Data Structures Using 'C'

SECOND YEAR DIPLOMA

(Code : 22317)

Maharashtra State Board of Technical Education (MSBTE)

Semester III – Computer Engineering Program Group
(CO / CM / CW / IF)

Strictly as per new revised 'I' Scheme w.e.f. academic year 2018-2019

Dilip Kumar Sultania

B.Tech.(hons.) Computer Science and Engineering
I.I.T. , Kharagpur.

TechKnowledge™ Publications

MDO15A Price ₹ 215/-

Data Structures Using 'C'

Dilip Kumar Sultania

Second Year Diploma : Semester III (MSBTE)

[Computer Engineering Program Group (CO / CM / CW / IF)]

Copyright © by Author. All rights reserved. No part of this publication may be reproduced, copied, or stored in a retrieval system, distributed or transmitted in any form or by any means, including photocopy, recording, or other electronic or mechanical methods, without the prior written permission of the publisher.

This book is sold subject to the condition that it shall not, by the way of trade or otherwise, be lent, resold, hired out, or otherwise circulated without the publisher's prior written consent in any form of binding or cover other than which it is published and without a similar condition including this condition being imposed on the subsequent purchaser and without limiting the rights under copyright reserved above.

First Printed in India : July 2007 **(Pune University)**
First Edition : June 2018 **(As per 'I' Scheme)**
Second Revised Edition : June 2019 **(TechKnowledge Publications)**

This edition is for sale in India, Bangladesh, Bhutan, Maldives, Nepal, Pakistan, Sri Lanka and designated countries in South-East Asia. Sale and purchase of this book outside of these countries is unauthorized by the publisher.

Printed at : 37/2, Ashtavinayak Industrial Estate, Near Pari Company,
Narhe, Pune, Maharashtra State, India.
Pune – 411041

ISBN : 978-93-89233-02-5

Published by
TechKnowledge Publications
Head Office : B/5, First floor, Maniratna Complex, Taware Colony, Aranyeshwar Corner,
Pune - 411 009. Maharashtra State, India
Ph : 91-20-24221234, 91-20-24225678.
Email : info@techknowledgebooks.com,
Website : www.techknowledgebooks.com

[22317] (FID : MDO15) (New Book Code : MDO15A)

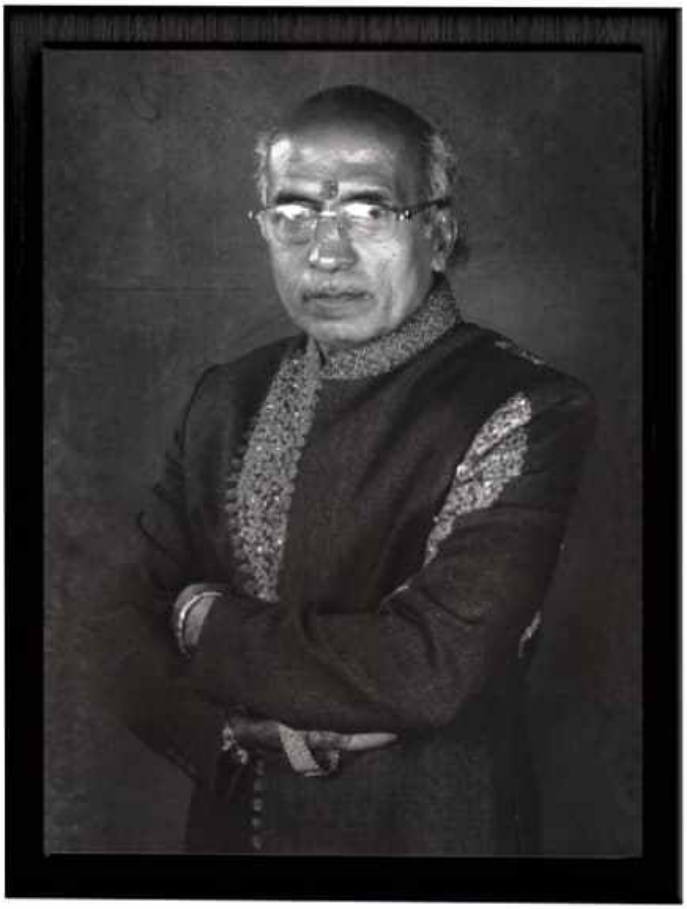
We dedicate this Publication soulfully and wholeheartedly,

in loving memory of our beloved founder director

Late Shri. Pradeepsheth Lalchandji Lunawat,

who will always be an inspiration, a positive force and strong support

behind us.



Lt. Shri. Pradeepji L. Lunawat

Soulful Tribute and Gratitude for all Your

Sacrifices, Hardwork and 40 years of Strong Vision...

Preface

My Dear Students,

I am extremely happy to come out with this book on "Data Structures Using 'C' " for you. The topics within the chapters have been arranged in a proper sequence to ensure smooth flow of the subject.

A large number of solved examples have also been included. Therefore, we are sure that this book will cater all your needs for this subject.

I present this book in the loving memory of **Late Shri. Pradeepji Lunawat,** our source of inspiration and a strong foundation of **"TechKnowledge Publications"**. He will always be remembered in our heart and motivate us to achieve our milestone.

I am also thankful to the staff members of TechKnowledge Publications and others for their efforts to make this book as good as it is. I have jointly made every possible efforts to eliminate all the errors in this book. However if you find any, please let me know, because that will help me to improve further.

I am also thankful to my family members and friends for patience and encouragement.

- Dilip Kumar Sultania

□□□

Syllabus

Unit	Unit Outcomes (UOs) (in cognitive domain)	Topics and Sub-topics
Unit - I **Introduction to Data Structures** **Refer chapter 1**	1a. Classify the given type of Data Structures based on their characteristics. 1b. Explain complexity of the given algorithm in terms of time and space. 1c. Explain the given operations to be performed on the given type of data structures.	1.1 Concept and need of DS, Abstract Data Type 1.2 Types of Data Structures : (i) Linear Data Structures (ii) Non-Linear Data Structures 1.3 Algorithm Complexity : (i) Time (ii) Space 1.4 Operations on Data Structures: (i) Traversing, (ii) Searching, (iii) Insertion, (iv) Deletion, (v) Sorting
Unit - II **Searching and Sorting** **Refer chapter 2**	2a. Explain working of the given search method with an example. 2b. Write an algorithm to search the given key using binary Search method. 2c. Write an Algorithm to sort data using a specified sorting method. 2d. Explain the working of given sorting method step-by-step with an example and small data set.	2.1 Searching : searching an item in a data set using following methods : (i) Linear Search, (ii) Binary Search 2.2 Sorting : sorting of data set in an order using following methods : (i) Bubble Sort (ii) Selection Sort (iii) Insertion Sort (iv) Quick Sort (v) Radix Sort.
Unit - III **Stacks and Queues** **Refer chapters 3 & 4**	3a. Develop an algorithm to perform PUSH and POP operations for the given item in a stack. 3b. Convert the given expression from Infix to Prefix/Postfix using Stack. 3c. Write steps to evaluate the given expression using the stack. 3d. Develop a program to perform the given operation on a linear Queue. 3e. Write Algorithm to perform the given operations on circular queue.	3.1 Introduction to Stack – Stack representation in memory using array – Stack as an ADT – Stack Operations – PUSH, POP – Stack Operations conditions – Stack Full / Stack Overflow, Stack Empty / Stack Underflow. – Applications of Stack o Reversing a list o Polish notations 3.2 Conversion of infix to postfix expression, Evaluation of postfix expression, Converting an infix into prefix expression, Evaluation of prefix expression, Recursion, Tower of Hanoi 3.3 Introduction to Queue : – Queue representation in memory using array – Queue as an ADT

Unit	Unit Outcomes (UOs) (in cognitive domain)	Topics and Sub-topics
		– Types of Queues : Linear Queue, Circular Queue, Concept of Priority queue – Queue Operations – INSERT, DELETE – Queue Operations Conditions – Queue Full, Queue Empty – Applications of Queue
Unit - IV **Linked List** Refer chapter 5	4a. Create relevant structure to represent the given node using linked list. 4b. Develop algorithm to insert the given item in linear linked list. 4c. Develop algorithm to delete the given item from linear linked list 4.d. Develop algorithm to traverse a circular linked list.	4.1 Introduction to Linked List Terminologies : node, Address, Pointer, Information field/Data field, Next pointer, Null Pointer, Empty list. 4.2 Type of lists : Linear list, Circular list 4.3 Operations on a singly linked list : Traversing a singly linked list, Searching a key in linked list, Inserting a new node in a linked list, Deleting a node from a linked list.
Unit - V **Trees and Graphs** Refer chapters 6 & 7	5a. Draw Binary Search Tree for the given data set. 5b. Write algorithms to traverse the tree using the given method. 5c. Construct Expression tree for the given data. 5d. Represent the given Graph using adjacency matrix and adjacency list.	5.1 Introduction to Trees Terminologies : tree, degree of a node, degree of a tree, level of a node, leaf node, Depth/Height of a tree, In-degree and out-Degree, Path, Ancestor and descendant nodes 5.2 Tree Types and Traversal methods Types of Trees : General tree, Binary tree, Binary search tree (BST). Binary tree traversal : In order traversal, Preorder traversal, Post order traversal 5.3 Expression tree. 5.4 Introduction to Graph terminologies : graph, node (Vertices), arcs (edge). Directed graph, undirected graph, in-degree, out-degree, adjacent, successor, predecessor, relation, path, sink, articulation point. 5.5 Adjacency List, Adjacency Matrix of directed/undirected graph.

Suggested Practicals /Exercises

Sr. No.	Practical Outcomes (PrOs)	Unit No.	Approx. Hrs Required
1.	Implement a 'C' program for performing following operations on Array : Creation, Insertion, Deletion, Display	I	02*
2.	Implement a 'C' program to search a particular data from the given Array using : (i) Linear Search,	II	02*
3.	Implement a 'C' program to search a particular data from the given Array using Binary Search	II	02*
4.	Implement a 'C' program to sort an array using following methods : (i) Bubble Sort, (ii) Selection Sort (iii) Insertion Sort	II	02*
5.	Implement a 'C' program to sort an array using following methods : (ii) Selection	II	02
6.	Implement a 'C' program to sort an array using following methods : (iii) Insertion Sort	II	02
7.	Write C program to perform PUSH and POP operations on stack using array.	III	02*
8.	Write C program to perform INSERT and DELETE operations on Linear Queue using array. Part - I	III	02
9.	Write C program to perform INSERT and DELETE operations on Linear Queue using array. Part - II	III	02
10.	Write C program to perform INSERT and DELETE operations on Circular Queue using array, Part - I	III	02
11.	Write C program to perform INSERT and DELETE operations on Circular Queue using array. Part - II	III	02
12.	Write C program to perform the operations (Insert, Delete, Traverse, and Search) on Singly Linked List, Part - I	IV	02*
13.	Write C program to perform the operations (Insert, Delete, Traverse, and Search) on Singly Linked List. Part - II	IV	02
14.	Write C program to perform the operations (Insert, Delete, Traverse, and Search) on Circular Singly Linked List. Part - I	IV	02*
15.	Write C program to perform the operations (Insert, Delete, Traverse, and Search) on Circular Singly Linked List, Part - II	IV	02
16.	Write C program to Implement BST (Binary Search Tree) and traverse the tree (Inorder, Preorder, Postorder)	V	02*
	Total		**32**

Note : The Practicals marked as ' * ' are compulsory.

❑❑❑

INDEX

UNIT I

Chapter 1 : Introduction to Data Structure 1-1 to 1-18

Syllabus : Concept and need of DS, Abstract Data Type.

Types of Data Structures :
(i) Linear Data Structures, (ii) Non-Linear Data Structures

Algorithm Complexity : (i) Time, (ii) Space

Operations on Data Structures :
(i) Traversing, (ii) Searching, (iii) Insertion,
(iv) Deletion, (v) Sorting

1.1 Data Representation 1-1
1.1.1 Data 1-1
1.1.2 Data Types 1-1
1.1.3 Data Types in 'C' 1-2
1.1.4 User-defined Type 1-3
1.1.5 Derived Data Types 1-3
1.1.6 Structure 1-3
1.1.7 Union 1-3
1.2 Abstract Data Types (ADT) 1-4
1.3 Concept of Data Structures 1-5
1.3.1 Need of Data Structure 1-5
1.3.2 Linear and Non-Linear 1-6
1.4 Classification of Data Structure 1-7
1.4.1 Primitive and Non-Primitive 1-8
1.5 Operations on Data Structures 1-8
1.6 Introduction to Algorithm 1-9
1.7 Different Approaches for Designing an Algorithm 1-11
1.7.1 Top-down Approach 1-11
1.7.2 Bottom-up Approach 1-11
1.8 Complexity in Terms of Time and Space 1-11
1.8.1 Algorithm Analysis 1-11
1.8.2 Measuring the Running Time of a Program (Time Complexity) 1-13
1.8.3 Measurement of Growth Rate (Asymptotic Growth Rate) 1-13
1.8.3(A) Asymptotic Consideration 1-13
1.8.3(B) Constant Factor in Complexity Measure 1-14
1.9 The Notation O : (pronounced as big-oh), ($O(n^2)$ is pronounced as big-oh of n^2)] 1-14
1.10 MSBTE Questions and Answers 1-15

UNIT II

Chapter 2 : Searching and Sorting 2-1 to 2-44

Syllabus :

Searching : searching an item in a data set using following methods : (i) Linear Search, (ii) Binary Search

Sorting : sorting of data set in an order using following methods : (i) Bubble Sort, (ii) Selection Sort (iii) Insertion Sort, (iv) Radix Sort.

2.1 Sorting 2-1
2.1.1 Sort Stability 2-2
2.1.2 Sort Efficiency 2-2
2.1.3 Passes 2-3
2.2 Bubble Sort 2-3
2.3 Selection Sort 2-6
2.4 Insertion Sort 2-8
2.4.1 Sorting an Array of Strings using Insertion Sort 2-10
2.4.2 Sorting an Array of Records on the given Key using Insertion Sort 2-11
2.5 Radix Sort 2-15
2.5.1 Algorithm for Radix Sort 2-16
2.5.2 C - Function for Radix Sort 2-16
2.5.3 Analysis of Radix Sort 2-18
2.6 Quick Sort 2-22
2.6.1 Picking a Pivot 2-23
2.6.2 Partitioning 2-23
2.6.3 Running Time of Quick Sort 2-29
2.6.3(A) Worst-case Analysis 2-29
2.6.3(B) Best-case Analysis 2-29
2.6.3(C) Average-case Analysis 2-30
2.6.4 Advantages and Disadvantages of Quick Sort 2-30
2.7 Comparison of Sorting Algorithms 2-30

2.7.1 Best-case, Worst-case and Average-case Analysis of Sorting Algorithm 2-31
2.7.2 Compare Quick Sort and Radix Sort 2-32
2.8 Searching 2-32
2.8.1 Sequential Search (Linear Search) 2-33
2.8.2 Sequential Search on a Sorted Array 2-34
2.8.3 Binary Search 2-34
2.8.4 Difference between Linear Search and Binary Search 2-40
2.9 MSBTE Questions and Answers 2-41

UNIT III

Chapter 3 : Stacks 3-1 to 3-46

Syllabus :

Introduction to Stack :

Stack representation in memory using array

Stack as an ADT

Stack Operations : PUSH, POP

Stack Operations Conditions : Stack Full/Stack Overflow, Stack Empty/Stack Underflow.

Applications of stack

- Reversing a Stack
- Polish notations

Conversion of infix to postfix expression, Evaluation of postfix expression, Converting an infix into prefix expression, Evaluation of prefix expression Recursion, Tower of Hanoi.

3.1 Introduction to Stack 3-1
3.2 Stack as an ADT 3-2
3.3 Operations on Stacks 3-2
3.4 Array Representation 3-3
3.4.1 'C' Functions for Primitive Operations on a Stack 3-3
3.5 Program Showing Stack Operations 3-5
3.5.1 Operations on Stack Considering Overflow and Underflow [Array Implementation] 3-7
3.5.2 Reversing a List 3-9
3.6 Applications of Stack 3-9
3.6.1 Expression Representation 3-9
3.6.2 Evaluation of a Postfix Expression using a Stack 3-11
3.6.3 Evaluation of a Prefix Expression 3-16
3.6.4 Conversion of an Expression from Infix to Postfix 3-18
3.6.5 Conversion of an Expression from Infix to Prefix 3-26
3.7 Expression Conversion (A Fast Method) 3-30
3.7.1 Infix to Postfix 3-30
3.7.2 Infix to Prefix 3-30
3.7.3 Postfix to Prefix 3-31
3.7.4 Prefix to Infix 3-33
3.8 Introduction to Recursion 3-36
3.8.1 Converting a Recursive Function to an Equivalent C-function 3-37
3.8.1(A) Finding Factorial of an Integer Number 3-37
3.8.2 Tower of Hanoi Problem 3-39
3.8.3 Stack in Recursive Function 3-41
3.9 MSBTE Questions and Answers 3-42

Chapter 4 : Queues 4-1 to 4-23

Syllabus :

Introduction to Queue :

Queue representation in memory using array

Queue as an ADT

Types of Queues : Linear Queue, Circular Queue, Concept of Priority queue.

Queue Operations : INSERT, DELETE

Queue Operations Conditions : Queue Full, Queue Empty

Applications of Queue.

4.1 Introduction 4-1
4.1.1 Array Representation and Implementation of Queues 4-1
4.1.2 Comparison between Array Representation and the Linked Representation of a Queue 4-2
4.1.3 Comparison between STACK and QUEUE 4-3
4.2 Operations on Queue (Implemented using Array) 4-3
4.2.1 Queue as an ADT 4-7
4.3 Types of Queue 4-8
4.3.1 Circular Queue (Array) 4-8
4.3.2 Implementation of a Circular Movement inside a Linear Array 4-9
4.4 Priority Queue 4-13
4.4.1 Implementation of Priority Queues 4-14
4.5 Applications of Queue 4-18
4.5.1 Josephus Problem 4-18

4.5.2 Job Scheduling 4-19
4.5.3 Queue Simulation 4-20
4.6 MSBTE Questions and Answers 4-20

UNIT IV

Chapter 5 : Linked List 5-1 to 5-27

Syllabus :

Introduction to Linked List Terminologies : node, Address, Pointer, Information field/Data field, Next pointer, Null Pointer, empty list.

Type of lists : Linear list, Circular list

Operations on a singly linked list : Traversing a singly linked list, Searching a key in linked list, Inserting a new node in a linked list, Deleting a node from a linked list.

5.1 Representation and Implementation of Singly Linked Lists 5-1
5.1.1 Comparison between Array and Linked Lists 5-1
5.1.2 Representation 5-2
5.1.3 Implementation 5-2
5.1.4 Types of Linked List 5-4
5.1.4(A) Singly Linked List 5-4
5.1.4(B) Doubly Linked List 5-4
5.1.4(C) A Circular Linked List 5-5
5.2 Basic Linked List Operations 5-6
5.2.1 Creating a Linked List 5-6
5.2.2 Traversing a Linked List 5-8
5.2.3 Counting Number of Nodes in a Linked List through Count Function 5-8
5.2.4 Printing a List through Print Function 5-9
5.2.5 Inserting an Item 5-10
5.2.5(A) Inserting an Item at the End of a Linked List 5-11
5.2.5(B) Inserting a Data 'x' at a Given Location 'LOC' in a Linked List, Referenced by 'Head' 5-11
5.2.5(C) Inserting an Element in a Priority Linked List 5-12
5.2.6 Deleting an Item 5-13
5.2.6(A) Deletion of the Last Node of a Linked List 5-14
5.2.6(B) Deletion of a Node at Location 'LOC' from a Linked List 5-15
5.2.6(C) Delete a Linked List, Referenced by the Pointer Head 5-16
5.2.7 Concatenation of Two Linked Lists 5-16
5.2.8 Inversion of Linked List 5-17
5.2.9 Searching a Data 'x' in a Linked List, Referenced by the Pointer Head 5-18
5.2.10 Searching an Element x in a Sorted Linked List 5-19
5.2.11 Handling of Records through Linked List 5-20
5.2.12 Head Node in Linked List 5-20
5.3 Circular Linked List 5-21
5.4 MSBTE Questions and Answers 5-25

UNIT V

Chapter 6 : Trees 6-1 to 6-49

Syllabus :

Terminologies : tree, degree of a node, degree of a tree, level of a node, leaf node. Depth/Height of a tree, In-degree and Out-Degree, Path, Ancestor and descendant nodes

Tree Types and Traversal methods Types of Trees : General tree, Binary tree, Binary search tree (BST).

Binary tree traversal : In order traversal, Preorder traversal, Post order traversal.
Expression tree.

6.1 Introduction to Trees 6-1
6.1.1 Basic Terms 6-1
6.2 Binary Tree 6-3
6.2.1 Representation of a Binary Tree using an Array 6-3
6.2.2 Linked Representation of a Binary Tree 6-5
6.3 General Tree 6-6
6.3.1 Node Declaration for a Tree 6-6
6.4 Types of Binary Tree 6-8
6.4.1 Full Binary Tree 6-8
6.4.2 Complete Binary Tree 6-8
6.4.3 Skewed Binary Tree 6-8
6.4.4 Strictly Binary Tree 6-9
6.4.5 Extended Binary Tree (2-Tree) 6-9
6.5 Binary Tree Traversal 6-9
6.5.1 Preorder Traversal (Recursive) 6-10
6.5.2 Inorder Traversal (Recursive) 6-11
6.5.3 Postorder Traversal (Recursive) 6-12
6.6 Tree Traversal Examples 6-14
6.7 Binary Search Tree (BST) 6-20

6.7.1 Definition 6-20
6.7.2 Operations on a Binary Search Tree 6-21
6.7.2(A) Initialize Operation 6-21
6.7.2(B) Find Operation 6-21
6.7.2(C) Makeempty Operation 6-22
6.7.2(D) Insert Operation 6-22
6.7.2(E) Example on Creation of a BST 6-23
6.7.2(F) Delete Operation 6-28
6.7.2(G) Create 6-30
6.7.2(H) FindMin 6-31
6.7.2(I) FindMax 6-31
6.7.3 Program for Various Operations on BST 6-31
6.8 Application of Trees 6-34
6.8.1 Expression Trees 6-34
6.8.2 Conversion of an Expression into Binary Tree 6-35
6.9 MSBTE Questions and Answers 6-43

Chapter 7 : Graphs **7-1 to 7-20**

Syllabus :

Introduction to Graph terminologies : graph, node (Vertices), arcs (edge), directed graph, undirected graph, in-degree, out-degree, adjacent, successor, predecessor, relation, path, sink, articulation point.

Adjacency List, Adjacency Matrix of directed/undirected graph.

7.1 Terminology and Representation 7-1
7.1.1 Definition 7-1
7.1.2 Undirected Graph 7-2
7.1.3 Directed Graph 7-2
7.1.4 Complete Graph 7-2
7.1.5 Weighted Graph 7-2
7.1.6 Adjacent Nodes 7-3
7.1.7 Path 7-3
7.1.8 Cycle 7-3
7.1.9 Connected Graph 7-3
7.1.10 Subgraph 7-3
7.1.11 Component 7-3
7.1.12 Degree of a Vertex 7-4
7.1.13 Self Edges or Self Loops 7-4
7.1.14 Multigraph 7-4
7.1.15 Tree 7-4
7.1.16 Spanning Trees 7-5
7.1.17 Minimal Spanning Tree 7-5
7.1.18 Applications of Spanning Tree 7-5
7.1.19 Successor, Predecessor, Relation 7-5
7.1.20 Sink 7-5
7.1.21 Articulation Point 7-6
7.2 Representation of Graphs 7-6
7.2.1 Adjacency Matrix 7-6
7.2.2 Adjacency List 7-7
7.2.3 Examples on Graph Representation 7-8
7.2.4 Programs on Graph Representation 7-14
7.3 MSBTE Questions and Answers 7-17

List of Practicals **L-1 to L-25**

L.1 Program Based on Array L-1
L.2 Various Searching Methods L-2
L.3 Sorting Methods L-4
L.4 Operations on Stack L-6
L.5 Implementation of Linear Queue L-8
L.6 Implementation of Circular Queue L-10
L.7 Operations on Singly Linked List L-12
L.8 Singly Circular Linked List L-16
L.9 Operations on BST L-21

- **Appendix A :** Solved MSBTE Question Paper of Winter 2018 ..... A-1 to A-3

❑❑❑

CHAPTER 1

Unit - I

Introduction to Data Structure

Syllabus Topics

Concept and need of DS, Abstract Data Type.

Types of Data Structures : (i) Linear Data Structures (ii) Non-Linear Data Structures

Algorithm Complexity : (i) Time (ii) Space

Operations on Data Structures : (i) Traversing, (ii) Searching, (iii) Insertion, (iv) Deletion, (v) Sorting.

1.1 Data Representation

Data is a collection of numbers, alphabets and symbols combined to represent information. A computer takes raw data as input and after processing of data it produces refined data as output. We might say that computer science is the study of data.

- Binary system of representation is used in computer.
- An integer number is represented by its binary equivalent
- A negative number is represented using 2's complement representation
- A character is represented using its ASCII code.
- A real number is represented using IEEE representation format.

1.1.1 Data

Data is a collection of numbers, alphabets and symbols combined to represent information. A computer takes raw data as input and after processing of data it produces refined data as output. We might say that computer science is the study of data.

Atomic data

They are non-decomposible entity. For example, an integer value 523 or a character value 'A' cannot be further divided. If we further divide the value 523 in three digits '5', '2' and '3' then the meaning may be lost.

Composite data

It is a composition of several atomic data and hence it can be further divided into atomic data.

For example, date of birth (say 15/3/1984) can be separated into three atomic values. First one gives the day of the month, second one gives the month and the last one is the year.

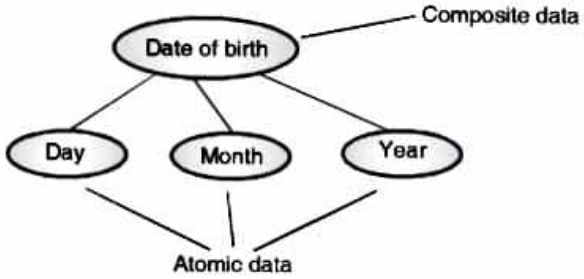


Fig. 1.1.1

1.1.2 Data Types

A data type is a term which refers to the kind of data that variables may hold in a programming language.

Ex. : int x ; [x can hold, integer type data]

Every programming language has a method for declaring a set of variable of a particular type. A value stored in a variable can not be interpreted properly without knowing its type. A byte of information stored in computer memory may represent an integer value, a character value, a BCD (Binary Coded Decimal) value or a Boolean value. Therefore, it is necessary that the value stored in memory must be treated as of a particular type and interpreted accordingly.

1.1.3 Data Types in 'C'

➔ (MSBTE : S-11)

Q. 1.1.1 Enlist and explain primitive data types in C. (Refer section 1.1.3) **S-11**

In every programming language there is set of built-in data types. Language allows variables to name data of that type. It also allows a set of operations (through operators like +, –, /, % etc) for manipulation of these values. Every programming language has its own representation for primitive data types.

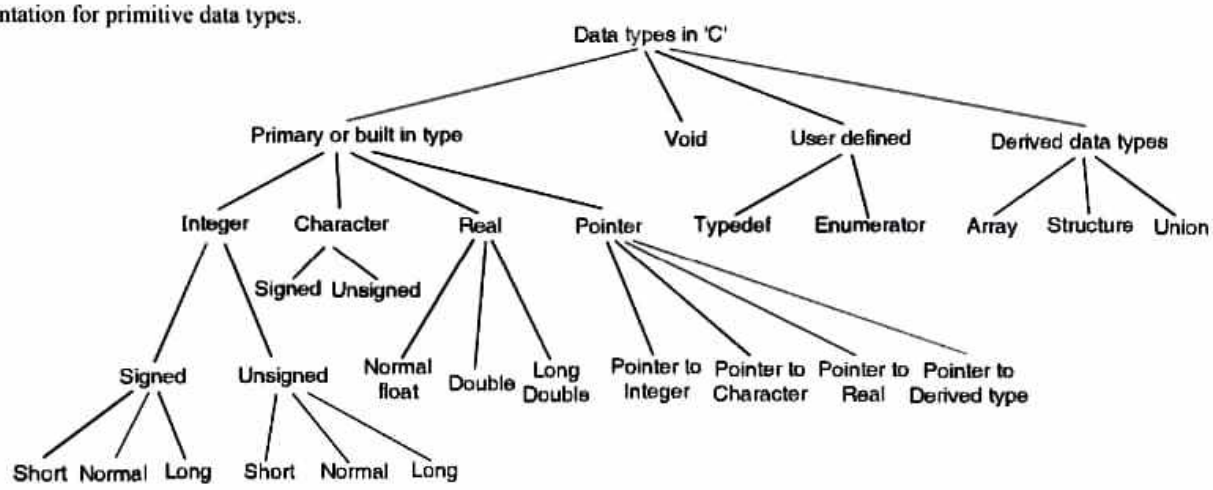


Fig. 1.1.2 : Data types in 'C'

Table 1.1.1 : Size and range of data types on a 16-bit m/c

Type	Size (bits)	Range	Representation
Char or signed char	8	– 128 to 127	2's complement ASCII
Unsigned char	8	0 to 255	Binary ASCII
int or signed int	16	– 32,768 to 32,767	2's complement
Unsigned int	16	0 to 65535	Binary
Short int or signed short int	8	– 128 to 127	2's complement
Unsigned short int	8	0 to 255	Binary
Long int or signed long int	32	– 2, 147, 483, 648 to 2, 147, 483, 647	2's complement
Unsigned long int	32	0 to 4, 294, 967, 295	Binary
Float	32	3.4E – 38 to 3.4E + 38	IEEE
Double	64	1.7E – 308 to 1.7E + 308	IEEE
Long double	80	3.4E – 4932 to 1.1E + 4932	IEEE

Program 1.1.1 : Program to accept and display 'int' and 'char' type of data.

Q. 1.1.2 Write a 'C' program to accept 'int' and 'char' type of data and display the same. (Refer section 1.1.3) **S-11**

```
# include <stdio.h>
# include <conio.h>
  void main ()
    {
        int x ;
        char y ;
        printf("\n enter an integer value :");
        scanf("%d", &x);
        printf("\n entered value = %d", x);
```

```
        printf("\n enter a character value :");
        scanf("%C", &y);
        printf("\n entered value = %(", y);
        getch();
    }
```

1.1.4 User-defined Type

"type definition" allows user to define an identifier that will represent an existing data type.

For example : `typedef float marks ;`

After above declaration, marks can be used for float.

`marks phy, che, math;`

Here, marks is another name for float. Marks can be used to declare variables of the type float. This approach, enhances **readability** of a program.

Another user-defined data type is **enumerated data** type.

An example :

```
enum day (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
enum day today ;
```

Variable today, which is declared of enum day type can be assigned a value from the set (Mon, Tue, Wed, Thu, Fri, Sat, Sun).

```
today = Thu ;            /* A valid statement */
if (today == Mon)        /* A valid statement */
today = Fri ;            /* A valid statement */
```

1.1.5 Derived Data Types ➔(MSBTE : W-12)

Q. 1.1.3 Explain the term : Derived data type. (Refer section 1.1.5) **W-12**

Array : A group of similar elements is stored in contiguous memory.

Ex. int a[6] ; /* defining an array of size 6 */

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]

int a[6] defines an integer array of six elements. These elements are stored in contiguous memory locations. An element of the array is accessed as "a[i] " where a is name of the array and i is index.

1.1.6 Structure

It allows, one to group non-homogeneous elements. It becomes necessary to group all attributes of an entity or properties of an object in a single cell. Without structure, it becomes extremely difficult to handle a complex data structure.

For example

```
struct student
{
    char name [35];
    int rollno;
    char address [120];
    int gender;
} ;
struct student S1, S2 ;
```

Elements of a structure type variable can be accessed through dot (·) operator.

```
S1.name;          /* name of student s */
S1.rollno;
S1.address;
S1.gender;
```

> Array is used for storing of homogeneous elements, structure can be used for storing of non-homogeneous elements.

1.1.7 Union

Unions are a concept borrowed from structures and therefore follow the same syntax as structures.

However, there is major distinction between them in terms of storage. In structures, each member has its own storage location, whereas all the members of a union share the same memory locations.

Example :

```
Union item
    {   int m;
        float x ;
        char c ;
    } item_cd ;
```

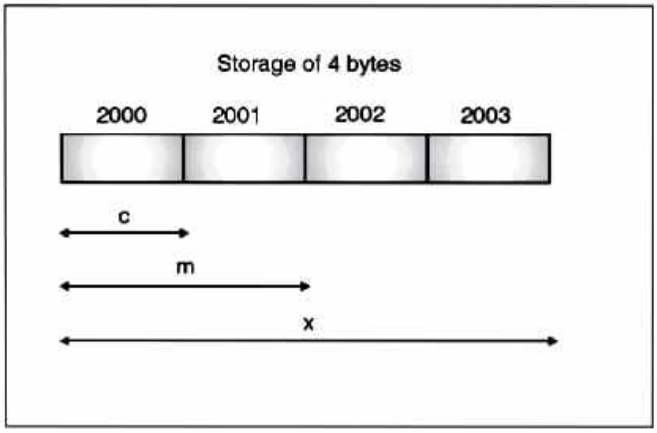


Fig. 1.1.3 : Sharing of storage locations by the members of union

To access a union member, we can use the same syntax that we use for structure members.

i.e. item_cd.m

item_cd.x

item_cd.c

Syllabus Topic : Abstract Data Type

1.2 Abstract Data Types (ADT)

➔ (MSBTE : W-12)

Q. 1.2.1 Describe abstract data type. (Refer section 1.2) **W-12**

The concept at abstraction is commonly found in computer science. A big program is never written as a monolithic piece of program, instead it is broken down in smaller modules (may be called a function or procedure) and each module is developed independently.

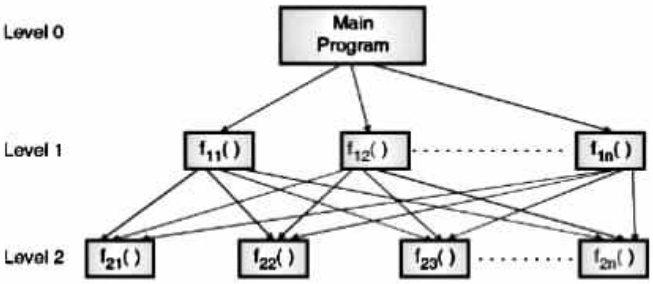


Fig. 1.2.1 : Hierarchical organized program

When the program is hierarchical organized as shown in the Fig. 1.2.1, then the "main program" utilizes services of the functions appearing at level 1. Similarly, functions written at level 1 utilizes services of functions written at level 2. Main program uses the services at the next level functions without knowing their implementation details. Thus a level of **abstraction** is created. When an abstraction is created at any level, our concern is limited to "what it can do" and not "how it is done".

☞ Abstraction in case of data

Abstraction for primitive types (char, int, float) is provided by the compiler. For example, we use integer type data and also, perform various operations on them without knowing :

(1) Representation

(2) How various operations are performed on them.

Example :

```
int x, y, z ;
x = -13 ;
```

Constant –13 is converted to 2's complement and then stored in x. Representation is handled by the compiler

```
x = y + z ;
```

Meaning of the operation '+' is defined by the compiler and its implementation details remain hidden from the user.

Implementation details (representation) and how various operations are implemented remain hidden from the user. User is only concerned about, how to use these operations.

Objects such as lists, sets and graphs along with associated operations, can be viewed as abstract data type. Integer, char, real are primitive data types and there are set of operations associated with them. For the 'set' ADT (Abstract Data Type),

We might have operations like union, intersection, size and complement. Once the data type set is defined (representation and associated functions) then the ADT set can be used in any application program.

☞ The abstract data type "List"

A list is a sequence of 0 or more elements of a given type (element type could be integer, float etc.). Such a list is often represented as :

$a_1, a_2 \ldots\ldots\ldots a_n$

– n = Number of elements in the list

– a_i = i^{th} element of the list

- When n = 0, the list is empty having no element.
- Each element other than a_1 has a predecessor
- Each element, other than a_n has a successor.

In order to form an ADT from the mathematical notation of a list, we must represent the list and define a set of operations on objects of type List.

Representation of a list

A List can be represented in 'C' using a structure.

```
typedef struct List
{  int data[50] ;
   int n ;
}  List ;
```

A List of maximum of 50 integer type elements
n is the actual number of elements in the List.

A set of representative operations on a list

1. Insert(L, X, P) : Insert X at position P in list L.
2. Locate(X, L) : This function returns the position of element X on list L.
3. Retrieve(L, P) : This function returns the element at position P on list L.
4. Delete(L, P) : Delete the element at position P of the list L.
5. MAKENULL(L) : Creates an empty list L.
6. PrintList(L) : Print the elements of L in the order of occurrence.

Syllabus Topic : Concept of Data Structures

1.3 Concept of Data Structures

(MSBTE : S-08, S-10, S-11, S-12, W-12, S-13, W-13,W-14, W-15, W-16)

Q. 1.3.1 What is data structure? (Refer section 1.3) **S-08, S-10, S-12, S-13**

Q. 1.3.2 Define data structure. (Refer section 1.3) **S-11, W-12, S-13, W-13, W-14**

Q. 1.3.3 Define data structure. Enlist any two types of non-linear data structure. (Refer sections 1.3 and 1.3.2) **W-15**

Q. 1.3.4 Define data structure. (Refer section 1.3) **W-16**

The data structure is concerned with the following things :

1. Organization of data
2. Associativity among data elements
3. Accessing methods
4. Operations on data
5. Processing alternatives for data

While representing data, relationship between individual elements of the data should be considered. The data structure deals with representation of data considering not only the elements stored but also their relationship to each other.

For writing an efficient program, a proper data structure should be selected.

- A proper data structure should be selected so that the relationship between data elements can be expressed.
- Processing and accessing of data should be efficient.

A data structure can also be defined as an instance of ADT. A data structure is formally defined to be a triplet (D,F,A)

where, D stands for a set of domains
F stands for a set of operations
A stands for a set of axioms

Syllabus Topic : Need of Data Structure

1.3.1 Need of Data Structure

(MSBTE : S-08, W-08, S-13, S-16)

Q. 1.3.5 Why do we need data structure ? (Refer section 1.3.1) **S-08, S-13**

Q. 1.3.6 State the need of data structure. (Refer section 1.3.1) **W-08**

Q. 1.3.7 State the need of data structure. (Refer sections 1.3.1) **S-16**

There is a close connection between the structuring of data and the synthesis of algorithm. For each data item, there are number of operations. Data should be represented in a way that makes efficient implementation of operation.

If a linear data is represented using linked organization then modification is less time consuming whereas accessing a large list becomes inefficient. In sequential organization, accessing is efficient but modifications are costly.

Syllabus Topic : Types of Data Structures (I) Linear Data Structures and (II) Non-linear Data Structures

1.3.2 Linear and Non-Linear

➔ **(MSBTE : S-09, S-11, W-11, S-12, W-12, S-14, W-15)**

Q. 1.3.8 What are linear and non-linear data structures ? Give two example of each.
(Refer section 1.3.2) **S-09, W-12**

Q. 1.3.9 Enlist any two types of non-linear data structures along with example.
(Refer section 1.3.2) **S-11**

Q. 1.3.10 Explain linear and non-linear data structures.
(Refer section 1.3.2) **W-11**

Q. 1.3.11 Explain non linear data structure.
(Refer section 1.3.2) **S-12**

Q. 1.3.12 Explain the concept of linear list with example.
(Refer section 1.3.2) **S-14**

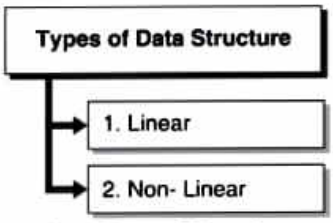
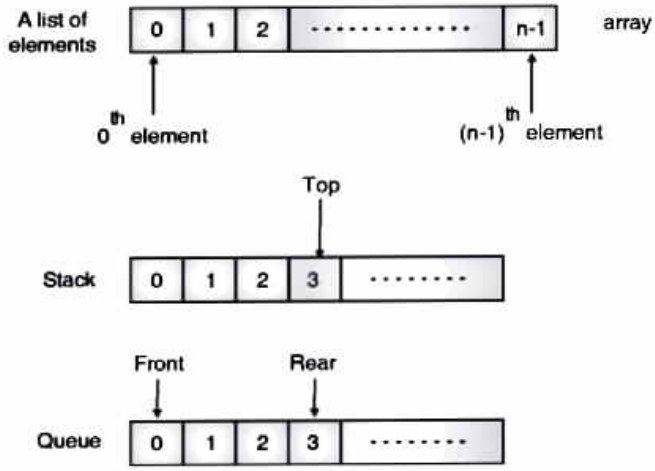


Fig. 1.3.1 : Types of Data Structure

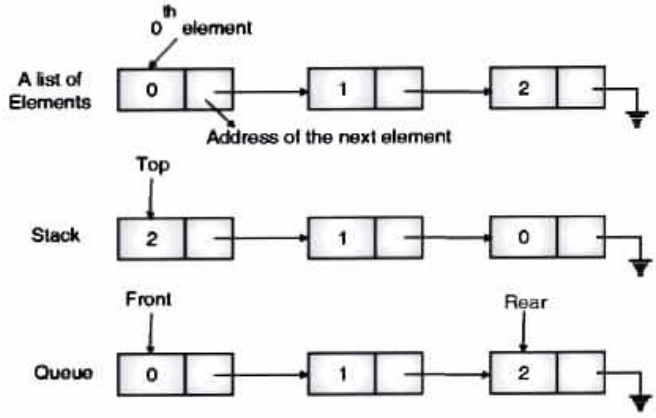
➔ **1. Linear**

Elements are arranged in a Linear fashion (one dimension). All one-one relation can be handled through Linear data structures. Lists, stacks and queues are examples of linear data structure.

