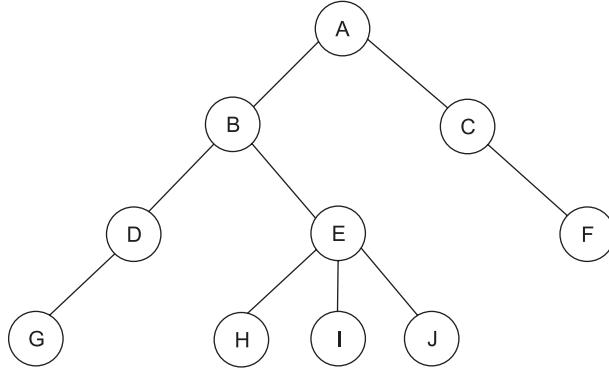
In this chapter, we will learn about the different types of trees and their related operations. Most importantly, we will focus on binary tree and its variants, which are widely used in the field of computer science.

## 6.2 BASIC CONCEPT

A tree is defined as a finite set of elements or nodes, such that

1. One of the nodes present at the top of the tree is marked as root node.
2. The remaining elements are partitioned across multiple subtrees present below the root node.

Figure 6.1 shows a sample tree T.



**Fig. 6.1** Tree T

Here, T is a simple tree containing ten nodes with A being the root node. The node A contains two subtrees. The left subtree starts at node B while the right subtree starts at node C. Both the subtrees further contain subtrees below them, thus indicating recursive nature of the tree data structure. Each node in the tree has zero or more child nodes.

### 6.2.1 Tree Terminology

There are a number of key terms associated with trees. Table 6.1 lists some of the important key terms.

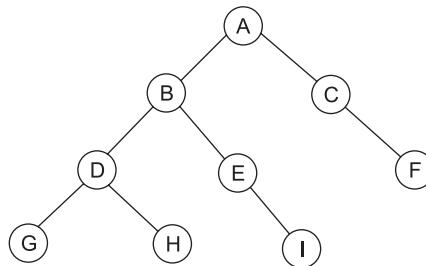
**Table 6.1** Tree terminology

Key Term	Description	Example (Refer to Fig. 6.1)
Node	It is the data element of a tree. Apart from storing a value, it also specifies links to the other nodes.	A, B, C, D

Key Term	Description	Example (Refer to Fig. 6.1)
Root	It is the top node in a tree.	A
Parent	A node that has one or more child nodes present below it is referred as parent node.	B is the parent node of D and E
Child	All nodes in a tree except the root node are child nodes of their immediate predecessor nodes.	H, I and J are child nodes of E
Leaf	It is the terminal node that does not have any child nodes.	G, H, I, J and F are leaf nodes
Internal node	All nodes except root and leaf nodes are referred as internal nodes.	B, C, D and E are internal nodes
Sibling	All the child nodes of a parent node are referred as siblings.	D and E are siblings
Degree	The degree of a node is the number of subtrees coming out of the node.	Degree of A is 2 Degree of E is 3
Level	All the tree nodes are present at different levels. Root node is at level 0, its child nodes are at level 1, and so on.	A is at level 0 B and C are at level 1 G, H, I, J are at level 3
Depth or Height	It is the maximum level of a node in the tree.	Depth of tree T is 3
Path	It is the sequence of nodes from source node till destination node.	A–B–E–J

### 6.3 BINARY TREE

Binary tree is one of the most widely used non-linear data structures in the field of computer science. It is a restricted form of a general tree. The restriction that it applies to a general tree is that its nodes can have a maximum degree of 2. That means, the nodes of a binary tree can have zero, one or two child nodes but not more than that. Figure 6.2 shows a binary tree.



**Fig. 6.2** Binary tree

As shown in the above binary tree, all nodes have a maximum degree of 2. The maximum number of nodes that can be present at level  $n$  is  $2^n$ .

### 6.3.1 Binary Tree Concepts

Before we learn how binary trees are represented in memory, let us discuss some of the key concepts associated with binary trees. Table 6.2 lists these key concepts.

**Table 6.2** *Binary tree concepts*

Concept	Description	Example
Strictly binary tree	A binary tree is called strictly binary if all its nodes barring the leaf nodes contain two child nodes.	<pre> graph TD     A((A)) --- B((B))     A --- C((C))     B --- D((D))     B --- E((E))     D --- F((F))     D --- G((G))     E --- H((H))     E --- I((I))   </pre>
Complete binary tree	A binary tree of depth $d$ is called complete binary tree if all its levels from 0 to $d-1$ contain maximum possible number of nodes and all the leaf nodes present at level $d$ are placed towards the left side.	<pre> graph TD     A((A)) --- B((B))     A --- C((C))     B --- D((D))     B --- E((E))     C --- F((F))     C --- G((G))     D --- H((H))     D --- I((I))     E --- J((J))     E --- K((K))     F --- L((L))     G --- O((O))   </pre>
Perfect binary tree	A binary tree is called perfect binary tree if all its leaf nodes are at the lowest level and all the non-leaf nodes contain two child nodes.	<pre> graph TD     A((A)) --- B((B))     A --- C((C))     B --- D((D))     B --- E((E))     C --- F((F))     C --- G((G))     D --- H((H))     D --- I((I))     E --- J((J))     E --- K((K))     F --- L((L))     G --- O((O))   </pre>
Balanced binary tree	A binary tree is called balanced binary tree if the depths of the subtrees of all its nodes do not differ by more than 1.	<pre> graph TD     A((A)) --- B((B))     A --- C((C))     B --- D((D))     B --- E((E))     C --- F((F))     C --- G((G))     D --- H((H))     D --- I((I))   </pre>

## 6.4 BINARY TREE REPRESENTATION

The sequential representation of binary trees is done by using arrays while the linked representation is done by using linked lists.

### 6.4.1 Array Representation

In the array representation of binary trees, one-dimensional array is used for storing the node elements. The following rules are applied while storing the node elements in the array:

1. The root node is stored at the first position in the array while its left and right child nodes are stored at the successive positions.
2. If a node is stored at index location  $i$  then its left child node will be stored at location  $2i+1$  while the right child node will be stored at location  $2i+2$ .

Let us consider a binary tree  $T_1$ , as shown in Fig. 6.3.

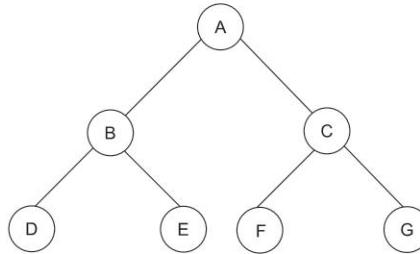


Fig. 6.3 Binary tree  $T_1$

Here,  $T_1$  is a binary tree containing seven nodes with A being the root node. B and C are the left and right child nodes of A respectively. Let us apply the rules explained earlier to arrive at the array representation of binary tree  $T_1$ . Figure 6.4 shows the array representation.

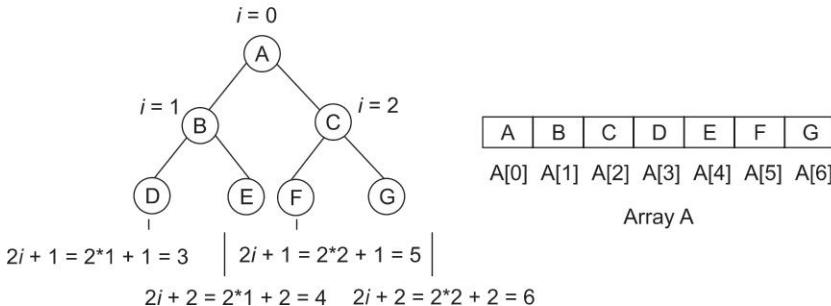


Fig. 6.4 Array representation of binary tree  $T_1$



#### Check Point

##### 1. What is a leaf?

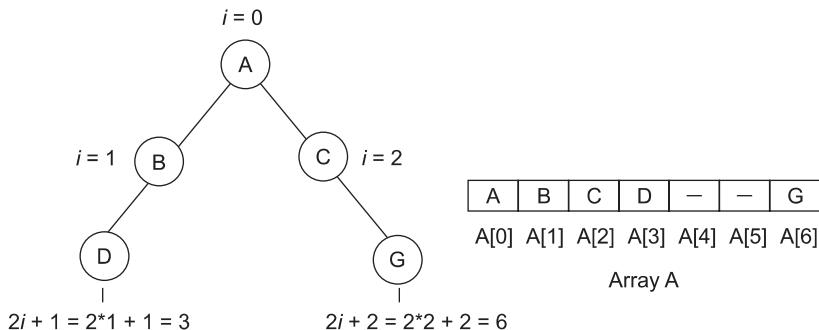
**Ans.** It is the terminal node in a tree that does not have any child nodes.

##### 2. What is a balanced binary tree?

**Ans.** A binary tree is called balanced binary tree if the depths of the subtrees of all its nodes do not differ by more than 1.

Figure 6.4 shows the array index values for each of the tree nodes. Array A is used for storing the node values.

Now, let us modify the binary tree  $T_1$  a little by deleting nodes E and F. The revised array representation of  $T_1$  is shown in Fig. 6.5.



**Fig. 6.5** Revised array representation of binary tree  $T_1$

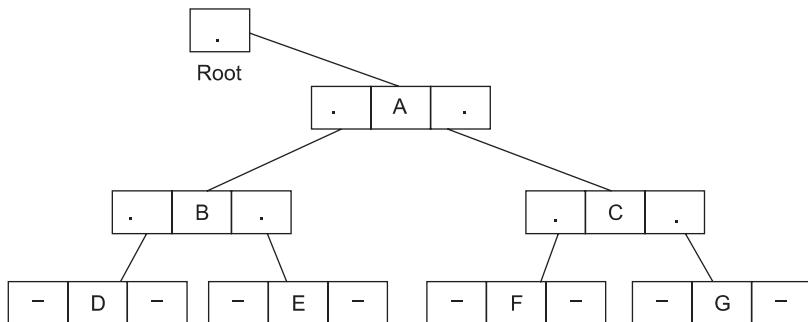
As we can see in Fig. 6.5, even after removing two elements from the tree, it still requires the same number of memory locations for storing the node elements. This is the main disadvantage of array representation of binary trees. It efficiently utilises the memory space only when the tree is a complete binary tree. Otherwise, there are always some memory locations lying vacant in the array.

## 6.4.2 Linked Representation

To avoid the disadvantages associated with array representation, linked representation is used for implementing binary trees. It uses a linked list for storing the node elements. Each tree node is represented with the help of the linked list node comprising of the following fields:

1. **INFO** Stores the value of the tree node.
2. **LEFT** Stores a pointer to the left child.
3. **RIGHT** Stores a pointer to the right child.

In addition, there is a special pointer that points at the root node. Figure 6.6 shows how linked list is used for representing a binary tree in memory.



**Fig. 6.6** Linked representation of binary tree

The linked representation of binary tree uses dynamic memory allocation technique for adding new nodes to the tree. It reserves only that much amount of memory space as is required for storing its node values. Thus, linked representation is more efficient as compared to array representation.

**Example 6.1** Write a program to implement a binary tree using linked list.

**Program 6.1** *Implementation of a binary tree*

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

struct bin_tree
{
    int INFO;
    struct node *LEFT, *RIGHT;
};

typedef struct bin_tree node;

node *insert(node *,int); /*Function prototype for inserting a new node*/
void display (node *); /*Function prototype for displaying the tree nodes*/
int count = 1; /*Counter for ascertaining left or right position for the
new node*/

void main()
{
    struct node *root = NULL; →
    int element, choice;

    clrscr();

    /*Displaying a menu of choices*/
    while(1)
    {
        clrscr();
        printf("Select an option\n");
        printf("\n1 - Insert");
        printf("\n2 - Display");
        printf("\n3 - Exit");

        printf("\n\nEnter your choice: ");
        scanf("%d", &choice);

        switch(choice)
        {
            case 1:
            {
```

*Here, the node of the tree is realised with the help of a structure declaration. The INFO field stores the node value while the LEFT and RIGHT pointers point at the left and right subtrees respectively.*

*Since the root node has no parents, its location is tracked with the help of a special pointer called root.*

```

printf("\n\nEnter the node value: ");
scanf("%d",&element);
root = insert(root,element); /*Calling the insert function for inserting
a new element into the tree*/
getch();
break;
}

case 2:
{
display(root); /*Calling the display function for printing the node
values*/
getch();
break;
}

case 3:
{
exit(1);
break;
}

default:
{
printf("\nIncorrect choice. Please try again.");
getch();
break;
}
}
}
}
}
}

node *insert(node *r, int n)
{
if(r==NULL)
{
r=(node*) malloc (sizeof(node));
r->LEFT = r->RIGHT = NULL;
r->INFO = n;
count=count+1;
}
else
{
if(count%2==0)
r->LEFT = insert(r->LEFT, n);
else
r->RIGHT = insert(r->RIGHT, n);
}
return(r);
}

```

*The use of dynamic memory allocation ensures that memory space for a new node is allocated only at the time of its creation.*

```
}

void display(node * r)
{
    if(r->LEFT!=NULL)
        display(r->LEFT);
    printf("%d\n",r->INFO);
    if(r->RIGHT!=NULL)
        display(r->RIGHT);
}
```

## Output

```
Select an option
```

```
1 - Insert
2 - Display
3 - Exit
```

```
Enter your choice: 1
```

```
Enter the node value: 1
```

```
Enter your choice: 1
```

```
Enter the node value: 2
```

```
Enter your choice: 1
```

```
Enter the node value: 3
```

```
Enter your choice: 1
```

```
Enter the node value: 4
```

```
Enter your choice: 1
```

```
Enter the node value: 5
```

```
Enter your choice: 1
```

```
Enter the node value: 6
```

```
Select an option
```

```
1 - Insert
2 - Display
3 - Exit
```

```
Enter your choice: 2
```

```
6  
4  
2  
1  
3  
5
```

Select an option

```
1 - Insert  
2 - Display  
3 - Exit
```

Enter your choice: 3

### Program analysis

Key Statement	Purpose
<code>node *insert(node *,int);</code> <code>void display (node *);</code>	Declares function prototypes for inserting and displaying binary tree nodes
<code>root = insert(root,element);</code>	Calls the <i>insert()</i> function for inserting a new node into the binary tree
<code>display(root);</code>	Calls the <i>display()</i> function for displaying the binary tree nodes
<code>if(count%2==0)     r-&gt;LEFT = insert(r-&gt;LEFT, n);     else         r-&gt;RIGHT = insert(r-&gt;RIGHT, n);</code>	Checks the value of the <i>count</i> variable to insert the new node either in the left or right subtree

## 6.5 BINARY TREE TRAVERSAL

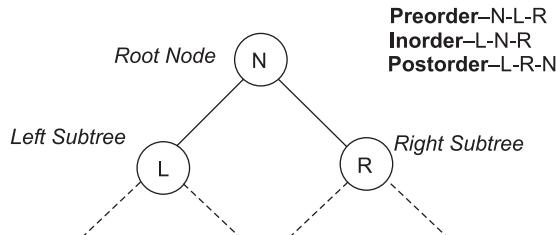
Traversal is the process of visiting the various elements of a data structure. Binary tree traversal can be performed using three methods:

1. Preorder
  2. Inorder
  3. Postorder
1. **Preorder** The preorder traversal method performs the following operations:
    - (a) Process the root node (N).
    - (b) Traverse the left subtree of N (L).
    - (c) Traverse the right subtree of N (R).
  2. **Inorder** The inorder traversal method performs the following operations:
    - (a) Traverse the left subtree of N (L).
    - (b) Process the root node (N).
    - (c) Traverse the right subtree of N (R).
  3. **Postorder** The postorder traversal method performs the following operations:

- (a) Traverse the left subtree of N (L).
- (b) Traverse the right subtree of N (R).
- (c) Process the root node (N).

Figure 6.7 shows an illustration of the different binary tree traversal methods.

**Example 6.2** Consider the following binary tree:



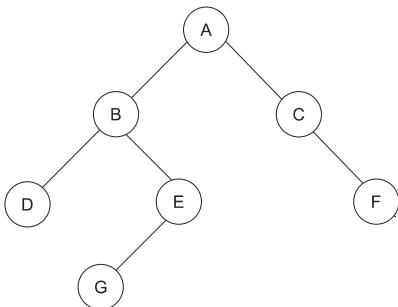
**Fig. 6.7 Binary tree traversal**

For the above binary tree, deduce the following:

- (a) Preorder traversal sequence
- (b) Inorder traversal sequence
- (c) Postorder traversal sequence

**Solution**

- (a) Preorder traversal sequence  
A-B-D-E-G-C-F
- (b) Inorder traversal sequence  
D-B-G-E-A-C-F
- (c) Postorder traversal sequence  
D-G-E-B-F-C-A



**Example 6.3** Write algorithms for the following:

- (a) Preorder traversal
- (b) Inorder traversal
- (c) Postorder traversal

**Solution**

- (a) Preorder

```

preorder(root)
Step 1: Start
Step 2: Display root
Step 3: Function Call preorder(root->LEFT)
Step 4: Function Call preorder(root->RIGHT)
Step 5: Stop
  
```

- (b) Inorder

```

inorder(root)
Step 1: Start
  
```

```
Step 2: Function Call inorder(root->LEFT)
Step 3: Display root
Step 4: Function Call inorder(root->RIGHT)
Step 5: Stop
```

(c) Postorder

```
postorder(root)
Step 1: Start
Step 2: Function Call postorder(root->LEFT)
Step 3: Function Call postorder(root->RIGHT)
Step 4: Display root
Step 5: Stop
```

**Example 6.4** Modify the program shown in Example 6.1 to add preorder, inorder and postorder traversals to the linked implementation of binary tree.

**Program 6.2** Preorder, inorder, and postorder traversal of binary tree

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

struct bin_tree
{
    int INFO;
    struct node *LEFT, *RIGHT;
};

typedef struct bin_tree node;  
  
node *insert(node *,int); /*Function prototype for inserting a new node*/
void preorder(node *); /*Function prototype for displaying preorder traversal path*/
void inorder(node *); /*Function prototype for displaying inorder traversal path*/
void postorder(node *); /*Function prototype for displaying postorder traversal path*/  
  
int count = 1; /*Counter for ascertaining left or right position for the new node*/  
  
void main()
{
    struct node *root = NULL;
    int element, choice;  
  
clrscr();
```

*Declaration of function prototypes  
for each of the traversal methods.*

```

/*Displaying a menu of choices*/
while(1)
{
clrscr();
printf("Select an option\n");
printf("\n1 - Insert");
printf("\n2 - Preorder");
printf("\n3 - Inorder");
printf("\n4 - Postorder");
printf("\n5 - Exit");

printf("\n\nEnter your choice: ");
scanf("%d", &choice);

switch(choice)
{
case 1:
{
printf("\n\nEnter the node value: ");
scanf("%d", &element);
root = insert(root,element); /*Calling the insert function for inserting
a new element into the tree*/
getch();
break;
}

case 2:
{
preorder(root); /*Calling the preorder function*/
getch();
break;
}

case 3:
{
inorder(root); /*Calling the inorder function*/
getch();
break;
}

case 4:
{
postorder(root); /*Calling the postorder function*/
getch();
break;
}

case 5:
{
}
}

```

```

exit(1);
break;
}

default:
{
printf("\nIncorrect choice. Please try again.");
getch();
break;
}
}
}
}

node *insert(node *r, int n)
{
if(r==NULL)
{
r=(node*) malloc (sizeof(node));
r->LEFT = r->RIGHT = NULL;
r->INFO = n;
count=count+1;
}
else
{
if(count%2==0)
r->LEFT = insert(r->LEFT, n);
else
r->RIGHT = insert(r->RIGHT, n);
}
return(r);
}

void preorder(node *r)
{
if(r!=NULL)
{
printf("%d\n", r->INFO);
preorder(r->LEFT); ←
preorder(r->RIGHT); }
}

void inorder(node *r)
{
if(r!=NULL)
{
inorder(r->LEFT);

```

*Recursive function calls apply the traversal sequence to each of the nodes in the left and right subtrees.*

```
printf("%d\n", r->INFO);
inorder(r->RIGHT);
}
}

void postorder(node *r)
{
if(r!=NULL)
{
postorder(r->LEFT);
postorder(r->RIGHT);
printf("%d\n", r->INFO);
}
}
```

## Output

```
Select an option
```

```
1 - Insert
2 - Preorder
3 - Inorder
4 - Postorder
5 - Exit
```

```
Enter your choice: 1
```

```
Enter the node value: 1
```

```
Enter your choice: 1
```

```
Enter the node value: 2
```

```
Enter your choice: 1
```

```
Enter the node value: 3
```

```
Enter your choice: 1
```

```
Enter the node value: 4
```

```
Enter your choice: 1
```

```
Enter the node value: 5
```

```
Enter your choice: 1
```

```
Enter the node value: 6
```

Select an option

- 1 - Insert
- 2 - Preorder
- 3 - Inorder
- 4 - Postorder
- 5 - Exit

Enter your choice: 2

1 ← *Preorder traversal sequence* |  
2  
4  
6  
3  
5

Select an option

- 1 - Insert
- 2 - Preorder
- 3 - Inorder
- 4 - Postorder
- 5 - Exit

Enter your choice: 3

6 ← *Inorder traversal sequence* |  
4  
2  
1  
3  
5

Select an option

- 1 - Insert
- 2 - Preorder
- 3 - Inorder
- 4 - Postorder
- 5 - Exit

Enter your choice: 4

6 ← *Postorder traversal sequence* |  
4  
2  
5  
3  
1

Select an option

- 1 - Insert
- 2 - Preorder

```

3 - Inorder
4 - Postorder
5 - Exit

```

Enter your choice: 5

### Program analysis

Key Statement	Purpose
<code>node *insert(node *,int);</code> <code>void preorder(node *);</code> <code>void inorder(node *);</code> <code>void postorder(node *);</code>	Declares the function prototypes for inserting a new node and traversing the binary tree using different traversal methods
<code>preorder(root);</code>	Calls the <i>preorder()</i> function to traverse the binary tree in preorder sequence
<code>inorder(root);</code>	Calls the <i>inorder()</i> function to traverse the binary tree in inorder sequence
<code>postorder(root);</code>	Calls the <i>postorder()</i> function to traverse the binary tree in postorder sequence

## 6.6 BINARY SEARCH TREE

A binary tree is referred as a binary search tree if for any node  $n$  in the tree:

1. the node elements in the left subtree of  $n$  are lesser in value than  $n$ .
2. the node elements in the right subtree of  $n$  are greater than or equal to  $n$ .

