Data Structures
Using C

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. His best-selling books, among others include

- *Fundamentals of Computers*
- *Computing Fundamentals and C Programming*
- *Programming in ANSI C*, 6e
- *Programming in Java*, 4e
- *Programming in BASIC*, 3e
- *Programming in C#*, 3e
- *Numerical Methods*
- *Reliability Engineering*

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 Using C

E Balagurusamy

Chairman
EBG Foundation
Coimbatore

McGraw Hill Education (India) Private Limited
NEW DELHI

McGraw Hill Education Offices

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

McGraw Hill Education (India) Private Limited

Published by McGraw Hill Education (India) Private Limited
P-24, Green Park Extension, New Delhi 110 016

Data Structures Using C

ISBN (13): 978-1-25-902954-7
ISBN (10): 1-25-902954-9

Vice President and Managing Director: *Ajay Shukla*

Head—Higher Education Publishing and Marketing: *Vibha Mahajan*
Publishing Manager—SEM & Tech Ed.: *Shalini Jha*
Asst. Sponsoring Editor: *Smruti Snigdha*
Editorial Researcher: *Amiya Mahapatra*
Manager—Production Systems: *Satinder S. Baveja*
Sr. Production Manager: *P. L. Pandita*
Copy Editor: *Preyoshi Kundu*

Asst. General Manager—Higher Education Marketing: *Vijay Sarathi*
Sr. Product Specialist—SEM and Tech Voc.: *Tina Jajoriya*
Sr. Graphic Designer—Cover: *Meenu Raghav*

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

Typeset at Bukprint India, B-180A, Guru Nanak Pura, Laxmi Nagar-110 092 and printed at Sheel Printers Pvt. Ltd., D-132, Hoisery Complex, Phase II, Noida

Cover: Sheel Printers Pvt. Ltd.

RALYCRCORBXXX

Contents

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 Using C* 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.

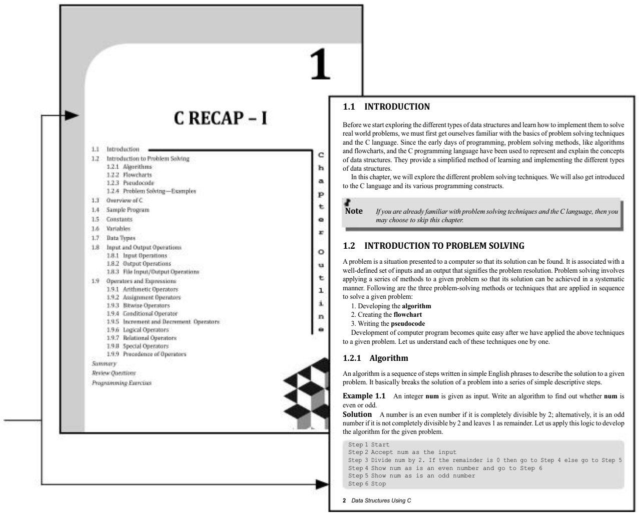
Salient Features of the Book

- 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 *over 400 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.



1

C RECAP – I

1.1 Introduction
1.2 Introduction to Problem Solving
  1.2.1 Algorithms
  1.2.2 Flowcharts
  1.2.3 Pseudocode
  1.2.4 Problem Solving—Examples
1.3 Overview of C
1.4 Sample Program
1.5 Constants
1.6 Variables
1.7 Data Types
1.8 Input and Output Operations
  1.8.1 Input Operations
  1.8.2 Output Operations
  1.8.3 File Input/Output Operations
1.9 Operators and Expressions
  1.9.1 Arithmetic Operators
  1.9.2 Assignment Operators
  1.9.3 Bitwise Operators
  1.9.4 Conditional Operator
  1.9.5 Increment and Decrement Operators
  1.9.6 Logical Operators
  1.9.7 Relational Operators
  1.9.8 Special Operators
  1.9.9 Precedence of Operators
Summary
Review Questions
Programming Exercises

Chapter Outline

1.1 INTRODUCTION

Before we start exploring the different types of data structures and learn how to implement them to solve real world problems, we must first get ourselves familiar with the basics of problem solving techniques and the C language. Since the early days of programming, problem solving methods, like algorithms and flowcharts, and the C programming language have been used to represent and explain the concepts of data structures. They provide a simplified method of learning and implementing the different types of data structures.

In this chapter, we will explore the different problem solving techniques. We will also get introduced to the C language and its various programming constructs.

Note *If you are already familiar with problem solving techniques and the C language, then you may choose to skip this chapter.*

1.2 INTRODUCTION TO PROBLEM SOLVING

A problem is a situation presented to a computer so that its solution can be found. It is associated with a well-defined set of inputs and an output that signifies the problem resolution. Problem solving involves applying a series of methods to a given problem so that its solution can be achieved in a systematic manner. Following are the three problem-solving methods or techniques that are applied in sequence to solve a given problem:

1. Developing the **algorithm**
2. Creating the **flowchart**
3. Writing the **pseudocode**

Development of computer program becomes quite easy after we have applied the above techniques to a given problem. Let us understand each of these techniques one by one.

1.2.1 Algorithm

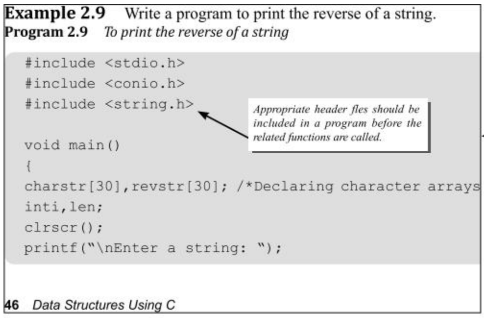
An algorithm is a sequence of steps written in simple English phrases to describe the solution to a given problem. It basically breaks the solution of a problem into a series of simple descriptive steps.

Example 1.1 An integer **num** is given as input. Write an algorithm to find out whether **num** is even or odd.

Solution A number is an even number if it is completely divisible by 2; alternatively, it is an odd number if it is not completely divisible by 2 and leaves 1 as remainder. Let us apply this logic to develop the algorithm for the given problem.

```
Step 1 Start
Step 2 Accept num as the input
Step 3 Divide num by 2. If the remainder is 0 then go to Step 4 else go to Step 5
Step 4 Show num as is an even number and go to Step 6
Step 5 Show num as is an odd number
Step 6 Stop
```

2 *Data Structures Using C*



Example 2.9 Write a program to print the reverse of a string.

Program 2.9 *To print the reverse of a string*

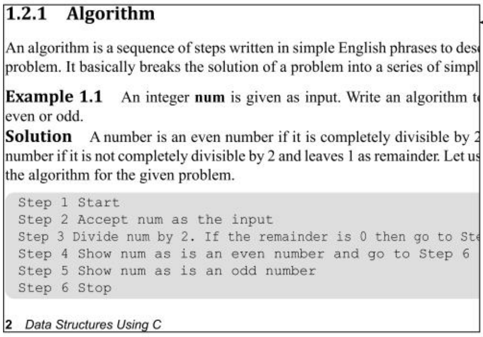
```
#include <stdio.h>
#include <conio.h>
#include <string.h>

void main()
{
charstr[30],revstr[30]; /*Declaring character arrays
inti,len;
clrscr();
printf("\nEnter a string: ");
```

Appropriate header fles should be included in a program before the related functions are called.

46 *Data Structures Using C*

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.



1.2.1 Algorithm

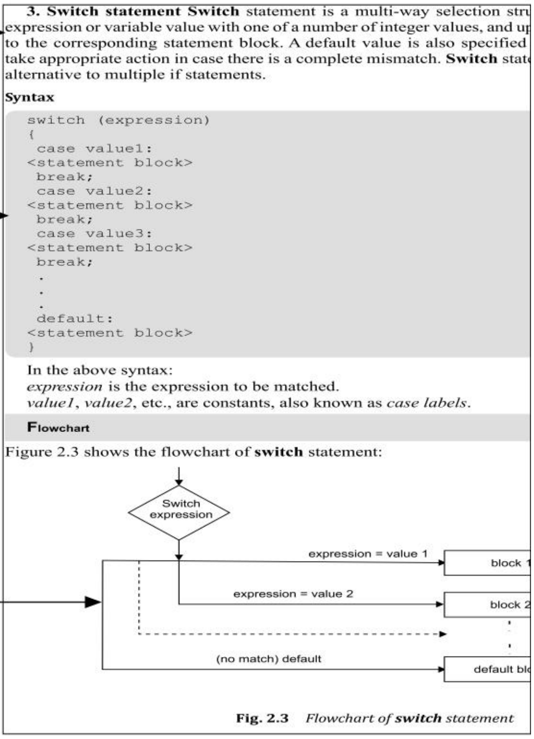
An algorithm is a sequence of steps written in simple English phrases to des
problem. It basically breaks the solution of a problem into a series of simpl

Example 1.1 An integer **num** is given as input. Write an algorithm t
even or odd.

Solution A number is an even number if it is completely divisible by 2
number if it is not completely divisible by 2 and leaves 1 as remainder. Let us
the algorithm for the given problem.

```
Step 1 Start
Step 2 Accept num as the input
Step 3 Divide num by 2. If the remainder is 0 then go to Ste
Step 4 Show num as is an even number and go to Step 6
Step 5 Show num as is an odd number
Step 6 Stop
```

2 *Data Structures Using C*



3. Switch statement Switch statement is a multi-way selection stru
expression or variable value with one of a number of integer values, and up
to the corresponding statement block. A default value is also specified
take appropriate action in case there is a complete mismatch. **Switch** stat
alternative to multiple if statements.

Syntax

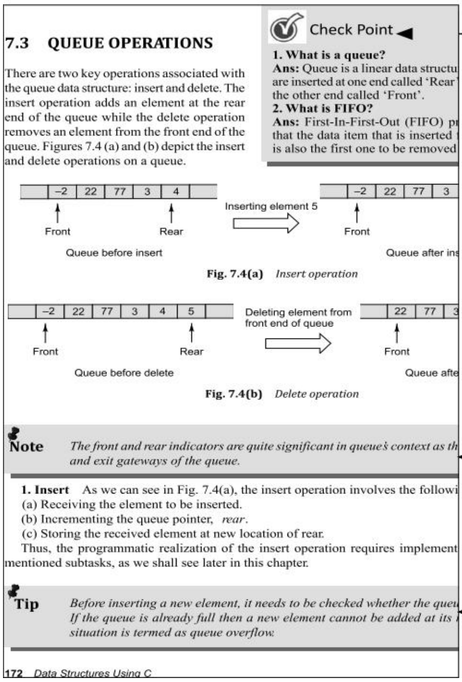
```
switch (expression)
{
 case value1:
<statement block>
 break;
 case value2:
<statement block>
 break;
 case value3:
<statement block>
 break;
 .
 .
 .
 default:
<statement block>
}
```

In the above syntax:
expression is the expression to be matched.
value1, *value2*, etc., are constants, also known as *case labels*.

Flowchart

Figure 2.3 shows the flowchart of **switch** statement:

Fig. 2.3 *Flowchart of* ***switch*** *statement*



7.3 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 7.4 (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 structu
are inserted at one end called 'Rear'
the other end called 'Front'.
2. What is FIFO?
Ans: First-In-First-Out (FIFO) pr
that the data item that is inserted
is also the first one to be removed

Fig. 7.4(a) *Insert operation*

Fig. 7.4(b) *Delete operation*

Note *The front and rear indicators are quite significant in queue's context as th and exit gateways of the queue.*

1. Insert As we can see in Fig. 7.4(a), the insert operation involves the followi
(a) Receiving the element to be inserted.
(b) Incrementing the queue pointer, *rear*.
(c) Storing the received element at new location of rear.
Thus, the programmatic realization of the insert operation requires implement
mentioned subtasks, as we shall see later in this chapter.

Tip *Before inserting a new element, it needs to be checked whether the queu If the queue is already full then a new element cannot be added at its situation is termed as queue overflow.*

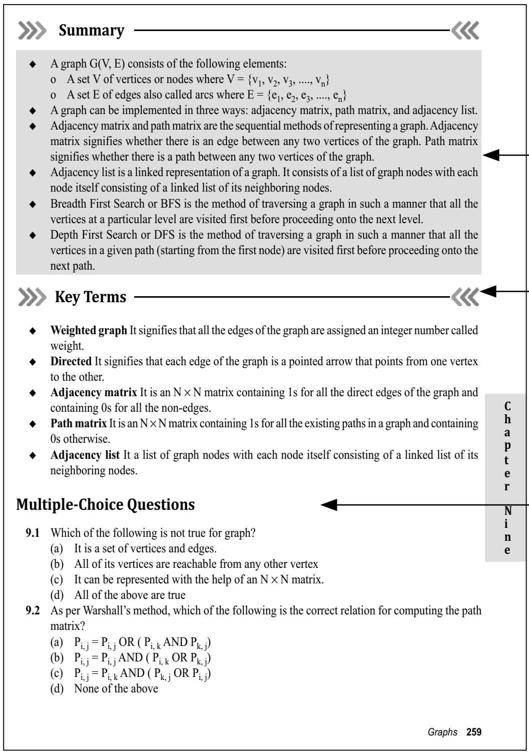
172 *Data Structures Using C*

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.



Summary

- A graph G(V, E) consists of the following elements:
  - o A set V of vertices or nodes where V = {v_1, v_2, v_3, ..., v_n}
  - o A set E of edges also called arcs where E = {e_1, e_2, e_3, ..., 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 × 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 × 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.

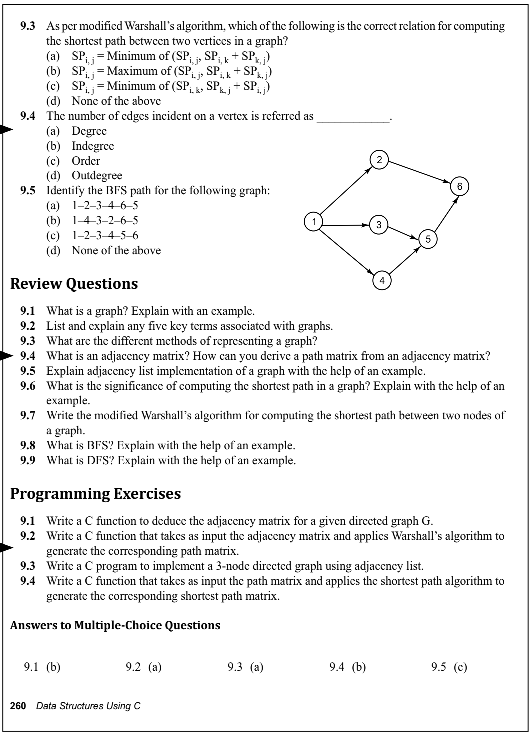
Chapter Nine

Multiple-Choice Questions

9.1 Which of the following is not true for graph?
(a) It is a set of vertices and edges.
(b) All of its vertices are reachable from any other vertex
(c) It can be represented with the help of an N × N matrix.
(d) All of the above are true

9.2 As per Warshall's method, which of the following is the correct relation for computing the path matrix?
(a) $P_{i,j} = P_{i,j}$ OR ($P_{i,k}$ AND $P_{k,j}$)
(b) $P_{i,j} = P_{i,j}$ AND ($P_{i,k}$ OR $P_{k,j}$)
(c) $P_{i,j} = P_{i,k}$ AND ($P_{k,j}$ OR $P_{i,j}$)
(d) None of the above

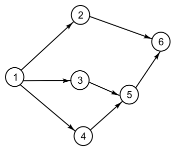
Graphs **259**



9.3 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?
(a) $SP_{i,j}$ = Minimum of ($SP_{i,j}$, $SP_{i,k} + SP_{k,j}$)
(b) $SP_{i,j}$ = Maximum of ($SP_{i,j}$, $SP_{i,k} + SP_{k,j}$)
(c) $SP_{i,j}$ = Minimum of ($SP_{i,k}$, $SP_{k,j} + SP_{i,j}$)
(d) None of the above

9.4 The number of edges incident on a vertex is referred as ___________.
(a) Degree
(b) Indegree
(c) Order
(d) Outdegree

9.5 Identify the BFS path for the following graph:
(a) 1–2–3–4–6–5
(b) 1–4–3–2–6–5
(c) 1–2–3–4–5–6
(d) None of the above



Review Questions

9.1 What is a graph? Explain with an example.
9.2 List and explain any five key terms associated with graphs.
9.3 What are the different methods of representing a graph?
9.4 What is an adjacency matrix? How can you derive a path matrix from an adjacency matrix?
9.5 Explain adjacency list implementation of a graph with the help of an example.
9.6 What is the significance of computing the shortest path in a graph? Explain with the help of an example.
9.7 Write the modified Warshall's algorithm for computing the shortest path between two nodes of a graph.
9.8 What is BFS? Explain with the help of an example.
9.9 What is DFS? Explain with the help of an example.

Programming Exercises

9.1 Write a C function to deduce the adjacency matrix for a given directed graph G.
9.2 Write a C function that takes as input the adjacency matrix and applies Warshall's algorithm to generate the corresponding path matrix.
9.3 Write a C program to implement a 3-node directed graph using adjacency list.
9.4 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

9.1 (b)   9.2 (a)   9.3 (a)   9.4 (b)   9.5 (c)

260 *Data Structures Using C*

Chapter Organization

This book is organized into 11 chapters, which explain concepts like Arrays, Stacks, Queues, Linked Lists, Trees and Graphs.

Chapters 1 and 2 provide a quick recap to the C programming language. **Chapter 3** introduces algorithm and its related concepts. It also provides a brief introduction to the different types of data structures. **Chapter 4** 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 5** explains the concept of linked list along with its different variants. **Chapters 6 and 7** 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 8 and 9**

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 10** 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. Finally, **Chapter 11** explains how all the data structures taught previously are put into use for solving mathematical and other real-world problems.

Web Supplements

Following materials can be accessed at ***http://www.mhhe.com/balagurusamy/dsuc***

- *PowerPoint slides*
- *Computer programs for labs*
- *Links for additional resources*

Acknowledgements

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

Shashank Dwivedi	*UCER, Allahabad, Uttar Pradesh*
Rajiv Pandey	*Amity University Lucknow Campus, Lucknow, Uttar Pradesh*
Mahua Banerjee	*Xavier Institute of Social Service, Ranchi, Jharkhand*
Sameer Bhave	*Indore Professional Studies Academy, Indore, Madhya Pradesh*
D Lakshmi, Adithya	*Institute of Technology, Coimbatore, Tamil Nadu*
A Sharada	*G Narayanamma Institute of Technology and Science, Hyderabad, Andhra Pradesh*

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

Publisher's Note

Suggestions and constructive criticism always go a long way in enhancing any endeavor. We request all readers to email us their valuable comments/views/feedback for the betterment of the book at *tmh.csefeedback@gmail.com* mentioning the title and the author's name in the subject line.

Please report any piracy spotted by you as well!

1

C RECAP – I

Chapter Outline

1.1 INTRODUCTION

Before we start exploring the different types of data structures and learn how to implement them to solve real world problems, we must first get ourselves familiar with the basics of problem solving techniques and the C language. Since the early days of programming, problem solving methods, like algorithms and flowcharts, and the C programming language have been used to represent and explain the concepts of data structures. They provide a simplified method of learning and implementing the different types of data structures.

In this chapter, we will explore the different problem solving techniques. We will also get introduced to the C language and its various programming constructs.

Note *If you are already familiar with problem solving techniques and the C language, then you may choose to skip this chapter.*

1.2 INTRODUCTION TO PROBLEM SOLVING

A problem is a situation presented to a computer so that its solution can be found. It is associated with a well-defined set of inputs and an output that signifies the problem resolution. Problem solving involves applying a series of methods to a given problem so that its solution can be achieved in a systematic manner. Following are the three problem-solving methods or techniques that are applied in sequence to solve a given problem:

1. Developing the **algorithm**
2. Creating the **flowchart**
3. Writing the **pseudocode**

Development of computer program becomes quite easy after we have applied the above techniques to a given problem. Let us understand each of these techniques one by one.

1.2.1 Algorithm

An algorithm is a sequence of steps written in simple English phrases to describe the solution to a given problem. It basically breaks the solution of a problem into a series of simple descriptive steps.

Example 1.1 An integer **num** is given as input. Write an algorithm to find out whether **num** is even or odd.

Solution A number is an even number if it is completely divisible by 2; alternatively, it is an odd number if it is not completely divisible by 2 and leaves 1 as remainder. Let us apply this logic to develop the algorithm for the given problem.

```
Step 1 Start
Step 2 Accept num as the input
Step 3 Divide num by 2. If the remainder is 0 then go to Step 4 else go to Step 5
Step 4 Show num as is an even number and go to Step 6
Step 5 Show num as is an odd number
Step 6 Stop
```

As we can see in the above algorithm, the solution to the given problem has been described in the form of a series of steps. Each algorithm starts and ends with a Start and Stop statement, respectively. Table 1.1 lists the key characteristics, advantages and disadvantages of algorithms:

Table 1.1 *Characteristics, advantages, and disadvantages of algorithms*

Characteristics	Advantages	Disadvantages
An algorithm completely solves the given problem.	It eases the process of actual development of program code.	For large algorithms, it becomes difficult to understand the flow of program control.
Algorithm instructions are simple and concise.	It allows the programmers to use the most efficient solution as per time and space complexity.	It lacks the visual representation of programming logic as is prevalent in flowcharts.
Algorithm instructions are ordered.	It breaks down the solution of a problem into a series of simplified sequential steps.	There are no standard conventions to be followed while developing algorithms.
An algorithm begins with *Start* and ends with *Stop* instruction.	Its simplified way of representing program instructions enables other programmers to easily understand and modify it.	It may take considerable amount of time to write the algorithm for a given problem.

1.2.2 Flowchart

A flowchart can be referred as a pictorial representation of an algorithm. It uses various graphical elements to describe the flow of information and control. The objective of using flowcharts to describe the problem solution is to ease the understanding of programming logic. This helps in developing the corresponding programming code easier and faster.

Example 1.2 Using flowcharts, solve the problem given in Example 1.1.

Solution The logic to solve the problem given in Example 1.1 remains the same. Let us apply the logic to draw a flowchart.

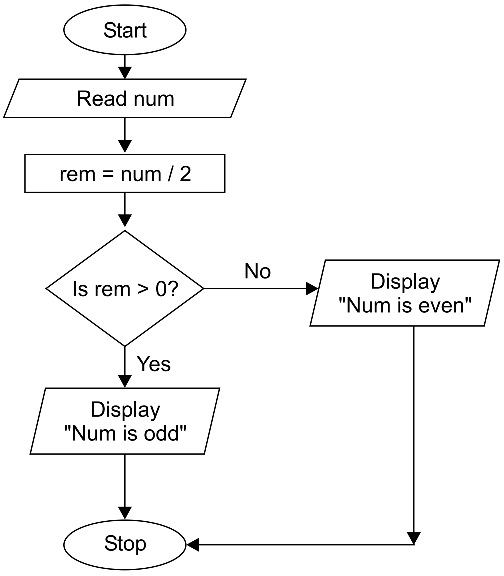
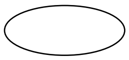
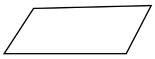
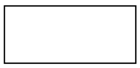
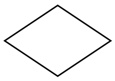
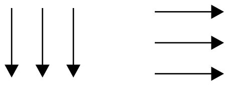
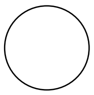
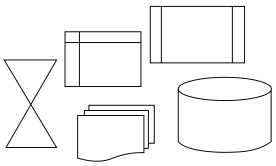


Fig. 1.1 *Flowchart to determine even or odd number*

As shown in Fig. 1.1, different symbols are used in a flowchart to represent different operations. Table 1.2 lists the key flowchart symbols and along with their description:

Table 1.2 *Flowchart symbols*

Name	Symbol	Description
Start and End		Represents the start or end of a process.
Input and Output		Represents an input or output operation.
Process		Represents a processing step that performs certain computations.
Decision		Represents a decision-making step.
Arrow		Represents the flow of control and information.
Connector		Represents the continuation of steps when a flowchart spans across multiple pages.
Additional Symbols		Represent advanced operations.

Similar to algorithms, flowcharts also have certain key characteristics, advantages and disadvantages associated with them. These are described in Table 1.3.

Table 1.3 *Characteristics, advantages, and disadvantages of flowcharts*

Characteristics	Advantages	Disadvantages
Standard flow of control in a flowchart is either from top to bottom or from left to right.	The visual representation of flow of program control is easier to understand.	It does not work well for large programs. The flowchart becomes too complex and confusing.
Flowchart instructions are simple and concise.	It eases the process of actual development of program code.	A slight modification in a flowchart may require significant amount of rework.
None of the arrows in a flowchart intersect each other.	Flowcharts ease debugging and help in identifying and removing logical errors.	The use of graphical elements makes it a little tedious and time consuming to draw a flowchart.
A flowchart always begins with a Start symbol and ends with a Stop symbol.	It acts as documentation for program flow.	At times, excessive use of connectors may become a little more confusing.

1.2.3 Pseudocode

Pseudocode takes the algorithm one step further. It represents the problem solution with the help of generic syntax and normal English phrases. It makes the development of program code easier by utilizing high-level programming constructs for representing conditional and looping scenarios.

Example 1.3 Using pseudocode, solve the problem given in Example 1.1.

Solution The logic to solve the problem given in Example 1.1 remains the same. Let us apply the logic to write the corresponding pseudocode.

```
BEGIN
DEFINE: Integer num
DISPLAY: "Enter a number: "
READ: num
IF: num%2=0
DISPLAY: "'num' is an even number"
ELSE
DISPLAY: "'num' is an odd number"
END IF
END
```

As we can see in the above pseudocode, the solution to the given problem has been described with the help of various labels each representing certain programming action. Each pseudocode starts and ends with a **BEGIN** and **END** statement, respectively. Table 1.4 lists the key characteristics, advantages and disadvantages of pseudocodes.

Table 1.4 *Characteristics, advantages, and disadvantages of pseudocodes*

Characteristics	Advantages	Disadvantages
It represents each solution step with a simple and concise action statement.	It is simple and easy to understand.	It lacks visual representation of programming logic like flowcharts.
It uses generic labels to describe programming constructs.	Converting a pseudocode into actual program code is a lot easier.	There are no standard specifications for developing pseudocodes.
It completely solves the given problem.	It is easy to modify and update pseudocodes as compared to algorithms and flowcharts.	Unlike flowcharts, it may become a little difficult to understand the flow of program control in a pseudocode.
It starts with **BEGIN** statement and ends with an **END** statement.	It is quite flexible and does not require the programmer to memorize any special symbols.	A pseudocode written for a complex problem may become quite lengthy.

1.2.4 Examples of Problem Solving

Table 1.5 shows the application of problem solving techniques to solve common problems.

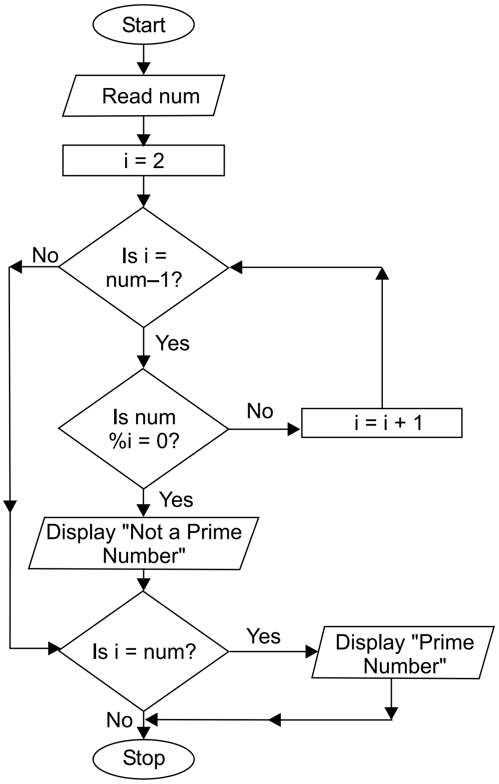
Table 1.5 *Applying problem solving techniques*

Problem 1 An integer **num** is given as input. Find out whether **num** is prime or not.

Algorithm	Pseudocode

`Step 1 - Start`
`Step 2 – Accept a number from the user (num)`
`Step 3 – Initialize looping counter i = 2`
`Step 4 – Repeat Step 5 while i < num`
`Step 5 – If remainder of num divided by i (num%i) is 0 then go to Step 6 else go to Step 4`
`Step 6 - Display "num is not a prime number" and break from the loop`
`Step 7 – If i=num then go to Step 8 Else go to Step 9`
`Step 8 – Display "num is a prime number"`
`Step 9 - Stop`
	`BEGIN`
	`DEFINE: Integer num, i`
	`DISPLAY: "Enter a number: "`
	`READ: num`
	`  FOR: i = 2 to num-1`
	`  IF: num%i=0`
	`DISPLAY: "'num' is not a prime number"`
	`BREAK`
	`  END IF`
	`END FOR`

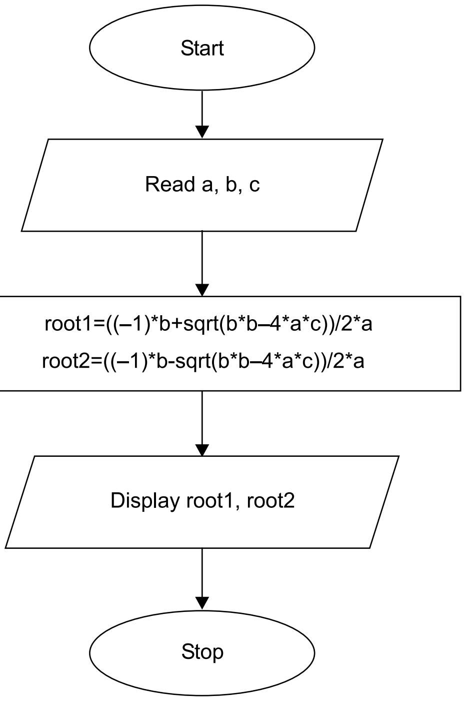
Flowchart



Problem 2 Apply the problem solving techniques to find the roots of a quadratic equation.