☞ **Representation of linear data structures in an array**



☞ **Representation of linear data structures through linked structure**



➔ **2. Non-linear**

All one-many, many-one or many–many relations are handled through non-linear data structures. Every data element can have a number of predecessors as well as successors. Tree graphs and tables are examples of non-linear data structures.

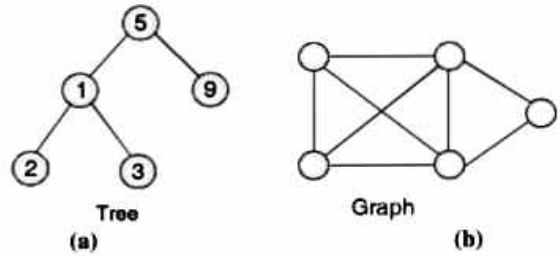
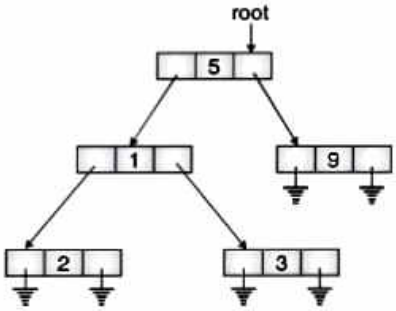


Fig. 1.3.2 Contd...

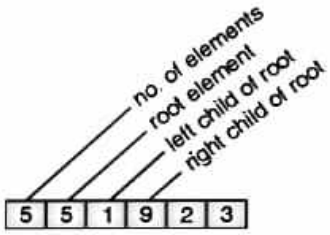
5	9	1
6	4	13
2	5	0
9	6	11

(c) Table

Fig. 1.3.2 : Non- linear data structures



(a) Representation of the binary tree through linked structure



(b) Representation of the binary tree through an array

Fig. 1.3.3 : Representation of tree of Fig. 1.3.2(a)

1.4 Classification of Data Structure

➔ **(MSBTE : S-08, W-08, S-09, W-09, S-10, S-12, S-13, W-14, W-16, W-17)**

Q. 1.4.1 Give classification of data structure and give two examples of each. (Refer section 1.4) **S-08, S-13**

Q. 1.4.2 State different types of data types. (Refer section 1.4) **S-09**

Q. 1.4.3 Define primitive data structure. (Refer section 1.4) **W-08, W-09**

Q. 1.4.4 Give classification of data structure. (Refer section 1.4) **W-08, S-10, S-12, W-14, W-16**

Q. 1.4.5 Describe classification of data structure with example of each. (Refer sections 1.4) **W-17**

Data structures are normally divided into two categories :

1. Primitive data structure .
2. Non – Primitive data structure (derived) .

Classification of data structure is shown in Fig. 1.4.1

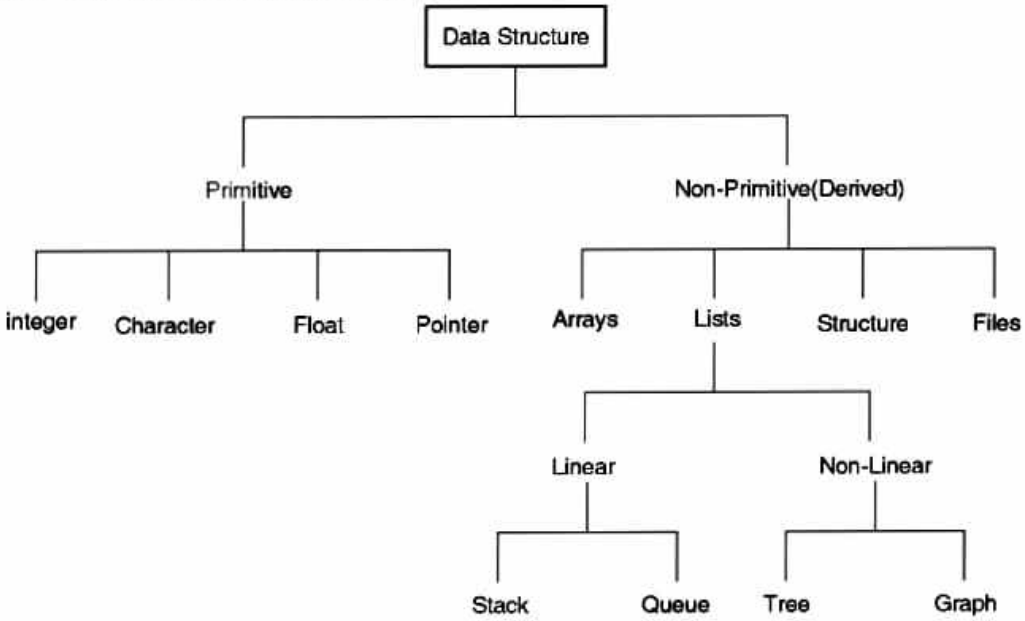


Fig. 1.4.1 : Classification of data structure

1.4.1 Primitive and Non-Primitive

➔ **(MSBTE : W-09, W-10, W-12, W-13, S-15)**

Q. 1.4.6 Describe the term "primitive data structures" Enlist four types of primitive data structure.
(Refer section 1.4.1) **W-09**

Q. 1.4.7 What do you mean by primitive and non-primitive data types ?
(Refer section 1.4.1) **W-10**

Q. 1.4.8 Explain the term : Primitive data type.
(Refer section 1.4.1) **W-12**

Q. 1.4.9 Define the term :
(1) Primitive data structure
(2) Non primitive data structure.
(Refer section 1.4.1) **W-13, S-15**

☞ **Primitive**

The integers, reals, logical data, character data, pointers and reference are primitive data structures. These data types are available in most programming languages as built in type. Data objects of primitive data types can be operated upon by machine level instructions.

☞ **Non-primitive**

These data structures are derived from primitive data structures. A set of homogeneous and heterogeneous data elements are stored together.

Examples of Non-primitive data structures :

Array, structure, union, linked-list, stack, queue, tree, graph.

Some of the most commonly used operations that can be performed on data structures are shown in Fig. 1.4.2.

Fig. 1.4.2 : Data structure operations

Syllabus Topic : Operations on Data Structures (i) Traversing (ii) Searching (iii) Insertion (iv) Deletion (v) Sorting

1.5 Operations on Data Structures

➔ **(MSBTE : W-08, S-10, W-10, S-12, W-12, S-13, W-14, S-15, W-15, S-16, W-17)**

Q. 1.5.1 Describe various types of operation that can be performed on data structure.
(Refer section 1.5) **S-10**

Q. 1.5.2 Give 4 operations of data structure.
(Refer section 1.5) **W-08, S-13**

Q. 1.5.3 Enlist data structure operations.
(Refer section 1.5) **W-10, W-14, S-15**

Q. 1.5.4 List and explain any four operations on data structure. (Refer section 1.5) **S-12, W-17**

Q. 1.5.5 Describe data structure operations.
(Refer section 1.5) **W-12, S-16**

Q. 1.5.6 Describe any four operations on data structure.
(Refer section 1.5) **W-15**

Many operations are performed on a data structure. Typical operations on a data structure are :

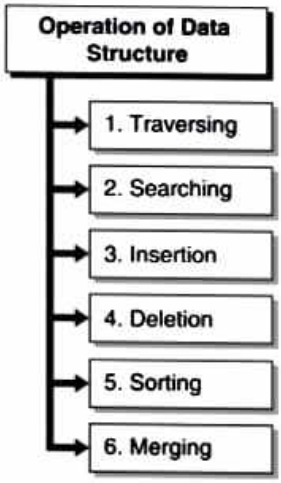


Fig. 1.5.1 : Opeation of Data Structure

➔ 1. **Traversing** a data structure is accessing each data and accessing only once.

➔ 2. **Searching** is finding the location of a data in within the given data structure.

➔ 3. **Inserting** is adding a new data in the data structure.

➔ 4. **Deleting** is removing a data from the data structure.

➜ 5. **Sorting** is arranging of data in some logical order.

➜ 6. **Merging** is combining of two similar data structures.

1.6 Introduction to Algorithm

➔ **(MSBTE : S-09, S-11, W-12, S-16, W-17)**

Q. 1.6.1 Define algorithm **with example.** (Refer section 1.6) S-09

Q. 1.6.2 **Enlist and explain characteristics of algorithm.** **(Refer section 1.6)** S-11

Q. 1.6.3 **Define term algorithm and explain** characteristics **of algorithm.** (Refer section 1.6) W-12

Q. 1.6.4 Define **an algorithm. (Refer section 1.6)** S-16

Q. 1.6.5 **Define the term algorithm.** **(Refer** section 1.6) W-17

The field of Computer Science revolves around writing of programs for several problems for various domains. A program consists of data structures and algorithms. An algorithm is a set of steps required to solve a problem. These steps are performed on a sample data representing an instance of the problem. Thus an algorithm maps a set of input data (from input domain) to a set of output data (in output domain) through a sequence of operations. An algorithm must have the following properties :

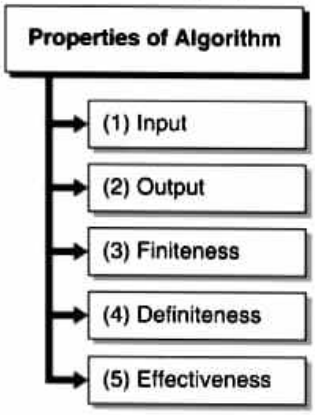


Fig. 1.6.1 : Properties of Algorithm

➜ **(1) Input** : Input data, supplied externally (zero or more).

➜ **(2) Output :** Result of the program.

➜ **(3) Finiteness :** In every case, algorithm terminates after a finite number of steps.

➜ **(4) Definiteness :** The steps should be clear and unambiguous.

➜ **(5) Effectiveness :** An algorithm should be written using basic instructions. It should be feasible to convert the algorithm in a computer program.

Take the problem of finding the GCD (Greatest Common Divisor) of two positive integers an example.

Inputs to the algorithm are two positive integers. Output is a positive integer which is GCD of two positive integers given as input.

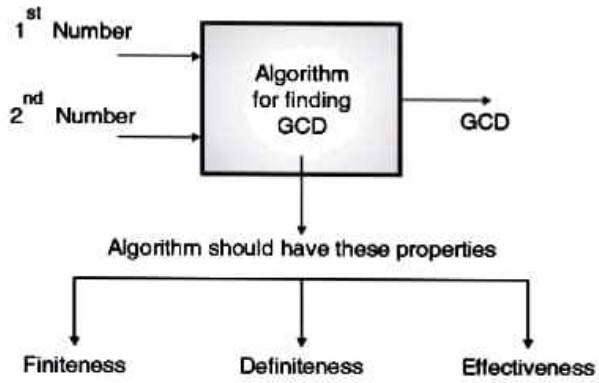


Fig. 1.6.2

The algorithm is described as a series of steps of basic operations. These steps must be performed in a sequence. Each step of the algorithm is labeled.

Step 1 : Read two positive integers and store them in x and y.

Step 2 : Divide x by y. Let the remainder be r and the quotient be q.

Step 3 : If r is zero then go to step 7.

Step 4 : Assign y to x.

Step 5 : Assign r to y.

Step 6 : goto step 2.

Step 7 : Print y (the required GCD)

Step 8 : stop

The steps mentioned in the above algorithm are simple and unambiguous. Anybody, carrying out these steps will clearly know what to do in each step. Hence, the above algorithm satisfies the **definiteness** property of an algorithm.

Table 1.6.1 depicts step-wise execution of the above algorithm on two input numbers 15 and 9.

Table 1.6.1

Steps performed	Value of the variables				
	x	y	r	q	
Step 1	15	9	–	–	Input data supplied externally
Step 2	15	9	6	1	
Step 3	15	9	6	1	
Step 4	9	9	6	1	
Step 5	9	6	6	1	
Step 6	9	6	6	1	
Step 2	9	6	3	1	
Step 3	9	6	3	1	
Step 4	6	6	3	1	
Step 5	6	3	3	1	
Step 6	6	3	3	1	
Step 2	6	3	0	2	
Step 3	6	3	0	2	
Step 7	→ Print the output (GCD) as 3.				

Algorithm terminates after a finite number of steps. It is easy to prove that the algorithm will terminate for any two given integer numbers. Thus the above algorithm satisfies the **finiteness** property of the algorithm. Every programming language has input statement, output statement, assignment statement, conditional branching statement, unconditional branching statement and hence the above algorithm can easily be converted to a program. Thus the above algorithm satisfies the **effectiveness** property of the algorithm.

An algorithm can be described in many ways. A language like English can be used but it should be free from any ambiguity. Flowcharts, consisting of graphical symbols like :

Table 1.6.2 : Symbols used in flowcharts

Symbol	Purpose
	Terminal Symbol (Start / Stop)
	Assignment Statements, expression etc.
	Read / Print
	Decision making
	Connector
→	Flow indication

Example 1.6.1 : Write an algorithm for adding 10 numbers.

Solution : Flowchart

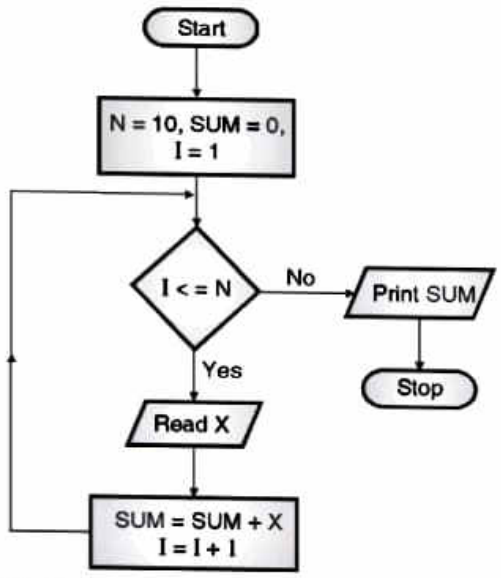


Fig. Ex. 1.6.1 : Flowchart for adding 10 numbers

Algorithm as a series as steps :

Step 1 : Assign 10 to N

Step 2 : Assign 0 to SUM

Step 3 : Assign 1 to I

Step 4 : if (I > N) go to step 9

Step 5 : read x

Step 6 : Assign SUM + X To SUM

Step 7 : Assign I + 1 to I

Step 8 : Go to step 4

Step 9 : Print SUM

Step 10 : Stop

1.7 Different Approaches for Designing an Algorithm

➔ **(MSBTE : S-08, W-09, S-10, W-10, S-12, W-13, S-14, W-14, S-15, W-15, S-17, W-17)**

Q. 1.7.1 Explain different approaches to design an algorithm. (Refer section 1.7) **S-08, W-09, S-10, S-12, S-14, S-15**

Q. 1.7.2 What are different approaches for designing an algorithm ? (Refer section 1.7) **W-10**

Q. 1.7.3 Enlist approaches to design an algorithm. (Refer section 1.7) **W-13, W-14**

Q. 1.7.4 Describe different approaches to design an algorithm. (Refer section 1.7) **S-17, W-17**

Design of an algorithm is an important issue. There are several approaches for designing an algorithm. These approaches include :

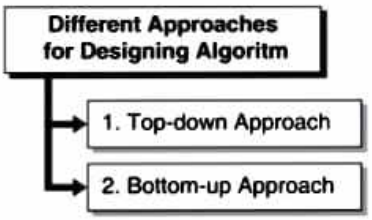


Fig. 1.7.1 : Different approaches for Designing Algorithm

1.7.1 Top-down Approach

➔ **(MSBTE : W-15, W-16)**

Q. 1.7.5 Describe Top-down and bottom-up approach by giving example. (Refer sections 1.7.1 and 1.7.2) **W-15**

Q. 1.7.6 Compare Top-down approach v/s Bottom-up approach [any four points]. (Refer sections 1.7.1 and 1.7.2) **W-16**

One has to write a program to solve a specific task. A single procedure is never written to solve a large task. A programmer tries to partition the solution into subtasks. Each subtask is similarly decomposed until all tasks are expressed within a programming language. Thus the top-down method of design is based on decomposing a large problem into sub-problems and keep repeating this process till the resulting sub-problems are so simple that their solutions can be expressed in a few lines of a programming language.

1.7.2 Bottom-up Approach

➔ **(MSBTE : W-15, W-16)**

Q. 1.7.7 Describe Top-down and bottom-up approach by giving example. (Refer sections 1.7.1 and 1.7.2) **W-15**

Q. 1.7.8 Compare Top-down approach v/s Bottom-up approach [any four points]. (Refer sections 1.7.1 and 1.7.2) **W-16**

This approach is reverse of top-down approach. Here, the different parts of the problem are first solved using a programming language and then these pieces of programs are combined into a complete program.

☞ **Example of Top-down and bottom-up approach**

In top – down, we travel from root node to Leaf node. In bottom – down approach, we follow from leaf node to root node.

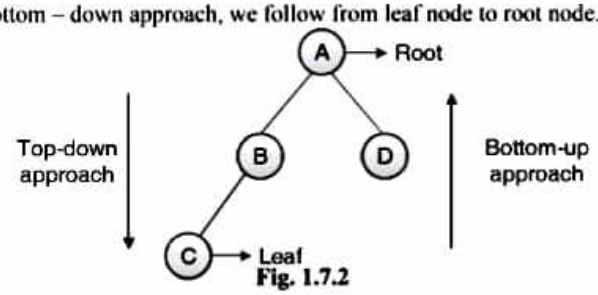


Fig. 1.7.2

Syllabus Topic : Algorithm Complexity (i) Time (ii) Space

1.8 Complexity in Terms of Time and Space

1.8.1 Algorithm Analysis

➔ **(MSBTE : W-08, S-09, W-09, W-10, S-11, S-12, W-12, S-13, W-13, S-14, W-14, S-15, W-15, S-16, W-16, S-17, W-17)**

Q. 1.8.1 Explain different ways of analyzing algorithms. (Refer section 1.8.1) **W-08, W-09**

Q. 1.8.2 Compare time complexity and space complexity with respect to algorithm. (Refer section 1.8.1) **W-08**

Q. 1.8.3 Explain time and space complexity related to algorithms and also state their importance. (Refer section 1.8.1) **S-09, S-12**

Q. 1.8.4 Explain the time and space complexity of an algorithm. (Refer section 1.8.1) **W-09, S-13, S-14**

Q. 1.8.5 Define:

(1) Time complexity

(2) Space complexity of an algorithm

(Refer section 1.8.1) **W-10**

Q. 1.8.6 Explain the two criteria required in performance analysis of an algorithms. (Refer section 1.8.1) **S-11**

Q. 1.8.7 Explain time and space complexity related to algorithms with examples. (Refer section 1.8.1) **W-12, W-13**

Q. 1.8.8 Describe algorithm analysis in term of time complexity and space complexity. (Refer section 1.8.1) **W-14**

Q. 1.8.9 Explain time and space complexity with example. (Refer sections 1.8.1 and 1.8.2) **S-15**

Q. 1.8.10 Define time complexity and space complexity. (Refer sections 1.8.1 and 1.8.2) **W-15**

Q. 1.8.11 How an algorithm is analysed? (Refer section 1.8.1) **S-16**

Q. 1.8.12 What is complexity of an algorithm ? Describe time complexity and space complexity. (Refer section 1.8) **W-16**

Q.1.8.13 Define complexity and classify it. (Refer section 1.8.1) **S-17**

Q. 1.8.14 Describe time and space trade off and time and space complexity with example of each. (Refer section 1.8.1) **S-17**

Q. 1.8.15 Describe with example, time complexity and space complexity of an algorithm. (Refer sections 1.8.1) **W-17**

Most often there are many algorithms for solving a problem. On what basis should we choose an algorithm ? There are often two contradictory goals.

(i) Algorithm should be easy to understand, write and debug .

(ii) Algorithm should make efficient use of computer resources like CPU, memory etc.

When we write a program to be used a few times, goal (i) is most important. Cost of writing the program will have an upper hand over the cost of running the program, when the program is to be used many times, the cost of running the program and hence the goal (ii) should be given more weightage.

Analyzing a program should quantify the requirement of computing resources during execution. Most important of these resources are computer time and memory. Analysis of algorithms focuses on computation of space and time complexity. Space requirement means the space required to store input data either static or dynamic. Space required on top of the system stack to handle recursion/function Call should also be considered. Computing time, an algorithm might require for its execution, would normally depend on the size of the input.

```
#include <stdio.h>
void main( )
{
    int i, n, sum, x ;
   sum = 0 ;
    printf("\n enter no of data to be added");
    scanf("%d",&n) ;
    for (i =1 ; i<= n ; i++)
   {
        scanf("%d", &x) ;
        sum = sum + x ;
    }
    printf("\n sum = %d", sum) ;
}
```

Space requirement = Space required to store the variables i, n, sum and x = 2 + 2+ 2 +2 =8

[an integer requires 2 bytes of memory space.]

Calculation of computation time

Statement	Frequency	Computation time
sum = 0	1	t_1
printf ("\n Enter no of data to be added");	1	t_2
scanf ("%d", & n)	1	t_3
for (i = 1 ; i <=n ; i++)	n + 1	$(n + 1) t_4$
scanf ("%d", &x)	n	nt_5
sum = sum + x	n	nt_6
printf("\n sum = %d", sum)	1	t_7

Total computation time $= t_1 + t_2 + t_3 + n(t_4 + t_5 + t_6) + t_4 + t_7$

$$T = n(t_4 + t_5 + t_6) + (t_1 + t_2 + t_3 + t_4 + t_7)$$

For large n, T can be approximated to

$$T \approx n(t_4 + t_5 + t_6) = kn$$

where

$k = t_4 + t_5 + t_6$, ...a constant value.

Thus, $T = kn$

or $T \propto n$

(i) t_1, t_2 .....t_7 are computer dependent, on a faster computer, the execution time of an instruction will be less.

(ii) for (i = 1;i <= n ; i++), will be executed n + 1 times and not n times. For all values of i between 1 and n, the condition i < = n will evaluate to true and when i becomes n+1, the condition i < = n will evaluate to false. Thus the above instruction will be executed n + 1 times.

Example 1.8.1 : Determine the frequency count for all statements in the following program segment

```
1.  i = 1;
2.  while ( i <= n)
    {
3.  x = x + 1;
4.  i = i + 1 ;
    }
```

Solution :

Statement No.	Frequency
1	1
2	n + 1
3	n
4	n.

1.8.2 Measuring the Running Time of a Program (Time Complexity)

➔ **(MSBTE : S-15, W-15)**

Q. 1.8.16 Explain time and space complexity with example.
(Refer sections 1.8.1 and 1.8.2) **S-15**

Q. 1.8.17 Define time complexity and space complexity.
(Refer sections 1.8.1 and 1.8.2) **W-15**

Running time of a program can be judged on the basis of factors such as :

1. Input to the program.
2. Size of the program.
3. Machine language instruction set.
4. The machine we are executing on.
5. Time required to execute each machine instruction.
6. The time complexity of the algorithm of the program.

1.8.3 Measurement of Growth Rate (Asymptotic Growth Rate)

1.8.3(A) Asymptotic Consideration

When considering time complexities $f_1(n)$ and $f_2(n)$ of two different algorithms for a given problem of size n, we need to consider and compare the behaviour of the two functions only for large n. If the relative behaviour of two functions for smaller values conflict with the relative behaviour for larger values, then we ignore the conflicting behaviour for smaller values. For example, consider the following functions.

$f_1(n) = 100n^2$, $f_2(n) = 5n^3$

Representing time complexities of two solutions of a problem.

n	$f_1(n)$	$f_2(n)$
1	100	5
5	2500	625
10	10000	5000
20	40000	40000

$f_1(n) \geq f_2(n)$ for $n \leq 20$

We would still prefer the solution having $f_1(n)$ as time complexity because, $f_1(n) \leq f_2(n)$ for all $n \geq 20$

1.8.3(B) Constant Factor in Complexity Measure

Let us consider an algorithm having a timing complexity given by the function $f(n) = 100\ n^2$

where [100 – a constant, n – size of the problem].

The time required for solving a problem, depends not only on the size of the problem but also, on the hardware and software used to execute the solution. The effect of hardware and software on the time required may closely be approximated by a constant.

Suppose, a new computer executes a program two times faster than another computer. Then irrespective of the size of the problem, the new computer solves the problem roughly two times faster than the computer. Thus we conclude that the time requirement for execution of a solution, changes by a constant factor on change in hardware or software.

An important consequence of the above discussion is that if the time taken by one machine in executing a solution is of the order of n^2 (say), then time take by every machine is of the order of n^2. Thus, function different from each other by constant factor, when treated as time complexities, should not be treated as different. i.e. should be treated as complexity wise same.

Following functions have the same time complexities :

$f_1(n) = 5n^2$, $f_2(n) = 100n^2$

$f_3(n) = 1000n^2$, $f_4(n) = n^2$

Time complexity of $f_1(n)$ = Time complexity of $f_2(n)$

= Time complexity of $f_3(n)$

= Time complexity of $f_4(n)$

1.9 The Notation O : (pronounced as big-oh), ($O(n^2)$ is pronounced as big-oh of n^2)]

➔ **(MSBTE : S-08, S-10, W-11, S-12, S-13, S-14, W-16)**

Q. 1.9.1 Describe big 'O' notation used in algorithms. (Refer section 1.9) **S-08, S-10, S-12, S-13, S-14**

Q. 1.9.2 What is big 'O' notation in algorithms ? (Refer section 1.9) **W-11**

Q. 1.9.3 Define Big 'O' Notation.(Refer section 1.9) **W-16**

Provides asymptotic upper bound for a given functions. Let f(x) and g(x) be two functions then f(x) is said to be O(g(x)) if there exist two positive integer/real number constants c and k such that

$$f(x) \leq cg(x) \text{ for all } x \geq k$$

The constant k is due to asymptotic consideration and the constant c for hardware/software environment.

Example 1.9.1 : $f(x) = x^2 + 5x$

$g(x) = x^2$ let us take c = 2

Solution :

x	$x^2 + 5x$	$2x^2$	
1	5	2	
2	14	8	$f(x) \leq c\,x^2$
5	50	50	For $x \geq 5$
			(asymptotic behavior)

$f(x) \leq cg(x)$ for all $x \geq k$ where c =2 and k = 5

Example 1.9.2 : For the function defined by $f(x) = 5x^3 + 6x^2 + 1$ show that $f(x) = O(x^3)$

Solution :

$f(x) = 5x^3 + 6x^2 + 1 \leq 5x^3 + 6x^3 + 1x^3$ for all $x \geq 1$

or, $f(x) \leq 12x^3$ for $x \geq 1$

(by replacing each term of x by the highest degree term x^3)

there exist c =12 and k = 1 such that

$f(x) \leq c\,x^3$ for all $x \geq k$

Hence f(x) is $O(x^3)$

Note : Time complexity of a polynomial is same as the time complexity of the highest degree term.

Calculating the running time of a program

Example 1.9.3 : Arrange the following computing functions as per their growth rate. $\log_2 n$, n, $n \log_2 n$, n^2, n^3, 2^n.

Solution :

Common computing functions and their growth rates

$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65536
5	32	160	1024	32768	2, 147, 483, 648

Thus, $O(2^n) > O(n^3) > O(n^2) > O(n \log_2 n) > O(n) > O(\log_2 n)$

Example 1.9.4 : Give time complexity of any four sorting algorithms. S-10

Solution :

1. Bubble sort – $O(n^2)$
2. Selection sort – $O(n^2)$
3. Insertion sort – $O(n^2)$
4. Merge sort – $O(n \log n)$

Example 1.9.5 : What do you mean by frequency count ? Why only frequency count is important in deciding the time complexity of an algorithm ?

Solution :

Total time taken by a statement in execution depends on :

(i) The amount of time a single execution will take.

(ii) Number of times the given statement will be executed.

The product of these numbers will be the total time taken by the statement. Number of times a statement will be executed is called its frequency count. Frequency count may vary from data set to data set.

The constant factor in complexity measure

Let us consider an algorithm having a timing complexity given by the function $f(n) = 100\, n^2$

where, [100 – a constant, n – size of the problem].

The time required for solving a problem, depends not only on the size of the problem but also on the hardware and software used to execute the solution. The effect of hardware and software on the time required may closely be approximated by a constant.

Suppose, a new computer executes a program two times faster than another computer. Then irrespective of the size of the problem, the new computer solves the problem roughly two times faster than the computer. Thus we conclude that the time requirement for execution of a solution, changes by a constant factor on change in hardware or software.

An important consequence of the above discussion is that if the time taken by one machine in executing a solution is of the order of n^2 (say), then time take by every machine is of the order of n^2. Thus, function different from each other by constant factor, when treated as time complexities, should not be treated as different. i.e. should be treated as complexity wise same.

Following functions have the same time complexities :

$$f_1(n) = 5n^2$$

$$f_2(n) = 100n^2$$

$$f_3(n) = 1000n^2$$

$$f_4(n) = n^2$$

Time complexity of $f_1(n)$ = Time complexity of $f_2(n)$

= Time complexity of $f_3(n)$

= Time complexity of $f_4(n)$

1.10 MSBTE Questions and Answers

➔ Summer 2008 – Total Marks 14

Q. 1 What is a data structure ? Why do we need data structure ?**(Sections 1.3 and 1.3.1)** **(4 Marks)**

Q. 2 Give classification of data structure and give two examples of each.**(Section 1.4)** **(4 Marks)**

Q. 3 Describe Big 'O' notation used in algorithms. **(Section 1.9)** **(2 Marks)**

Q. 4 Explain different approaches to design an algorithm. **(Section 1.7)** **(4 Marks)**

➔ Winter 2008 – Total Marks 12

Q. 5 Define primitive Data Structure. Give 4 operations of Data structure.**(Sections 1.4 and 1.5)** **(2 Marks)**

Q. 6 State the need of data structure. Give classification of Data Structure.**(Sections 1.3.1 and 1.4)(4 Marks)**

Q. 7 Compare Time Complexity and Space Complexity with respect to algorithm. **(Section 1.8.1) (2 Marks)**

Q. 8 Explain different way of analyzing algorithms. **(Section 1.8.1) (4 Marks)**

➔ Summer 2009 – Total Marks 12

Q. 9 State different types of data types. **(Section 1.4) (2 Marks)**

Q. 10 What are linear and non linear data structures ? Give two examples of each. **(Section 1.3.2) (4 Marks)**

Q. 11 Define Algorithm with example. **(Section 1.6) (2 Marks)**

Q. 12 Explain time and space complexity related to algorithms and also state their importance. **(Section 1.8.1) (4 Marks)**

➔ Winter 2009 – Total Marks 16

Q. 13 Describe the term "Primitive data structures" Enlist four types of primitive data structure. **(Section 1.4.1) (2 Marks)**

Q. 14 Enlist different ways of analyzing algorithm. **(Section 1.8.1) (2 Marks)**

Q. 15 Explain the time and space complexity of an algorithm. **(Section 1.8.1) (4 Marks)**

Q. 16 Explain different approaches for designing algorithm. **(Section 1.7) (4 Marks)**

➔ Summer 2010 – Total Marks 18

Q. 17 Describe various types of operation that can be performed on data structure. **(Section 1.5)(6 Marks)**

Q. 18 What is data structure? Explain classification of data structure.**(Sections 1.3 and 1.4) (4 Marks)**

Q. 19 Describe Big 'O' notation used in algorithms. Give time complexity of any four sorting algorithms. **(Section 11.9 and Example 1.9.4) (4 Marks)**

Q. 20 Explain different approaches to design an algorithm. **(Section 1.7) (4 Marks)**

➔ Winter 2010 – Total Marks 12

Q. 21 Enlist data structure operations. **(Section 1.5) (2 Marks)**

Q. 22 What do you mean by primitive and non-primitive data types ? **(Section 1.4.1) (4 Marks)**

Q. 23 What are different approaches for designing an algorithm ? **(Section 1.7) (2 Marks)**

Q. 24 Define :

(1) Time complexity

(2) Space complexity of an algorithm.

(Section 1.8.1) (4 Marks)

➔ Summer 2011 – Total Marks 24

Q. 25 Define data structure? Enlist any two types of non-linear data structures along with example. **(Sections 1.3 and 1.3.2) (4 Marks)**

Q. 26 Enlist and explain primitive data types in C. Write a 'C' program to accept 'int' and 'char' type of data and display the same. **(Section 1.1.3, Program 1.1.1) (8 Marks)**

Q. 27 Enlist and explain characteristics of Algorithm. **(Section 1.6) (4 Marks)**

Q. 28 Explain the two criteria required in performance analysis of an algorithms. **(Section 1.8) (8 Marks)**

➔ Winter 2011 – Total Marks 20

Q. 29 Explain Linear and Non Linear data structures. **(Section 1.3.2) (10 Marks)**

Q. 30 What is Big 'O' notation in Algorithms. **(Section 1.9) (10 Marks)**

➔ Summer 2012 – Total Marks 20

Q. 31 Give the classification of data types. **(Section 1.4) (2 Marks)**

Q. 32 What is data structure ? Explain Non Linear data structure. **(Sections 1.3 and 1.3.2) (4 Marks)**

Q. 33 Describe big 'O' notation used in algorithms. **(Section 1.9)** **(2 Marks)**

Q. 34 Explain time and space complexity related to algorithms and also state their importance. **(Section 1.8.1)** **(4 Marks)**

Q. 35 Explain different approaches for designing an algorithm. **(Section 1.7)** **(4 Marks)**

Q. 36 List and explain any four operations on data structure. **(Section 1.5)** **(4 Marks)**

➔ Winter 2012 – Total Marks 20

Q. 37 Explain following terms :

(i) Primitive data type.

(ii) Derived data type.

(Sections 1.4.1 and 1.1.5) **(2 Marks)**

Q. 38 Describe abstract data type.**(Section 1.2)** **(2 Marks)**

Q. 39 Define data structure and describe data structure operations. **(Sections 1.3 and 1.5)** **(4 Marks)**

Q. 40 Explain time and space complexity related to algorithms with examples. **(Section 1.8)** **(4 Marks)**

Q. 41 What are linear and non-linear data structures ? Give two examples of each. **(Section 1.3.2)** **(4 Marks)**

Q. 42 Define term algorithm and explain characteristics of algorithm. **(Section 1.6)** **(4 Marks)**

➔ Summer 2013 – Total Marks 14

Q. 43 Define Data structure and give 4 basic operations of it. **(Sections 1.3 and 1.5)** **(2 Marks)**

Q. 44 Give classification of data structure. **(Section 1.4)** **(2 Marks)**

Q. 45 Describe big 'O' notation used in algorithm. **(Section 1.9)** **(2 Marks)**

Q. 46 What is data structure? Why do we need it? **(Sections 1.3 and 1.3.1)** **(4 Marks)**

Q. 47 Explain time and space complexity of an algorithm. **(Section 1.8.1)** **(4 Marks)**

➔ Winter 2013 – Total Marks 08

Q. 48 Define the term :

(1) Primitive data structure

(2) Non primitive data structure.

(Section 1.4.1) **(2 Marks)**

Q. 49 Define data structure ? Enlist approaches to design an algorithm. **(Sections 1.3 and 1.7)** **(2 Marks)**

Q. 50 Describe time and space complexity related to algorithms with example. **(Section 1.8.1)** **(4 Marks)**

➔ Summer 2014 – Total Marks 16

Q. 51 Explain time complexity and space complexity. **(Section 1.8)** **(4 Marks)**

Q. 52 Explain the two approaches of designing an algorithm. **(Section 1.7)** **(4 Marks)**

Q. 53 Explain the big 'O' notation. **(Section 1.9)** **(4 Marks)**

Q. 54 Explain the concept of linear list with example. **(Section 1.3.2)** **(4 Marks)**

➔ Winter 2014 – Total Marks 12

Q. 55 Define data structure and give it's classification. **(Sections 1.3 and 1.4)** **(2 Marks)**

Q. 56 Enlist various types of operation on data structure. **(Section 1.5)** **(2 Marks)**

Q. 57 Describe the different approach to design an algorithm. **(Section 1.7)** **(4 Marks)**

Q. 58 Describe algorithm analysis in term of time complexity and space complexity. **(Section 1.8)** **(4 Marks)**

➔ Summer 2015 – Total Marks 12

Q. 1(A)(a) Define primitive and non-primitive data structure. *(Ans. : Refer section 1.4.1)* **(2 Marks)**

Q. 1(A)(b) Enlist the operations on data structure. *(Ans. : Refer section 1.5)* **(2 Marks)**

Q. 1 (B)(a) Explain different approaches to design an algorithm. *(Ans. : Refer section 1.7)* **(4 Marks)**

Q. 4(a) Explain time and space complexity with example.
(Ans. : Refer sections 1.8.1 and 1.8.2) **(4 Marks)**

➔ Winter 2015 – Total Marks 12

Q. 1(A)(a) Define time complexity and space complexity.
(Ans. : Refer sections 1.8.1 and 1.8.2) **(2 Marks)**

Q. 1(A)(b) Define data structure. Enlist any two types of non-linear data structure.
(Ans. : Refer sections 1.3 and 1.3.2) **(2 Marks)**

Q. 1(B)(a) Describe any four operations on data structure.
(Ans. : Refer section 1.5) **(4 Marks)**

Q. 4(a) Describe Top-down and bottom-up approach by giving example.
(Ans. : Refer sections 1.7.1 and 1.7.2) **(4 Marks)**

➔ Summer 2016 – Total Marks 08

Q. 1(a) State the need of data structure. Write the operations performed using data structures.
(Ans. : Refer sections 1.3.1 and 1.5) **(4 Marks)**

Q. 4(a) Define an algorithm. How it is analysed?
(Ans. : Refer sections 1.6 and 1.8.1) **(4 Marks)**

➔ Winter 2016 – Total Marks 12

Q. 1(A)(1) Define Big 'O' Notation.
(Ans. : Refer section 1.9) **(2 Marks)**

Q. 1(A)(2) Define data structure and give its classification.
(Ans. : Refer sections 1.3 and 1.4) **(2 Marks)**

Q. 1(B)(1) What is complexity of an algorithm ? Describe time complexity and space complexity.
(Ans. : Refer section 1.8) **(4 Marks)**

Q. 4(a) Compare Top-down approach v/s Bottom-up approach [any four points].
(Ans. : Refer sections 1.7.1 and 1.7.2) **(4 Marks)**

➔ Summer 2017 – Total Marks 16

Q.1(A)(a) Define complexity and classify it.
(Ans. : Refer section 1.8.1) **(4 Marks)**

Q.1(A)(b) State limitations of the Big '0' notation **(2 Marks)**

Ans. :

It does and give a proper, answer when the data set size N is small. When N is small, tell you which algorithm will be faster in anyspecific case.

Q.1(A)(d) Stack is linear data structure. Yes/No ? Justify your answer. **(2 Marks)**

Ans. : Yes, stack is a linear data structure. It can be stored in an arrary and opertation on it requires linear operations.

Q. 1(B)(a) Describe different approaches to design an algorithm. *(Ans. : Refer section 1.7)* **(4 Marks)**

Q. 3(a) Describe time and space trade off and time and space complexity with example of each.
(Ans. : Refer section 1.8.1) **(4 Marks)**

➔ Winter 2017 – Total Marks 12

Q. 1(A)(a) List any four operations performed on data structure. *(Ans. : Refer section 1.5)* **(2 Marks)**

Q. 1(A)(b) Define the term algorithm. Also list approaches to design an algorithm.
(Ans. : Refer sections 1.6 and 1.7) **(2 Marks)**

Q. 1(B)(a) Describe with example, time complexity and space complexity of an algorithm.
(Ans. : Refer sections 1.8.1) **(4 Marks)**

Q. 4(a) Descirbe classification of data structure with example of each.
(Ans. : Refer sections 1.4) **(4 Marks)**

□□□

CHAPTER 2

Unit - II

Searching and Sorting

Syllabus Topics

Searching : searching an item in a data set using following methods :

(i) Linear Search (ii) Binary Search

Sorting : sorting of data set in an order using following methods :

(i) Bubble Sort (ii) Selection Sort

(iii) Insertion Sort (iv) Radix Sort.