Thus, binary search tree arranges its node elements in a sorted manner. As the name suggests, the most important application of a binary search tree is searching. The average running time of searching an element in a binary search tree is  $O(\log n)$ , which is better than other data structures like array and linked lists.

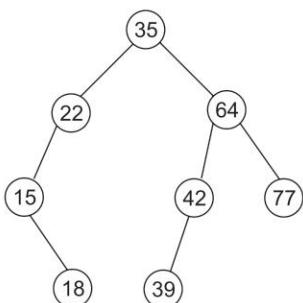


Fig. 6.8 Binary search tree

Figure 6.8 shows a sample binary search tree.

As we can see in the figure, all the nodes in the left subtree are less than the nodes in the right subtree.

The various operations performed on a binary search tree are:

1. Insert
2. Search
3. Delete

1. **Insert** The insert operation involves adding an element into the binary tree. The location of the new element is determined in such a manner that insertion does not disturb the sort order of the tree.



### Check Point

1. Which node in a binary tree does not have a parent node?

Ans. Root

2. Which tree traversal method processes the root node first and then the left and right subtrees?

Ans. Preorder

**Example 6.5** Write a C function for inserting an element into a binary search tree.

```
node *insert(node *r, int n)
{
if(r==NULL)
{
r=(node*) malloc (sizeof(node));
r->LEFT = r->RIGHT = NULL;
r->INFO = n;
}
else if(n<r->INFO)
r->LEFT = insert(r->LEFT, n);
else if(n>r->INFO)
r->RIGHT = insert(r->RIGHT, n);
else if(n==r->INFO)
printf("\nInsert Operation failed: Duplicate Entry!!");
return(r);
}
```

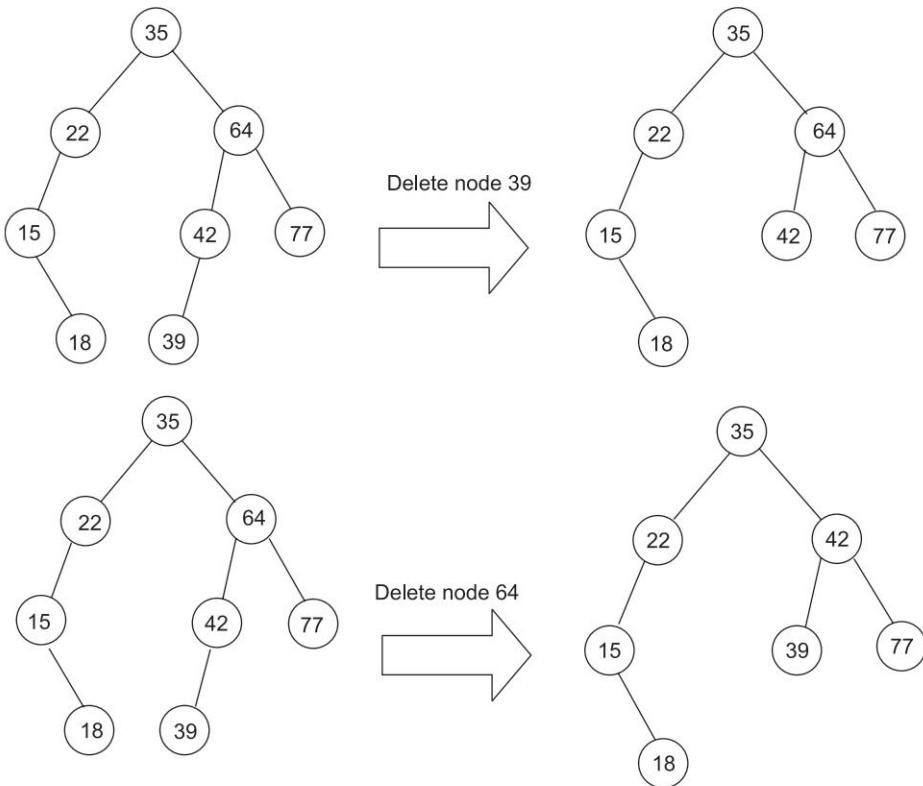
*A series of recursive function calls are required to identify the precise location where the new node will be inserted.*

- 2. Search** The search operation involves traversing the various nodes of the binary tree to search the desired element. The sorted nature of the tree greatly benefits the search operation as with each iteration, the number of nodes to be searched gets reduced. For example, if the value to be searched is less than the root value then the remainder of the search operation will only be performed in the left subtree while the right subtree will be completely ignored.

**Example 6.6** Write a C function for searching an element in a binary search tree.

```
void search(node *r,int n)
{
if(r==NULL)
{
printf("\n%d not present in the tree!!",n);
return;
}
else if(n==r->INFO)
printf("\nElement %d is present in the tree!!",n);
else if(n<r->INFO)
search(r->LEFT,n);
else
search(r->RIGHT,n);
}
```

- 3. Delete** The delete operation involves removing an element from the binary search tree. It is important to ensure that after the element is removed from the tree, the other elements are shuffled in such a manner that the sort order of the tree is regained. The delete operation is depicted in Fig. 6.9.



**Fig. 6.9** Deleting an element from binary search tree

As we can see in Fig. 6.9, if the node to be deleted is a leaf node, then it is simply deleted without requiring any shuffling of other nodes. However, if the node to be deleted is an internal node then appropriate shuffling is required to ensure that the tree regains its sort order.

**Example 6.7** Write a C function for deleting an element from a binary search tree.

```
int del(node *r,int n)
{
    node *ptr;
    if(r==NULL)
    {
        return(0);
    }
    else if(n<r->INFO)
        return(del(r->LEFT,n));
    else if(n>r->INFO)
        return(del(r->RIGHT,n));
    else
    {
```

A return value of 0 signifies unsuccessful search while a return value of 1 signifies successful search.

```

if (r->LEFT==NULL)
{
ptr=r;
r=r->RIGHT;
free(ptr);
return(1);
}
else if (r->RIGHT==NULL)
{
ptr=r;
r=r->LEFT;
free(ptr);
return(1);
}
else
{
ptr=r->LEFT;
while (ptr->RIGHT!=NULL)
ptr=ptr->RIGHT;
r->INFO=ptr->INFO;
return(del(r->LEFT,ptr->INFO));
}
}
}
}

```

**4. Implementation** The implementation of a binary search tree requires implementing the insert, search, and delete operations.

**Example 6.8** Write a C program for implementing a binary search tree.

Program 6.3 uses the insert (Example 6.5), search (Example 6.6), and delete (Example 6.7) functions.

**Program 6.3** *Implementation of a binary tree*

```

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

struct BST
{
    int INFO;
    struct node *LEFT, *RIGHT;
};

typedef struct BST node;
node *insert(node *,int); /*Function prototype for inserting a new node*/
void search(node *,int); /*Function prototype for searching a node*/
int del(node *,int); /*Function prototype for deleting a node*/
void display(node*);

void main()

```

```

{
    struct node *root = NULL;
    int element, choice, num, flag;

    clrscr();

    /*Displaying a menu of choices*/
    while(1)
    {
        clrscr();
        printf("Select an option\n");
        printf("\n1 - Insert");
        printf("\n2 - Search");
        printf("\n3 - Delete");
        printf("\n4 - Display");
        printf("\n5 - Exit");

        printf("\n\nEnter your choice: ");
        scanf("%d", &choice);

        switch(choice)
        {
            case 1:
            {
                printf("\n\nEnter the node value: ");
                scanf("%d", &element);
                root = insert(root,element); /*Calling the insert function for inserting
a new element into the tree*/
                getch();
                break;
            }

            case 2:
            {
                printf("\nEnter the element to be searched: ");
                scanf("%d", &num);
                search(root,num);
                getch();
                break;
            }

            case 3:
            {
                printf("\n\nEnter the element to be deleted: ");
                scanf("%d", &num);
                flag=del(root,num);
                if(flag==1)
                    printf("\nElement %d deleted from the list",num);
                else

```

```

printf("\nElement %d not present in the list",num);
getch();
break;
}

case 4:
{
display(root);
getch();
break;
}

case 5:
{
exit(1);
break;
}

default:
{
printf("\nIncorrect choice. Please try again.");
getch();
break;
}
}
}
}
}

void display(node * r)
{
if(r->LEFT!=NULL)
display(r->LEFT);
printf("%d\n",r->INFO);
if(r->RIGHT!=NULL)
display(r->RIGHT);
}
}

```

## Output

Select an option

1 - Insert  
 2 - Search  
 3 - Delete  
 4 - Display  
 5 - Exit

Enter your choice: 1

Enter the node value: 6

```
Enter your choice: 1  
Enter the node value: 1  
Enter your choice: 1  
Enter the node value: 5  
Enter your choice: 1  
Enter the node value: 2  
Enter your choice: 1  
Enter the node value: 4  
Enter your choice: 1  
Enter the node value: 3  
Select an option
```

```
1 - Insert  
2 - Search  
3 - Delete  
4 - Display  
5 - Exit
```

```
Enter your choice: 4  
1  
2  
3  
4  
5  
6
```

*Irrespective of the order in which elements are added to the binary search tree, the insert function always stores them in a sorted manner.*

```
Enter your choice: 2  
Enter the element to be searched: 7  
7 not present in the tree!!  
Enter your choice: 2  
Enter the element to be searched: 4  
Element 4 is present in the tree!!
```

```
Select an option
```

```
1 - Insert  
2 - Search  
3 - Delete  
4 - Display  
5 - Exit
```

```
Enter your choice: 3
```

```
Enter the element to be deleted: 8
```

```
Element 8 not present in the list
```

```
Enter the element to be deleted: 4
```

```
Element 4 deleted from the list
```

```
Select an option
```

```
1 - Insert  
2 - Search  
3 - Delete  
4 - Display  
5 - Exit
```

```
Enter your choice: 5
```

### Program analysis

Key Statement	Purpose
<code>node *insert(node *,int); void search(node *,int); int del(node *,int); void display(node*);</code>	Declares function prototypes for performing operations on the binary search tree
<code>root = insert(root,element);</code>	Calls the <i>insert()</i> function for inserting a new node into the binary search tree
<code>search(root,num);</code>	Calls the <i>search()</i> function for performing search operation on the binary search tree
<code>flag=del(root,num);</code>	Calls the <i>del()</i> function for deleting an element from the binary search tree
<code>display(root);</code>	Calls the <i>display()</i> function for displaying the nodes of the binary search tree

## 6.7 TREE VARIANTS

Based on the concept of trees, binary trees and binary search trees various tree variants have been deduced. Each of these variants possesses distinct characteristics and serves specific purposes. For

example, balanced binary trees balance their nodes in such a way that the height of the tree is always kept to a minimum, thus ensuring better average case performance at the time of searching.

In the subsequent sections, we will learn about the various tree variants.

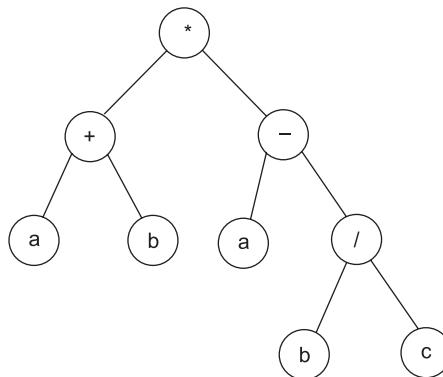
### 6.7.1 Expression Trees

Expression tree is nothing but a binary tree containing mathematical expression. The internal nodes of the tree are used to store operators while the leaf or terminal nodes are used to store operands. Various compilers and parsers use expression trees for evaluating arithmetic and logical expressions.

Consider the following expression:

$$(a+b)*(a-b/c)$$

The expression tree for the above expression is shown in Fig. 6.10.



**Fig. 6.10** Expression tree

As shown in the above tree, the internal nodes store the operators while the leaf nodes store the operands. While constructing a binary tree from a given expression, the following precedence rules are followed:

1. Parentheses are evaluated first.
2. The exponential expressions are evaluated next.
3. Then, division and multiplication operations are evaluated.
4. Finally, addition and subtraction operations are evaluated.

Representing an expression using a binary tree has another key advantage. By applying the various traversal methods we can deduce the other representations of an expression. For example, the preorder traversal of an expression tree derives its prefix notation.

Table 6.3 shows the various expression notations deduced after traversing the expression tree shown in Fig. 6.10.

**Table 6.3** Expression notations

Expression Notation	Traversal Method	Example (Refer to Fig. 6.10)
Prefix	Preorder	*+ab-a/bc
Infix	Inorder	a+b*a-a/c
Postfix	Postorder	ab+abc/-*

## 6.7.2 Threaded Binary Trees

Let us recall the structure declaration of a tree node described during the linked implementation of a binary tree:

```
struct bin_tree
{
    int INFO;
    struct node *LEFT, *RIGHT;
};
```

As we can see in the above declaration, each node in a binary tree has two pointer nodes associated with it, i.e., LEFT and RIGHT. Now, in case of leaf nodes, these pointers contain NULL values. Considering the number of leaf nodes that are there in a typical binary tree, this leads to a lot of memory space getting wasted. Threaded binary trees offer an innovative alternate to avoid this memory wastage.

In a threaded binary tree, all nodes that do not have a right child contain a pointer or a thread to its inorder successor. The address of the inorder successor node is stored in the RIGHT pointer. But, how do we distinguish between a normal pointer and a thread pointer? This is done with the help of a Boolean variable, as shown in the below node declaration of a threaded binary tree:

```
struct t_tree
{
    int INFO;
    struct node *LEFT, *RIGHT;
    boolean LThread, RThread;
};
```

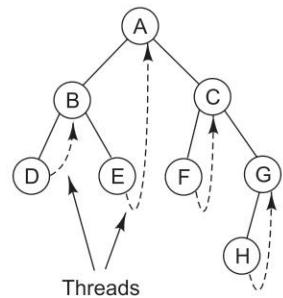


**Note** Just like a right thread points at the inorder successor, we can also make the left thread to point at the postorder successor so as to deduce the postorder traversal sequence.

Figure 6.11 shows a threaded binary tree.

As shown in the figure, nodes D, E, F and H contain threads to point at their inorder successors.

Now, what is the advantage of a threaded binary tree representation? Try to recall the algorithm for inorder traversal of a binary tree. The algorithm uses recursive function calls to determine the inorder traversal path. The execution of recursive function calls requires the use of stack and consumes both memory as well as time. The threaded tree traversal allows us to determine the inorder sequence using an iterative approach instead of a recursive approach.



**Fig. 6.11** Threaded binary tree

**Example 6.9** Write the algorithm for traversal of a threaded binary tree to generate the inorder sequence.

### Solution

```
inorder(node)
Step 1: Start
Step 2: Set current = leftmost(node)
//current refers to the current node
//leftmost function returns the left most node value in a subtree
Step 3: while current != NULL repeat Steps 4-7
Step 4: Display current
Step 5: If current->RThread != NULL goto Step 6 else goto Step 7
Step 6: Set current = current->RIGHT
Step 7: Set current = leftmost(current->RIGHT)
Step 8: Stop

leftmost (node)
Step 1: Start
Step 2: Set ptr = node
Step 3: if ptr = NULL goto Step 4 else goto Step 5
Step 4: Return NULL and goto Step 8
Step 5: while ptr->LEFT != NULL repeat Step 6
Step 6: Set ptr = ptr->LEFT
Step 7: Return ptr
Step 8: Stop
```

If we apply the above algorithm on the threaded binary tree shown in Fig. 6.11, then we will obtain the following inorder sequence:

D-B-E-A-F-C-H-G



### Check Point

1. In an expression tree, the internal nodes contain \_\_\_\_\_ while the leaf nodes contain \_\_\_\_\_.

Ans. operators, operands

2. In a threaded binary tree, a RIGHT thread points at the \_\_\_\_\_ successor of a node.

Ans. Inorder

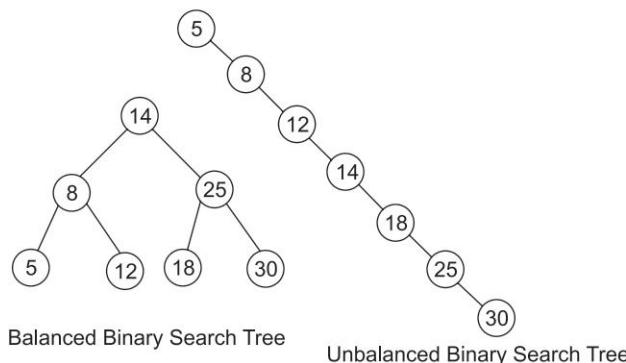


Fig. 6.12 Binary search trees

### 6.7.3 Balanced Trees

In the previous sections, we saw how nodes are added to a binary search tree. With each addition of a node in a tree, there is a possibility that the height of the tree may also get changed. The height of a tree has a direct affect on its efficiency to perform the search operation. For instance, consider the binary search trees shown in Fig. 6.12.

Both the binary search trees shown in the above figure contain the same nodes however the height of the first tree is 2 while that of the second tree is 6. To search element 30 in the above trees, we need to dig a lot deeper in the second tree as compared to the first tree. Thus, while implementing binary trees, it is important to keep the height of the tree in check.

There are various binary search trees that keep the tree balanced whenever a new node is added by shuffling the tree nodes appropriately. These are:

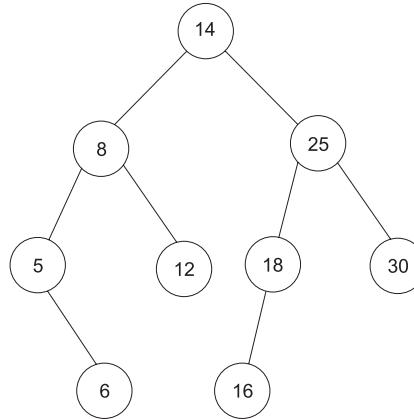
1. AVL tree
2. Red-Black tree

**1. AVL tree** AVL tree, also called *height-balanced tree* was defined by mathematicians Adelson, Velskii and Landis in the year 1962. The main characteristic of an AVL tree is that for all its nodes, the height of the left subtree and the height of the right subtree never differ by more than

At any point of time, an AVL tree node is in any one of the following states:

- (a) *Balanced* The height of left subtree is equal to the height of right subtree.
- (b) *Left heavy* The height of left subtree is one more than the height of right subtree.
- (c) *Right heavy* The height of right subtree is one more than the height of left subtree.