Algorithm	Pseudocode
Step 1 - Start Step 2 – Accept three numbers (a, b, c) from the user for the quadratic equation ax^2 + bx + c Step 3 – Calculate root1=((-1)*b+sqrt(b*b-4*a*c))/2*a Step 4 – Calculate root2=((-1)*b-sqrt(b*b-4*a*c))/2*a Step 5 – Display the computed roots of the quadratic equation Step 6 - Stop	BEGIN DEFINE: Integer a, b, c DEFINE: Real root1, root2 DISPLAY: "Enter the values of a, b and c for the quadratic equation ax^2 + bx + c: " READ: a, b, c COMPUTE: root1=((-1)*b+sqrt(b*b-4*a*c))/2*a COMPUTE: root2=((-1)*b-sqrt(b*b-4*a*c))/2*a DISPLAY: "The roots of the quadratic equation are 'root1' and 'root2' END

FLOWCHART

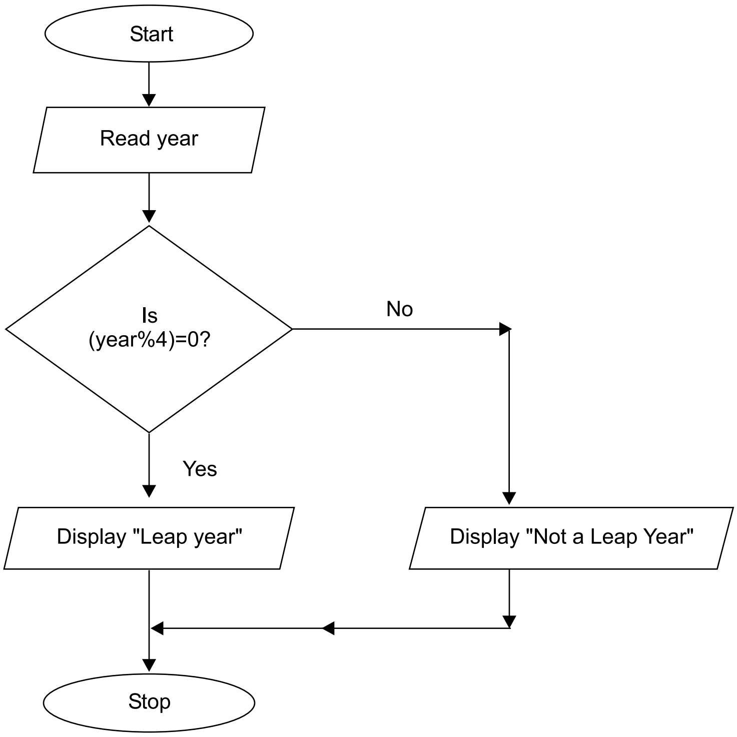

Start
Read a, b, c
root1=((–1)*b+sqrt(b*b–4*a*c))/2*a
root2=((–1)*b-sqrt(b*b–4*a*c))/2*a
Display root1, root2
Stop

Problem 3 Apply the problem solving techniques to determine whether the given year is a leap year or not.

Algorithm	Pseudocode

Step 1 - Start
Step 2 – Accept an year value from the user (year)
Step 3 – If remainder of year value divided by 4 (year%4) is 0 then go to Step 4 else go to Step 5
Step 4 – Display "'year' is a leap year" and go to Step 6
Step 5 – Display "'year' is not a leap year"]
Step 6 - Stop
	BEGIN
	DEFINE: Integer year
	DISPLAY: "Enter the year value: "
	READ: year
	IF: year%4=0
	  DISPLAY: "'year' is a leap year"
	ELSE
	  DISPLAY: "'year' is not a leap year"
	END IF
	END

Flowchart

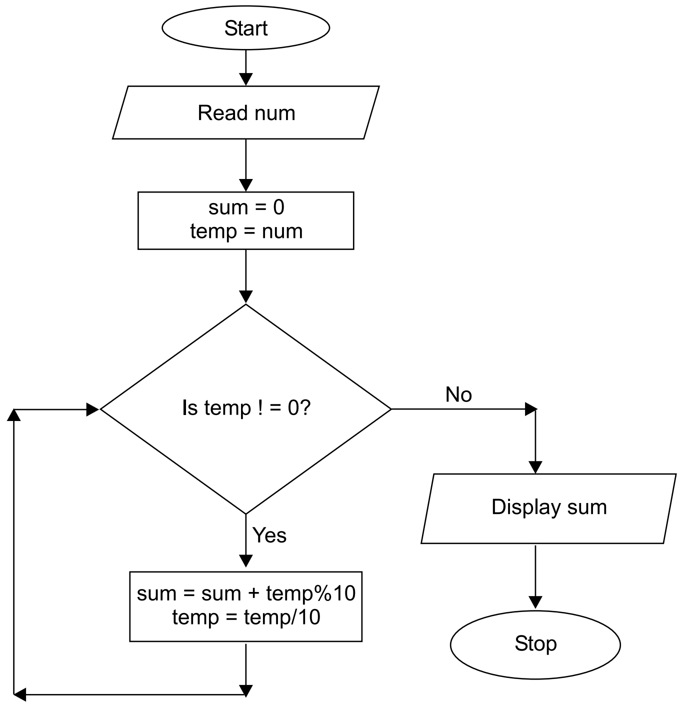


Problem 4 Apply the problem solving techniques, to find the sum of digits of an integer.

Algorithm	Pseudocode

`Step 1 - Start`
`Step 2 – Accept an integer value from the user (num)`
`Step 3 – Define a variable Sum to store the sum of digits and initialize it to 0`
`Step 4 – Assign the value of num to a temporary variable (temp=num)`
`Step 5 – Repeat Steps 6-7 while temp is not equal to 0 (temp!=0)`
`Step 6 – Calculate Sum = Sum+(temp%10)`
`Step 7 – Calculate temp=temp/10`
`Step 8 – Display Sum as the result containing sum of digits of num`
`Step 9 - Stop`
	`BEGIN`
	`DEFINE: Long Integer num, temp`
	`DEFINE: Integer sum`
	`SET: sum=0`
	`DISPLAY: "Enter an integer value: "`
	`READ: num`
	`SET: temp=num`
	`REPEAT`
	`  COMPUTE: sum = sum+temp%10`
	`  COMPUTE: temp=temp/10`
	`UNTIL: temp!=0`
	`DISPLAY: "The sum of digits of 'num' is 'sum'"`
	`END`

FLOWCHART

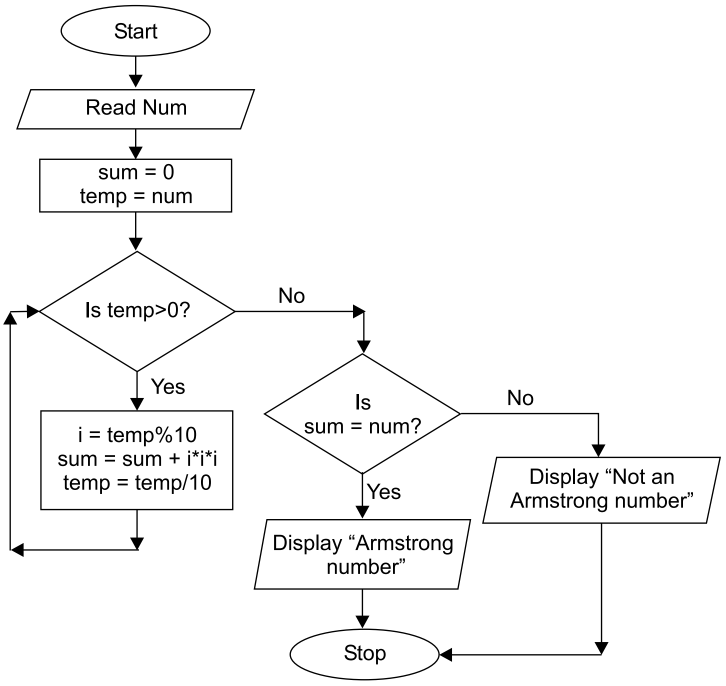


Problem 5 Apply the problem solving techniques, to determine whether a given number is Armstrong or not.

Algorithm	Pseudocode

`Step 1 - Start`
`Step 2 – Accept a number from the user (num)`
`Step 3 – Store the value of num in a temporary variable temp, temp=num`
`Step 4 – Define a variable sum and initialize it to 0`
`Step 5 – Repeat Steps 6-8 while temp > 0`
`Step 6 – Calculate i=temp%10;`
`Step 7 – Calculate sum=sum+i*i*i;`
`Step 8 – Calculate temp=temp/10;`
`Step 9 – if num is equal to sum then go to Step 10 else go to Step 11`
`Step 10 – Display "num is an Armstrong number" and go to Step 12`
`Step 11 – Display "num is not an Armstrong number"`
`Step 12 - Stop`
	`BEGIN`
	`DEFINE: Integer num, temp, sum, i`
	`SET: sum = 0`
	`DISPLAY: "Enter a number: "`
	`READ: num`
	`SET: temp=num`
	`REPEAT`
	`  COMPUTE: i=temp%10`
	`  COMPUTE: sum=sum+i*i*i`
	`  COMPUTE: temp=temp/10`
	`UNTIL: temp>0`
	`IF: sum=num`
	`  DISPLAY: "'num' is an Armstrong number"`
	`ELSE`
	`  DISPLAY: "'num' is not an Armstrong number"`
	`END IF`
	`END`

FLOWCHART



1.3 OVERVIEW OF C

C language was developed by Dennis Ritchie at Bell Laboratories in the year 1972. Its powerful features, modular programming approach and simple syntax made C one of the most preferred languages in the late 70s and 80s amongst programmers. It was widely used for systems and application programming. Even today, C is a popular choice for building device drivers and networking applications. It is considered as an ideal language for someone who wants to begin his journey into the world of computer programming.

Following are some of the key features of C language:

1. **It is a middle-level language** It combines the features of both low level and high level languages.
2. **It follows the structured programming approach** It divides a large program into a number of smaller modules.
3. **It is machine independent** A C program written on one computer can be easily ported to another computer.
4. **It is extensible** It supports a number of built-in functions for performing common operations. These functions are stored in standard libraries. Programmers can also add their user-defined functions to these libraries.
5. **It supports a variety of data types and operators;** thus, making program development fast and efficient.

1.4 SAMPLE PROGRAM

Below is a sample of C program:

```
#include <stdio.h>
/*This is a sample program*/
int main ()
{
 printf("C Recap\n");
}
```

Let us examine the various elements of the above program:

1. # include <stdio.h> This is a file-inclusion preprocessor directive. It includes the contents of the header file **stdio.h** into the program before compilation. The input/output functions **printf** and **scanf** as well as some macro definitions are stored in the standard input/output (**stdio**) header file. Using the **#include** preprocessing directive, we can include other header files as well, such as **conio.h** and **math.h**. The choice of a particular header file depends on the programming situation at hand.

2. /*This is sample program*/ This is a comment statement. It specifies the descriptive statements related to the program. The compiler treats the sequence of characters enclosed within **/*** and ***/** symbols as wide space characters, and skips their execution. Comment statements are primarily used to include certain key information about the programming logic. It helps the other programmers to easily update the program, whenever required.

3. int main () This is the starting point of a program's execution. It redirects the control to the other functions included in the program. The control is sent back to the main function at the end of the called function or when the return statement is encountered.

{ } The opening and closing braces are used to group the executable instructions/statements of a program.

printf This is a standard library function that prints the text enclosed within quotes (" ") on to the console screen. The \n character used inside the **printf** function is a type of escape sequence that prints the subsequent text in new line. There are also other escape sequences available, such as \t to insert tab, \f for a new page, and \r for carriage return.

These are the typical elements of a C program. In addition to these elements, there are various other elements used in a program, such as user-defined functions, built-in functions, variables, constants and so on. We shall learn about all these programming elements and much more in the subsequent sections.

1.5 CONSTANTS

C allows you to define certain entities in a program, whose values do not change during the course of program execution. Such entities are known as *constants*. A constant can be of various types, as explained below:

1. **Integer constants** Represent integer values.
2. **Real constants** Represent fractional or floating point values.
3. **Character constants** Represent single characters enclosed within single quotes (' ').
4. **String constants** Represent a string of characters enclosed within double quotes (" ").

You should not use any comma or blank space while defining integer, real, or character constants.

Note *Since the value of constants remains the same throughout the execution of a program, the compiler will show an error whenever you try to assign a new value to a constant.*

You can define a constant in any of the following ways:

1. **#define statement** You can use this preprocessor directive at the beginning of a program to define the name and value of a constant. The directive replaces the constant by its value at each of its occurrence in the program. Such constant is also referred as symbolic constant.
   Syntax `#define <Constant_name> <Constant_value>`
   Example `#define g 9.8`
2. **const keyword** You can use the **const** *keyword to define a constant in a program and assign it a value.*
   Syntax `const datatype <Constant_name> <Constant_value>`
   Example `const float g 9.8`
3. **Enumerated data type** You can use the **enum** keyword to define a constant as an enumerated data type and assign it a set of constant values.
   Syntax `enum <Constant_name> <set of values>`
   Example `enum option {yes, no};`

1.6 VARIABLES

Unlike constants, variables are the entities that can take different values during the course of execution of a program. Basically, a variable represents the name of the memory location where the variable value is stored.

The rules for defining a variable vary from one compiler to the compiler. As far as the length of the variable name is concerned, the limit is not strictly defined; though, a variable name of eight characters is generally preferred. Also, the pre-defined C keywords, such as **int** or **char** cannot be used as a variable name in a program.

You may define a variable with or without an initial value. The initial value, if assigned to a variable, may change during the program's execution.

The syntax and valid examples of variable declaration are given below.

Syntax `Datatype <Variable_name>;`

```
    Datatype <Variable_name> = <initial value>;
```

Example `int a;`

or

```
int a = 10;
```

Example 1.4 Write a C program to demonstrate the use of variables and constants.

The following program uses the **#define** statement to define a symbolic constant.

Program 1.1 *Use of **#define** statement*

```
/*Program for demonstrating use of Symbolic Constants*/
#include <stdio.h>
#include <conio.h>
#define CP 50 /*Defining Symbolic Constant for Cost Price*/
void main( )
{
 int SP = 70;
 int profit = SP-CP;
 printf("Profit = %d",profit);
 getch();
}
```

The use of symbolic constant makes it easier to modify a program at a later stage. The constant value needs to be changed only at one place instead of its various instances in the program.

Output

```
Profit = 20
```

Program analysis

Key Statement	Purpose
`int` *SP = 70;*	Declares and initializes the variable *SP* with an initial value of *70*
`int` *profit = SP–CP;*	Declares and initializes the variable *profit* with the resultant value of the expression, *SP–CP*

1.7 DATA TYPES

C supports a number of data types for handling different types of data such as integer, real, and string. The choice of a data type solely depends on the programming situation at hand. For instance, if sorting of numbers is to be performed then **int** data type is used for storing integers. Similarly, for printing name, a character array is used for storing a character string.

Data types in C are primarily categorized into three types:

1. **Primary data types** Represent the fundamental data types of C, which are **int, char,** float, double and void. There are several extensions possible to some of the fundamental data types, such as short **int**, unsigned **int**, long **double** and so on. Here, short, unsigned and long are used as qualifiers for manipulating the size and range of the fundamental data types.
2. **Derived data types** Represent the data types derived from the fundamental data types. Examples of derived data types include arrays, structures and pointers. These are covered in detail in Chapter 2.
3. **User-defined data types** Represent the user-specified identifiers used in place of existing data types. For example,

```
typedef float temperature;
temperature t1, t2, t3;
```

Here, temperature is a user-defined data type used to declare float type variables.

The enumerated data type is another instance of user-defined data type that allows you to declare variables that can be assigned only one of the pre-specified values. These values are enclosed within a pair of braces at the time of variable declaration. For example,

```
enum logic {HIGH, LOW};
enum logic l;
l = HIGH;
```

Table 1.6 lists the size and range of primary data types and their extensions.

Table 1.6 *Primary data types and their extensions*

Data Type	Size	Range
char or signed **char**	8	–128 to 127
unsigned **char**	8	0 to 255
int or signed **int**	16	–32768 to 32767
unsigned **int**	16	0 to 65535
short **int** or signed short **int**	8	–128 to 127
unsigned short **int**	8	0 to 255
long **int** or signed long **int**	32	–2147483648 to 2147483647
unsigned long **int**	32	0 to 4294967295
float	32	3.4E – 38 to 3.4E + 38
double	64	1.7E – 308 to 1.7E + 308
long double	80	3.4E – 4932 to 1.1E + 4932

Example 1.5 Write a C program to demonstrate the use of data types and their extensions.

Program 1.2 *Use of data types and extensions*

```
/*Program for demonstrating the use of data types and their extensions*/
#include <stdio.h>
#include <conio.h>

void main( )
{

 int num = 10000;
 int num1;
 long int num2;
 clrscr();

 num1 = 10000*10;
 num2 = 10000*10;

 printf("num = %d", num);
 printf("\nnum1 = %d", num1);
 printf("\nnum2 = %ld", num2);
 getch();
}
```

This statement will output a garbage value as a value larger than the capacity of an integer variable was allocated to num1.

Output

```
num = 10000
num1 = -31072
num2 = 100000
```

Program analysis

Key Statement	Purpose
int num1;	Declares an integer variable `num1`
long int num2;	Declares a long integer variable `num2`

1.8 INPUT AND OUTPUT OPERATIONS

Input and output operations in C are performed with the help of function calls, such as **printf** and **scanf.** These standard input and output functions are stored in the header file **stdio.h.** Thus, each program must include this header file by using the following statement:

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

Let us now explore the common input and output operations in C.

1.8.1 Input Operations

Table 1.7 lists the common input operations along with their function descriptions.

Table 1.7 *Input operations*

Input Operation	Function	Syntax and Example

Reading a character	**getchar()**	**Syntax**
		`<variable_name> = getchar();`
		Example
		`a = getchar();`

Reading formatted input data	**scanf()**	**Syntax**
		`scanf("control string", arg1, arg2,...argn);`
		Example
		`scanf("%2d %15c %lf",&num, name, &x);`

1.8.2 Output Operations

Table 1.8 lists the common output operations along with their function descriptions.

Table 1.8 *Output operations*

Output Operation	Function	Syntax and Example

Writing a character	**putchar()**	*Syntax*
		`putchar(<variable_name>);`
		Example
		`putchar(a);`

Writing formatted output data	**printf()**	*Syntax*
		`printf("control string", arg1, arg2,...argn);`
		Example
		`printf("%2d %15c %lf",num, name, x);`

Example 1.6 Write a program to demonstrate input and output operations in C.

Program 1.3 *Input and output operations*

```
/*Program for demonstrating input and output operations*/
#include <stdio.h>
#include <conio.h>

void main( )
{

int age;
float avg_marks;
 char name[10];
```

```
  clrscr();

  printf("Enter student's name: ");
  scanf("%s",&name);

  printf("Enter student's age: ");
  scanf("%d",&age);

  printf("Enter student's average marks: ");
  scanf("%f",&avg_marks);

  printf("\nThe student details are:\n");
  printf("Name: %s",name);
  printf("\nAge: %d",age);
  printf("\nAverage Marks: %.2f",avg_marks);
  getch();
}
```

Output

```
Enter student's name: Kartik
Enter student's age: 15
Enter student's average marks: 75.45

The student details are:
Name: Kartik
Age: 15
Average Marks: 75.45
```

Program analysis

Key Statement	Purpose
scanf("%s",&name);	Reads a string value from the console
scanf("%d",&age);	Reads an integer value from the console
scanf("%f",&avg_marks);	Reads a real value from the console
printf("Name: %s",name);	Displays a string value on the console
printf("\nAge: %d",age);	Displays an integer value on the console
printf("\nAverage Marks: %.2f",avg_marks);	Displays a real value on the console

1.8.3 File Input/Output Operations

C makes use of streams for performing input or output operations on files containing data. A file can be a text stream file or a binary stream file. A stream is defined as a sequence of bytes organized into lines. A line may have zero or more characters. The standard input stream is called **stdin** and the standard output stream is called **stdout**. Figure 1.2 shows the logical representation of streams.

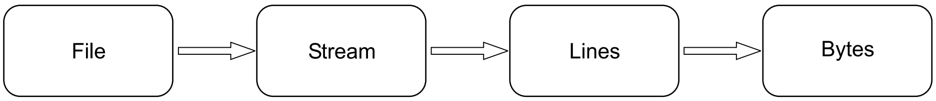


Fig. 1.2 *Logical representation of streams*

The input/output operations on files include opening, reading, writing and closing a file. To perform these operations, I/O library functions are used. The use of I/O functions establishes a connection between the physical file and a stream.

Opening a File You can open a file in read, write or append mode by using the **fopen()** function. The syntax for using the **fopen** function is shown below.

fopen(<filename>, <mode>);

C allows you to simultaneously open multiple files in a program.

Note *If you open a file in write or append mode but the file does not exist, then the system automatically creates a new file with the specified name. Also, if an existing file is opened in write mode, then its contents are overwritten with the new text.*

Reading a File You can use any of the following functions to read data from a file:

1. **fscanf** Reads formatted input from the file stream.
2. **getc** Reads an unformatted character from the file stream.
3. **fgets** Reads an unformatted character string from the file stream.

Writing to a File You can use any of the following functions to write data to a file:

1. **fprintf** Writes formatted data to a file stream.
2. **putc** Writes an unformatted character to a file stream.
3. **fputs** Writes an unformatted character string to a file stream.

Closing a File To close a file, **fclose** function is used. The file that needs to be closed is indicated with the corresponding file pointer. The syntax for using the **fclose** function is shown below.

fclose(<file_pointer>);

Example 1.7 Write a program to count the number of characters in a file.

Program 1.4 *To count the number of characters in a file*

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

void main(int argc, char *argv[])
{
 FILE *fs;
char ch;
long count=0;
clrscr();
if(argc!=2)
{
printf("Invalid number of arguments.");
exit(0);
}
fs = fopen(argv[1],"r");
if(fs==NULL)
{
printf("Source file cannot be opened.");
exit(0);
}
while(1)
{
ch=fgetc(fs);
if (ch==EOF)
break;
else
count=count+1;
}
fclose(fs);
printf("\nThe number of characters in %s is %ld",argv[1],count);
getch();
}
```

Output

```
D:\TC\BIN>countchar.exe s1.txt

The number of characters in s1.txt is 15
```

Program analysis

Key Statement	Purpose
void main(int argc, char *argv[])	Reads command line arguments
fs = fopen(argv[1],"r");	Opens the file specified by **argv[1]** in read-only mode
ch=fgetc(fs);	Reads a character from file stream **fs**
fclose(fs);	Closes the file stream **fs**

1.9 OPERATORS AND EXPRESSIONS

Operators are the symbols that are used to perform arithmetic and logical computations on operands. An operand can be an integer, floating point or character variable; or it can be a constant. Operators and operands together constitute an expression. The output of an expression depends on the data type of the operands.

The following example illustrates the difference between operators, operands, and expressions:

Example	A * B + 3
Operators	*, +
Operands	**A, B, 3**
Expression	**A * B + 3**

The various types of operators supported by C are:

1. Arithmetic
2. Assignment
3. Bitwise
4. Conditional
5. Increment and Decrement
6. Logical
7. Relational
8. Special

Depending on the number of operands required, the above operators can be classified as unary or binary operators. As the names suggests, unary operators act upon a single operand while binary operators are applied on two operands.

1.9.1 Arithmetic Operators

Table 1.9 lists the various arithmetic operators.

Table 1.9 *Arithmetic operators*

Name	Symbol	Description	Unary/Binary	Example ($a = 10$, $b = 5$)
Addition	+	To find sum of two or more operands or To represent a positive operand	Both	$a + b = 15$ or $+ a$
Subtraction	–	To find difference of two or more operands or To represent a negative operand	Both	$a – b = 5$ or $–a$
Multiplication	*	To find product of two or more operands	Binary	$a*b = 50$
Division	/	To divide two or more operands	Binary	$a/b = 2$
Modulo Division	%	To find remainder of an integer division	Binary	$a\%b = 0$

If an expression has multiple operators with same precedence, then they are executed in the order of their appearance from left to right. That means, the operator, which appears first in the expression is executed first.

1.9.2 Assignment Operators

Table 1.10 lists the various assignment operators.

Table 1.10 *Assignment operators*

Name	Symbol	Description	Unary/Binary	Example ($a = 10$, $b = 5$)
Assignment	=	To assign the value of an expression, variable or constant to a variable	Binary	$b = a$ Thus, $b = 10$
Shorthand assignment	+=	To assign the value of an expression, variable or constant to a variable and perform the specified arithmetic operation on that variable	Binary	$b += a$ This means, that the value of a i.e., 10 is added to the value of b and the result is assigned to b as its new value. Thus, $b = 15$

1.9.3 Bitwise Operators

Bitwise operators are used for manipulating the integers at bit level. The binary representation of the integer gets altered as per the type of binary operator applied.

Table 1.11 lists the various bitwise operators.

Table 1.11 *Bitwise operators*

Name	Symbol	Description	Unary/Binary	Example ($a = 10$, $b = 5$)
AND	&	To perform logical AND operation at bit level	Binary	0101 & 0011 = 0001
OR	|	To perform logical OR operation at bit level	Binary	0101 & 0011 = 0111
Exclusive OR	^	To perform logical exclusive OR operation at bit level	Binary	0101 & 0011 = 0110
Shift Left	<<	To shift the bits to the left	Unary	00010111 << 1 = 00101110
Shift Right	>>	To shift the bits to the right	Unary	00010111 >> 1 = 00001011
Complement or Not	~	To negate each bit of the binary number	Unary	~ 0001 = 1110

1.9.4 Conditional Operator

Conditional operator **(? :)** is the only ternary operator supported by C. It requires three different expressions, as shown in the syntax below.

```
<Conditional_Expression> ? <Expression1> : <Expression2>
```

If the conditional expression evaluates to TRUE, then Expression1 is evaluated, else Expression2 is evaluated.

For example, consider the following conditional expression. Assume, a =10 and b = 5:

$c = (a + b > 10)\ ?\ a : b;$

When the above statement is executed, the expression ($a + b > 10$) is evaluated first. Since this condition is true, the value of a is assigned to c.

Now, again consider the following conditional expression:

$c = (a + b > 20)\ ?\ a : b$

When the above statement is executed, the expression ($a + b > 20$) is evaluated first. Since this condition is false, the value of b is assigned to c.

1.9.5 Increment and Decrement Operators

Table 1.12 lists the increment and decrement operators.

Table 1.12 *Increment and decrement operators*

Name	Symbol	Description	Unary/Binary	Example (a = 10, b = 5)
Increment	++	To increase the value of operand by 1	Unary	$b = ++a$ $b = 11$
Decrement	– –	To decrease the value of operand by 1	Unary	$b = --a$ $b = 9$

Increment and decrement operators may be used as a prefix or a postfix in an expression. When used as prefix (example ++a or – –b), the operator first changes the variable's value before the corresponding expression is evaluated. Alternatively, when it used as postfix (for example, a++ or b– –), the operator changes the variable's value after the expression has been evaluated.

For instance, consider the expression,

$b = ++a;$

Assume $a = 5$

After the above expression is executed, the values of a and b become

$b = 6$ and $a = 6$

Now, consider the expression

$b = a++;$

Assume $a = 5$.

After the above expression is executed, the values of a and b become:

$b = 5$ and $a = 6$

Note that since the increment operator was used as a postfix, the value of variable a changed after the expression was evaluated.

1.9.6 Logical Operators

The output of an expression containing logical operators is either 0 or 1. The output of 1 signifies a true condition while 0 signifies a false condition. Table 1.13 lists the various logical operators.

Table 1.13 *Logical operators*

Name	Symbol	Description	Unary/Binary	Example ($a = 10, b = 5$)
AND	&&	To return true if the conditions are true	Binary	$a < 20$ && $b > 6$ will return 0 as both the conditions are not true
OR	||	To return true if at least one of the conditions is true	Binary	$a < 20$ || $b > 6$ will return 1 as one of the conditions is true
NOT	!	To negate the value of a relational expression	Unary	!$a < 20$ will return 0 as the negated value

Logical operators are typically used in conjunction with relational operators to form logical expressions. Relational operators are explained in the next subsection.

1.9.7 Relational Operators

The output of relational operators is either 0 or 1. The output of 1 signifies a true condition while 0 signifies a false condition. Table 1.14 lists the various relational operators.

Table 1.14 *Relational operators*

Name	Symbol	Description	Unary/Binary	Example ($a = 10, b = 5$)
Less than	<	To check if the value of one variable is less than the other variable	Binary	$a < 20$ will return 1as the condition is true
Less than or equal to	<=	To check if the value of one variable is less than or equal to the other variable	Binary	a < =20 will return 1 as the condition is true
Greater than	>	To check if the value of one variable is greater than the other variable	Binary	$a > 20$ will return 0 as the condition is false
Greater than or equal to	>=	To check if the value of one variable is greater than or equal to the other variable	Binary	a > =20 will return 0 as the condition is false
Equal to	==	To check if the value of two variables is equal	Binary	a = = b will return 0 as the condition is false
Not equal to	!=	To check if the value of two variables is not equal	Binary	a!=b, it will return 1 as the condition is true

As already explained, relational operators are used in conjunction with logical operators to form logical expressions.

1.9.8 Special Operators

Table 1.15 lists the special operators.

Table 1.15 *Special operators*

Name	Symbol	Description	Unary/Binary	Example ($a = 10$, $b = 5$)
Comma	,	To link the related expressions together	Binary	$x = (a = 10, b = 5, c = a–b)$; will assign $c = 5$ Comma operator is also used inside while and for loops
Sizeof	**sizeof();**	To retrieve the number of bytes a variable or a constant occupies in memory	Unary	x = sizeof (a); will assign the size of variable a to x
Pointer operators	&, *	To work with pointers	Unary	&a retrieves the address of variable a *a retrieves the value references by pointer variable a

1.9.9 Precedence of Operators

The precedence order of operators is the order in which operators are executed in an expression. Table 1.16 lists the precedence of operators from highest to lowest level.

Table 1.16 *Precedence of operators*

Precedence Level	Operators
1	-, ++, – –, !, sizeof()
2	*, /, %
3	+, –
4	<, <=, >, >=, =
5	==, !=
6	&&
7	||
8	,

Example 1.8 Write a program to accept two complex numbers and find their sum.

Program 1.5 *To accept two complex numbers and find their sum*

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

  void main()
  {
  struct complex
  {
       double real;
       double img;
  };

  struct complex c1, c2, c3;
  clrscr();
  printf("\n  Enter  two  Complex  Numbers  (x+iy):\n\n  Real  Part  of  First 
Number: ");
  scanf("%lf",&c1.real);
  printf("\n Imaginary Part of First Number: ");
  scanf("%lf",&c1.img);
  printf("\n Real Part of Second Number: ");
  scanf("%lf",&c2.real);
  printf("\n Imaginary Part of Second Number: ");
  scanf("%lf",&c2.img);
  c3.real=c1.real+c2.real;
  c3.img=c1.img+c2.img;
  printf("\n\n%.2lf+(%.2lf)i + %.2lf+(%.2lf)i = %.2lf+(%.2lf)i", c1.real, 
c1.img, c2.real, c2.img, c3.real, c3.img);
  getch();
  }
```

Output

```
  Enter two Complex Numbers (x+iy):

       Real Part of First Number: 22

       Imaginary Part of First Number: 4

       Real Part of Second Number: 5

       Imaginary Part of Second Number: 3

  22.00+(4.00)i + 5.00+(3.00)i = 27.00+(7.00)i
```

Program analysis

Key Statement	Purpose

struct complex
{
double real;
double img;
};
	Defines a structure named *complex*

struct complex *c*1, *c*2, *c*3;	Declares structure variables
***c*3.real=*c*1.real+*c*2.real;**	Performs arithmetic operation on structure members

Summary

- A problem is a situation presented to a computer so that its solution can be found.
- Problem solving involves applying a series of methods to a given problem so that its solution can be achieved in a systematic manner.
- An algorithm is a sequence of steps written in simple English phrases to describe the solution to a given problem.
- A flowchart can be referred as a pictorial representation of an algorithm. It uses various graphical elements to describe the flow of information and control.
- Pseudocode represents the problem solution with the help of generic syntax and normal English phrases.
- C is a programming language that combines the features of both low level and high level languages. Its key features are: structured programming approach, machine independence, extensibility, support for a variety of data types and operators.
- Constants are the entities whose values do not change during the course of program execution. Various types of constants are: integer, real, character and string.
- Variables are the entities that can take different values during the course of execution of a program.
- C supports a number of data types for handling different types of data such as integer, real, and string.
- Fundamental data types supported by C are **int**, **char**, **float**, **double** and **void**.
- C supports a number of built-in functions for performing input and output operations. These functions are **getchar()**, **putchar()**, **printf()**, **scanf()**, etc.
- Operators are the symbols that are used to perform arithmetic and logical computations on operands.
- Operators in C are classified under different categories, such as arithmetic, assignment, increment, decrement, relational, logical, etc.
- Operator precedence refers to the order in which operators are executed in an expression.