2.1 Sorting

➔ **(MSBTE : W-09, S-11, W-11, S-12, W-12, W-13, W-14, S-15, W-15, S-16)**

Q. 2.1.1 What is **sorting ?** Enlist different methods.

(Refer Section 2.1) W-09, W-12

Q. 2.1.2 Define sorting? **Enlist and explain the types of sorting with example.**

(Refer Section 2.1) S-11, W-13, W-14, S-15

Q. 2.1.3 **Explain the concept of sorting algorithms.**

(Refer Section 2.1) W-11

Q. 2.1.4 **What is sorting ? Give its types.**

(Refer Section 2.1) S-12

Q. 2.1.5 **What is sorting ? Enlist two categories of sorting. (Refer section 2.1)** W-15

Q. 2.1.6 **Define internal and external sorting.**

(Refer section 2.1) S-16

Sorting is a process of ordering a list of elements in either ascending or descending order. Sorting can be divided into two categories :

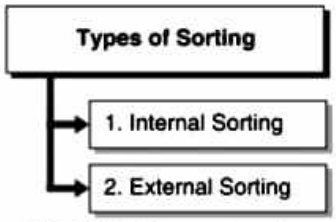


Fig. 2.1.1 : Types of sorting

➔ **1. Internal sorting**

Internal sorting takes place in the main memory of the computer. Internal sorting can take advantage of the random access nature of the main memory. Elements to be sorted are stored in an integer array. These elements can be sorted using one of the various algorithms discussed in this chapter.

➔ **2. External sorting**

External sorting is carried on secondary storage. External sorting becomes a necessity if the number of elements to be sorted is too large to fit in main memory. External sorting algorithms should always take into account that movement of data between secondary storage and main memory is best done by moving a block of contiguous elements.

- Simple sorting algorithms like bubble sort, insertion sort take $O(n^2)$ time to sort n elements.
- More complicated algorithms like quick sort, merge sort, heap sort take O(n log n) time to sort n elements.
- Sorting algorithms like bubble sort, insertion sort, merge sort are stable. Whereas quick sort and heap sort are unstable. A sorting algorithm is said to be **stable** if after sorting, identical elements appear in the same sequence as in the original unsorted list.

2.1.1 Sort Stability

➔ **(MSBTE : W-11)**

Q. 2.1.7 Explain the concept of sorting algorithms. (Refer Section 2.1.1) **W-11**

Let us try to understand the concept of sort stability with the help of an example.

Name	Subject	Marks
Mohan	Phy	65
Sohan	Che	70
Mohan	Che	68
Amit	Che	74
Sohan	Phy	75

(a) Unsorted list

Name	Subject	Marks
Amit	Che	74
Mohan	Phy	65
Mohan	Che	68
Sohan	Che	70
Sohan	Phy	75

(b) Sorted list (stable)

Name	Subject	Marks
Amit	Che	74
Mohan	Che	68
Mohan	Phy	65
Sohan	Che	70
Sohan	Phy	75

Fig. 2.1.2(c) : Sorted list (not stable)

Fig. 2.1.2(a) gives a list of unsorted records. These records are to be sorted on name.

Two records appear with the name 'Mohan'. Similarly there are two records with name 'Sohan'. Any sorting method will place the records with non-distinct keys in consecutive locations. If the original ordering among the records with non-distinct keys is preserved in sorted file then such a sorting method is known as stable.

Sorted list of Fig. 2.1.2(b) has been produced using a stable sorting method. Relative ordering of two records of 'Mohan' and two records of 'Sohan' is preserved in sorted list.

Sorted list of Fig. 2.1.2(c) has been produced using a sorting method which is not stable. Relative ordering of two records of 'Mohan' is not preserved in sorted list.

Some examples of stable sorting algorithms are :

(1) Bubble sort (2) Selection sort

(3) Insertion sort (4) Merge sort

Some examples of unstable sorting algorithms are :

(1) Quick sort (2) Heap sort

(3) Shell sort

2.1.2 Sort Efficiency

➔ **(MSBTE : W-11, W-12)**

Q. 2.1.8 Explain the concept of sorting algorithms. (Refer Section 2.1.2) **W-11**

Q. 2.1.9 Explain efficiency of sorting algorithm. (Refer Section 2.1.2) **W-12**

In order to find the amount of time required to sort a list of n elements by a particular method, we must find the number of comparisons required to sort. Some methods are data sensitive and for such methods finding of exact number of comparisons become difficult. Usually, data sensitive algorithms are analyzed for various cases.

(1) Best case

(2) Worst case

(3) Average case

Average case of an algorithm is calculated with an assumption that data distribution is random. Most of the primitive sorting algorithms like bubble sort, selection sort and insertion sort are not suited for sorting a large file. These algorithms have a timing requirement of $O(n^2)$. On the contrary, these sorting algorithms require hardly any additional memory space for sorting. A list with few records should preferably be sorted using these algorithms. Advance sorting algorithms like quick sort, merge sort and heap sort have a timing requirement of O(n log n) but they too have some additional problems as :

(1) Complex algorithm

(2) Complex data structure

(3) Unstable sorting algorithm

(4) Additional memory requirement

(5) Data sensitivity

These algorithms should be used when the list to be sorted contains considerably large number of records.

Syllabus Topic : Sorting : Sorting of Data Set in an Order using Methods

2.1.3 Passes

➔ **(MSBTE : W-11)**

Q. 2.1.10 Explain the concept of sorting algorithms.
(Refer Section 2.1.3) **W-11**

Most of the sorting algorithms work in passes. In every pass, a number is placed at position where it will appear in the sorted list. Inside a pass, numbers are compared and exchanged as required by the algorithm.

For example, if a list of a n elements is to be sorted using bubble sort then sorting of n elements will require n – 1 passes. If the elements are stored in an array from a [0] to a [n – 1] then at the end of pass 1, largest element will be in a [n – 1] at the end of pass 2, second largest element will be in a [n – 2].

At the end of pass n – 1, the smallest element will be in a [0].

In the loop of bubble sort, given below

```
for (i = 1; i < n; i++)
    for (j = 0, j < n – i; j++)
    {
    }
```

Outer loop on i is for passes. Inner loop is for comparison and exchange (if required) of two adjacent elements.

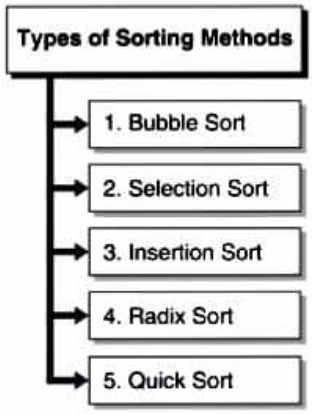


Fig. 2.1.3 : Types of sorting methods

Syllabus Topic : Bubble Sort

2.2 Bubble Sort

➔ **(MSBTE : W-09, W-10, W-11, S-15)**

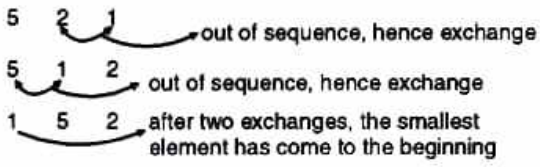
Q. 2.2.1 Describe bubble sort with the help of example. Take minimum four values.
(Refer section 2.2) **W-09**

Q. 2.2.2 Write an algorithm for bubble sort.
(Refer section 2.2) **W-10**

Q. 2.2.3 Explain the sorting technique : Bubble sort
(Refer section 2.2) **W-11**

Q. 2.2.4 Give the principal of bubble sort.
(Refer section 2.2) **S-15**

Bubble sort is one of the simplest and the most popular sorting method. The basic idea behind bubble sort is as a bubble rises up in water, the smallest element goes to the beginning. This method is based on successive selecting the smallest element through exchange of adjacent element.



First pass of the bubble sort

Let n be the number of elements in an array a[ ]. The first pass begins with the comparison of a[n – 1] and a[n – 2]. If a[n – 2] is larger than a[n – 1], the two elements are exchanged. The smaller elements, now at a[n – 2] is compared with a[n – 3] and if necessary the elements are exchanged to place the smaller one in a[n – 3]. Comparison progresses backward and after the last comparison of A[1] and A[0] and possible exchange the smallest element will be placed at a[0].

As a variation, the first pass could begin comparison with a[0] and a[1]. If a[0] is larger than a[1], the two elements are exchanged. Larger one will be placed in a[1] after the first comparison. Comparison can work forward and after the last comparison of a[n – 2] and a[n – 1], the larger element will be placed in a[n –1]

5 2 1 out of sequence, hence exchange
2 5 1 out of sequence, hence exchange
2 1 5 after two exchanges, the largest element has come to the end.

First pass of the bubble sort with comparison in the forward direction.

The **second pass** is an exact replica of the first pass except that this time, the pass ends with the comparison and possible exchange of a[n – 3] and a[n – 2]. After the end of second pass, the second largest element will be placed at a[n – 2].

Bubble sort requires a total of n – 1 passes. If n – 1 elements are arranged (one element each pass) in ascending order from a[1] to a[n – 1], smallest element will finally left at a[0].

☞ **Loop for sorting an array a[ ] having n elements**

```
for(i = 1; i < n; i ++)
    for(j = 0; j < n – 2i; j ++)
        if(a[j] > a[j + 1])
            exchange a[j] and a[j + 1]
```

Outer loop is for passes and **the inner loop is for** comparisons.

In pass 1 (i = 1), there will **be n – 1 comparisons** (j = 0 to n – 2)

In pass 2 (i = 2), there **will be n – 2 comparisons (j = 0 to** n – 3)

:

In pass n – 1 (i = n – 1), there will be 1 comparison (j = 0 to 0)

Original array with n = 6 5 9 6 2 8 1

First pass i = 1

Comparisons
j = 0 5 9 6 2 8 1
j = 1 5 9 6 2 8 1
j = 2 5 6 9 2 8 1
j = 3 5 6 2 9 8 1
j = 4 5 6 2 8 9 1

Second pass i = 2

Comparisons
j = 0 5 6 2 8 1 9
j = 1 5 6 2 8 1 9
j = 2 5 2 6 8 1 9
j = 3 5 2 6 8 1 9

Third pass i = 3

Comparisons
j = 0 5 2 6 1 8 9
j = 1 2 5 6 1 8 9
j = 2 2 5 6 1 8 9

Fourth pass i = 4

Comparisons
j = 0 2 5 1 6 8 9
j = 1 2 5 1 6 8 9

Fifth pass i = 5

Comparisons
j = 0 2 1 5 6 8 9

Sorted array a[ ] → **12 5 6 8 9**

Illustration of bubble sort.

➢ **Program 2.2.1 : Program for sorting an integer array using bubble sort.**

Q. 2.2.5 Write a program for bubble sort in 'C/C++' language.
(Refer Program 2.2.1) **W-08, W-13, S-14**

Q. 2.2.6 Write a program to sort an array of ten elements with bubble sort.
(Refer Program 2.2.1) **S-10, S-13, W-14**

```
#include<conio.h>
#include<stdio.h>
void bubble_sort(int[],int);
void main()
{
    int a[30],n,i;
    printf("\nEnter no of elements :");
    scanf("%d",&n);
    printf("\nEnter array elements :");
    for(i=0;i<n;i++)
        scanf("%d",&a[i]);
    bubble_sort(a,n);
        printf("\nSorted array is :");
    for(i=0;i<n;i++)
        printf("%d",a[i]);
    getch();
}
void bubble_sort(int a[],int n)
{
    int i,j,temp;
    for(i=1;i<n;i++)
        for(j=0;j<n-i;j++)
            if(a[j]>a[j+1])
            {
              temp=a[j];
              a[j]=a[j+1];
              a[j+1]=temp;
            }
}
```

Output

```
Enter no of elements : 6
Enter array elements :    57 89 64 56 77 333
Sorted array is :   56 57    64    77 89 333
```

Improved version of bubble sort

In our algorithm for bubble sort, sorting algorithm requires n – 1 passes. The method may even be terminated earlier if no exchange is found necessary in an earlier pass. Absence of any exchange in a pass ensures that the elements are already sorted and there is no point in continuing further.

- Absence of any exchange in inner loop of bubble sort can be detected through a variable flag.
- Before entering the inner loop, flag is set to 0(flag = 0)
- If an exchange is required then flag is set to 1 in inner loop.
- On exiting the inner loop, if the value of the flag is found to be 0, algorithm terminates.

➢ **'C' function for improved bubble sort.**

```
void imp_bubble sort(int a[ ], int n)
{
    int i, j, temp, flag;
    flag = 1;
    for(i = 1; i < n && flag = = 1; i ++)
    {
        flag = 0
        for(j = 0; j < n – i; j ++)
        if(a[j] > a[j + 1])
        {
            flag = 1;
            temp = a[j] ;
            a[j] = a[j + 1];
            a[j + 1] = temp;
        }
    }
}
```

Analysis of bubble sort

Let us consider the loop for bubble sort :

```
Line 1    for(i = 1; i < n; i + 1)
Line 2        for(j = 0; j < n – i; j ++)
Line 3            if(a[j] > a[j + 1])
```

Line 4 exchange a[j] and a[j + 1];

For a fixed value of i, the loop of lines(2 – 4) runs n – i times.

Hence total number of comparisons

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

$$= (n-1) + (n-2) + \ldots\ 3 + 2 + 1$$

$$= \frac{n(n-1)}{2}$$

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

Example 2.2.1 : **Sort the following numbers in ascending order using Bubble sort. A = {19, 2, 27, 3, 7, 5, 31}** **S-12**

Solution :

Pass-I

19 2 27 3 7 5 31
2 19 27 3 7 5 31
2 19 27 3 7 5 31
2 19 3 27 7 5 31
2 19 3 7 27 5 31
2 19 3 7 5 27 31
2 19 3 7 5 27 31

Pass-II

2 19 3 7 5 27 31
2 19 3 7 5 27 31
2 3 19 7 5 27 31
2 3 7 19 5 27 31
2 3 7 5 19 27 31
2 3 7 5 19 27 31

Pass-III

2 3 7 5 19 27 31
2 3 7 5 19 27 31
2 3 7 5 19 27 31
2 3 5 7 19 27 31
2 3 5 7 19 27 31

Pass-IV

2 3 5 7 19 27 31
2 3 5 7 19 27 31
2 3 5 7 19 27 31
2 3 5 7 19 27 31

Pass-V

2 3 5 7 19 27 31
2 3 5 7 19 27 31
2 3 5 7 19 27 31

Pass-VI

2 3 5 7 19 27 31
2 3 5 7 19 27 31

Syllabus Topic : Selection Sort

2.3 Selection Sort

➔ **(MSBTE : W-08, W-11, W-12, W-15)**

Q. 2.3.1 Describe the principle of selection sort with one example. (Refer Section 2.3) **W-08**

Q. 2.3.2 Explain the sorting technique : Selection sort (Refer Section 2.3) **W-11**

Q. 2.3.3 Describe selection sort with algorithm and one example, assume suitable data. (Refer Section 2.3) **W-12**

Q. 2.3.4 Write complexity of selection sort. (Refer section 2.3) **W-15**

Selection sort is a very simple sorting method. In the ith pass, we select the element with lowest value among a[i], a[i + 1] … , a[n – 1] and we swap it with a[i]. As a result, after i passes (pass number 0 to i – 1) first i elements will be in sorted order. Selection sort can be described by

```
for(i = 0; i < n – 1; i ++)
    select the smallest element among
        a[i], …. , a[n –1] and
    swap it with a[i];
```

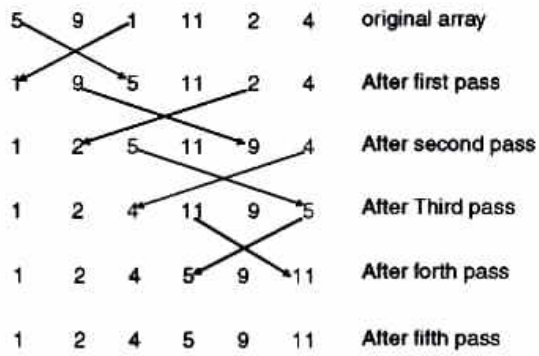


Illustration of selection sort

➢ **Program 2.3.1 : A sample program for selection sort.**

Q. 2.3.5 Write a program in 'C' language for selection sort. (Refer Program 2.3.1) **S-08, S-10, W-12, S-13**

Q. 2.3.6 Write a program to arrange 10 elements in an array in ascending order. (Refer Program 2.3.1) **S-14**

Q. 2.3.7 Write a 'C' program for the selection sort. (Refer Program 2.3.1) **S-15, S-17**

Q. 2.3.8 Write a program in 'C' language for selection sort. (Refer Program 2.3.1) **W-15**

Q. 2.3.9 Write a program in 'C' to sort the given numbers in descending order using selection sort. (Refer Program 2.3.1) **S-16**

```
#include <stdio.h>
#include <conio.h>
void selection_sort(int [],int);
void main()
{
    int a[50],n,i;
    clrscr();
    printf("\nEnter no of elements :");
    scanf("%d",&n);
    printf("\nEnter array elements :");
    for(i=0;i<n;i++)
       scanf("%d",&a[i]);
    selection_sort(a,n);
    getch();
}
void selection_sort(int a[],int n)
{
    int i,j,k,temp;
    printf("\nUnsorted Data:");
    for(k=0;k<n;k++)
        printf("%5d",a[k]);
    for(i=0;i<n-1;i++)
     {
      k=i;
      for(j=i+1;j<n;j++)
       if(a[j]<a[k])
           k=j;
      if(k!=i)
       {
        temp=a[i];
        a[i]=a[k];
        a[k]=temp;
       }
     }
  printf("\nSorted data:");
  for(k=0;k<n;k++)
    printf("%5d",a[k]);
}
```

☞ Analysis of selection sort

Selection sort is not data sensitive. In i^{th} pass, $n - i$ comparisons will be needed to select the smallest element.

Thus, the number of comparisons needed to sort an array having n elements.

$$= (n-1) + (n-2) + \ldots + 2 + 1$$

$$= \frac{n(n-1)}{2}$$

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

Example 2.3.1 : Compare the three sorting algorithms
(a) Bubble sort (b) Selection sort
(c) Insertion sort

Solution :

A sorting algorithm can be judged on the basis of following parameters :

1. **Simplicity of algorithm**

   All the three sorting algorithms are equally simple to write.

2. **Timing complexity**

   Bubble sort and selection sort are not data sensitive. Both of them have a timing requirement of $O(n^2)$.

   Insertion sort is data sensitive. It works much faster if the data is partially sorted.

   Best case behaviour (when input data is sorted) = $O(n)$.

   Worst case behaviour (when input data is in descending order) = $O(n^2)$.

3. **Sort stability**

   All the three sorting algorithms are stable.

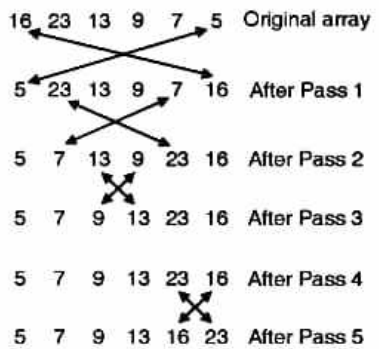
4. **Storage requirement**

   No additional storage is required. All of them are in-place sorting algorithms.

5. All of them can be used for internal as well as external sorting.

Example 2.3.2 : Sort the following numbers using selection sort.Numbers :
16, 23, 13, 9, 7, 5 **S-08, S-10, S-13**

Solution :



Example 2.3.3 : **Sort the number in ascending order using selection sort A = { 42, 23, 74, 11, 65 }.** **W-13**

Solution :

42	23	74	11	65	Original array
11	23	74	42	65	After pass 1
11	23	42	74	65	After pass 2
11	23	42	65	44	After pass 3

Syllabus Topic : Insertion Sort

2.4 Insertion Sort

➔ **(MSBTE : W-08, S-09, W-12, W-16)**

Q. 2.4.1 **Write the algorithm for Insertion sort.**
(Refer Section 2.4) **W-08, S-09**

Q. 2.4.2 **Give the principles of insertion sort along with algorithm. (Refer Section 2.4)** **W-12**

Q. 2.4.3 **Describe working of inserting sort. Demonstrate working of insertion sort algorithm to sort 6 elements.**
(Refer section 2.4) **W-16**

An element can always be placed at a right place in sorted list of element for example

List of elements(sorted) 5 9 10 15 20

Element to be placed = 6

If the element 6 is to be inserted in a sorted list of elements (5, 9, 10, 15, 20), its rightful place will be between 5 and 9. Elements with value > 6 should be moved right by one place. Thus creating a space for the incoming element.

5 [9 10 15 20] → Moved right by 1 place

5 6 9 10 15 20 → Element 6 is inserted between 5 and 9

Insertion sort is based on the principle of inserting the element at its correct place in a previously sorted list. i can be varied from 1 to n – 1 to sort the entire array.

Index - 0 1 2 3 4 5 6 Initial unsorted list

[5] [0 1 9 2 6 4]

A list of sorted element(a list of single element is always sorted)

a list of unsorted element

1st iteration (Place element at location '1' i.e. a[1], at its correct place)

0 1 2 3 4 5 6

[0 5] [1 9 2 6 4]

Sorted Unsorted

2nd iteration (Place a[2] at its correct place)

[0 1 5] [9 2 6 4]

Sorted Unsorted

3rd iteration (Place a[3] at its correct place)

[0 1 5 9] [2 6 4]

Sorted Unsorted

4th iteration (Place a[4] at its correct place)

[0 1 2 5 9] [6 4]

sorted Unsorted

5th iteration (Place a[5] at its correct place)

[0 1 2 5 6 9] [4]

Sorted Unsorted

6th iteration (Place a[6] at its correct place)

0 1 2 4 5 6 9

Fig. 2.4.1 : Sorting of elements using insertion sort

Insertion sort requires n-1 passes to sort an array having n elements. Before the ith pass starts, elements at position 0 through i-1 are already sorted.

Index → 0 1 2 3 4 5 6

a List → [1 5 9 11] [6 2 5 1]

↑

at the beginning of pass = 4

Element at position i = 4 can be inserted at its correct place after he following operations :

temp = a[4]

a[4] = a[3]

a[3] = a[2]

a[2] = temp

All elements larger than 6 are moved right by 1 place. i^{th} element is saved in the variable temp to protect its value before it is overwritten due to data movement.

i^{th} element can also be inserted at its correct place with the help of the following program segment.

```
temp = a[i] ;
for( j = i – 1 ; j > = 0 && temp < a[ j] ; j--)
    a[ j + 1] = a[j]; /*move input */
a[ j + 1] = temp;
```

➢ **'C' function for insertion sort**

```
void insertion_sort(int a[ ], int n)
{
    int i, j, temp;
    for(i=1;i<n;i++)        /* passes 1 through n – 1 */
    {
        temp = a[i];
        for( j = i – 1; j > = 0 && a[ j] > temp; j --)
            a[j + 1] = a[ j];
        a[ j + 1] = temp;
    }
}
```

☞ **Analysis of insertion sort**

For loop of lines (3 – 4) will be executed $\sum_{i-1}^{n-1} i$ times under its worst case behaviour. If the number to be sorted are initially in descending order then the for loop of lines (3 – 4) will make i iterations during pass i.

Elements 20 10 8 6 4 2 1 – initially

								Pass	Positions moved
10	20	8	6	4	2	1		1	1
8	10	20	6	4	2	1		2	2
6	8	10	20	4	2	1		3	3
4	6	8	10	20	2	1		4	4
2	4	6	8	10	20	1		5	5
1	2	4	6	8	10	20		6	6

Thus, the total number of movement of data (if the list is initially in descending order) for sorting using insertion sort = $\sum_{i=1}^{n-1} i$

$$= \frac{n(n-1)}{2} = 0(n^2).$$

Worst case behaviour $O(n^2)$

If the input list presented for sorting is presorted the running time will be O(n), because the test a[j] > temp in the inner loop (lines 3 – 4) will fail immediately. Thus only one comparison is made in each pass.

Total number of comparisons = n – 1

Best case behaviour O(n)

Insertion sort can exploit the partially sorted nature of data. Insertion sort is highly efficient if the array is already in almost sorted order.

➢ **Program 2.4.1 : A sample program for insertion sort (for an array of integers)**

Q. 2.4.4 Write a program for insertion sort. (Refer Program 2.4.1) **S-08**

```
#include<conio.h>
#include<stdio.h>
void insertion_sort(int[],int);
void main()
{
    int a[50],n,i;
    printf("\nEnter no. of elements :");
    scanf("%d",&n);
    printf("\nEnter array elements :");
```

```
    for(i=0;i<n;i++)
        scanf("%d",&a[i]);
    insertion_sort(a,n);
   printf("\nSorted array is :");
    for(i=0;i<n;i++)
        printf("%d",a[i]);
    getch();
}
void insertion_sort(int a[],int n)
{
    int i,j,temp;
    for(i=1;i<n;i++)
    {
        temp=a[i];
        for( j=i-1;j>=0 && a[ j]>temp;j--)
            a[ j+1]=a[ j];
        a[ j+1]=temp;
    }
}
```

Output

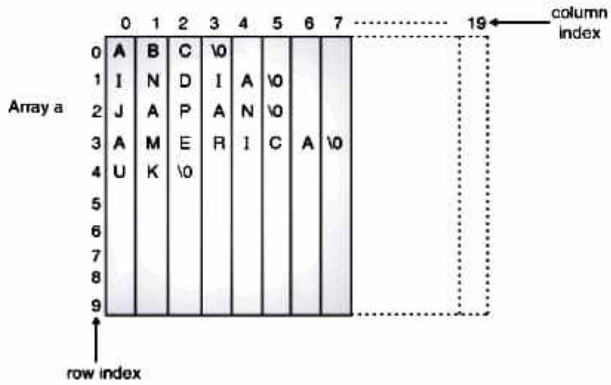
```
Enter no of elements  :  6
Enter array elements  :  57   89  64  56 77  333
Sorted array is       :  56   57  64 77 89  333
```

2.4.1 Sorting an Array of Strings using Insertion Sort

A list of strings can be represented using a two dimensional array.

For example : A list {"ABC", "INDIA", "JAPAN", "AMERICA", "UK"} of strings can be represented as given below :

```
char a[10][20];
```



a[0] is the string "ABC"

a[1] is the string "INDIA"

a[2] is the string "JAPAN"

a[3] is the string "AMERCIA"

a[4] is the string "UK"

- Two numbers a[j] and temp are compared using the relational operation "a[j] > temp". Whereas the two string a[j] and temp should be compared using the library function strcmp(a[j], temp).
- Function strcpy(a[j+1], a[j]) should be used to move the string from location j to j+1 in the array.

➢ **Program 2.4.2 : Program showing sorting of strings using insertion sort.**

```
#include<conio.h>
#include<stdio.h>
#include<string.h>
void insertion_sort(char[][20],int);
void main()
{
    char a[10][20];
    int i,n;
    printf("\n Enter no of strings :");
    scanf("%d",&n);
    printf("\nEnter strings :");
    for(i=0;i<n;i++)
        gets(a[i]);
    insertion_sort(a,n);
```

```
    printf("\n Sorted strings :\n");
    for(i=0;i<n;i++)
        printf("%s",a[i]);
    getch();
}
void insertion_sort(char a[][20],int n)
{
    int i,j;
    char temp[20];
    for(i=0;i<n;i++)
    {
        strcpy(temp,a[i]);
        for(j=i-1;j>=0 && strcmp(a[j],temp)>0;j--)
            strcpy(a[j+1],a[j]);
        strcpy(a[j+1],temp);
    }
}
```

Output

```
Enter no of strings   : 4
Enter strings    : xyz abcccc aaa
Sorted strings   : aaa abcccc xyz
```

2.4.2 Sorting an Array of Records on the given Key using Insertion Sort

A company maintains record of its employees. Each record has the following format :

ENO	NAME	OCCUPATION	LOCATION

It is required to sort employee records on "ENO".

- **Program 2.4.3 : Program to sort employee records on "ENO".**

```
#include<stdio.h>
#include<conio.h>
struct employee
{
    int ENO;
    char name[30];
    char occup[15];
    char loca[20];
};
void main()
{
    struct employee temp;
    int i,j,n;
    static struct employee
employees[]={{15,"MOHAN","PROG","DELHI"},
                {5,"RAM","ANAL","PUNE"},
                {12,"JOHN","MANA","MUMBAI"},
                {3,"SOHAN","PROG","NASIK"}
            };
    n=4;
    for(i=1;i<n;i++)
    {
        temp=employees[i];
        for(j=i-1;j>=0 && employees[j].ENO >
temp.ENO;j--)
            employees[j+1]=employees[j];
        employees[j+1]=temp;
    }
    for(i=0;i<n;i++)
    {
    printf("\n%d\t%7s\t%5s\t%7s",employees[i].ENO,
employees[i].name, employees[i].occup,employees[i].loca);
        getch();
    }
}
```

Output

```
3     SOHAN   PROG    NASIK
5     RAM     ANAL    PUNE
```

12 JOHN MANA MUMBAI

15 MOHAN PROG DELHI

➢ **Program 2.4.4 : Program to insert an element in an array.**

Q. 2.4.5 Write a 'C' program to insert an element in an array. (Refer Program 2.4.4) **W-13**

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a[30],x,n,i,loc;
    /*
        x  - element to be inserted
        n  - no.of elements in the array
        i  - for scanning of the array
        loc - place where the new element is to be inserted */
    printf("\n Enter no of elements :");
    scanf("%d",&n);
    /* read n elements in an array */
    for(i=0;i<n;i++)
        scanf("%d",&a[i]);
    printf("\n Enter the element to be inserted :");
    scanf("%d",&x);
    printf("\n Enter the location");
    scanf("%d",&loc);
    /* create space at the specified location */
    for(i=n-1;i>=loc-1;i—)
        a[i+1]=a[i];
    n++;
    a[loc-1]=x;    /* Element inserted */
    /* Printing of result */
    for(i=0;i<n;i++)
        printf("\n %d",a[i]);
}
```

Output

```
Enter no of elements :4
12  13  14  15
Enter the element to be inserted :20
Enter the location 2
12  20  13  14  15
```

☞ **Example**

Here are five integers 1, 7, 3, 2, 0. Sort them using insertion sort.

Pass (i)	Comparisons (j)	List to sort	Remarks
		1 7 3 2 0	original list
i = 1	j = 0	1 7 3 2 0	7 > 1, therefore inner loop terminates
i = 2	j = 1	1 7 3 2 0	7 > 3, move 7 right
	j = 0	1 7 7 2 0 1 3 7 2 0	3 > 1, inner loop terminates insert 3
i = 3	j = 2 j = 1 j = 0	1 3 7 7 0 1 3 3 7 0 1 3 3 7 0 1 2 3 7 0	7 > 2, move 7 right 3 > 2, move 2 right 2 > 1, inner loop terminates insert 2
i = 4	j = 3 j = 2 j = 1 j = 0 j = – 1	1 2 3 7 7 1 2 3 3 7 1 2 2 3 7 1 1 2 3 7 1 1 2 3 7 0 1 2 3 7	7 > 0 3 > 0 2 > 0 1 > 0 j = – 1, inner loop terminates insert 0

Hence the sorted list = {0, 1, 2, 3, 7}

Example

Show that the average case running time of insertion sort is $O(n^2)$

Number of comparisons (average) required to locate the point of insertion of the ith element.

$$= \frac{1}{i}[1 + 2 + 3 + \ldots + i] = \frac{1}{i} \times \frac{i[i+1]}{2} = \frac{i+1}{2}$$

Since, in the outer loop i vary from 1 to n – 1.

∴ Total number of comparisons

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

$$= \frac{1}{2}\left[(n-1) + \frac{n(n-1)}{2}\right]$$

$$= \frac{1}{2}\left[\frac{2(n-1) + n(n-1)}{2}\right]$$

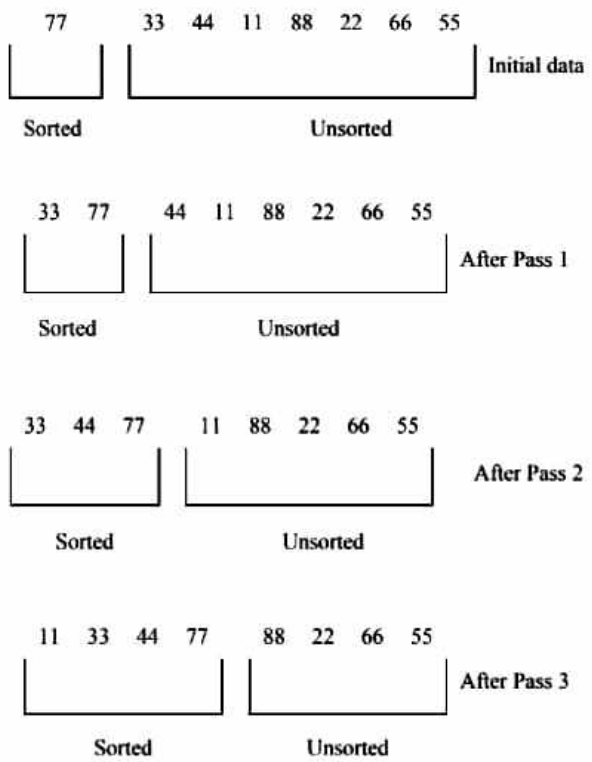
$$= \frac{1}{4}[(n+2)(n-1)]$$

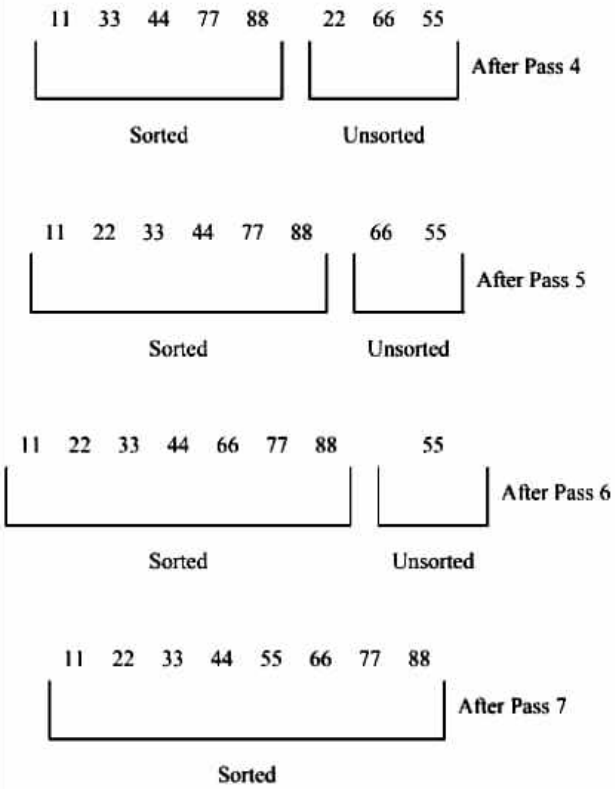
$$= \frac{1}{4}[n^2 + n - 2]$$

$$= O(n^2)$$

Example 2.4.1 : Arrange the given numbers in ascending order using insertion sort.Numbers : 77, 33, 44, 11, 88, 22, 66, 55 **W-08**

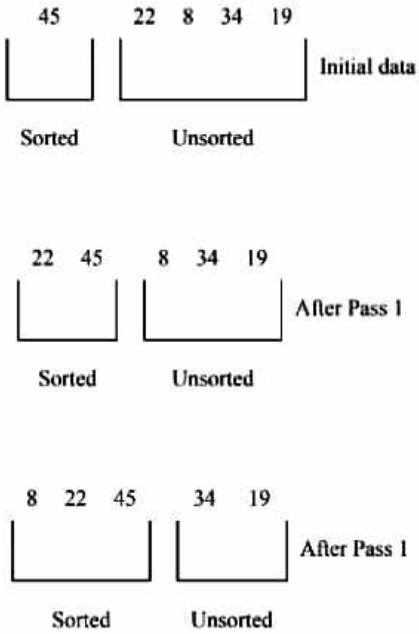
Solution :

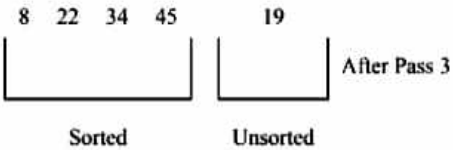


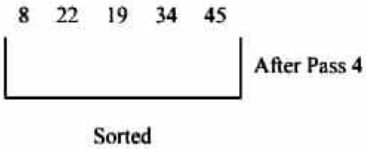


Example 2.4.2 : Arrange the given numbers in ascending order using insertion sort.Numbers : 45, 22, 8, 34, 19 **S-09**

Solution :

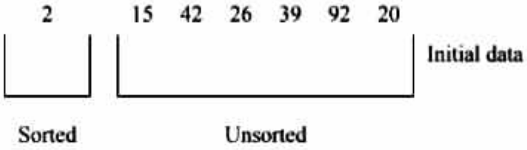


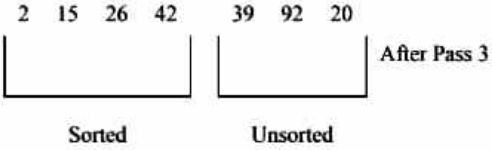


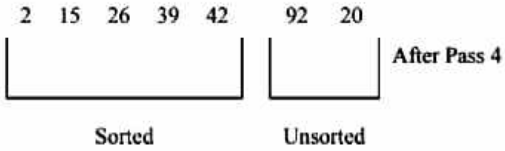


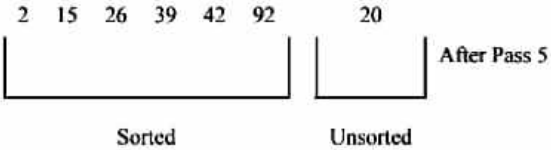
Example 2.4.3 : **Describe insertion sort algorithm and give steps of insertion sort for sorting the following list in ascending order. List : 2, 15, 42, 26, 39, 92, 20 also find total number of comparison made.** W-14

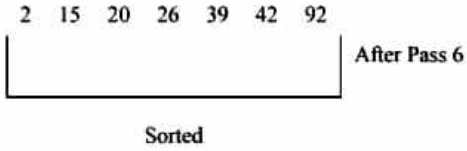
Solution :







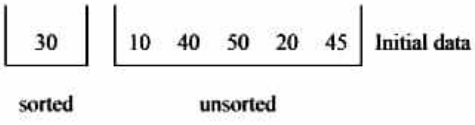


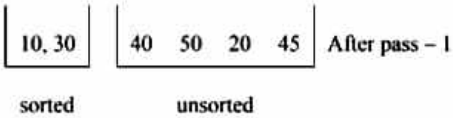


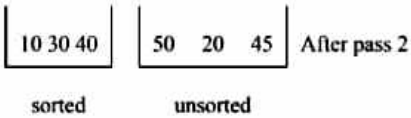
Example 2.4.4 : **Elaborate the steps for performing insertion sort for given elements of array.**

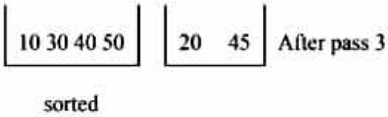
30 10 40 50 20 45 S-17

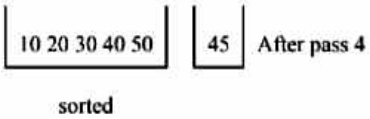
Solution :











10 20 30 40 45 50	After pass 45

sorted

Syllabus Topic : Radix Sort

2.5 Radix Sort

➔ **(MSBTE : S-08, W-10, W-13, W-15)**

Q. 2.5.1 **Explain radix sort with example.**
(Refer Section 2.5) **S-08**

Q. 2.5.2 **Describe radix sort algorithm along with an example and state its complexity.**
(Refer Section 2.5) **W-10**

Q. 2.5.3 **Describe the working principle of radix sort ?**
(Refer Section 2.5) **W-13**

Q. 2.5.4 **Describe radix sort algorithm.**
(Refer section 2.5) **W-15**

Radix sort is generalization of bucket sort. To sort decimal numbers, where the radix or base is 10, we need 10 buckets. These buckets are numbered 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Sorting is done in passes.

Number of passes required to sort using shell sort is equal to number of digits in the largest number in the list.

Range	Passes
0 to 99	2 passes
0 to 999	3 passes
0 to 9999	4 passes

- In the first pass, numbers are sorted on least significant digit. Numbers with the same least significant digit are stored in the same bucket.
- In the 2nd pass, numbers are sorted on the second least significant digit.
- At the end of every pass, numbers in buckets are merged to produce a common list.
- Number of passes depends on the range of numbers being sorted.

The following example shows the action of radix sort.

Initial numbers

10 5 99 105 55 100 135 141 137 200 199

Buckets after 1st pass

Bucket no.	0	1	2	3	4	5	6	7	8	9
	200					135				
	100					55				
	10	141				105				199
						5		137		99

Merged list = 10 100 200 141 5 105 55 135 137 99 199

Buckets after 2nd pass

0	1	2	3	4	5	6	7	8	9
100									
5									
200			137						199
100	10		135	141	55				99

Merged list = 100 200 5 105 10 135 137 141 55 99 199

Buckets after 3rd pass

0	1	2	3	4	5	6	7	8	9
	199								
	141								
99	137								
55	135								
10	105								
5	100	200							

Merged list = 5 10 55 99 100 105 135 137 141 199 200

2.5.1 Algorithm for Radix Sort

/*numbers to be sorted are in array a[ ] and there are N numbers */

1. large = find the largest number in a [ ]
2. passes = Number of digits in large
3. Div = 1 /* divisor for extracting i^{th} least significant digit */
4. for (i = 1; i < = passes; i++)

   {

   1. Initialize all buckets b0 to b9
   2. for each number x from a[0] to a [N – 1]

      {

      1. bucket_no = (x/div)%10
      2. insert x in bucket with bucket_number

      }

   3. Copy elements of buckets back in array a[ ]
   4. DIV = DIV * 10

   }

5. exit

2.5.2 C - Function for Radix Sort

Input to functions are :

n = number of elements in the array a [ ]

a[ ] = Array holding elements.

Buckets are represented using a two dimensional array buckets [10] [10]. Each bucket can hold upto ten values. Another array count [10] gives number of elements in the corresponding

bucket. If buckets [1] has five elements then the value of count [1] will be five.

Whenever, an element is inserted in i^{th} bucket, count [i] should be incremented by 1.