Figure 6.13 shows an AVL tree.



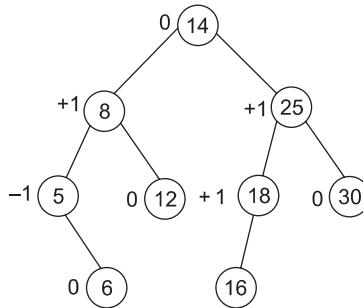
**Fig. 6.13** AVL tree

As shown in Fig. 6.13, the height of left and right subtrees of each node differs by not more than 1.

Now, how is an AVL tree created and maintained? This is done by associating a balance factor (BF) with each node that keeps a track of the height balance for that particular node. BF for a node is calculated by using the following formula:

$$BF = \text{Height of Left Subtree} - \text{Height of Right Subtree}$$

Let us apply the above formula to calculate the balance factor for each node of the AVL tree shown in Fig. 6.13. Figure 6.14 shows the updated AVL tree.



**Fig. 6.14** AVL tree with balance factors

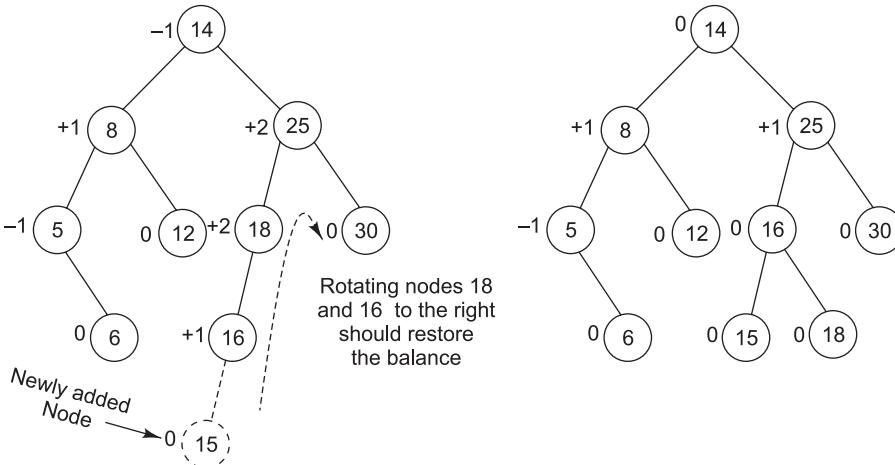
As shown in Fig. 6.14, the balance factors of all the nodes are not more than 1, which is the key characteristic of an AVL tree.

The structure declaration of an AVL tree node contains an additional field for storing the balance factor, as shown below:

```
struct avl_node
{
    int INFO;
    struct node *LEFT, *RIGHT;
    int BF;
};
```

Whenever a new node is inserted in an AVL tree, a slight disbalance is created at the point of insertion which reflects in the balance factors of the nodes in its preceding path till the root node. To restore the balance of the tree, left and right rotations are carried out to move the nodes towards the right or left. This is repeated until the balance factors of all the nodes are reduced below 1.

Figure 6.15 shows the insertion of node value 15 into the AVL tree shown in Fig. 6.14. It depicts how the disbalance resulting out of the insert operation is corrected.



**Fig. 6.15** Inserting an element in an AVL tree

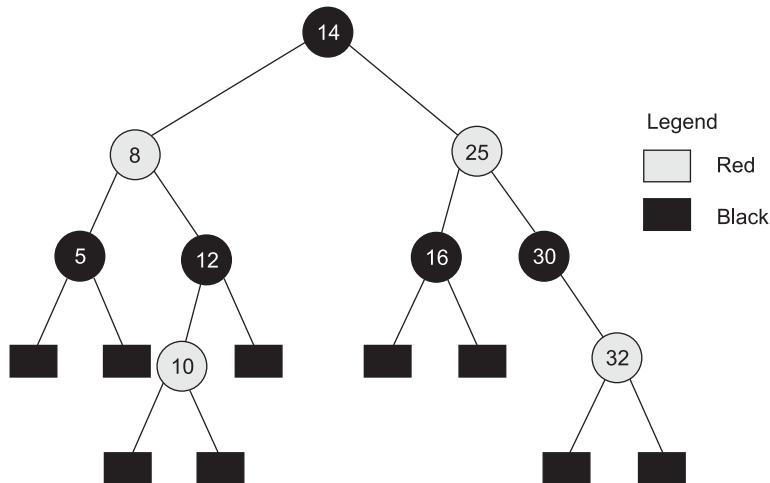
The delete operation follows a similar approach. A left or right rotation may need to be carried out if a node is deleted from an AVL tree.

**2. Red-Black tree** Red-Black tree is a self-balancing binary search tree that has an average running time of  $O(\log n)$  for insert, delete and search operations. As the name suggests, the red-black tree associates a color attribute with each node, which can possess only two values, red or black. That means each node in a red-black tree is either red or black colored. Apart from possessing the properties of a typical binary search tree, a red-black tree possesses the following properties:

- (a) Each node is either red or black in color.
- (b) The root node is black colored.
- (c) The leaf nodes are black colored. It includes the NULL children.
- (d) The child nodes of all red-colored nodes are black.
- (e) Each path from a given node to any of its leaf nodes contains equal number of black nodes. The number of such black nodes is also referred as black-height (bh) of the node.

The above properties ensure that the length of the longest path from the root node to a leaf node is less than roughly twice of the shortest path. This ensures that the balance of the tree is always kept under check. The key advantage of a red-black tree is that its worst case running time is better than most of the other binary search trees.

Figure 6.16 shows a red-black tree.



**Fig. 6.16** Red-Black tree

The insert and delete operations on a red-black tree require small number of rotations as well as change of colors of some of the nodes so that the tree complies with all the properties of a red-black tree. However, the average running time of these operations is  $O(\log n)$ .

#### 6.7.4 Splay Trees

The concept of splay trees is based on the assumption that when a particular element is accessed from a binary search tree then there are high chances that the same element would be accessed again in future. Now, if the element is placed deep in the tree then all such repetitive accesses would be inefficient. To make the repetitive accesses of a node efficient, splay tree shifts the accessed node towards the root

two levels at a time. This shifting is done through splay rotations. Table 6.4 shows the various types of splay rotations along with an illustration.

**Table 6.4 Splay rotations**

Splay Rotation	Occurrence	Illustration
Zig	When root node P is the parent of the node N being accessed.	
Zigzag	When node N is the right child of parent P, which itself is the left child of grandparent G. Or, when node N is the left child of parent P, which itself is the right child of grandparent G.	
Zigzig	When both node N and parent P are left or right child of grandparent G.	

Splay rotations ensure that all future accesses of a node are efficient as compared to its first time access. Let us now discuss what happens when typical tree-related operations are performed on a splay tree:

**1. Insert** New element is inserted at the root.

**2. Search** There are two possibilities:

(a) *Successful search* The searched node is moved to the root position.

(b) *Unsuccessful search* The last node accessed during the unsuccessful search operation is moved to the root position.

**3. Delete** The element to be deleted is first brought to the root position. After deleting the root node, the largest node in the left subtree is moved to the root position.

## 6.7.5 m-way Trees

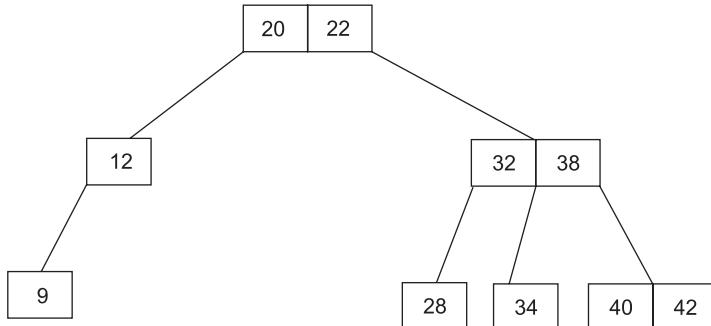
Binary search trees are more suitable for smaller data sets where the data is static. However, for large data sets which require dynamic access (example file storage); binary search trees are not exactly suitable. For such cases, the nodes of the tree are required to store large amounts of data. This is achieved with the help of m-way trees.

m-way search trees are an extension of binary search trees having the following properties:

1. Each node of the tree stores 1 to  $m-1$  number of keys.
2. The keys are stored in a sorted manner inside the node.

3. A node containing  $k$  values can have a maximum of  $k+1$  subtrees.
4. The subtree pointed by pointer  $T_i$  has values less than the key value of  $k_{i+1}$ .
5. All the subtrees are  $m$ -way trees.

Figure 6.17 shows a sample  $m$ -way tree.



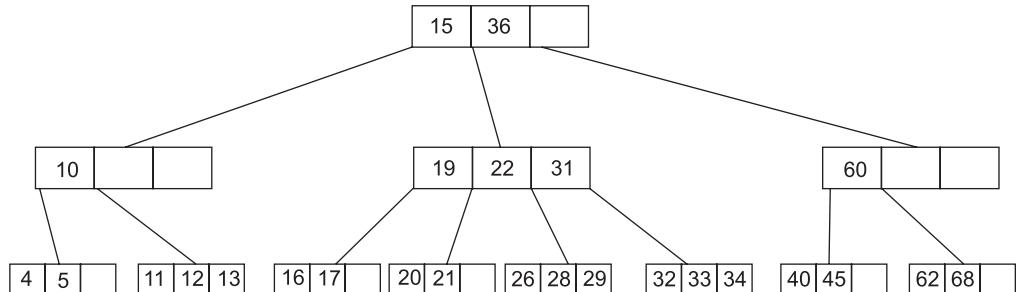
**Fig. 6.17** Sample  $m$ -way tree

(a) *B tree* To ensure efficiency while searching an  $m$ -way tree it is important to control its height. This is achieved with the help of *B tree*. A *B tree* is nothing but a height balanced  $m$ -way search tree.

A *B tree* of order  $m$  has the following properties:

- i. Root node is either a leaf node or it contains child nodes ranging from 2 to  $m$ .
- ii. All internal nodes contain a maximum of  $m-1$  keys.
- iii. Number of children of internal nodes ranges from  $m/2$  to  $m$ .
- iv. Number of keys stored in the leaf nodes ranges from  $(m-1)/2$  to  $m-1$ . All the keys are stored in a sorted manner.
- v. All leaf nodes are at the same depth.

Figure 6.18 shows a sample *B tree*.



**Fig. 6.18** Sample *B tree*

An element is inserted in a *B tree* by first identifying the location where the new node should be inserted. If the existing node is not full, the new element is inserted within the existing node and an appropriate pointer is created linking it with the parent node. However, if the exiting node is full then it is split into three parts. The middle part is accommodated with the parent node while the new element is inserted in one of the child nodes.

Similarly, deletion of an element from a *B tree* is done by first removing the element from a node and then carrying out appropriate redistributions to ensure that the tree stays true to its properties.

(b)  $B^+$  tree  $B^+$  tree is a variant of B tree that is mainly used for implementing index sequential access of records. The main difference between  $B^+$  tree and B tree is that in  $B^+$  tree data records are only stored in the leaf nodes. The internal nodes of a  $B^+$  tree are only used for storing the key values. The key values help in performing the search operation. If the target element is less than a key value then the search proceeds towards its left pointer. Similarly, if the target element is greater than a key value then the search proceeds towards its right pointer.

A  $B^+$  tree of order m has the following properties:

- The internal nodes contain up to  $m-1$  keys.
- The number of children of internal nodes lies between  $m/2$  and  $m$ .
- The subtree between keys  $k_1$  and  $k_2$  contains values  $v$  such that  $k_1 \leq v < k_2$ .
- All leaf nodes are at the same level.
- All the leaf nodes are sequentially connected through a linked list.

$B^+$  tree is typically used for implementing index sequential file organization in a database. The internal nodes are used for representing index values through which data records in the sequence set are accessed.

Figure 6.19 shows a sample  $B^+$  tree.

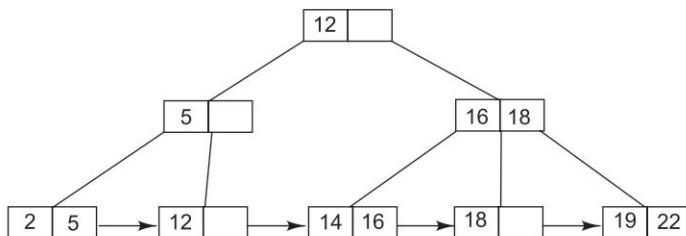


Fig. 6.19 Sample  $B^+$  tree



### Check Point

1. In which situation is a zig-zig splay rotation performed?

Ans. When both node N and parent P are left or right child of grandparent G.

2.  $B^+$  tree implementation helps in performing \_\_\_\_\_ search.

Ans. Index-sequential search.

## Summary

- ◆ Tree is a non-linear data structure which stores the data elements in a hierarchical manner.
- ◆ The top node of a tree is marked as a root node while the remaining nodes are partitioned across the subtrees present under the root node.
- ◆ A binary tree is a restricted form of a general tree that can have zero, one or two child nodes but not more than that.
- ◆ Traversal is the process of visiting the various elements of a data structure. Binary tree traversal can be performed using three methods: preorder, inorder and postorder.
- ◆ If N represents the parent node, L represents the left subtree and R represents the right subtree then

- o *preorder sequence* N–L–R
  - o *inorder sequence* L–N–R
  - o *postorder sequence* L–R–N
  - ◆ A binary search tree arranges its node elements in a sorted manner. The node elements in the left subtree are less than the parent node while the node elements in the right subtree are greater than or equal to the parent node.
  - ◆ Expression tree is a binary tree whose internal nodes store operators while the leaf or terminal nodes store the operands.
  - ◆ A threaded binary tree uses the empty NULL pointers of nodes to create threads to their inorder successors. This increases the inorder traversal efficiency by preventing the use of recursive function calls.
  - ◆ In an AVL tree, the height of the left subtree and the height of the right subtree differ by not more than 1. Keeping the height of the tree in check ensures that the search efficiency is optimized.
  - ◆ Red-Black tree is a self-balancing binary search tree that has an average running time of O(logn) for insert, delete and search operations. Each node in a red-black tree is colored either red or black.
  - ◆ m-way search trees are a generalized form of binary search trees that are used for storing large amounts of data. The two types of m-way trees are: B tree and B<sup>+</sup> tree.

## Key Terms

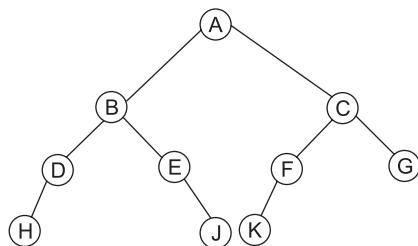
- ◆ **Root** It is the top node in a tree.
  - ◆ **Leaf** It is the terminal node that does not have any child nodes.
  - ◆ **Depth or Height** It is the maximum level of a node in a tree.
  - ◆ **INFO** Stores the value of the tree node.
  - ◆ **LEFT** Stores a pointer to the left child.
  - ◆ **RIGHT** Stores a pointer to the right child.
  - ◆ **Preorder** Traverses the tree in N–L–R order.
  - ◆ **Inorder** Traverses the tree in L–N–R order.
  - ◆ **Postorder** Traverses the tree in L–R–N order.
  - ◆ **Thread** Stores the address of the inorder successor of a node in a threaded binary tree.
  - ◆ **Balance factor** Height of Left Subtree – Height of Right Subtree

## Multiple-Choice Questions

- 6.4** The expression,  $(a+b)*(a-b)$  is stored in an expression tree. What will be its preorder sequence?
- $(a+b)*(a-b)$
  - $+ab-ab^*$
  - $ab+ab-^*$
  - $*+ab-ab$
- 6.5** Which of the following trees stores its elements in a sorted manner?
- General tree
  - Binary tree
  - Binary search tree
  - None of the above

## Review Questions

- What is a tree? Explain any five key terms associated with a tree.
- What is the difference between complete binary tree and perfect binary tree?
- What are the two types of balanced binary trees? Explain with the help of an illustration.
- What is the advantage of linked implementation of a binary tree over array implementation?
- What are the different types of tree traversal methods? Explain with the help of an example.
- Deduce the preorder and postorder sequences for the following binary tree:



- What is a binary search tree? Explain with the help of an example.
- What is an expression tree? Explain with the help of an example.
- What is a splay tree? Explain the different types of splay rotations.
- What is an m-way tree? Explain the two instances of m-way trees.

## Programming Exercises

- Write a function in C to count the number of nodes in a binary tree.
- Write a function in C to display the elements of a binary search tree in ascending order.
- Write a function in C that displays all the leaf nodes of a binary tree.
- Write a function in C that returns the degree of a binary tree node.
- Write a C function that transforms a given binary tree into a binary search tree.

## Answers to Multiple-Choice Questions

- 6.1 (b)      6.2 (b)      6.3 (c)      6.4 (d)      6.5 (c)



## **UNIT-IV**

---

# **Non Linear Data Structures – Graphs**

### **CHAPTER**

Chapter 7: Graphs



# 7

# GRAPHS

Chapter outline

- 7.1 Introduction
- 7.2 Basic Concept
- 7.3 Graph Terminology
- 7.4 Graph Implementation
  - 7.4.1 Implementing Graphs using Adjacency Matrix
  - 7.4.2 Implementing Graphs using Path Matrix
  - 7.4.3 Implementing Graphs using Adjacency List
- 7.5 Shortest Path Algorithm
- 7.6 Graph Traversal
  - 7.6.1 Breadth First Search
  - 7.6.2 Depth First Search

*Summary*

*Key Terms*

*Multiple-Choice Questions*

*Review Questions*

*Programming Exercises*

*Answers to Multiple-Choice Questions*



## 7.1 INTRODUCTION

Till now, we have learnt about different types of data structures, such as arrays, linked lists, trees, etc. In this chapter, we will learn about another important data structure called graph. It is similar to the mathematical graph structure, which comprises of a set of vertices connected with each other through edges. Some of the typical operations performed on a graph data structure include finding possible paths between two nodes and finding the shortest possible path.

Graph data structure finds its application in varied domains, such as computer network analysis, travel application, chip designing, gaming and so on.

In this chapter, we will learn how a graph data structure is represented and what algorithms are used for graph traversal. We will also learn about the shortest path algorithm that allows us to find the shortest path between two nodes.

## 7.2 BASIC CONCEPT

A graph  $G$  consists of the following elements:

- A set  $V$  of vertices or nodes where  $V = \{v_1, v_2, v_3, \dots, v_n\}$
- A set  $E$  of edges also called arcs where  $E = \{e_1, e_2, e_3, \dots, e_n\}$

Here,  $G = (V, E)$ .

Figure 7.1 shows a sample graph  $G$ .

In Fig. 7.1,  $e_1$  is an edge between  $v_1$  and  $v_2$  vertices while  $e_2$  is an edge between  $v_2$  and  $v_3$  vertices. Thus, we can generically represent an edge  $e$  as  $e = (u, v)$  where  $e$  connects both  $u$  and  $v$  vertices.

Now,  $e = (u, v)$  means the same thing as  $e = (v, u)$ . This means that the ordering of the vertices has no significance here. Thus, we can call the graph  $G$  as **undirected graph**.

If we replace each edge of the Graph  $G$  with arrows, then it will become a **directed graph** or **diagraph**, as shown in Fig. 7.2.