Review Questions

1.1 What is an algorithm? What are its various characteristics?
1.2 List the advantages and disadvantages of using flowcharts for problem solving.
1.3 What is a pseudocode? Explain with the help of an example.
1.4 List any three features of C programming language.
1.5 What is a constant? Explain the different types of constants with the help of example.
1.6 What is an input function? Explain with the help of an example.
1.7 Explain the difference between **getc** and **putc** functions.
1.8 What is an operator? What are its various types?

Programming Exercises

1.1 You are required to convert the value of temperature from degree Celsius to degree Fahrenheit. Apply the various problem solving techniques to arrive at the desired solution to this problem. Also, write a C program for the problem and test the program for three different set of values.
1.2 Write a C program to read three integers and then display their sum.
1.3 Write a program in C to compute the area of a circle. Use the **# define** statement to define pi as named constant.
1.4 Using the fundamental C data types, write a program to read the marks obtained by a student in three different subjects and then compute and display their average.
1.5 Write a C program to create and open a file named 'employee.dat'.
1.6 Using the modulo division operator, write a program to determine if the given number is divisible by 7 or not.
1.7 An integer variable **num** contains 2 as its value. Print the contents of **num** after a left shift operation.
1.8 Write a C program to demonstrate what impact does the postfix and prefix usage of increment operator has on the value of an expression.
1.9 Write a C program that uses any one of the special operators of C.
1.10 You are required to demonstrate the operator precedence rules of C. Choose a suitable expression and write a program that evaluates the value of the expression.

2

C RECAP – II

Chapter Outline

2.1 INTRODUCTION

In the previous chapter, we learnt the basics of problem solving techniques and the C language. In this chapter, we will focus on some of the advanced C programming constructs, such as arrays, functions, structures, and unions. But, first of all, we will begin by exploring the various control statements provided by C. Control statements allow the programmers implement the solution logic by controlling and manipulating the program flow. These statements not only allow you to manipulate the statement execution sequence but also let you repeat the execution of a statement block for certain specified number of times.

At the end of this chapter, we will learn about one of the most important and advanced C programming concepts, called *pointers*.

Note *If you are already familiar with advanced C programming constructs, then you may choose to skip this chapter.*

2.2 CONTROL STATEMENTS

C language allows you to manipulate the flow of program control by using various control statements. These control statements are categorized into the following types:

1. **Decision making statements** Decide which actions are to be performed.
2. **Looping statements** Decide how many times an action is to be performed.

One of the significant differences between decision making and looping statements is that the decision making statements direct the flow of program control from top to bottom without any iteration; while, looping statements direct the flow of program control from top to bottom, and then again to the top, in an iterative manner.

A program may contain both decision making as well as looping statements.

2.2.1 Decision Making Statements

C language supports the following decision-making statements:

1. **If**
2. **If Else**
3. **Switch**

1. **If statement** If statement executes a specific set of instructions only if the given expression is true. However, if the expression evaluates to a false value, then the set of instructions are skipped at the time of program execution. A program can contain multiple if statements.

Syntax

```
if(expression)
{
 statement 1;
 statement 2;
 .
 .
 statement n;
}
```

FLOWCHART

Figure 2.1 shows the flowchart of **if** statement:

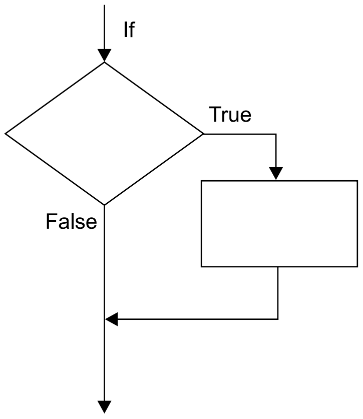


Fig. 2.1 *Flowchart of **if** statement*

Example 2.1 Using **if** statement, write a program to find out if the given number is positive or negative.

Program 2.1 *Determine whether given number is positive or negative*

```
#include<stdio.h>
#include<conio.h>
void main()
{
 int num;
 clrscr();
 printf("Enter an integer value: ");
 scanf("%d", &num);
 if(num>=0)
 printf("%d is a positive number",num);
 if(num<0)
 printf("%d is a negative number",num);
 getch();
}
```

If there is only a single statement inside the if block then the braces are not required to be added.

Output

```
Enter an integer value: -9
-9 is a negative number
```

Program analysis

Key Statement	Purpose
if(num>=0)	Uses **if** statement to check whether *num* is greater than or equal to zero
if(num<0)	Uses **if** statement to check whether *num* is less than zero

Tip *Do not add a semicolon at the end of an **if** expression, otherwise the block of statements specified in the if block will always be executed regardless of the outcome of the conditional expression statement.*

2. If else statement **If else** statement comprises two sets of instruction, one each with **if** and **else** block. If the given expression statement is true then the **if**-block instructions are executed. However, if the expression evaluates to a false value, then the **if**-block instructions are skipped and the **else**-block instructions are executed. A program can contain multiple **if else** statements.

Syntax

```
if(expression)
{
 statement 1;
 statement 2;
 .
 .
 statement n;
}
else
{
 statement 1;
 statement 2;
 .
 .
 statement n;
}
```

FLOWCHART

Figure 2.2 shows the flowchart of **if else** statement:

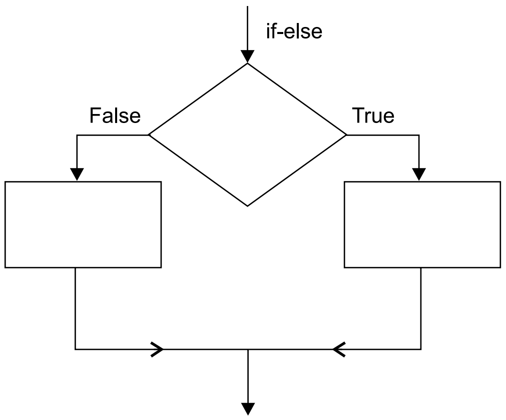


Fig. 2.2 *Flowchart of **if else** statement*

Example 2.2 Using **if else** statement, write a program to find out if the given number is positive or negative.

Program 2.2 *Determine whether given number is positive or negative*

```
#include<stdio.h>
#include<conio.h>
void main()
{
 int num;
 clrscr();
 printf("Enter an integer value: ");
 scanf("%d", &num);
 if(num>= 0)
 printf("%d is a positive number",num);
 else
 printf("%d is a negative number",num);
 getch();
}
```

If-else block

Output

```
Enter an integer value: 22
22 is a positive number
```

Program analysis

Key Statement	Purpose

if(num>= 0)
printf("%d is a positive number",num);
else
printf("%d is a negative number",num);
	if-else block is used to display the output based on the value of *num* variable

3. Switch statement Switch statement is a multi-way selection structure that matches the given expression or variable value with one of a number of integer values, and upon successful match branches to the corresponding statement block. A default value is also specified with the switch statement to take appropriate action in case there is a complete mismatch. **Switch** statement can be considered as an alternative to multiple if statements.

Syntax

```
switch (expression)
{
 case value1:
<statement block>
 break;
 case value2:
<statement block>
 break;
 case value3:
<statement block>
 break;
 .
 .
 .
 default:
<statement block>
}
```

In the above syntax:
expression is the expression to be matched.
value1, *value2*, etc., are constants, also known as *case labels*.

Flowchart

Figure 2.3 shows the flowchart of **switch** statement:

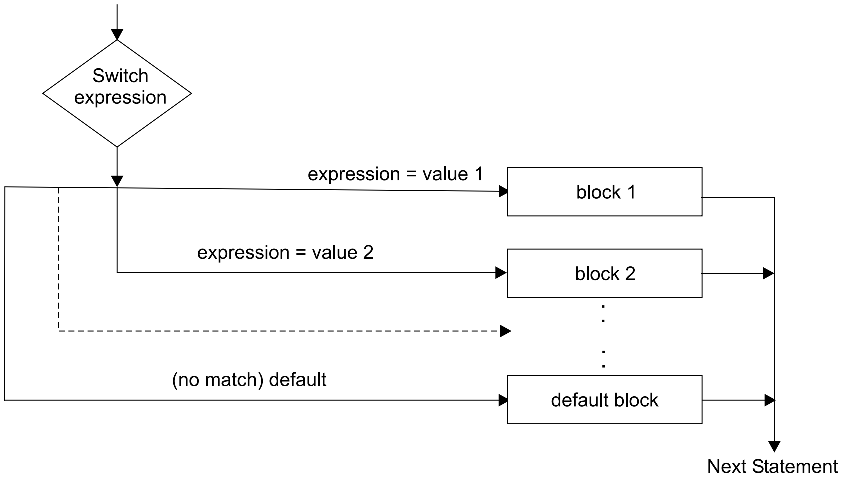


Fig. 2.3 *Flowchart of* ***switch*** *statement*

Example 2.3 Using switch statement, write a program to count the frequency of individual digits in a number.

Program 2.3 *To count the frequency of individual digits in a number*

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

void main()
{
long int num1,num2;
int temp,d1=0,d2=0,d3=0,d4=0,d5=0,d6=0,d7=0,d8=0,d9=0,d0=0;
clrscr();

printf("\nEnter a number:");
scanf("%ld",&num1);

num2=num1;
while(num1!=0)
{
temp=num1%10;
switch(temp) /*Switch statement*/
{
case 1:
    d1++; /*Counting number of Ones*/
    break;
case 2:
    d2++; /*Counting number of Twos*/
    break;
case 3:
    d3++; /*Counting number of Threes*/
    break;
case 4:
    d4++; /*Counting number of Fours*/
    break;
case 5:
    d5++; /*Counting number of Fives*/
    break;
case 0:
    d0++; /*Counting number of Zeros*/
    break;
case 6:
    d6++; /*Counting number of Sixes*/
    break;
case 7:
    d7++; /*Counting number of Sevens*/
    break;
case 8:
    d8++; /*Counting number of Eights*/
    break;
case 9:
    d9++; /*Counting number of Nines*/
    break;
default:
     ; /*Do-nothing*/
}
num1=num1/10;
}

/*Displaying the frequency of individual digits in a number*/
printf("\nThe no of 0s in %ld are %d",num2,d0);
printf("\nThe no of 1s in %ld are %d",num2,d1);
printf("\nThe no of 2s in %ld are %d",num2,d2);
printf("\nThe no of 3s in %ld are %d",num2,d3);
printf("\nThe no of 4s in %ld are %d",num2,d4);
printf("\nThe no of 5s in %ld are %d",num2,d5);
printf("\nThe no of 6s in %ld are %d",num2,d6);
printf("\nThe no of 7s in %ld are %d",num2,d7);
printf("\nThe no of 8s in %ld are %d",num2,d8);
printf("\nThe no of 9s in %ld are %d",num2,d9);

getch();
}
```

Output

```
Enter the number: 889653442

The no of 0s in 889653442 are 0
The no of 1s in 889653442 are 0
The no of 2s in 889653442 are 1
The no of 3s in 889653442 are 1
The no of 4s in 889653442 are 2
The no of 5s in 889653442 are 1
The no of 6s in 889653442 are 1
The no of 7s in 889653442 are 0
The no of 8s in 889653442 are 2
The no of 9s in 889653442 are 1
```

Program analysis

Key Statement	Purpose
while (num1!=0)	Uses the **while** statement to repetitively execute the enclosing statements until the value of *num1* is not equal to zero

switch (temp)
{
case 1:
    d1++;
    break;
	Uses the **switch** statement to select the case-block that matches the given expression

Note *The break statement is inserted with each case block so that the control exits from the switch construct immediately after meeting the true condition; otherwise, all the remaining case blocks will be sequentially validated.*

2.2.2 Looping Statements

C language supports the following looping statements:

1. **While**
2. **Do-While**
3. **For**

1. While loop is used for repetitively executing a set of instructions until the specified condition is true. As soon as the condition becomes false, the control is automatically transferred out of the loop. A **while** loop can be considered as an equivalent of a repetitive set of **if** statements.

Syntax

```
while (expression)
{
statement 1;
statement 2;
.
.
statement n;
}
```

Figure 2.4 shows the flowchart of **while** loop.

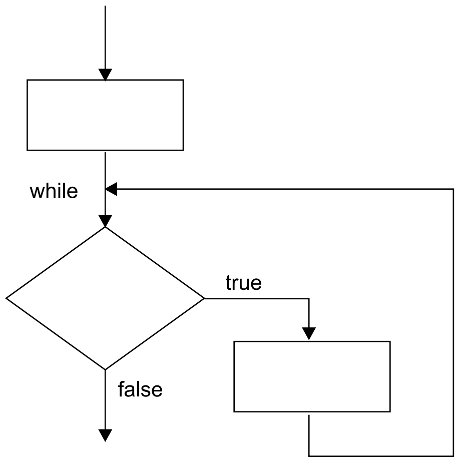


Fig. 2.4 *Flowchart of* ***while*** *loop*

Example 2.4 Using **while** loop, write a program to compute the sum of digits of a number.

Program 2.4 *Sum of digits of a number*

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

void main()
{
long num, temp;
int sum=0;
clrscr();
printf("\nEnter an integer value: ");
scanf("%ld",&num);

temp=num;
/*Calculating sum of digits*/
while(temp!=0)
{
sum = sum+temp%10;
temp=temp/10;
}

printf("\nThe sum of digits of %ld is %d",num,sum);
getch();
}
```

If there are multiple statements inside the while block then the same must be included inside braces; otherwise only the immediate statement after the while statement will be considered.

Output

```
Enter an integer value: 123456

The sum of digits of 123456 is 21
```

Program analysis

Key Statement	Purpose
while(temp!=0)	Uses the **while** statement to repetitively execute the enclosing statements until the value of *temp* is not equal to zero
temp=temp/10;	Alters the value of the control variable *temp* at each iteration of the **while** loop

2. Do-While Loop Similar to the **while** loop, the **do-while** loop is also used for repetitively executing a set of instructions until the specified condition is true. The difference between **while** loop and **do-while** loop is that in while loop the conditional expression is evaluated first and if it is true, the associated statement block is executed. However, in **do-while** loop the statement block is executed first and then the conditional expression is evaluated. If the expression evaluates to a true value, then the statement block is executed again; else the control is transferred out of the **do-while** loop. Hence, in case of **do-while** loop, the statement block is executed at least once even if the conditional expression evaluates to a false value at its very first attempt. This is in contrast to the while statement which executes the statement block if and only if the conditional expression is true.

Syntax

```
do
{
statement 1;
statement 2;
.
.
statement n;
}while(expression);
```

Figure 2.5 shows the flowchart of **do-while** loop.

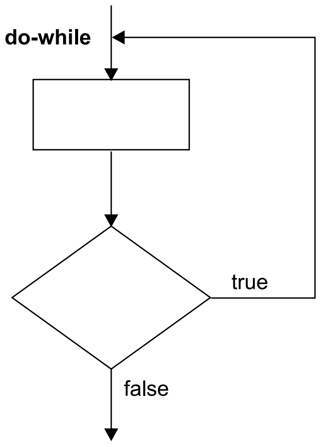


Fig. 2.5 *Flowchart of **do-while** loop*

Example 2.5 Using **do-while** loop, write a program to compute the sum of digits of a number.

Program 2.5 *Sum of digits of a number*

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

void main()
{
 long num, temp;
int sum=0;
clrscr();
printf("\nEnter an integer value: ");
scanf("%ld",&num);

 temp=num;
 /*Calculating sum of digits*/
do
{
sum = sum+temp%10;
temp=temp/10;
} while(temp!=0);

printf("\nThe sum of digits of %ld is %d",num,sum);
getch();
}
```

A compiler error will be generated if semi-colon is not inserted after while statement .

Output

```
Enter an integer value: 123456

The sum of digits of 123456 is 21
```

Program analysis

Key Statement	Purpose
do	**do** statement marks the beginning of the do-while block
} while(temp!=0);	**while** statement, specified at the end of the do-while block, contains the conditional expression for the looping construct

3. For Loop The functionality of **for** loop is exactly the same as **while** loop, however its specification is a little different. It includes the initialization statement, conditional expression and the modifier statement of the loop inside a single one-line **for** statement. At the end of each iteration, the value of the variable is updated as per the modifier expression and the condition is again evaluated. This process continues until the conditional expression holds true. The **for** loop is more compact and easy to implement in comparison to **while** loop.

Syntax

```
for (expression1; expression2; expression3)
{
 statement 1;
 statement 2;
 .
 .
 statement n;
}
```

In the above syntax:

Expression1 is the initialization statement that initializes the looping variable.

Expression2 is the conditional expression that states the looping condition.

Expression3 is the modifier expression that modifies the looping variable.

Figure 2.6 shows the flowchart of **for** loop.

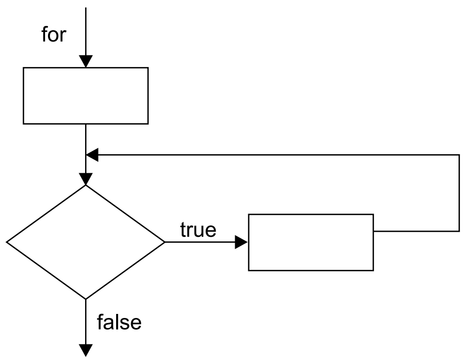


Figure 2.6 *Flowchart of **for** loop*

Example 2.6 Using **for** loop, write a program to print integers from 1 to 10.

Program 2.6 *To print integers from 1 to 10*

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

void main()
{
 int i;
 clrscr();
 printf("Integers from 1 to 10:\n");
 for(i=1;i<=10;i++)
 printf("%d\n",i)
 getch();
}
```

The initialization statement, conditional expression, and modifier expression are separated by semi-colons.

Output

```
Integers from 1 to 10
1
2
3
4
5
6
7
8
9
10
```

Program analysis

Key Statement	Purpose
for(i=1;i<=10;i++)	Uses the **for** statement to repetitively execute the enclosing statements until the given expression is true

2.3 ARRAYS

Array is a linear data structure that groups elements of similar types and stores them at contiguous memory locations. The concept of arrays can be easily related with real-life scenarios. For example, percentage marks of 50 students, salary amounts of 100 employees and names of books present in a library are all examples of arrays.

Each element in an array is assigned an index number called *array subscript*. It is used to identify the location of element in the array. In C language, the index number of the base element (first element) of an array starts with 0. Therefore, the location of the last element of an array containing n elements is always n-1. An array is referred using the array name defined at the time of array declaration. Each individual array element is referred using array name and index number.

Let us consider an array A containing four elements. Figure 2.7 shows the logical representation of array A:

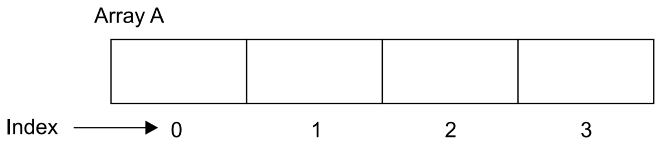


Fig. 2.7 *Logical representation of array*

Some of the typical operations performed on arrays include:

1. **Insertion** Inserts an element into the array.
2. **Deletion** Deletes an element from the array.
3. **Traversal** Accesses each element of the array.
4. **Sort** Rearranges the array elements in a specific order, such as ascending or descending.
5. **Search** Searches a key value in the array.

Note *For more information on array operations, refer to Chapter 4.*

C supports the following types of arrays:

1. **Single-dimensional arrays** Represent elements of the array in a single column. The array A shown in Figure 2.7 is an instance of single-dimensional array.
2. **Multi-dimensional arrays** Represent elements of an array in multiple columns and rows. A multi-dimensional array is also referred as *array of arrays*. Such arrays are commonly used to realize the mathematical concept of matrices.

Before an array can be used in a program, it needs to be first declared and initialized. You can initialize an array either at the time of its declaration or initialize it later in the program with the help of assignment operator.

The subsequent sections explain the declaration and initialization of single and multi-dimensional arrays along with related syntax and examples.

2.3.1 Single-Dimensional Array

Array Declaration

Syntax

```
<data-type> <array_name>[size];
```

Example

```
int A[10];
```

Array Initialization

Syntax

```
<data-type> <array_name>[size] = {element1, element2, …..., elementn};
<array_name>[index_number] = <element>;
```

Example

```
int A[3]={2,4,8};
A[0]=2;
```

Array size (in bytes)

The size of a single-dimensional array can be calculated by using the following formula:

```
array size = length of array * size of data type
```

Example

Consider the following array:

```
int A[4] = {10, 20, 30, 40};
```

Here, size of array A = 4 * 2 = 8 bytes.

Array Representation

Figure 2.8 shows the logical representation of single-dimensional array.

A[i]	A[0]	A[1]	A[2]	A[3]
	10	20	30	40
Memory address	b	b+2	b+4	b+6

Fig. 2.8 *Logical representation of single-dimensional array*

Example 2.7 Using single-dimensional array, write a program to find the average of 10 numbers.

Program 2.7 *To find average of 10 numbers*

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

void main()
```

```
{
 int a[10], sum, i;
 float ave;
 clrscr();

 printf("Enter ten integers: ");
 for(i=0;i<=9;i++)
 scanf("%d",&a[i]);

 sum=0;

 for(i=0;i<=9;i++)
 sum=sum+a[i];

 ave=1.0*sum/10;

 /*Displaying the results*/
 printf("\nAverage = %.2f",ave);
 getch();
}
```

Output

```
Enter ten integers:
1
2
3
4
5
6
7
8
9
10

Average = 5.50
```

Program analysis

Key Statement	Purpose
int a[10], sum, i;	Declares a 10-element array *a* along with other variables
scanf("%d",&a[i]);	Reads an array element value

2.3.2 Multi-Dimensional Array

Array Declaration

Syntax

```
<data-type> <array_name>[row-subscript][column-subscript];
```

Example

```
int A[2][2];
```

Array Initialization

Syntax

```
<array_name> [row-index_number][column-index_number] = <element>;
<data-type>  <array_name>[row-subscript][column-subscript]  =
{element1, element2, ….., elementn};
```

Example

```
A[1][0]=3;
int A[2][2] = {1,2,3,4};
```

The above declaration statement initializes array A in the following manner:

```
A[0][0]=1
A[0][1]=2
A[1][0]=3
A[1][1]=4 */
```

Array size (in bytes)

The size of a multi-dimensional array can be calculated by using the following formula:

```
array size = row-size * column-size * size of data type
```

Example Consider the following array:

```
int A[2][2] = {10, 20, 30, 40};
```

Here, size of array A = 2* 2 * 2 = 8 bytes.

Array Representation

Figure 2.9 shows the logical representation of multi-dimensional array.

A[i][j]	A[0][0]	A[0][1]	A[1][0]	A[1][1]
Memory	10	20	30	40
address	b	b+2	b+4	b+6

Fig. 2.9 *Logical representation of multi-dimensional array*

Example 2.8 Using multi-dimensional array, write a program to represent a 2×2 matrix.

Program 2.8 *2×2 matrix*

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

void main()
{
int i,j,M[2][2];
clrscr();
printf("Enter the elements of the 2 × 2 matrix:\n");
 for(i=0; i<2;i++)
   for(j=0; j<2;j++)
   {
printf("M[%d][%d] = ", i, j);
scanf("%d",& M[i][j]);
 }

printf("The matrix represented by the 2 × 2 2D array is:\n");
 for(i=0; i<2; i++)
 {
printf("\n\t\t ");
 for(j=0; j<2; j++)
printf("%d ", M[i][j]);
 }

getch();
}
```

The outer loop signifies matrix rows while the inner loop signifies matrix columns.

Output

```
Enter the elements of the 2 × 2 matrix:
M[0][0] = 1
M[0][1] = 2
M[1][0] = 3
M[1][1] = 4
The matrix represented by the 2 × 2 2D array is:

           1   2
           3   4
```

Program analysis

Key Statement	Purpose
int i,j,M[2][2];	Declares a two-dimensional array *M* along with other variables
scanf("%d",&M[i][j]);	Reads an element value for a two-dimensional array
printf("%d ",M[i][j]);	Displays an element value of a two-dimensional array

2.4 STRINGS

C supports the concept of strings to store a string of characters. Strings are nothing but character arrays that store the string characters at contiguous memory locations with a null character stored at the last location to indicate the end of the string. Just like an integer array stores a group of integers and a real array stores a group of floating point values, the character array stores a group of characters that collectively represent a string.

The syntax and example of string declaration and initialization is shown below:

Syntax

```
<data_type> <string_name>[size]="string_characters";
```

Example

```
char str[10]="hello";
```

C supports a number of built-in functions that make it easier to work with strings. Table 2.1 lists these string-based functions:

Table 2.1 *String functions*

Type	Function

Input/output string functions	**gets()** Receives a character string from the console.
	Example
	`char str[50];`
	`gets(str);`
	puts() Prints the string data on the console.
	Example
	`char str[50];`
	`gets(str);`
	`puts(str);`

String-handling functions	**strcat()** Joins two strings together. **strlen()** Determines the number of characters in a string. **strcpy()** Copies the characters of one string into another string. **strcmp()** Compares whether two strings are equal or not.

Note *C supports several other functions such as* **strstr**, **subchr**, **strncpy**, *etc., that help perform specific string manipulation tasks.*

Example 2.9 Write a program to print the reverse of a string.

Program 2.9 *To print the reverse of a string*

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

void main()
{
charstr[30],revstr[30]; /*Declaring character arrays for storing strings*/
inti,len;
clrscr();
printf("\nEnter a string: ");
```

Appropriate header files should be included in a program before the related functions are called.

```
gets(str);
len=strlen(str);
for(i=0;i<len;i++)
revstr[len-i-1]=str[i];
revstr[len]='\0';

printf("\n\nThe reverse of string %s is %s",str,revstr);

getch();
}
```

Output

```
Enter a string: New Delhi
The reverse of New Delhi is ihleDweN
```

Program analysis

Key Statement	Purpose
gets(str);	Reads a string *str*
len=strlen(str);	Calculates the length of the string *str* and stores it in *len* variable

Example 2.10 Write a program to concatenate two strings.

Program 2.10 *To concatenate two strings*

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

void main( )
{
char str1[40], str2[10];
clrscr();

 /*Reading strings*/
printf("Enter string 1:\n");
gets(str1);

printf("Enter string2:\n");
gets(str2);

printf("The result of concatenating strings %s and %s is: ",str1,str2);
printf("%s",strcat(str1,str2,strlen(str2)));

getch();
}
```

Here, we have called the string function directly from the printf statement.

Output

```
Enter string 1:
Taj
Enter string2:
Mahal
The result of concatenating strings Taj and Mahal is: TajMahal
```

Program analysis

Key Statement	Purpose
printf("%s",strcat(str1,str2,strlen(str2)));	Concatenates the strings *str1* and *str2* and displays the result

2.5 BUILT-IN FUNCTIONS

C supports a number of built-in functions that make the job of a programmer easier. For instance, the string handling functions such as **strcpy** and **strcmp** are all built-in functions that perform standard pre-defined string manipulation tasks. The built-in functions are stored in the related header files that need to be included in a program before the functions are actually called.

Table 2.2 gives a snapshot of the various built-in functions provided by C along with their corresponding header file names.

Table 2.2 *Built-in functions of C*

Header File	Functions
<stdio.h>	**printf(), fprintf(), scanf(), getchar(), gets(), putchar(), puts(), fopen(), fclose(), feof(), ferror()**
<stdlib.h>	**atoi(), rand(), calloc(), malloc(), free(), exit(), abs()**
<string.h>	**strcpy(), strncpy(), strcat(), strcmp(), strncmp(), strchr(), strstr()**
<time.h>	**time (), difftime(), mktime(), ctime()**
<math.h>	**sin(), cos(), tan(), sinh(), cosh(), tanh(), log(), log10(), pow(), sqrt(), fopen(), fclose(), feof(), ferror()**

Example 2.11 Using built-in C functions, write a program to compute the square root of an integer.

Program 2.11 *To compute the square root of an integer*

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

void main()
{
intnum;
```

Here, we have included the header file math.h before using a mathematical function in the program.