```
void radix (int a[ ], int n)
    {   int buckets[10][10], count[10];
        int passes, large, div, bucketno, i, j, k;
    /* find the largest number in a [ ]  */
        large = a[0];
        for (i = 1; i < n; i++)
            if (a[ i] > large)
                large = a[i];
/* find number of digits in large and store the count in passes
*/
    passes = 0;
    while (large > 0)
        {   passes++;
            large = large /10;
        }
    /* perform radix sort */
        div = 1;
    for (i = 1; i < = passes; i++)
    { /*initialize bucket */
        for ( j = 0; j < = 9; j++)
            count [ j] = 0;
    /* insert elements in respective buckets */
        for ( j = 0; j < n; j++)
        {   bucketno = (a[ j]/div)%10;
            buckets[bucketno][count[bucketno]] = a[ j];
            count [bucketno]++;
        }
/* merge elements of buckets and store them in a[0] ...
a[n – 1] */
    j = 0;
    for (bucketno = 0; bucketno < = 9; bucketno++)
        for (k=0; k < count[bucketno]; k++)
            a [ j++] = buckets[bucketno] [k];
            div = div * 10;
    }
}
```

- **Program 2.5.1 : Program to sort elements of an array using radix Sort**

```
#include <stdio.h>
#include <conio.h>
void radix(int a[ ], int n);
void main( )
    {   int a[30], n, i;
        printf("\n enter no. of elements");
        scanf("%d",&n);
        printf("\n enter data :");
        for (i = 0; i < n; i++)
            scanf("%d", &a[i]);
            radix (a, n);
        printf("\n sorted data :");
        for (i = 0; i < n;  i++)
            printf("%d\t", a[i]);
    }
void radix (int a[ ], int n)
    {   int buckets[10][10], count[10];
        int passes, large, div, bucketno, i, j, k;
    /* find the largest number in a[ ]  */
        large = a[0];
        for (i = 1; i < n; i++)
            if (a[ i] > large)
                large = a[i];
 /* find number of digits in large and store the count in passes
*/
    passes = 0;
    while(large > 0)
```

```
        {   passes++;
            large = large /10;
        }
    /* perform radix sort */
    div = 1;
    for (i = 1; i < = passes; i++)
    { /*initialize bucket */
        for ( j = 0; j < = 9; j++)
            count [j] = 0;
    /* insert elements in respective buckets */
        for ( j = 0; j < n; j++)
        {   bucketno = (a[j]/div)%10;
            buckets[bucketno][count[bucketno]] = a[ j];
            count [bucketno]++;
        }
/* merge elements of buckets and store them in a[0] ...
a[n – 1]*/
    j = 0;
    for (bucketno = 0; bucketno < = 9; bucketno++)
        for (k=0; k < count[bucketno]; k++)
            a [ j++] = buckets[bucketno] [k];
            div = div * 10;
    }
```

Output

```
enter no. of elements  : 11
enter data      : 10  5 99 105  55  100 135 141 137   200 199
sorted data     : 5  10  55  99  100 105 135 137 141  199 200
```

2.5.3 Analysis of Radix Sort

If the number to be sorted are k digits long then inside every pass :

- Distribution of elements among buckets take O(n) time
- Merging of elements take O(n) time

Number of passes = k

∴ Overall complexity is O(kn)

if k is regarded as a constant

then

O(kn) = O(n).

Thus, timing complexity of Radix Sort = O(n)

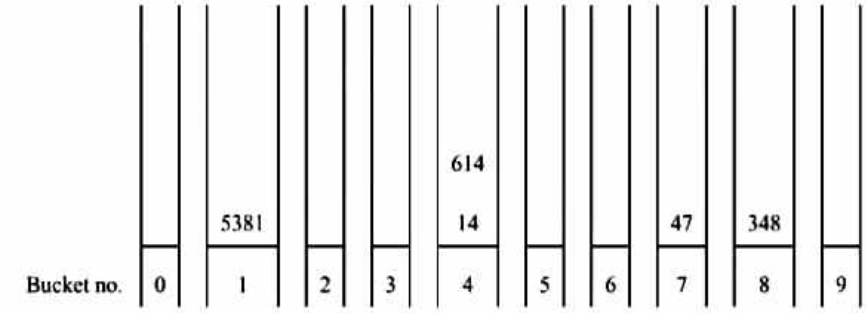
Space requirement : There is additional memory requirement for buckets and storing number of elements in each bucket.

Radix sort is the most preferred algorithm for sorting of strings

Example 2.5.1 : Sort the given numbers in ascending order using Radix sort. Numbers : 348, 14, 614, 5381, 47 **W-08, S-15**

Solution :

Buckets after pass 1



Merged list : 5381, 14, 614, 47, 348

Buckets after pass 2

Bucket no.	0	1	2	3	4	5	6	7	8	9
		614 14			348 47				5381	

Merged list : 14, 614, 47, 348, 5381

Bucket after pass 3

Bucket no.	0	1	2	3	4	5	6	7	8	9
	47 14			5381 348			614			

Merged list = 14, 47, 348, 5381, 614

Buckets after pass 4

Bucket no.	0	1	2	3	4	5	6	7	8	9
	614 348 47 14					5381				

Merged list = 14, 47, 348, 614, 5381

Example 2.5.2 : Sort the given numbers in ascending order using radix sort. 361, 12, 527, 143, 9, 768, 348 **S-09**

Solution :

Buckets after pass 1

Bucket no.	0	1	2	3	4	5	6	7	8	9
		361	12	143				527	348 768	9

Merged list = 361, 12, 143, 527, 768, 348, 9

Buckets after pass 2

Bucket no.	0	1	2	3	4	5	6	7	8	9
	9	12	527		348 143		768 361			

Merged list : 9, 12, 527, 143, 348, 361, 768

Buckets after pass 3 :

Bucket no.	0	1	2	3	4	5	6	7	8	9
	12 9	143		361 348		527		768		

Merged list : 9, 12, 143, 348, 361, 527, 768.

Example 2.5.3 : Sort the following numbers in ascending order using radix sort. 12, 8, 25, 4, 66, 2, 98, 225. **W-15**

Solution :

Given data : 12, 8, 25, 4, 66, 2, 98, 225

Buckets after 1st pass

0	1	2	3	4	5	6	7	8	9
		2 12		4	225 25	66		98 8	

Merged list : 12, 2, 4, 25, 225, 66, 8, 98,

Buckets after 2nd pass

0	1	2	3	4	5	6	7	8	9
8 4 2	12	225 25				66			98

Merged list : 2, 4, 8, 12, 25, 225, 66, 98

Buckets after 3rd pass

0	1	2	3	4	5	6	7	8	9
98 66 25 12 8 4 2		225							

Merged list (sorted list) : 2, 4, 8, 12, 25, 66, 98, 225

Example 2.5.4 : Sort the following elements in descending order using radix sort.

7,103,15, 10, 3, 25, 28, 67, 304, 36, 49, 84 **S-16**

Solution : Buckets after 1st pass

Bucket no.	9	8	7	6	5	4	3	2	1	0
	49	28	67 7	36	25 15	84 304	3 103			10

Merged list = 49 28 7 67 36 15 25 304 84 103 3 10

Buckets after 2nd pass

Bucket no.	9	8	7	6	5	4	3	2	1	0
		084		067		084 049	036	025 028	010 015	003 103 304 007

Merged list = 084 067 049 084 036 028 025 015 010 007 304 103 003

Buckets after 3rd pass

Bucket no.	9	8	7	6	5	4	3	2	1	0
							304		103	003 007 010 015 025 028 036 049 067 084

Descending order list

Merged list = 304 103 84 67 49 36 28 25 15 10 7 3

Example 2.5.5 Sort following elements by Radix sort algorithm 87.3, 2.34, 7.29, 3.59, 45.8, 3.79, 3.20, 422. **W-16**

Solution :

Step 1 : **Multiplying each number by 100, we get**

8730, 234, 729, 359, 4530, 379, 320, 422

Sorting :

Bucket after pass 1

0	1	2	3	4	5	6	7	8	9
320 4580 8730		422		234					379 359 729

Merged list = 8730, 4580, 320, 422, 234, 729, 359, 379

Bucket after pass 2

0	1	2	3	4	5	6	7	8	9
		729 422 320	234 8730		359		379	4580	

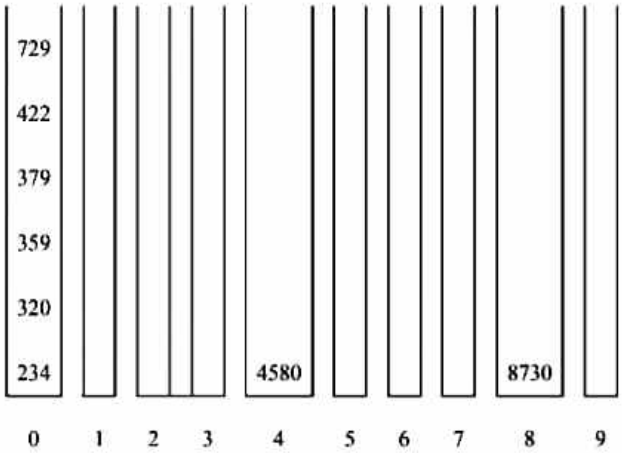
Merged list = 320, 422, 729, 8730, 234, 359, 379, 4580

Bucket after pass 3

0	1	2	3	4	5	6	7	8	9
		234	379 359 320	422	4580		8730 729		

Merged list = 234, 320, 359, 379, 422, 4580, 729, 8730.

Bucket after pass 4



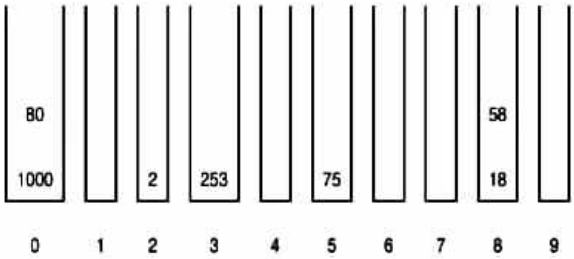
Merged list = 234, 320, 359, 379, 422, 729, 4580, 8730.

Dividing the list by 100, we get the sorted list : 2.34, 3.20, 3.59, 3.79, 4.22, 7.29, 45.80, 87.30.

Example 2.5.6 **Perform radix sort on the following list to arrange all array elements in ascending order : 18, 253, 1000, 2, 80, 75, 58** W-17

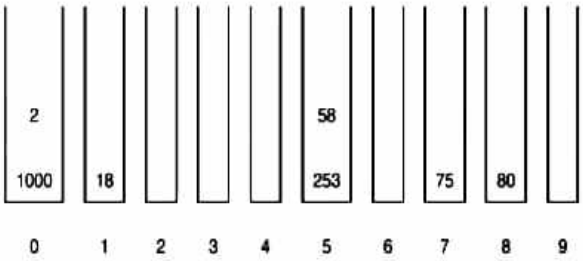
Solution :

Buckets after 1st pass



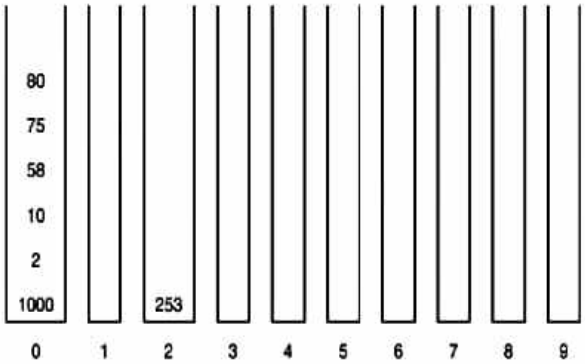
Merged list = 1000 80 2 253 75 18 58

Buckets after 2nd pass



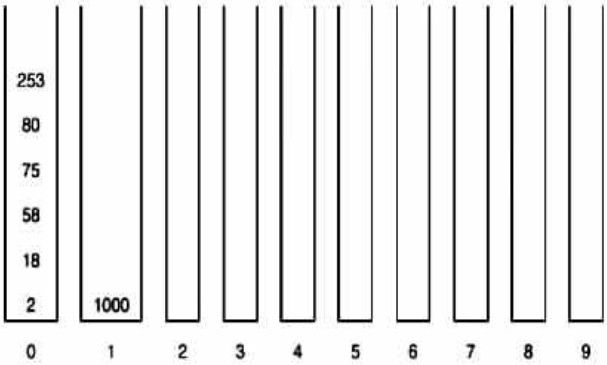
Merged list = 1000 2 18 253 58 75 80

Buckets after 3rd pass



Merged list = 1000 2 18 58 75 80 253

Buckets after 4th pass



Merged list = 2 18 58 75 80 253 1000

Syllabus Topic : Quick Sort

2.6 Quick Sort

➔ **(MSBTE : S-10, S-12, S-14, W-15)**

Q. 2.6.1 Write and explain the procedure for recursive quick sort. Justify the same with example. (Refer Section 2.6) S-10

Q. 2.6.2 Explain quick sort. (Refer Section 2.6) S-12

Q. 2.6.3 Write an algorithm for quick sort. (Refer Section 2.6) S-14

Q. 2.6.4 Explain working of quick sort with a suitable example. (Refer section 2.6) W-15

Q. 2.6.5 Describe quick sort. State its advantages and disadvantages. (Refer sections 2.6 and 2.6.4) W-15

Quick sort is the fastest internal sorting algorithm with the time complexity = O(n log n). The basic algorithm to sort an array a[ ] of n elements can be described recursively as follows :

1. If n < = 1, then return
2. Pick any element V in a[ ]. This is called the pivot.

   Rearrange elements of the array by moving all elements x_i > V right of V and all elements $x_i \le$ V left of V. If the place of the V after re-arrangement is j, all elements with value less than V, appear in a[0], a[1] ... a[j – 1] and all those with value greater than V appear in a[j + 1] ... a[n – 1].

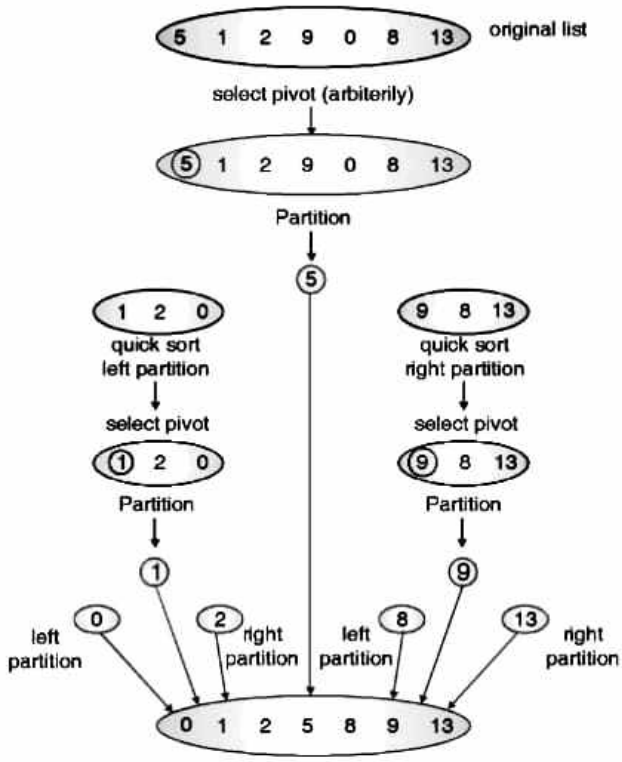


Fig. 2.6.1 : Illustration of quick sort

3. Apply quick sort recursively to a[0] ..... a[j – 1] and to a[j + 1] ....a[n – 1]

   Entire array will thus be sorted by as selecting an element V.

   (a) Partitioning the array around V.

   (b) Recursively, sorting the left partition.

   (c) Recursively sorting the right partition.

2.6.1 Picking a Pivot

We will choose the popular choice of the first element as the pivot. There are several strategies for pivoting. These strategies will be discussed later in the chapter.

Choose the first element as the pivot

2.6.2 Partitioning

Let the elements of the array a[*l*], a[*l* + 1] .... a[n – 1], a[n] are to be partitioned around the pivot V = a[*l*]. Initial value of *l* will be 0 and that of n will be n – 1. As recursion proceeds, these values will change. Index of the pivot (say K) will satisfy the following conditions :

$K \ge l$

$K \le n$

Two cursors (index variable) are initialized with

$i = l + 1$ and

$j = n - 1$

i moves towards right searching for an element greater than V. j moves towards left searching for an element smaller than V. When these elements are found they are interchanged. Again i moves towards right and j towards left and exchange is made whenever necessary. The process ends when i is equal to j or i > j.

j gives the index of the pivot element after termination of the above process.

Above procedure is explained with the help of the following example.

Let the array of number be

0	1	2	3	4	5	6	7	8	9	10	11	12
30	35	10	15	20	34	5	18	6	11	13	26	38

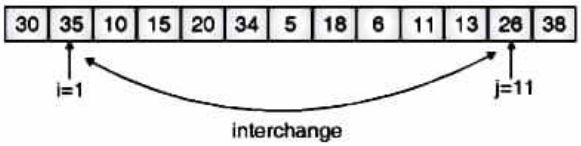
Initially *l* = 0, j = 12, V = a[*l*] {i.e. V = 30}

i is set to *l* + 1 and j is set to n – 1 as shown below

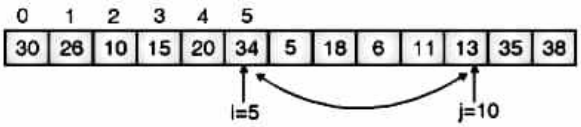
30	35	10	15	20	34	5	18	6	11	13	26	38
	↑											↑
	i = 1											j = 12

Now i moves right, while a[i] < 30. Hence i does not move further to right as a[1] > 30.

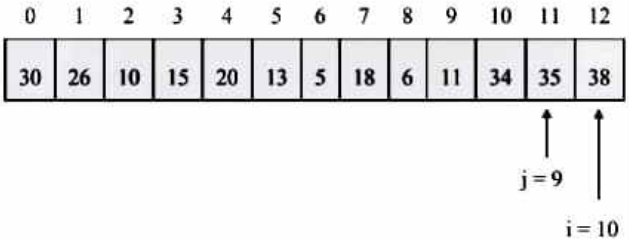
Then, j moves left, while a[j] > 30.



At this point a[i] and a[j] are interchanged and the movement of i and j resumes.

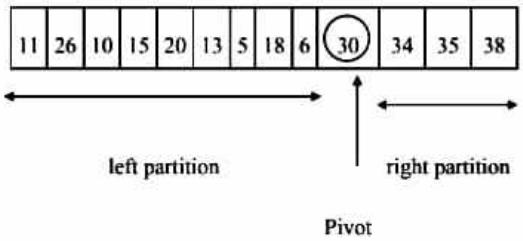


i moves right to a[5] as 34 > 30. j moves left to a[10] as 13 < 30. Once again a[i] and a[j] are swapped.



Now, i moves to a[10] as 34 > 30 and j moves to a[9] as 11 < 30.

Since, i > j, the process ends. j gives the location of the pivot element. Pivot element a[*l* with *l* = 0] is interchanged with a[i] to partition the array.



Please note that all element a[i] with l ≤ i < j are less than 30 and all elements a[j] with u ≥ i > j are greater than 30.

➢ **'C' function for the above partitioning algorithm.**

```
int partition(int a[],int l,int u)
{
   int v,i,j,temp;
   v=a[l];
   i=l;
   j=u+1;
   do
   {
      do
         i++;
      while(a[i]<v && i<=u);
         do
            j--;
         while(v<a[j]);
            if(i<j)
         {
            temp=a[i];
            a[i]=a[j];
            a[j]=temp;
         }
   }while(i<j);
   a[l]=a[j];
   a[j]=v;
   return(j);
}
```

➢ **'C' function for sorting an array of element using quick sort.**

```
void quick_sort(int a[],int l,int u)
{
   int j;
   if(l<u)
   {
      j=partition(a,l,u);
      quick_sort(a,l,j-1);
      quick_sort(a,j+1,u);
   }
}
```

➢ **Program 2.6.1 : Write a program in 'C' to implement quick sort.**

```
#include<conio.h>
#include<stdio.h>
void quick_sort(int[],int,int);
```

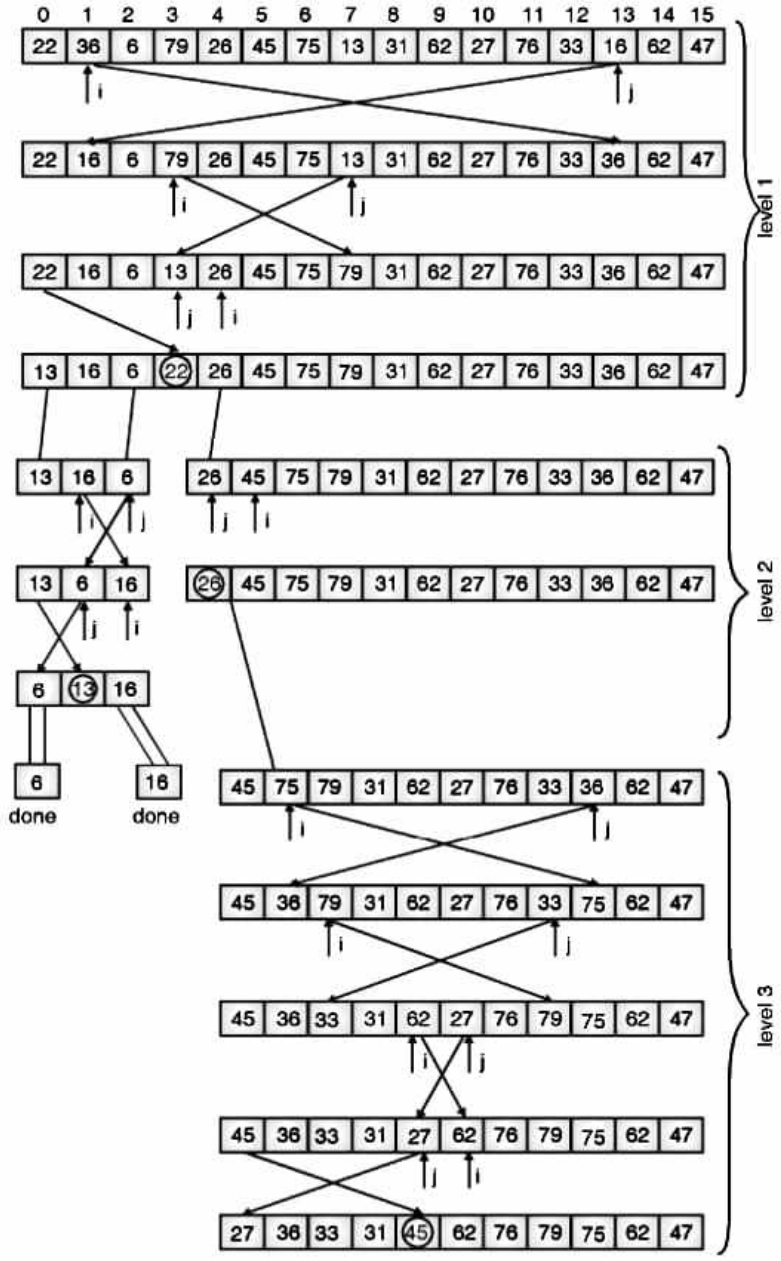
```
int partition(int[],int,int);
void main()
{
    int a[30],n,i;
    printf("\nEnter no of elements :");
    scanf("%d",&n);
    printf("\nEnter array elements :");
    for(i=0;i<n;i++)
        scanf("%d",&a[i]);
        quick_sort(a,0,n-1);
        printf("\nSorted array is :");
    for(i=0;i<n;i++)
        printf("%d",a[i]);
    getch();
}
void quick_sort(int a[],int l,int u)
{
    int j;
    if(l<u)
    {
        j=partition(a,l,u);
        quick_sort(a,l,j-1);
        quick_sort(a,j+1,u);
    }
}
int partition(int a[],int l,int u)
{
   int v,i,j,temp;
   v=a[l];
   i=l;
   j=u+1;
   do
   {
    do
        i++;
    while(a[i]<v && i<=u);

    do
        j--;
    while(v<a[j]);
    if(i<j)
    {
        temp=a[i];
        a[i]=a[j];
        a[j]=temp;
    }
   }while(i<j);
   a[l]=a[j];
   a[j]=v;
   return(j);
}
```

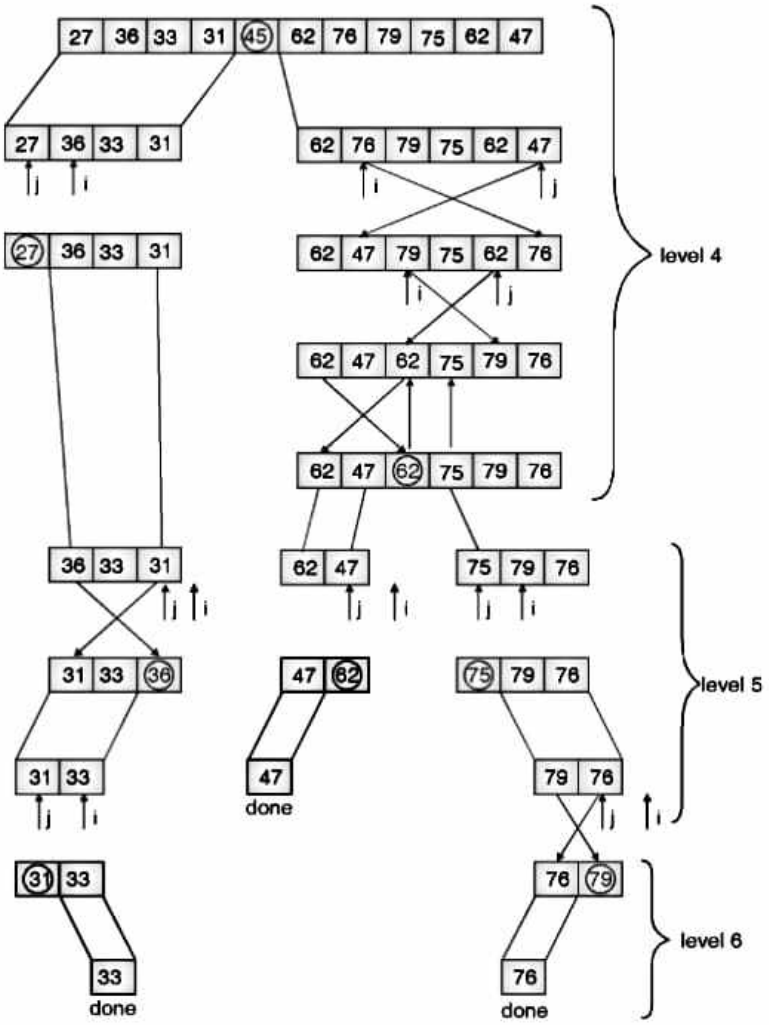
Output

```
Enter no of elements :  4
Enter array elements :  23   1    55   33
Sorted array is      :  1    23   33   55
```

Example 2.6.1 : Here are sixteen integers : 22, 36, 6, 7, 9, 26, 45, 75, 13, 31, 62, 27, 76, 33, 16, 62, 49. Sort them using quick sort.

Solution :





Note : Partition point at each level is encircled.

Example 2.6.2 : By using Quick sort, sort the following numbers in non-decreasing order. Show the contents by array after each pass. 45, 12, 5, 78, 19, 2, 56, 1, 62

Solution :

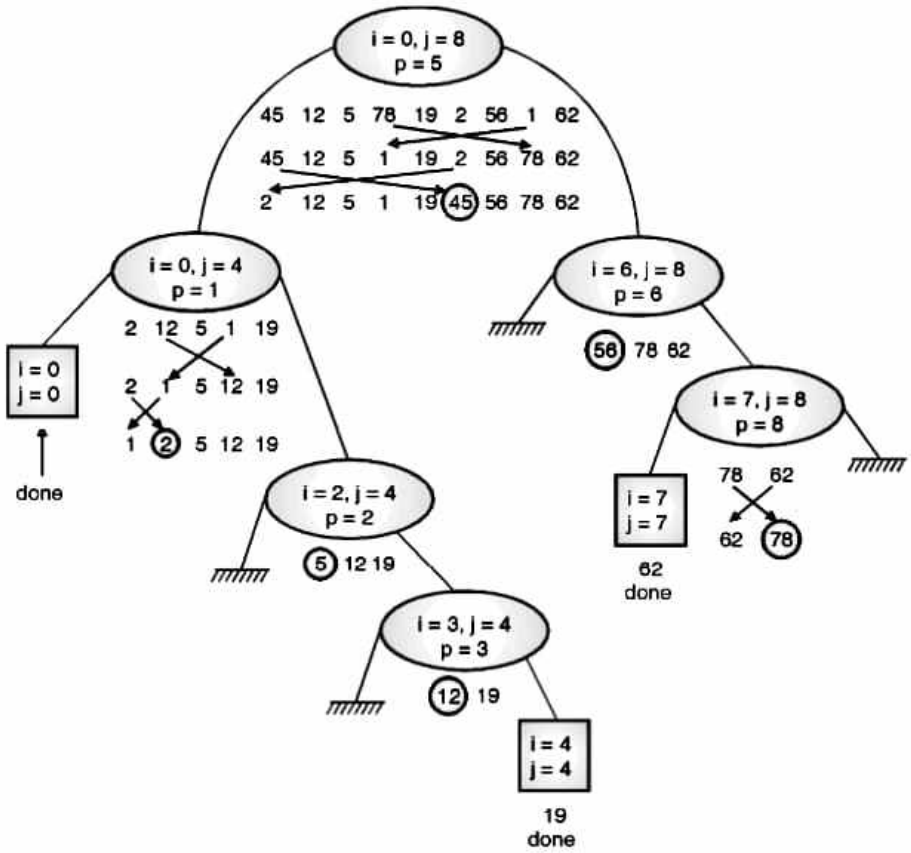


Fig. Ex. 2.6.2

Data after each pass

Pass	Data								
1	2	12	5	1	19	45	56	78	62
2	1	2	5	12	19	45	56	78	62
3	1	2	5	12	19	45	56	78	62
4	1	2	5	12	19	45	56	78	62
5	1	2	5	12	19	45	56	78	62
6	1	2	5	12	19	45	56	62	78

Example 2.6.3 : **Sort the following numbers in descending order using Quick sort. A = {3, 12, 5, 19, 1, 17}** S-12

Solution :

Sorting the list, A = {3, 12, 5, 19, 1, 17} in the descending order :

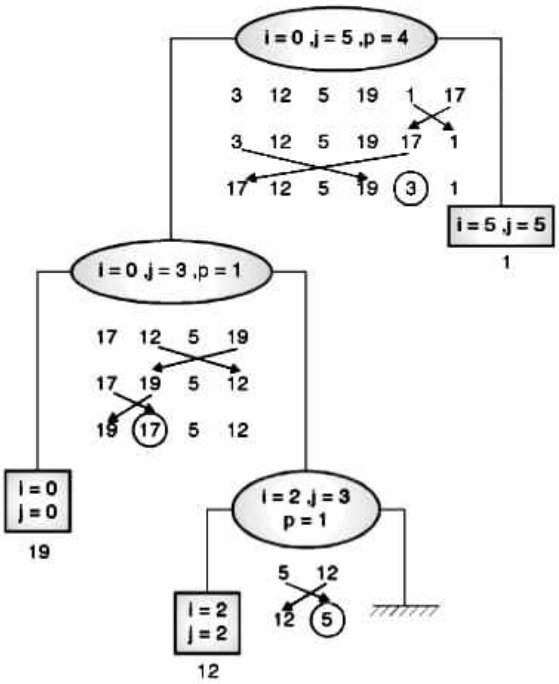


Fig. Ex. 2.6.3

Sorted list of data = 19 17 12 5 3 1

2.6.3 Running Time of Quick Sort

Quick sort takes O(n log n) time on the average to sort n elements and $O(n^2)$ time in worst case. Quick sort is a recursive algorithm. Its timing behaviour can be expressed using a recurrence formula. Solution of the recurrence formula will give its timing complexity. Since, the recursion terminates when n = 0 or 1.

We can take,

$$T(0) = 1 \qquad T(1) = 1$$

Running time of quick sort = running time of the left partition + running time of the right partition + time spent in partitioning the elements.

Since, partitioning requires scanning of the array linearly. It can be taken as cn, where c is a constant.

Running Time of the left partition = T(j), left partition contains j elements.

Running Time of the right partition = T(n – j – 1)

$$\therefore T(n) = T(j) + T(n - j - 1) + cn$$

2.6.3(A) Worst-case Analysis

In the worst case, either j = 0 or n – j – 1 is equal to 0. j will always be 0 if the data presented for sorting is already sorted in ascending order. n – j – 1 will always be 0 if the data presented for sorting is already sorted in descending order.

Let us take the case of j = 0

$$T(N) = T(0) + T(N-1) + CN \text{ for } N > 1$$

neglecting T(0) being insignificant

$$\begin{aligned} T(N) &= T(N-1) + CN \\ &= T(N-2) + C(N-1) + CN \\ &= T(N-3) + C(N-2) + C(N-1) + CN \\ &\vdots \\ &= T(1) + C \cdot 2 + C \cdot 3 + \ldots + C(N-2) + C(N-1) + CN \\ &= T(1) + C \cdot \sum_{i=2}^{N} i \\ &= O(N^2) \end{aligned}$$

2.6.3(B) Best-case Analysis

Running time will be minimum if the pivot is always found in the middle. Left partition and the right partition will have equal number of elements.

$$\begin{aligned} \therefore T(N) &= 2T(N/2) + CN \\ &= 2^2 T(N/2^2) + 2CN \\ &= 2^3 T(N/2^3) + 3CN \end{aligned}$$

Let us assume that $N/2^h = 1$

$$\therefore N = 2^h \text{ or } \log_2 N = h$$

$$\begin{aligned} \therefore T(N) &= 2^h T(N/2^h) + h \cdot C \cdot N \\ &= 2^h T(1) + C\,h\,N \\ &= N\,T(1) + C \cdot N \cdot \log_2 N \\ &= O(N \log_2 N) \end{aligned}$$

2.6.3(C) Average-case Analysis

Let us consider the recurrence relation,

$$T(N) = T(j) + T(N - j - 1) + CN$$

Since, the variable j could take any value from 0 to O(n – 1) with equal probability,

The recurrence relation for average case can be written as,

$$T(N) = \frac{1}{N}\left[\sum_{j=0}^{N-1} T(j) + \sum_{j=0}^{N-1} T(N-j-1)\right] + CN$$

As j varies from 0 to N – 1, N – j – 1 varies from N – 1 to 0

$$\text{Hence } \sum_{j=0}^{N-1} T(N-j-1) = \sum_{j=0}^{N-1} T(j)$$

$$\therefore T(N) = \frac{2}{N}\sum_{j=0}^{N-1} T(j) + CN \qquad \ldots(2.6.1)$$

$$\text{Or} \quad NT(N) = 2\sum_{j=0}^{N-1} T(j) + CN^2 \qquad \ldots(2.6.2)$$

With N written as N – 1, Equation (2.8.2) becomes

$$(N-1)\,T(N-1) = 2\sum_{j=0}^{N-2} T(j) + C(N-1)^2 \qquad \ldots(2.6.3)$$

If we subtract Equation(2.8.3) from Equation(2.8.2), we get

$$NT(N) - (N-1)\,T(N-1) = 2T(N-1) + 2CN - C$$

Dropping C being insignificant and arranging terms.

$$NT(N) = N + 1)\,T(N-1) + 2CN$$

Dividing throughout by N(N + 1)

$$\text{We get,} \quad \frac{T(N)}{N+1} = \frac{T(N-1)}{N} + \frac{2C}{N+1}$$

$$= \frac{T(N-2)}{N-1} + \frac{2C}{N} + \frac{2C}{N+1}$$

$$= \frac{T(N-3)}{N-2} + \frac{2C}{N-1} + \frac{2C}{N} + \frac{2C}{N+1}$$

$$\vdots$$

$$= \frac{T(1)}{2} + \frac{2C}{3} + \ldots \frac{2C}{N} + \frac{2C}{N+1}$$

$$= \frac{T(1)}{2} + 2C\sum_{i=3}^{N+1} \frac{1}{i}$$

$$2C\sum_{i=3}^{N+1} \frac{1}{i} \approx \int \frac{1}{N} = \log N$$

$$\therefore \frac{T(N)}{N+1} = \frac{T(1)}{2} + \log N$$

$$\therefore T(N) = \frac{T(1)}{2}(N+1) + (N+1)\log N$$

$$= O(N \log N)$$

Quick sort gives its worst behavior when data presented for sorting is already sorted in either ascending or descending order.

2.6.4 Advantages and Disadvantages of Quick Sort

➔ **(MSBTE : W-15)**

☞ **Advantages of quick sort**

1. It is the fastest internal sorting algorithm with time complexity = O(n log n).
2. It is an in-place algorithm and does not require additional memory to maintain another copy of data.

☞ **Disadvantages**

1. Sorting method is not stable.
2. It cannot be used for external sorting.
3. Quick sort algorithm degenerates if the data being sorted is already sorted either in ascending or descending order.

2.7 Comparison of Sorting Algorithms

➔ **(MSBTE : W-11, W-14)**

Q. 2.7.1 **Efficiency of sorting algorithms.**
(Refer Section 2.7) **W-11**

Q. 2.7.2 **Differentiate between the radix sort and shell sort methods. (Refer Section 2.7)** **W-14**

Define the following terms with respective sorting.

(i) sort stability (ii) efficiency (iii) passes

Sorting algorithm	Efficiency	Passes	Sort stability
Bubble sort	$O(n^2)$	n – 1	stable
Selection sort	$O(n^2)$	n – 1	stable
Insertion sort Best case Worst case	 O(n) $O(n^2)$	 n – 1 n – 1	stable

Sorting algorithm	Efficiency	Passes	Sort stability
Quick sort Best case Worst case	 O(n log n) $O(n^2)$	 log n n – 1	unstable
Merge sort	O(n log n)	log n	stable
Shell sort Best case Worst case	 O(n) $O(n^2)$	 log n log n	unstable
Radix sort	O(n)	No. of digits in the largest number	stable

Sorting algorithm	Best-case	Worst-case	Average-case
Bubble sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion sort	O(n)	$O(n^2)$	$O(n^2)$
Quick sort	O(n log n)	$O(n^2)$	O(n log n)
Merge sort	O(n log n)	O(n log n)	O(n log n)
Shell sort	O(n)	$O(n^2)$	$O(n^2)$
Radix sort	O(n)	O(n)	O(n)

If the sorting algorithm is not data sensitive, its best case, worst case and average case timing behaviour will be same.

Bubble sort, selection sort, merge sort and radix sort algorithms are not data sensitive. Quick sort and shell algorithms are data sensitive.

2.7.1 Best-case, Worst-case and Average-case Analysis of Sorting Algorithm

➔ **(MSBTE : W-17)**

Q. 2.7.3 Give complexity of following methods. **W-17**

(i) Bubble sort (Refer section 2.7.1[$O(n^2)$])

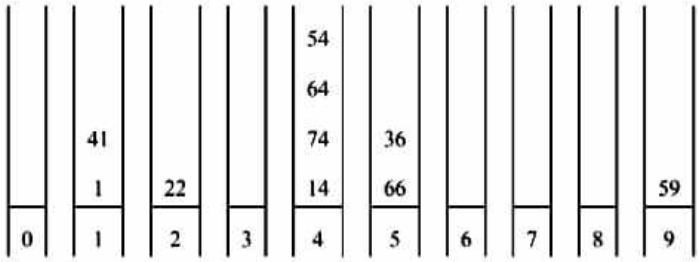
(ii) Radix sort (Refer section 2.7.1[O(n)])

Example 2.7.1 : Sort the following numbers in ascending order using radix sort

14, 1, 66, 74, 22, 36, 41, 59, 64, 54

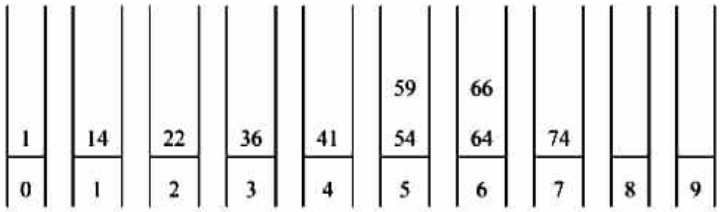
Solution :

Buckets after 1st pass



merged list = 1 41 22 14 74 64 54 66 36 59

Buckets after 2nd pass



merged list = 1 14 22 36 41 54 59 64 66 74

2.7.2 Compare Quick Sort and Radix Sort

➔ (MSBTE : W-09, S-13)

Q. 2.7.4 Compare quick sort and radix sort with respect to working principle and time complexity.
(Refer Section 2.7.2) W-09, S-13

- Radix sort is the fastest internal sorting algorithm with time complexity = O(d.n), where

  d = number of digits in the largest number in the array.

  n = number of data elements to be sorted.

- The radix sort uses queue data structure. The working principle of radix sort is summarized below.

  1. Number of passes required to sort using radix sort is equal to number of digits in the largest number in the list.
  2. In the first pass, numbers are distributed among 10 queue (queue no. 0 to 9) on the basis of least significant digit.

     In the second pass on the basis on second digit from the right and so on.
  3. At the end of every pass, number in buckets are merged to produce a common list.

- Radix sort is widely used for sorting of strings.
- Radix sort algorithm is stable.
- There is an additional overhead of maintaining the queue data structure.

The working principle of quick sort is summarized below :

1. select an element V as a partition element.
2. Partition the array around V such that every element in the left partition ≤ V and every element in the right partition ≥ V.
3. Left partition is subjected to quick sort recursively.
4. Right partition is subjected to quick sort recursively.

- Quick sort is an in-place internal sorting algorithm without any additional memory for storing data.
- Quick sort algorithm is not stable.
- Quick sort algorithm degenerates if the data being sorted is already sorted, either in ascending or descending order.

Syllabus Topic : Searching an Item in a Data set using Methods

2.8 Searching

➔ (MSBTE : W-09, S-10, W-11, W-12, S-13, W-13, W-15, W-16)

Q. 2.8.1 What is searching ? Enlist different methods.
(Refer Section 2.8) W-09, W-12, W-13

Q. 2.8.2 What is searching ?
(Refer Section 2.8) S-10, S-13

Q. 2.8.3 Briefly explain searching.
(Refer Section 2.8) W-11

Q. 2.8.4 Define searching. State two methods of searching.
(Refer sections 2.8, 2.8.1 and 2.8.3) W-15

Q. 2.8.5 Define searching. Give its type.
(Refer section 2.8) W-16

Searching is a technique of finding an element in a given list of elements. List of elements could be represented using an :

(a) Array (b) Linked List

(c) A binary tree (d) A B-tree

(e) Heap

Elements could also be stored in file. Searching technique should be able to locate the element to be searched as quickly as possible. Many a time, it is necessary to search a list of records to identify a particular record. Usually, each record is uniquely identified by its key field and searching is carried out on the basis of key field. If search results in locating the desired record then the search is said to be successful. Otherwise, the search operation is said to be unsuccessful. The complexity of any searching algorithm depends on number of comparisons required to find the element. Performance of searching algorithm can be found by counting the number of comparisons in order to find the given element.

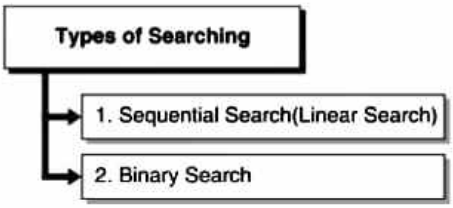


Fig. 2.8.1 : Types of searching

Syllabus Topic : Linear Search

2.8.1 Sequential Search (Linear Search)

➔ **(MSBTE : S-09, S-10, W-12, S-13, S-15, W-15, W-17)**

Q. 2.8.6 Explain linear search with example. Explain the linear search algorithm. Also give its limitations. (Refer Section 2.8.1) **S-09 , S-10**

Q. 2.8.7 Explain linear search with suitable example. (Refer Section 2.8.1) **W-12, S-13**

Q. 2.8.8 Explain the linear search algorithm. Also give its limitations. (Refer section 2.8.1) **S-15**

Q. 2.8.9 Give complexity of following methods. **W-17**

(i) Linear search (Refer section 2.8.1[O(n)])

(ii) Binary search (Refer section 2.8.3 [O(log n)])

In sequential search elements are examined sequentially starting from the first element. The process of searching terminates when the list is exhausted or a comparison results in success.

Algorithm for searching an element 'key' in an array 'a[ ]' having n elements :

The search algorithm starts comparison between the first element of a[ ] and "key". As long as a comparison does not result in success, the algorithm proceeds to compare the next element of "a[ ] with "key". The process terminates when the list is exhausted or the element is found.

➢ **'C' function for sequential search.**