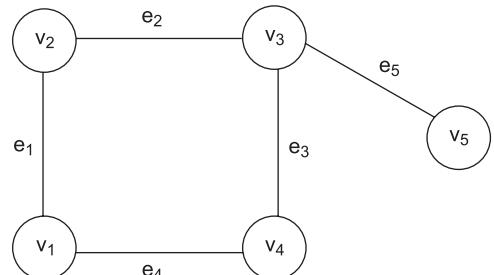


Fig. 7.1 Graph  $G$

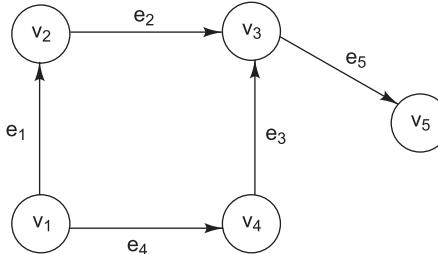


Fig. 7.2 Directed graph

In Fig. 7.2 graph, the set of vertices and edges are:

$$V(G) = \{v_1, v_2, v_3, v_4, v_5\}$$

$$E(G) = \{(v_1, v_2), (v_2, v_3), (v_1, v_4), (v_4, v_3), (v_3, v_5)\}$$

## 7.3 GRAPH TERMINOLOGY

There are a number of key terms associated with the concept of graphs. Table 7.1 explains some of these important key terms.

**Table 7.1** Graph terminology

Key Terms	Description
Adjacent node	If $e(u, v)$ represents an edge between $u$ and $v$ vertices then both $u$ and $v$ are called adjacent to each other. That means, $u$ is adjacent to $v$ and $v$ is adjacent to $u$ .
Predecessor node	If $e(u, v)$ represents a directed edge from $u$ to $v$ then $u$ is a predecessor node of $v$ .
Successor node	If $e(u, v)$ represents a directed edge from $u$ to $v$ then $v$ is a successor node of $u$ .
Degree	Degree of a vertex is the number of edges connected to a vertex. For example, in the graph shown in Fig. 7.1, the degree of vertex $v_3$ is 3.
Indegree	In a directed graph, indegree of a vertex is the number of edges ending at the vertex.
Outdegree	In a directed graph, outdegree of a vertex is the number of edges beginning at the vertex.
Path	A path is a sequence of vertices each adjacent to the next. For example, in the graph shown in Fig. 7.2, the path between the vertices $v_1$ and $v_5$ is $v_1-v_2-v_3-v_5$ .
Cycle	It is a path that starts and ends at the same vertex.
Loop	It is an edge whose endpoints are same that is, $e = (u, u)$ .
Weight	It is a non-negative number assigned to an edge. It is also called length.
Order	Order of a graph is the number of the vertices contained in the graph.
Labeled Graph	It is a graph that has labeled edges.
Weighted Graph	It is a graph that has weights assigned to each of its edges.
Connected Graph	It is an undirected graph in which there is a path between each pair of nodes.
Strongly Connected Graph	It is a directed graph in which there is a route between each pair of nodes.
Complete Graph	It is an undirected graph in which there is a direct edge between each pair of nodes.
Tree	It is a connected graph with no cycles.



### Note

*There are no standards defined related to the use of graph terminology. Thus, you may find the same concept being referred with different names at different places. For instance, a graph edge could also be referred as an arc or a link.*

## 7.4 GRAPH IMPLEMENTATION

Graphs are nothing but a collection of nodes and edges. Thus, while representing graphs in memory the only focus is on capturing details related to the different vertices and edges. Graphs can be implemented using the following methods:

1. Adjacency matrix
2. Path matrix
3. Adjacency list

### 7.4.1 Implementing Graphs Using Adjacency Matrix

Consider a graph  $G = (V, E)$  having  $N$  nodes. The adjacency matrix of graph  $G$  is defined as an  $N \times N$  matrix  $A$ , where:

1.  $A_{i,j} = 1$ , if there is an edge from vertex  $v_i$  to  $v_j$
2. and  $A_{i,j} = 0$ , if there is no edge from vertex  $v_i$  to  $v_j$

Let us try and understand the concept of adjacency matrix with the help of an example: Consider the graph shown in Fig. 7.3.

The adjacency matrix of the above graph will be

Here, 0s represent that there is no directed edge between the corresponding vertices while 1s represent the presence of a directed edge.

**Example 7.1** Write a program in C to represent a graph using adjacency matrix.

Program 7.1 represents the directed graph shown in Fig. 7.3 with the help of adjacency matrix.

**Program 7.1** C program to represent adjacency matrix

```
#include <stdio.h>
#include <conio.h>

void main()
{
    int A[5][5];
    int i,j;
    clrscr();
```



#### Check Point

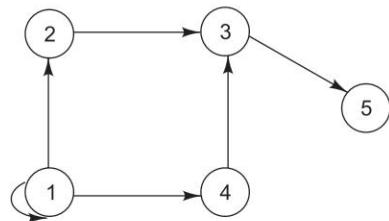
##### 1. In a weighted directed graph, what does the term weighted and directed signify?

**Ans.** The term weighted signifies that all the edges of the graph are assigned an integer number called weight. The term directed signifies that each edge of the graph is a pointed arrow that points from one vertex to the other.

##### 2. What is indegree?

**Ans.** Indegree of a vertex is the number of edges incident on the vertex.

$A_{i,j}$	1	2	3	4	5
1	1	1	0	1	0
2	0	0	1	0	0
3	0	0	0	0	1
4	0	0	1	0	0
5	0	0	0	0	0



**Fig. 7.3** Graph

```

for(i=0;i<5;i++)
for(j=0;j<5;j++)
A[i][j]=0; /*Initializing the array A*/

/*Creating adjacency matrix*/
A[0][0]=1;
A[0][1]=1;
A[0][3]=1;
A[1][2]=1;
A[2][4]=1;
A[3][2]=1;

/*Printing Adjacency Matrix*/
printf("Adjacency Matrix:");
for(i=0;i<5;i++)
{
printf("\n");
for(j=0;j<5;j++)
printf("%d ",A[i][j]);
}

getch();
}

```

*Since, we have already initialized all the elements of the adjacency matrix to 0, we do not need to explicitly write the assignment statements for non-edges.*

## Output

```

Adjacency Matrix:
1 1 0 1 0
0 0 1 0 0
0 0 0 0 1
0 0 1 0 0
0 0 0 0 0

```

## Program analysis

Key Statement	Purpose
int A[5][5];	Declares a two-dimensional array for storing the adjacency matrix
A[i][j]=0;	Initializes the array A before storing the adjacency matrix values

## Advantages and Disadvantages

The advantage of adjacency matrix representation of a graph is that it is simple to implement. However it also has certain disadvantages, such as the following:

1. It requires  $O(n^2)$  memory space even if the adjacency matrix is sparse.
2. It proves to be an inefficient representation for graphs that have large number of vertices.

## 7.4.2 Implementing Graphs Using Path Matrix

Consider a digraph  $G = (V, E)$  having  $N$  nodes. The path matrix of graph  $G$  is defined as an  $N \times N$  matrix  $P$ , where

1.  $P_{i,j} = 1$ , if there is a path from vertex  $v_i$  to  $v_j$ , and
2.  $P_{i,j} = 0$ , if there is no path from vertex  $v_i$  to  $v_j$ .

Now, the path matrix  $P$  can be deduced using the adjacency matrix of  $G$ , as depicted below:

$$P_N = A + A^2 + A^3 + \dots + A^N$$

Here,  $A^2$  is the square of the adjacency matrix  $A$ ,  $A^3$  is the cube of  $A$ , and so on. All the non-zero entries resulting from the addition operation above are replaced by 1 to arrive at the path matrix.

This method of deriving the path matrix by computing powers of adjacency matrix is not very efficient, as it requires performing a number of matrix multiplication operations. Warshall has suggested a more simplified method of deriving the path matrix from the adjacency matrix. Warshall's method determines the presence of a path between  $v_i$  and  $v_j$  by

1. identifying a direct path from  $v_i$  to  $v_j$ , and
2. identifying an indirect path from  $v_i$  and  $v_j$  that is, a path from  $v_i$  to  $v_k$  and  $v_k$  to  $v_j$ .

That is,  $P_{i,j} = P_{i,j} \text{ OR } (P_{i,k} \text{ AND } P_{k,j})$

Here, OR represents the logical OR operation and AND represents the logical AND operation.

**Example 7.2** Write the Warshall's algorithm for deriving the path matrix of a digraph  $G$ .

```
path_matrix(Adjacency Matrix A[], N)
Step 1: Start
Step 2: Set P[] = A[]
Step 3: Set i = j = k = 1
Step 4: Repeat Steps 5-10 while k <= N
Step 5: Repeat Steps 6-9 while i <= N
Step 6: Repeat Steps 7-8 while j <= N
Step 7: P[i,j] = P[i,j] OR (P[i,k] AND P[k,j])
Step 8: j = j + 1
Step 9: i = i + 1
Step 10: k = k + 1
Step 11: Display path matrix P[]
Step 12: Stop
```

**Example 7.3** Modify the program shown in Example 7.1 and apply Warshall's algorithm to derive the path matrix of a diagraph.

Program 7.2 uses Warshall's algorithm to derive the path matrix of the digraph shown in Fig. 7.3.

**Program 7.2** *Deriving path matrix using Warshall's algorithm*

```
#include <stdio.h>
#include <conio.h>

int AND(int, int); /* Function prototype for performing logical AND
operation*/
int OR(int, int); /* Function prototype for performing logical OR operation*/
```

```

void main()
{
    int A[5][5], P[5][5];
    int i,j,k;
    clrscr();

    for(i=0;i<5;i++)
        for(j=0;j<5;j++)
            A[i][j]=0;

    /*Creating adjacency matrix*/
    A[0][0]=1;
    A[0][1]=1;
    A[0][3]=1;
    A[1][2]=1;
    A[2][4]=1;
    A[3][2]=1;

    /*Printing adjacency matrix*/
    printf("Adjacency Matrix: \n");
    for(i=0;i<5;i++)
    {
        printf("\n");
        for(j=0;j<5;j++)
            printf("%d ",A[i][j]);
    }

    for(i=0;i<5;i++)
        for(j=0;j<5;j++)
            P[i][j]=A[i][j];

    /*Creating path matrix*/
    for(k=0;k<5;k++)
        for(i=0;i<5;i++)
            for(j=0;j<5;j++)
                P[i][j]=OR(P[i][j],AND(P[i][k],P[k][j]));

    /*Printing path matrix*/
    printf("\n\nPath Matrix: \n");
    for(i=0;i<5;i++)
    {
        printf("\n");
        for(j=0;j<5;j++)
            printf("%d ",P[i][j]);
    }

    getch();
}

```

*Applying Warshall's method.*

```

int AND(int x, int y)
{
    return (x*y);
}

int OR(int x, int y)
{
    if(x==0 && y==0)
        return(0);
    else
        return(1);
}

```

*Since the operands associated with the AND and OR operations are integers, we cannot apply the relational operators of C to obtain the desired result.*

## Output

Adjacency Matrix:

```

1 1 0 1 0
0 0 1 0 0
0 0 0 0 1
0 0 1 0 0
0 0 0 0 0

```

Path Matrix:

```

1 1 1 1 1
0 0 1 0 1
0 0 0 0 1
0 0 1 0 1
0 0 0 0 0

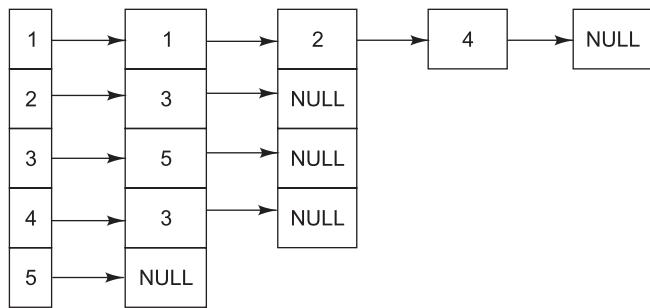
```

## Program analysis

Key Statement	Purpose
int A[5][5], P[5][5];	Declares two-dimensional arrays for storing adjacency and path matrices
P[i][j]=A[i][j];	Copies the adjacency matrix values into the path matrix array
P[i][j]=OR(P[i][j], AND(P[i][k], P[k][j]));	Applies the Warshall's method to generate the path matrix values
printf("%d ",P[i][j]);	Prints the path matrix values

### 7.4.3 Implementing Graphs Using Adjacency List

Adjacency list is a linked representation of a graph. It consists of a list of graph nodes with each node itself consisting of a linked list of its neighboring nodes. Figure 7.4 shows the adjacency list of the directed graph shown in Fig. 7.3.



**Fig. 7.4** Adjacency list

The adjacency list shown above contains five nodes with each node pointing towards its successor nodes.

**Example 7.4** Write a program in C to represent a graph using adjacency list.

Program 7.3 represents a directed graph using adjacency list.

**Program 7.3** Representing graph using adjacency list

```

#include<stdio.h>
#include<conio.h>
#include<stdlib.h>

struct vertex
{
    struct vertex *edge[10];
    int id;
}node[10];

void display(int);

void main()
{
    int i,j,N;
    char ch;
    clrscr();

    i=j=N=0;
    printf("Enter number of graph vertices: ");
    scanf("%d",&N);

    for(i=0;i<N;i++)
    {
        node[i].id=i;

        fflush(stdin);
    }
}
  
```

```

for(j=0;j<N;j++)
{
fflush(stdin);
printf("Edge from %d to %d? (y/n): ",i+1,j+1);
scanf("%c",&ch);
if(ch=='y')
node[i].edge[j]=&node[j];
else
node[i].edge[j]=NULL;
}

display(N);
getch();
}

void display(int num)
{
int i,j;
printf("\n");
for(i=0;i<num;i++)
{
printf("Edges of node[%d] are: ",i+1);
for(j=0;j<num;j++)
{
if(node[i].edge[j]==NULL)
continue;
printf("(%d-%d) ",i+1,node[i].edge[j]->id+1);
}
printf("\n");
}
}

```

## Output

```

Enter number of graph vertices: 3
Edge from 1 to 1? (y/n): n
Edge from 1 to 2? (y/n): y
Edge from 1 to 3? (y/n): y
Edge from 2 to 1? (y/n): y
Edge from 2 to 2? (y/n): n
Edge from 2 to 3? (y/n): y
Edge from 3 to 1? (y/n): n
Edge from 3 to 2? (y/n): y
Edge from 3 to 3? (y/n): y

```

```

Edges of node[1] are: (1-2) (1-3)
Edges of node[2] are: (2-1) (2-3)
Edges of node[3] are: (3-2) (3-3)

```

## Program analysis

Key Statement	Purpose
<pre>struct vertex {     struct vertex *edge[10];     int id; }node[10];</pre>	Declares a structure to represent a graph node
<b>void display(int);</b>	Declares the function prototype for displaying the graph represented by adjacency list
<pre>if(ch=='y')     node[i].edge[j]=&amp;node[j]; else     node[i].edge[j]=NULL;</pre>	Stores information related to edges that is whether or not an edge is present between two vertices of the graph



### Note

*Adjacency matrix and path matrix are both examples of sequential representation of graphs. Adjacency list is an example of linked representation of graphs.*

## 7.5 SHORTEST PATH ALGORITHM

One of the most common problems associated with graphs is to find the shortest path from one node to the other. It finds its relevance in a number of real-life applications. For example, consider a scenario where a freight carrier departing from Delhi is required to drop consignments at Lucknow, Jaipur, Ahmedabad, Pune, and Hyderabad airports. In this situation, the route taken by the carrier is determined by assessing the distance between each of these cities. The final route undertaken should be the shortest path that covers all these cities. We can relate this scenario with a graph data structure where each city represents a graph node while the distance between the cities represents the weight of the edges.

Consider the weighted digraph shown in Fig. 7.5.

The weight matrix of the above graph will be

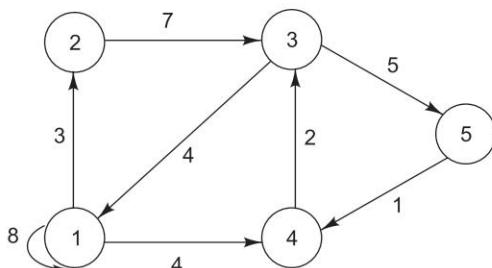


Fig. 7.5 Weighted digraph



### Check Point

1. Write the Warshall's relation for computing a path matrix?

Ans.  $P[i,j] = P[i,j] \text{ OR } (P[i,k] \text{ AND } P[k,j])$

2. The end of adjacency list of a graph node is signified by \_\_\_\_\_.

Ans. NULL pointer

$W_{i,j}$	1	2	3	4	5
1	8	3	0	4	0
2	0	0	7	0	0
3	4	0	0	0	5
4	0	0	2	0	0
5	0	0	0	1	0

Here,  $W_{i,j}$  represents the weight of the edge from node  $v_i$  to  $v_j$ . The value 0 signifies that there is no direct edge between the corresponding nodes. Now, a modification of the Warshall's algorithm can be applied to the weight matrix to derive the shortest path matrix  $SP$  that represents the weight of the shortest possible path between any two nodes of a graph.

It begins with replacing all 0's in the weight matrix with  $\infty$ , as shown below.

$SP_{i,j}$	1	2	3	4	5
1	8	3	8	4	8
2	8	8	7	8	8
3	4	8	8	8	5
4	8	8	2	8	8
5	8	8	8	1	8

Now, the following relation is applied to arrive at the shortest path matrix:

$$SP_{i,j} = \text{Minimum of } (SP_{i,j}, SP_{i,k} + SP_{k,j})$$

The shortest path matrix obtained after applying the above relation for each graph node is

$SP_{i,j}$	1	2	3	4	5
1	8	3	6	4	11
2	11	14	7	13	12
3	4	7	8	6	5
4	6	9	2	8	7
5	7	10	3	1	8

**Example 7.5** Write the modified Warshall's algorithm for deriving the shortest path matrix of a digraph G.

```
shortest_path_matrix(Path Matrix P[], N)
Step 1: Start
Step 2: Set i = j = 1
Step 3: Repeat Steps 4-9 while i<=N
Step 4: Repeat Steps 5-8 while j<=N
Step 5: if P[i,j]=0 goto Step 6 else goto Step 7
Step 6: Set SP[i,j]= 8
Step 7: Set SP[i,j]=P[i,j]
Step 8: j = j + 1
Step 9: i = i + 1
Step 10: Set i = j = k = 1
Step 11: Repeat Steps 12-17 while k<=N
Step 12: Repeat Steps 13-16 while i <=N
Step 13: Repeat Steps 14-15 while j <=N
Step 14: SP[i,j] = MINIMUM(SP[i,j], SP[i,k]+SP[k,j])
Step 15: j = j + 1
Step 16: i = i + 1
Step 17: k = k + 1
```

```
Step 18: Display shortest path matrix SP[]
Step 19: Stop
```

**Example 7.6** Write a program in C to deduce the shortest path matrix of a weighted digraph G. Program 7.4 uses modified Warshall's algorithm to derive the shortest path matrix of the weighted digraph shown in Fig. 7.5.