```
double result;
clrscr();

printf("\nEnter an integer:\n");
scanf("%d",&num);

result=sqrt(num); /*Calling the built-in function sqrt()*/
printf("\nThe square root of %d is = %.2lf",num, result);
getch();
}
```

Output

```
Enter an integer:
2

The square root of 2 is = 1.41
```

Program analysis

Key Statement	Purpose
result=sqrt(num);	Uses the built-in function *sqrt* to compute the square root of variable **num** and stores the resultant value in *result* variable

Example 2.12 Using built-in C functions, write a program to generate random numbers.

Program 2.12 *To generate random numbers*

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

void main()
{
clrscr();

srand(time(NULL)); /*The time value needs to be given as input before
generating random number*/
printf("Random number: %d", rand()); /*Calling the rand() function to
generate the random number*/
getch();
}
```

Here, we have given the current time value as an input for the random number generator function.

Output

```
Random number: 14045
```

Program analysis

Key Statement	Purpose
srand(time(NULL));	Uses the built-in function **srand()** to use the time value as an input for generating random numbers
printf("Random number: %d", rand());	Uses the built-in function **rand()** to generate a random number value and display it as output

2.6 USER-DEFINED FUNCTIONS

As the name suggests, user-defined functions are custom functions defined by the programmer to perform specific tasks. A typical approach while programming in C is to divide the problem solution into a number of subtasks and create a function to take care of each of the subtasks. These functions call each other as per the programming logic to arrive at the end solution.

To use user-defined functions in a program, you need to take care of the following elements:

1. **Function declaration** It is a declaration statement included at the beginning of a program to provide the function details to the compiler. It is also known as function prototype.
2. **Function call** It is used to invoke the function. It must supply the required parameter values along with the function call.
3. **Function definition** It is the actual code module written to implement the function.

Syntax

```
  /*Function Prototype*/
  <return_type> <function_name>( type1 parameter1, type 2 parameter2,
...., typen parametern);
  .
  .
  /*Function Call*/
  <type_variable> = <function_name>(value1, value2, ...., valuem);
  .
  .
  /*Function Definition*/
  <return_type> <function_name>(type1 parameter1, type 2 parameter2,
...., typen parametern)
  {
   statement1;
   statement2;
   .
   .
   statementn;
  }
```

Example

```
  /*Function Prototype*/
```

```
long factorial(int num);
.
.
/*Function Call*/
fact = factorial(x);
.
.
/*Function Definition*/
long fact(int n)
{
 long f;
 if( n == 0 )
 return(1);
 else
 f = n*fact(n - 1);
 return(f);
}
```

Note *A function may have void return type to indicate no return value.*

Example 2.13 Using functions, write a program to compute the area of a rectangle.

Program 2.13 *To compute the area of a rectangle*

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

  void main()
  {
  void area(float,float); /*Function prototype */
  float l, b;

  clrscr();
  printf("Enter the length and breadth of the rectangle:\n");
  scanf("%f %f",&l,&b) ;/*Reading the length and breadth values of the
rectangle*/

  area(l,b);/*Function Call*/
  getch();
  }

  void area(length, breadth)
  {
   /*Computing and displaying the area of the rectangle*/
  printf("The area of the rectangle with length %.2f and breadth %.2f is =
%.2f", length,breadth,(length*breadth));
  }
```

Output

```
Enter the length and breadth of the rectangle:
2
5
The area of the rectangle with length 2.00 and breadth 5.00 is = 10.00
```

Program analysis

Key Statement	Purpose
void area(float,float);	Declares the prototype for the user-defined function area
area(l,b);	Calls the **area()** function with a set of parameter values

Example 2.14 Using functions, write a program to implement a simple arithmetic calculator.

Program 2.14 *To implement a simple arithmetic calculator*

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

/*Function prototypes for basic arithmetic operations*/
float add(float,float);
float sub(float,float);
floatmul(float,float);
float div(float,float);

void main()
{
int choice;
float num1, num2;
clrscr();
printf("**********Simple Calc***********");/*Displaying CalC menu*/
printf("\n\nChoose a type of operation from the following: ");
printf("\n\t1. Addition");
printf(«\n\t2. Subtraction»);
printf(«\n\t3. Multiplication»);
printf(«\n\t4. Division\n»);
scanf("%d", &choice);/*Reading user's choice*/
printf("\n\nEnter the two operands: ");
scanf("%f %f", &num1, & num2);/*Reading operands on which chosen operation
is to be performed*/

 /*Using switch statement to choose the right operation*/
switch (choice)
 {
case 1:
printf("\n%.2f + %.2f = %.2lf", num1, num2, add(num1,num2)); /*Calling
the add function*/
break;
```

```
  case 2:
  printf("\n%.2f - %.2f = %.2lf", num1, num2, sub(num1,num2)); /*Calling
the sub function*/
  break;

  case 3:
  printf("\n%.2f * %.2f = %.2lf", num1, num2, mul(num1,num2)); /*Calling
the mul function*/
  break;

  case 4:
  printf("\n%.2f / %.2f = %.2lf", num1, num2, div(num1,num2)); /*Calling
the div function*/
  break;

  default:
  printf("\nIncorrect Choice!");
   }

  getch();
  }

  /*Function definitions*/
  float add(float x,float y)
  {
  return(x+y);
  }

  float sub(float x,float y)
  {
  return(x-y);
  }

  floatmul(float x,float y)
  {
  return(x*y);
  }

  float div(float x,float y)
  {
  return(x/y);
  }
```

Here, we have used the default block for showing an error message in case wrong input is entered.

Output

```
  *********Simple Calc**********

  Choose a type of operation from the following:
   1. Addition
   2. Subtraction
   3. Multiplication
   4. Division
  3


  Enter the two operands: 11
  5

  11.00 * 5.00 = 55.00
```

Program analysis

Key Statement	Purpose
• **float add(float,float);** • **float sub(float,float);** • **float mul(float,float);** • **float div(float,float);**	Declares the prototypes for the user-defined functions
• **printf("\n%.2f + %.2f = %.2lf", num1, num2, add(num1,num2));** • **printf("\n%.2f - %.2f = %.2lf", num1, num2, sub(num1,num2));** • **printf("\n%.2f * %.2f = %.2lf", num1, num2, mul(num1,num2));** • **printf("\n%.2f / %.2f = %.2lf", num1, num2, div(num1,num2));**	Calls the user-defined functions as a part of the **printf** output statement

2.7 STRUCTURES

Structures are used to group together data of different types inside a single unit. It removes one of the key limitations of arrays which can only store data elements with similar types. Thus, structures can be easily used in situations where data elements with distinct types are required to be stored as a single entity; for example, a student record containing data elements such as name, roll no, marks, etc.

Before a structure can be used in a program, it needs to be first declared and initialized. The subsequent sections explain the declaration and initialization of a structure along with the method of accessing its data members.

2.7.1 Structure Declaration

Syntax

```
struct <structure_name>
{
 <data_type> member1;
 <data_type> member2;
 .
 .
 <data_type> membern;
}var1;
struct <structure_name> var2;
```

Example

```
struct book
{
 char      title[20];
 char      author[15];
 int pages;
 float price;
}book1;
struct book book2;
```

Note *Structure variables can be instantiated either by appending the variable names at the end of the structure definition or by writing a separate declaration statement.*

2.7.2 Structure Initialization

Syntax

```
structure_variable1.member1= value;
structure_variable2.member2= value;
```

Example

```
book.title = "Data Structures";
book.pages = 450;
```

Example 2.15 Write a program that uses a simple structure to store students' details.

Program 2.15 *To store students' details*

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

void main ()
{
 int num, i=0;

 /*Structure Declaration*/
 struct student
 {
 char name[30];
 int rollno;
 int t_marks;
 };

 struct student std[10];
 clrscr();

 printf("Enter the number of students: ");
```

This statement declares an array of structures.

```
   scanf("%d",&num);

   /*Reading student details*/
   for(i=0;i<num;i++)
   {
   printf("\nEnter the details for student %d",i+1);
   printf("\n\n Name ");
   scanf("%s",std[i].name);
   printf("\n Roll No. ");
   scanf("%d",&std[i].rollno);
   printf("\n Total Marks ");
   scanf("%d",&std[i].t_marks);
   }

   /*Displaying student details*/
   printf("\n Press any key to display the student details!");
   getch();

   for(i=0;i<num;i++)
    printf("\n student %d \n Name %s \n Roll No. %d \n Total Marks
%d\n",i+1,std[i].name, std[i].rollno, std[i].t_marks);

  getch();
  }
```

Output

```
  Enter the number of students: 2

  Enter the details for student 1

   Name Rohit

   Roll No. 32

   Total Marks 375

  Enter the details for student 2

   Name Brijesh

   Roll No. 6

   Total Marks 400

   Press any key to display the student details!
   student 1
   Name Rohit
   Roll No. 32
   Total Marks 375
   student 2
   Name Brijesh
   Roll No. 6
   Total Marks 400
```

Program analysis

Key Statement	Purpose

**struct student
{
char name[30];
int rollno;
int t_marks;
};**
	Declares a structure *student* for storing student-related data

struct student std[10];	Declares an array of *student* structure
scanf("%s",std[i].name);	Reads value for a structure member
printf("\n student %d \n Name %s \n Roll No. %d \n Total Marks %d\n",i+1,std[i].name, std[i].rollno, std[i].t_marks);	Displays the structure member values

Example 2.16 Write a program to perform complex number addition using structures.

Program 2.16 *To perform complex number addition using structures*

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

void main()
{
 struct complex/*Realizing a complex number using structure*/
 {
 double real;/*Real part*/
 double img;/*Imaginary part*/
 };
 struct complex c1, c2, c3;
 clrscr();

 /*Reading the 1st complex number*/
 printf("\n Enter two Complex Numbers (x+iy):\n\n Real Part of First
 Number: ");
 scanf("%lf",&c1.real);
 printf("\n Imaginary Part of First Number: ");
 scanf("%lf",&c1.img);

 /*Reading the 2nd complex number*/
 printf("\n Real Part of Second Number: ");
```

```
   scanf("%lf",&c2.real);
   printf("\n Imaginary Part of Second Number: ");
   scanf("%lf",&c2.img);

   /*Performing complex number addition*/
   c3.real=c1.real+c2.real;
   c3.img=c1.img+c2.img;

   /*Displaying the result*/
   printf("\n\n %.2lf+(%.2lf)i + %.2lf+(%.2lf)i = %.2lf+(%.2lf)i", c1.real,
   c1.img, c2.real, c2.img, c3.real, c3.img);

   getch();
  }
```

The ***dot (.)*** *operator is used for accessing individual structure members.*

Output

```
 Enter two Complex Numbers (x+iy):

  Real Part of First Number: 1

  Imaginary Part of First Number: 2

  Real Part of Second Number: 3

  Imaginary Part of Second Number: 4


  1.00+(2.00)i + 3.00+(4.00)i = 4.00+(6.00)i
```

Program analysis

Key Statement	Purpose

**struct complex
{
double real;
double img;
};**
	Declares a structure *complex* for storing complex numbers

struct complex c1, c2, c3;	Declares variables of the *complex* structure
scanf("%lf",&c1.real);	Reads real value for the *complex* structure
scanf("%lf",&c1.img);	Reads imaginary value for the *complex* structure
printf("\n\n %.2lf+(%.2lf)i + %.2lf+(%.2lf)i = %.2lf+(%.2lf)i", c1.real, c1.img, c2.real, c2.img, c3.real, c3.img);	Displays the complex number represented by the *complex* structure

2.8 UNIONS

Like structures, unions also allow us to group together dissimilar type elements inside a single unit. But there is a significant difference between structures and unions in the way they are implemented in the system. The size of a structure is equal to the sum of the sizes of its constituent members. In contrast, the size of a union is equal to the size of its largest sized element. This is because unions allow only one member to be utilized at any given point of time. That is, union members can only be manipulated exclusive of each other.

Syntax

```
union <union_name>
{
 type1 var1;
 type2 var2;
 .
 .
 typen varn;
};
```

Example

```
union result
{
int       marks;
char      grade;
float     percent;
};
```

Unions are particularly useful in situations where it is not required to simultaneously access the data members. In such situations, unions prove to be memory efficient in comparison to structures.

Example 2.17 Write a program to demonstrate the use of unions.

Program 2.17 *Use of unions*

```
/*Program for demonstrating the use of unions*/
#include <stdio.h>
#include <conio.h>

void main ()
{
 /*Union Declaration*/
 union student
 {
 int roll_no;
 char result;
 }st1,st2;
 clrscr();

 /*Initializing union variables*/
 st1.roll_no=20;
```

```
   st2.result='P';

   /*Accessing and printing the values correctly*/
   printf("\nRoll NO: %d",st1.roll_no);
   printf("\nResult: %c",st2.result);

   printf("\n\n");

   /*Accessing and printing the values incorrectly*/
   printf("\nRoll NO: %d",st2.roll_no);
   printf("\nResult: %c",st1.result);

   getch();
  }
```

Output

```
Roll NO: 20
Result: P


Roll NO: 12880
Result: ¶
```

Program analysis

Key Statement	Purpose

**union student
{
int roll_no;
char result;
}st1,st2;**
	Defines a union named *student* and declares its variables

**st1.roll_no=20;
st2.result='P';**
	Initializes the union members

**printf("\nRoll NO: %d",st1.roll_no);
printf("\nResult: %c",st2.result);**
	Displays the values of the union members

2.8.1 Structures vs. Unions

Table 2.3 lists the key differences between structures and unions.

Table 2.3 *Structures vs. Unions*

Structures	Unions
It is defined with 'struct' keyword.	It is defined with 'union' keyword.
All members of a structure can be utilized simultaneously.	Only one member of a union can be utilized at any given point of time.

Structures	Unions
The size of a structure is equal to the sum of the sizes of its members.	The size of a union is equal to the size of its largest member.
Structure members are stored at discrete locations in memory.	All the union members share common memory space.
Structures are not considered as memory efficient in comparison to unions.	Unions are considered as memory efficient in situations where the members are not required to be accessed simultaneously.

2.9 POINTERS

Pointer is a derived data type that stores memory addresses as its value. It points or indicates the location where another variable is stored. Pointers are particularly used for dynamic memory management. Figure 2.10 shows the logical representation of a pointer:

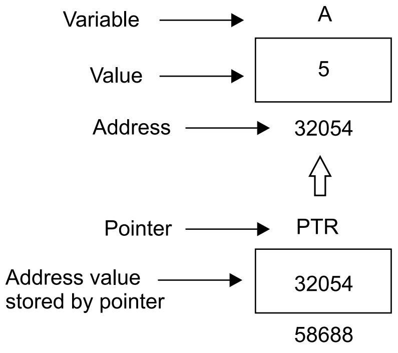


Fig. 2.10 *Logical representation of pointer*

2.9.1 Pointer Declaration and Initialization

A pointer is declared with the help of * operator and initialized with the help of & operator, as shown in the following code snippet:

```
int num=10;
int *ptr; /Pointer declaration*/
ptr=&num; /Pointer initialization*/
```

In the above code, the address of num variable is allocated to pointer variable *ptr*. Whenever * is preceded with a variable name, it refers to the value stored at the location being pointed by the variable. On the other hand, whenever & is preceded with a variable name, it refers to the memory address of the variable.

Like any other variable, pointer variables can also be used in expressions to form pointer expressions. This is depicted below:

```
a = *b +*c;
```

```
 z = z + *y;
 *c=*ptr+5;
```

Example 2.18 Using the concept of pointers, write a program to print the address and value of a variable.

Program 2.18 *To print address and value of a variable*

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

void main()
{

 int a;
 int *ptr; /*Declaring pointer variable*/
 clrscr();

 a=50;
 ptr=&a; /*Pointer allocation*/

 printf("address of a = %u\tvalue of a = %d\n",ptr,*ptr);
 printf("address of ptr = %u\tvalue of ptr = %u",&ptr,ptr);

 getch();
}
```

%u specifier is used for printing address values.

Output

```
address of a = 65524 value of a = 50
address of ptr = 65522 value of ptr = 65524
```

Program analysis

Key Statement	Purpose
int *ptr;	Declares a pointer variable *ptr*
ptr=&a;	Assigns an address value to the *ptr* variable

Summary

- C language allows you to manipulate the flow of program control by using control statements.
- Control statements are of two types: decision making statements and looping statements.
- Decision making statements: **if**, **if else**, and **switch**.
- Looping statements: **while**, **do-while**, and **for**.
- Array is a linear data structure that groups elements of similar types and stores them at contiguous memory locations.
- Common array operations are insertion, deletion, traversal, searching, and sorting.
- Arrays are of two types: single-dimensional and multi-dimensional.
- String is a character array that stores string characters at contiguous memory locations.

- C supports a number of built-in functions that can be used by including the appropriate header file in the program.
- User-defined functions are custom functions defined by the programmer to perform specific tasks.
- A structure groups together data of different types inside a single unit.
- A structure is used in situations where data elements with distinct types are required to be stored as a single entity.
- The size of a structure is equal to the sum of the sizes of its constituent members.
- The size of a union is equal to the size of its largest sized element.
- Pointer is a derived data type that stores memory addresses as its value. It is particularly used for dynamic memory management.

Review Questions

2.1 Explain the difference between decision-making and looping statements with the help of an example.
2.2 Write the syntax of switch statement.
2.3 What is the main difference between while and do-while statements?
2.4 What is an array? What are the various types of operations performed on an array?
2.5 What is a multi-dimensional array? Explain with the help of an example.
2.6 Explain any two string-handling functions with the help of an example.
2.7 What is a user-defined function? Explain with the help of an example.
2.8 Write the syntax for using a structure in a program.
2.9 What is the difference between structure and union?
2.10 What is a pointer? Explain with the help of an example.

Programming Exercises

2.1 Write a program to find the largest among three numbers.
2.2 Write a C program to find the factorial of a number using while loop.
2.3 Using for loop, write a program to generate the following pyramid structure:

```
              0
            1 0 1
          2 1 0 1 2
        3 2 1 0 1 2 3
      4 3 2 1 0 1 2 3 4
    5 4 3 2 1 0 1 2 3 4 5
  6 5 4 3 2 1 0 1 2 3 4 5 6
7 6 5 4 3 2 1 0 1 2 3 4 5 6 7
```

2.4 Using for loop, write a program to generate the following pyramid structure:

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

2.5 Write a program to compute the sum of elements of an integer array.
2.6 Using multi-dimensional array, write a program to realize a 3×3 matrix.
2.7 Write a program to compare two strings and find out if they are same or not.
2.8 Write a function 'fact' that computes and returns the factorial of the number passed as parameter.
2.9 Create an employee structure in C to store the details of five employees.
2.10 Create a five-element array and print the address of each of its elements.

3

INTRODUCTION TO ALGORITHM AND DATA STRUCTURES

Chapter Outline

3.1 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.

3.2 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 3.1 shows a simple illustration of how algorithms are used for solving computational problems.

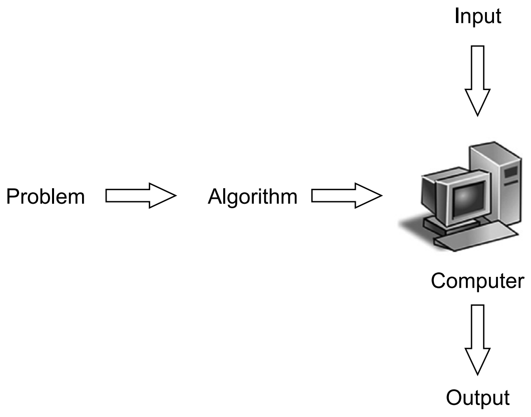


Fig. 3.1 *Use of algorithms for solving computational problems*

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

3.2.1 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.

3.2.2 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 ith element of the array A.

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

Example 3.1 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 3.2 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 3.3 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
```

3.2.3 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.

3.3 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

3.3.1 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

$f(n) \leq Cg(n)$ and $n > n_0$

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

Figure 3.2 shows the graphical representation of big-oh notation.

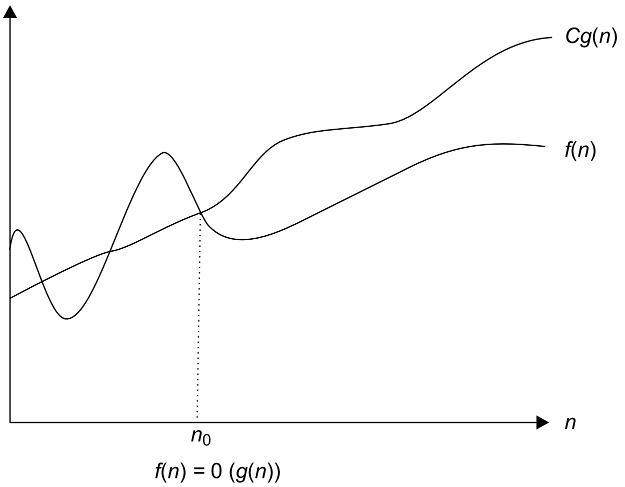


Fig. 3.2 *Graphical representation of big-oh notation*

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

1. O(1) → Constant
2. O(n) → Linear
3. O(n^2) → Quadratic

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

Example 3.4 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 3.5 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 $n = 1$,
$f(n) = 2(1) + 2$
$= 4$
$g(n) = (1)^2$
$= 1$
i.e., $f(n) > g(n)$
For $n = 2$,
$f(n) = 2(2) + 2$
$= 6$
$g(n) = (2)^2$
$= 4$
i.e., $f(n) > g(n)$
For $n = 3$,
$f(n) = 2(3) + 2$
$= 8$
$g(n) = (3)^2$
$= 9$
i.e., $f(n) < g(n)$
Therefore, $f(n) \leq Cg(n)$ is true if $n > 2$.

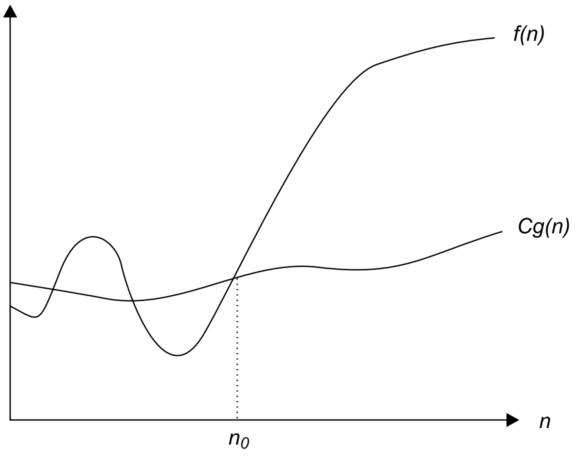
3.3.2 Omega Notation

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 'Ω'. 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 3.3 shows the graphical representation of omega notation.



f(n) = o (g(n))

Fig. 3.3 *Graphical representation of Omega notation*

Example 3.6 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 $n = 0$,

$f(n) = 2(0)^2 + 4$

$= 4$

$g(n) = 6(0)$

$= 0$

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

For $n = 1$,

$f(n) = 2(1)^2 + 4$

$= 2 + 4$

$= 6$

$g(n) = 6(1)$

$= 6$

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 3.7 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 $n > 1$.

3.3.3 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 ‘θ’. 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 3.4 shows the graphical representation of theta notation.

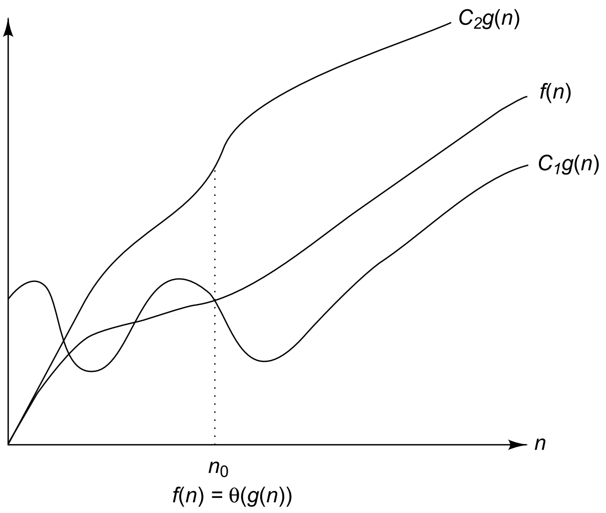


Fig. 3.4 *Graphical representation of Theta notation*

Example 3.8 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, $f(n) = 2n + 8 = \theta(n)$

3.4 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.

3.4.1 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.

3.5 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 3.5 shows the classification of different types of data structures.

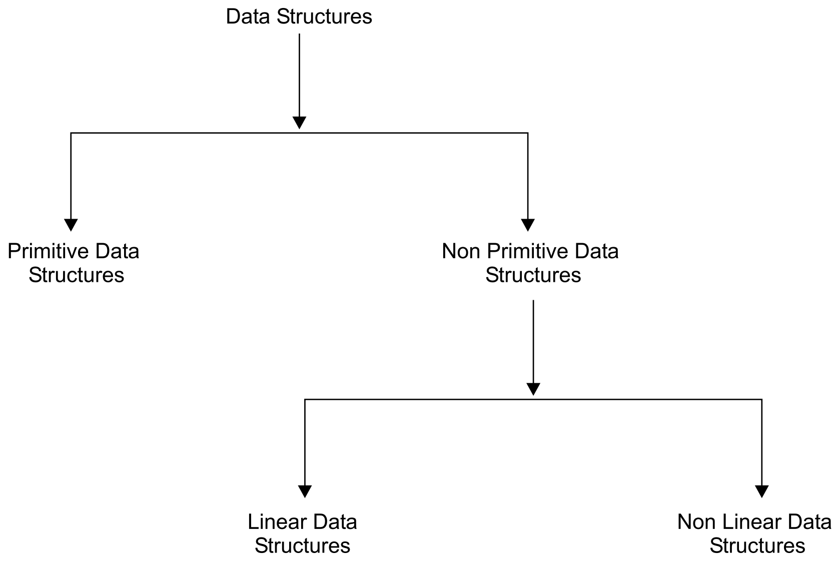

Data Structures
Primitive Data Structures
Non Primitive Data Structures
Linear Data Structures
Non Linear Data Structures

Fig. 3.5 *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.

3.5.1 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 6th element of the array 'list'.

Figure 3.6 shows the logical representation of arrays.

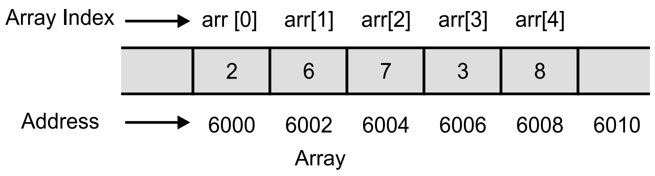


Fig. 3.6 *Logical representation of arrays*

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.*

3.5.2 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 3.7 shows the logical representation of a linked list.

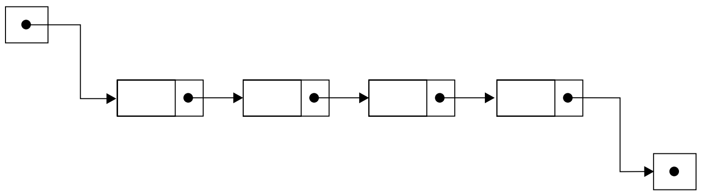


Fig. 3.7 *Logical representation of linked lists*

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.*

3.5.3 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 3.8 shows the logical representation of a stack.

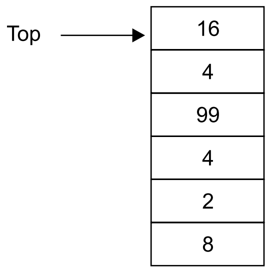


Fig. 3.8 *Logical representation of stacks*

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.*

3.5.4 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 3.9 shows the logical representation of a queue.

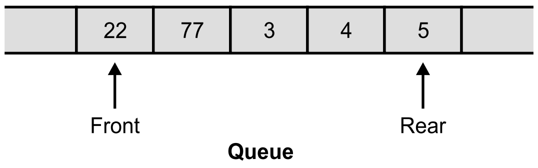


Fig. 3.9 *Logical representation of queues*

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.*

3.5.5 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 3.10 shows the representation of a tree.

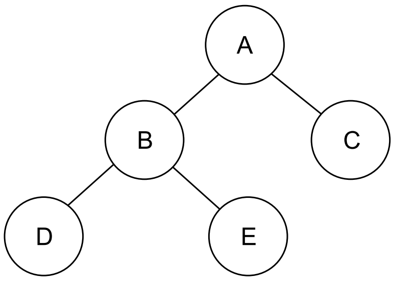


Fig. 3.10 *Tree*

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

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

3.5.6 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 3.11 shows the representation of a graph.

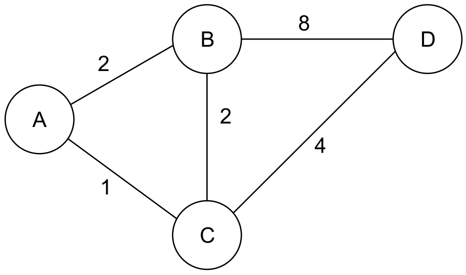


Fig. 3.11 *Graph*

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.*

3.6 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.

3.6.1 Data Structure Efficiency

Table 3.1 lists the efficiency of various data structure types for different operations.

Table 3.1 *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(logn)	O(logn)	O(logn)

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

3.1 Which of the following is not true for algorithms?

(a) It is a set of instructions that defines the solution for a given problem.

(b) 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.

3.2 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

3.3 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

3.4 ____________ 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

3.5 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

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

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

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

3.9 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

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

Review Questions

3.1 What is an algorithm? Explain with the help of an example.
3.2 List the characteristics of an algorithm.
3.3 How are algorithms represented? Explain with the help of an example.
3.4 What is algorithm complexity?
3.5 What is an asymptotic notation? Explain the various types of asymptotic notations.
3.6 What is a data structure? What are its various characteristics?
3.7 What are the different types of data structures?

3.8 Explain any two non-primitive data structures.
3.9 What is a tree? Why is it used?
3.10 List the typical operations associated with derived data structure types.

Answers to Multiple-Choice Questions

3.1 (c)	3.2 (a)	3.3 (b)	3.4 (c)	3.5 (a)
3.6 (a) and (c)	3.7 (d)	3.8 (a)	3.9 (d)	3.10 (d)

4

ARRAYS

Chapter Outline

4.1 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.

4.2 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 4.1.

Table 4.1 *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 3×3 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.