```
int sequential (int a[ ] , int key, int n)
{
    int i = 0 ;
    while (i < n)
    {
        if (a[i] == key)
            return (i) ;
        i++ ;
    }
    return (– 1) ;
}
```

The function returns the index of the element in 'a[ ]' for successful search. A value –1 is returned if the element is not found.

☞ **Analysis of sequential search algorithm**

Number of comparisons required for a successful search is not fixed. It depends on the location (place) being occupied by the key element. An element at i^{th} location can be searched after i-comparisons. Number of comparisons required is probabilistic in nature. We could best calculate the average number of comparisons required for searching an element.

Let P_i is the probability that the element to be searched will be found at i^{th} position. i number of comparisons will be required to search the ith element.

∴ Expected number of comparisons for a successful search

$$C = 1.P_1 + 2.P_2 + \ldots + nPn$$

Since the element could be found at any location with equal probability.

$$P_1 = P_2 \ldots = P_n = 1/n$$

$$\therefore \quad C = \frac{1}{n} + \frac{2}{n} + \ldots + \frac{n}{n} = \frac{1}{n}(1 + 2 + , + n) = \frac{n(n+1)}{2n} = \frac{n+1}{2}$$

Time complexity of sequential search = O(n).

☞ **Limitations of sequential search**

1. Its time complexity is of the order of O(n). It should not be used for a long list.

➢ **Program 2.8.1 : A sample program for sequential search.**

Q. 2.8.10 Write a program to find the position of the element 40 using Linear Search Method in the array given below. A = {08, 90, 14, 32, 40, 54} (Refer Program 2.8.1) **S-11**

Q. 2.8.11 WAP to implement linear search for 10 elements in an array. (Refer Program 2.8.1) **S-14**

Q. 2.8.12 Write a program to search an element in an array. Display position of element. (Refer Program 2.8.1) **W-16**

Q. 2.8.13 Write a program for linear search. Find position of element 30 using linear search algorithm in given sequence.

10 5 20 25 8 30 40

(Refer Program 2.8.1) **S-17**

```
#include<stdio.h>
#include<conio.h>
int sequential_search(int a[],int key,int n);
void main()
{
    int a[50],n,key,i;
    printf("\nNo.of elements :");
    scanf("%d",&n);
    printf("\nEnter %d numbers :",n);
    for(i=0;i<n;i++)
        scanf("%d",&a[i]);
    printf("element to be searched :");
    scanf("%d",&key);
    i=sequential_search(a,key,n);
    if(i==-1)
        printf("\n Not found");
    else
        printf("\nFound at location = %d",i+1);
    getch();
}
int sequential_search(int a[],int key,int n)
{
    int i;
    i=0;
    while(i<n && key != a[i])
        i++;
    /* search terminates here */

    if(i<n)
        return(i);
    return(-1);
}
```

Output

```
No. of elements : 4
Enter 4 numbers : 66 77 88 99
element to be searched : 88
Found at location = 3
```

2.8.2 Sequential Search on a Sorted Array

Expected number of comparisons required for unsuccessful search can be reduced if the array is sorted.

Array a[ ] 1 5 9 14 23 25

Element to be searched = 10 — Search should terminate here

Assuming elements are sorted in ascending order, the search should terminate as soon as an element with a value equal to or greater than "key" (element to be searched) is found. After termination of search, if the value of the element is equal to "key" then search is successful. If the value of the element is greater than "key" then search ends unsuccessfully.

➢ **'C' function for sequential search in a sorted array**

```
int search_seq_sorted (int a[ ], int key, int n)
{
    int i ;
    i = 0 ;
    while (i<n && key > a[i])
        i + + ;
/* search terminates here */
    if (i < n && key == a[i])
        return (i) ;
    return (–1) ;
}
```

Syllabus Topic : Binary Search

2.8.3 Binary Search

➔ **(MSBTE : S-08, W-11, W-12, S-14, W-14, W-15, W-16, W-17)**

Q. 2.8.14 Explain binary search algorithm and also give its advantages. (Refer Section 2.8.3) **S-08**

Q. 2.8.15 Describe binary search. (Refer Section 2.8.3) **W-11**

Q. 2.8.16 Explain binary search with suitable example. (Refer Section 2.8.3) **W-12, W-14**

Q. 2.8.17 Write an algorithm to implement binary search. (Refer Section 2.8.3) **S-14**

Q. 2.8.18 Define searching. State two methods of searching. *(Refer sections 2.8, 2.8.1 and 2.8.3)* **W-15**

Q. 2.8.19 Describe binary search algorithm. Give example to search an element using binary search algorithm. (Refer section 2.8.3) **W-16**

Q. 2.8.20 Describe working of binary search method. (Refer section 2.8.3) **W-17**

Linear search has a time complexity O(n). such algorithms are not suitable for searching when number of elements is large. Binary search exhibits much better timing behaviour in case of large volume of data with timing complexity

$O(\log_2 n)$

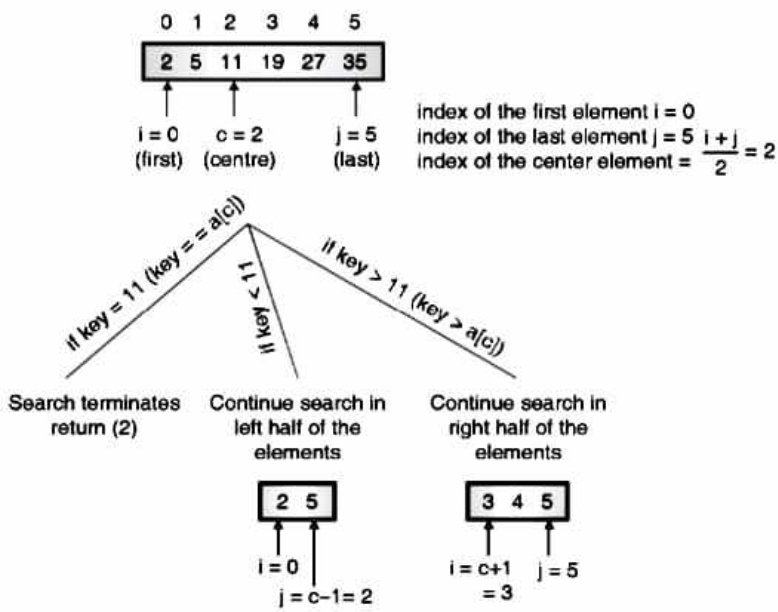
Number of comparisons for $n = 2^{20}$ (1 million)

Sequential search (in worst case) = 2^{20} comparisons.

Binary search (in worst case) = $\log_2 2^{20} = 20$ comparisons.

Linear search (sequential search) may need 1 million comparisons for searching an element in an array having 1 million elements. Binary search will require, just 20 comparisons for the same task. Binary search uses a much better method of searching. Binary search is applicable only when the given array is sorted.

- This method makes a comparison between the "key" (element to be searched) and the middle element of the array.
- Since elements are sorted, comparisons may result in either a match or comparison could be continued with either left half of elements or right half of the elements.



- Left half of elements could be selected by simply making $j = c-1$
- Right half of element could be selected by simply making $i = c+1$
- Process of selecting either the left half or the right half continues until the element is found or element is not there.

Example 1 : [Searching the element 29 in the given array]

Index	0	1	2	3	4	5	6	7	8
Elements	5	9	11	15	25	29	30	35	40

$i = 0$ $\quad c = \frac{i+j}{2} = \frac{0+8}{2} = 4$ $\quad j = 8$

Element to be searched, key = 29

Step 1 : Since key > a[c] (29>25) right half is selected.

index	5	6	7	8
elements	29	30	35	40
	i = 5	c = 6		j = 8

Step 2 : Since key < a[c] (29 < 30) left half is selected.

Index elements **5** **element is found and search is successful.**

29

i = 5 c = 5 j = 5

Example 2 : Searching the element 6 in the given array.

Index	0	1	2	3	4	5	6	7	8
Elements	5	9	11	15	25	29	30	35	40

Initially	i =0	j = 8	c = 4	[Since key <a[c]; 6 < 25 j = c−1]
Next iteration	i = 0	j = 3	c= 1	[Since key <a[c]; 6 < 9 j = c−1]
Next iteration	i = 0	j = 0	c = 0	[Since key <a[c]; 6 < 5 j = c−1]
Next iteration	i = 1	j = 0	c = 0	Element not found.

i > j indicates, element is not found

➢ **'C' Function for binary search (Non-recrusive)**

```
int bin_search(int a[],int i, int j, int key)
{
    int c;
    c=(i+j)/2;
    while(a[c]!=key && i<=j)
    {
    /* search as long as element is not
        found and it could be there*/
    if(key>a[c])
        i=c+1; // select right half
    else
        j=c-1; //select left half
    c=(i+j)/2;
    }
    if(i<=j)
        return(c);
    return(-1);
}
```

☞ **Analysis of binary search algorithm worst case behaviour (Unsuccessful search)**

Let us assume that the array contains n elements.

∴ The maximum number of elements after 1 comparison = n/2

The maximum number of elements after 2 comparisons $= n/2^2$

The maximum number of elements after h comparisons $= \frac{n}{2^h}$

For the lowest value of h[maximum number of elements left = 1]

$$\therefore \frac{n}{2^h} = 1 \text{ or } 2^h = n$$

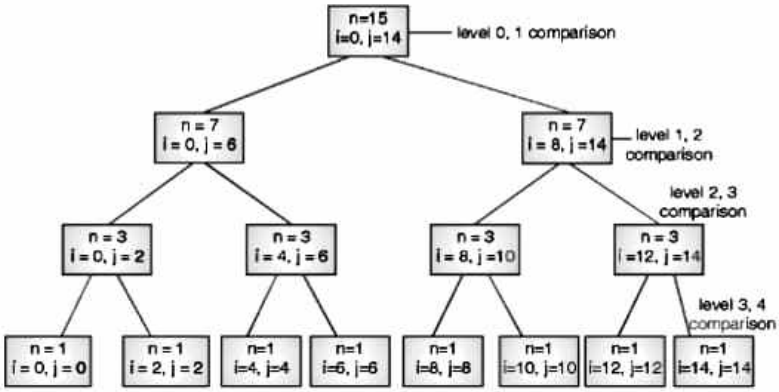
or $$h = \log_2 n = O(\text{Log}_2 n)$$

Thus, the worst case behaviors of binary search algorithm has a timing complex = $O(\text{Log}_2 n)$

A searching algorithm requires maximum number of comparisons in unsuccessful search.

☞ **Average case behaviour [Successful search]**

Let us try to understand the average case behaviour with the help of an array having 15 elements.



- Binary search starts with 15 elements and if the element is found at the centre, search terminates with a success with 1 comparison i.e. success at level 0.
- At level 1, search starts with either left half of the elements or the right half of the elements. Probability of successful search at level 1 is twice that of a successful search at level 0. Level zero has one event (one node) and level 1 has 2 events (2 nodes) of success.
- Relation between last level (h = 3) and number of elements (n = 15) is given by $2^{3+1} = 15+1$

Or $2^{h+1} = n+1$

Or $h+1 = \log_2 (n+1)$

Or $h = [\log_2 (n+1)] - 1$

∴ Total number of comparison to search every element of array is given by :

$$C = 1.2^0+2.2^1+\ldots+(h+1)2^h \quad \ldots(2.8.1)$$

$$2C = 1.2^1+2.2^2+\ldots+h2^h+(h+1)\,2^{h+1} \quad \ldots(2.8.2)$$

Subtracting Equation(2.10.2) from Equation(2.10.1)

$$-C = 1.2^0+2^1+2^2+\ldots 2^h-(h+1)2^{h+1}$$

Or $$C = (h+1)\,2^{h+1}-[2^0+2^1+2^2+\ldots+2^h]$$

$$= (h+1)2^{h+1}-(-1+2^{h+1})$$

$$= h.\,2^{h+1}+1$$

Therefore, average number of comparisons

$$= \frac{h.\,2^{h+1}+1}{n} = \frac{h.\,2^{h+1}+1}{2^{h+1}-1}$$

$$\approx h \text{ for large } n$$

$$= O \log_2 (n)$$

Thus, expected number of comparisons for successful as well as unsuccessful search O ($\log_2$ n)

➢ **Recursive function for binary search.**

```
int bin_search(int a[],int i,int j,int key)
{
    int c;
    if(i<=j)
    {
        c=(i+j)/2;
        if(key==a[c])
            return(c);
        / * Change it to <
        if the numbers are in descending order */
        if(key>a[c])
            return(bin_search(a,c+1,j,key));
        return(bin_search(a,i,c-1,key));
    }
    return(-1);
}
```

➢ **Program 2.8.2 : Program to implement binary search algorithm.**

Q. 2.8.21 Write 'C/C++' program for binary search. (Refer Program 2.8.2) W-08, S-12, W-13

Q. 2.8.22 Write a 'C' program for recursive binary search using function. (Refer Program 2.8.2) **S-13**

Q. 2.8.23 Write a 'C' program for binary search. (Refer Program 2.8.2) **S-15**

```
#include<stdio.h>
#include<conio.h>
void bubble_sort(int [],int);
int bin_search(int [],int,int);
void main()
{
    int a[30],n,i,key,result;
    printf("\nEnter number of elements :");
    scanf("%d",&n);
    printf("\nEnter the array elements :");
    for(i=0;i<n;i++)
        scanf("%d",&a[i]);
    printf("\nEnter the element to be searched :");
    scanf("%d",&key);
    bubble_sort(a,n);
    result=bin_search(a,key,n);
    if(result==-1)
        printf("\nElement not found :");
    else
        printf("\nElement  is found at location %d",result+1);
    getch();
}
void bubble_sort(int a[],int n)
{
    int i,j,temp;
    for(i=1;i<n;i++)
        for(j=0;j<n-i;j++)
            if(a[j]>a[j+1])
            {
                temp=a[j];
                a[j]=a[j+1];
                a[j+1]=temp;
            }
}
int bin_search(int a[],int key,int n)
{
    int i,j,c;
    i=0;
    j=n-1;
    c=(i+j)/2;
    while(a[c]!=key && i<=j)
    {
        if(key>a[c])
            i=c+1;
        else
            j=c-1;
        c=(i+j)/2;
    }
    if(i<=j)
        return(c);
    return(-1);
}
```

Output

```
Enter number of elements : 6
Enter the array elements :12 34 54 23 11 90
Enter the element to be searched : 11
Element  is found at location 1
```

Example 2.8.1 : Apply binary search on the following numbers stored in array from A [0] to A [10] 9, 17, 23, 38, 45, 50, 57, 76, 79, 90, 100 to search numbers – 10 to 100.

Solution :

Searching 10

0	1	2	3	4	5	6	7	8	9	10					
9	17	23	38	45	50	57	76	79	90	100			i	j	k
↑					↑					↑			0	10	5
i					k					j					

Step 1 : Since 10 < A [5], j = K – 1 = 4

0	1	2	3	4											i	j	k
9	17	23	38	45											0	4	2
↑		↑		↑													
i		k		J													

Step 2 : Since 10 < A [2], j = K – 1 = 1

0	1									
9	17							i	j	k
↑↑	↑							0	1	0
i k	j									

Step 3 : Since 10 > A [0], j = K + 1 = 1

				i	j	k
1				1	1	1
17						
↑ ↑ ↑						
i j k						

Step 4 : Since 10 < A [1], j = K – 1 = 0

As i becomes less than j, element 10 is not in the array A [ ]

Searching 100 :

0	1	2	3	4	5	6	7	8	9	10				
9	17	23	38	45	50	57	76	79	90	100		i	j	K
↑					↑					↑		0	10	5
i					k					j				

Step 1 : Since 100 > A [5], i = K + 1 = 6

6	7	8	9	10										
57	76	79	90	100								i	j	K
↑		↑		↑								6	10	8
i		k		j										

Step 2 : Since 100 > A [8], i = K + 1 = 9

9	10													
90	100											i	j	K
↑↑	↑											9	10	9
i k	j													

Step 3 : Since 100 > A [9], i = K + 1 = 10

10														
100												i	j	K
↑ ↑ ↑												10	10	10
i j k														

Since, the element to be searched is found at A [10], search terminates with a success.

Example 2.8.2 : **Find the position of element 29 using binary search method in Array 'A' given below :A = {11, 5, 21, 3, 29, 17, 2, 43}** S-12, S-15

Solution : Data should be sorted for binary search.

Sorted list for 'A' = {2, 3, 5, 11, 17, 21, 29, 43}

Finding the position of element 29 :

Sr. No.	List	i	j	k(centre)
1.	2, 3, 5, 11, 17, 21, 29, 43	0	7	(0 + 7)/2 = 3
2.	17, 21, 29, 43	4	7	(4 + 7)/2 = 5
3.	$\boxed{29}$, 43	6	7	(6 + 7)/2 = 6

Position (index) of 29 in the array = 6

Example 2.8.3 : Find the position of element 88 using binary search method in an array given below:
A = { 77, 33, 44, 11, 88, 22, 66, 55 } W-13

Solution :

Data should be sorted for binary search.

Sorted list for 'A' = {11, 22, 33, 44, 55, 66, 77, 88}

Finding the position of element 88 :

Sr. no.	List	i	j	k(centre)
1.	A, B, C, D, E, F, G, H, I, J ↑ 4 The list, right of E (Position = 4) is taken for the next iteration	0	9	$\frac{(0+9)}{2} = 4$
2.	F, G, [H], I, J ↑ ↑ 5 9	5	9	$\frac{(5+9)}{2} = 7$

Position (index) of 'H' in the array = 7.

Computational Complexity

Example 2.8.4 Define computational complexity. Give complexity for the following :

i) Binary search ii) Bubble sort

iii) Quick sort iv) Linear search S-16

Solution :

Computational complexity

It is a method of specifying the computing time of an algorithm might require for its execution. It normally depends on the size of input and the algorithm when considering time complexities $f_1(n)$ and $f_2(n)$ of two different algorithms for a given problem of size n, we need to consider and compare behavior of two functions for large n.

The time complexity is often represented using big 'O' notation.

Complexity

Binary search	–	O(log n)
Bubble sort	–	$O(n^2)$
Quick sort	best case worst case	O(n logn) $O(n^2)$
Linear search	–	O(n)

Example 2.8.5 Find location of element 'H' by using binary search algorithm in the given list.
B F D A C E I G H J S-16

Solution :

Data should be sorted for binary search

Sorted list = { A, B, C, D, E, F, G, H, I, J }

Finding position of elements 'H'

Example 2.8.6 Given stepwise procedure to search 65 in the following list :

List : 23, 12, 5, 29, 10, 65, 55, 70. W-17

Solution :

Step 1 : Data is arranged in the ascending order

5 10 12 23 29 55 65 70

Step 2 : Binary search for x = 65

	Data								i	j	k(cartre)
Sr. No.	0	1	2	3	4	5	6	7			
1	5	10	12	23	29	55	65	75	0	7	3
				↑							
2					29	55	65	75	4	7	5
3							65	75	6	7	6
							↑				

The element 65 is found at the location 6.

2.8.4 Difference between Linear Search and Binary Search

➔ (MSBTE : W-15)

Q. 2.8.24 Distinguish between linear search and binary search.(any 4 points)(Refer section 2.8.4) W-15

Sr. No.	Linear search	Binary search
1.	Data need not be sorted.	Data should be sorted.
2.	Time complexity O(n).	Time complexity O(log n).

Sr. No.	Linear search	Binary search
3.	Not suitable when number of elements is large.	Suitable when number of elements is large.
4.	A very simple algorithm.	Algorithm is difficult to understand.

2.9 MSBTE Questions and Answers

➔ Summer 2008 – Total Marks 24

Q. 1 Write a program in 'C' language for selection sort and arrange the given numbers in ascending order using selection sort.

Numbers : 16, 23, 13, 9,7,5

(Program 2.3.1, Example 2.3.2) **(8 Marks)**

Q. 2 Explain radix sort with example.
(Section 2.5) **(4 Marks)**

Q. 3 Write a program for insertion sort and shell sort.
(Program 2.4.1 and Program 2.7.1) **(4 Marks)**

Q. 4 Explain binary search algorithm and also give it's advantages.**(Section 2.8.3)** **(4 Marks)**

➔ Winter 2008 – Total Marks 26

Q. 5 Write 'C/C++' program for binary search.
(Program 2.8.2) **(3 Marks)**

Q. 6 Sort the given numbers in ascending order using radix sort Numbers : 348, 14, 614, 5381, 47.
(Example 2.5.1) **(3 Marks)**

Q. 7 Write the algorithm for Insertion sort and arrange the given numbers in ascending order using insertion sort.

Numbers : 77, 33, 44, 11, 88, 22, 66, 55

(Section 2.4, Example 2.4.1) **(8 Marks)**

Q. 8 Describe the principle of selection sort with one example. **(Section 2.3)** **(4 Marks)**

Q. 9 Write a program for bubble sort in 'C/C++' language. **(Program 2.2.1)** **(4 Marks)**

➔ Summer 2009 – Total Marks 20

Q. 10 Explain the linear search algorithm. Also give its limitations. **(Section 2.8.1)** **(4 Marks)**

Q. 11 Write an algorithm for insertion sort and arrange the given numbers in ascending order using insertion sort numbers 45, 22, 8, 34, 19.
(Section 2.4 and Example 2.4.2) **(8 Marks)**

Q. 12 Arrange the given in ascending order using radix sort. 361, 12, 527, 143, 9, 768, 348.
(Example 2.5.1) **(4 Marks)**

➔ Winter 2009 – Total Marks 22

Q. 13 What is searching and sorting ? Enlist different methods. **(Sections 2.1 and 2.8)** **(4 Marks)**

Q. 14 Describe bubble sort with the help of example. Take minimum four values. **(Section 2.2)** **(6 Marks)**

Q. 15 Compare quick sort and radix sort with respect to working principle and time complexity.
(Section 2.7.2) **(4 Marks)**

➔ Summer 2010 – Total Marks 22

Q. 16 What is searching ? Explain linear search with example.
(Sections 2.8 and 2.8.1) **(4 Marks)**

Q. 17 Write and explain the procedure for recursive quick sort. Justify the same with example.
(Sections 2.6 and 2.6.1) **(6 Marks)**

Q. 18 Write a program in 'C' language for selection sort and arrange the given numbers in ascending order using selection sort number : 16, 23, 13, 9, 7, 5.
(Example 2.3.2, Program 2.3.1) **(8 Marks)**

Q. 19 Write a program to sort an array of ten elements with bubble sort.
(Program 2.2.1) **(4 Marks)**

➔ Winter 2010 – Total Marks 20

Q. 20 Write an algorithm for bubble sort.
(Section 2.2) **(4 Marks)**

Q. 21 Consider the following array

DATA : 53 64 23 66 98 12 76 44 33 10

Write an algorithm to find an item 22 from above array using linear search. Write its complexity.
(Section 2.8.1 and Similar to Program 2.8.1)
(8 Marks)

Q. 22 Describe radix sort algorithm along with an example and state its complexity.
(Sections 2.5 and 2.5.3) **(4 Marks)**

➔ Summer 2011 – Total Marks 12

Q. 23 Define sorting? Enlist and explain the types of sorting with example. **(Section 2.1)** **(4 Marks)**

Q. 24 Write a program to find the position of the element 40 using linear search method in the array given below.

A = {08, 90, 14, 32, 40, 54} **(Program 2.8.1)**

(8 Marks)

➔ Winter 2011 – Total Marks 30

Q. 25 Explain the concept of sorting algorithms and efficiency of sorting algorithms.
(Sections 2.1 to 2.1.3 and 2.7) **(10 Marks)**

Q. 26 Explain the following sorting techniques : **(10 Marks)**

(i) Bubble sort **(Section 2.2)**

(ii) Selection sort **(Section 2.3)**

Q. 27 Briefly explain searching and describe binary search.

(Sections 2.8 and 2.8.3) **(10 Marks)**

➔ Summer 2012 – Total Marks 22

Q. 28 What is sorting ? Give its types.
(Section 2.1) **(2 Marks)**

Q. 29 Explain quick sort, state its advantage and disadvantages. Sort the following numbers in descending order using quick sort.

A = {3, 12, 5, 19, 1, 17}

(Sections 2.6 and 2.9.2, Example 2.6.3) **(8 Marks)**

Q. 30 Find the position of element 29 using binary search method in Array 'A' given below :

A = {11, 5, 21, 3, 29, 17, 2, 43}

Write 'C' program for binary search.

(Example 2.8.2, Program 2.8.2) **(8 Marks)**

Q. 31 Sort the following numbers in ascending order using Bubble sort.

A = {19, 2, 27, 3, 7, 5, 31}

(Example 2.2.1) **(4 Marks)**

➔ Winter 2012 – Total Marks 22

Q. 32 Give the principles of insertion sort along with algorithm.**(Section 2.4)** **(2 Marks)**

Q. 33 Explain efficiency of sorting algorithm.
(Section 2.1.2) **(2 Marks)**

Q. 34 What is searching and sorting ? Enlist different methods.

(Sections 2.1 and 2.8) **(2 Marks)**

Q. 35 Describe selection sort with algorithm and one example, assume suitable data. Write a program in 'C' language for selection sort.

(Section 2.3 and Program 2.3.1) **(8 Marks)**

Q. 36 Explain linear search and binary search with suitable example.

(Sections 2.8.1 and 2.8.3) **(4 Marks)**

➔ Summer 2013 – Total Marks 28

Q. 37 Write a 'C' program to sort ten numbers using bubble sort' technique. **(Program 2.2.1)** **(4 Marks)**

Q. 38 Write a program in 'C' language for selection sort and arrange the given number in ascending order using selection sort. Numbers - 16, 23, 13, 9, 7, 5.

(Program 2.3.1, Example 2.3.2) **(8 Marks)**

Q. 39 Compare quick sort and radix sort with respect to working principle and time complexity.

(Section 2.7.2) **(4 Marks)**

Q. 40 What is searching? Explain linear search with example. **(Sections 2.8 and 2.8.1)** **(4 Marks)**

Q. 41 Write a 'C' program for recursive binary search using function. **(Program 2.8.2)** **(4 Marks)**

➔ Winter 2013 – Total Marks 28

Q. 42 Define searching and sorting ? Enlist its types ?

(Sections 2.1 and 2.8) **(2 Marks)**

Q. 43 Describe the working principle of radix sort ?

(Section 2.5) **(4 Marks)**

Q. 44 Sort the number in ascending order using selection sort

A = { 42, 23, 74, 11, 65 }.

(Example 2.3.3) **(4 Marks)**

Q. 45 Write a 'C' program for bubble sorting method.

(Program 2.2.1) **(4 Marks)**

Q. 46 Write a 'C' program to insert an element in an array.

(Program 2.4.4) **(4 Marks)**

Q. 47 Find the position of element 88 using binary search method in

an array given below:

A = { 77, 33, 44, 11, 88, 22, 66, 55 }

Write a 'C' program for binary search.

(Example 2.8.3 and Program 2.8.2) **(8 Marks)**

➔ Summer 2014 – Total Marks 24

Q. 48 WAP to implement linear search for 10 elements in an array.
(Program 2.8.1) **(4 Marks)**

Q. 49 Write an algorithm for quick sort.
(Section 2.6) **(4 Marks)**

Q. 50 Write a program to arrange 10 elements in an array in ascending order. **(Program 2.3.1)** **(4 Marks)**

Q. 51 WAP to implement bubble sort.
(Program 2.2.1) **(4 Marks)**

Q. 52 Write an algorithm to implement binary search.
(Section 2.8.3) **(4 Marks)**

➔ Winter 2014 – Total Marks 24

Q. 53 Define sorting. Enlist different sorting technique.
(Section 2.1) **(2 Marks)**

Q. 54 Differentiate between the radix sort and shell sort methods. **(Section 2.7)** **(2 Marks)**

Q. 55 Write a program for sorting the array of ten elements using the Bubble sort method.
(Program 2.2.1) **(4 Marks)**

Q. 56 Describe binary search with an example.
(Section 2.8.3) **(4 Marks)**

Q. 57 Describe insertion sort algorithm and give steps of insertion sort for sorting the following list in ascending order.
List : 2, 15, 42, 26, 39, 92, 20
also find total number of comparison made.
(Example 2.4.3) **(8 Marks)**

➔ Summer 2015 – Total Marks 24

Q. 1(A)(c) Define sorting and give its type.
(Ans. : Refer section 2.1) **(2 Marks)**

Q. 1(A)(d) Give the principal of bubble sort.
(Ans. : Refer section 2.2) **(2 Marks)**

Q. 1 (B)(b) Explain the linear search algorithm. Also give its limitations.
(Ans. : Refer section 2.8.1) **(4 Marks)**

Q. 2(a) Sort the given number in ascending order using radix sort. Numbers : 348, 14, 614, 5381, 47.
(Ans. : Refer Example 2.5.1) **(4 Marks)**

Q. 2(f) Write a 'C' program for the selection sort.
(Ans. : Refer Program 2.3.1) **(4 Marks)**

Q. 5(a) Find the position of element 29 using binary search method in an array 'A' given below:
A = {11, 5, 21, 3, 29, 17, 2, 43}
Write a 'C' program for binary search.
(Ans. : Refer Example 2.8.2 and Program 2.8.2) **(8 Marks)**

➔ Winter 2015 – Total Marks 24

Q. 1(A)(c) Define searching. State two methods of searching.
(Ans. : Refer sections 2.8, 2.8.1 and 2.8.3) **(2 Marks)**

Q. 1(A)(g) What is sorting ? Enlist two categories of sorting. *(Ans. : Refer section 2.1)* **(2 Marks)**

Q. 1(B)(b) Distinguish between linear search and binary search. (any 4 points)
(Ans. : Refer Section 2.8.4) **(4 Marks)**

Q. 2(a) Describe quick sort. State its advantages and disadvantages.
(Ans. : Refer sections 2.6 and 2.6.3(C)) **(4 Marks)**

Q. 2(f) Write a program in 'C' language for selection sort. Write complexity of selection sort. *(Ans. : Refer program 2.3.1 and section 2.3)* **(4 Marks)**

Q. 5(a) Describe radix sort algorithm. Sort the following numbers in ascending order using radix sort. 12, 8, 25, 4, 66, 2, 98, 225.
(Ans. : Refer section 2.5 and Example 2.5.3) **(8 Marks)**

➔ Summer 2016 – Total Marks 24

Q. 1(b) Differentiate between binary search and sequential search.
(Ans. : Refer section 2.8.4) **(4 Marks)**

Q. 2(a) Write a program in 'C' to sort the given numbers in descending order using selection sort. *(Ans. : Refer Program 2.3.1)* **(4 Marks)**

Q. 3(a) Define internal and external sorting. Sort the following elements in descending order using radix sort.
7,103,15, 10, 3, 25, 28, 67, 304, 36, 49, 84
(Ans. : Refer section 2.1, Example 2.5.4) **(8 Marks)**

Q. 4(b) Find location of element 'H' by using binary search algorithm in the given list. **(4 Marks)**

B F D A C E I G H J

(Ans. : Refer Example 2.8.5)

Q. 5(a) Define computational complexity. Give complexity for the following :

i) Binary search ii) Bubble sort

iii) Quick sort iv) Linear search

(Ans. : Refer Example 2.8.4) **(4 Marks)**

Q. 5(b) Explain working of quick sort with a suitable example. *(Ans. : Refer section 2.6)* **(4 Marks)**

➔ Winter 2016 – Total Marks 24

Q.1(A)(3) Define searching. Give its type.

(Ans. : Refer section 2.8) **(2 Marks)**

Q.1(A)(7) State any four sorting technique. **(2 Marks)**

Ans. : Sorting techniques are :

(i) Bubble sort (ii) Selection sort

(iii) Insertion sort (iv) Merge sort

Q.1(B)(2) Describe binary search algorithm. Give example to search an element using binary search algorithm.

(Ans. : Refer section 2.8.3) **(4 Marks)**

Q. 2(a) Describe working of inserting sort. Demonstrate working of insertion sort algorithm to sort 6 elements.

(Ans. : Refer section 2.4) **(4 Marks)**

Q. 2(f) Write a program to search an element in an array. Display position of element.

(Ans. : Refer Program 2.8.1) **(4 Marks)**

Q. 5(a) Sort following elements by Radix sort algorithm

87.3, 2.34, 7.29, 3.59, 45.8, 3.79, 3.20, 422.

(Ans. : Refer Example 2.5.5) **(8 Marks)**

➔ Summer 2017 – Total Marks 16

Q. 2(a) Write a program for Binary search.

(Ans. : Refer Program 2.8.2) **(4 Marks)**

Q. 3(b) Write a program for selection sort.

(Ans. : Refer Program 2.3.1) **(4 Marks)**

Q. 4(a) Elaborate the steps for performing insertion sort for given elements of array.

30 10 40 50 20 45

(Ans. : Refer Example 2.4.4) **(4 Marks)**

Q. 5(a) Write a program for linear search. Find position of element 30 using linear search algorithm in given sequence.

10 5 20 25 8 30 40

(Ans. : Refer Program 2.8.1) **(4 Marks)**

➔ Winter 2017 – Total Marks 16

Q. 1(A)(c) State any two differences between linear search and binary search.

(Ans. : Refer Section 2.8.4) **(4 Marks)**

Q. 1(A)(f) Give complexity of following methods. **(2 Marks)**

(i) Bubble sort

(Ans. : Refer section 2.7.1[O(n^2)])

(ii) Radix sort

(Ans. : Refer section 2.7.1[O(n)])

(iii) Linear search

(Ans. : Refer section 2.8.1[O(n)])

(iv) Binary search

(Ans. : Refer section 2.8.3[O(log n)])

Q. 1(B)(b) Perform radix sort on the following list to arrange all array elements in ascending order :

18, 253, 1000, 2, 80, 75, 58

(Ans. : Refer Example 2.5.6) **(4 Marks)**

Q. 5(a) Describe working of binary search method. Given stepwise procedure to search 65 in the following list :

List : 23, 12, 5, 29, 10, 65, 55, 70.

(Ans. : Refer section 2.8.3 and Example 2.8.6) **(8 Marks)**

❑❑❑

CHAPTER 3

Unit - III

Stacks

Syllabus Topics

Introduction to Stack

- Stack representation in memory using array
- Stack as an ADT
- Stack Operations – PUSH, POP
- Stack Operations Conditions – Stack Full/Stack Overflow, Stack Empty/Stack Underflow.
- Applications of stack
  - Reversing a Stack
  - Polish notations

Conversion of infix to postfix expression, Evaluation of postfix expression, Converting an infix into prefix expression, Evaluation of prefix expression Recursion, Tower of Hanoi

Syllabus Topic : Introduction to Stack

3.1 Introduction to Stack

➔ **(MSBTE : S-09, S-10, W-12, S-13, S-15, S-16)**

Q. 3.1.1	**State the principle of stack with basic operations. (Refer section 3.1)**	**S-09**
Q. 3.1.2	**Define stack. (Refer section 3.1)**	**S-10, S-13, S-15**
Q. 3.1.3	**What is stack ? (Refer section 3.1)**	**W-12**
Q. 3.1.4	**State importance of top pointer in stack. (Refer section 3.1)**	**S-16**

- Stack is a LIFO (Last In First Out) structure. It is an ordered list of the same type of elements. A stack is a linear list where all insertions and deletions are permitted only at one end of the list. When elements are added to stack it grows at one end. Similarly, when elements are deleted from a stack, it shrinks at the same end.
- Fig. 3.1.1 shows expansion and shrinking of a stack. Initially stack is empty.

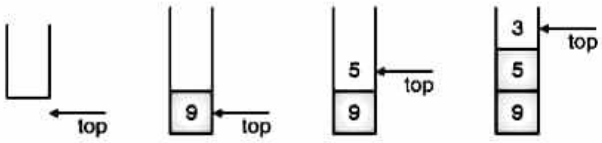


Fig. 3.1.1 : Insertion of 9,5,3 in a stack

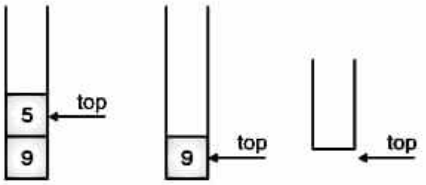


Fig. 3.1.2 : Deletion of 2 elements from the stack

- A variable top, points to the top element of the list.

Syllabus Topic : Stack as an ADT

3.2 Stack as an ADT

➔ **(MSBTE : S-14, S-15, W-15, S-17, W-17)**

Q. 3.2.1 Explain stack as an abstract data type. (Refer section 3.2) S-14, S-15, S-17, W-17

Q. 3.2.2 Describe the term stack overflow and stack underflow with example. (Refer sections 3.2(4), 3.2(5)) W-15

Stack is a LIFO structure. Stack can be represented using an array. A one-dimensional array can be used to hold elements of a stack. Another variable "top" is used to keep track of the index of the top most element.

Formally, a stack may be defined as follows :