**Program 7.4** Shortest path matrix of the weighted digraph using modified Warshall's algorithm

```
#include <stdio.h>
#include <conio.h>

int MIN(int, int); /*Function prototype for computing the minimum among
two integers*/

void main()
{
    int P[5][5], SP[5][5];
    int i,j,k;
    clrscr();

    for(i=0;i<5;i++)
        for(j=0;j<5;j++)
            P[i][j]=0;

    P[0][0]=8;
    P[0][1]=3;
    P[0][3]=4;
    P[1][2]=7;
    P[2][0]=4;
    P[2][4]=5;
    P[3][2]=2;
    P[4][3]=1;

    printf("Path Matrix: \n");
    for(i=0;i<5;i++)
    {
        printf("\n");
        for(j=0;j<5;j++)
            printf("%d\t",P[i][j]);
    }

    for(i=0;i<5;i++)
        for(j=0;j<5;j++)
            if(P[i][j]==0)
                SP[i][j]=999;
            else
                SP[i][j]=P[i][j];
}
```

```

for (k=0;k<5;k++)
for (i=0;i<5;i++)
for (j=0;j<5;j++)
SP[i][j]=MIN(SP[i][j],SP[i][k]+SP[k][j]);

printf("\n\nShortest Path Matrix: \n");
for (i=0;i<5;i++)
{
printf("\n");
for (j=0;j<5;j++)
printf("%d\t",SP[i][j]);
}

getch();
}

int MIN(int x, int y)
{
    if (x<=y)
        return(x);
    else
        return(y);
}

```

*The reason for not including this code within the main program is to ensure modularity and make the program less complex.*

## Output

Path Matrix:

8	3	0	4	0
0	0	7	0	0
4	0	0	0	5
0	0	2	0	0
0	0	0	1	0

Shortest Path Matrix:

8	3	6	4	11
11	14	7	13	12
4	7	8	6	5
6	9	2	8	7
7	10	3	1	8

## Program analysis

Key Statement	Purpose
<b>int P[5][5], SP[5][5];</b>	Declares two-dimensional arrays for storing path and shortest path matrices
<b>SP[i][j]=MIN(SP[i][j], SP[i][k]+SP[k][j]);</b>	Applies the modified Warshall's algorithm for generating the shortest path matrix values
<b>printf("%d\t",SP[i][j]);</b>	Prints the shortest path matrix values

## 7.6 GRAPH TRAVERSAL

One of the common tasks associated with graphs is to traverse or visit the graph nodes and edges in a systematic manner. There are two methods of traversing a graph:

1. Breadth First Search (BFS)
2. Depth First Search (DFS)

Both these methods consider the graph nodes to be in one of the following states at any given point of time:

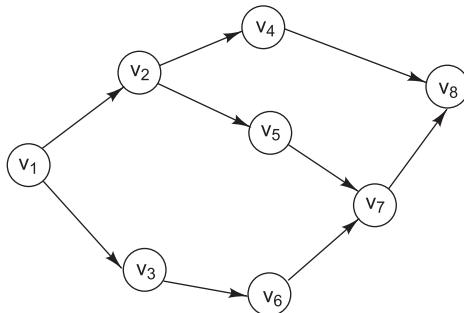
1. Ready state
2. Waiting state
3. Processed state

The state of a node keeps on changing as the graph traversal progresses. Once the state of a node becomes processed, it is considered as traversed or visited.

### 7.6.1 Breadth First Search

The BFS method begins with analyzing the starting node and then progresses by analysing its adjacent or neighbouring nodes. Once all the neighbouring nodes of the starting node are analyzed, the algorithm starts analyzing the neighboring nodes of each of the analyzed neighboring nodes. This method of graph traversal requires frequent backtracking to the already analyzed nodes. As a result, a data structure is required for storing information related to the neighboring nodes. The BFS method uses the queue data structure for storing the nodes data.

Consider the graph shown in Fig. 7.6.



**Fig. 7.6** Graph traversal

The BFS traversal sequence for the above graph will be:  $v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8$ . Another BFS traversal sequence can be:  $v_1, v_3, v_2, v_6, v_5, v_4, v_7, v_8$ .

**Example 7.7** Write an algorithm for the BFS graph traversal method.

```
BFS(adj[], status[], queue[], N)
Step 1: Start
Step 2: Set status[] = 1
Step 3: Push(queue, v1)
```

```

Step 4: Set status[v1]=2
Step 5: Repeat Step 6-11 while queue[] is not empty
Step 6: V = Pop(queue) ←
Step 7: status[V]=3
Step 8: Repeat Step 9-11 while adj(V) is not empty
Step 9: If adj(V) = 1 goto step 10 else goto step 8
Step 10: Push(queue, adj(V))
Step 11: Set adj[v]=2
Step 12: Stop

```

Here, the pop operation signifies the processing of a graph node.



**Note** *Pop means removing an element from the queue while push means inserting an element into the queue.*

## 7.6.2 Depth First Search

Unlike the BFS traversal method, which visits the graph nodes level by level, the DFS method visits the graph nodes along the different paths. It begins analyzing the nodes from the start to the end node and then proceeds along the next path from the start node. This process is repeated until all the graph nodes are visited.

The DFS method also requires frequent backtracking to the already analyzed nodes. It uses the stack data structure for storing information related to the previous nodes.

Let us again consider the graph shown in Fig. 7.6. The DFS traversal sequence for this graph will be:  $v_1, v_2, v_4, v_8, v_5, v_7, v_3, v_6$ . Another DFS traversal sequence can be:  $v_1, v_3, v_6, v_7, v_8, v_2, v_5, v_4$ .

**Example 7.8** Write an algorithm for the DFS graph traversal method.

```

DFS(adj[], status[], stack[], N)
Step 1: Start
Step 2: Set status[] = 1
Step 3: Push(stack, v1)
Step 4: Set status[v1]=2
Step 5: Repeat Step 6-11 while stack[] is not empty
    Step 6: V = Pop(stack) ←
    Step 7: status[V]=3
    Step 8: Repeat Step 9-11 while adj(V) is not empty
        Step 9: If adj(V) = 1 goto step 10 else goto step 8
        Step 10: Push(stack, adj(V))
        Step 11: Set adj[v]=2
    Step 12: Stop

```

Here, the pop operation signifies the processing of a graph node.



### Check Point

#### 1. What is BFS?

**Ans.** It is the method of traversing a graph in such a manner that all the vertices at a particular level are visited first before proceeding onto the next level.

#### 2. What is DFS?

**Ans.** It is the method of traversing a graph in such a manner that all the vertices in a given path (starting from the first node) are visited first before proceeding onto the next path.

## ▶▶▶ Summary



- ◆ A graph  $G(V, E)$  consists of the following elements:
  - A set  $V$  of vertices or nodes where  $V = \{v_1, v_2, v_3, \dots, v_n\}$
  - A set  $E$  of edges also called arcs where  $E = \{e_1, e_2, e_3, \dots, e_n\}$
- ◆ A graph can be implemented in three ways: adjacency matrix, path matrix, and adjacency list.
- ◆ Adjacency matrix and path matrix are the sequential methods of representing a graph. Adjacency matrix signifies whether there is an edge between any two vertices of the graph. Path matrix signifies whether there is a path between any two vertices of the graph.
- ◆ Adjacency list is a linked representation of a graph. It consists of a list of graph nodes with each node itself consisting of a linked list of its neighboring nodes.
- ◆ Breadth First Search or BFS is the method of traversing a graph in such a manner that all the vertices at a particular level are visited first before proceeding onto the next level.
- ◆ Depth First Search or DFS is the method of traversing a graph in such a manner that all the vertices in a given path (starting from the first node) are visited first before proceeding onto the next path.

## ▶▶▶ Key Terms

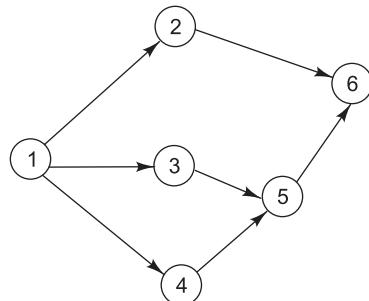


- ◆ **Weighted graph** It signifies that all the edges of the graph are assigned an integer number called weight.
- ◆ **Directed** It signifies that each edge of the graph is a pointed arrow that points from one vertex to the other.
- ◆ **Adjacency matrix** It is an  $N \times N$  matrix containing 1s for all the direct edges of the graph and containing 0s for all the non-edges.
- ◆ **Path matrix** It is an  $N \times N$  matrix containing 1s for all the existing paths in a graph and containing 0s otherwise.
- ◆ **Adjacency list** It a list of graph nodes with each node itself consisting of a linked list of its neighboring nodes.

## Multiple-Choice Questions

- 7.1 Which of the following is not true for graph?
- It is a set of vertices and edges.
  - All of its vertices are reachable from any other vertex
  - It can be represented with the help of an  $N \times N$  matrix.
  - All of the above are true
- 7.2 As per Warshall's method, which of the following is the correct relation for computing the path matrix?
- $P_{i,j} = P_{i,j} \text{ OR } (P_{i,k} \text{ AND } P_{k,j})$
  - $P_{i,j} = P_{i,j} \text{ AND } (P_{i,k} \text{ OR } P_{k,j})$
  - $P_{i,j} = P_{i,k} \text{ AND } (P_{k,j} \text{ OR } P_{i,j})$
  - None of the above

- 7.3** As per modified Warshall's algorithm, which of the following is the correct relation for computing the shortest path between two vertices in a graph?
- $SP_{i,j} = \text{Minimum of } (SP_{i,j}, SP_{i,k} + SP_{k,j})$
  - $SP_{i,j} = \text{Maximum of } (SP_{i,j}, SP_{i,k} + SP_{k,j})$
  - $SP_{i,j} = \text{Minimum of } (SP_{i,k}, SP_{k,j} + SP_{i,j})$
  - None of the above
- 7.4** The number of edges incident on a vertex is referred as \_\_\_\_\_.
- Degree
  - Indegree
  - Order
  - Outdegree
- 7.5** Identify the BFS path for the following graph:
- 1–2–3–4–6–5
  - 1–4–3–2–6–5
  - 1–2–3–4–5–6
  - None of the above



## Review Questions

- What is a graph? Explain with an example.
- List and explain any five key terms associated with graphs.
- What are the different methods of representing a graph?
- What is an adjacency matrix? How can you derive a path matrix from an adjacency matrix?
- Explain adjacency list implementation of a graph with the help of an example.
- What is the significance of computing the shortest path in a graph? Explain with the help of an example.
- Write the modified Warshall's algorithm for computing the shortest path between two nodes of a graph.
- What is BFS? Explain with the help of an example.
- What is DFS? Explain with the help of an example.

## Programming Exercises

- Write a C function to deduce the adjacency matrix for a given directed graph G.
- Write a C function that takes as input the adjacency matrix and applies Warshall's algorithm to generate the corresponding path matrix.
- Write a C program to implement a 3-node directed graph using adjacency list.
- Write a C function that takes as input the path matrix and applies the shortest path algorithm to generate the corresponding shortest path matrix.

## Answers to Multiple-Choice Questions

7.1 (b)

7.2 (a)

7.3 (a)

7.4 (b)

7.5 (c)

## **UNIT-V**

---

# **Searching, Sorting and Hashing Techniques**

### **CHAPTER**

Chapter 8: Sorting and Searching



# 8

## SORTING AND SEARCHING

Chapter outline

- 8.1 Introduction
- 8.2 Sorting Techniques
  - 8.2.1 Selection Sort
  - 8.2.2 Insertion Sort
  - 8.2.3 Bubble Sort
  - 8.2.4 Quick Sort
  - 8.2.5 Merge Sort
  - 8.2.6 Bucket Sort
- 8.3 Searching Operations
  - 8.3.1 Linear Search
  - 8.3.2 Binary Search
  - 8.3.3 Hashing

*Summary*

*Key Terms*

*Multiple-Choice Questions*

*Review Questions*

*Programming Exercises*

*Answers to Multiple-Choice Questions*



## 8.1 INTRODUCTION

Sorting and searching are two of the most common operations performed by computers all around the world. The sorting operation arranges the numerical and alphabetical data present in a list, in a specific order or sequence. Searching, on the other hand, locates a specific element across a given list of elements. At times, a list may require sorting before the search operation can be performed on it.

A telephone directory is one such example where both sorting and searching techniques are applied. The names of telephone subscribers are first alphabetically sorted and then posted on to the telephone directory. If one needs to search the telephone number of a particular subscriber in the telephone directory then it can be easily achieved by looking up the directory on the basis of the subscriber name. Now, consider the same scenario in the absence of a sorted list of subscribers. It would become very tough and painstaking to search the subscriber name in a directory where names are posted in a random fashion without any definite order.

There are a number of sorting techniques that can be employed to sort a given list of data elements. The suitability of a specific technique in a specific situation depends on a number of factors, such as

1. size of the data structure,
2. algorithm efficiency, and
3. programmer's knowledge of the technique.

While all the sorting methods produce the same result, that is a list of sorted elements, it is one or more of the above factors that play an important role in choosing a specific sorting technique in a given situation.

In this chapter, we will discuss the various searching and sorting methods.

## 8.2 SORTING TECHNIQUES

Consider a list L containing n elements, as shown below.

$$L_1, L_2, L_3, \dots, L_n$$

Now, there are  $n!$  ways in which the elements can be arranged within the list. We can apply a sorting technique to the list L to arrange the elements in either ascending or descending order.

If we sort the list in ascending order, then

$$L_1 \leq L_2 \leq L_3 \dots \leq L_n$$

Alternatively, if we arrange the list in descending order, then

$$L_1 \geq L_2 \geq L_3 \dots \geq L_n$$

**Example 8.1** Consider an array A containing five elements, as shown below.

	22	77	3	-1	5	
A[0]	A[1]	A[2]	A[3]	A[4]		

What would be the resultant array if it is sorted in  
ascending order  
descending order

**Solution** Array A sorted in ascending order.

	-1	3	5	22	77	
A[0]	A[1]	A[2]	A[3]	A[4]		

Array A sorted in descending order.

	77	22	5	3	-1	
A[0]	A[1]	A[2]	A[3]	A[4]		

As already explained, there are a number of methods that can be used to sort a given list of elements. We will discuss the following sorting methods in the forthcoming sections:

1. Selection sort
2. Insertion sort
3. Bubble sort
4. Quick sort
5. Merge sort
6. Bucket sort



### Mind Jog

**What is the internal sorting?**

All sorting techniques which require the data set to be present in the main memory are referred as internal sorting techniques.



#### Note

The application of a sorting technique is not restricted to an array or a list alone. In fact, we may apply sorting to other data structures such as structures or linked lists provided there is a subelement in the data structure based on which sorting can be performed.

### 8.2.1 Selection Sort

Selection sort is one of the most basic sorting techniques. It works on the principle of identifying the smallest element in the list and moving it to the beginning of the list. This process is repeated until all the elements in the list are sorted.

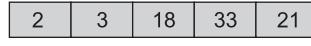
Let us consider an example where a list L contains five integers stored in a random fashion, as shown in Fig. 8.1.

18	3	2	33	21
List L				

**Fig. 8.1** List of integers

Now, if the list L is sorted using selection sort technique then first of all the first element in the list, i.e., 18 will be selected and compared with all the remaining elements in the list. The element which is found to be the lowest amongst the remaining set of elements will be swapped with the first element. Then, the second element will be selected and compared with the remaining elements in the list. This process is repeated until all the elements are rearranged in a sorted manner. Table 8.1 illustrates the sorting of list L in ascending order using selection sort.

**Table 8.1** Selection sort

Pass	Comparison	Resultant Array
1		
2		
3		
4		

 → denotes the currently selected element  
 → denotes the smallest element identified in the current pass

A single iteration of the selection sorting technique that brings the smallest element at the beginning of the list is called a pass. As we can see in the above table, four passes were required to sort a list of five elements. Hence, we can say that selection sort requires  $n-1$  passes to sort an array of  $n$  elements.

**Example 8.2** Write an algorithm to perform selection sort on a given array of integers.

```

selection(arr[], size)
Step 1: Start
Step 2: Set i = 0, loc = 0 and temp = 0
Step 3: Repeat Steps 4-6 while i < size
Step 4: Set loc = Min(arr, i, size)
Step 5: Swap the elements stored at arr[i] and a[loc] by performing the
following steps
  I Set temp = a[loc]
  II Set a[loc] = a[i]
  III Set a[i]=temp
Step 6: Set i = i +1
Step 8: Stop

Min(array[], LB, UB)
Step 1: Start
Step 2: Set m = LB
Step 3: Repeat Steps 4-6 while LB < U B
Step 4: if array[LB] < array[m] goto Step 5 else goto Step 6
Step 5: Set m = LB
Step 6: Set LB = LB +1
Step 7: Return m
Step 8: Stop
  
```

**Example 8.3** Write a C program to perform selection sort on an array of N elements.

Program 8.1 implements selection sorting technique in C. It uses the algorithm depicted in Example 8.2.

### Program 8.1 Selection sort

```
/*Program for performing selection sort*/
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>

void selection(int*, int); /*Function prototype for performing selection
sort*/
int Min(int*,int,int); /*Function prototype for finding minimum element in
the array*/
void main()
{
    int *arr;
    int i, N;
    clrscr();

    printf("Enter the number of elements in the array:\n");
    scanf("%d", &N);

    arr = (int*) malloc(sizeof(int)); /*Dynamic allocation of memory for the
array*/
    printf("Enter the %d elements to sort:\n",N);
    for (i=0;i<N;i++)
        scanf("%d", &arr[i]); /*Reading array elements*/
    selection(arr,N); /*Calling selection function*/

    printf("\nThe sorted elements are:\n");
    for(i=0;i<N;i++)
        printf("%d\n",arr[i]); /*Printing sorted array*/

    getch();
}

void selection(int *a, int size)
{
    int i=0,loc=0,temp=0;
    for(i=0;i<size;i++)
    {
        loc=Min(a,i,size); /*Calling Min function*/
        /*Swapping array elements*/
        temp=a[loc];
        a[loc]=a[i];
        a[i]=temp;
    }
}
```

*Function prototypes are declared globally to allow one function call the other.*

*The malloc function allocates only that much amount of memory space as is required for holding the array elements.*

```

int Min(int *array, int LB, int UB)
{
    int m=LB;
    /*Finding location of smallest element*/
    while(LB<UB)
    {
        if(array[LB]<array[m])
            m=LB;
        LB++;
    }
    return(m);
}

```

## Output

Enter the number of elements in the array:

5

Enter the 5 elements to sort:

18

3

2

33

21

The sorted elements are:

2

3

18

21

33

## Program analysis

Key Statement	Purpose
<code>loc=Min(a,i,size);</code>	Calls the <i>Min()</i> function to identify the location of the smallest element
<code>temp=a[loc]; a[loc]=a[i]; a[i]=temp;</code>	Swaps the array elements to move the smaller elements towards the start of the array

**Efficiency of Selection Sort** Assume that an array containing  $n$  elements is sorted using selection sort technique.

Now, the number of comparisons made during first pass =  $n-1$

Number of comparisons made during second pass =  $n-2$

Number of comparisons made during last pass = 1

$$\begin{aligned}
 \text{So, total number of comparisons} &= (n-1) + (n-2) + \dots + 1 \\
 &= n * (n-1) / 2 \\
 &= O(n^2)
 \end{aligned}$$

Thus, efficiency of selection sort =  $O(n^2)$

**Advantages and Disadvantages** Some of the key advantages of selection sorting technique are:

1. It is one of the simplest of sorting techniques.
2. It is easy to understand and implement.
3. It performs well in case of smaller lists.
4. It does not require additional memory space to perform sorting.

The disadvantages associated with selection sort that prevent the programmers from using it often are as follows:

1. The efficiency of  $O(n^2)$  is not well suited for large sized lists.
2. It does not leverage the presence of any existing sort pattern in the list.



#### Note

*Selection sort is an internal sorting technique and requires the entire data structure to be present in the main memory while performing sorting. As a result, it is not well suited for sorting large sized data structures.*

### 8.2.2 Insertion Sort

As the name suggests, insertion sort method sorts a list of elements by inserting each successive element in the previously sorted sublist. Such insertion of elements requires the other elements to be shuffled as required.

To understand the insertion sorting method, consider a scenario where an array A containing n elements needs to be sorted. Now, each pass of the insertion sorting method will insert the element  $A[i]$  into its appropriate position in the previously sorted subarray, i.e.,  $A[1], A[2], \dots, A[i-1]$ . The following list describes the tasks performed in each of the passes:

**Pass 1**  $A[2]$  is compared with  $A[1]$  and inserted either before or after  $A[1]$ . This makes  $A[1], A[2]$  a sorted sub array.

**Pass 2**  $A[3]$  is compared with both  $A[1]$  and  $A[2]$  and inserted at an appropriate place. This makes  $A[1], A[2], A[3]$  a sorted sub array.

**Pass  $n-1$**   $A[n]$  is compared with each element in the sub array  $A[1], A[2], A[3], \dots, A[n-1]$  and inserted at an appropriate position. This eventually makes the entire array A sorted.

Let us revisit the list L containing five integers stored in a random fashion, as shown in Fig. 8.1.

Now, if the list L is sorted using insertion sort technique then first of all the second element in the list, i.e., 3 will be selected and compared with the first element, i.e., 18. Since 3 is less than 18, the two elements will be interchanged. This process is repeated until all the elements are rearranged in a sorted manner. Table 8.2 illustrates the sorting of list L in ascending order using insertion sort technique.



#### Check Point

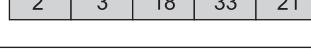
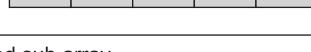
1. How many passes are required by the selection sort technique to sort an array of N elements?