4.3 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 4.1 shows the representation of array **arr** in memory:

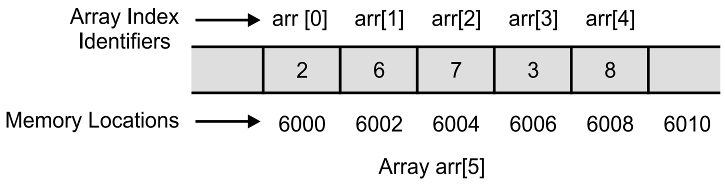


Fig. 4.1 *Array representation in memory*

As shown in Fig. 4.1, 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

```
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

```
Address of arr[2] = 6000 + 2 * 2
                  = 6000 + 4
                  = 6004
```

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

4.4 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 4.1 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
```

Mind Jog

What is 'array index out of bound'?
It is a runtime error that is encountered when a program tries to reference as address location outside of the defined array limits.

Program

Example 4.2 Write a C program to traverse each element of an array and print its value on the console.

Program 4.1 performs array traversal and prints the array elements as output. It uses the algorithm depicted in Example 4.1.

Program 4.1 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
void traverse(int*, int);	Declares the prototype for the *traverse()* function for traversing an array
traverse(arr,N);	Calls the *traverse()* function for traversing the array *arr* containing *N* elements
for(i=0;i<size;i++)	Uses the **for** 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.*

4.5 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.

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.

4.5.1 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 4.2 depicts the insertion of an element into an array.

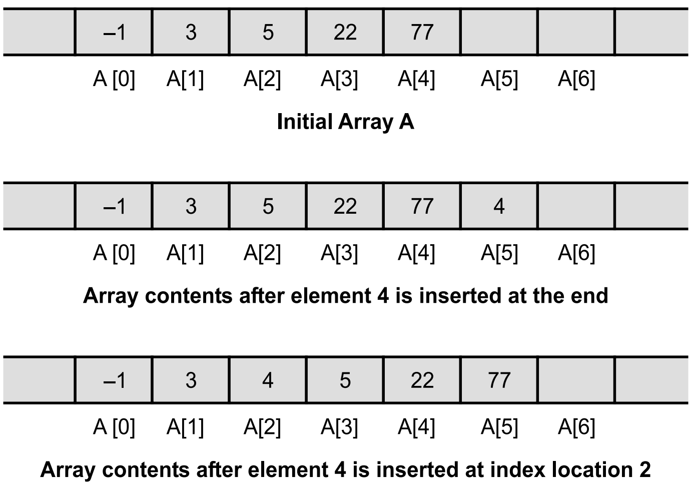


Fig. 4.2 *Array insertion*

Algorithm

Example 4.3 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 4.4 Write a C program to perform array insertion.

Program 4.2 performs array insertion and prints the updated array elements as output. It uses the algorithm depicted in Example 4.3.

Program 4.2 *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
for(i=N;i>=k;i—) array[i+1]=array[i];	Shuffles the array elements to the right to create space for inserting a new element.
array[k] = P;	Inserts a new element at the point of insertion.
N = N + 1;	Increments the total number of array elements by 1.

4.5.2 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 4.3 depicts the deletion of an element from an array.

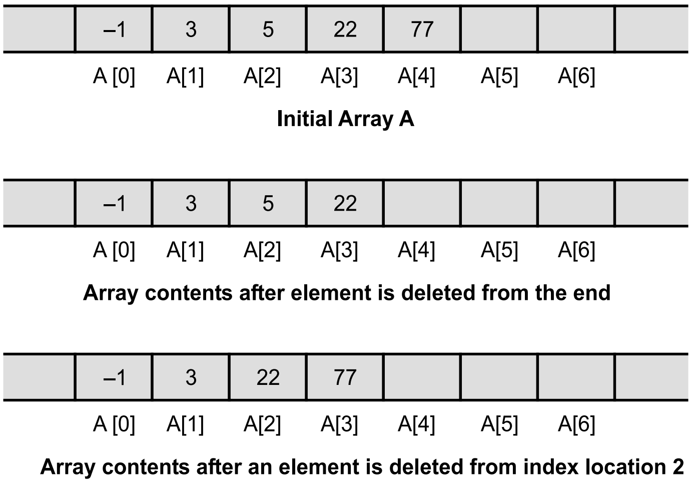


Fig. 4.3 *Array deletion*

Algorithm

Example 4.5 Write an algorithm to perform array deletion.
The following algorithm deletes the element at index location **k** in the array **A**[**N**], where **k**<=**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 4.6 Write a C program to perform array deletion.

Program 4.3 performs array deletion and prints the updated array elements as output. It uses the algorithm depicted in Example 4.5.

Program 4.3 *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
D = array[k];	Retrieves the element value that is to be deleted.
for(i=k;i<=N-2;i++) **array[i]=array[i+1];**	Shuffles the array elements to the left to fill the vacant space created after deleting the array element.
N = N – 1;	Decrements 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 the moved from their current positions.*

4.6 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.

4.6.1 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 4.7 Write a C program to sort an array of five elements.

Program 4.4 implements bubble sorting technique to sort an array of five elements.

Program 4.4 *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])
```

```
    {
     /*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
for(i=5;i>1;i—)	Uses *for* loop to control the number of passes of bubble sort algorithm
for(j=0;j<i-1;j++)	Uses *for* loop to compare the array elements in each pass of bubble sort algorithm
temp = array[j+1];	
array[j+1] = array[j];	
array[j] = temp;	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.*

4.6.2 Searching

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.

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 4.8 Write a C program to perform linear search on an array.

Program 4.5 applies linear searching technique on an array of five elements.

Program 4.5 *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*/
```

```
   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
scanf("%d",&k);	Reads the key value that needs to be searched in the array.
if(k==array[i])	Compares the key value with each array element.
break;	Takes the program control out of the **for** 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.*

4.7 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 gird-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 4.4 illustrates these representations:

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*

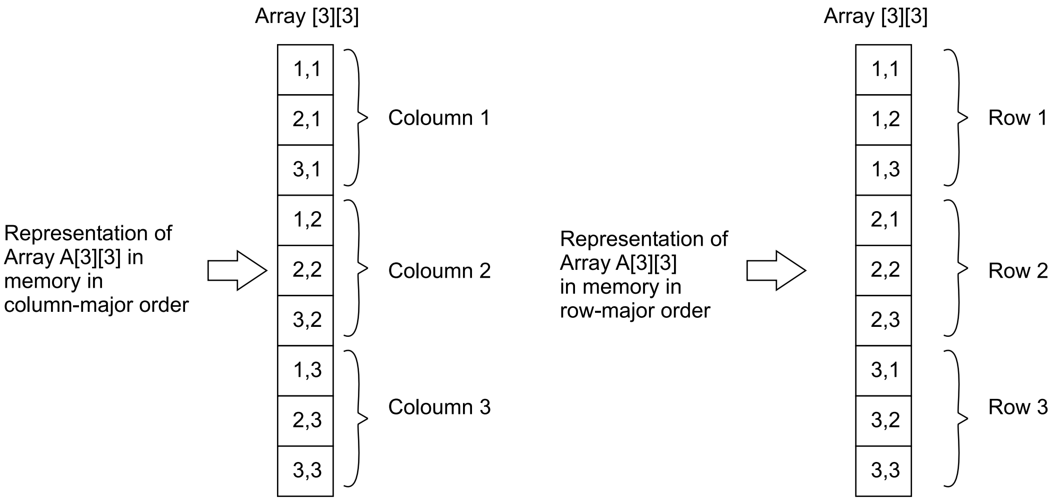

Array [3][3]
1,1
2,1
3,1
1,2
2,2
3,2
1,3
2,3
3,3
Coloumn 1
Coloumn 2
Coloumn 3
Representation of Array A[3][3] in memory in column-major order
Array [3][3]
1,1
1,2
1,3
2,1
2,2
2,3
3,1
3,2
3,3
Row 1
Row 2
Row 3
Representation of Array A[3][3] in memory in row-major order

Fig. 4.4 *Representation of two-dimensional array in memory*

As shown in Fig. 4.4, 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.

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

`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:

`Address of A[i,j] = B + W (m (j – LBC) + (i – LBR))`

Here, m represents the number of rows.

Example 4.9 A 10 × 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.

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.

Solution (a) Row major order

Address of `A[i,j] = B + W (n (i – LBR) + (j – LBC))`
Address of `A[4,7] = 200 + 2 (12 (4 – 1) + (7 – 1))`
= 200 + 2 (42)
= 284

(b) Column major order

Address of `A[i,j] = B + W (m (j – LBC) + (i – LBR))`
Address of `A[4,7] = 200 + 2 (10 (7 – 1) + (4 – 1))`
= 200 + 2 (63)
= 326

4.8 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 4.5 depicts the realization of a matrix through a two-dimensional array:

Mind Jog

What is a square matrix?
It is the matrix that has equal number of rows and columns.

$$M\,[3][4] \longrightarrow \left\{\begin{matrix} 0,0 & 0,1 & 0,2 & 0,3 \\ 1,0 & 1,1 & 1,2 & 1,3 \\ 2,0 & 2,1 & 2,2 & 2,3 \end{matrix}\right\}$$

Fig. 4.5 *Matrix represented by two-dimensional array*

Figure 4.5 shows the subscript values for each of the elements of the matrix M[3][4].

Program

Example 4.10 Write a C program to realize a 3×3 matrix.

Program 4.6 realizes a 3 × 3 matrix using a two-dimensional array.

Program 4.6 *3×3 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 × 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 × 3 2D array is:\n");
 for(i=0;i<3;i++)
 {
  printf("\n\t\t    ");
  for(j=0;j<3;j++)
   printf("%d    ",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 × 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 × 3 2D array is:

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

Program analysis

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

4.9 MATRIX OPERATIONS

The various operations associated with matrices are:

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

4.9.1 Addition

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

```
                        a1     a2     a3
If matrix A  =          a4     a5     a6
                        a7     a8     a9

                        b1     b2     b3
And, matrix B =         b4     b5     b6
                        b7     b8     b9

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

Program

Example 4.11 Write a C program to perform addition on two 3×3 matrices.

Program 4.7 *Adding on two 3×3 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 × 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 × 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 × 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 × 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*

4.9.2 Subtraction

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

```
                          a1     a2     a3
If matrix A     =         a4     a5     a6
                          a7     a8     a9



                          b1     b2     b3
And, matrix B   =         b4     b5     b6
                          b7     b8     b9

                          a1-b1    a2-b2    a3-b3
Then, A - B     =         a4-b4    a5-b5    a6-b6
                          a7-b7    a8-b8    a9-b9
```

A C program to perform matrix subtraction will be same as matrix addition (see Example 4.11). We just need to replace the +sign with a –sign.

4.9.3 Multiplication

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

```
                          a1     a2     a3
If matrix A  =            a4     a5     a6
                          a7     a8     a9



                          b1     b2     b3
And, matrix B =           b4     b5     b6
                          b7     b8     b9

              a1b1+a2b4+a3b7    a1b2+a2b5+a3b8    a1b3+a2b6+a3b9
Then, A × B = a4b1+a5b4+a6b7    a4b2+a5b5+a6b8    a4b3+a5b6+a6b9
              a7b1+a8b4+a9b7    a7b2+a8b5+a9b8    a7b3+a8b6+a9b9
```

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 × N matrix is multiplied with a N × P matrix, then the resultant matrix would be a M × P matrix.

Program

Example 4.12 Write a C program to perform multiplication on two 3×3 matrices.

Program 4.8 *Multiplying on two 3 × 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 × 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 × 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 × 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*

4.9.4 Transpose

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

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

Program

Example 4.13 Write a C program to transpose a given 3 × 3 matrix.

Program 4.9 *C program to transpose 3 × 3 matrix*

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

void main()
{
 int i,j,A[3][3],T[3][3];
 clrscr();
 printf("Enter a 3 × 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 × 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 4.1 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 4.2 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 4.3 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 4.4 Consider the following array of integers::

35 54 12 18 23 15 45 38

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

Solution Address of arr[3] = 3000 + 2 * 3
= 3000 + 6
= 3006

Problem 4.5 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.

Solution Address of A[i,j] = B + W (n (i – LBR) + (j – LBC))
Address of A[3,5] = 3000 + 2 (10 (3 – 1) + (5 – 1))
= 3000 + 2 (24)
= 3048

Problem 4.6 Solve Problem 5 in case of column order implementation.

Solution Address of A[i,j] = B + W (m (j – LBC) + (i – LBR))
Address of A[3,5] = 3000 + 2 (5 (3 – 1) + (5 – 1))
= 3000 + 2 (14)
= 3028

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

4.1 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.

4.2 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

4.3 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

4.4 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

4.5 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)

4.6 *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

4.7 A multi-dimensional array A[3][7] possesses how many number of elements?
(a) 10 (b) 21
(c) 17 (d) None of the above

4.8 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

4.1 What is an array? What are its various types?
4.2 Explain the representation of a one-dimensional array in memory with the help of an illustration.
4.3 What is array traversal? Why is it used?
4.4 What are the typical operations associated with arrays? Explain.
4.5 Write an algorithm for deleting an element at index location k in the array A[N].
4.6 What is the difference between sorting and searching?
4.7 Explain the representation of a two-dimensional array in memory with the help of an illustration.
4.8 Explain with the help of an illustration how a 2×2 matrix is stored in memory using column major order representation.
4.9 What is matrix multiplication? Explain with the help of an example.
4.10 What is the significance of transposing a matrix?

Programming Exercises

4.1 Write a C program to find the smallest element in an integer array.
4.2 Write a C program to read a value and insert it at the middle of an integer array.
4.3 Write a C program to sort an array of 10 integers.
4.4 Write a C program to demonstrate searching on an array of ten integers.
4.5 Write a C program to show how matrices are realized using two-dimensional arrays.
4.6 Write a C program to perform matrix subtraction.
4.7 Write a C program to perform transpose of a matrix.

Answers to Multiple-Choice Questions

4.1 (c)	4.2 (c)	4.3 (a)	4.4 (b)	4.5 (d)
4.6 (a)	4.7 (b)	4.8 (c)		

5

LINKED LISTS

Chapter Outline

5.1 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.

5.2 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.

5.2.1 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 5.1 depicts the logical representation of a linked list.

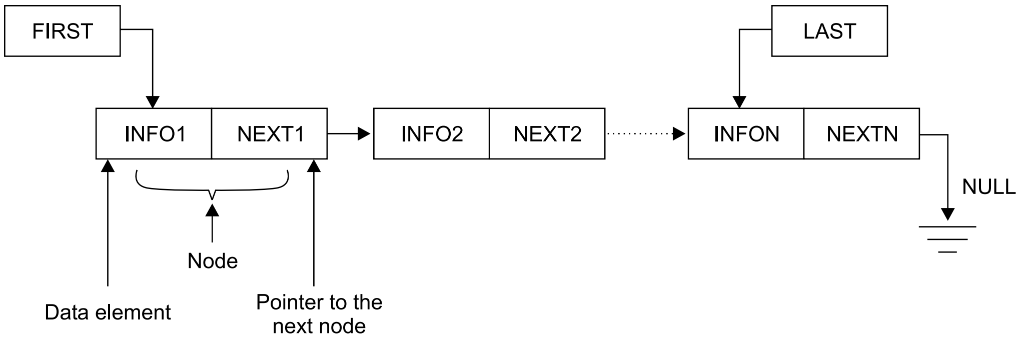


Fig. 5.1 *Logical representation of a linked list*

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

5.2.2 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.

5.2.3 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.

5.3 LINKED LIST IMPLEMENTATION

The implementation of a linked list involves two tasks:

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

5.3.1 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.

5.3.2 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 5.2 (a) and (b) show the insertion of a new element between two existing elements of a linked list.

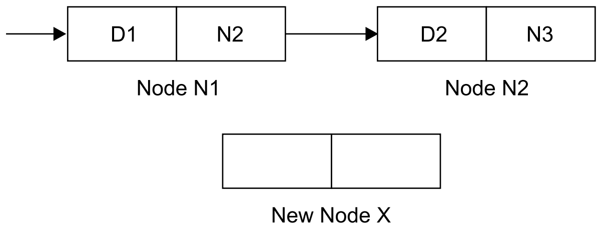


Fig 5.2 (a) *Creating a new element*

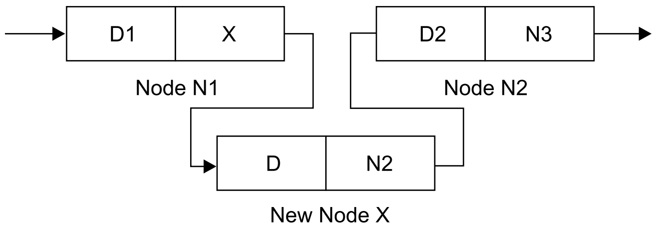


Fig 5.2 (b) *Inserting the newly created element*

Example 5.1 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 5.3 (a) and (b) show the deletion of an element that is present between two existing elements of a linked list.

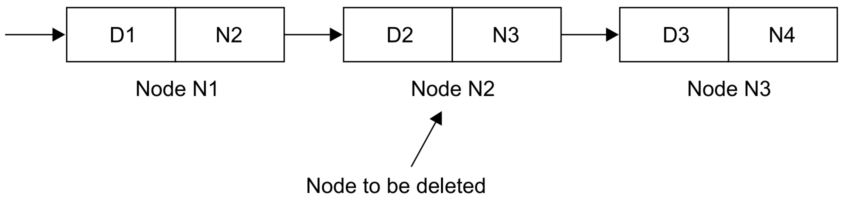


Fig 5.3 (a) *Identifying the node to be deleted*

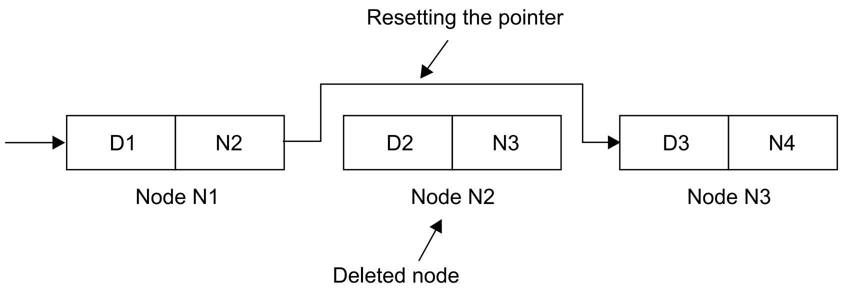


Fig 5.3 (b) *Deleting the node*

Example 5.2 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 5.3 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 5.4 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
```

5.3.3 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 5.5 Write a program to implement a linked list and perform its common operations.

Program 5.1 implements a linked list in C. It uses the insert (Example 5.1), delete (Example 5.2), search (Example 5.3) and print (Example 5.4) algorithms for realizing the common linked list operations.

Program 5.1 *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 successfuly 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.

```
 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

void insert(int);
int delete(int);
void print(void);
struct node *search (int);
	Declares the prototypes for the functions that perform linked list operations

while(1)	Initiates an infinite loop for displaying a menu of options; the loop terminates only when *exit()* function is called from the enclosing statement block
switch(choice)	Uses *switch* statement to select an appropriate *case* block as per user's choice
insert(num1);	Calls the *insert()* function for inserting an element into the linked list
num2=delete(num1);	Calls the *delete()* function for deleting an element from the linked list
location=search(num1);	Calls the *search()* function for searching an element in the linked list
print();	Calls the *print()* function for printing the linked list elements
default:	Refers to the *default* instruction block which is executed when the user enters an incorrect choice

Key Statement	Purpose
PTR->INFO = value;	Stores a *value* in the *INFO* part of the linked list node
FIRST = LAST = PTR;	Initializes the *FIRST* and *LAST* pointers of the linked list
LAST->NEXT = PTR;	Stores an address value in the *NEXT* part of the linked list node

5.4 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.

5.5 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 5.4 depicts the logical representation of a circular linked list.

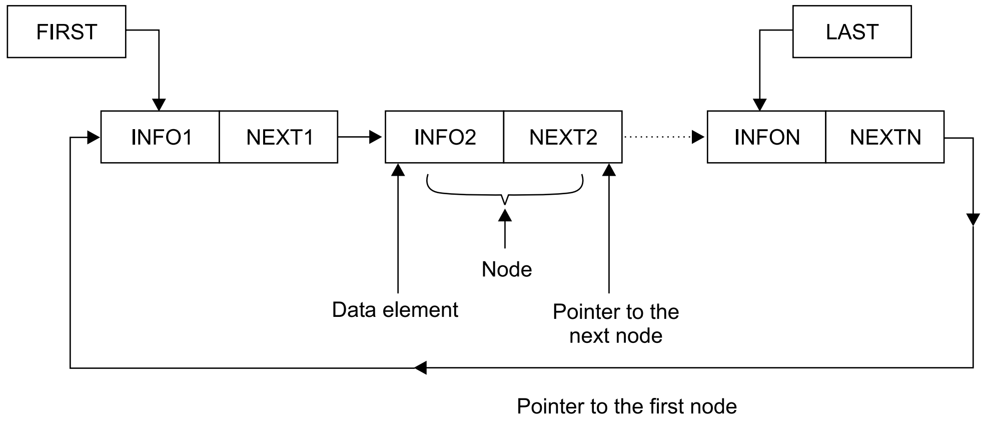


Fig. 5.4 *Logical representation of a circular linked list*

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.

5.5.1 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 5.6 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 5.7 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.

5.5.2 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 5.8 Write a program to implement a circular linked list and perform its common operations.

Program 5.2 implements a circular linked list in C. It uses the insert (Example 5.6) and delete (Example 5.7) algorithms for realizing the insert and delete operations on the circular linked list. For performing the search and print operations, the same algorithms (Example 5.3 and Example 5.4) have been used that were earlier used for implementing a singly linked list.

Program 5.2 *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;
```

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

```
 }
 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;
```

```
   }
   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


Select an option

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

Enter your choice: 5
```

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

Program analysis

Key Statement	Purpose

struct cl_node *FIRST = NULL;
struct cl_node *LAST = NULL;
	Declares pointers to the first and last nodes of the circular linked list

void insert(int);
int delete(int);
void print(void);
struct cl_node *search (int);
	Declares the prototypes for the functions that perform operations on the circular linked list

insert(num1);	Calls the *insert()* function for inserting an element into the circular linked list
num2=delete(num1);	Calls the *delete()* function for deleting an element from the circular linked list
location=search(num1);	Calls the *search()* function for searching an element in the circular linked list
print();	Calls the *print()* function for printing the elements of the circular linked list

5.6 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 5.5 depicts the logical representation of a doubly linked list:

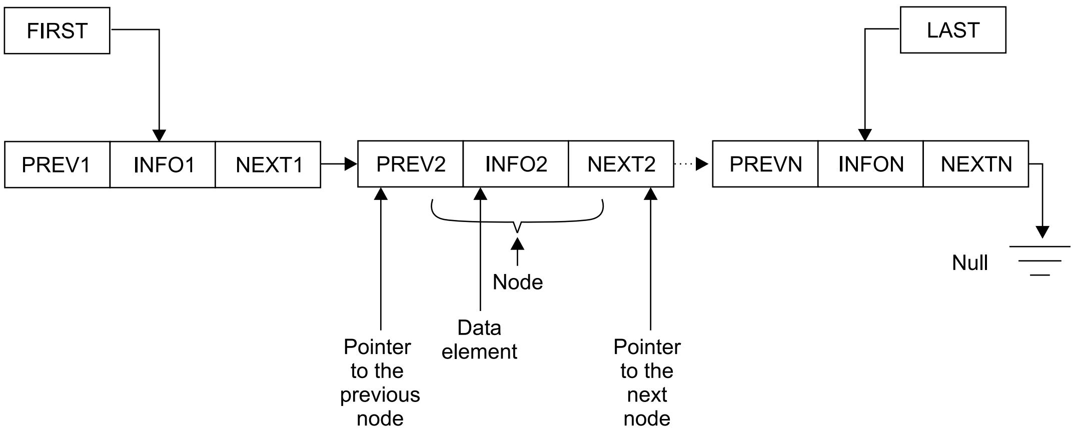


Fig. 5.5 *Logical representation of a doubly linked list*

The implementation of a doubly linked list involves two tasks:

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

5.6.1 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.

5.6.2 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 5.9 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 5.10 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.

5.6.3 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 5.11 Write a program to implement a doubly linked list and perform its common operations. Program 5.3 implements a doubly linked list in C. It uses the insert (Example 5.9) and delete (Example 5.10) algorithms for realizing the insert and delete operations on the doubly linked list. For performing the search and print operations, the same algorithms (Example 5.3 and Example 5.4) have been used that were earlier used for implementing a singly linked list.

Program 5.3 *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);
```

Chapter Five

```
   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 5.5 (singly linked list implementation).

Program analysis

Key Statement	Purpose

struct dl_node *FIRST = NULL;
struct dl_node *LAST = NULL;
	Declares pointers to the first and last nodes of the doubly linked list

void insert(int);
int delete(int);
void print(void);
struct dl_node *search (int);
	Declares the prototypes for the functions that perform operations on the doubly linked list

insert(num1);	Calls the *insert()* function for inserting an element into the doubly linked list
num2=delete(num1);	Calls the *delete()* function for deleting an element from the doubly linked list
location=search(num1);	Calls the *search()* function for searching an element in the doubly linked list
print();	Calls the *print()* function for printing the elements of the doubly linked list
struct dl_node *PTR = (struct dl_node*) malloc(sizeof(struct dl_node));	Creates a new node of the doubly linked list using dynamic memory allocation

PTR->NEXT = NULL;
PTR->PREVIOUS = LAST;
	Updates both the *NEXT* and *PREVIOUS* pointers of the node of a doubly linked list

Solved Problems

Problem 5.1 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 5.2 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

5.1 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.

5.2 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

5.3 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

5.4 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

5.5 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 forward direction.
(c) It allows list traversal in both forward and backward direction.
(d) It allows complete list traversal starting from any of the nodes.

5.6 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

5.7 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

5.8 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

5.1 What is a linked list? What are its various types?
5.2 Explain the representation of a linked list in memory with the help of an illustration.
5.3 Explain the typical operations that are performed on a linked list.
5.4 Explain the key advantages and disadvantages of linked lists.
5.5 What is a circular linked list? How is it different from a normal linked list?
5.6 What is a doubly linked list? Why is it used?
5.7 Write the algorithm for searching an element in a singly linked list.
5.8 Write the algorithm for inserting an element in a circular linked list.

Programming Exercises

5.1 Write a code snippet for declaring the node of a doubly linked list.
5.2 Write a C function to delete a node from a singly linked list.
5.3 Write a C function to insert a new node at the end of a circular linked list.
5.4 Write a C function to print the elements of a linked list.
5.5 Write a C function to print the elements of a doubly linked list in both forward and backward directions.

Answers to Multiple-Choice Questions

5.1 (c)	5.2 (d)	5.3 (a)	5.4 (b)	5.5 (c)
5.6 (a)	5.7 (b)	5.8 (a)		

6

STACKS

Chapter Outline

6.1 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.

6.2 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. 6.1 (a) and (b).

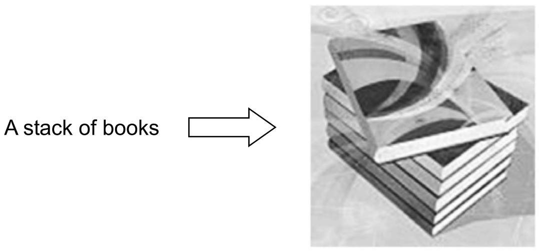


Fig. 6.1(a) *Stack of books*

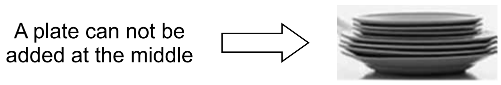


Fig. 6.1(b) *Stack of plates*

As we can see in Fig. 6.1, 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.

Mind Jog

Who discovered stacks?
Stack was first proposed in 1957 by a German computer scientist Friedrich L. Bauer.

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.

6.2.1 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 6.2 shows the logical representation of stacks in memory.

As we can see in Fig. 6.2, 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.*

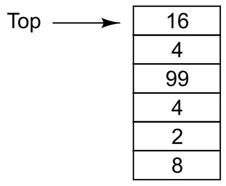


Fig. 6.2 *Logical representation of stacks*

6.2.2 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 6.1.

Table 6.1 *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.

6.3 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 6.3 (a) and (b) depict the push and pop operations on a stack.

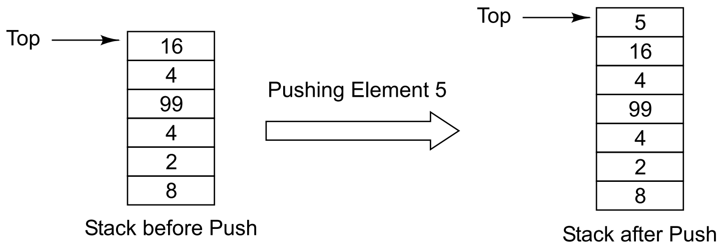


Fig. 6.3(a) *Push operation*

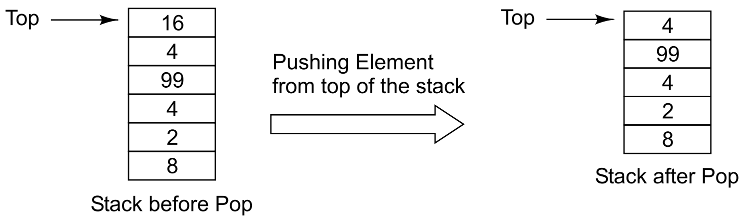


Fig. 6.3(b) *Pop operation*

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

6.3.1 Push

As we can see in Fig. 6.3 (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.*

6.3.2 Pop

As we can see in Fig. 6.3 (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.

Mind Jog

Which programming language provide built-in support for stacks?
LISP and Python

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.*

6.3.3 An Example of Stack Operations

Figure 6.4 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.

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.

6.4 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.

We can implement stacks by using arrays or linked lists. The advantages or disadvantages of array or linked implementations of stacks are the same that are associated with such types of data structures. However, both implementation types have their own usage in specific situations.

Mind Jog

Is stack a restricted data structure?
Yes, because limited operations can be performed on it.

6.4.1 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.