```
typedef struct stack
    {
        int data[SIZE];
        int top;
    } stack;
/* SIZE is a constant, maximum number of elements that can
be stored */
```

Associated with the object stack there are several operations that are necessary :

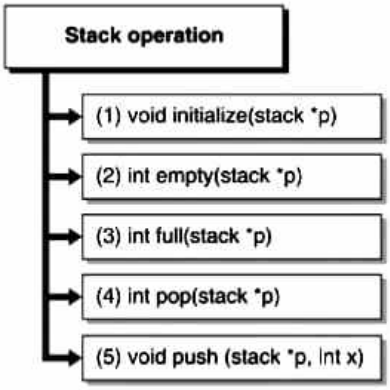


Fig. 3.2.1 : Stack Operation

➔ **(1) void initialize (stack *p)**

It initializes a stack as an empty stack. Initial value of top is set to –1.

➔ **(2) int empty (stack *p)**

Function checks whether the stack is empty. It returns 1 or 0 depending on whether the stack is empty or not.

➔ **(3) int full (stack *p)**

The function checks whether the stack is full. A stack is said to be full when the associated array is completely filled up. Whenever the stack is full, top points to the last element (i.e. data[SIZE-1]) of the array. The function returns 1 or 0 depending on whether the stack is full or not.

➔ **(4) int pop (stack *p)**

The function deletes topmost element from the stack and also returns it to the calling program. Deletion from an empty stack will cause underflow.

➔ **(5) void push (stack *p, int x)**

The function inserts the element x onto the stack pointed by P. Insertion will cause an overflow if the stack is full.

3.3 Operations on Stacks

➔ **(MSBTE : S-09, S-10, S-11, S-13, W-13, W-15)**

Q. 3.3.1 State the principle of stack with basic operations. (Refer section 3.3) S-09

Q. 3.3.2 What are the basic operations performed on it ? (Refer section 3.3) S-10, S-13

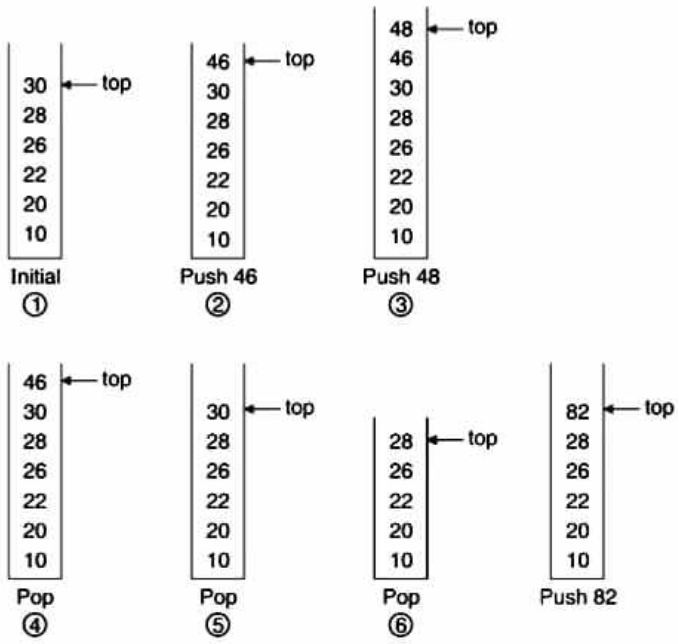
Q. 3.3.3 Explain four primitive operations on stack. (Refer section 3.3) S-11, W-13

Q. 3.3.4 Define push and pop operation of stack. (Refer section 3.3) W-15

1. Initialize( ) - Make a stack empty
2. Empty( ) - To determine if a stack is empty or not
3. Full( ) - To determine if a stack is full or not
4. Push( ) - if a stack is not full then push a new element at the top of the stack (similar to insert in a list)
5. Pop( ) - If a stack is not empty, then pop the element from its top(similar to delete( ) from a list)

Example 3.3.1 : The stack contain 10, 20, 22, 26, 28, 30, with 30 being at top of the stack. Show diagrammatically the affect of
PUSH 46
PUSH 48
POP
POP
POP
PUSH 82 **S-14**

Solution :



Syllabus Topic : Stack Representation in Memory using Array

3.4 Array Representation

➔ **(MSBTE : S-14, W-15)**

Q. 3.4.1	**Explain the concept of representing stack through arrays. (Refer section 3.4)**	**S-14**
Q. 3.4.2	**Describe the representation of stack using arrays. (Refer section 3.4)**	**W-15**

A one – dimensional array can be used to hold elements of a stack. Another variable "top" is used to keep track of the index of the top most element.

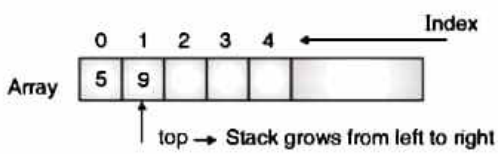


Fig. 3.4.1

Formally, a stack may be defined as follows :

```
typedef struct stack
 {
        int data[MAX] ;
        int top ;
 }stack ;
/*MAX is a constant, maximum number of elements that
can be stored */
```

- Initially, top is set to –1
- A stack with top as –1 is an empty stack.
- When the value of top becomes MAX – 1 after a series of insertions, it is full.
- After "push" operation top = top + 1 (Stack growing)
- After "pop" operation top = top – 1 (Stack shrinking)

Syllabus Topic : Stack Operations-PUSH, POP, Stack Operations Conditions- Stack Full, Stack Empty

3.4.1 'C' Functions for Primitive Operations on a Stack

➔ **(MSBTE : S-12, S-13, W-13, S-14, W-14, S-15, W-16, S-17)**

Q. 3.4.3 Explain PUSH and POP operation on stack using array representation. (Refer section 3.4.1) **S-12, W-14, S-15, W-16**

Q. 3.4.4 Write a procedure to PUSH on element on stack. (Refer section 3.4.1) **S-13**

Q. 3.4.5 Write a 'C' program to implement PUSH and POP function in stack as an array. (Refer section 3.4.1) **W-13**

Q. 3.4.6 What is the affect of PUSH and POP operation on to the stack ? (Refer section 3.4.1) **S-14**

Q. 3.4.7 Explain the concept of representing stack through arrays. (Refer section 3.4.1) **S-14**

Q. 3.4.8 Explain operations on stack using array. (Refer section 3.4.1) **S-17**

```
void initialize(stack *s)
{
    s → top = –1 ;
}
```

Initially, stack is empty.

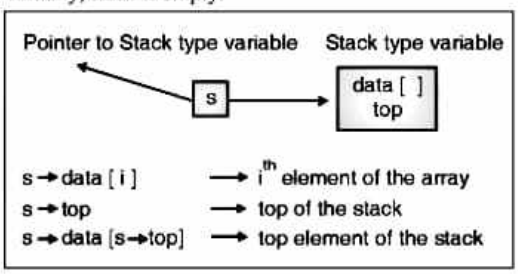


Fig. 3.4.2 : Accessing elements of a stack using a pointer

```
int empty(stack * s)
{
        if(s → top == – 1)
        return(1) ; /* stack is empty */
    return(0) ; /* stack empty condition is false*/
}
```

Above function checks whether a stack is empty [containing 0 elements]. Function returns a value 1(true) if the stack is empty. Function returns a value 0(false) if the stack is not empty (there are some elements).

```
int full(stack *s)
{
        if( s→top == MAX –1)
            return(1);
        return(0);
}
```

A stack is full, if no more elements can be added.

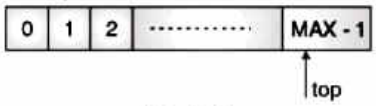


Fig. 3.4.3

Above function checks whether a stack is full (no more elements can be added). Function returns a value 1(true) if the stack is full. Function returns a value 0(false) if the stack is not full (more elements could be added).

```
void push(stack *s, int x)
{
    s → top = s → top + 1 ;
    s → data[s → top] = x;
}
```

Main program should ensure that the stack is not full before making a call to push( ) function. It may cause an overflow, otherwise.

```
int pop(stack *s)
{
    int x ;
    x = s → data[s → top] ;
    s → top = s →top –1 ;
    return(x);
}
```

Main program should ensure that the stack is not empty before making a call to pop( ) function. It may cause an underflow, otherwise.

The function push( ), increases the value of the top by 1 and the x stores the element at the top location in "data" array.

The function pop( ) returns the top element from the stack. It also, reduces the value of top by 1 (one element is deleted from the stack).

3.5 Program Showing Stack Operations

Convert a decimal number to binary using a stack :

Example

Input no = 21 (Decimal)

21%2 = 1	1
10%2 = 0	0 1
5%2 = 1	1 0 1
2%2 = 0	0 1 0 1
1%2 = 1	1 0 1 0 1

$(21)_{10} = (10101)_2$

A decimal number can be converted to binary form through repeated division of the number by 2. Remainder at each step is saved in a stack. Finally, the bits stored in stack can be printed one by one through POP( ) operation.

Program 3.5.1 : Program for conversion of decimal number to binary form.

```
#include<stdio.h>
#include<conio.h>
#define MAX 20
typedef struct stack
{
    int data[MAX];
    int top;
}stack;
void init(stack *);
int empty(stack *);
int full(stack *);
int pop(stack *);
void push(stack *,int);
void main()
{
    stack s;
    int x;
    init(&s);
    printf("\nEnter decimal number:");
    scanf("%d",&x);
    while((x!=0))
    {
        if(!full(&s))
        {
            push(&s,x%2);
            x=x/2;
        }
        else
        {
            printf("\nstack overflow");
            exit(0);
        }
```

```
    }
    printf("\n");
    while(!empty(&s))
    {
        x=pop(&s);
        printf("%d",x);
    }
}
void init(stack *s)
{
    s→top= –1;
}
int empty(stack *s)
{
    if(s→top= = –1)
        return(1);
    return(0);
}
int full(stack *s)
{
    if(s→top= =MAX-1)
        return(1);
    return(0);
}
void push(stack *s,int x)
{
    s→ top=s→ top+1;
    s→ data[s→ top]=x;
}
int pop(stack *s)
{
    int x;
    x=s→ data[s→ top];
    s→ top=s→ top –1;
    return(x);
}
```

Output

```
Enter decimal number:12
1100
```

Example 3.5.1 : Using stack functions, write an algorithm to determine if an input character string is of the form : $a^i b^i$ where $i \geq 1$

Solution :

```
1.  s : stack          [s is a stack]
2.  initialize s
3.  while (! end of input)
     {
            x ← read next character
            if (x is a)
              push (s, a);
            else
              pop (s); goto step 4.
     }
4.  while (! end of input)
        {
            x ← read next character
            if (stack is empty)
              {
                print "b's are more than a's"
                goto step 6.
              }
            else
                pop(s)
        }
5.  if (stack is empty)
            print "a's are equal to b's"
            else
              print "a's are more than b's"
6.  End
```

Syllabus Topic : Stack Overflow, Stack Underflow

3.5.1 Operations on Stack Considering Overflow and Underflow [Array Implementation]

➔ **(MSBTE : W-08, W-09, S-10, S-12, S-13, W-14, S-16, W-17)**

Q. 3.5.1 Explain the terms overflow and underflow with respect to stack. Use suitable data and diagram. (Refer section 3.5.1) **W-08, W-09, S-12, W-14**

Q. 3.5.2 Write a procedure to push an element on stack. Also give meaning of stack overflow term. (Refer section 3.5.1) **S-10**

Q. 3.5.3 Give meaning of stack overflow. (Refer section 3.5.1) **S-13**

Q. 3.5.4 Explain stack overflow and underflow. (Refer section 3.5.1) **S-16**

Q. 3.5.5 Describe following terms with suitable example.

i) Stack overflow

ii) Stack underflow (Refer section 3.5.1) **W-17**

☞ **Stack overflow :** Stack overflow is caused by insertion in a stack that is already full.

☞ **Stack underflow :** Deletion from an empty stack will cause a stack underflow.

➢ **'C' functions for operations on stack.**

```
void push()
{
    if(p→ top==N –1)
        printf("\noverflow !! cannot be inserted");
    else
    {
        p → top=p→ top+1;
        p→ data[p→ top]=x;
    }
}
int pop(stack *p)
{
    int x;
    if(p→ top== –1)
        printf("\nUnderflow !!! cannot be deleted");
    else
    {
        x=p→ data[p->top];
        p→ top=p→ top –1;
    }
    return(x);
}
void print(stack *p)
{
    int i;
    for(i=p→ top;i>=0;i–)
        printf("\n%d",p→ data[i]);
}
```

➢ **Program 3.5.2 : C-Program for implementing stack using array.**

Q. 3.5.6 Write a menu driven 'C' program for implementing stack using array. (Refer Program 3.5.2) **S-08, S-13**

```
/* Simulation of stack using an array */

#include<stdio.h>
#include<conio.h>
#define MAX 6
typedef struct stack
{
    int data[MAX];
    int top;
}stack;
void init(stack *);
```

```
int empty(stack *);
int full(stack *);
int pop(stack *);
void push(stack *,int);
void print(stack *);
void main()
{
    stack s;
    int x,op;
    init(&s);
    clrscr();
    do {
        printf("\n\n1)Push\n2)Pop\n3)Print\n4)Quit");
        printf("\nEnter Your choice: ");
        scanf("%d",&op);
        switch(op)
         {
        case 1:printf("\n enter a number :");
              scanf("%d",&x);
              if(!full(&s))
               push(&s,x);
              else
               printf("\nStack is full.....");
              break;
        case 2:if(!empty(&s))
              { x=pop(&s);
               printf("\npopped value= %d",x);
              }
              else
             printf("\nStack is empty.....");
              break;

        case 3:print(&s);break;
        }
    }while(op!=4);
}
void init(stack *s)
{
    s→ top=-1;
}
int empty(stack *s)
{
    if(s→ top==-1)
        return(1);
    return(0);
}
int full(stack *s)
{
    if(s→ top==MAX-1)
        return(1);
    return(0);
}
void push(stack *s,int x)
{
    s→ top=s→ top+1;
    s→data[s→ top]=x;
}
int pop(stack *s)
{
    int x;
    x=s→ data[s→ top];
    s→ top=s→ top-1;
    return(x);
}
void print(stack *s)
 {
    int i;
    printf("\n");
```

```
    for(i=s→ top;i>=0;i--)
        printf("%d ",s→ data[i]);
}
```

Syllabus Topic : Reversing a List

3.5.2 Reversing a List

➔ **(MSBTE : S-16)**

Q. 3.5.7 **Explain how stack can be used to reverse a list using suitable example. (Refer section 3.5.2)** **S-16**

Stack is a LIFO (Last In First Out) structure. It is an ordered list of the same type of elements. A stack is a linear list where all insertions and deletions are permitted only at one end of the list. Reversal of list is application of stack.

In reverse list, element is push one by one into stack, once all elements are inserted(pushed) into stack, then last element comes first at the time of pop(delete) operation. Because of in stack first topmost element is pop. Hence reversal of list occurs.

☞ **For Example**

Data	Operation	Stack	
3	Push	3	
5	Push	5 3	
7	Push	7 5 3	
–	POP	5 3	7
–	POP	3	7, 5
–	POP		7, 5, 3

Syllabus Topic : Applications of Stack

3.6 Applications of Stack

➔ **(MSBTE : S-08, S-09, W-13, W-14, W-15, S-17)**

Q. 3.6.1 **Explain any two applications of stack. (Refer sections 3.6, 3.6.2, 3.6.4 and 3.6.5)** **S-08, W-13**

Q. 3.6.2 **Write down applications of stacks. (Refer sections 3.6, 3.6.2, 3.6.4 and 3.6.5)** **S-09, W-14**

Q. 3.6.3 **State any four applications of stack. (Refer section 3.6)** **W-15**

Q. 3.6.4 **Explain any three application of stack in detail with example. (Refer section 3.6)** **S-17**

Stack data structure is very useful. Few of its usages are given below :

1. Expression conversion
   (a) Infix to postfix (b) Infix to prefix
   (c) Postfix to infix (d) Prefix to infix
2. Expression evaluation
3. Parsing
4. Simulation of recursion
5. Function call

Syllabus Topic : Polish Notations

3.6.1 Expression Representation

There are three popular methods for representation of an expression.

(a) infix x +y operator between operands

(b) prefix + x y operator before operands

(c) postfix x y + operator after operands

Example

Infix x + y * z

Prefix + x * y z – Polish notation

Postfix x y z * + – Reverse polish notation.

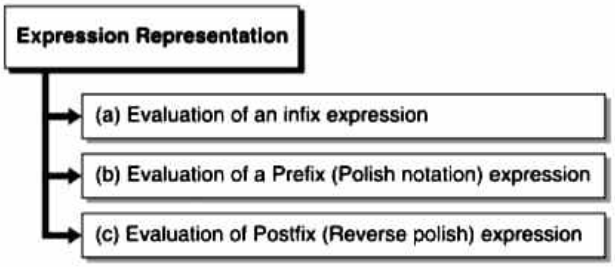


Fig. 3.6.1 : Expression representation

➜ (a) Evaluation of an infix expression

Infix expressions are evaluated left to right but operator precedence must be taken into account. To evaluate x + y * z, y and z will be multiplied first and then it will be added to x.

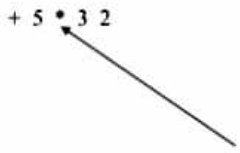
Infix expressions are not used for representation of an expression inside computer, due to additional complexity of handling of precedence.

➜ (b) Evaluation of a Prefix (Polish notation) expression

To understand the evaluation of prefix expression, let us consider an example

+ 5 * 3 2

Find an operator from right to left and perform the operation.



first operator

First operator is * and therefore, 3 * 2 are multiplied, expression becomes + 5 6.

First operator is + and therefore, 5 and 6 are added, expression becomes : 11

➜ (c) Evaluation of Postfix (Reverse polish) expression

5 3 2 * +

1. Find the first operator from left to right and perform the operation
2. First operator is * and therefore, 3 and 2 are multiplied expression becomes 5 6 +
3. Next operator is + and therefore, 5 and 6 are added expression becomes : 11

Prefix and postfix expressions are free from any precedence. They are more suited for mechanization. Computer uses postfix form for representation of an expression.

Q.3.6.5 Discuss the polish notation of arithmetic expression in application of stacks. (Refer section 3.6.1) W-11

Polish notation is also known as prefix notation. In this notation, the operator comes before operands.

☞ For example

The polish notation of the infix expression x + y * z is given by

+ x * yz .

The prefix form of an expression is free from precedence and hence it can be used for representation of an arithmetic expression inside a computer.

Stack data structure is very useful in context of prefix expression. Few of its usages are given below :

(a) Evaluation of a prefix expression.

(b) Conversion from infix to prefix form

Evaluation of a prefix expression is discussed in the **Section 3.6.3** and the conversion from infix to prefix form is discussed in the **Section 3.6.5.**

Syllabus Topic : Evaluation of a Postfix Expression

3.6.2 Evaluation of a Postfix Expression using a Stack

➔ **(MSBTE : S-08, S-09)**

Given expression :

Stack

6 5 3 + 9 * +

Initially stack is empty.

- First token is an operand, push 6 on the stack.

5 3 + 9 * +

- Next token is and operand, push 5 on the stack.

3 + 9 * +

- Next token is and operand, push 3 on the stack.

+ 9 * +

- Next token is an operator, pop two operands 3 and 5, add them and push the result on the stack.

9 * +

- Next token is an operand, push 9 on the stack.

* +

- Next token is an operator, pop two operands 9 and 8, multiply them and push the result on the stack.

+

- Next token is an operator, pop two operands 72 and 6, add them and push the result back on the stack.

NULL

Final Result

☞ **Algorithm for evaluation of an expression**

```
x : tokentype
        operand1, operand2 : operand_type ;
        operator : operator_ type ;
s : stack
        Initialize(s) ;
        x = nexttoken( ) ; /* read the next token */
          while(x)
        {
            if(x is an operator)
            {
                operand2 = pop(s);
                operand1 = pop(s);
                push(s, evaluate(x, operand1, operand2));
            }
            else
                push(s, x) ;
            x = nexttoken( ) ;
        }
    /* stack contains the evaluated value of the expression */
```

➢ **Program 3.6.1 : Program for evaluation of a postfix expression**

```
Assumption-- primary operators '-,+,*,/,%' operand -- a single digit
#include<stdio.h>
#include<conio.h>
#define MAX 20
typedef struct stack
{
    int data[MAX];
    int top;
}stack;
void init(stack *);
int empty(stack *);
int full(stack *);
```

```
int pop(stack *);
void push(stack *,int);
int evaluate(char x,int op1,int op2);
void main()
{
    stack s;
    char x;
    int op1,op2,val;
    init(&s);
    printf("\nEnter the expression(i.e 59+3*)\single digit
operand and operators only:");
    while((x=getchar())!='\n')
    {
        if(isdigit(x))
            push(&s,x-48); //x-48 for removing the effect of
ASCII
        else
        {
            op2=pop(&s);
            op1=pop(&s);
            val=evaluate(x,op1,op2);
            push(&s,val);
        }
    }
    val=pop(&s);
    printf("\nvalue of expression = %d",val);
}
int evaluate(char x,int op1,int op2)
{
    if(x=='+')
        return(op1+op2);
    if(x=='-')
        return(op1-op2);
    if(x=='*')
        return(op1*op2);
    if(x=='/')
        return(op1/op2);
    if(x=='%')
        return(op1%op2);
}
void init(stack *s)
{
    s-> top= -1;
}
int empty(stack *s)
{
    if(s-> top== -1)
        return(1);
    return(0);
}
int full(stack *s)
{
    if(s-> top==MAX -1)
        return(1);
    return(0);
}
void push(stack *s,int x)
{
    s-> top=s-> top+1;
    s-> data[s-> top]=x;
}
int pop(stack *s)
{
    int x;
    x=s-> data[s-> top];
    s-> top=s-> top -1;
    return(x);
}
```

Output

```
Enter the expression(i.e 59+3*)
single digit operand and operators only:653+9*+
value of expression = 78
```

Example 3.6.1 : Evaluate the following postfix expression and show stack after every step in tabular form. Given A = 5, B = 6, C = 2, D = 12, E = 4 ABC + *DE\– **W-10**

Solution :

Input	Stack
ABC + * DE/–	
BC + *DE/–	5
C + * DE/–	6 5
+ * DE/–	2 6 5
* DE/-	8 5
DE/–	40
E/–	12 40
/–	4 12 40
–	3 40
End	37

Value of the expression = 37

Example 3.6.2 : Consider the following arithmetic expression p, written in postfix notation. Translate it in infix notation and evaluate. **W-08**

Solution :

P : 12, 7, 3, –, /, 2, 1, 5, + *, +

A postfix expression can be converted to infix form by scanning it from left to right.

Step 1 : 12, $\boxed{7-3}$, /, 2, 1, 5, + *, +

Step 2 : $\boxed{12/(7-3)}$, 2, 1, 5, +, *, +

Step 3 : $\boxed{12/(7-3)}$, 2, $\boxed{1+5}$, *, +

Step 4 : $\boxed{12/(7-3)}$, $\boxed{(2*(1+5))}$ +

Step 5 : 12 / (7 – 3) + (2 * (1 + 5))

Value of the expression = 12 / (7 – 3) + (2 * (1 + 5))

= 12/4 + 2 * 6 = 3 + 2 * 6 = 15

Example 3.6.3 : Evaluate the following postfix expressions.

(i) 5 4 6 + * 4 9 3 / + *

(ii) 7 5 2 + * 4 1 1 + / –

Solution :

(i) 5 4 6 + * 4 9 3 / + *

	Input	Stack
1.	5 4 6 + * 4 9 3 / + *	
2.	4 6 + * 4 9 3 / + *	5
3	6 + * 4 9 3 / + *	4 5

	Input	Stack	
4.	+ * 4 9 3 / + *	6 4 5	
5.	* 4 9 3 / + *	10 5	4 + 6 = 10
6.	4 9 3 / + *	50	5 * 10 = 50
7.	9 3 / + *	4 50	
8.	3 / + *	9 4 50	
9.	/ + *	3 9 4 50	
10.	+ *	3 4 50	9/3 = 3
11.	*	7 50	4 + 3 = 7
12.	End	350	

Result = 350

(ii) 7 5 2 + * 4 1 1 + / –

	Input	Stack	
1.	7 5 2 + * 4 1 1 + / –		
2.	5 2 + * 4 1 1 + / –	7	
3	2 + * 4 1 1 + / –	5 7	
4.	+ * 4 1 1 + / –	2 5 7	
5.	*4 1 1 + / –	7 7	5 + 2 = 7
6.	4 1 1 + / –	49	7 * 7 = 49
7.	1 1 + / –	4 49	
8.	1 + / –	1 4 49	
9.	+ / –	1 1 4 49	

	Input	Stack	
10.	/ –	2 4 49	
11.	–	2 49	4 / 2 = 2
12.	End	47	49 – 2 = 47

Result = 47.

Example 3.6.4 : **Evaluate following postfix expression.**
A : 6, 2, 3, +, –, 3, 8, 2, +, +, *, 2, ↑, 3, + W-09, W-13

Solution :

	Input	Stack	
1.	6, 2, 3, +, –, 3, 8, 2, +, +, *, 2, ↑, 3, +	Empty	
2.	2, 3, +, –, 3, 8, 2, +, +, *, 2, ↑, 3, +	6	
3.	3, +, –, 3, 8, 2, +, +, *, 2, ↑, 3, +	2 6	
4.	+, –, 3, 8, 2, +, +, *, 2, ↑, 3, +	3 2 6	
5.	–, 3, 8, 2, +, +, *, 2, ↑, 3, +	5 6	2 + 3 = 5
6.	3, 8, 2, +, +, *, 2, ↑, 3, +	1	6 – 5 = 1
7.	8, 2, +, +, *, 2, ↑, 3, +	3 1	
8.	2, +, +, *, 2, ↑, 3, +	8 3 1	
9.	+, +, *, 2, ↑, 3, +	2 8 3 1	
10.	+, *, 2, ↑, 3, +	10 3 1	8 + 2 = 10
11.	*, 2, ↑, 3, +	13 1	3 + 10 = 13
12.	2, ↑, 3, +	13	1 × 13 = 13
13.	↑, 3, +	2 13	
14.	3, +	169	$13^2 = 169$
15.	+	3 169	
16.	End	172	169 + 3 = 172

∴ Result = 172

Syllabus Topic : Evaluation of a Prefix Expression

3.6.3 Evaluation of a Prefix Expression

- A prefix expression is evaluated by scanning it from right to left.
- An operand is pushed on top of the stack.
- In case of an operator, two operands are popped from the stack, evaluated and pushed back onto the stack.
- Final result is found on top of the stack.

Example 3.6.5 : Evaluate the following prefix expression using stack for A = 16, B = 2 , C = 3, D = 10 and E = 4. Show step by step the contents of stack.

Prefix Expression =

– + / A∧ BC * DE * AC

Solution :

Input	Stack	
– + / A∧BC * DE * AC		
– + / A∧BC * DE * A	3	Push
– + / A∧BC * DE *	16 3	Push
– + / A∧BC * DE	48	Evaluate
– + / A∧BC * D	4 48	Push
– + / A∧BC *	10 4 48	Push
– + / A∧BC	40 48	Evaluate
– + / A∧B	3 40 48	Push
– + / A∧	2 3 40 48	Push
– + / A	8 40 48	Evaluate
– + /	16 8 40 48	Push
– +	2 40 48	Evaluate
–	42 48	Evaluate
Null	– 6	Evaluate

Value of expression = – 6

☞ **Algorithm for evaluation of a prefix expression**

```
st : stack ;
initialize st ;
Prefix ← read the prefix expression ;
for ( i ← index of the last character of prefix, down to 0 )
    {  x ← prefix[i] ;
       if ( x is an operand )
       {  push(st, char_to_int(x)) ;
```

```
        }
    else
        {   operand1 ← pop(st) ;
            operand2 ← pop(st) ;
            val ← evaluate( X, operand1 , operand2)
            push(st, val) ;
        }
    }
    val  =  pop(st) ;
    print val ;
```

➢ **Program 3.6.2 : Evaluation of a prefix expression.**

```
/* program for Evaluation of prefix expression
  operators supported '+,-,*,/,%,^,(,)
  operands supported -- all single character operands */
#include<stdio.h>
#include<conio.h>
#include<ctype.h>
#define MAX 50

typedef struct stack
{
    int data[MAX];
    int top;
}stack;
void init(stack *);
int  empty(stack *);
int  full(stack *);
int  pop(stack *);
void push(stack *,int );
int  top(stack *); //value of the top element
void eval_prefix(char prefix[]);
int  evaluate(char x,int op1,int op2);
void main()
{ char prefix[30];
  clrscr();
  printf("\nEnter a prefix expression : ");
  gets(prefix);
  eval_prefix(prefix);
  getch();
}
void eval_prefix(char prefix[])
{
   stack s;
   char x;
   int op1,op2,val,i;
   init(&s);
   for(i=strlen(prefix)-1;i>=0;i--)
    {    x=prefix[i];
       if(isalpha(x))
          {  printf("\nEnter the value of %c : ",x);
          scanf("%d",&val);
          push(&s,val);
          }
       else
       {    //pop two operands and evaluate
          op1=pop(&s);
          op2=pop(&s);
          val=evaluate(x,op1,op2);
          push(&s,val);
        }
    }
    val=pop(&s);
    printf("\nvalue of expression = %d",val);
}
int evaluate(char x,int op1,int op2)
{
    if(x=='+')  return(op1+op2);
    if(x=='-')  return(op1- op2);
```

```
    if(x=='*') return(op1*op2);
    if(x=='/') return(op1/op2);
    if(x=='%') return(op1%op2);
}
void init(stack *s)
{
    s→ top=-1;
}
int empty(stack *s)
{
    if(s→ top== -1)     return(1);
    return(0);
}
int full(stack *s)
{
    if(s→ top==MAX -1)   return(1);
    return(0);
}
void push(stack *s, int x)
{
    s→ top=s→ top+1;
    s→ data[s→ top]=x;
}
int pop(stack *s)
{
    int x;
    x=s→ data[s→ top];
    s→ top=s→ top -1;
    return(x);
}
int top(stack * p)
{
    return(p→ data[p→ top]);
}
```

Syllabus Topic : Conversion of Expression Infix to Postfix

3.6.4 Conversion of an Expression from Infix to Postfix

➔ **(MSBTE : S-08, S-09)**

Example of an infix expression : (A + B ∧ C) * D + E ∧ 5 where A, B, C, D and E are integer constants and B ^ C means B^C

☞ **Manual method**

Step 1 :

In manual evaluation of the above expression, B ^ C must be calculated first. Hence, we convert B ∧ C to its equivalent postfix equivalent i.e. BC∧

(A + B ∧ C) * D + E ∧ 5 – original expression

(A + <u>BC∧</u>) * D + E ∧ 5 – BC∧ should treated as a single integer number (single token) and A + <u>BC∧</u> should be the next operation to be performed

<u>ABC∧ +</u> * D + E ∧ 5 – <u>ABC∧ +</u> * D is the next operation to be performed

<u>ABC∧ + D*</u> + E ∧ 5 – E ∧ 5 is the next operation to be performed

<u>ABC∧ + D*</u> + <u>E5∧</u> – + is the last operation to be performed

<u>ABC∧ + D* E5∧ +</u> → Final expression.

☞ **Algorithmic approach**

A + B * C

– In the above example, evaluation of * precedes evaluation of + as * has higher precedence over +.

Perform an operation if the current operator has equal or higher precedence over the succeeding operator.

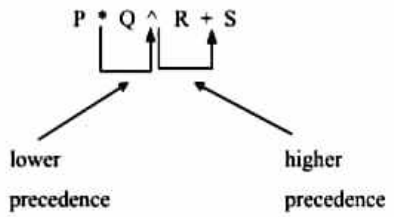
– Thus an operator coming on the right of the current operator will determine if the current operation should be performed.

P + Q + R

Same precedence and hence P + Q can be performed

P * Q + R

Higher precedence and hence P * Q can be performed



Sequence in which operations will be performed ^, *, +

- All operators must be saved on top of the stack until we get an equal or lower precedence operator.

Conversion of P * Q^R + S

Expression	Stack	Output	Remark
P*Q ∧ R + S	NULL	–	–
*Q∧R + S	NULL	P	Operand must be printed
Q∧R + S	*	P	* operation will be performed if the next operator is of lower or equal precedence.
∧R + S	*	PQ	Operand must be printed
R + S	* ∧	PQ	* Cannot be performed as ∧ has higher precedence
+ S	* ∧	PQR	Operand must be printed
+ S	NULL	PQR∧*	All higher precedence operators (compare to +) are popped and printed and finally the current operator is pushed on top of the stack.
S	+	PQR∧*	–
NULL	+	PQR∧* S	Operand must be printed
Null	Null	PQR∧* S +	Finally, all operators are popped and printed.

- **Algorithm for conversion of an expression from infix to postfix**