**Ans.**  $N-1$

2. What is the most significant disadvantage of selection sort?

**Ans.** One of the most critical disadvantages of selection sort is that its efficiency of  $O(n^2)$  does not make it suitable for large sized lists.

**Table 8.2** Insertion sort

Pass	Comparison	Resultant Array
1		
2		
3		
4		

 denotes the previously sorted sub array  
 denotes the current selection

As we can see in the above illustration, four passes were required to sort a list of five elements. Hence, we can say that insertion sort requires  $n-1$  passes to sort an array of  $n$  elements.

**Example 8.4** Write an algorithm to perform insertion sort on a given array of integers.

```

insertion(arr[], size)
Step 1: Start
Step 2: Set i = 1, j = 0 and temp = 0
Step 3: Repeat Steps 4-12 while i < size
Step 4: Set temp = arr[i]
Step 5: Set j = i-1
Step 6: Repeat Steps 7-10 while j>=0
Step 7: if arr[j] > temp goto Step 8 else goto Step 9
Step 8: Set arr[j+1] = arr[j]
Step 9: Branch out and go to Step 11
Step 10: Set j = j-1
Step 11: Set arr[j+1] = temp
Step 12: Set i = i + 1
Step 13: Stop
  
```

**Example 8.5** Write a C program to perform insertion sort on an array of N elements.

Program 8.2 implements insertion sorting technique in C. It uses the algorithm depicted in Example 8.4.

**Program 8.2** *Insertion sort*

```

#include <stdio.h>
#include <conio.h>
#include <stdlib.h>

void insertion(int [], int); /*Function prototype for performing insertion
sort*/
  
```

```

void main()
{
    int *arr;
    int i, N;
    clrscr();

    printf("Enter the number of elements in the array:\n");
    scanf("%d", &N);

    arr = (int*) malloc(sizeof(int)); /*Dynamic allocation of memory for the
array*/
}

printf("Enter the %d elements to sort:\n", N);
for (i=0;i<N;i++)
scanf("%d", &arr[i]); /*Reading array elements*/

insertion(arr,N); /*Calling insertion function*/

printf("\nThe sorted elements are:\n");
for(i=0;i<N;i++)
printf("%d\n", arr[i]); /*Printing sorted array*/

getch();
}

void insertion(int array[], int size)
{
    int i,j,temp;
    for(i=1;i<size;i++)
    {
        temp=array[i]; /*Selecting the next element to be inserted*/
        /*Inserting the element in previously sorted sub array*/
        for(j=i-1;j>=0;j--)
        if(array[j]>temp)
            array[j+1]=array[j];
        else
            break;
        array[j+1]=temp;
    }
}

```

*The break statement takes the control out of the looping construct as soon as the point of insertion is ascertained in the sorted sub array.*

## Output

```

Enter the number of elements in the array:
5
Enter the 5 elements to sort:
18
3
2
33

```

The sorted elements are:  
 2  
 3  
 18  
 21  
 33

### Program analysis

Key Statement	Purpose
<b>temp=array[i];</b>	Stores the next element to be inserted, in the <i>temp</i> variable
<b>for ( j=i-1;j&gt;=0;j—) if(array[j]&gt;temp) array[ j+1]=array[ j]; else break;</b>	Identifies the point of insertion for the element in the previously sorted sub array
<b>array [j+1]=temp;</b>	Inserts the element at the identified location

**Efficiency of Insertion Sort** Assume that an array containing  $n$  elements is sorted using insertion sort technique.

The minimum number of elements that must be scanned =  $n-1$

For each of the elements the maximum number of shifts possible =  $n-1$

Thus, efficiency of insertion sort =  $O(n^2)$

**Advantages and Disadvantages** Some of the key advantages of insertion sorting technique are:

1. It is one of the simplest sorting techniques that is easy to implement.
2. It performs well in case of smaller lists.
3. It leverages the presence of any existing sort pattern in the list, thus resulting in better efficiency.

The disadvantages associated with insertion sorting technique are as follows.

1. The efficiency of  $O(n^2)$  is not well suited for large sized lists.
2. It requires large number of elements to be shifted.



**Tip** *Insertion sorting technique should not be used with lists containing lengthy records as the worst case of  $O(n^2)$  may result in inefficient performance.*

### 8.2.3 Bubble Sort

Bubble sort is one of the oldest and simplest of sorting techniques. It focuses on bringing the largest element to the end of the list with each successive pass. Unlike selection sort, it does not perform a search to identify the largest element; instead it repeatedly compares two consecutive elements and moves the largest amongst them to the right. This process is repeated for all pairs of elements until the current iteration moves the largest element to the end of the list.

To understand the bubble sorting method, consider a scenario where an array A containing n elements needs to be sorted. In the first pass, elements A[1] and A[2] are compared and if A[1] is larger than A[2] then the two values are swapped. Next, A[2] and A[3] are compared. The last comparison of the first pass between A[n-1] and A[n] brings the largest element of the list to the end. The second pass repeats this process for the remaining n-1 elements. Finally, the last pass compares only the first two elements i.e., A[1] and A[2] to generate the sorted list.



### Check Point

**1. What is the efficiency of insertion sort?**

**Ans.** The efficiency of insertion sort is  $O(n^2)$ .

**2. What is the advantage of using insertion sort?**

**Ans.** The advantage of using insertion sort technique is that it is easy to implement and it performs well for small sized lists.

Let us revisit the list L containing five integers stored in a random fashion, as shown in Fig. 8.1.

Table 8.3 illustrates the sorting of list L in ascending order using bubble sort:

**Table 8.3 Bubble sort**

Pass	Comparison	Resultant Array																																													
1	<table border="1"> <tr><td>18</td><td>3</td><td>2</td><td>33</td><td>21</td></tr> <tr><td colspan="2" style="text-align: center;">↔</td><td></td><td></td><td></td></tr> </table> <table border="1"> <tr><td>3</td><td>18</td><td>2</td><td>33</td><td>21</td></tr> <tr><td colspan="2" style="text-align: center;">↔</td><td></td><td></td><td></td></tr> </table> <table border="1"> <tr><td>3</td><td>2</td><td>18</td><td>33</td><td>21</td></tr> <tr><td colspan="3" style="text-align: center;">↔</td><td></td><td></td></tr> </table> <table border="1"> <tr><td>3</td><td>2</td><td>18</td><td>33</td><td>21</td></tr> <tr><td colspan="4" style="text-align: center;">↔</td><td></td></tr> </table>	18	3	2	33	21	↔					3	18	2	33	21	↔					3	2	18	33	21	↔					3	2	18	33	21	↔					<table border="1"> <tr><td>3</td><td>2</td><td>18</td><td>21</td><td>33</td></tr> </table>	3	2	18	21	33
18	3	2	33	21																																											
↔																																															
3	18	2	33	21																																											
↔																																															
3	2	18	33	21																																											
↔																																															
3	2	18	33	21																																											
↔																																															
3	2	18	21	33																																											
2	<table border="1"> <tr><td>3</td><td>2</td><td>18</td><td>21</td><td>33</td></tr> <tr><td colspan="2" style="text-align: center;">↔</td><td></td><td></td><td></td></tr> </table> <table border="1"> <tr><td>2</td><td>3</td><td>18</td><td>21</td><td>33</td></tr> <tr><td colspan="2" style="text-align: center;">↔</td><td></td><td></td><td></td></tr> </table> <table border="1"> <tr><td>2</td><td>3</td><td>18</td><td>21</td><td>33</td></tr> <tr><td colspan="3" style="text-align: center;">↔</td><td></td><td></td></tr> </table>	3	2	18	21	33	↔					2	3	18	21	33	↔					2	3	18	21	33	↔					<table border="1"> <tr><td>2</td><td>3</td><td>18</td><td>21</td><td>33</td></tr> </table>	2	3	18	21	33										
3	2	18	21	33																																											
↔																																															
2	3	18	21	33																																											
↔																																															
2	3	18	21	33																																											
↔																																															
2	3	18	21	33																																											
3	<table border="1"> <tr><td>2</td><td>3</td><td>18</td><td>21</td><td>33</td></tr> <tr><td colspan="2" style="text-align: center;">↔</td><td></td><td></td><td></td></tr> </table> <table border="1"> <tr><td>2</td><td>3</td><td>18</td><td>21</td><td>33</td></tr> <tr><td colspan="3" style="text-align: center;">↔</td><td></td><td></td></tr> </table>	2	3	18	21	33	↔					2	3	18	21	33	↔					<table border="1"> <tr><td>2</td><td>3</td><td>18</td><td>21</td><td>33</td></tr> </table>	2	3	18	21	33																				
2	3	18	21	33																																											
↔																																															
2	3	18	21	33																																											
↔																																															
2	3	18	21	33																																											
4	<table border="1"> <tr><td>2</td><td>3</td><td>18</td><td>21</td><td>33</td></tr> <tr><td colspan="2" style="text-align: center;">↔</td><td></td><td></td><td></td></tr> </table>	2	3	18	21	33	↔					<table border="1"> <tr><td>2</td><td>3</td><td>18</td><td>21</td><td>33</td></tr> </table>	2	3	18	21	33																														
2	3	18	21	33																																											
↔																																															
2	3	18	21	33																																											

↔ → denotes the pair of consecutive elements being compared

As we can see in the above illustration, four passes were required to sort a list of five elements. Hence, we can say that bubble sort requires  $n-1$  passes to sort an array of  $n$  elements.

**Example 8.6** Write an algorithm to perform bubble sort on a given array of integers.

```

bubble(arr[], size)
Step 1: Start
Step 2: Set i = size, j = 0 and temp = 0
Step 3: Repeat Steps 4-9 while i > 1
Step 4: Set j = 0
Step 5: Repeat Steps 6-8 while j < i-1
Step 6: if arr[j] > arr[j+1] goto Step 7 else goto Step 8
Step 7: Swap the elements stored at arr[j] and arr[j+1] by performing the
following steps
    I Set temp = a[j+1]
    II Set arr[j+1] = arr[j]
    III Set a[j]=temp
Step 8: Set j = j + 1
Step 9: Set i = i - 1
Step 10: Stop

```

**Example 8.7** Write a C program to perform bubble sort on an array of N elements.

Program 8.3 implements bubble sorting technique in C. It uses the algorithm depicted in Example 8.6.

#### **Program 8.3** *Implementation of bubble sorting technique*

```

#include <stdio.h>
#include <conio.h>
#include <stdlib.h>

void bubble(int [], int); /*Function prototype for performing bubble sort*/

void main()
{
    int *arr;
    int i, N;
    clrscr();

    printf("Enter the number of elements in the array:\n");
    scanf("%d",&N);

    arr = (int*) malloc(sizeof(int)); /*Dynamic allocation of memory for the
array*/
}

printf("Enter the %d elements to sort:\n",N);
for (i=0;i<N;i++)
scanf("%d",&arr[i]); /*Reading array elements*/

bubble(arr,N); /*Calling bubble function*/

printf("\nThe sorted elements are:\n");

```

```

for(i=0;i<N;i++)
printf("%d\n",arr[i]); /*Printing sorted array*/

getch();
}

void bubble(int array[], int size)
{
    int i, j, temp;
    for(i=size;i>1;i--)
        for(j=0;j<i-1;j++)
            if (array[j]>array[j+1])
            {
                /*Swapping adjacent elements*/
                temp = array[j+1];
                array[j+1] = array[j];
                array[j] = temp;
            }
}

```

The outer loop controls the number of passes while the inner loop controls the number of comparisons made in each pass.

## Output

Enter the number of elements in the array:

5

Enter the 5 elements to sort:

18

3

2

33

21

The sorted elements are:

2

3

18

21

33

## Program analysis

Key Statement	Purpose
<b>if(array[j]&gt;array[j+1])</b>	Compares the adjacent array elements in each pass
<b>temp = array[j+1]; array[j+1] = array[j]; array[j] = temp;</b>	Swaps the array elements as per the sort order

**Efficiency of Bubble Sort** Assume that an array containing  $n$  elements is sorted using bubble sort technique.

Number of comparisons made in first pass =  $n-1$

Number of comparisons made in second pass =  $n-2$

Number of comparisons made in last pass = 1

$$\begin{aligned}\text{Total number of comparisons made} &= (n-1) + (n-2) + \dots + 1 \\ &= n * (n - 1) / 2 \\ &= O(n^2)\end{aligned}$$

Thus, efficiency of bubble sort =  $O(n^2)$

**Advantages and Disadvantages** Some of the key advantages of bubble sorting technique are:

1. It is easy to understand and implement.
2. It leverages the presence of any existing sort pattern in the list, thus resulting in better efficiency.

The disadvantages associated with bubble sorting technique are given below.

1. The efficiency of  $O(n^2)$  is not well suited for large sized lists.
2. It requires large number of elements to be shifted.
3. It is slow in execution as large elements are moved towards the end of the list in a step-by-step fashion.



**Note** *Bubble sort leverages any existing sort pattern in a list quite well. Its best case efficiency on an already sorted list is  $O(n)$  which is better than a number of other sorting techniques.*

## 8.2.4 Quick Sort

As the name suggests, quick sort is one of the fastest sorting methods that is based on divide and conquer strategy. It divides the given list into a number of sub lists and then works on each of the sub lists to obtain the sorted output. It first chooses one of the list elements as a key value and then tries to place the key value at its final position in the list. Once, the key value is positioned correctly, the two sub lists to the left and right of the key value are processed in the similar fashion until the entire list becomes sorted.



**Note**

*The key value is also called as pivot element.*

Consider an array containing six elements, as shown below.

34      99      5      2      57      40

Initially, the first list element i.e., 34 is chosen as the pivot element. Now, the list is scanned from right to left to identify the first element that is less than 34. This element is 2. So, both the elements are swapped and the list becomes:



### Check Point

**1. Why is the bubble sorting technique slow in execution?**

**Ans.** Bubble sorting technique is considered as slow because it moves the elements to the end of the list in a step-by-step fashion.

**2. How many passes are required by bubble sort to sort an array of N elements?**

**Ans.**  $N-1$

2        99        5        34        57        40

Now, the list is scanned from left to right till the place where 34 is stored and the control stops at the first element that is greater than 34. This element is 99. So, both the elements are swapped and the list becomes:

2        34        5        99        57        40

Now, the list is again scanned from right to left starting with element 99 and ending at element 34. Element 5 is found to be lesser than 34, thus both the elements are swapped. Now, the list becomes:

2        5        34        99        57        40

Now, there are no elements present between 5 and 34, thus we can assume that 34 has attained its final position in the list.

Now, the two sublists to the left and right of the pivot element are identified, as shown below:

2        5        (34)        99        57        40

Now, each of these lists is processed in the same fashion and eventually all the elements are placed at appropriate positions in the final sorted list.

**Example 8.8** Write an algorithm to perform quick sort on a given array of integers.

```
quick(arr[], LB, UB)
Step 1: Start
Step 2: Set pivot=0, nxt_pvt=0, left=LB, right=UB
Step 3: Set pivot = arr[left] to select the first element as the pivot element
Step 4: Repeat Steps 5-14 while LB < UB
Step 5: Repeat Step 6 while arr[UB] >= pivot and LB < UB
Step 6: Set UB = UB - 1
Step 7: if LB is not equal to UB goto Step 8 else goto Step 10
Step 8: Set arr[LB]=arr[UB]
Step 9: Set LB = LB + 1
Step 10: Repeat Step 11 while arr[LB] <= pivot and LB < UB
Step 11: Set LB = LB + 1
Step 12: if LB is not equal to UB goto Step 13 else goto Step 15
Step 13: Set arr[UB]=arr[LB]
Step 14: Set UB = UB - 1
Step 15: Set arr[LB]=pivot
Step 16: Set nxt_pvt = LB
Step 17: Set LB = left and UB = right
Step 18: if LB < nxt_pvt goto Step 19 else goto Step 20
Step 19: Apply quick sort in the left sub list by calling module quick(arr,
LB, nxt_pvt-1)
Step 20: if UB > nxt_pvt goto Step 21 else goto Step 22
Step 21: Apply quick sort in the right sub list by calling module quick(arr,
nxt_pvt+1, UB)
Step 22: Stop
```

**Example 8.9** Write a C program to perform quick sort on an array of N elements.

Program 8.4 implements quick sorting technique in C. It uses the algorithm depicted in Example 8.8.

#### **Program 8.4** Implementation of quick sorting technique

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>

void quick(int [], int, int); /*Function prototype for performing quick
sort*/

void main()
{
    int *arr;
    int i, N;
    clrscr();

    printf("Enter the number of elements in the array:\n");
    scanf("%d",&N);

    arr = (int*) malloc(sizeof(int)); /*Dynamic allocation of memory for the
array*/

    printf("Enter the %d elements to sort:\n",N);
    for (i=0;i<N;i++)
        scanf("%d",&arr[i]); /*Reading array elements*/

    quick(arr,0,N-1); /*Calling quick function*/

    printf("\nThe sorted elements are:\n");
    for(i=0;i<N;i++)
        printf("%d\n",arr[i]); /*Printing sorted array*/

    getch();
}

void quick(int array[], int LB, int UB)
{
    int pivot, nxt_pvt, left, right;
    left = LB;
    right = UB;
pivot = array[left];
    while(LB<UB)
    {
        /*Scanning the list from right to left to identify the element lesser
than pivot element*/
        while((array[UB] >= pivot) && (LB<UB))
            UB--;

        if(LB!=UB)
        {
```

```

array[LB]=array[UB]; /*Shuffling pivot element*/
LB++;
}

/*Scanning the list from left to right to identify the element greater
than pivot element*/
while((array[LB]<=pivot) && (LB<UB))
LB++;
if(LB != UB)
{
array[UB] = array[LB]; /*Shuffling pivot element*/
UB--;
}
}

array[LB]=pivot;
nxt_pvt=LB;
LB=left;
UB=right;

if(LB<nxt_pvt)
quick(array, LB, nxt_pvt-1);
if(UB>nxt_pvt)
quick(array, nxt_pvt+1, UB);
}

```

*The quick sort module is called recursively until the entire list is sorted.*

## Output

```

Enter the number of elements in the array:
6
Enter the 6 elements to sort:
34
99
5
2
57
40

```

The sorted elements are:

```

2
5
34
40
57
99

```

## Program analysis

Key Statement	Purpose
<b>pivot = array[left];</b>	Initializes the pivot element
<b>nxt_pvt=LB;</b> <b>LB=left;</b> <b>UB=right;</b>	Generates the next set of pivot, lower bound and upper bound values

Key Statement	Purpose
<pre>if(LB&lt;nxt_pvt)     quick(array, LB, nxt_pvt-1); if(UB&gt;nxt_pvt)     quick(array, nxt_pvt+1, UB);</pre>	Recursively calls the <i>quick()</i> function as per the next set of pivot, <i>UB</i> and <i>LB</i> values

**Efficiency of Quick Sort** Assume that an array containing  $n$  elements is to be sorted using quick sort technique. Let us analyze the efficiency of quick sort in best case and worst case scenarios.

**Best Case** In the best case, the pivot element always divides the list in to two equal halves. Here, we are assuming that the number of elements in the list is a power of 2. That means,  $n = 2^m$  or  $m = \log_2 n$

Number of comparisons made in first pass =  $n$

Number of comparisons made in second pass =  $2*(n / 2)$

Number of comparisons made in the third pass =  $4*(n / 4)$

Number of comparisons made in the fourth pass =  $8*(n / 8)$

Number of comparisons made in the  $k^{\text{th}}$  pass =  $k*(n / k)$

$$\begin{aligned} \text{Now, total number of comparisons} &= O(n) + O(n) + O(n) + \dots + m \text{ terms} \\ &= O(n * m) \\ &= O(n \log n) \end{aligned}$$

Thus, efficiency of quick sort in best case scenario =  $O(n \log n)$

**Worst Case** It may happen that the pivot element divides the lists in unequal partitions. In the worst case, there would be no element in one of the lists while the other list will contain all the elements.

In such a case, number of comparisons made in first pass =  $n-1$

Number of comparisons made in second pass =  $n-2$

Number of comparisons made in last pass = 1

$$\begin{aligned} \text{Total number of comparisons} &= (n-1) + (n-2) + \dots + 1 \\ &= n * (n-1) / 2 \\ &= O(n^2) \end{aligned}$$

Thus, efficiency of quick sort in worst case scenario =  $O(n^2)$

**Advantages and Disadvantages** Some of the key advantages of quick sorting technique are:

1. It is one of the fastest sorting algorithms.
2. Its implementation does not require any additional memory.

The disadvantages associated with quick sorting technique are as follows.

1. The worst case efficiency of  $O(n^2)$  is not well suited for large sized lists.
2. Its algorithm is considered as a little more complex in comparison to some other sorting techniques.



### Note

*The choice of the pivot element may have a direct impact on the performance of the quick sort algorithm, considering that there could be some pre-existing sort order present in the input list. As a result, different implementations of the quick sorting technique use first, last, middle or at times some randomly chosen element as the pivot element.*

## 8.2.5 Merge Sort

Merge sort is another sorting technique that is based on divide-and-conquer approach. It divides a list into several sub lists of equal sizes and sorts them individually. It then merges the various sub lists in pairs to eventually form the original list, while ensuring that the sort order is not disturbed.

Consider a list L containing n elements on which merge sort is to be performed. Initially, the n elements of the list L are considered as n different sublists of one element each. Since, a list having one element is sorted in itself, thus there is no further action required on these sublists. Now, each of the sublists is merged in pairs to form  $n/2$  sublists having two elements each. While merging two lists the elements are compared and placed in a sorted fashion in the new sublist. This process is repeated until the original list is formed with elements arranged in a sorted fashion.

Consider an array containing six elements, as shown below:

34    99    5    2    57    40    8    29

Figure 8.2 shows how merge sort is performed on the above list.

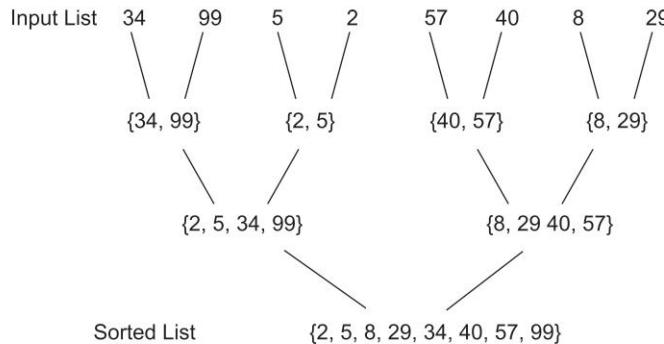


Fig. 8.2 Merge sort

As we can see in the above illustration, the sorted sublists are progressively merged in each pass to eventually generate the original list, sorted in ascending order.

**Example 8.10** Write an algorithm to perform merge sort on a given array of integers.