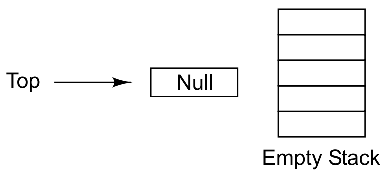
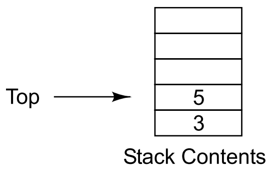
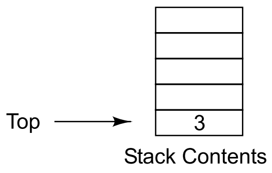
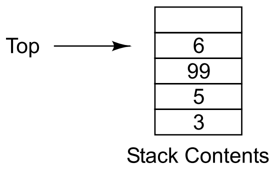
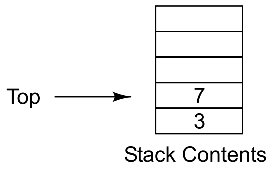
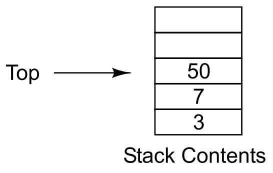
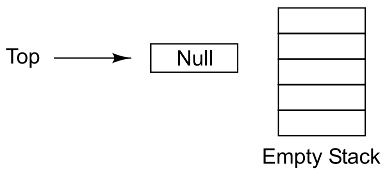
Initial Empty Stack	
Push (3), Push (5)	
Pop ()	
Push(7), Push (99), Push (6)	
Pop (), Pop ()	
Push (50)	
Pop (), Pop (), Pop ()	

Fig 6.4 *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 6.1 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 6.2 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 6.3 Write a program to implement a stack using arrays and perform its common operations.

Program 6.1 implements a stack using arrays in C. It uses the push (Example 6.1) and pop (Example 6.2) algorithms for realizing the common stack operations.

Program 6.1 *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*/

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

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

```
  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

Tip *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
int stack[100];	Declares an array to represent a stack

void push(int);
int pop();
void display();
	Declares prototypes for the functions that perform stack operations

push(num1);	Calls the *push()* function for inserting an element into the stack
num2=pop();	Calls the *pop()* function for deleting an element from the stack
display();	Calls the *display()* function for displaying the stack elements

top=top+1;
stack[top]=element;
	Inserts an element at the top of the stack and updates the stack pointer

6.4.2 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 6.4 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 6.5 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 6.6 Write a program to implement a stack using linked lists and perform its common operations.

Program 6.2 implements a stack using linked lists in C. It uses the push (Example 6.4) and pop (Example 6.5) algorithms for realizing the common stack operations.

Program 6.2 *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;
```

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

```
  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);
```

```
    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;
    }
    }
   }
```

malloc function is used for dynamic or runtime reservation of space for new stack elements

```
   /*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()
   {
    if(top==NULL) /*Checking whether the stack is empty*/
    {
    printf("\n\STACK 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("\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

Chapter Six

Output

The output of the above program is same as the output of the program shown in Example 6.3.

Program analysis

Key Statement	Purpose

**struct stack
{
int element;
struct stack *next;
}*top;**
	Uses linked list to represent a stack and declares the stack pointer

**void push(int);
int pop();
void display();**
	Declares prototypes for the functions that perform stack operations

push(num1);	Calls the *push()* function for inserting an element into the stack
num2=pop();	Calls the *pop()* function for removing an element from the stack
display();	Calls the *display()* function for displaying the stack elements

**ptr->element=value;
ptr->next=top;
top=ptr;**
	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 6.1 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 6.2 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 6.3 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 6.4 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 6.5 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 6.6 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

6.1 Which of the following is not true for stacks?
(a) It is a linear data strucure.
(b) It allows insertion/deletion of elements only at one end
(c) It is widely used by systems processes, such as compilation and program control
(d) It is based on First-In-First-Out principle

6.2 Which of the following is not an example of a stack?
(a) Collection of tiles one over another
(b) A set of bangles worn by a lady on her arm
(c) A line up of people waiting for the bus at the bus stop
(d) A pileup of boxes in a warehouse one over another

6.3 Tower of Hanoi can be regarded as a problem of which of the following data structures?
(a) Stack (b) Queue
(c) Graph (d) Tree

6.4 Recursive function calls are executed using which of the following data structures?
(a) Stack (b) Queue
(c) Graph (d) Tree

6.5 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)
(a) 11, 2, 1 (b) 8, 11, 7
(c) 7, 5, 8 (d) 5, 8, 7

6.6 Which of the following is best suitable for storing a simple collection of employee records?
(a) Stack (b) Queue
(c) Array (d) None of the above

6.7 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?
(a) top = top + 1; stack [top] = item;
(b) stack [top] = item; top = top + 1;
(c) stack [top++] = item;
(d) Both (a) and (c) are correct

6.8 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) top = top – 1; item = stack [top];
(b) item = stack [top]; top = top – 1;
(c) item = stack [--top];
(d) Both (b) and (c) are correct

6.9 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

6.10 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

6.1 What is a stack? Explain with examples.
6.2 Briefly describe the LIFO principle.
6.3 What is a top pointer? Explain its significance.
6.4 What are the different application areas of stack data structure?
6.5 Give any four real-life examples that principally resemble the stack data structure.
6.6 Explain the logical representation of stacks in memory with the help of an example.
6.7 Explain push and pop operations with the help of examples.
6.8 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 ( )
6.9 What will happen if we keep on pushing elements into a stack one after another?
6.10 What will happen if we continue to pop out elements from a stack one after another?
6.11 How are stacks implemented?
6.12 What is the advantage of linked implementation of stacks over array implementation?
6.13 What role does dynamic memory management techniques play in linked implementation of stacks?
6.14 Briefly explain the overflow and underflow conditions along with their remedies.
6.15 Can an overflow situation occur even with linked implementation of stacks that uses dynamic memory allocation techniques? Explain.

Programming Exercises

6.1 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.

6.2 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.

6.3 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.

6.4 Modify the C program solution of Question 6.3 to store the *N* elements in sorted fashion with the largest element stored at the *TOP*.

6.5 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.

6.6 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) A larger disk can not be placed on a smaller disk

(b) Only one disc can be shifted at a time

Write a C program to find a solution to the above problem using stacks.

6.7 An input text string comprises the following:

(a) Letters

(b) Digits

(c) 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.

6.8 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

6.1 (d)	6.2 (c)	6.3 (a)	6.4 (a)	6.5 (c)
6.6 (c)	6.7 (a)	6.8 (b)	6.9 (b)	6.10 (c)

7

QUEUES

Chapter Outline

7.1 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.

7.2 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. 7.1 (a) and (b).

Fig. 7.1(a) *Queue of people*

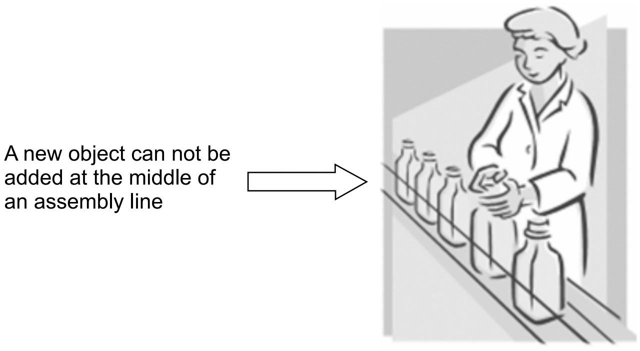

A new object can not be added at the middle of an assembly line

Fig. 7.1(b) *An assembly line*

As we can see in Fig. 7.1(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. 7.1(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 7.2 depicts this scenario.

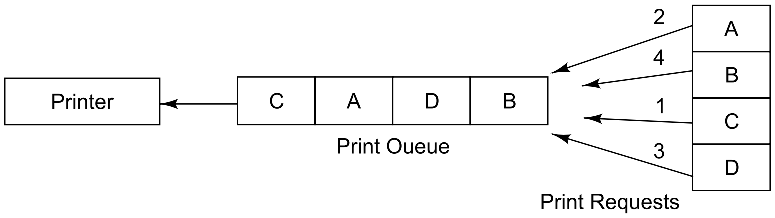


Fig. 7.2 *Queue implementation*

As we can see in the Fig. 7.2, 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 deaueue.

7.2.1 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 7.3 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.

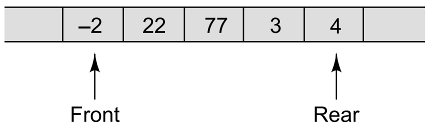


Fig. 7.3 *Logical representation of queues*

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.*

7.3 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 7.4 (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.

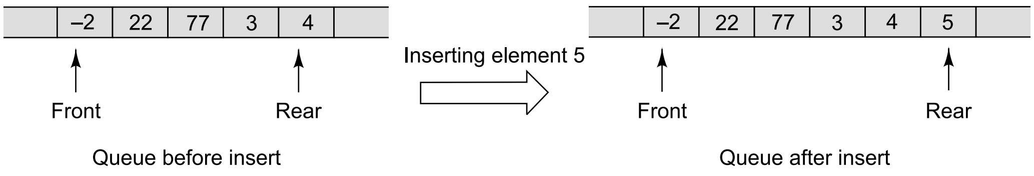


Fig. 7.4(a) *Insert operation*

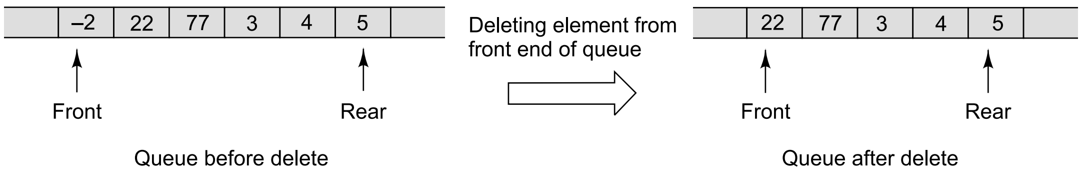


Fig. 7.4(b) *Delete operation*

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. 7.4(a), the insert operation involves the following subtasks:
(a) Receiving the element to be inserted.
(b) Incrementing the queue pointer, *rear*.
(c) 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. 7.4 (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 7.5 shows how the queue contents change after a series of insert and delete operations.

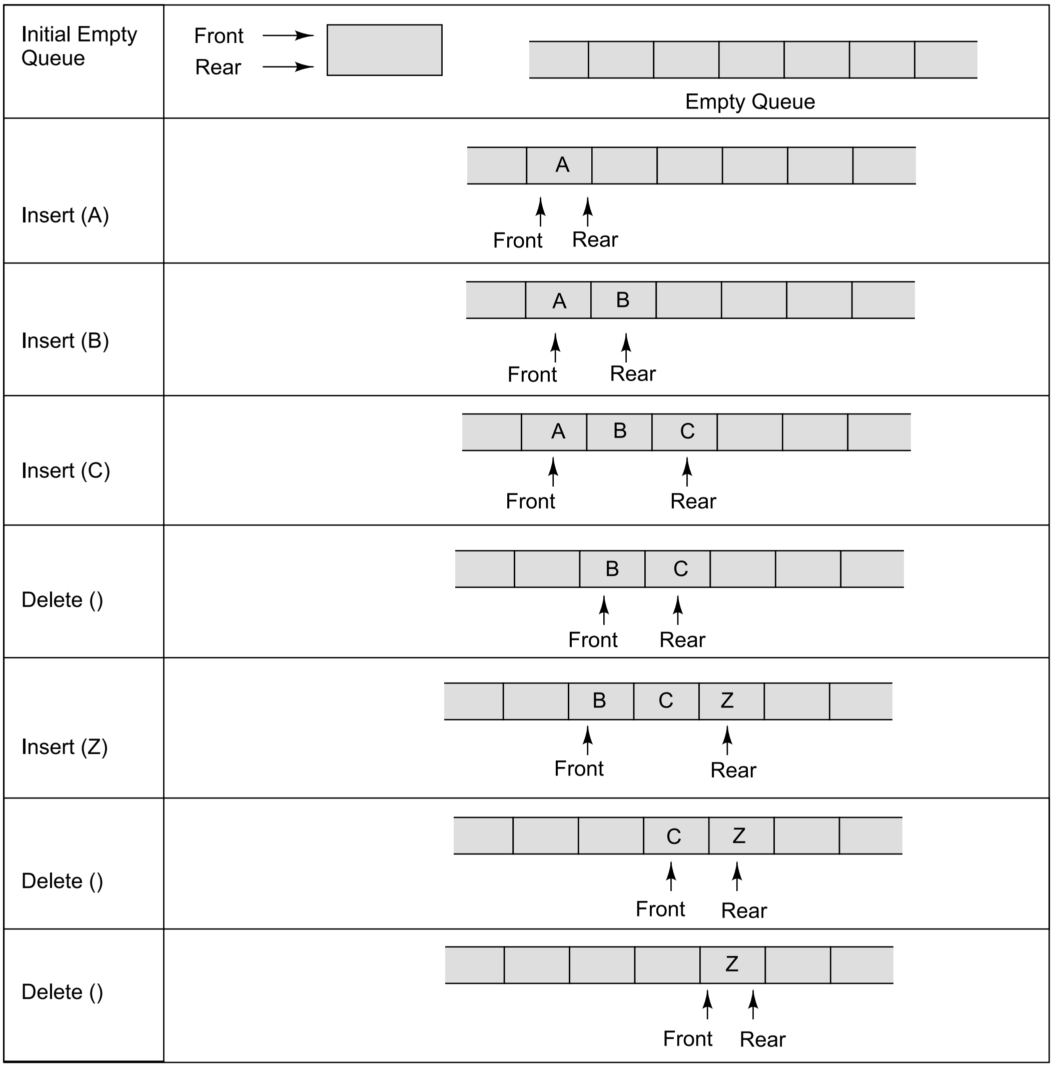


Fig. 7.5 *Queue operations*

We can see in Fig. 7.5 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.*

7.4 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.

Mind Jog

What is a bounded queue?
It is a queue restricted to a fixed number of elements.

7.4.1 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.

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.

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 7.1 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
```

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 7.2 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 7.3 Write a program to implement a queue using arrays and perform its common operations.

Program 7.1 implements a queue using arrays in C. It uses the insert (Example 7.1) and delete (Example 7.2) functions for realizing the common queue operations.

Program 7.1 *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*/

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()
```

If we do not initialize the front and rear variables then they may continue to store garbage value which may lead to erroneous results

Chapter Seven

```
{
 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
int queue[100];	Declares an array to represent a queue

void insert(int);
int del();
void display();
	Declares prototypes for the functions that perform queue operations

insert(num1);	Calls the *insert()* function for inserting an element into the queue
num2=del();	Calls the del*()* function for deleting an element from the queue
display();	Calls the *display()* function for displaying the queue elements

rear=rear+1;
queue[rear]=element;
	Inserts an element at the end of the queue and updates the rear pointer

if(front==-1)	Checks whether or not the queue is empty

7.4.2 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 7.4 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 7.5 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 7.6 Write a program to implement a queue using linked lists and perform its common operations.

Program 7.2 implements a queue using linked lists in C. It uses the insert (Example 7.4) and delete (Example 7.5) functions for realizing the common queue operations.

Program 7.2 *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*/
  };
```

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

```
  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)
   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 the queue is empty then the stack pointer (front) will point at NULL

```
 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);
 }
}
```

Output

The output of the above program is same as the output of the program shown in Example 7.3.

Program analysis

Key Statement	Purpose

struct queue
{
int element;
struct queue *next;
};
	Uses linked list to represent a queue

struct queue *front=NULL;
struct queue *rear = NULL;
	Declares queue pointers

void insert(int);
int del();
void display(void);
	Declares prototypes for the functions that perform queue operations

insert(num1);	Calls the *insert()* function for inserting an element into the queue
num2=del();	Calls the del*()* function for deleting an element from the queue
display();	Calls the *display()* function for displaying the queue elements

rear->next = ptr;
ptr->next = NULL;
rear = ptr;
	Inserts an element at the end of the queue and updates the rear pointer

if(front==NULL)	Checks whether or not the queue is empty

7.5 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.

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.

Figure 7.6 shows the logical representation of a circular queue.

As we can see in Fig. 7.6, 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 7.7 shows the different states of a circular queue during insert and delete operations.

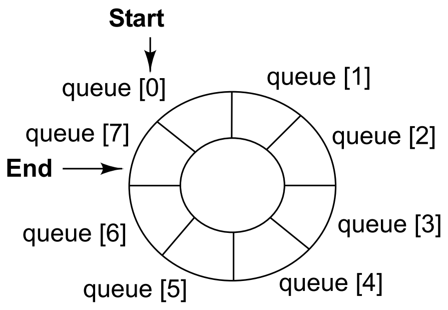


Fig. 7.6 *Circular queue*

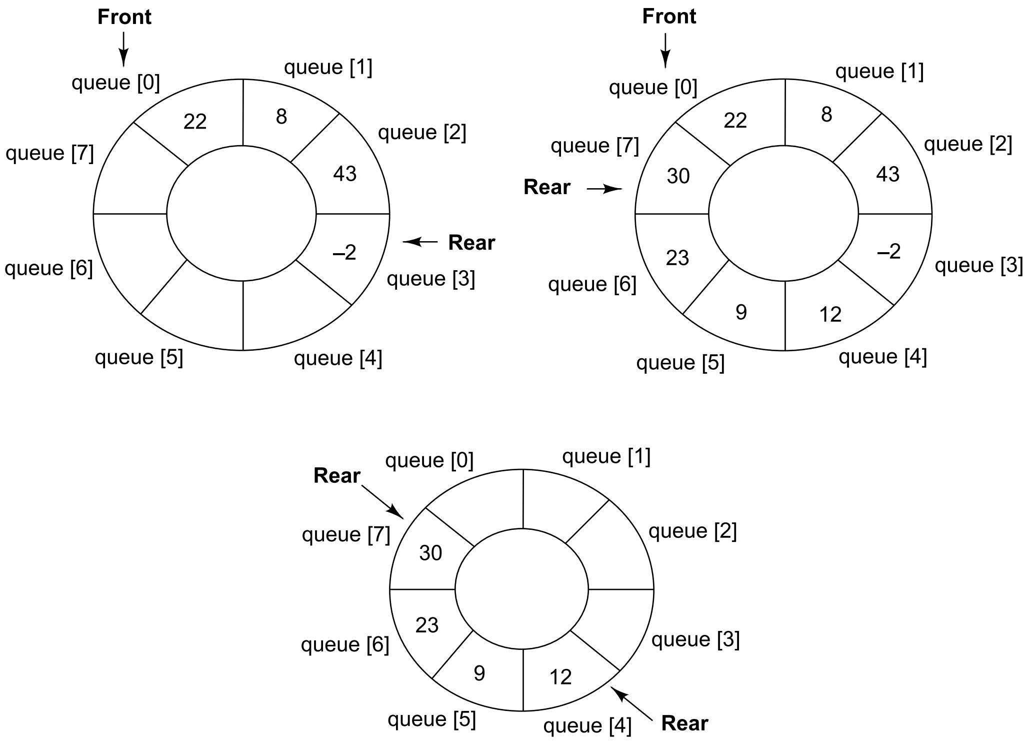


Fig. 7.7 *Inserting and deleting elements in a circular queue*

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 7.7 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 7.8 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 7.9 Write a program to implement a circular queue using arrays and perform its common operations.

Program 7.3 implements a circular queue using arrays in C. It uses the insert (Example 7.7) and delete (Example 7.8) functions for realizing the common queue operations.

Program 7.3 *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",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
if((front==0 && rear ==4) || front==rear+1)	Checks whether the circular queue is full or not
i=front; **while(i!=rear)**	Traverses the elements of the circular queue

7.6 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).

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.

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:

```
  struct queue /*Node of a priority queue*/
  {
   int element;
   int priority;
   struct queue *next; /*Pointer to the next queue node*/
  };
```

Implementation

Example 7.10 Write a program to implement a priority queue using linked lists and perform its common operations.

Program 7.4 implements a priority queue using linked lists in C.

Program 7.4 *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\t\tPriority\n");
 /*Printing queue elements along with their priority*/
 printf("Front->");
 while(ptr!=NULL)
 {
 printf("\t %d\t\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

s*struct 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

7.7 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).

1. **Input-restricted deque** It allows deletion from both the ends but restricts the insertion at only one end.
2. **Output-restricted deque** It allows insertion at both the ends but restricts the deletion at only one end.

Figure 7.8 shows the logical representation of a deque.

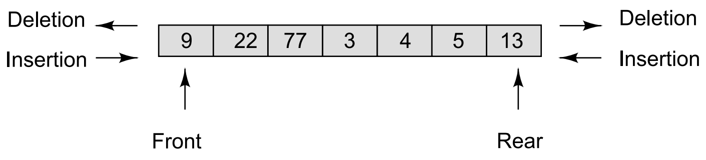


Fig. 7.8 *Double-ended queue*

As shown in Fig. 7.8, 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:

1. **i_front** Insertion at front end of the queue.
2. **d_front** Deletion from front end of the queue.
3. **i_rear** Insertion at rear end of the queue.
4. **d_rear** Deletion from rear end of the queue.

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.

Example 7.11 Write C functions to realize the four possible insert and delete operations for array-implemented double-ended queues.

Program 7.5 *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*/
}
```

Program 7.6 *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 7.7 *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 7.8 *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 7.1 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)
Step 1 R = R + 1 = 3 + 1 = 4
Step 2 Q [R] = Q [4] = 16

Queue contents

Queue (Q)	4	5	-9	66	16			
Index	*0*	*1*	2	*3*	*4*	*5*	*6*	*7*

F↑ *R* ↑

(b) Delete (Q)
Step 1 Item = Q [F] = Q [0] = 4
Step 2 F = F + 1 = 0 + 1 = 1

Queue contents

Queue (Q)		5	–9	66	16			
Index	*0*	*1*	*2*	*3*	*4*	5	*6*	*7*

F↑ *R* ↑

(c) Delete (Q)
Step 1 Item = Q [F] = Q [1] = 5
Step 2 F = F + 1 = 1 + 1 = 2

Queue contents

Queue (Q)			–9	66	16			
Index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*

F↑ *R* ↑

(d) Insert (Q, 7)
Step 1 R = R + 1 = 4 + 1 = 5
Step 2 Q [R] = Q [5] = 7

Queue contents

Queue (Q)			–9	66	16	7		
Index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*

F↑ *R* ↑

(e) Delete (Q)
Step 1 Item = Q [F] = Q [2] = –9
Step 2 F = F + 1 = 2 + 1 = 3

Queue contents

Queue (Q)				66	16	7		
Index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*

F↑ *R* ↑

(f) Insert (Q, –2)
Step 1 R = R + 1 = 5 + 1 = 6
Step 2 Q [R] = Q [6] = –2

Queue contents

Queue (Q)			–9	66	16	7	–2	
Index	*0*	*1*	*2*	*3*	*4*	5	*6*	*7*
				F↑				*R* ↑

Problem 7.2 Consider the following two states of a queue Q:

State 1

Queue (Q)		4	–1	8				
Index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*
			F↑		*R* ↑			

State 2

Queue (Q)				3	11	22	33	
Index	*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 7.3 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 7.4 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

7.1 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.

7.2 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.

7.3 CPU scheduler can be implemented by which of the following data structures?
(a) Stack (b) Queue
(c) Graph (d) Tree

7.4 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

7.5 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

7.6 Which of the following is best suitable for implementing a print scheduler?
(a) Stack (b) Queue
(c) Array (d) None of the above

7.7 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

7.8 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

7.9 If a delete operation is performed on an empty queue, then which of the following situations will occur?

(a) Overflow	(b) Underflow
(c) Array out of bound	(d) None of the above

7.10 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

7.1 What is a queue? Explain with examples.
7.2 Briefly describe the FIFO principle.
7.3 What are front and rear pointers? Explain their significance.
7.4 What are the different application areas of queue data structure?
7.5 Give any three real-life examples that principally resemble the queue data structure.
7.6 Explain the logical representation of queue in memory with the help of an example.
7.7 Explain insert and delete queue operations with the help of examples.
7.8 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 ( )
7.9 What is a priority queue? How is it different from a normal queue?
7.10 Explain the significance of double-ended queue.
7.11 How are queues implemented?
7.12 What is the advantage of linked implementation of queues over array implementation?
7.13 What is the objective of implementing a queue in circular fashion?
7.14 List the differences between stack and queue data structures.
7.15 How is a double-ended queue implemented?

Programming Exercises

7.1 Write a C function to print the elements of a queue implemented using linked list.
7.2 Write a C function to perform the delete operation on an array-implemented circular queue.
7.3 Write a C function to insert an element into a priority queue as per priority.
7.4 Write a C function to perform the delete operation at the end of a double-ended queue.
7.5 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

7.1 (b)	7.2 (a)	7.3 (b)	7.4 (d)	7.5 (a)
7.6 (b)	7.7 (d)	7.8 (b)	7.9 (b)	7.10 (a)

8

TREES

Chapter Outline

8.1 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.

8.2 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 8.1 shows a sample tree T.

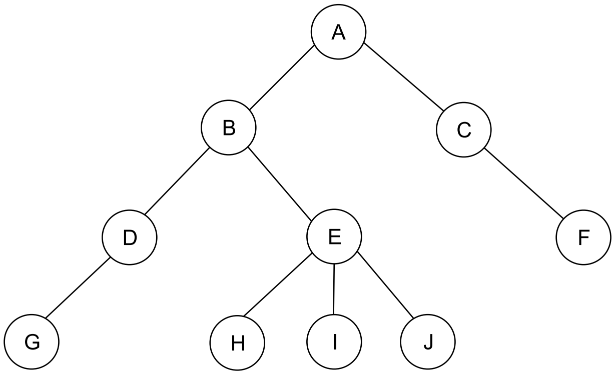


Fig. 8.1 *Tree T*

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.

8.2.1 Tree Terminology

There are a number of key terms associated with trees. Table 8.1 lists some of the important key terms.

Table 8.1 *Tree terminology*

Key Term	Description	Example (Refer to Fig. 8.1)
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. 8.1)
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

8.3 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 8.2 shows a binary tree.

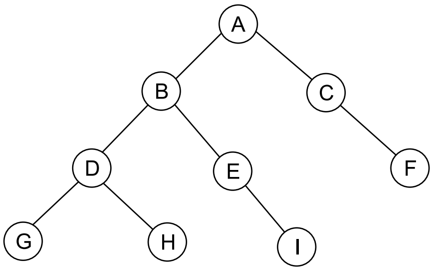


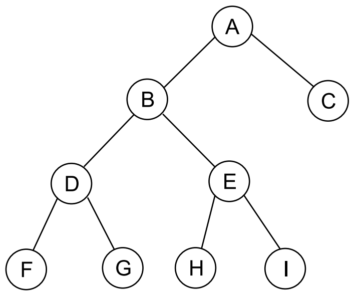
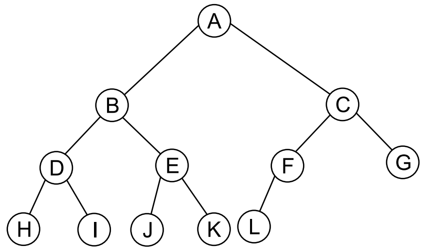
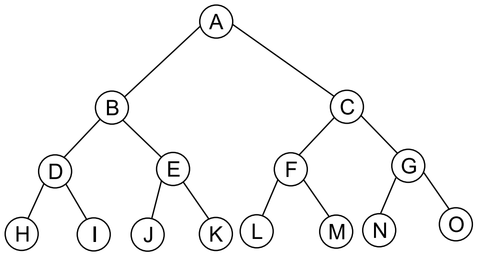
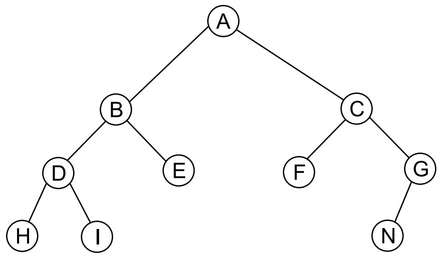
Fig. 8.2 *Binary tree*

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.

8.3.1 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 8.2 lists these key concepts.

Table 8.2 *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.	
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.	
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.	
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.	

8.4 BINARY TREE REPRESENTATION

The sequential representation of binary trees is done by using arrays while the linked representation is done by using linked lists.

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.

8.4.1 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 **2*i*+1** while the right child node will be stored at location **2*i*+2**.

Let us consider a binary tree T_1, as shown in Fig. 8.3.

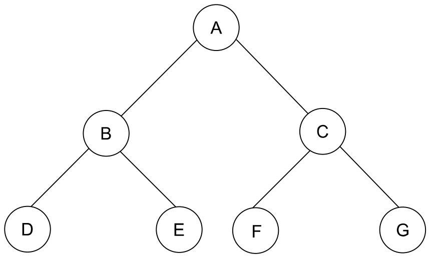


Fig. 8.3 *Binary tree T_1*

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 8.4 shows the array representation.

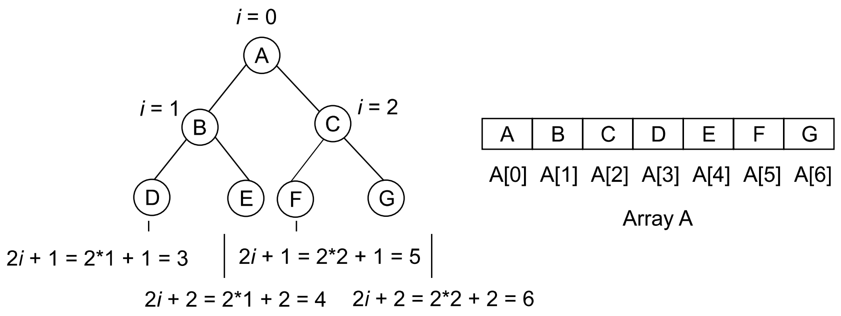


Fig. 8.4 *Array representation of binary tree T_1*

Figure 8.4 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. 8.5.

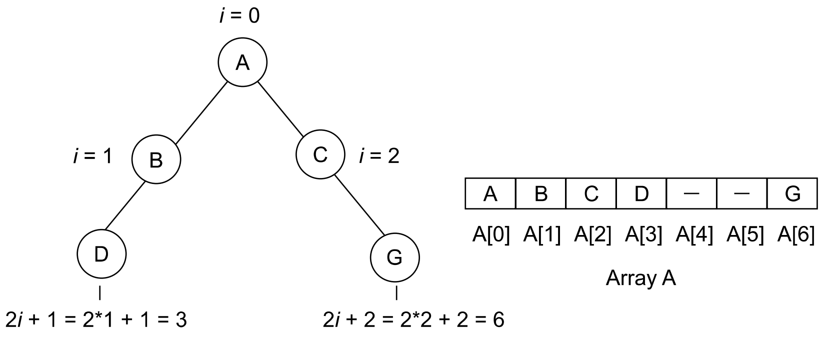


Fig. 8.5 *Revised array representation of binary tree T_1*

As we can see in Fig. 8.5, 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.

8.4.2 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 8.6 shows how linked list is used for representing a binary tree in memory.

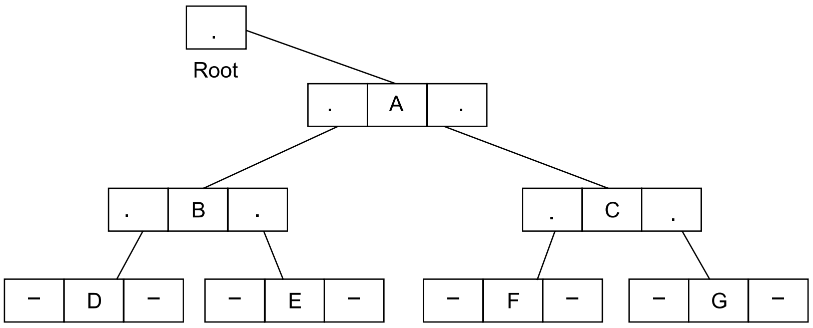


Fig. 8.6 *Linked representation of binary tree*

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 8.1 Write a program to implement a binary tree using linked list.

Program 8.1 *Implementation of a binary tree*

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

 struct bin_tree
 {
  int INFO;
  struct node *LEFT, *RIGHT;
 };
```