```
s : stack
while(more token)
{
    x ← next token ;
    if(x is an operand)
        print x
    else
    {
        while(precedence(x) <= precedence(top(s))
            print(pop(s))
        push(s,x)
    }
}
while(! empty(s))
print(pop(s)) ;
```

Example 3.6.6 : Convert the following expression from infix to postfix using a stack.

a && b ¦¦ c ¦¦ ! (e > f)

Solution :

Expression	Stack	Output	Comment
a &&b¦¦ c ¦¦ ! (e>f)	NULL	–	Initial condition
&&b ¦¦ c ¦¦ ! (e >f)	NULL	a	Print a
b ¦¦ c ¦¦ ! (e > f)	&&	a	Push &&
¦¦ c ¦¦ ! (e>f)	&&	ab	Print b
¦¦ c ¦¦ ! (e>f)	NULL	ab &&	pop and print higher precedence operators
c ¦¦ ! (e>f)	¦¦	ab &&	push the current operator
¦¦ ! (e>f)	¦¦	ab&& c	Print c

Expression	Stack	Output	Comment
‖ ! (e>f)	NULL	ab&&c ‖	Pop and print equal or higher precedence operators
!(e > f)	‖	ab&&c ‖	Push the current operator
! (e>f)	‖	ab&&c ‖	Pop and print equal or higher precedence operators
(e > f)	‖ !	ab&&c ‖	Push the current operator
e > f	‖ ! (	ab&&c ‖	'(' should always be pushed.
> f)	‖ ! (	ab&&c ‖ e	Print e
> f)	‖ ! (	ab&&c ‖ e	Pop and print equal or higher precedence operators. Remember '(' has lowest precedence
f)	‖ ! (>	ab&&c ‖ e	Push the current operator
)	‖ ! (>	ab&&c ‖ e f	Print f
NULL	‖ !	ab&&c ‖ e f >	when the next input is ')' then all operators until '(' should be popped and printed
NULL	NULL	ab&&c ‖ e f>! ‖	Pop and print all operations.

Example 3.6.7 : Convert the following arithmetic expression into postfix and show stack status after every step at each step :

(i) (A + B) * D + E / (F + A * D) + C

(ii) A * (B + C) / D – G

Solution :

(i) (A + B) * D + E / (F + A * D) + C

Stack	Input	Output
Empty	(A + B) * D + E / (F + A * D) + C	–
(	A + B) * D + E / (F + A * D) + C	–
(	+ B) * D + E / (F + A * D) + C	A
(+	B) * D + E / (F + A * D) + C	A
(+	) * D + E / (F + A * D) + C	AB
Empty	* D + E / (F + A * D) + C	AB+
*	D + E / (F + A * D) + C	AB+
*	+ E / (F + A * D) + C	AB+D
+	E / (F + A * D) + C	AB+D*
+	/ (F + A * D) + C	AB+D*E
+/	(F + A * D) + C	AB+D*E
+/(	F + A * D) + C	AB+D*E
+/(	+ A * D) + C	AB+D*EF
+/(+	A * D) + C	AB+D*EF
+/(+	* D) + C	AB+D*EFA
+/(+*	D) + C	AB+D*EFA
+/(+*	) + C	AB+D*EFAD
+/	+ C	AB +D*EFAD*+
+	C	AB +D*EFAD*+ / +
+	End	AB +D*EFAD*+/+C
Empty	End	AB +D*EFAD*+/+C+

(ii) A * (B + C) / D – G

Stack	Input	Output
Empty	A * (B + C) / D – G	–
Empty	* (B + C) / D – G	A

Stack	Input	Output
*	(B + C) / D – G	A
*(	B + C) / D – G	A
*(	+ C) / D – G	AB
*(+	C) / D – G	AB
*(+	) / D – G	ABC
*	/ D – G	ABC+
/	D – G	ABC+*
/	– G	ABC+*D
–	G	ABC+*D/
–	End	ABC+*D/G
Empty	End	ABC+*D/G–

Example 3.6.8 : Trace the conversion of infix to postfix form in tabular form. (A + B * C/D – E + F / G / (H + I))

Solution :

Sr. No.	Stack	Input	Output (Postfix)
1.	Empty	(A + B * C/D – E + F / G / (H + I))	–
2.	(	A + B * C/D – E + F / G / (H + I))	–
3.	(	+ B * C/D – E + F / G / (H + I))	A
4.	(+	B * C/D – E + F / G / (H + I))	A
5.	(+	* C/D – E + F / G / (H + I))	AB
6.	(+*	C/D – E + F / G / (H + I))	AB
7.	(+*	/D – E + F / G / (H + I))	ABC
8.	(+/	D – E + F / G / (H + I))	ABC*
9.	(+/	– E + F / G / (H + I))	ABC*D
10.	(–	E + F / G / (H + I))	ABC*D / +
11.	(–	+ F / G / (H + I))	ABC*D / + E
12.	(+	F / G / (H + I))	ABC*D / + E–
13.	(+	/ G / (H + I))	ABC*D / + E – F
14.	(+/	G / (H + I))	ABC*D / + E – F
15.	(+/	/ (H + I))	ABC*D / + E – FG
16.	(+/	(H + I))	ABC*D / + E – FG /
17.	(+/(	H + I))	ABC*D / + E – FG /
18.	(+/(	+ I))	ABC*D / + E – FG / H
19.	(+/(+	I))	ABC*D / + E – FG / H
20.	(+/(+	))	ABC*D / + E – FG / HI
21.	(+/	)	ABC*D / + E – FG / HI +
22.	Empty	End	ABC*D / + E – FG / HI + / +

Postfix form : ABC*D / + E – FG / HI + / +

Example 3.6.9 : Convert following expression Q into postfix form. Q = (A + B) * C – D/E * (F/G). Show stack representation. **S-10**

Solution :

Sr. No.	Stack	Input	Output
1.	Empty	(A + B) * C – D/E*(F/G)	–
2.	(	A + B) *C – D/E*(F/G)	–
3.	(	+ B) * C – D/E * (F/G)	A
4.	(+	B) * C – D/E *(F/G)	A
5.	(+	) * C – D/E * (F/G)	AB
6.	Empty	* C – D/E * (F/G)	AB+
7.	*	C – D/E * (F/G)	AB+
8.	*	– D/E * (F/G)	AB+C
9.	–	D/E * (F/G)	AB+C*

Sr. No.	Stack	Input	Output
10.	–	/E * (F/G)	AB+C*D
11.	–/	E * (F/G)	AB+C*D
12.	–/	*(F/G)	AB+C*DE
13.	–*	(F/G)	AB+C*DE/
14.	–*(	F/G)	AB+C*DE/
15.	–*(	/G)	AB+C*DE/F
16.	–*(/	G)	AB+C*DE/F
17.	–*(/	)	AB+C*DE/FG
18.	–*	End	AB+C*DE/FG/
19.	Empty	End	AB+C*DE/FG/*–

∴ Postfix expression = AB + C * DE/ FG /*–

Example 3.6.10 : Translate the given infix expression to postfix expression using stack and show the details of stack at each step. Expression : ((A + B) * D)↑ (E – F) **S-09, S-13, W-14**

Solution :

Sr. No.	Stack	Input	Output
1.	Empty	((A + B) * D) ↑ (E –F)	–
2.	(	(A + B) * D) ↑ (E –F)	–
3.	((	A + B) * D) ↑ (E –F)	–
4.	((	+ B) * D) ↑ (E –F)	A
5.	((+	B) * D) ↑ (E –F)	A
6.	((+	) * D) ↑ (E –F)	AB
7.	(	* D) ↑ (E –F)	AB+
8.	(*	D) ↑ (E –F)	AB+
9.	(*	) ↑ (E –F)	AB+D
10.	Empty	↑ (E –F)	AB+D*
11.	↑	(E –F)	AB+D*
12.	↑(	E –F)	AB+D*E
13.	↑(	–F)	AB+D*E
14.	↑(–	F)	AB+D*E
15.	↑(–	)	AB+D*EF
16.	↑	End	AB+D*EF–
17.	Empty	End	AB+D*EF–↑

∴ Postfix expression = AB+D*EF–↑

Example 3.6.11 : Convert the given infix expression to post fix expression using stack and also show the details of stack at each step of conversion expression :

(a * b + c/d) * (e + f ↑ g) **S-12**

Solution :

Sr. No.	Stack	Given expression (infix)	Result (postfix)
1.	Empty	(a*b + c/d) * (e + f ↑ g)	–
2.	(	a*b + c/d * (e + f ↑ g)	–
3.	(	*b + c/d) * (e + f ↑ g)	a
4.	(*	b + c/d) * (e + f ↑ g)	a
5.	(*	+ c/d) * (e + f ↑ g)	ab
6.	(+	c/d) * (e + f ↑ g)	ab*
7.	(+	/d) * (e + f ↑ g)	ab*c
8.	(+/	d) * (e + f ↑ g)	ab*c
9.	(+/	) * (e + f ↑ g)	ab*cd
10.	Empty	* (e + f ↑ g)	ab*cd/+
11.	*	(e + f ↑ g)	ab * cd/+
12.	*(	e + f ↑ g)	ab * cd/+
13.	*(	+ f ↑ g)	ab * cd/+e
14.	*(+	f ↑ g)	ab * cd/+e

Sr. No.	Stack	Given expression (infix)	Result (postfix)
15.	*(+	↑ g)	ab * cd/+ef
16.	*(+↑	g)	ab * cd/+ef
17.	*(+↑	)	ab * cd/+efg
18.	*		ab * cd/+efg ↑ +
19.	Empty	End	ab * cd/+efg ↑ +*

∴ Postfix expression = ab*cd/+ efg↑ + *

Example 3.6.12 : Convert Infix string into prefix string with stack operations show the content of stack in each step? **S-11**

Solution :

It is a fully parenthesized expression. It can easily be converted into its equivalent prefix form by moving each operator to replace its corresponding left parentheses. This can be done with the help of a stack by scanning the infix expression from right to left.

Stack	Infix expression	Output (Prefix expression)
Empty	((A + B)* (C – D))	–
Empty	((A + B)* (C – D)	–
Empty	((A + B)* (C –	D
–	((A + B)* (C	D
–	((A + B)* (	CD
Empty	((A + B)*	– CD
*	((A + B)	– CD
*	((A + B	– CD
*	((A +	B – CD
*+	((A	B – CD
*+	((	AB – CD
*	(	+AB – CD
Empty	End	*+ AB – CD

Example 3.6.13 : Convert the given infix expression to postfix expression using stack and the details of stack at each step of conversion. Expression : a ↑ b * c – d + e / f / (g + h)Note : ↑ indicates exponent operator. **W -12, S-15**

Solution :

Sr. No.	Stack	Given expression (Infix)	Result (Postfix)
1.	Empty	a↑ b*c – d + e / f / (g + h)	–
2.	–	↑ b*c – d + e / f / (g + h)	a
3.	↑	b *c – d + e / f / (g + h)	a
4.	↑	* c – d + e / f / (g + h)	ab
5.	↑	c – d + e / f / (g +h)	ab
6.	*	– d + e / f / (g + h)	ab ↑c
7.	–	d + e / f / (g + h)	ab ↑ c*
8.	–	+ e / f / (g + h)	ab ↑ c * d
9.	+	e / f / (g + h)	ab ↑ c * d –
10.	+	/ f / (g + h)	ab ↑ c * d – e
11.	+ /	f / (g + h)	ab ↑ c * d – e
12.	+ /	/ (g + h)	ab ↑ c * d – ef
13.	+/	(g + h)	ab ↑ c * d – ef /
14.	+ / (	g + h)	ab ↑ c * d – ef /
15.	+ / (	+ h)	ab ↑ c * d – ef / g
16.	+ / (+	h)	ab ↑ c * d – ef / g
17.	+ / (+	)	ab ↑ c * d – ef / gh
18.	+ /	End	ab ↑ c * d – ef /gh +
19.	Empty	End	ab ↑ c * d – ef/ gh + / +

∴ Postfix expression is = ab ↑ c * d – ef / gh + /+

Example 3.6.14 : Convert following expression into postfix exponent indicates operat or form with illustration of all steps.

A + B ↑ C * (D/E) – F% G **S-17**

Solution :

Sr. No.	Stack	Infix Expression	Postfix Expression
1.	Empty	A + B ↑ C * (D/E) – F% G	–
2.	Empty	+ B ↑ C * (D/E) – F% G	A
3.	+	B ↑ C * (D/E) – F% G	A
4.	+	↑ C * (D/E) – F% G	AB
5.	+ ↑	C * (D/E) – F% G	AB
6.	+ ↑	* (D/E) – F% G	ABC
7.	+ *	(D/E) – F% G	ABC↑
8.	+ * (	D/E) – F% G	ABC↑
9.	+ * (	/E) – F% G	ABC↑D
10.	+ * (/	E) – F% G	ABC↑D
11.	+ * (/	) – F% G	ABC↑DE
12.	+ *	– F% G	ABC↑DE/
13.	–	F% G	ABC↑DE/*+
14.	–	% G	ABC↑DE/*+F
15.	– %	G	ABC↑DE/*+F
16.	– %	End	ABC↑DE/*+FG
17.	Empty	End	ABC↑DE/*+FG%–

∴ Result = ABC↑DE/*+FG%–

Example 3.6.15 : Convert following infix expression into a postfix expression. Show all steps.

(P + (Q * R – (S/T↑U) * V) * W) **W-17**

Solution :

Sr. No.	Stack	Infix Expression	Postfix Expression
1.	Empty	(P + (Q * R – (S/T ↑ U) * V) * W)	–
2.	(	P + (Q * R – (S/T ↑ U) * V) * W)	–
3.	(	+ (Q * R – (S/T ↑ U) * V) * W)	P
4.	(+	(Q * R – (S/T ↑ U) * V) * W)	P
5.	(+ (	Q * R – (S/T ↑ U) * V) * W)	P
6.	(+ (	* R – (S/T ↑ U) * V) * W)	PQ
7.	(+ (*	R – (S/T ↑ U) * V) * W)	PQ
8.	(+ (*	– (S/T ↑ U) * V) * W)	PQR
9.	(+ (–	(S/T ↑ U) * V) * W)	PQR*
10.	(+ (– (	S/T ↑ U) * V) * W)	PQR*
11.	(+ (– (	/T ↑ U) * V) * W)	PQR*S
12.	(+ (– (/	T ↑ U) * V) * W)	PQR*S
13.	(+ (– (/	↑ U) * V) * W)	PQR*ST
14.	(+ (– (/↑	U) * V) * W)	PQR*ST
15.	(+ (– (/↑	) * V) * W)	PQR*STU
16.	(+ (–	* V) * W)	PQR*STU↑/
17.	(+ (– *	V) * W)	PQR*STU↑/
18.	(+ (– *	) * W)	PQR*STU↑/V
19.	(+	* W)	PQR*STU↑/V*–
20	(+*	W)	PQR*STU↑/V*–
21	(+*	)	PQR*STU↑/V*–W
22	Empty	End	PQR*STU↑/V*–W*+

Example 3.6.16 : Write an algorithm to convert infix string to postfix (fully parenthesized). **W-15**

Solution :

Algorithm to convert infix string to postfix :

This can be done by moving each operator to replace its corresponding closing parenthesis. This can be done with the help of a stack.

```
S: stack
while (more token)
{
        x ← next token;
        if(x is an operand)
            print x
        else
            if (x is an operator)
                push (s, x);
        else
            if (x is a closing parenthesis)
                print (pop(s))
}
```

- **Program 3.6.3 : Program for conversion of infix to its postfix form operators supported '+,-,*,/,%,^,(,) operands supported -- all single character operands**

```
/* program for conversion of  infix into its postfix form
   operators supported '+,-,*,/,%,^,(,)
   operands supported -- all single character operands */

#include<stdio.h>
#include<conio.h>
#include<ctype.h>
#define MAX 50
typedef struct stack
{
    int data[MAX];
    int top;
}stack;
int  precedence(char);
void init(stack *);
int  empty(stack *);
int  full(stack *);
int  pop(stack *);
void push(stack *,int );
int  top(stack *); //value of the top element
void infix_to_postfix(char infix[],char postfix[]);
void main()
 { char infix[30],postfix[30];
   clrscr();
   printf("\nEnter an infix expression : ");
   gets(infix);
   infix_to_postfix(infix,postfix);
   printf("\nPostfix : %s  ",postfix);
   getch();
 }
void infix_to_postfix(char infix[],char postfix[])
{   stack s;
    char x;
    int i,j;//i-index for infix[],j-index for postfix
    char token;
    init(&s);
    j=0;
    for(i=0;infix[i]!='\0';i++)
     {    token=infix[i];
       if(isalnum(token))
           postfix[j++]=token;
       else
           if(token == '(')
               push(&s,'(');
           else
               if(token == ')')
                 while((x=pop(&s))!='(')
                   postfix[j++]=x;
               else
               {
    while(precedence(token)<=precedence(top(&s)) &&
!empty(&s))
                   {
                       x=pop(&s);
```

```
                postfix[j++]=x;
            }
            push(&s,token);
        }
    }
    while(!empty(&s))
    {
        x=pop(&s);
        postfix[j++]=x;
    }
postfix[j]='\0';
}
int precedence(char x)
{
    if(x == '(')                    return(0);
    if(x == '+' || x == '-')        return(1);
    if(x == '*' || x == '/' || x == '%') return(2);
    return(3);
}
void init(stack *s)
{
    s→ top=-1;
}
int empty(stack *s)
{
    if(s→ top==-1)  return(1);
    return(0);
}
int full(stack *s)
{
    if(s→ top==MAX-1)   return(1);
    return(0);
}
void push(stack *s,int x)
{
    s→ top=s→ top+1;
    s→ data[s→ top]=x;
}
int pop(stack *s)
{
    int x;
    x=s→ data[s→ top];
    s→ top=s→ top-1;
    return(x);
}
int top(stack * p)
{
    return(p→ data[p→ top]);
}
```

Output

```
Enter infix expression: a*(b+c)/d+g
abc+*d/g+
```

Syllabus Topic : Conversion an Infix into Prefix Expression

3.6.5 Conversion of an Expression from Infix to Prefix

➔ **(MSBTE : S-08, S-09)**

Example of a infix expression : (A + B ∧ C) * D + E ∧ 5 where A, B, C, D and E are integer constant and B ∧ C stands for B^C.

☞ **Manual method**

In manual evaluation of the above expression, B^C must be calculated first. Hence, we convert B^C to its equivalent pretfix equivalent i.e. ^BC.

(A + B^C) * D + E^5 – original expression

(A + ^BC) * D + E^5 – ^BC should be treated as a single operand.

+ A^BC * D + E^5 – A + ^BC is the next operation.

* + A^BCD + E^5 – + A^BC * D is the next operation.

* + A^ BCD + ^E5 – E^5 is the next operation.

+ * + A^BCD^E5 – Addition is the last operation.

Algorithmic approach

An infix expression can be converted into prefix form with the help of following steps :

1. Reverse the infix expression
2. Make every '(' (opening bracket) as ')' (closing bracket) and every ')' as '('
3. Convert the modified expression to postfix form. Use the algorithm given in the Section 3.6.4.
4. Reverse the postfix expression.

Above algorithm can be explained by carrying out the steps from 1 to 4 on the example :

(A + B ^ C) * D + E ^ 5

Step 1 : Reverse the infix expression.

5 ^ E + D *) C ^ B + A(

Step 2 : Make every '(' as ')' and every ')' and '('

5 ^ E + D * (C ^ B + A)

Step 3 : Convert the expression to postfix form :

Expression (input)	Stack	Output	Comment
5 ^ E + D * (C ^ B + A)	Empty	–	Initially
^ E + D * (C ^ B + A)	Empty	5	Print
E + D * (C ^ B + A)	^	5	Push
+ D * (C ^ B + A)	^	5 E	Push
D * (C ^ B + A)	+	5 E ^	Pop and Push
* (C ^ B + A)	+	5 E ^ D	Print
(C ^ B + A)	+*	5 E ^ D	Push
C ^ B + A)	+*(	5 E ^ D	Push
^ B + A)	+*(	5 E ^ D C	Print
B + A)	+*(^	5 E ^ D C	Push
+ A)	+*(^	5 E ^ D C B	Print
A)	+*(+	5 E ^ D C B ^	Pop and Push
)	+*(+	5 E ^ D C B ^ A	Print
End	+*	5 E ^ D C B ^ A +	Pop until '('
End	Empty	5 E ^ D C B ^ A + * +	Pop everything

Step 4 : Reverse the expression

+ * + A ^ B C D ^ E 5

Pseudo code for conversion of an expression from infix into prefix.

```
{
   S : stack of characters ;
   infix ← read the infix expression;
   infix ← reverse the infix expression ;
   infix ← reverse the direction of brackets ;
   prefix ← A NULL string
   i = 0 ;
   while ( infix [i] is not equal to '\0)
      {  x ← infix [i] ;
         if ( x is an operand )
            add the character x at the end of prefix ;
      else
         if ( x is '(' )
            push ( S, '(')
         else
            if ( x is ')' )
            {  while ( ( x ← pop(S)) ! = '(' )
                  prefix ← prefix + pop(S) ;
                        //+ is for concatenation
            }
             else
            {  while ( precedence (x) < precedence (top(S))
                  prefix ← prefix + pop(S ) ;
                  //+ is for concatenation
               push (S, X) ;
            }
```

```
        i ← i + 1 ;
    }
        while ( !empty (S) )
            prefix ← prefix + pop(S) ;
        prefix ← reverse the prefix ;
        print prefix ;
  }
```

➢ **Program 3.6.4 : To convert infix expression to prefix expression.**

```
/* program for conversion of infix into its prefix form
  operators supported '+,-,*,/,%,^,(,)
  operands supported -- all single character operands
*/
#include<stdio.h>
#include<conio.h>
#include<ctype.h>
#define MAX 50

typedef struct stack
{
    int data[MAX];
    int top;
}stack;
int  precedence(char);
void init(stack *);
int  empty(stack *);
int  full(stack *);
int  pop(stack *);
void push(stack *,int );
int  top(stack *); //value of the top element
void infix_to_prefix(char infix[],char prefix[]);
void infix_to_postfix(char infix[],char postfix[]);

void main()
{ char infix[30],prefix[30];
  clrscr();
  printf("\nEnter an infix expression : ");
  gets(infix);
  infix_to_prefix(infix,prefix);
  printf("\n prefix: %s  ",prefix);
  getch();
}
void infix_to_prefix(char infix[],char prefix[])
 {  int i,j;
   char temp,in1[30];
    // reverse the infix expression  and store it in in1[]
   for(i=strlen(infix)-1,j=0;i>=0;i--,j++)
    in1[j]=infix[i];
   in1[j]='\0';
   // reverse the direction of brackets
   for(i=0;in1[i]!='\0';i++)
    {
    if(in1[i]=='(')
        in1[i]=')';
    else
        if(in1[i]==')')
         in1[i]='(';
    }
   // convert from infix to postfix
   infix_to_postfix(in1,prefix);
   //reverse the final expression
  for(i=0,j=strlen(prefix)-1;i<j;i++,j--)
   {
   temp=prefix[i];
   prefix[i]=prefix[j];
   prefix[j]=temp;
   }
 }
```

```
void infix_to_postfix(char infix[],char postfix[])
{   stack s;
    char x;
    int i,j;//i-index for infix[],j-index for postfix
    char token;
    init(&s);
    j=0;
    for(i=0;infix[i]!='\0';i++)
     {    token=infix[i];
        if(isalnum(token))
            postfix[j++]=token;
        else
            if(token == '(')
                push(&s,'(');
            else
                if(token == ')')
                  while((x=pop(&s))!='(')
                    postfix[j++]=x;
                else
                {
    while(precedence(token)<precedence(top(&s)) &&
!empty(&s))
                    {
                        x=pop(&s);
                        postfix[j++]=x;
                    }
                    push(&s,token);
                }
        }
    while(!empty(&s))
{
        x=pop(&s);
        postfix[j++]=x;
    }
postfix[j]='\0';
}
int precedence(char x)
{
    if(x == '(')                    return(0);
    if(x == '+' || x == '-')            return(1);
    if(x == '*' || x == '/' || x == '%') return(2);
    return(3);
}
void init(stack *s)
{
    s→ top=-1;
}
int empty(stack *s)
{
    if(s→ top==-1)  return(1);
    return(0);
}
int full(stack *s)
{
    if(s→ top==MAX-1)    return(1);
    return(0);
}
void push(stack *s,int x)
{
    s→ top=s→ top+1;
    s→ data[s→ top]=x;
}
int pop(stack *s)
{
    int x;
    x=s→ data[s→ top];
    s→ top=s→ top-1;
    return(x);
```

```
}
int top(stack * p)
{
    return(p→ data[p→ top]);
}
```

3.7 Expression Conversion (A Fast Method)

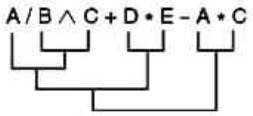
3.7.1 Infix to Postfix

☞ **1st method**

Example considered for conversion is :

A / B^C + D*E – A*C

– Operands are grouped as shown below. Operator precedence and associativity rules must be followed while grouping.



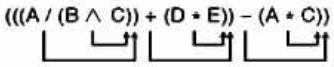
– Operators are moved at the end of the group.

Postfix expression of the given infix expression = ABC^ / DE* + AC* –

☞ **2nd method**

An infix expression can be converted into postfix form by fully parenthesizing the infix expression and then moving operators to replace their corresponding right parentheses.

Given infix expression : A/B^C+D*E – A*C. After fully parenthesizing the expression, we get :



After moving operators to replace their corresponding right parentheses, we get :

ABC^ / DE* + AC * –

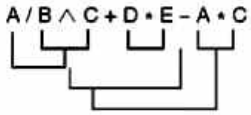
3.7.2 Infix to Prefix

☞ **1st method**

Example considered for conversion is :

A/B^C + D*E – A*C

– Operators are grouped as shown below. Operator precedence and associativity rules must be followed while grouping.



– Operators are moved at the beginning of the group.

Prefix expression of the given infix expression

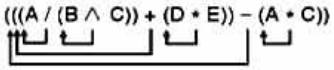
= – + /A^BC*DE*AC

☞ **2nd method**

An infix expression can be converted into prefix form by fully parenthesizing the infix expression and then moving operators to replace their corresponding left parentheses.

Given infix expression : A/B^C + D*E – A*C

After fully parenthesizing the expression, we get :



After moving operators to replace their corresponding left parentheses, we get :

+ A^BC * DE*AC

Example 3.7.1 : Convert the given infix string to prefix expression and show the details of stack at each step.(A – B/C) * (D * E – F) W-13

Solution :

Step 1 : Reversing the expression, we get

(F – E * D) * (C/B – A)

Step 2 : Converting to postfix

Sr. No.	Stack	Expression	Output
1.	Empty	(F – E * D) * (C/B – A)	–
2.	(	F – E * D) * (C/B – A)	–
3.	(	– E * D) * (C/B – A)	F
4.	(–	E * D) * (C/B – A)	F
5.	(–	* D) * (C/B – A)	FE
6.	(– *	D) * (C/B – A)	FE

Sr. No.	Stack	Expression	Output
7.	(– *	) * (C/B – A)	FED
8.	Empty	* (C/B – A)	FED* –
9.	*	(C/B – A)	FED* –
10.	* (	C/B – A)	FED * –
11.	* (	/B – A)	FED * – C
12.	* (/	B – A)	FED * – C
13.	* (/	– A)	FED * – CB
14.	* (–	A)	FED * – CB/
15.	* (–	)	FED * – CB/A
16.	*	End	FED * – CB/A –
17.	Empty	End	FED * – CB/A – *

Step 3 : Reversing the expression :

Prefix expression : * – A/BC – * DEF

3.7.3 Postfix to Prefix

Postfix expression

ABC^ /DE* + AC* –

Group the tokens as shown below

- Grouping is done left to right
- All operators are moved at the beginning of the group.

– + / A ^ BC * DE * AC

☞ **Postfix to Infix**

- All operators are moved between the two operands of the group

A / B^C + D * E – A * C

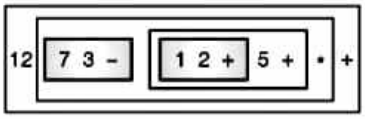
Example 3.7.2 : **Consider the following arithmetic expression P, written in postfix notation :P : 12, 7, 3, –, 1, 2, +, 5, +, *, +**
(i) Translate P, into its equivalent infix expression.
(ii) Evaluate the infix expression.
W-08

Solution :

(i) Postfix to infix form

Postfix expression (P) = 12, 7, 3, –, 1, 2, +, 5, +, *, +

Step 1 : Grouping of tokens as per the sequence of evaluation.



Step 2 : Operator are moved between the two operands of the group.

12 + (7 – 3) * (1 + 2 + 5)

(ii) Evaluate the infix expression

12 + (7 – 3) * (1 + 2 + 5) = 12 + 4 * 8 = 12 + 32 = 44

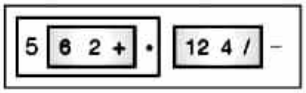
Example 3.7.3 : **Convert the following arithmetic expression P written in postfix notation into infix. P : 5, 6, 2, +, *, 12, 4, /,– Also evaluate P for final value.**
S-09, S-15

Solution :

(i) Postfix to infix form :

Postfix expression (P) = 5, 6, 2, +, *, 12, 4, /, -

Step 1 : Grouping of tokens as per the sequence of evaluation.



Step 2 : Operators are moved between the two operands of the group.

5 * (6 + 2) – 12/4

(ii) Evaluation of 5, 6, 2, +, *, 12, 4, /,–

Sr. No.	Input	Stack	Initially
1.	5, 6, 2, +, *, 12, 4, /, –		

Sr. No.	Input	Stack	Initially
2.	6, 2, +, *, 12, 4, /, –	5	
3.	2, +, *, 12, 4, /, –	6 5	
4.	+, *, 12, 4, /, –	2 6 5	
5.	*, 12, 4, /, –	8 5	6 + 2 = 8
6.	12, 4, /, –	40	5 * 8 = 40
7.	4, /, –	12 40	
8.	/, –	4 12 40	
9.	–	3 40	12/4 = 3
10.	End	37	40 – 3 = 37

∴ Result = 37

Example 3.7.4 : Consider the following arithmetic expression P written in postfix notation : P : 4, 2, ↑, 3 * 3, –, 8, 4, /, 1, 1, +, /, +.Evaluate this expression to find its value.Note : ↑ indicates exponent operator. **W-12**

Solution :

Sr. No.	Input	Stack	
1.	4, 2, ↑, 3, * 3, –, 8, 4, /, 1, 1, +, /, +		
2.	2, ↑, 3, * 3, –, 8, 4, /, 1, 1, +, /, +	4	
3.	↑, 3, * 3, –, 8, 4, /, 1, 1, +, /, +	2 4	
4.	3, * 3, –, 8, 4, /, 1, 1, +, /, +	16	4 ↑ 2 = 16
5.	*, 3, –, 8, 4, /, 1, 1, +, /, +	3 16	
6.	3, –, 8, 4, /, 1, 1, +, /, +	48	16 * 3 = 48
7.	–, 8, 4, /, 1, 1, +, /, +	3 48	
8.	8, 4, /, 1, 1, +, /, +	45	48 – 3 = 45
9.	4, /, 1, 1, +, /, +	8 45	
10.	/, 1, 1, +, /, +	4 8 45	
11.	1, 1, +, /, +	2 45	8/4 = 2

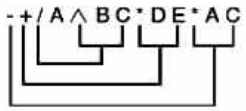
Sr. No.	Input	Stack	
12.	1, +, /, +	1 2 45	
13.	+, /, +	1 1 2 45	
14.	/, +	2 2 45	1 + 1 = 2
15.	+	1 45	2/2 = 1
16	End	46	45 + 1 = 46

∴ Result = 46

3.7.4 Prefix to Infix

Prefix expression – + / A^BC * DE * AC

Group the tokens as shown below



Grouping is done right to left

All operators are moved between the two operands of the group

A / B^C + D *E – A * C

☞ **Prefix to postfix**

All operators are moved at the end of the group

ABC^/ DE * + AC * /

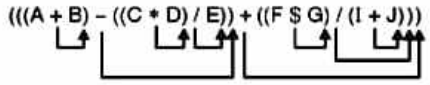
Example 3.7.5 : Transform the expression to postfix and evaluate postfix expression by assuming A = 1, B = 2, C = 3, D = 4, E = 6, F = 6, G = 1, I = 3 and J = 3.

A + B – C * D / E + F $ G / (I + J)

Solution :

(i) Infix to postfix conversion

1. Fully parenthesizing the expression, we get :



2. Moving operators to replace their corresponding right parenthesis.

AB + CD * E / – FG $ IJ + / +

(ii) Evaluation of postfix expression

Sr. No.	Input	Stack	
1.	AB + CD * E / – FG $ IJ + / +		
2.	B + CD * E / – FG $ IJ + / +	1	
3	+ CD * E / – FG $ IJ + / +	2 1	
4.	CD * E / – FG $ IJ + / +	3	1 + 2 = 3
5.	D * E / – FG $ IJ + / +	3 3	
6.	* E / – FG $ IJ + / +	4 3 3	
7.	E / – FG $ IJ + / +	12 3	3 * 4 = 12

Sr. No.	Input	Stack	
8.	/ – FG $ IJ + / +	6 12 3	
9.	– FG $ IJ + / +	2 3	12/6 = 2
10.	FG $ IJ + / +	1	3 – 2 = 1
11.	G $ IJ + / +	6 1	
12.	$ IJ + / +	1 6 1	
13.	IJ + / +	6 1	6′ = 6
14.	J + / +	3 6 1	
15.	+ / +	3 3 6 1	
16.	/ +	6 6 1	3 + 3 = 6
17.	+	1 1	6/6 = 1
18.	End	2	

Value = 2

Example 3.7.6 : Convert infix string ((A + B) * (C – D)) / (E + F) into prefix string with stack. Show the content of stack in each step. **W-15**

Solution :

Given infix string : ((A + B) * (C – D)) / (E + F)

Step 1 : Reversing the string, we get

(F + E)/((D – C)* (B + A))

Step 2 : Converting the expression obtained in step 1 in postfix form

Stack	Expression	Postfix form
Empty	(F + E)/((D – C)* (B + A))	–
(	F + E)/((D – C)* (B + A))	–
(	+ E)/((D – C)* (B + A))	F
(+	E)/((D – C)* (B + A))	F
(+	)/((D – C)* (B + A))	FE
Empty	/((D – C)* (B + A))	FE+
/	((D – C)* (B + A))	FE+
/(	(D – C)* (B + A))	FE+
/((	D – C)* (B + A))	FE+
/((	– C)* (B + A))	FE+D
/((–	C)* (B + A))	FE+D
/((–	)* (B + A))	FE+DC
/(	* (B + A))	FE+DC–
/(*	(B + A))	FE+DC–

Stack	Expression	Postfix form
/(*(	B + A))	FE+DC–
/(*(	+ A))	FE+DC–B
/(*(+	A))	FE+DC–B
/(*(+	))	FE+DC–BA
/(*	)	FE+DC-BA+
/	-	FE+DC-BA+*
Empty	-	FE+DC-BA+*/

Step 3 : Reversing the expression obtained in step 2 to get the required prefix expression.

/* + AB – CD + EF

Example 3.7.7 : **Consider the following arithmetic expression written in postfix notation :**

10, 2, *, 15, 3, /, +, 12, 3, 2, ↑, +, . Evaluate this expression to find its value. **W-15**

Solution :

Given : 10, 2, *, 15, 3, /, +, 12, 3, 2, ↑, +, +

= 20, 15, 3, /, +, 12, 3, 2, ↑, +, + [10 * 2 = 20]

= 20, 5, +, 12, 3, 2, ↑, +, + [15/3 = 5]

= 25, 12, 3, 2, ↑, +, + [20 + 5 = 25]

= 25, 12, 9, +, + [$3^2 = 9$]

= 25, 21, + [12 + 9 = 21]

= 46 [25 + 21 = 46]

Example 3.7.8 : **Translate the following infix expression to its equivalent prefix expression,**

(x + y) * (p – q).

Evaluate the above prefix expression with following values :

x = 2, y = 3,p = 6, q = 2. **S-16**

Solution :

Step 1 : Reversing the expression, we get,

(q – p) * (y + x)

Step 2 : Postfix equivalence of the reversed expression

Sr. No.	Stack	Input	Postfix equivalent
1.	Empty	(q – p) * (y + x)	–
2.	(	q – p) * (y + x)	–
3.	(	– p) * (y + x)	q
4.	(–	p) * (y + x)	q
5.	(–	) * (y + x)	qp
6.	Empty	* (y + x)	qp–
7.	*	(y + x)	qp–
8.	*(	y + x)	qp–
9.	*(	+ x)	qp – y
10.	*(+	x)	qp – y
11.	*(+	)	qp – yx
12.	*	–	qp – yx+
13.	Empty	–	qp – yx + *

Step 3 : Reversing the expression, we get the required prefix expression.

Prefix expression ⇒ *+ xy – pq

Step 4 : Evaluation of prefix expression

Sr. No.	Prefix expression	Stack	
1.	* + xy – pq		
2.	* + xy – p	2	q = 2
3.	* + xy –	6 2	p = 6
4.	* + xy	4	6 – 2 = 4
5.	* + x	3 4	y = 3

Sr. No.	Prefix expression	Stack	
6.	* +	2 3 4	x = 2
7.	*	5 4	3 + 2 = 5
8.	–	20	

∴ **Result = 20**

Example 3.7.9 : Find out prefix equivalent of the following expression :

i) [(A + B) + C] * D

ii) A [(B* C) + D] **W-16**

Solution :

(i) After fully parenthesizing the expression we get :

(((A + B) + C) * D)

After moving operators to replace their corresponding left parenthesis we get,

Prefix : *++ ABCD

(ii) After fully parenthesizing the expression we get

(A * ((B * C) + D))

After moving operators to replace their corresponding left parenthesis we get,

Prefix : * A + * B C D

Example 3.7.10 : Convert the given infix expression to postfix expression using stack and the details of stack at each step of conversion.

EXPRESSION : P * Q ↑ R – S/T + [U/V] **W-16**

Solution :

Sr. No.	Stack	Infix Expression	Postfix Expression
1.	Empty	P*Q↑R – S/T+[U/V]	–
2.	Empty	*Q↑R – S/T+[U/V]	P
3.	*	Q↑R – S/T + [U/V]	P
4.	*	↑R – S/T + [U/V]	PQ
5.	*↑	R – S/T + [U/V]	PQ
6.	*↑	– S/T + [U/V]	PQR
7.	–	S/T + [U/V]	PQR↑*
8.	–	/T + [U/V]	PQR↑*S
9.	–/	T + [U/V]	PQR↑*S
10.	–/	+ [U/V]	PQR↑*ST
11.	+	[U/V]	PQR↑*ST/–
12.	+[	U/V]	PQR↑*ST/–
13.	+[	/V]	PQR↑*ST/–U
14.	+[/	V]	PQR↑*ST/–U
15.	+[/	]	PQR↑*ST/–UV
16.	+	End	PQR↑*ST/–UV/
17.	Empty	End	PQR↑*ST/–UV/+

∴ **Result = PQR↑*ST/–UV/+**

Syllabus Topic : Recursion

3.8 Introduction to Recursion

➔ **(MSBTE : W-09, S-11, W-14, S-15, W-16, S-17)**

Q. 3.8.1 What is recursion ? (Refer section 3.8) **W-09**

Q. 3.8.2 Define recursion?
(Refer section 3.8) **S-11, W-14, S-15, W-16**

Q. 3.8.3 How stack is used in recursion ? Describe with suitable example.
(Refer sections 3.8 and 3.8.3) **W-16**

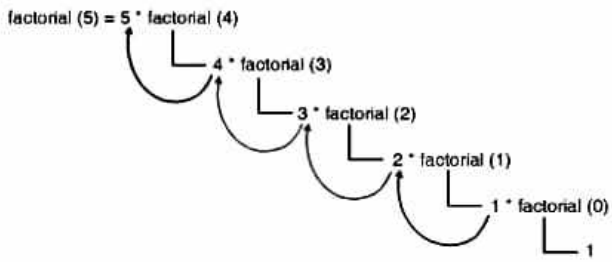
Q. 3.8.4 Recursion is one of the application of stack-YES/NO ? Explain it for calculating the factorial of a number 5 using recursion.
(Refer (Yes) section 3.8) **S-17**

Recursion is a fundamental concept in mathematics. When a function is defined in terms of itself then it is called a recursive function. Consider the definition of factorial of a positive integer n.

$$\text{Factorial}(n) = \begin{cases} 1 & \text{if } (n=0) \\ n * \text{factorial}(n-1), & \text{otherwise} \end{cases}$$

Function "factorial ( )" is defined in terms of itself for n > 0. Value of the function at n = 0 is 1 and it is called the base. Recursion terminates on reaching the base.

For example :



Recursion expands when n > 0 and it starts winding up on hitting the base (n = 0).

3.8.1 Converting a Recursive Function to an Equivalent C-function

3.8.1(A) Finding Factorial of an Integer Number

☞ **Recursive definition of factorial of an integer**

$$f(n) = 1 \quad \text{if } (n == 0)$$
$$n * f(n-1), \quad \text{otherwise}$$

➢ **'C' function for finding factorial.**

```
int factorial (int n)
{
    if (n = = 0)
        return (1);
    return (n * factorial (n – 1));
}
```

Sequence of calls to be made has been shown in the Fig. 3.8.1 to give a better insight into recursion.

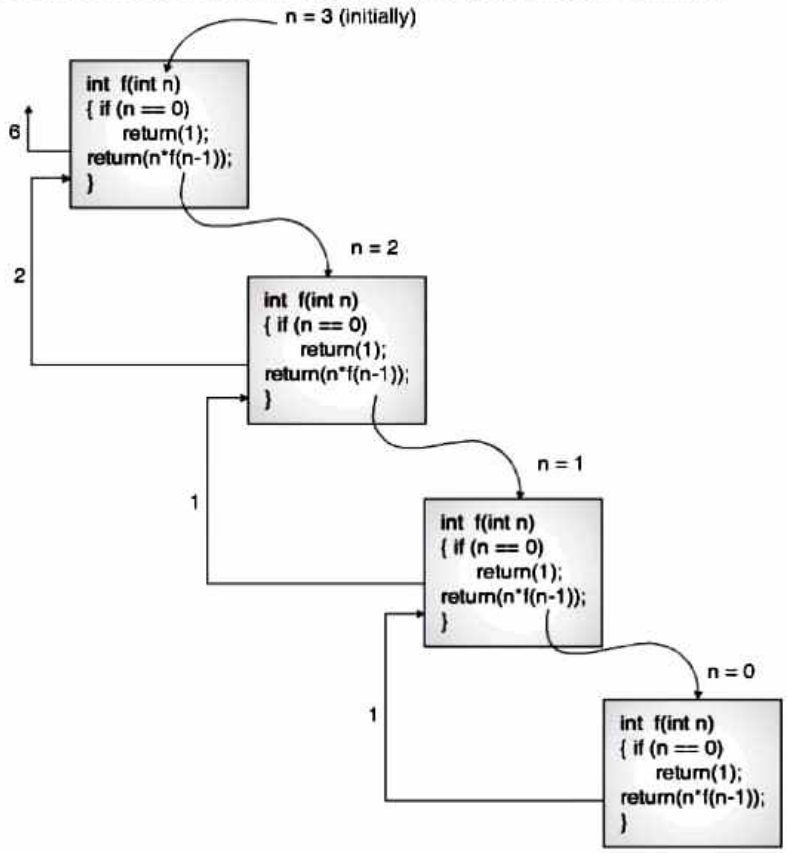


Fig. 3.8.1

Sequence of calls could also be shown through a recursion tree.

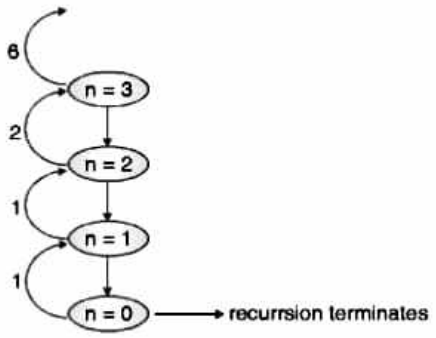


Fig. 3.8.2 : Recursion tree for finding factorial (3)

➢ **Program 3.8.1 : Program for finding factorial of a number through the use of recursive function.**

Q. 3.8.5 Write a 'C' program to calculate the factorial of number using recursion ?
(Refer section 3.8.1(A))
S-11, W-13, S-14, W-14

```
#include<stdio.h>
#include<conio.h>
int fact(int);
void main()
{
    int x,n;
    printf("\n Enter the value of n :");
    scanf("%d",&n);
    x=fact(n);
    printf("\n %d",x);
    getch();
}
int fact(int n)
{
    if(n==0)
        return(1);
    return(n*fact(n-1));
}
```

Output

```
Enter the value of n :5
120
```

☞ **Finding n^{th} term of Fibonacci sequence**

Recursive definition

$T_n = T_{n-1} + T_{n-2}$ if $n > 1$

$T_1 = 1$

$T_0 = 0$

➢ **'C' function for finding n^{th} term of a Fibonacci sequence.**

```
int fib (int n)
{
    if (n == 0)
        return (0);
    if (n == 1)
        return (1);
    return (fib(n – 1) + fib(n – 2));
}
```

In the above example fib(n) = fib(n – 1) + fib(n – 2) for n > 1 is a case of binary recursion. The first term on the right fib (n – 1) is known as left recursion and the second term on the right fib (n – 2) is known as right recursion.

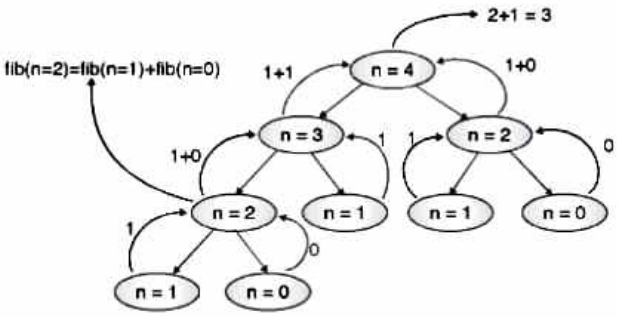


Fig. 3.8.3 : A recursion tree, showing calculation of T_4

☞ **Fibonacci series**

0	1	1	2	3
|	|	|	|	|
T_0	T_1	T_2	T_3	T_4

➢ **Program 3.8.2 : A C-program to generate Fibonacci series using recursion.**

Q. 3.8.6	Write a 'C/C++' program to generate Fibonacci series using recursion. (Refer section 3.8.1(A))	**W-08**
Q. 3.8.7	Write a 'C' program to display Fibonacci series using recursive function. (Refer Section 3.8.1(A))	**W-17**