```
mergesort(arr[], size)
Step 1: Start
Step 2: Set mid = 0
Step 3: if size = 1 goto Step 4 else goto Step 5
Step 4: Stop and return back to the calling module
Step 5: Set mid = size / 2
Step 6: Call module mergesort(arr, mid)
```



### Check Point

1. What is the best case efficiency of quick sort?

Ans.  $O(n\log n)$ .

2. What is a pivot element?

Ans. It is a key value that is selected and shuffled continuously as per the quick sort algorithm until it attains its final position in the list.

```
Step 7: Call module mergesort(arr+mid, size-mid)
Step 8: Call module merge(arr, mid, arr+mid, size-mid)
Step 9: Stop
```

```
merge(a[], size1, b[], size2)
Step 1: Start
Step 2: Initialize a temporary array, temp_array[size1+size2]
Step 3: Set i=0, j=0, k=0
Step 4: Repeat Step 5-9 while i < size1 and j < size2
Step 5: If a[i] < b[j] goto Step 6 else goto Step 8
Step 6: Set temp_array[k] = a[i]
Step 7: Set k = k + 1 and i = i + 1
Step 8: Set temp_array[k] = b[j]
Step 9: Set k = k + 1 and j = j + 1
Step 10: Repeat Step 11-12 while i < size1
Step 11: Set temp_array[k] = a[i]
Step 12: Set k = k + 1 and i = i + 1
Step 13: Repeat Step 14-15 while j < size2
Step 14: Set temp_array[k] = b[j]
Step 15: Set k = k + 1 and j = j + 1
Step 16: Set a[] = temp_array[]
Step 17: Stop
```

**Example 8.11** Write a C program to perform merge sort on an array of N elements.

Program 8.5 implements merge sorting technique in C. It uses the algorithm depicted in Example 8.10.

**Program 8.5** *Implementation of merge sort technique in C*

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>

void mergesort(int*, int); /*Function prototype for performing merge sort*/
void merge(int*, int, int*, int); /*Function prototype for merging two
arrays*/
void main()
{
    int *arr;
    int i, N;
    clrscr();

    printf("Enter the number of elements in the array:\n");
    scanf("%d", &N);

    arr = (int*) malloc(sizeof(int)*N); /*Dynamic allocation of memory for
the array*/

    printf("Enter the %d elements to sort:\n", N);
    for (i=0;i<N;i++)
```

```

scanf("%d",&arr[i]); /*Reading array elements*/

mergesort(arr,N); /*Calling mergesort function*/

printf("\nThe sorted elements are:\n");
for(i=0;i<N;i++)
printf("%d\n",arr[i]); /*Printing sorted array*/

getch();
}

void mergesort(int *array, int size)
{
    int mid;
    if(size==1)
        return;
    else
    {
        mid = size/2;
        /*Making recursive calls to mergesort function*/
        mergesort(array, mid);
        mergesort(array+mid, size-mid);
        merge(array, mid, array+mid, size-mid); /*Calling merge function*/
    }
}

void merge(int *a, int s1, int *b, int s2)
{
    int i, j, k, *temp_arr;
    temp_arr=(int*) malloc((s2+s1) * sizeof(int)); /*Dynamic allocation of a
temporary array in memory*/
    i=j=k=0;
    while(i < s1 && j < s2)
        temp_arr[k++] = (a[i]<b[j]) ? a[i++] : b[j++];

    while(i < s1)
        temp_arr[k++] = a[i++];
    while(j < s2)
        temp_arr[k++] = b[j++];

    for(i=0;i<k;i++)
        a[i] = temp_arr[i];

    free(temp_arr);
}

```

*Here, the use of increment and conditional operators has simplified the code and reduced it to a single line.*

## Output

```
Enter the number of elements in the array:
```

```
8
```

```
Enter the 8 elements to sort:
```

```
34
```

```
99
```

```
5
```

```
2
```

```
57
```

```
40
```

```
8
```

```
29
```

```
The sorted elements are:
```

```
2
```

```
5
```

```
8
```

```
29
```

```
34
```

```
40
```

```
57
```

```
99
```

## Program analysis

Key Statement	Purpose
<code>mid = size/2;</code>	Initializes the <i>mid</i> variable
<code>mergesort(array, mid);</code> <code>mergesort(array+mid, size-mid);</code>	Recursively calls the <i>mergesort()</i> function to sort the individual sub arrays
<code>merge(array, mid, array+mid, size-mid);</code>	Calls the <i>merge</i> function to merge two sorted sub arrays

**Efficiency of Merge Sort** Assume that an array containing  $n$  elements is sorted using merge sort technique. As we have already seen, merge sort is divided into two submodules. One module keeps on dividing the list until  $n$  different sublists of one element each are generated. Alternatively, the other module keeps on appending pairs of sublists until the main list with  $n$  elements is generated.

Now, the number of steps required for dividing the list =  $\log_2 n$

The number of steps required for merging the lists =  $\log_2 n$

Total number of steps =  $2\log_2 n$

Now, in each step all  $n$  list elements are compared.

Thus, total number of comparisons made =  $n * 2\log_2 n$

$$= 2n\log_2 n$$

$$= O(n\log_2 n)$$

Thus, efficiency of merge sort =  $O(n\log_2 n)$

**Advantages and Disadvantages** Some of the key advantages of merge sorting technique are:

1. It is a fast and stable sorting method.

2. It always ensures an efficiency of  $O(n\log n)$ .  
The disadvantages associated with merge sorting technique are as follows.

1. It requires additional memory space to perform sorting. The size of the additional space is in direct proportion to the size of the input list.
2. Even though the number of comparisons made by merge sort are nearly optimal, its performance is slightly lesser than that of quick sort.

### 8.2.6 Bucket Sort

Bucket sort distributes the list of elements across different buckets in such a way that any bucket  $m$  contains elements greater than the elements of bucket  $m-1$  but less than the elements of bucket  $m+1$ . The elements within each bucket are sorted individually either by using some alternate sorting technique or by recursively applying bucket sort technique. In the end, elements of all the buckets are merged to generate the sorted list. This technique is particularly effective for smaller range of data series.

Consider a list containing ten integers stored in a random fashion, as shown in Fig. 8.3.

2	27	13	18	21	43	42	39	31	4
---	----	----	----	----	----	----	----	----	---

Fig. 8.3 List of integers

In the above list, all elements are between the range of 0 to 50. So, let us create five buckets for storing ten elements each. Figure 8.4 shows how these buckets are used for sorting the list.

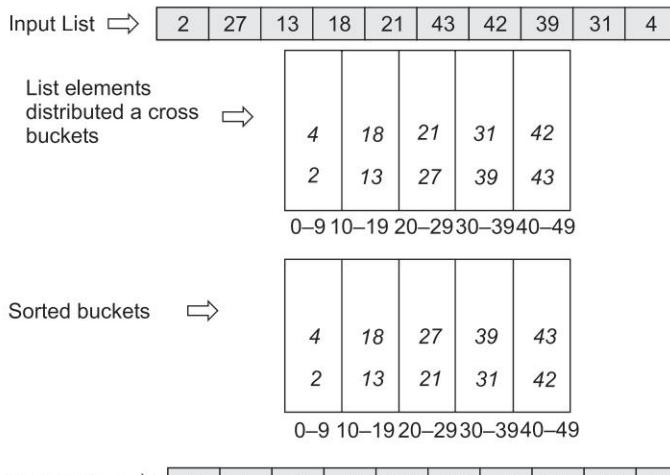


Fig. 8.4 Bucket sort



### Check Point

1. What is the efficiency of merge sort?

Ans.  $O(n\log n)$ .

2. What is the most significant advantage of merge sort?

Ans. It always ensures an efficiency of  $O(n\log n)$ .



### Mind Jog

#### What is a stable sort?

*It is that sorting methodology which preserves the original order of the duplicate values in the final sorted list.*

As we can see in the above illustration, the list elements are first distributed as per their values across different buckets. Then, each of the buckets are individually sorted and later merged to generate the original sorted list.

**Example 8.12** Write an algorithm to perform bucket sort on a given array of integers.

**Assumption:** The input list elements are within the range of 0 to 49.

**bucket(arr[], size)**

Step 1: Start

Step 2: Set  $i = 0$ ,  $j = 0$  and  $k = 0$

Step 3: Initialize an array  $c[5]$  and set all its values to 0; it keeps track of the number of elements in each of the five buckets

Step 4: Create five buckets by initializing a 2-D array  $b[5][10]$  and set all its values to 0

Step 5: Now, distribute the input list elements across different buckets.  
To do this, repeat Steps 6-16 while  $i < \text{size}$

Step 6: if  $0 \leq \text{arr}[i] \leq 9$  then goto Step 7 else goto Step 8

Step 7: Set  $b[0][c[0]] = \text{arr}[i]$  and  $c[0] = c[0] + 1$

Step 8: if  $10 \leq \text{arr}[i] \leq 19$  then goto Step 9 else goto Step 10

Step 9: Set  $b[1][c[1]] = \text{arr}[i]$  and  $c[1] = c[1] + 1$

Step 10: if  $20 \leq \text{arr}[i] \leq 29$  then goto Step 11 else goto Step 12

Step 11: Set  $b[2][c[2]] = \text{arr}[i]$  and  $c[2] = c[2] + 1$

Step 12: if  $30 \leq \text{arr}[i] \leq 39$  then goto Step 13 else goto Step 14

Step 13: Set  $b[3][c[3]] = \text{arr}[i]$  and  $c[3] = c[3] + 1$

Step 14: if  $40 \leq \text{arr}[i] \leq 49$  then goto Step 15 else goto Step 16

Step 15: Set  $b[4][c[4]] = \text{arr}[i]$  and  $c[4] = c[4] + 1$

Step 16: Set  $i = i + 1$

Step 17: Sort each of the buckets  $b[][]$  by calling insertion sort module  
`insertion(&b[],[],c[])`

Step 18: Merge all the buckets together into the main array by setting  
`array [] = b[][]`

Step 19: Stop

**Example 8.13** Write a C program to perform bucket sort on an array of N elements.

Program 8.6 implements bucket sorting technique in C. It uses the algorithms depicted in Example 8.4 and Example 8.12.

**Program 8.6** *Implementation of bucket sorting technique*

```
#include <stdio.h>
#include <conio.h>
```

```

#include <stdlib.h>

void insertion(int*, int); /*Function prototype for performing insertion
sort*/
void bucket(int*, int); /*Function prototype for performing bucket sort*/

void main()
{
    int *arr;
    int i, N;
    clrscr();

    printf("Enter the number of elements in the array:\n");
    scanf("%d", &N);

    arr = (int*) malloc(sizeof(int)); /*Dynamic allocation of memory for the
array*/

    printf("Enter the %d elements to sort:\n", N);
    for (i=0;i<N;i++)
        scanf("%d", &arr[i]); /*Reading array elements*/

    bucket(arr,N); /*Calling bucket function*/

    printf("\nThe sorted elements are:\n");
    for(i=0;i<N;i++)
        printf("%d\n",arr[i]); /*Printing sorted array*/

    getch();
}

/*Insertion sort function for sorting elements in a bucket*/
void insertion(int *array, int size)
{
    int i=0,j=0,temp=0;
    for(i=1;i<size;i++)
    {
        temp=array[i];
        for(j=i-1;j>=0;j--)
        if(array[j]>temp)
            array[j+1]=array[j];
        else
            break;
        array[j+1]=temp;
    }
}

void bucket(int *array, int size)
{

```

```

int i, j, k, b[5][10];
int c[5];

for(i=0;i<5;i++)
c[i]=0;

/*Distributing elements across different buckets*/
for(i=0;i<size;i++)
{
if(array[i]>=0 && array[i]<=9)
b[0][c[0]++]=array[i];

if(array[i]>=10 && array[i]<=19)
b[1][c[1]++]=array[i];

if(array[i]>=20 && array[i]<=29)
b[2][c[2]++]=array[i];

if(array[i]>=30 && array[i]<=39)
b[3][c[3]++]=array[i];

if(array[i]>=40 && array[i]<=49)
b[4][c[4]++]=array[i];
}

/*Sorting elements in each bucket using insertion sort*/
for(i=0;i<5;i++)
if(c[i]!=0)
insertion(&b[i][0], c[i]); /*Calling insert function*/

/*Merging buckets to form the original list*/
i=0;
k=0;
while(i<5)
{
if(c[i]==0)
{
i=i+1;
continue;
}

for(j=0;j<c[i];j++)
array[k++]=b[i][j];
i=i+1;
}
}

```

*Here, elements in each of the bucket are sorted using insertion sort technique.*

## Output

```

Enter the number of elements in the array:
10

```

```
Enter the 10 elements to sort:
```

```
2  
27  
13  
18  
21  
43  
42  
39  
31  
4
```

```
The sorted elements are:
```

```
2  
4  
13  
18  
21  
27  
31  
39  
42  
43
```

### Program analysis

Key Statement	Purpose
<pre>for(i=0;i&lt;5;i++) if(c[i]!=0) insertion(&amp;b[i][0], c[i]);</pre>	Repeatedly calls the <i>insertion()</i> function to perform insertion sort on each of the buckets
<pre>for(j=0;j&lt;c[i];j++) array[k++]=b[i][j];</pre>	Forms the original array from the bucket elements



#### Note

*The bucket sort algorithm is particularly suited for lists having elements within a specific range. The above program is based on the assumption that the elements in the input list are within the range of 0 to 49.*

**Efficiency of Bucket Sorting** Assume that an array containing  $n$  elements is sorted using bucket sort technique.

In the worst case, all elements of the list will be placed in a single bucket.

Now, each bucket is sorted using insertion sort, whose efficiency =  $O(n^2)$

Thus, worst case efficiency of bucket sort =  $O(n^2)$

**Advantages and Disadvantages** Some of the key advantages of bucket sorting technique are:

1. It preserves the order of repetitive values in the list.
2. It performs well for large size lists having elements in a smaller range.

The disadvantages of bucket sort are as follows:

1. It works only if the range of input values is fixed.
2. It requires additional space to perform the sorting operation.

## 8.3 SEARCHING TECHNIQUES

Searching refers to determining whether an element is present in a given list of elements or not. If the element is found to be present in the list then the search is considered as successful, otherwise it is considered as an unsuccessful search. The search operation returns the location or address of the element found.

There are various searching methods that can be employed to perform search on a data set. The choice of a particular searching method in a given situation depends on a number of factors, such as

1. order of elements in the list, i.e., random or sorted
2. size of the list

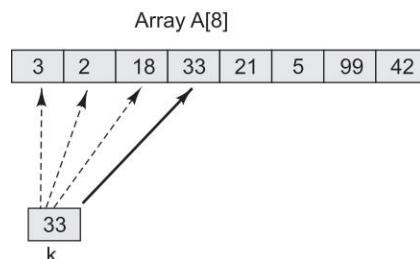
Let us explore the various searching methods one by one.

### 8.3.1 Linear Search

It is one of the conventional searching techniques that sequentially searches for an element in the list. It typically starts with the first element in the list and moves towards the end in a step-by-step fashion. In each iteration, it compares the element to be searched with the list element, and if there is a match, the location of the list element is returned.

Consider an array of integers A containing  $n$  elements. Let  $k$  be the value that needs to be searched. The linear search technique will first compare  $A[0]$  with  $k$  to find out if they are same. If the two values are found to be same then the index value, i.e., 0 will be returned as the location containing  $k$ . However, if the two values are not same then  $k$  will be compared with  $A[1]$ . This process will be repeated until the element is not found. If the last comparison between  $k$  and  $A[n-1]$  is also negative then the search will be considered as unsuccessful.

Figure 8.5 depicts the linear search technique performed on an array of integers.



**Fig. 8.5** Linear search

As shown in Fig. 8.5, the value  $k$  is repeatedly compared with each element of the array A. As soon as the element is found, the corresponding index location is returned and the search operation is terminated.



### Check Point

1. What is the worst case efficiency of bucket sort?

Ans.  $O(n^2)$ .

2. What is the distinctive advantage of using bucket sort technique?

Ans. It preserves the order of duplicate elements in the final sorted list.

**Example 8.14** Write an algorithm to perform linear search on a given array of integers.

```
linear(arr[], size, k)
Step 1: Start
Step 2: Set i = 0
Step 3: Repeat Steps 4-6 while i < size
Step 4: if k = arr[i] goto Step 5 else goto Step 6
Step 5: Return i and goto Step 9
Step 6: Set i = i + 1
Step 7: If i = size goto Step 8 else goto Step 9
Step 8: Return NULL and goto Step 9
Step 9: Stop
```

**Example 8.15** Write a C program to perform linear search on an array of N elements.

Program 8.7 implements linear search technique in C. It uses the algorithm depicted in Example 8.14.