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.

```
 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;
```

Since the root node has no parents, its location is tracked with the help of a special pointer called root.

```
  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:
  {
   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

node *insert(node *,int);
void display (node *);
	Declares function prototypes for inserting and displaying binary tree nodes

root = insert(root,element);	Calls the *insert()* function for inserting a new node into the binary tree
display(root);	Calls the *display()* function for displaying the binary tree nodes

if(count%2==0)
r->LEFT = insert(r->LEFT, n);
else
r->RIGHT = insert(r->RIGHT, n);
	Checks the value of the *count* variable to insert the new node either in the left or right subtree

8.5 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 8.7 shows an illustration of the different binary tree traversal methods.

Example 8.2 Consider the following binary tree:

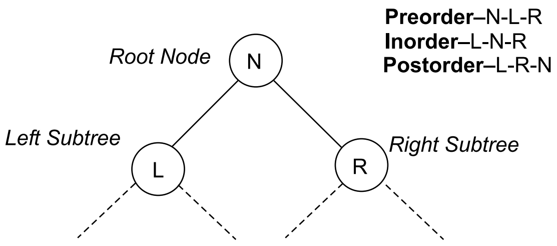
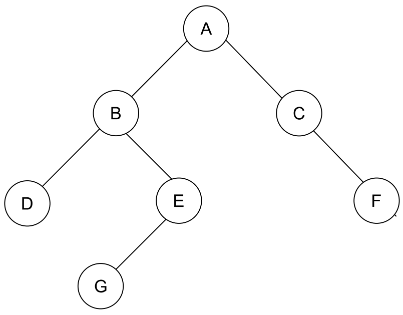


Fig. 8.7 *Binary tree traversal*



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 8.3 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 8.4 Modify the program shown in Example 8.1 to add preorder, inorder and postorder traversals to the linked implementation of binary tree.

Program 8.2 *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

node *insert(node *,int);
void preorder(node *);
void inorder(node *);
void postorder(node *);
	Declares the function prototypes for inserting a new node and traversing the binary tree using different traversal methods

preorder(root);	Calls the *preorder()* function to traverse the binary tree in preorder sequence
inorder(root);	Calls the *inorder()* function to traverse the binary tree in inorder sequence
postorder(root);	Calls the *postorder()* function to traverse the binary tree in postorder sequence

8.6 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.

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

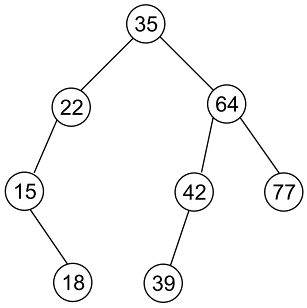


Fig. 8.8 *Binary search tree*

Figure 8.8 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.

Example 8.5 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 8.6 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. 8.9.

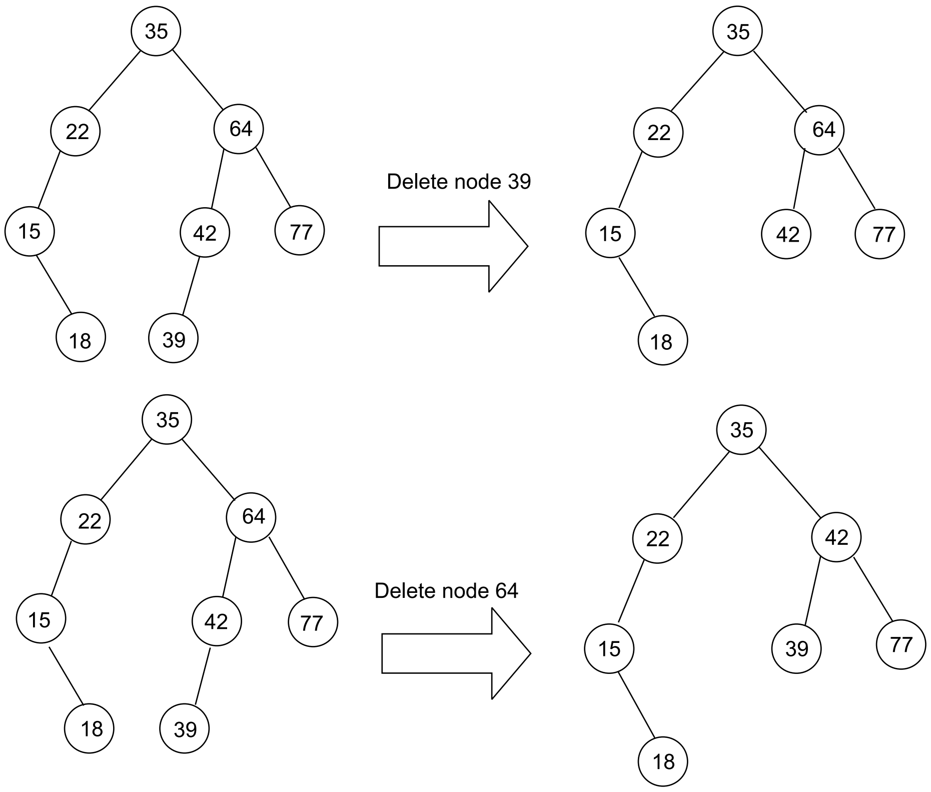


Fig. 8.9 *Deleting an element from binary search tree*

As we can see in Fig. 8.9, 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 8.7 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 8.8 Write a C program for implementing a binary search tree.
Program 8.3 uses the insert (Example 8.5), search (Example 8.6), and delete (Example 8.7) functions.

Program 8.3 *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

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!!
```

Irrespective of the order in which elements are added to the binary search tree, the insert function always stores them in a sorted manner.

```
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

node *insert(node *,int);
void search(node *,int);
int del(node *,int);
void display(node*);
	Declares function prototypes for performing operations on the binary search tree

root = insert(root,element);	Calls the *insert()*function for inserting a new node into the binary search tree
search(root,num);	Calls the *search()*function for performing search operation on the binary search tree
flag=del(root,num);	Calls the *del()*function for deleting an element from the binary search tree
display(root);	Calls the *display()*function for displaying the nodes of the binary search tree

8.7 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.

8.7.1 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. 8.10.

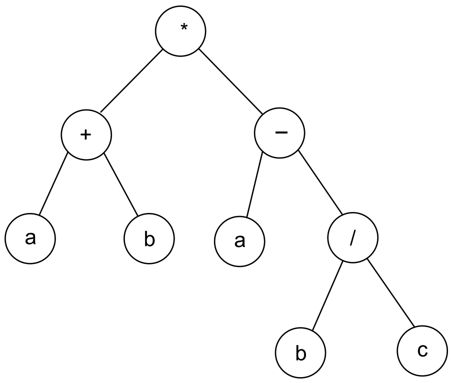


Fig. 8.10 *Expression tree*

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 8.3 shows the various expression notations deduced after traversing the expression tree shown in Fig. 8.10.

Table 8.3 *Expression notations*

Expression Notation	Traversal Method	Example (Refer to Fig. 8.10)
Prefix	Preorder	*+ab–a/bc
Infix	Inorder	a+b*a–a/c
Postfix	Postorder	ab+abc/-*

8.7.2 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 8.11 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.

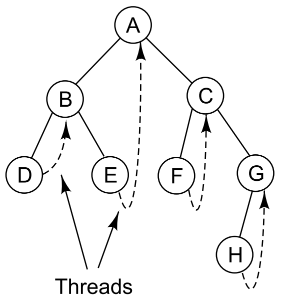


Fig. 8.11 *Threaded binary tree*

Example 8.9 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. 8.11, 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

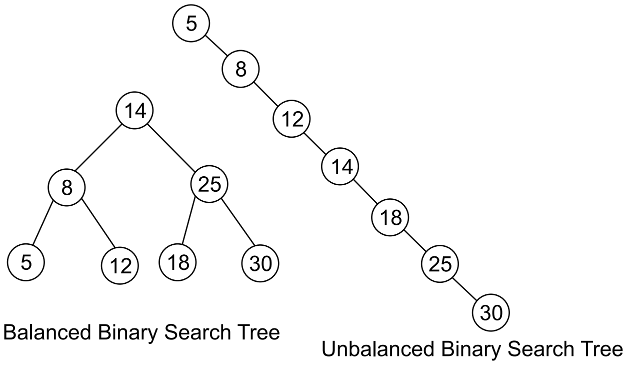


Fig. 8.12 *Binary search trees*

8.7.3 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. 8.12.

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 8.13 shows an AVL tree.

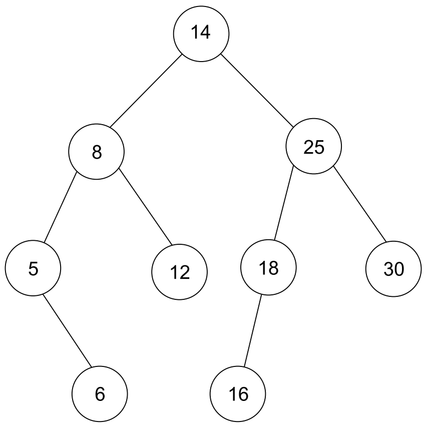


Fig. 8.13 *AVL tree*

As shown in Fig. 8.13, 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 = Height of Left Subtree – 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. 8.13. Figure 8.14 shows the updated AVL tree.

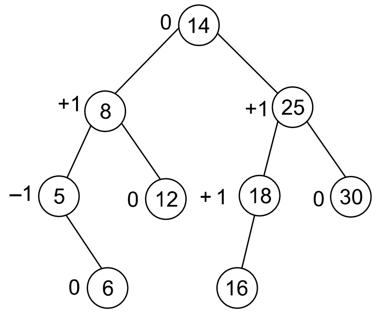


Fig. 8.14 *AVL tree with balance factors*

As shown in Fig. 8.14, 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 8.15 shows the insertion of node value 15 into the AVL tree shown in Fig. 8.14. It depicts how the disbalance resulting out of the insert operation is corrected.

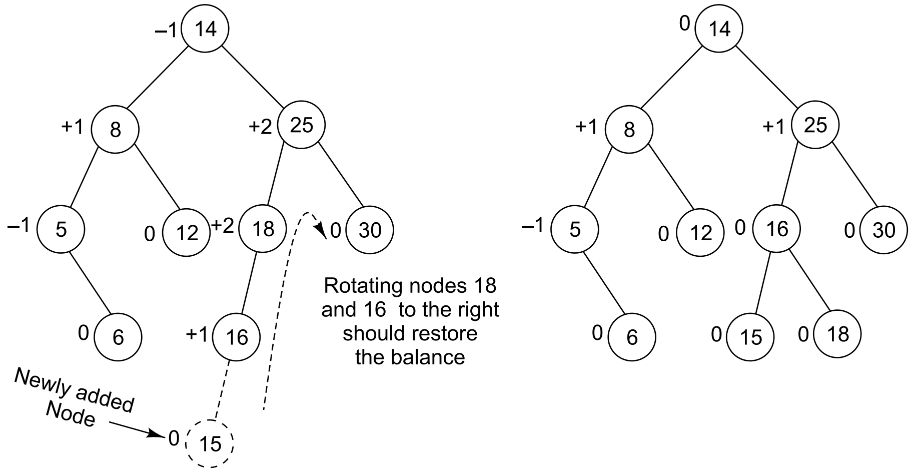


Fig. 8.15 *Inserting an element in an AVL tree*

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 (logn) 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 8.16 shows a red-black tree.

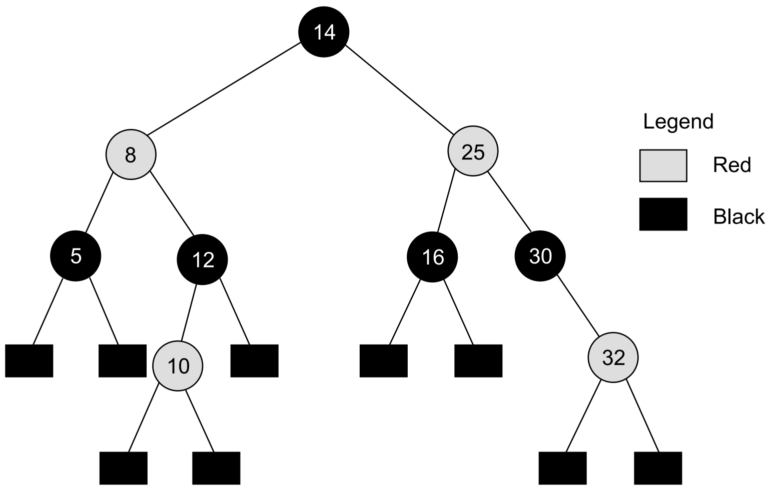


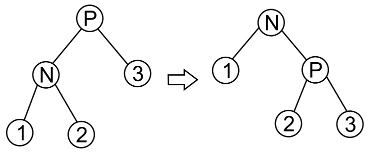
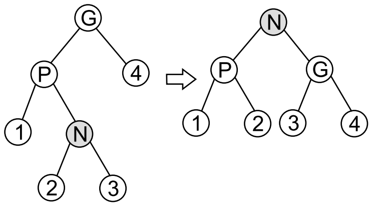
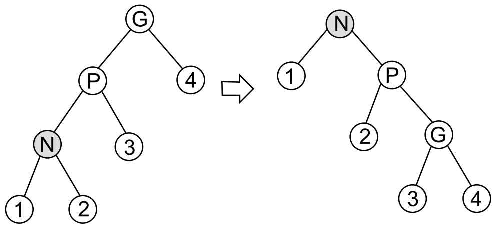
Fig. 8.16 *Red-Black tree*

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 (logn).

8.7.4 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 8.4 shows the various types of splay rotations along with an illustration.

Table 8.4 *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.

8.7.5 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 8.17 shows a sample m-way tree.

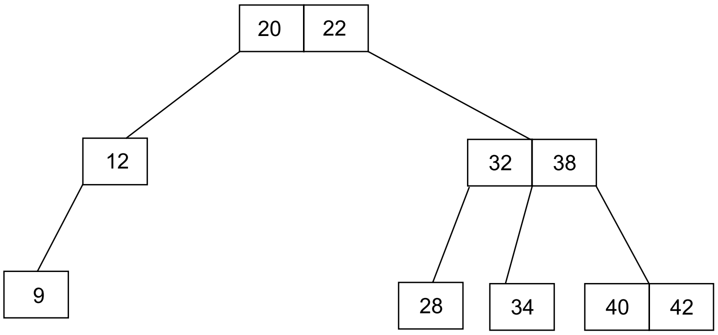


Fig. 8.17 *Sample m-way tree*

(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 8.18 shows a sample B tree.

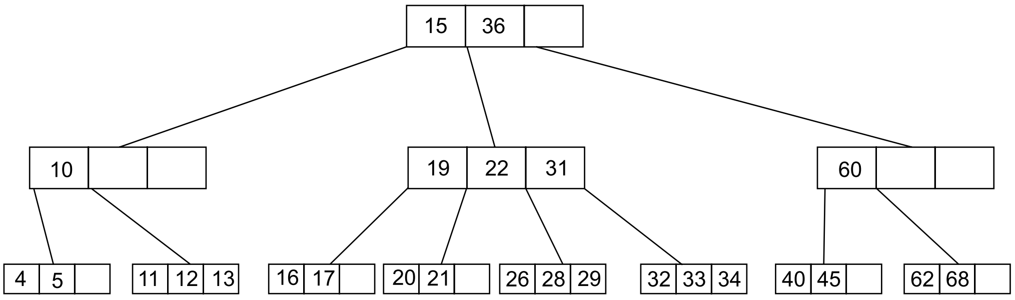


Fig. 8.18 *Sample B tree*

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:

i. The internal nodes contain up to m–1 keys.
ii. The number of children of internal nodes lies between m/2 and m.
iii. The subtree between keys k1 and k2 contains values v such that $k1 \le v < k2$.
iv. All leaf nodes are at the same level.
v. 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 8.19 shows a sample B^+ tree.

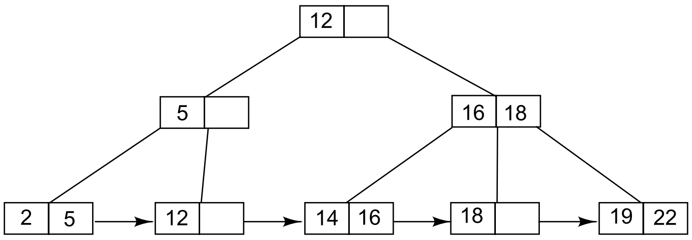


Fig. 8.19 *Sample B^+ tree*

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^+ 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

8.1 All the child nodes of a parent node are referred as ________?
(a) neighbors (b) siblings
(c) internal nodes (d) leaf nodes

8.2 The degree of a binary tree is
(a) 1 (b) 2
(c) 3 (d) *n*, where *n* is the number of nodes in the tree

8.3 The right pointer of a threaded binary tree points at
(a) NULL (b) Root
(c) inorder successor (d) postorder successor

8.4 The expression, (a+b)*(a–b) is stored in an expression tree. What will be its preorder sequence?

(a) (a+b)*(a–b) (b) +ab-ab*

(c) ab+ab-* (d) *+ab–ab

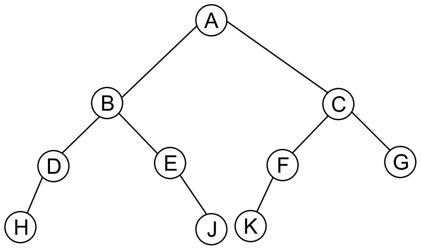
8.5 Which of the following trees stores its elements in a sorted manner?

(a) General tree (b) Binary tree

(c) Binary search tree (d) None of the above

Review Questions

8.1 What is a tree? Explain any five key terms associated with a tree.
8.2 What is the difference between complete binary tree and perfect binary tree?
8.3 What are the two types of balanced binary trees? Explain with the help of an illustration.
8.4 What is the advantage of linked implementation of a binary tree over array implementation?
8.5 What are the different types of tree traversal methods? Explain with the help of an example.
8.6 Deduce the preorder and postorder sequences for the following binary tree:



8.7 What is a binary search tree? Explain with the help of an example.
8.8 What is an expression tree? Explain with the help of an example.
8.9 What is a splay tree? Explain the different types of splay rotations.
8.10 hat is an m-way tree? Explain the two instances of m-way trees.

Programming Exercises

8.1 Write a function in C to count the number of nodes in a binary tree.
8.2 Write a function in C to display the elements of a binary search tree in ascending order.
8.3 Write a function in C that displays all the leaf nodes of a binary tree.
8.4 Write a function in C that returns the degree of a binary tree node.
8.5 Write a C function that transforms a given binary tree into a binary search tree.

Answers to Multiple-Choice Questions

8.1 (b) 8.2 (b) 8.3 (c) 8.4 (d) 8.5 (c)

9

GRAPHS

Chapter Outline

9.1 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.

9.2 BASIC CONCEPT

A graph G consists of the following elements:

- A set V of vertices or nodes where V = $\{v_1, v_2, v_3, ..., v_n\}$
- A set E of edges also called arcs where E = $\{e_1, e_2, e_3, ..., e_n\}$

Here, G = (V, E).

Figure 9.1 shows a sample graph G.

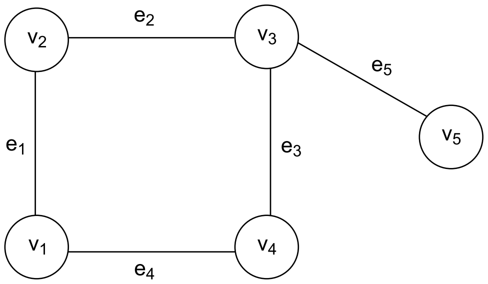


Fig. 9.1 *Graph G*

In Fig. 9.1, 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. 9.2.

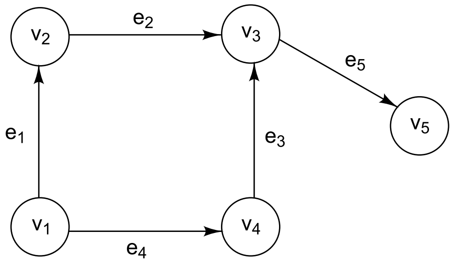


Fig. 9.2 *Directed graph*

In Fig. 9.2 graph, the set of vertices and edges are:

V(G) = $\{v_1, v_2, v_3, v_4, v_5\}$

E(G) = $\{(v_1, v_2), (v_2, v_3), (v_1, v_4), (v_4, v_3) , (v_3, v_5)\}$

9.3 GRAPH TERMINOLOGY

There are a number of key terms associated with the concept of graphs. Table 9.1 explains some of these important key terms.

Table 9.1 *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. 9.1, 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. 9.2, 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.*

9.4 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

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.

9.4.1 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 × 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. 9.3.

The adjacency matrix of the above graph will be

$A_{i,j}$	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

Here, 0s represent that there is no directed edge between the corresponding vertices while 1s represent the presence of a directed edge.

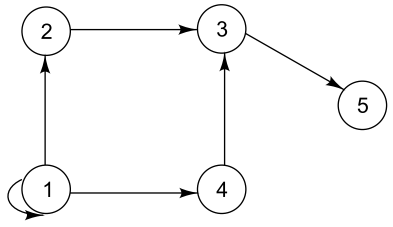


Fig. 9.3 *Graph*

Example 9.1 Write a program in C to represent a graph using adjacency matrix.

Program 9.1 represents the directed graph shown in Fig. 9.3 with the help of adjacency matrix.

Program 9.1 *C program to represent adjacency matrix*

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


void main()
{
 int A[5][5];
 int i,j;
 clrscr();
```

```
  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.

9.4.2 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 × 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:

$P_N = A + A^2 + A^3 + \ldots + A^N$

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, $P_{i,j} = P_{i,j}$ OR ($P_{i,k}$ AND $P_{k,j}$)

Here, OR represents the logical OR operation and AND represents the logical AND operation.

Example 9.2 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 9.3 Modify the program shown in Example 9.1 and apply Warshall's algorithm to derive the path matrix of a diagraph.

Program 9.2 uses Warshall's algorithm to derive the path matrix of the digraph shown in Fig. 9.3.

Program 9.2 *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

9.4.3 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 9.4 shows the adjacency list of the directed graph shown in Fig. 9.3.

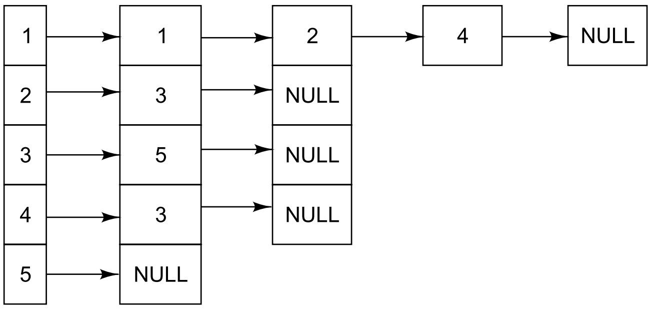


Fig. 9.4 *Adjacency list*

The adjacency list shown above contains five nodes with each node pointing towards its successor nodes.

Example 9.4 Write a program in C to represent a graph using adjacency list.

Program 9.3 represents a directed graph using adjacency list.

Program 9.3 *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

**struct vertex
{
struct vertex *edge[10];
int id;
}node[10];**
	Declares a structure to represent a graph node

void display(int);	Declares the function prototype for displaying the graph represented by adjacency list

**if(ch=='y')
node[i].edge[j]=&node[j];
else
node[i].edge[j]=NULL;**
	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.*

9.5 SHORTEST PATH ALGORITHM

Check Point

1. Write the Warshall's relation for computing a path matrix?
Ans. *P[i,j] = P[i,j] OR (P[i,k] AND P[k,j])*
2. The end of adjacency list of a graph node is signified by ______________.
Ans. NULL pointer

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, Ahemadabad, 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. 9.5.

The weight matrix of the above graph will be

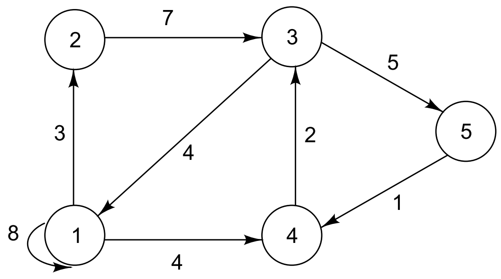


Fig. 9.5 *Weighted digraph*

$W_{i,j}$	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 ∞, 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:

$SP_{i, j}$ = 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 9.5 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 9.6 Write a program in C to deduce the shortest path matrix of a weighted digraph G.

Program 9.4 uses modified Warshall's algorithm to derive the shortest path matrix of the weighted digraph shown in Fig. 9.5.

Program 9.4 *Shortest path matrix of the weighted diagraph 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];
```

Chapter Nine

```
 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
int P[5][5], SP[5][5];	Declares two-dimensional arrays for storing path and shortest path matrices
SP[i][j]=MIN(SP[i][j], SP[i][k]+SP[k][j]);	Applies the modified Warshall's algorithm for generating the shortest path matrix values
printf("%d\t",SP[i][j]);	Prints the shortest path matrix values

9.6 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.

9.6.1 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. 9.6.

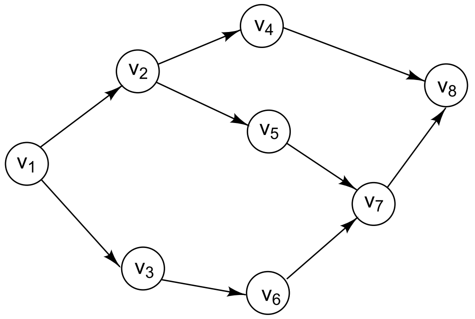


Fig. 9.6 *Graph traversal*

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 9.7 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.*

9.6.2 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. 9.6. 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 9.8 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:
  - o A set V of vertices or nodes where V = $\{v_1, v_2, v_3, ..., v_n\}$
  - o A set E of edges also called arcs where E = $\{e_1, e_2, e_3, ..., e_n\}$
- 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

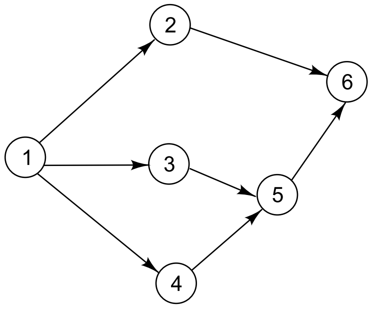
9.1 Which of the following is not true for graph?
(a) It is a set of vertices and edges.
(b) All of its vertices are reachable from any other vertex
(c) It can be represented with the help of an $N \times N$ matrix.
(d) All of the above are true

9.2 As per Warshall's method, which of the following is the correct relation for computing the path matrix?
(a) $P_{i,j} = P_{i,j}$ OR ($P_{i,k}$ AND $P_{k,j}$)
(b) $P_{i,j} = P_{i,j}$ AND ($P_{i,k}$ OR $P_{k,j}$)
(c) $P_{i,j} = P_{i,k}$ AND ($P_{k,j}$ OR $P_{i,j}$)
(d) None of the above

9.3 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?
(a) $SP_{i,j}$ = Minimum of ($SP_{i,j}$, $SP_{i,k} + SP_{k,j}$)
(b) $SP_{i,j}$ = Maximum of ($SP_{i,j}$, $SP_{i,k} + SP_{k,j}$)
(c) $SP_{i,j}$ = Minimum of ($SP_{i,k}$, $SP_{k,j} + SP_{i,j}$)
(d) None of the above

9.4 The number of edges incident on a vertex is referred as ____________.
(a) Degree
(b) Indegree
(c) Order
(d) Outdegree

9.5 Identify the BFS path for the following graph:
(a) 1–2–3–4–6–5
(b) 1–4–3–2–6–5
(c) 1–2–3–4–5–6
(d) None of the above



Review Questions

9.1 What is a graph? Explain with an example.
9.2 List and explain any five key terms associated with graphs.
9.3 What are the different methods of representing a graph?
9.4 What is an adjacency matrix? How can you derive a path matrix from an adjacency matrix?
9.5 Explain adjacency list implementation of a graph with the help of an example.
9.6 What is the significance of computing the shortest path in a graph? Explain with the help of an example.
9.7 Write the modified Warshall's algorithm for computing the shortest path between two nodes of a graph.
9.8 What is BFS? Explain with the help of an example.
9.9 What is DFS? Explain with the help of an example.

Programming Exercises

9.1 Write a C function to deduce the adjacency matrix for a given directed graph G.
9.2 Write a C function that takes as input the adjacency matrix and applies Warshall's algorithm to generate the corresponding path matrix.
9.3 Write a C program to implement a 3-node directed graph using adjacency list.
9.4 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

9.1 (b)	9.2 (a)	9.3 (a)	9.4 (b)	9.5 (c)

10

SORTING AND SEARCHING

Chapter Outline

10.1 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.

10.2 SORTING TECHNIQUES

Consider a list L containing n elements, as shown below.

$L_1, L_2, L_3, ..., L_n$

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

$L_1 \leq L_2 \leq L_3 .... \leq L_n$

Alternatively, if we arrange the list in descending order, then

$L_1 \geq L_2 \geq L_3 .... \geq L_n$

Example 10.1 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.*

10.2.1 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. 10.1.

18	3	2	33	21

List L

Fig. 10.1 *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 10.1 illustrates the sorting of list L in ascending order using selection sort.

Table 10.1 *Selection sort*

Pass	Comparison	Resultant Array
1	$\underbrace{18}$ 3 $\textcircled{2}$ 33 21	2 3 18 33 21
2	2 $\underbrace{\textcircled{3}}$ 18 33 21	2 3 18 33 21
3	2 3 $\underbrace{\textcircled{18}}$ 33 21	2 3 18 33 21
4	2 3 18 $\underbrace{33}$ $\textcircled{21}$	2 3 18 21 33
$\underbrace{\quad}$ → 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 10.2 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 10.3 Write a C program to perform selection sort on an array of N elements.

Program 10.1 implements selection sorting technique in C. It uses the algorithm depicted in Example 10.2.

Program 10.1 *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.

Chapter Ten

```
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
loc=Min(a,i,size);	Calls the *Min()* function to identify the location of the smallest element

temp=a[loc];
a[loc]=a[i];
a[i]=temp;
	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
So, total number of comparisons = (n–1) + (n–2) + ... + 1
= n * (n–1) / 2
= $O(n^2)$
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.*

10.2.2 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], …, A[i–1]. The following list describes the tasks performed in each of the passes:

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.

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], … 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. 10.1.

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 10.2 illustrates the sorting of list L in ascending order using insertion sort technique.

Table 10.2 *Insertion sort*

Pass	Comparison	Resultant Array
1	$\underbrace{18}\ \textcircled{3}\ 2\ 33\ 21$	3 18 2 21 33
2	$\underbrace{18\ 3}\ \textcircled{2}\ 33\ 21$	2 3 18 33 21
3	$\underbrace{2\ 3\ 18}\ \textcircled{33}\ 21$	2 3 18 33 21
4	$\underbrace{2\ 3\ 18\ 33}\ \textcircled{21}$	3 2 18 21 33
$\underbrace{\quad}\rightarrow$ denotes the previously sorted sub array ○ $\rightarrow$ 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 10.4 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 10.5 Write a C program to perform insertion sort on an array of N elements.

Program 10.2 implements insertion sorting technique in C. It uses the algorithm depicted in Example 10.4.

Program 10.2 *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
```

```
21

The sorted elements are:
2
3
18
21
33
```

Program analysis

Key Statement	Purpose
temp=array[i];	Stores the next element to be inserted, in the *temp* variable

for(j=i-1;j>=0;j—)
if(array[j]>temp)
array[j+1]=array[j];
else
break;
	Identifies the point of insertion for the element in the previously sorted sub array

array[j+1]=temp;	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.*

10.2.3 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. 10.1. Table 10.3 illustrates the sorting of list L in ascending order using bubble sort:

Table 10.3 *Bubble sort*

Pass	Comparison	Resultant Array
1	$\underbrace{18 \quad 3} \quad 2 \quad 33 \quad 21$ $3 \quad \underbrace{18 \quad 2} \quad 33 \quad 21$ $3 \quad 2 \quad \underbrace{18 \quad 33} \quad 21$ $3 \quad 2 \quad 18 \quad \underbrace{33 \quad 21}$	$3 \quad 2 \quad 18 \quad 21 \quad 33$
2	$\underbrace{3 \quad 2} \quad 18 \quad 21 \quad 33$ $2 \quad \underbrace{3 \quad 18} \quad 21 \quad 33$ $2 \quad 3 \quad \underbrace{18 \quad 21} \quad 33$	$2 \quad 3 \quad 18 \quad 21 \quad 33$
3	$\underbrace{2 \quad 3} \quad 18 \quad 21 \quad 33$ $2 \quad \underbrace{3 \quad 18} \quad 21 \quad 33$	$2 \quad 3 \quad 18 \quad 21 \quad 33$
4	$\underbrace{2 \quad 3} \quad 18 \quad 21 \quad 33$	$2 \quad 3 \quad 18 \quad 21 \quad 33$
$\underbrace{\qquad} \rightarrow$ 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 10.6 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 10.7 Write a C program to perform bubble sort on an array of N elements.

Program 10.3 implements bubble sorting technique in C. It uses the algorithm depicted in Example 10.6.

Program 10.3 *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
if(array[j]>array[j+1])	Compares the adjacent array elements in each pass

temp = array[j+1];
array[j+1] = array[j];
array[j] = temp;
	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
Total number of comparisons made = (n–1) + (n–2) + ... + 1
= n * (n – 1) / 2
= $O(n^2)$
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.*

10.2.4 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.

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

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:

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:

$$\underbrace{2 \quad 5} \quad \textcircled{34} \quad \underbrace{99 \quad 57 \quad 40}$$

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 10.8 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 10.9 Write a C program to perform quick sort on an array of N elements.

Program 10.4 implements quick sorting technique in C. It uses the algorithm depicted in Example 10.8.

Program 10.4 *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
pivot = array[left];	Initializes the pivot element

nxt_pvt=LB;
LB=left;
UB=right;
	Generates the next set of pivot, lower bound and upper bound values

Key Statement	Purpose

if(LB<nxt_pvt)
quick(array, LB, nxt_pvt-1);
if(UB>nxt_pvt)
quick(array, nxt_pvt+1, UB);
	Recursively calls the *quick*() function as per the next set of *pivot*, *UB* and *LB* 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^th pass = k*(n / k)

Now, total number of comparisons = O(n) + O(n) + O(n) +...+ m *terms*
= O(n * m)
= O(n log n)
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
Total number of comparisons = (n–1) + (n–2) + ... + 1
= n * (n–1) / 2
= $O(n^2)$
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.*

10.2.5 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.

Check Point

1. What is the best case efficiency of quick sort?
Ans. O(nlogn).
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.

Consider an array containing six elements, as shown below:

34 99 5 2 57 40 8 29

Figure 10.2 shows how merge sort is performed on the above list.

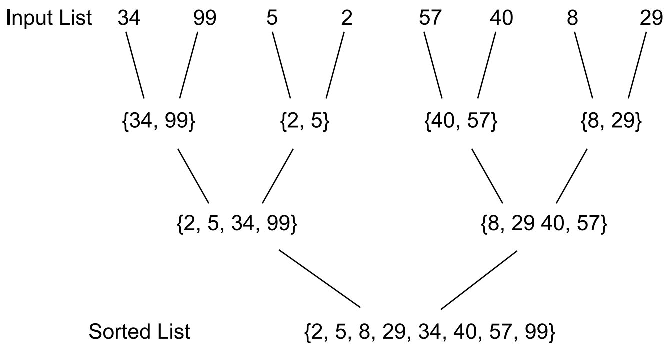


Fig. 10.2 *Merge sort*

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 10.10 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)
```

```
  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 10.11 Write a C program to perform merge sort on an array of N elements.

Program 10.5 implements merge sorting technique in C. It uses the algorithm depicted in Example 10.10.

Program 10.5 *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
mid = size/2;	Initializes the *mid* variable

mergesort(array, mid);
mergesort(array+mid, size-mid);
	Recursively calls the *mergesort*() function to sort the individual sub arrays

merge(array, mid, array+mid, size-mid);	Calls the merge 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(nlog*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.

Check Point

1. What is the efficiency of merge sort?
Ans. O(nlogn).
2. What is the most significant advantage of merge sort?
Ans. It always ensures an efficiency of O(nlogn).

Mind Jog

What is a stable sort?
It is that sorting medhodology which preserves the original order of the duplicate values in the final sorted list.

10.2.6 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. 10.3.

2	27	13	18	21	43	42	39	31	4

Fig. 10.3 *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 10.4 shows how these buckets are used for sorting the list.

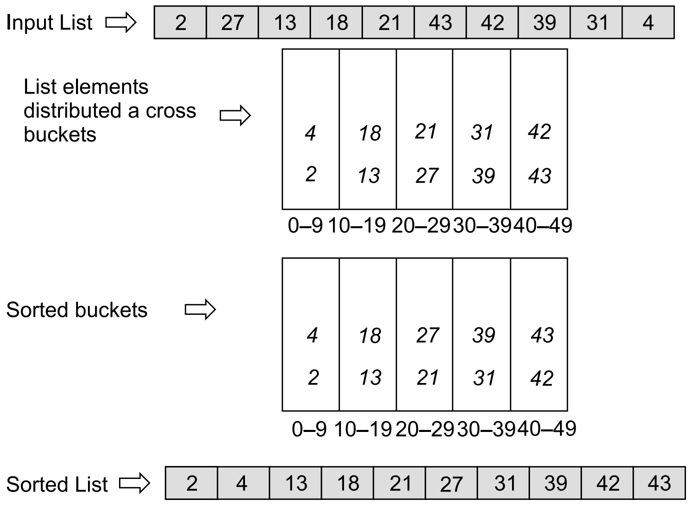


Fig. 10.4 *Bucket sort*

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 10.12 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 < size
  Step 6: if 0 <= arr[i] <=9 then goto Step 7 else goto Step 8
  Step 7: Set b[0][c[0]] = arr[i] and c[0] = c[0] + 1
  Step 8: if 10 <= arr[i] <=19 then goto Step 9 else goto Step 10
  Step 9: Set b[1][c[1]] = arr[i] and c[1] = c[1] + 1
  Step 10: if 20 <= arr[i] <=29 then goto Step 11 else goto Step 12
  Step 11: Set b[2][c[2]] = arr[i] and c[2] = c[2] + 1
  Step 12: if 30 <= arr[i] <=39 then goto Step 13 else goto Step 14
  Step 13: Set b[3][c[3]] = arr[i] and c[3] = c[3] + 1
  Step 14: if 40 <= arr[i] <=49 then goto Step 15 else goto Step 16
  Step 15: Set b[4][c[4]] = 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 10.13 Write a C program to perform bucket sort on an array of N elements.

Program 10.6 implements bucket sorting technique in C. It uses the algorithms depicted in Example 10.4 and Example 10.12.

Program 10.6 *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

for(i=0;i<5;i++)
 if(c[i]!=0)
 insertion(&b[i][0], c[i]);
	Repeatedly calls the *insertion*() function to perform insertion sort on each of the buckets

for(j=0;j<c[i];j++)
 array[k++]=b[i][j];
	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.

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.

10.3 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.

10.3.1 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 10.5 depicts the linear search technique performed on an array of integers.

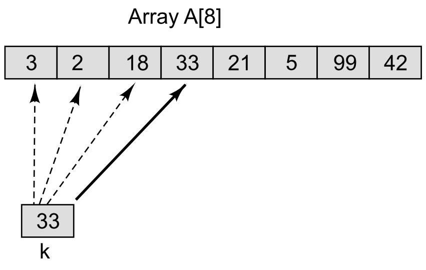


Fig. 10.5 *Linear search*

As shown in Fig. 10.5, 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.

Example 10.14 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 10.15 Write a C program to perform linear search on an array of N elements.

Program 10.7 implements linear search technique in C. It uses the algorithm depicted in Example 10.14.

Program 10.7 *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
index=linear(arr,N,k);	Calls the *linear()* function to perform linear search on the array *arr*

for(i=0;i<size;i++)
if(num==array[i])
	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*.

Thus, efficiency of linear search = $(1 + 2 + \ldots + n) / n$
$= n\,(n+1) / 2n$
$= O(n)$

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.

10.3.2 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. It 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. 10.6. Let k = 21 be the value that needs to be searched.

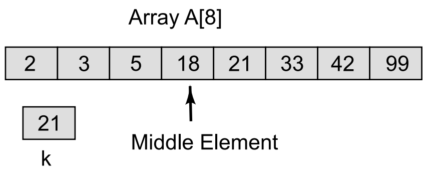


Fig. 10.6 *Binary search*

As we can see in Fig. 10.6, 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}.

4. Now, again k is compared with the middle element of {21, 33}, which is 21. Thus, it matches with k.
5. The index value of 21, i.e., 4 is returned and the search is considered as successful.

Example 10.16 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 10.17 Write a C program to perform binary search on an array of N elements.

Program 10.8 implements binary search technique in C. It uses the algorithm depicted in Example 10.16.

Program 10.8 *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

  Enter the element to be searched:
  21

  Element 21 is stored at index location 4 in the array arr[8]
```

We must ensure that we input elements in a sorted manner as that is the prerequisite for performing binary search.

Chapter Ten

Program analysis

Key Statement	Purpose
index=binary(arr,N,k);	Calls the *binary()* function to perform binary search on the array *arr*

if(array[k]==num)
return(k);
	Returns the value of *k* in case of successful search

else if(array[k]<num)
i=k+1;
else
j=k-1;
	Updates the values of *i* and *j* 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 = log2n
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.

Check Point

1. What is the worst case efficiency of binary search?
Ans. O(log2n).
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.

10.3.3 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 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 10.7 depicts the hash functionality.

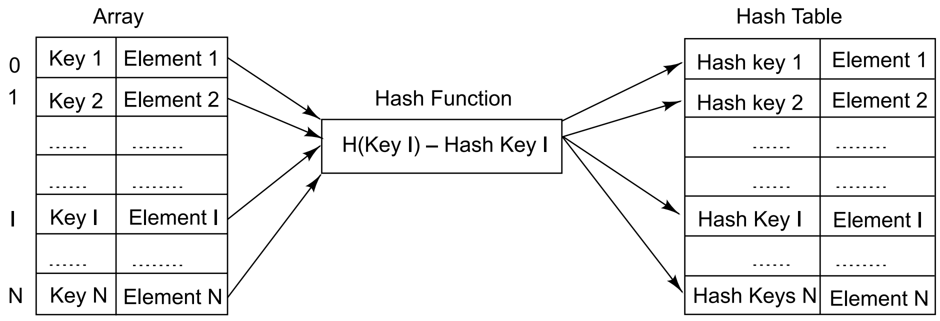


Fig. 10.7 *Hashing*

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. 10.8.

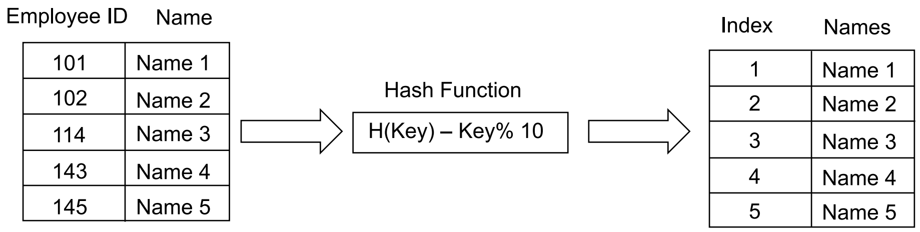


Fig. 10.8 *Generating hash keys*

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. 10.5. 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 10.18 Write a program to implement a hash function. Assume that the input keys are within the range 10001 and 10999.

Program 10.9 *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
hk = hash(key);	Calls the hash() function to generate the hash key value

Solved Problems

Problem 10.1 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

Initial array	35 18 7 12 5 23 16 3 1
Pass 1	1 35 18 12 7 23 16 5 3
Pass 2	1 3 35 18 12 23 16 7 5
Pass 3	1 3 5 35 18 23 16 12 7
Pass 4	1 3 5 7 35 23 18 16 12
Pass 5	1 3 5 7 12 35 23 18 16
Pass 6	1 3 5 7 12 16 35 23 18
Pass 7	1 3 5 7 12 16 18 35 23
Pass 8	1 3 5 7 12 16 18 23 35 (sorted array)

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.

Problem 10.2 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

Initial array	74 39 35 32 97 84
Pass 1	32 39 35 74 97 84
Pass 2	32 35 39 74 97 84
Pass 3	32 35 39 74 97 84
Pass 4	32 35 39 74 97 84
Pass 5	32 35 39 74 84 97 (sorted array)

Problem 10.3 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.

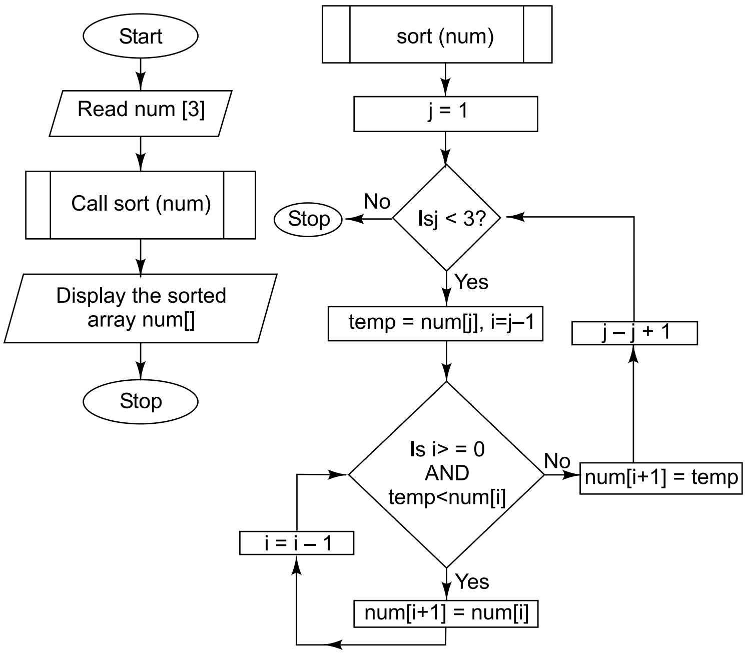
Solution

Initial Array	35 54 12 18 23 15 45 38
Pass 1	18 54 12 35 23 15 45 38
Pass 2	18 15 12 35 23 54 45 38
Pass 3	12 15 18 35 23 54 45 38
Pass 4	12 15 18 35 23 54 45 38
Pass 5	12 15 18 35 23 54 45 38
Pass 6	12 15 18 54 23 35 45 38
Pass 7	12 15 18 38 23 35 45 54
Pass 8	12 15 18 23 38 35 45 54

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 10.4 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

10.1 Which of the following is the fastest searching technique?
(a) Bubble (b) Quick
(c) Insertion (d) Bucket

10.2 Which of the following searching techniques mandatorily requires the list to be already sorted?
(a) Linear (b) Binary
(c) Hash (d) None of the above

10.3 Which of the following does not have an efficiency of O(n^2)?
(a) Selection (b) Insertion
(c) Bubble (d) Merge

10.4 What is the worst case efficiency of bucket sorting technique?
(a) $O(n^2)$ (b) $O(n)$
(c) $O(n)$ (d) $O(n\log n)$

10.5 What is the worst case efficiency of binary search technique?
(a) $O(\log_2 n)$ (b) $O(n)$
(c) $O(n\log_2 n)$ (d) $O(1)$

10.6 Pivot element is associated with which of the following?
(a) Binary search (b) Quick sort
(c) Selection sort (d) Hashing

10.7 Which of the following searching techniques is most suitable for large databases?
(a) Hashing (b) Linear
(c) Binary (d) All of the above

10.8 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

10.1 What is sorting? List the various sorting techniques.
10.2 What is searching? List the various searching techniques.
10.3 Why quick sort is considered the fastest sorting technique?
10.4 What is a pivot element? Where is it used?
10.5 Deduce the worst case efficiency of binary search technique.
10.6 What is bubble sorting? What is it considered to be slow?
10.7 Explain the insertion and selection sorting techniques with examples.
10.8 What is hashing? Explain with example.
10.9 What is a collision? How can it be prevented?
10.10 Explain the role of buckets in bucket sorting technique.

Programming Exercises

10.1 Write a program in C that takes as input five integers and displays them in a sorted sequence.
10.2 Write a C function that applies the bubble sorting technique to sort a set of alphanumeric characters as per their ASCII values.
10.3 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.
10.4 Write a C function that performs linear search on an array of real values.
10.5 Write a C program that sorts the given set of integers and performs binary search on them.

Answers to Multiple-Choice Questions

10.1 (a)	10.2 (b)	10.3 (d)	10.4 (a)	10.5 (a)
10.6 (b)	10.7 (a)	10.8 (a)		

11

APPLICATION OF DATA STRUCTURES

Chapter Outline

11.1 INTRODUCTION

In the previous chapters, we learned about various types of data structures, such as stacks, queues, linked lists, trees and graphs. In this chapter, we will see how these data structures are actually put into use for solving mathematical and other real-world problems.

11.2 APPLICATION OF STACKS

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. The various applications of stacks are:

1. Infix to Postfix Conversion
2. Postfix Evaluation
3. Tree Traversal

11.2.1 Infix to Postfix Conversion

Problem 11.1 Write a program in C that uses the stack data structure to convert an infix expression into postfix expression. The program should also evaluate the value of the resultant postfix expression.

Program 11.1 *Using stack data structure to convert an infix expression into postfix expression*

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

  int stack[100];
  int top=-1;

  void in2post(char []); /*Function prototype for converting infix expression
to postfix expression*/
  void push(int); /*Function prototype for pushing an element into a stack */
  int pop(); /*Function prototype for removing an element from a stack */
  int prec(char); /*Function prototype for determining precedence of
operators*/
  int cal(char []);/*Function prototype for calculating the value of a postfix
expression*/

  void main()
  {
   char in[100],post[100];
   clrscr();
   printf("Enter an infix expression:  ");
   gets(in);
   in2post(in);
   getch();
  }
```

```
void in2post(char in_exp[])
{
 int x=0,y=0,z,result=0;
 char a,c, post_exp[100];
 char t;
 push("\0");
 t=in_exp[x];
 while(t!='\0')
 {
  if(isalnum(t))
  {
   post_exp[y]=t;
   y++;
  }
  else if(t=='(')
  {
   push('(');
  }
  else if(t==')')
  {
   while(stack[top]!='(')
   {
    c=pop();
    post_exp[y]=c;
    y++;
   }
   c=pop();
  }
  else
  {
   while(prec(stack[top])>=prec(t))
   {
    c=pop();
    post_exp[y]=c;
    y++;
   }
   push(t);
  }
  x++;
  t=in_exp[x];
 }

 while(top!=-1)
 {
  c=pop();
  post_exp[y]=c;
  y++;
 }
```

```
   printf("\nThe equivalent postfix expression is: ");

   for(z=0;z<y;z++)
    printf("%c",post_exp[z]);
   printf("\n\nDo you want to evaluate the postfix expression? (Y/N): ");
   scanf("%c",&a);

   if(a=='y' || a=='Y')
   {
    result=cal(post_exp);
    printf("\nResult =  %d\n",result);
    getch();
   }

   else if(a=='n' || a=='N')
   {
    exit(0);
   }
 }


 int cal(char post[])
 {
  int m,n,x,y,j=0,len;
  len=strlen(post);
  while(j<len)
  {
   if(isdigit(post[j]))
   {
    x=post[j]-'0';
    push(x);
   }
   else
   {
    m=pop();
    n=pop();

    switch(post[j])
    {
     case '+':x=n+m;
     break;
     case '-':x=n-m;
     break;
     case '*':x=n*m;
     break;
     case '/':x=n/m;
     break;
    }
    push(x);
    }
    j++;
   }
  if(top>0)
  {
   printf("Discrepancy between number of operators and operands.");
   exit(0);
  }
  else
  {
   y=pop();
   return (y);
  }
   return 0;
}

int prec(char t)
{
 switch(t)
 {
  case '(':return (10);
  case ')':return (9);
  case '+':return (7);
  case '-':return (7);
  case '*':return (8);
  case '/':return (8);
  case '\0':return (0);
  default: printf("Invalid Expression!");
  break;
 }
 return 0;
}


void push(int n)
{
 if(top==99)
 {
  printf("\n\n\tStack Full!");
  getch();
  exit(1);
 }
 stack[++top]=n;
}


int pop()
{
 if(top==-1)
 {
  getch();
 }
 return(stack[top—]);
}
```

Output

```
Enter an infix expression: 1+2*3

The equivalent postfix expression is: 123*+

Do you want to evaluate the postfix expression? (Y/N): y

Result = 7
```

11.2.2 Tree Traversal using Stacks

Problem 11.2 Write a function in C that uses the stack data structure to determine the preorder traversal path in a binary tree.

```
void preorder(node *r)
{
 node * ptr;
 top=0;
 stack[top]=-1;
 ptr=r;
 while(ptr!=NULL)
 {
  printf("%c—>",ptr->INFO);
  if(ptr->RIGHT!=NULL)
  {
   top=top+1;
   stack[top]=ptr->RIGHT;
  }

  if(ptr->LEFT!=NULL)
   ptr=ptr->LEFT;
  else
  {
   ptr=stack[top];
   top=top-1;
  }
 }
 printf("END");
}
```

11.3 APPLICATION OF QUEUES

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 or 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.

Queue data structure is mainly used by operating systems and system programs for addressing situations that require adherence to FIFO principle.

Problem 11.3 List down the key application areas of queue data structure.

Solution Queues are typically used to implement the following:

1. CPU scheduling
2. Disk scheduling
3. IO buffer
4. Priority queue

Note *For more information on implementing priority queue, refer to Chapter 7.*

11.4 APPLICATION OF LINKED LISTS

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. Linked lists are used for implementing various data structures, such as stacks, queues, trees, etc.

11.4.1 Representing Polynomials using Linked Lists

Problem 11.4 Explain how linked lists can be used for representing polynomial expressions.

Solution A linked list can be used for representing a polynomial expression by storing polynomial terms in the form of interconnected list nodes. Each node contains the following:

1. Variable
2. Exponent
3. Coefficient

Addition of two polynomials is carried out by adding the nodes having equivalent exponents and storing the result in a separate linked list.

The following structure declaration represents a polynomial term:

```
struct p_term
{
    int coeff;
    int pow;
    struct p_term* next;
};
```

11.4.2 Representing Other Data Structures using Linked Lists

While linked list is itself a data structure, it is also used for implementing other data structures. One of the key features of a linked list is that it uses dynamic memory management technique for storing data. This makes linked list-based data structures more efficient as compared to static array-based data structures.

Note *To see how linked lists are used for implementing stacks, queues and trees data structures refer to Chapters 6, 7 and 8 respectively.*

11.5 APPLICATION OF TREES

Tree is a non-linear data structure which stores the data elements in a hierarchical manner. 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. One of the most widely used tree data structures is the binary search tree. It 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.

11.5.1 Using Binary Search Tree to Sort a List

Problem 11.5 An array of integers is given. Use the binary search tree to sort the list of integers.

Solution To sort a list using binary search tree, perform the following steps:

1. Read the list elements and create the binary search tree.
2. Read the tree in inorder sequence to generate the sorted list.

Creating Binary Search Tree

The following code snippet shows how an array is used as an input for creating a binary search tree:

```
.
.

struct BST
{
  int INFO;
  struct node *LEFT, *RIGHT;
};

typedef struct BST node;


void main()
{
.
.
struct node *root = NULL;
int i;
int list[10]={77,44,13,2,3,89,100,32,24,19};
for(i=0;i<10;i+=)
 root = insert(root,list[i]);
.
.
}
```

```
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)
   ;
 return(r);
}
```

Generating Sorted List The following code snippet shows how a binary search tree is traversed in inorder fashion to generate the sorted list of integers.

```
.
.
i=0;
.
void main()
{
.
inorder(root);

printf("Sorted List:");
for(i=0;i<10;i+=)
 printf("\nlist[%d]=%d",i,list[i]);
.
}

void inorder(node *r)
{
 if(r!=NULL)
 {
  inorder(r->LEFT);
  list[i]=r->INFO;
  i=i+1;
  inorder(r->RIGHT);
 }
}
```

11.5.2 Using Heap Tree Structure to Perform Sorting

A heap is a complete binary tree with each node of the tree satisfying the property that the node value is either greater (or less) than all the nodes in its subtree. A maxheap is obtained when all the nodes are greater than their respective subtree nodes. Similarly, a minheap is obtained when all the nodes are less than their respective subtree nodes.

Figure 11.1 shows a maxheap.

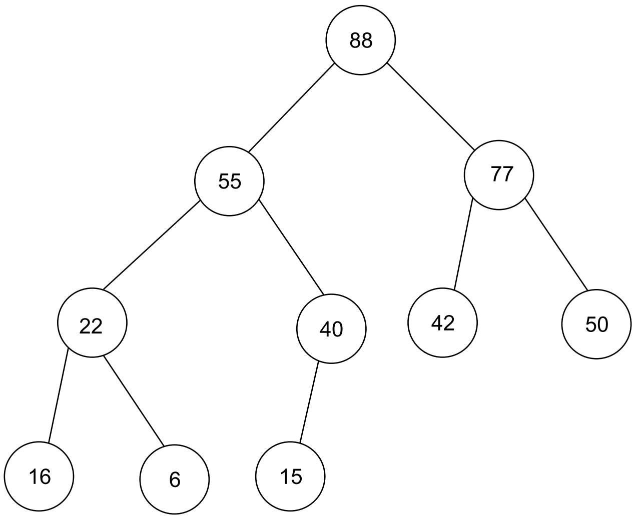


Fig. 11.1 *Maxheap*

Problem 11.6 Use the heap tree structure to sort a list of integers.

Solution The process of using the heap tree structure to generate a sorted list of integers is called heap sort. Heap sort is performed by first building a heap and then removing the top element of the heap one after the other to generate the sorted list.

The following code snippet shows how an element is inserted and stored at an appropriate place in the heap.

```
  void insert(void)
  {
   int parent,ptr;
   n=n+1; /*n is the number of elements in the heap*/
   ptr=n;
   while(ptr>1)
   {
    parent=ptr/2;
    if(element<=heap[parent]) /*element is the new element being inserted
in the heap*/
    {
     heap[ptr]=element; /*heap is a linear array that stores heap elements*/
     return;
    }
    heap[ptr]=heap[parent];
    ptr=parent;
   }
   heap[1]=element; /*Here, element is inserted at the root position of the
heap*/
   return;
  }
```

After a maxheap is created, the top node is removed one by one and placed in an array. Once the heap is empty, the array elements are retrieved to obtain the sorted list.

11.6 APPLICATION OF GRAPHS

Graph data structure 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, etc.

Problem 11.7 A directed, acyclic graph G = (V, E) is given. Use the topological sorting technique to create a linear order of vertices such that for all edges $E_{i,j}$ in the graph vertex i is placed to the left of vertex j.

Solution The following algorithm shows how to perform topological sorting:

```
//LIST - Is a list that is used for storing the sorted elements
//E' - Is a set of nodes that do not contain any incoming edges

while (E' != NULL)
{
 i = retreive (E')
 insert (L,i)

 for ($E_{x,y}$) //For all edges of the tree from node x to node y
 {
  Delete $E_{x,y}$
  if x has no more incoming edges
   insert x into E'
 }
}
if ($E_{x,y}$ = EMPTY) //If there are no edges in the graph
 Display LIST as the topologically sorted list
else
 Throw error (graph is cyclic)
```

Summary

- In real programming situations, various instances of these data structures are used in conjunction to obtain the desired results.
- Performance plays a key role in such situations and the right choice of a data structure makes all the difference.

Index