**Program 8.7** *Implementation of linear search technique*

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>

int linear(int [], int, int); /*Function prototype for performing linear
search*/

void main()
{
    int *arr;
    int i, N, k, index;
    clrscr();

    printf("Enter the number of elements in the array arr:\n");
    scanf("%d",&N);

    arr = (int*) malloc(sizeof(int)); /*Dynamic allocation of memory for the
array*/

    printf("\nEnter the %d elements of the array arr:\n",N);
    for (i=0;i<N;i++)
        scanf("%d",&arr[i]); /*Reading array elements*/

    printf("\nEnter the element to be searched:\n");
    scanf("%d",&k);

    index=linear(arr,N,k); /*Calling linear function*/

    /*Printing search results*/
    if(index==999)
        printf("\nElement %d is not present in array arr[%d]",k,N);
    else
```

```

        printf("\nElement %d is stored at index location %d in the array
arr[%d]",k,index,N);

        getch();
    }

int linear(int array[], int size, int num)
{
    int i;
    for(i=0;i<size;i++) /*Scanning array elements one by one*/
    if(num==array[i])
        return(i); /*Successful Search*/
    if(i==size)
        return(-999); /*Unsuccessful Search*/
}

```

*Here, -999 is being used as a NULL value to indicate unsuccessful search.*

## Output

Enter the number of elements in the array arr:

8

Enter the 8 elements of the array arr:

3  
2  
18  
33  
21  
5  
99  
42

Enter the element to be searched:

33

Element 33 is stored at index location 3 in the array arr[8]

## Program analysis

Key Statement	Purpose
<b>index=linear(arr,N,k);</b>	Calls the <i>linear()</i> function to perform linear search on the array <i>arr</i>
<b>for(i=0;i&lt;size;i++)</b> <b>if(num==array[i])</b>	Compares each array element with the value that needs to be searched

**Efficiency of Linear Search** Assume that an array containing  $n$  elements is to be searched for the value  $k$ . In the best case,  $k$  would be the first element in the list, thus requiring only one comparison. In the worst case, it would be last element in the list, thus requiring  $n$  comparisons.

To compute the efficiency of linear search we can add all the possible number of comparisons and divide it by  $n$ .

$$\begin{aligned}
 \text{Thus, efficiency of linear search} &= (1 + 2 + \dots + n) / n \\
 &= n(n+1) / 2n \\
 &= O(n)
 \end{aligned}$$

**Advantages and Disadvantages** Some of the key advantages of linear search technique are:

1. It is a simple searching technique that is easy to implement.
2. It does not require the list to be sorted in a particular order.

The disadvantages associated with it are as follows:

1. It is quite inefficient for large sized lists.
2. It does not leverage the presence of any pre-existing sort order in a list.



### Check Point

1. What is the efficiency of linear search?

Ans.  $O(n)$ .

2. What is the key advantage of linear search?

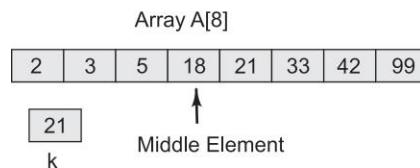
Ans. It does not require the list to be sorted in a particular order.

## 8.3.2 Binary Search

Binary search technique has a prerequisite – it requires the elements of a data structure (list) to be already arranged in a sorted manner before search can be performed in it. It begins by comparing the element that is present at the middle of the list. If there is a match then the search ends immediately and the location of the middle element is returned. However, if there is a mismatch then it focuses the search either in the left or the right sub list depending on whether the target element is lesser than or greater than middle element. The same methodology is repeatedly followed until the target element is found.

Binary search follows the same analogy as that of a telephone directory that we had discussed earlier. One needs to keep focusing on a smaller subset of directory pages every time there is a mismatch. However, such a search would not have been possible had the directory entries were not already sorted.

Consider an array of integers A containing eight elements, as shown in Fig. 8.6. Let k = 21 be the value that needs to be searched.



**Fig. 8.6** Binary search

As we can see in Fig. 8.6, the array A on which binary search is to be performed is already sorted. The following steps describe how binary search is performed on array A to search for value k:

1. First of all, the middle element in the array A is identified, which is 18.
2. Now, k is compared with 18. Since k is greater than 18, the search is focused on the right sub list.
3. The middle element in the right sub list is 33. Since k is less than 33, the search is focused on the left sub list, which is {21, 33}.

- Now, again k is compared with the middle element of {21, 33}, which is 21. Thus, it matches with k.
- The index value of 21, i.e., 4 is returned and the search is considered as successful.

**Example 8.16** Write an algorithm to perform binary search on a given array of integers.

```
binary(arr[], size, num)
Step 1: Start
Step 2: Set i = 0, j = size, k = 0
Step 3: Repeat Steps 4-9 while i <= j
Step 4: Set k = (i + j)/2
Step 5: If arr[k] = num goto Step 6 else goto Step 7
Step 6: return k and goto Step 11
Step 7: If array[k] < num goto Step 8 else goto Step 9
Step 8: i = k + 1
Step 9: j = k - 1
Step 10: Return NULL and goto Step 11
Step 11: Stop
```

**Example 8.17** Write a C program to perform binary search on an array of N elements.

Program 8.8 implements binary search technique in C. It uses the algorithm depicted in Example 8.16.

**Program 8.8** *Implementation of binary search technique*

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>

int binary(int [], int, int); /*Function prototype for performing binary
search*/

void main()
{
    int *arr;
    int i, N, k, index;
    clrscr();

    printf("Enter the number of elements in the array arr:\n");
    scanf("%d",&N);

    arr = (int*) malloc(sizeof(int)); /*Dynamic allocation of memory for the
array*/

    printf("\nEnter the %d elements of the array arr in sorted format:\n",N);
    for (i=0;i<N;i++)
        scanf("%d",&arr[i]); /*Reading array elements*/

    printf("\nEnter the element to be searched:\n");
    scanf("%d",&k);
```

```

index=binary(arr,N,k); /*Calling binary function*/

/*Printing search results*/
if(index== -999)
printf("\nElement %d is not present in array arr[%d]",k,N);
else
printf("\nElement %d is stored at index location %d in the array
arr[%d]",k,index,N);

getch();
}

int binary(int array[], int size, int num)
{
    int i=0,j=size, k;
    while(i<=j)
    {
        k=(i+j)/2;
        if(array[k]==num)
            return(k); /*Successful search*/
        else if(array[k]<num)
            i=k+1;
        else
            j=k-1;
    }
    return(-999); /*Unsuccessful search*/
}

```

## Output

Enter the number of elements in the array arr:  
8

Enter the 8 elements of the array arr in sorted format:

2  
3  
5  
18  
21  
33  
42  
99

*We must ensure that we input elements in a sorted manner as that is the prerequisite for performing binary search.*

Enter the element to be searched:  
21

Element 21 is stored at index location 4 in the array arr[8]

## Program analysis

Key Statement	Purpose
<code>index=binary(arr,N,k);</code>	Calls the <i>binary()</i> function to perform binary search on the array <i>arr</i>
<code>if(array[k]==num) return(k);</code>	Returns the value of <i>k</i> in case of successful search
<code>else if(array[k]&lt;num) i=k+1; else j=k-1;</code>	Updates the values of <i>i</i> and <i>j</i> in case of unsuccessful search

## Efficiency of Binary Search

**Best Case** The best case for a binary search algorithm occurs when the element to be searched is present at the middle of the list. In this case, only one comparison is made to perform the search operation.

Thus, efficiency =  $O(1)$

**Worst Case** The worst case for a binary search algorithm occurs when the element to be searched is not present in the list. In this case, the list is continuously divided until only one element is left for comparison.

Let  $n$  be the number of list elements and  $c$  be the total number of comparisons made in the worst case. Now, after every single comparison, the number of list elements left to be searched is reduced by 2.

Thus,  $c = \log_2 n$

Hence, efficiency =  $O(\log_2 n)$

**Advantages and Disadvantages** Some of the key advantages of binary search technique are:

1. It requires lesser number of iterations.
2. It is a lot faster than linear search.

The disadvantages associated with it are as follows:

1. Unlike linear search, it requires the list to be sorted before search can be performed.
2. In comparison to linear search, the binary search technique may seem to be a little difficult to implement.

### 8.3.3 Hashing

So far we have learnt two of the most fundamental searching techniques, i.e., linear and binary search. Both these searching techniques find their usage across varied programming situations. However, in case of large databases, an altogether different searching technique is widely used. This technique is called hashing.

Hashing finds the location of an element in a data structure without making any comparisons. In contrast to the other comparison-based searching techniques, like linear and binary search, hashing uses



### Check Point

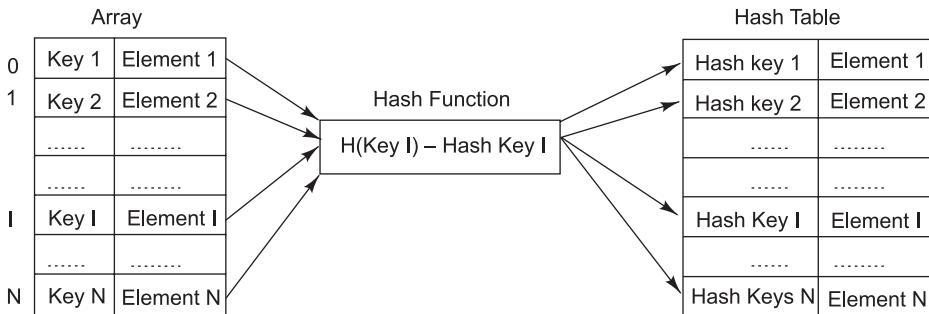
1. What is the worst case efficiency of binary search?

Ans.  $O(\log_2 n)$ .

2. What is the key prerequisite for performing binary search?

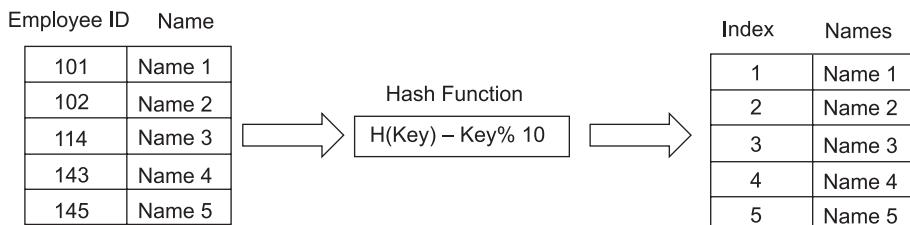
Ans. The key prerequisite for performing binary search is that the input list must already be sorted.

a mathematical function to determine the location of an element. This mathematical function called *hash function* accepts a value, known as *key*, as input and generates an output known as *hash key*. The hash function generates hash keys and stores elements corresponding to each hash key in the hash table. The keys that hash function accepts as input could be a digit associated with the input elements. In other words, we can say that a hash table is a data structure, which is implemented by a hash function and used for searching elements in quick time. In a hash table, hash keys act as the addresses of the elements. Figure 8.7 depicts the hash functionality.



**Fig. 8.7 Hashing**

Let us consider a simple example of a file containing information for five employees of a company. Each record in that file contains the name and a three digit numeric Employee ID of the employee. In this case, the hash function will implement a hash table of five slots using Employee IDs as the keys. That means, the hash function will take Employee IDs as input and generate the hash keys, as shown in Fig. 8.8.



**Fig. 8.8 Generating hash keys**

In the hash table generated in the above example, the hash function is *Employee ID%10*. Thus, for Employee ID 101, hash key will be calculated as 1. Therefore, Name1 will be stored at position 1 in the hash table. For Employee ID 102, hash key will be 2, hence Name2 will be stored at position 2 in the hash table. Similarly, Name3, Name4, and Name5 will be stored at position 4, 3, and 5 respectively, as shown in Fig. 8.5. Later, whenever an employee record is searched using the Employee ID, the hash function will indicate the exact position of that record in the hash table.

**Collision** As we have already learnt, the hash function takes some key values as input, performs some mathematical calculation, and generates hash key to ascertain the position in the hash table where the record corresponding to the key will be stored. However, it is quite possible that the hash function

generates same hash keys for two different key values. That means, two different records are indicated to be stored at the same position in the hash table. This situation is termed as *collision*. As a result, a hash function must be designed in such a way that the possibility of a collision is negligible. Various techniques such as, linear probing, chaining without replacement, and chaining with replacement are used to evade the chances of a collision.

**Perfect Hashing** Perfect hashing ensures that there is no possibility of collision occurrence. It can be achieved only when the set of input keys are known beforehand. As a result, collision prevention measures are programmatically included while developing the hash function.

**Example 8.18** Write a program to implement a hash function. Assume that the input keys are within the range 10001 and 10999.

**Program 8.9** *Implementation of a hash function*

```
#include <stdio.h>
#include <conio.h>

int hash(int); /*Function prototype for generating hash keys*/

void main()
{
    int key, hk;
    clrscr();

    printf("Enter the next key:");
    scanf("%d",&key);

    hk = hash(key);

    printf("\nThe hash key generated for the key %d is %d",key,hk);

    getch();
}

int hash(int k)
{
    return(k - 10000);
}
```

**Output**

```
Enter the next key: 10765
```

```
The hash key generated for the key 10765 is 765
```

**Program analysis**

Key Statement	Purpose
<b>hk = hash(key);</b>	Calls the hash() function to generate the hash key value

## Solved Problems

**Problem 8.1** Consider the following array of integers:

35 18 7 12 5 23 16 3 1

Create a snapshot of the above array at each pass if the bubble sorting technique is applied on it.

### Solution

<b>Initial array</b>	35 18 7 12 5 23 16 3 1
<b>Pass 1</b>	1 35 18 12 7 23 16 5 3
<b>Pass 2</b>	1 3 35 18 12 23 16 7 5
<b>Pass 3</b>	1 3 5 35 18 23 16 12 7
<b>Pass 4</b>	1 3 5 7 35 23 18 16 12
<b>Pass 5</b>	1 3 5 7 12 35 23 18 16
<b>Pass 6</b>	1 3 5 7 12 16 35 23 18
<b>Pass 7</b>	1 3 5 7 12 16 18 35 23
<b>Pass 8</b>	1 3 5 7 12 16 18 23 35 (sorted array)

**Problem 8.2** Consider the following array of integers:

74 39 35 32 97 84

Create a snapshot of the above array at each pass if the selection sorting technique is applied on it.

### Solution

<b>Initial array</b>	74 39 35 32 97 84
<b>Pass 1</b>	32 39 35 74 97 84
<b>Pass 2</b>	32 35 39 74 97 84
<b>Pass 3</b>	32 35 39 74 97 84
<b>Pass 4</b>	32 35 39 74 97 84
<b>Pass 5</b>	32 35 39 74 84 97 (sorted array)

**Problem 8.3** Consider the following array of integers:

35 54 12 18 23 15 45 38

Create a snapshot of the above array at each pass if the quick sorting technique is applied on it.

### Solution

<b>Initial Array</b>	35 54 12 18 23 15 45 38
<b>Pass 1</b>	18 54 12 35 23 15 45 38
<b>Pass 2</b>	18 15 12 35 23 54 45 38
<b>Pass 3</b>	12 15 18 35 23 54 45 38
<b>Pass 4</b>	12 15 18 35 23 54 45 38
<b>Pass 5</b>	12 15 18 35 23 54 45 38
<b>Pass 6</b>	12 15 18 54 23 35 45 38
<b>Pass 7</b>	12 15 18 38 23 35 45 54
<b>Pass 8</b>	12 15 18 23 38 35 45 54



### Check Point

#### 1. What is a hash table?

**Ans.** It is a data structure, which is implemented by a hash function and used for searching elements in quick time.

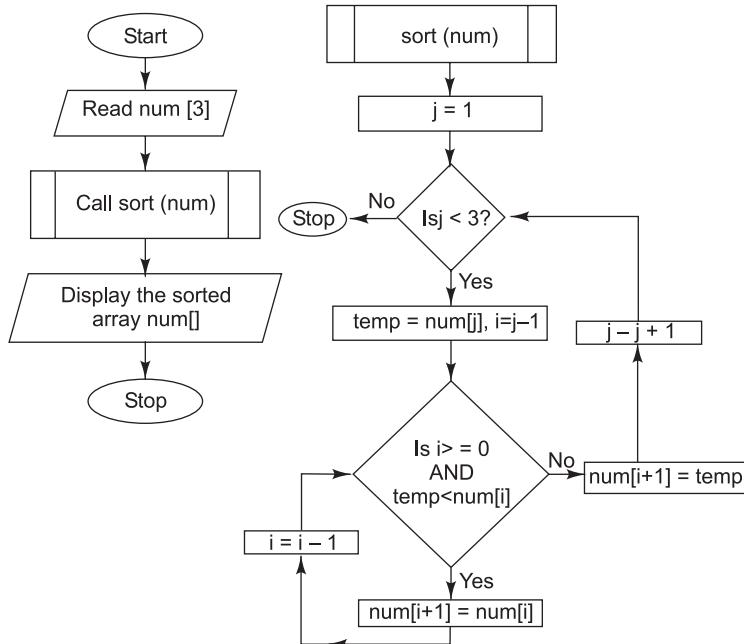
#### 2. What is perfect hashing?

**Ans.** Perfect hashing reduces the possibility of collision occurrence to zero.

**Pass 9** 12 15 18 23 38 35 45 54  
**Pass 10** 12 15 18 23 35 38 45 54  
**Pass 11** 12 15 18 23 35 38 45 54  
**Pass 12** 12 15 18 23 35 38 45 54  
**Pass 13** 12 15 18 23 35 38 45 54 (sorted array)

**Problem 8.4** Draw a flowchart for sorting three integers using insertion sort technique.

#### FLOWCHART



#### Summary

- ◆ Sorting is the process of arranging the numerical and alphabetical data present in a list, in a specific order or sequence.
- ◆ Searching is the process of locating a specific element across a given list of elements.
- ◆ Selection sort works on the principle of identifying the smallest element in the list and moving it to the beginning of the list.
- ◆ Insertion sort method sorts a list of elements by inserting each successive element in the previously sorted sublist. The insertion of elements requires the other elements to be shuffled appropriately.
- ◆ Bubble sort works by bringing the largest element to the end of the list with each successive pass.

- ◆ Quick sort is one of the fastest sorting methods that is based on divide and conqueror approach. It revolves around a key element called pivot to perform the sorting operation.
  - ◆ Merge sort divides a list into several sublists of equal sizes and sorts them individually. The sorted sublists are later merged to form the original list.
  - ◆ Bucket sort distributes the list of elements across different buckets and sorts each of the buckets individually. These buckets are later merged to form the original sorted list.
  - ◆ Linear search is one of the conventional searching techniques that sequentially searches for an element in the list
  - ◆ Binary search works on an already sorted list to perform the search operation. It repetitively looks for the middle element of the list until the target element is found.
  - ◆ Hashing finds the location of an element in a data structure without making any comparisons. It uses the hash function to determine the location of an element.
  - ◆ Collision is a situation where the hash function generates same hash keys for two different key values.
  - ◆ Perfect hashing ensures that there is no possibility of collision occurrence.

Key Terms —————

- ◆ **Pivot** It is a key value that is selected and shuffled continuously as per the quick sort algorithm until it attains its final position in the list.
  - ◆ **Bucket** It represents a logical data structure for temporarily storing the elements that fall within a specific range.
  - ◆ **Hash** It is a mathematical function that generates hash keys for indicating the location where elements are to be stored in the hash table
  - ◆ **Hash table** It is a data structure, which is implemented by a hash function and used for searching elements in quick time.

## Multiple-Choice Questions

- 8.6** Pivot element is associated with which of the following?

  - (a) Binary search
  - (b) Quick sort
  - (c) Selection sort
  - (d) Hashing

**8.7** Which of the following searching techniques is most suitable for large databases?

  - (a) Hashing
  - (b) Linear
  - (c) Binary
  - (d) All of the above

**8.8** What is the best case efficiency of binary search?

  - (a)  $O(1)$
  - (b)  $O(0)$
  - (c)  $O(n)$
  - (d) None of the above

## Review Questions

- 8.1** What is sorting? List the various sorting techniques.
  - 8.2** What is searching? List the various searching techniques.
  - 8.3** Why quick sort is considered the fastest sorting technique?
  - 8.4** What is a pivot element? Where is it used?
  - 8.5** Deduce the worst case efficiency of binary search technique.
  - 8.6** What is bubble sorting? What is it considered to be slow?
  - 8.7** Explain the insertion and selection sorting techniques with examples.
  - 8.8** What is hashing? Explain with example.
  - 8.9** What is a collision? How can it be prevented?
  - 8.10** Explain the role of buckets in bucket sorting technique.

## Programming Exercises

- 8.1 Write a program in C that takes as input five integers and displays them in a sorted sequence.
  - 8.2 Write a C function that applies the bubble sorting technique to sort a set of alphanumeric characters as per their ASCII values.
  - 8.3 Write a C program that uses the insertion sorting technique to sort an array of structures. The sorting must be performed on the basis of one of the structure members.
  - 8.4 Write a C function that performs linear search on an array of real values.
  - 8.5 Write a C program that sorts the given set of integers and performs binary search on them.

# Answers to Multiple-Choice Questions

8.1 (a)                  8.2 (b)                  8.3 (d)                  8.4 (a)                  8.5 (a)  
8.6 (b)                  8.7 (a)                  8.8 (a)