```
#include<stdio.h>
#include<conio.h>
int fib(int);
void main()
  {
    int i, terms,val;
    printf("\n enter no. of terms:");
    scanf("%d",&terms);
    for (i = 0; i <terms; i++)
      {
        val=fib(i);
        printf("\n%d",val);
      }
  }
int fib(int n)
    {
      if(n==0)
        return(0);
      if(n==1)
        return(1);
      return(fib(n – 1) + fib(n – 2));
    }
```

Syllabus Topic : Tower of Hanoi

3.8.2 Tower of Hanoi Problem

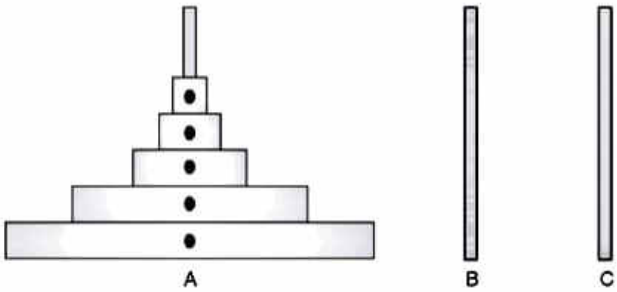


Fig. 3.8.4 : Disks are stacked on peg A in decreasing order of size

A tower of n disks is stacked in decreasing order on peg "A". These disks are to be transferred to peg "B" with the help of peg "C". Following rules must be followed while transferring disks from one peg to another peg.

(a) Only one disk can be transferred at a time.

(b) At no time a larger disk can be placed on a smaller disk.

In order to define the problem of Tower of Hanoi recursively, We must express n disks problem in terms of a problem of n – 1 disks.

☞ **n-disks problem**

Move n disks from peg A to peg B using peg C.

Step 1 : Move n – 1 disks from peg A to peg C.

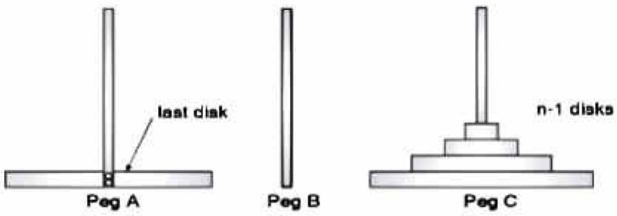


Fig. 3.8.5

Step 2 : Move the only disk from peg A to peg B.

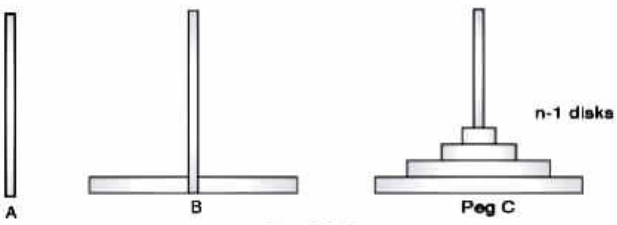


Fig. 3.8.6

Step 3 : Move the n – 1 disks from peg C to peg B.

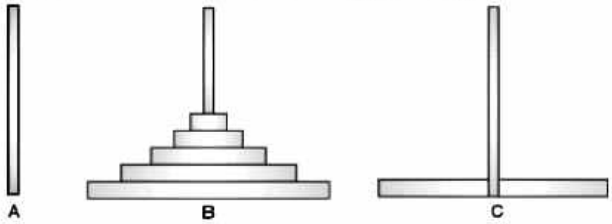


Fig. 3.8.7

Recursive definition

$$TOH(n, x, y, z) = \left\{ \begin{array}{l} TOH(n-1, x, z, y) \\ \text{move a disk from x to y} \\ TOH(n-1, z, y, x) \end{array} \right\} \text{ if } n \geq 1$$

n = Number of disks

x = Source peg

y = Destination peg

z = Empty peg to be used for transfer

TOH (n, x, y, z) = transfer n plates from peg x to peg y using peg z.

TOH (n – 1, x, z, y) = transfer n – 1 plates from peg x to peg z using peg y.

TOH (n – 1, z, y, x) = transfer n – 1 plates from z to y using peg x.

➢ **Program 3.8.3 : Program for tower of Hanoi.**

```
#include<stdio.h>
#include<conio.h>
void TOH(int n,char x,char y,char z);
void main()
{
    int n;
    printf("\n Enter number of plates:");
    scanf("%d",&n);
    TOH(n,'A','B','C');
    getch();
}
void TOH(int n,char x,char y,char z)
{
    if(n>0)
    {
        TOH(n-1,x,z,y);
        printf("\n%c -> %c",x,y);
        TOH(n-1,z,y,x);
    }
}
```

Output

```
Enter number of plates:3
A -> B
A -> C
B -> C
A -> B
C -> A
C -> B
A -> B
```

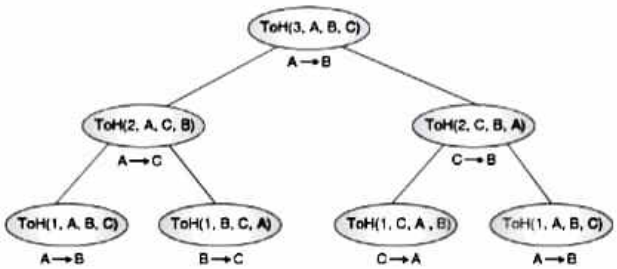


Fig. 3.8.8 : Recursion tree for the function TOH ( ) for n = 3 plates

Sequence of moves

(1) A → B (2) A → C (3) B → C

(4) A → B (5) C → A (6) C → B

(7) A → B

Sequence of moves can easily be written by making an inorder traversal on recursion tree. Concept of inorder traversal is explained in the chapter on "tree".

Example 3.8.1 : Write a C program for GCD using recursion.

Solution :

GCD of two numbers m and n can be defined as given below.

$$GCD(m, n) = \begin{cases} GCD(n, m) & \text{if } m < n \\ n & \text{if m is divisible by n} \\ GCD(n, m \% n) & \text{otherwise} \end{cases}$$

C-function for GCD

```
int GCD (int m, int n)
  {
      if (m < n)
          return (GCD (n, m));
      if (m%n ==0)
          return (n);
      return (GCD (n, m%n));
  }
```

➢ **Program 3.8.4 : Write a 'C' program for multiplication of natural numbers using recursion.**

Q. 3.8.8 Write a 'C' program for multiplication of natural numbers using recursion. (Refer section 3.8.2) **W-09, S-15**

```
#include<stdio.h>
#include<conio.h>
int multiply(int, int);
void main()
  { int x, y, result;
    printf("\n enter 2 numbers:");
    scanf("%d%d", &x, &y),
    result = multiply (x, y),
    printf("\n Product=%d", result);
    getch();
  }
  int multiply (int a, int b)
    { if(a==0||b==0)
          return(0);
      if(b==1)
          return(a);
      return(a + multiply(a, b – 1));
    }
```

3.8.3 Stack in Recursive Function

➔ **(MSBTE : W-12, W-16)**

Q. 3.8.9 Explain how stack is used in recursive function with example. (Refer section 3.8.3) **W-12**

Q. 3.8.10 Describe recursion related to stacks and give two examples. (Refer section 3.8.3) **W-16**

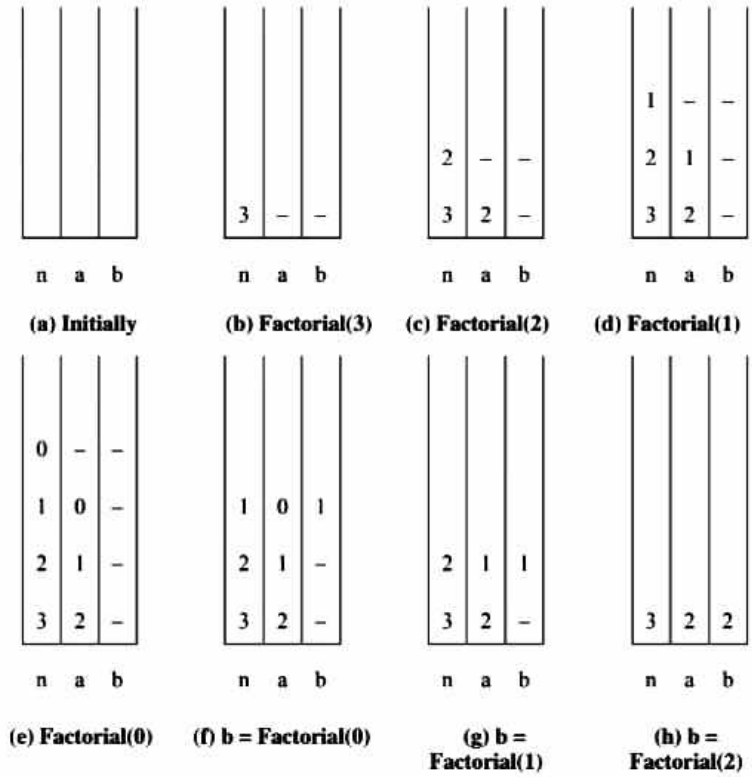
- A recursive function is characterized by that fact that it calls itself. In recursive calls many invocation of a function coexist during the execution of a program. A copy of the local variables of the function must be allocated for each invocation. These variables are saved on top of a stack.
- Each time a function is called, a new set of local variables and parameters is stored on top of a stack. Only these values are referenced within the function call.
- In case of return from the function, the most recent allocation of local variables and parameters are removed from the stack and the previous copy is re-activated.

☞ **Example**

Let us consider working of the recursive function for finding factorial of a given number. The recursive algorithm to compute n! in 'C' language is given below.

```
int factorial(int n)
  {
    int a,b;
    if(n==0)
        return(1);
    a = n–1;
    b = factorial(a);
    return (n * b);
  }
```

Private variables a, b and the parameter n is stored on top of the stack. The stack at various times during the execution of the program is shown below. Initial value of n is takes as 3.



Return value $= 3 \times 2 = 6$

3.9 MSBTE Questions and Answers

➔ Summer 2008 – Total Marks 10

Q. 1 Explain any two applications of stack.

(Sections 3.6, 3.6.2, 3.6.4 and 3.6.5) **(4 Marks)**

Q. 2 Write a menu driven 'C' program for implementing stack using array. **(Program 3.5.2)** **(6 Marks)**

➔ Winter 2008 – Total Marks 12

Q. 3 Define the terms 'overflow' and 'underflow' with respect to stacks.**(Section 3.5.1)** **(2 Marks)**

Q. 4 Write a 'C/C++' program to generate Fibonacci series using recursion. **(Program 3.8.2)** **(4 Marks)**

Q. 5 Consider the following arithmetic expression P, written in Postfix notation :

P : 12, 7, 3, –, 1, 2, 1, 5, +, *, +

(i) Translate P, into its equivalent Infix expression.

(ii) Evaluate the Infix expression.

(Example 3.7.2) **(6 Marks)**

➔ Summer 2009 – Total Marks 16

Q. 6 State the principle of stack with basic operations.

(Sections 3.1 and 3.3) **(2 Marks)**

Q. 7 Write down the applications of stacks.

(Sections 3.6, 3.6.2, 3.6.4 and 3.6.5) **(4 Marks)**

Q. 8 Translate the given Infix expression to postfix expression
using stack and show the details of stack at each step.

Expression : ((A + B) * D) ↑ (E – F)

(Example 3.6.10) **(6 Marks)**

Q. 9 Convert the following arithmetic expression P written in postfix notation into infix :

P : 5, 6, 2, + , *, 12, 4, /, –

Also evaluate P for final value.

(Example 3.7.3) **(4 Marks)**

➔ Winter 2009 – Total Marks 14

Q. 10 Evaluate following postfix expression.

A : 6, 2, 3, +, –, 3, 8, 2, 1, +, *, 2, ↑, 3, +.

(Example 3.6.4) **(6 Marks)**

Q. 11 What is recursion ? Write a 'C' program for multiplication of natural numbers using recursion. **(Section 3.8, Program 3.8.4)** **(4 Marks)**

Q. 12 Explain the terms overflow and underflow with respect to stack. Use suitable data and diagram. **(Section 3.5.1)** **(4 Marks)**

➔ Summer 2010 – Total Marks 14

Q. 13 Define stack. What are the basic operations performed on it ? **(Sections 3.1 and 3.3)** **(4 Marks)**

Q. 14 Convert following expression Q into postfix form.

Q = (A +. B) * C – D/E * (F/G). Show stack representation. **(Example 3.6.9)** **(6 Marks)**

Q. 15 Write a procedure to push an element on stack. Also give meaning of stack overflow term. **(Section 3.5.1)** **(4 Marks)**

➔ Winter 2010 – Total Marks 06

Q. 16 Consider the following arithmetic expression P written in postfix notation :

P : 5, 6, 2, +, *, 12, 4, /, –

Evaluate this expression to find its value.

(Example 3.6.1) **(6 Marks)**

➔ Summer 2011 – Total Marks 20

Q. 17 Explain four primitive operations on stack. **(Section 3.3)** **(4 Marks)**

Q. 18 Convert Infix string into prefix string with stack operations. Show the content of stack in each step? **(Example 3.6.12)** **(8 Marks)**

Q. 19 Define recursion? Write a 'C' program to calculate the factorial of number using recursion ?

(Section 3.8 and Program 3.8.1) **(8 Marks)**

➔ Winter 2011 – Total Marks 10

Q. 20 Discuss the polish notation of arithmetic expression in application of stacks. **(Section 3.6.1)** **(10 Marks)**

➔ Summer 2012 – Total Marks 18

Q. 21 Define the term stack overflow and stack underflow.

(Section 3.5.1) **(2 Marks)**

Q. 22 Convert the given infix expression to post fix expression using stack and also show the details of stack at each step of conversion expression :

(a * b + c/d) * (e + f ↑ g) **(Example 3.6.11)(8 Marks)**

Q. 23 What is recursion ? Explain how stack is used in recursive function with example.

(Sections 3.8 and 3.8.3) **(4 Marks)**

Q. 24 Explain push and pop operation on stack using array representation. **(Section 3.4.1)** **(4 Marks)**

➔ Winter 2012 – Total Marks 18

Q. 25 What is stack ? Explain primitive operation of stacks. **(Sections 3.1 and 3.3)** **(2 Marks)**

Q. 26 Convert the given infix expression to postfix expression using stack and the details of stack at each step of conversion.

Expression : a ↑ b * c – d + e / f / (g + h)

Note : ↑ indicates exponent operator

(Example 3.6.13) **(8 Marks)**

Q. 27 Describe recursion related to stacks and give two examples. **(Section 3.8.3)** **(4 Marks)**

Q. 28 Consider the following arithmetic expression P written in postfix notation :

P : 4, 2, ↑, 3, *3, –, 8, 4, 1, 1, 1, +, /, +.

Evaluate this expression to find its value.

Note : ↑ indicates exponent operator.

(Example 3.7.4) **(4 Marks)**

➔ Summer 2013 – Total Marks 18

Q. 29 Define stack and give basic operations performed on it. **(Sections 3.1 and 3.3)** **(2 Marks)**

Q. 30 Write a procedure to push on element on stack. Also give meaning of stack overflow.

(Sections 3.4.1 and 3.5.1) **(4 Marks)**

Q. 31 Convert the following Infix expression into postfix expression using stack and also show the details of stack content after each step. Expression : ((A + B) * D) ↑ (E – F) **(Example 3.6.10)** **(8 Marks)**

Q. 32 Write a 'C' program for implementation of stack using array. **(Program 3.5.2)** **(4 Marks)**

➔ Winter 2013 – Total Marks 30

Q. 33 Write any two applications of stacks

(Section 3.6) **(2 Marks)**

Q. 34 Write a 'C' program to calculate factorial of number using recursion. **(Program 3.8.1)** **(4 Marks)**

Q. 35 Describe primitive operation on stack.
(Section 3.3) **(4 Marks)**

Q. 36 Evaluate the given postfix expression to find its value
(6, 2, 3, +, –, 3, 8, 2, +, +, *, 2, ↑, 3, +)
(Example 3.6.4) **(4 Marks)**

Q. 37 Write a 'C' program to implement push and pop function in stack as an array.
(Section 3.4.1) **(8 Marks)**

Q. 38 Convert the given infix string to prefix expression and show the details of stack at each step.
(A – B/C) * (D * E – F) **(Example 3.7.1)** **(8 Marks)**

➔ Summer 2014 – Total Marks 24

Q. 39 Explain stack as an abstract data type.
(Section 3.2) **(4 Marks)**

Q. 40 Convert the following infix expression to its postfix form
(i) A + B – C*D/E + F
(ii) A * B – C ↑ D + E/F **(4 Marks)**

Ans. :

(i) A + B – C*D/E + F :

AB +CD*E/– F+

(ii) A * B – C ↑ D + E/F :

AB * CD ↑ – EF/+

Q. 41 WAP using recursion to print the factorial of a number. **(Program 3.8.1)** **(4 Marks)**

Q. 42 What is the affect of PUSH and POP operation on to the stack ? The stack contain 10, 20, 22, 26, 28, 30, with 30 being at top of the stack. Show diagrammatically the affect of

PUSH 46
PUSH 48
POP
POP
POP
PUSH 82

(Section 3.4.1 and Example 3.3.1) **(8 Marks)**

Q. 43 Explain the concept of representing stack through arrays. **(Sections 3.4 and 3.4.1)** **(4 Marks)**

➔ Winter 2014 – Total Marks 30

Q. 44 State applications of stack. **(Section 3.6)** **(2 Marks)**

Q. 45 Find out infix equivalent of the following postfix expression.
(i) AB + C * D – (ii) ABC * + D – **(4 Marks)**

Ans. :

(I) AB + C * D – :

(A + B) * C – D

(II) ABC * + D – :

(A + B * C) – D

Q. 46 Explain PUSH and POP operation on stack using array representation. **(Section 3.4.1)** **(4 Marks)**

Q. 47 Explain the term 'overflow' and 'underflow' with respect to stack use suitable data and diagram.
(Section 3.5.1) **(4 Marks)**

Q. 48 Define recursion. Write a 'C' program to calculate the factorial of number using recursion.
(Section 3.8 and Program 3.8.1) **(8 Marks)**

Q. 49 Covert the following infix expression into a postfix expression and show details of stack at each step of conversion.
Expression: ((A + B) * D) ↑ (E – F)
(Example 3.6.10) **(8 Marks)**

➔ Summer 2015 – Total Marks 30

Q. 1(A)(e) Define stack.
(Ans. : Refer section 3.1) **(2 Marks)**

Q. 2(b) Convert the following arithmetic expression P written in postfix notation into infix.
P: 5,6, 2, +, *, 12, 4, /, —
also evaluate P for final value.
(Ans. : Refer Example 3.7.3) **(4 Marks)**

Q. 3(a) Define recursion. Write a 'C' program for multiplication of natural numbers using recursion.
(Ans. : Refer section 3.8 and Program 3.8.4)
(4 Marks)

Q. 4(b) Explain stack as an abstract data type.
(Ans. : Refer section 3.2) **(4 Marks)**

Q. 5(b) Convert the given infix expression to postfix expression using stack and the details of stack at each step of conversion

Expression :a ↑ b * c-d+e/f /(g + h)

Note : ↑ indicates exponent operator.

(Ans. : Refer Example 3.6.12) **(8 Marks)**

Q. 6(a) Explain PUSH and POP operation on stack with algorithm and example.
(Ans. : Refer section 3.4.1) **(8 Marks)**

➔ Winter 2015 – Total Marks 30

Q. 1(A)(d) Define push and pop operation of stack.
(Ans. : Refer section 3.3) **(2 Marks)**

Q. 2(b) Convert infix string ((A + B) * (C – D)) / (E + F) into prefix string with stack. Show the content of stack in each step.
(Ans. : Refer Example 3.7.6) **(4 Marks)**

Q. 3(a) Write an algorithm to convert infix string to postfix (fully parenthesized).
(Ans. : Refer Example 3.6.16) **(4 Marks)**

Q. 4(b) Describe the representation of stack using arrays. *(Ans. : Refer section 3.4)* **(4 Marks)**

Q. 5(b) Consider the following arithmetic expression written in postfix notation :

10, 2, *, 15, 3, /, +, 12, 3, 2, ↑, +, . Evaluate this expression to find its value.
(Ans. : Refer Example 3.7.7) **(8 Marks)**

Q. 6(a) Describe the term stack overflow and stack underflow with example. State any four applications of stack.
(Ans. : Refer sections 3.2(4), 3.2(5) and 3.6) **(8 Marks)**

➔ Summer 2016 – Total Marks 16

Q. 2(b) Explain how stack can be used to reverse a list using suitable example.
(Ans. : Refer section 3.5.2) **(4 Marks)**

Q. 3(b) Translate the following infix expression to its equivalent prefix expression,
(x + y) * (p – q).
Evaluate the above prefix expression with following values :
x = 2, y = 3,p = 6, q = 2.
(Ans. : Refer Example 3.7.8) **(8 Marks)**

Q. 5(c) State importance of top pointer in stack. Explain stack overflow and underflow.
(Ans. : Refer sections 3.1 and 3.5.1) **(4 Marks)**

➔ Winter 2016 – Total Marks 22

Q. 1(A)(4) Define Recursion. State any two application where recursion used.
(Ans. : Refer section 3.8) **(2 Marks)**

Ans. : Application where recursion used are :

1. Tower of Hanoi 2. Fibonacci Series

Q. 2(b) Find out prefix equivalent of the following expression :

i) [(A + B) + C] * D ii) A [(B* C) + D]
(Ans. : Refer Example 3.7.9) **(4 Marks)**

Q. 3(a) Describe PUSH and POP operation on stack using array representation.
(Ans. : Refer section 3.4.1) **(4 Marks)**

Q. 4(b) How stack is used in recursion ? Describe with suitable example.
(Ans. : Refer sections 3.8 and 3.8.3) **(4 Marks)**

Q. 5(b) Convert the given infix expression to postfix expression using stack and the details of stack at each step of conversion.

EXPRESSION : P * Q ↑ R – S/T + [U/V]
(Ans. : Refer Example 3.7.10) **(8 Marks)**

➔ Summer 2017 – Total Marks 20

Q. 2(b) Convert following expression into postfix exponent indicates operat or form with illustration of all steps.

A + B ↑ C * (D/E) – F% G

(Ans. : Refer Example 3.6.14) **(4 Marks)**

Q. 3(c) Explain operations on stack using array.
(Ans. : Refer section 3.4.1) **(4 Marks)**

Q. 4(b) Recursion is one of the application of stack-YES/NO ? Explain it for calculating the factorial of a number 5 using recursion.
(Ans. : Refer (Yes) section 3.8) **(4 Marks)**

Q. 5(b) Explain any three application of stack in detail with example.

(Ans. : Refer section 3.6) **(8 Marks)**

➔ Winter 2017 – Total Marks 24

Q. 2(b) Write a 'C' program to display Fibonacci series using recursive function.

(Ans. : Refer program 3.8.2) **(8 Marks)**

Q. 3(a) Desribe stack as an abstract data type.

(Ans. : Refer section 3.2) **(4 Marks)**

Q. 4(b) Describe following terms with suitable example.

i) Stack overflow

ii) Stack underflow

(Ans. : Refer section 3.5.1) **(4 Marks)**

Q. 5(b) Convert following infix expression into a postfix expression. Show all steps.

(P + (Q * R – (S/T↑U) * V) * W)

(Ans. : Refer Example 3.6.15) **(8 Marks)**

❑❑❑

CHAPTER 4

Unit - III

Queues

Syllabus Topics

Introduction to Queue :

- Queue representation in memory using array
- Queue as an ADT
- Types of Queues : Linear Queue, Circular Queue, Concept of Priority queue.
- Queue Operations : INSERT, DELETE
- Queue Operations Conditions : Queue Full, Queue Empty
- Applications of Queue

Syllabus Topic : Introduction to Queue

4.1 Introduction

➔ **(MSBTE : W-10, W-12, W-13, W-15, S-16)**

Q. 4.1.1 Define queue. (Refer section 4.1) **W-10, W-13**

Q. 4.1.2 Explain the front and rear for queue. (Refer section 4.1) **W-12**

Q. 4.1.3 Define the terms FRONT and REAR of queue. (Refer section 4.1) **W-15**

Q. 4.1.4 Explain the terms front and rear with relevance to queue. (Refer section 4.1) **S-16**

It is a special kind of list, where items are inserted at one end (the rear) and deleted from the other end (front). Queue is a FIFO (First In First Out) list.

We come across the term queue in our day to day life. We see a queue at a railway reservation counter, or a movie theatre ticket counter. Before getting the service, one has to wait in the queue. After receiving the service, one leaves the queue. Service is provided at one end (the front) and people join at the other end (rear).

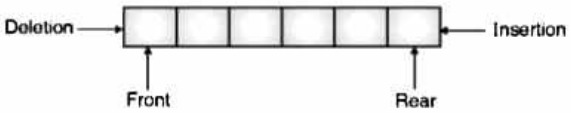


Fig. 4.1.1 : Insertion of elements is done at the rear end and deletion from the front end

Syllabus Topic : Queue Representation in Memory using Array

4.1.1 Array Representation and Implementation of Queues

➔ **(MSBTE : W-13, S-17, W-17)**

Q. 4.1.5 Explain how two pointer front and rear related to queue with diagram. (Refer section 4.1.1) **W-13**

Q. 4.1.6 Sketch representation of queue as an array. (Refer section 4.1.1) **S-17**

Q. 4.1.7 Define following term : empty Queue. (Refer section 4.1.1) **W-17**

An array representation of queue requires three entities :

(a) An array to hold queue elements.

(b) A variable to hold the index of the front element.

(c) A variable to hold the index of the rear element.

A queue data type may be defined formally as follows :

```
# define MAX 30

typedef struct queue

{

    int data[MAX];

   int front, rear;

} queue;
```

- During initialization of a queue, its front and rear are set to –1.

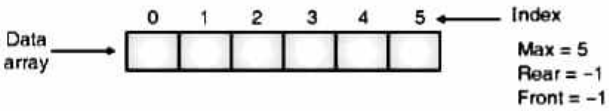


Fig. 4.1.2 : An empty queue after initialization

- Fig. 4.1.3 shows the status of an queue after insertion of the element '5'.

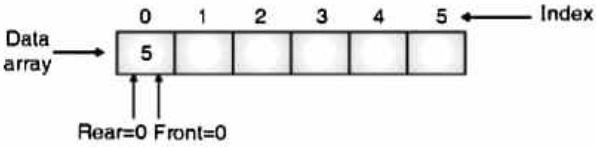


Fig. 4.1.3 : A queue after insertion of the first element '5'

- On subsequent insertions, front remains at the same place, where rear advances.

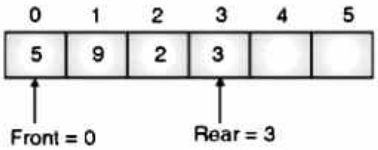


Fig. 4.1.4 : A queue after insertion of four elements

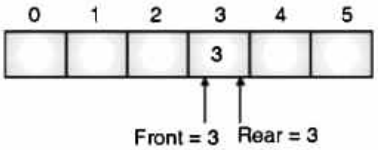


Fig. 4.1.5 : A queue after 3 successive deletions

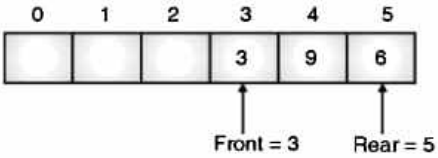


Fig. 4.1.6 : A queue after 2 successive insertions

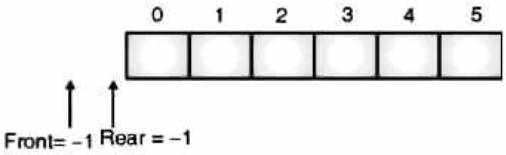


Fig. 4.1.7 : A queue after 3 successive deletions

Following points can be observed :

(1) If the queue is empty then front = –1 and rear = –1.

(2) If the queue is full then rear = MAX – 1.

(where MAX is the size of the array used for storing of queue elements).

(3) If rear = front then the queue contains just one element.

(4) If rear > front then queue is non-empty.

Problem with the above representation of the queue :

Problem of overflow : Refer to the Fig. 4.1.6, the queue has become full as rear = MAX –1. There are three vacant spaces (location 0, 1, 2) but these spaces cannot be utilized.

Problem of overflow can be handled by moving the queue elements to their left by number of vacant spaces. This operation could be very time consuming for a large queue.

4.1.2 Comparison between Array Representation and the Linked Representation of a Queue

- Maximum size of the queue is fixed at the time of compilation, when the queue is represented using an array, this causes wastage of storage space. If the queue size is arbitrarily large and if less area is reserved for the queue then there will be a problem of frequent overflow.
- In case of a queue using linked structure, no memory area is reserved in advance. Memory for a node is acquired during run time, whenever a fresh insertion is to be made. There is no problem of overflow. As long as the system has free memory it can be given for a queue element.
- Array representation of a queue is simple to implement whereas the linked representation requires additional knowledge of linked list and dynamic data structure. Many

programming languages do not support dynamic data structure.

- In linked representation, additional memory is required to store the address of the next element. There is no such requirement in case of array representation of a queue, as array is stored in contiguous memory locations.

4.1.3 Comparison between STACK and QUEUE

➔ **(MSBTE : W-09, S-11, S-15, S-17)**

Q. 4.1.8 **Distinguish between STACK and QUEUE.** **(Refer section 4.1.3)** **W-09**

Q. 4.1.9 **Distinguish between stack and queue with minimum four points?** **(Refer section 4.1.3)** **S-11, S-15, S-17**

- Stack is a LIFO (Last in First Out) structure. Queue is a FIFO (First in First Out) structure.
- In a stack, all insertions and deletions are permitted only at one end of the list. In a queue, items are inserted at one end (the rear) and deleted from the other end (front).
- Stack is always implemented using a linear array but queue should preferably be implemented using a circular array.
- Stack is widely used in recursion and processing of expressions. Queue is more useful in many of the real life applications.

Syllabus Topic : Queue Operation - INSERT, DELETE, Queue Operations Conditions - Queue Full, Queue Empty

4.2 Operations on Queue (Implemented using Array)

➔ **(MSBTE : S-08, S-09, S-10, W-12, W-13, W-14, S-15, W-15, S-16)**

Q. 4.2.1 **Explain any four terms in queue.** **(Refer section 4.2)** **S-08**

Q. 4.2.2 **Write a procedure to insert an element into a queue and to delete an element from a queue** **(Refer section 4.2)** **S-09, W-13, S-15, W-15**

Q. 4.2.3 **Write the procedure for inserting an element in queue. (Refer section 4.2)** **S-10, W-14**

Q. 4.2.4 **Explain queues implementation using linked list. (Refer section 4.2)** **W-12**

Q. 4.2.5 **Enlist any four operations of queue. (Refer section 4.2)** **W-15**

Q. 4.2.6 **List the operations to be performed at front and rear end. (Refer section 4.2)** **S-16**

A set of useful operations on a queue includes :

(1) **initialize( ) :** Initializes a queue by setting the value of rear and front to –1.

(2) **enqueue( ) :** Inserts an element at the rear end of the queue.

(3) **dequeue( ) :** Deletes the front element and returns the same.

(4) **empty( ) :** It returns true(1) if the queue is empty and returns false(0) if the queue is not empty.

(5) **full( ) :** It returns true(1) if the queue is full and returns false(0) if the queue is not full.

(6) **print( ) :** Printing of queue elements.

☞ **Realization of queue operations through C-function**

(a) Defining the maximum size of the queue.

```
#define MAX 50
```

(b) Data structure declaration

```
typedef struct Q
{
    int R, F;
    int data[MAX];
} Q;
```

"R" and "F" store the index of the rear and the front element respectively. "data[MAX]" is used to store the queue elements.

(c) Declaring a queue type variable.

$Q\ q_1, q_2$;

q_1 and q_2 are queue type variables

(d) Declaring a queue type pointer variable.

$Q\ q_1, *P$;

$P = \&q_1$

A pointer variable "P" can be used to store the address of a queue type variable.

➢ **'C' function to initialize()**

```
void initialize(Q *P)
{
    P → R = – 1;
    P → F = – 1;
}
```

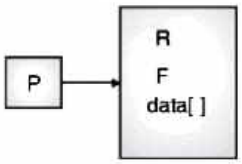


Fig. 4.2.1 : Elements of the structure 'Q' can be accessed through the pointer P (P → R, P → F, P → data[ ])

In the function "void initialize (Q *P)" a "Q" type variable is passed by address. Members of the structure type variables can be accessed through a pointer as shown in the Fig. 4.2.1. Initially the queue is empty and hence the value of the two variables "R (rear)" and "F (front)" is set to – 1.

➢ **'C' function to check whether queue is empty or not.**

```
int empty(Q *P)
{
    if(P → R == – 1)
        return(1);
    return(0);
}
```

Whenever the queue is empty, P → R (i.e. Rear field (R) of the queue type variable whose address is stored in P) or P → F will be – 1.

➢ **'C' function to check whether queue is full or not.**

```
int full(Q *P)
{
    if(P → R == MAX – 1)
        return(1);
    return(0);
}
```

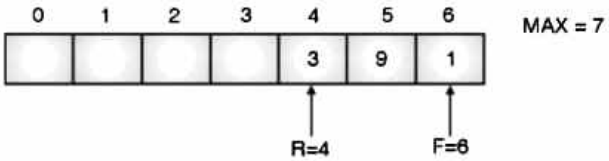


Fig. 4.2.2 : The queue is full as there is no space left for further insertion

1) enqueue( )

☞ **Algorithm for insertion in a queue**

(1) Inserting in an empty queue.

(a) R = F = 0;

(b) data[R] = x /* x is the element to be inserted */

/* Both R and F will point to the only element of the queue */

(2) Inserting in a non-empty queue.

R = R + 1;

data[R] = x

/* rear is advanced by 1 and the element x is stored at the rear end of the queue */.

Note : Insertion should be carried out after it is checked that the queue is not full.

➢ **'C' function for insertion in a queue.**

```
void enqueue(Q *P, int x)
{
    if(P → R == – 1)        /* empty queue */
    {
        P → R = P → F = 0;
        P → data[ P→ R] = x;
    }
    else
    {
        P → R = P → R + 1;
        P → data[P → R] = x;
    }
}
```

2) dequeue( )

☞ **Algorithm for deletion from a queue**

(1) Deletion of the last element or only element(R = F)

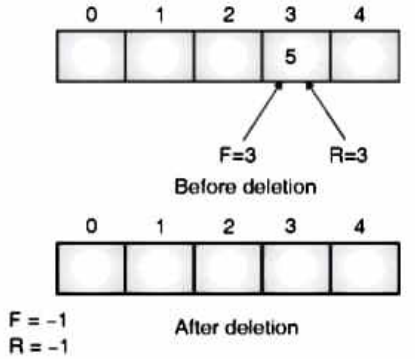
(a) x = data[F] (b) R = F = – 1;

(c) return(x)

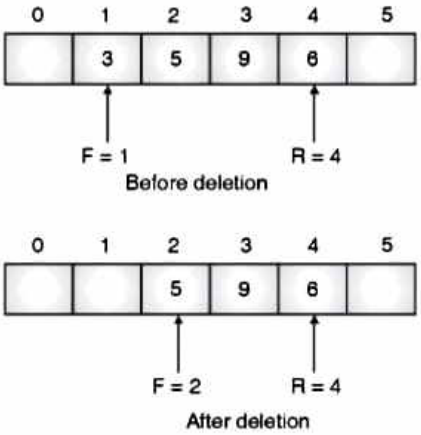
(2) Deletion of the element when the queue length > 1

(a) x = data[F] (b) F = F + 1

(c) return(x);



(a) Deletion of the last element



(b) Deletion of an element from the queue when the queue length > 1

Fig. 4.2.3

Note :	Deletion of an element from the queue should be carried out after checking that it is not empty.

➢ **'C' function for deletion from a queue**

```
int dequeue (Q*P)
  {
    int x;
    x = P → data [P→F];
    if (P→ R= =P→ F)
    {
      P→ R= –1;
      P→ F = –1;
    }
    else
      P→ F=P→ F+1;
    return(x);
    }
```

➢ **Program 4.2.1 : A menu driven program for queue.**

Q. 4.2.7	Write a menu driven program in 'C' language for queue having menus : Store , retrieve and display queue. (Refer Program 4.2.1)	**S-08**
Q. 4.2.8	Write a 'C' program to implement a queue with insert and delete operation. (Refer Program 4.2)	**S-12, W-15, S-17**
Q. 4.2.9	Write a menu driven program to insert, delete an element in a queue and display the queue. (Refer Program 4.2.1)	**S-14**
Q. 4.2.10	Write a program to insert element in queue. (Refer Program 4.2.1)	**W-16**
Q. 4.2.11	Write a code delete an element in queue. (Refer Program 4.2.1)	**W-16**

```
/* Simulation of queue using an array */
#include<conio.h>
#include<stdio.h>
#define MAX 5

typedef struct Q
{
  int R,F;
  int data[MAX];
}Q;
void initialise(Q *P);
```

```
int empty(Q *P);
int full(Q *P);
void enqueue(Q *P,int x);
int dequeue(Q *P);
void print(Q *P);
void main()
{
    Q q;
    int op,x;
    initialise(&q);
    clrscr();
    do
      {
        printf("\n\n1)Insert\n2)Delete\n3)Print\n4)Quit");
        printf("\nEnter Your Choice:");
        scanf("%d",&op);
        switch(op)
          {
            case 1: printf("\n Enter a value:");
                scanf("%d",&x);
                if(!full(&q))
                    enqueue(&q,x);
                else
                    printf("\nQueue is full !!!!");
                break;
            case 2: if(!empty(&q))
                  {
                    x=dequeue(&q);
                    printf("\Deleted Data=%d",x);
                  }
                else
                printf("\nQueue is empty !!!!");
                break;
            case 3: print(&q);break;
          }
    }while(op!=4);
}
void initialise(Q *P)
{
    P→ R=-1;
    P→ F=-1;
}
int empty(Q *P)
{
    if(P→ R==-1)
        return(1);
    return(0);
}
int full(Q *P)
{
    if(P→ R==MAX-1)
        return(1);
    return(0);
}
void enqueue(Q *P,int x)
{
    if(P→ R==-1)
    {
        P→ R=P→ F=0;
        P→ data[P→ R]=x;
    }
    else
    {
        P→ R=P→ R+1%MAX;
        P→ data[P → R]=x;
    }
}
int dequeue(Q *P)
{
```

```
    int x;
    x=P→ data[P→ F];
    if(P→ R==P→ F)
    {
        P→ R=-1;
        P→ F=-1;
    }
    else
        P→ F=P→ F+1;
    return(x);
}
void print(Q *P)
{
    int i;

    if(!empty(P))
    {
        printf("\n");
        for(i=P→ F;i<=P→ R;i++)
            printf("%d\t",P→ data[i]);
    }
}
```

Syllabus Topic : Queue as an ADT

4.2.1 Queue as an ADT

➔ **(MSBTE : S-10, W-11, S-13, S-14, W-14, W-15, W-17)**

Q. 4.2.12 Explain queue as an abstract data type. (Refer section 4.2.1) **S-10, S-13, S-14, W-14**

Q. 4.2.13 Write a brief note on queue as an abstract data type. (Refer section 4.2.1) **W-11**

Q. 4.2.14 Describe how queue works as an abstract data type. (Refer section 4.2.1) **W-15**

Q. 4.2.15 Define following term : Queue as an Abstract Data Type. (Refer section 4.2.1) **W-17**

☞ **Data structure declaration**

```
typedef struct Q
    {
        int R, F;
        int data [MAX];
    } Q;
```

R and F store the indices of the rear and the front element respectively. The array data[] is used to store the queue elements. MAX is a predefined constant.

☞ **Prototype of queue functions**

1. void initialize (Q*);
2. int empty (Q*);
3. int full (Q*);
4. void enqueue(Q*, int);
5. int dequeue(Q*);
6. void print (Q*);

Example 4.2.1 : Consider the following queue of characters, implemented as array of six memory locations :

Front = 2, Rear = 3

Queue : –, A, D, –, –, –

Where '–' denotes empty cell. Describe the queue as the following operations take place

(i) Add 'S' (ii) Add 'J'

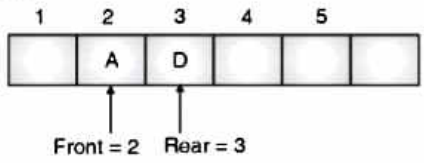
(iii) Delete two letters

(iv) Shift towards left to bring all free spaces to the right side

(v) Insert M, H, I and delete on letter.

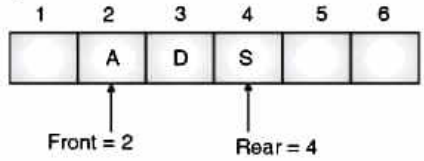
Solution :

Initial queue :



(a)

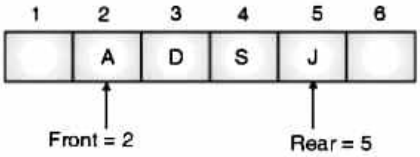
(i) Add 'S'



(b)

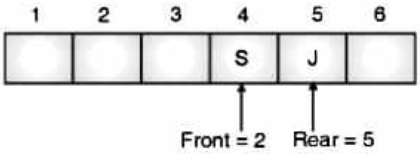
Fig. Ex. 4.2.1 Contd...

(ii) Add 'J'



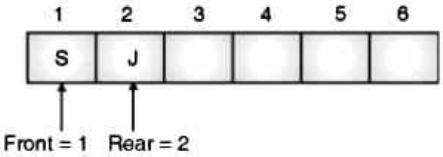
(c)

(iii) Delete two letters



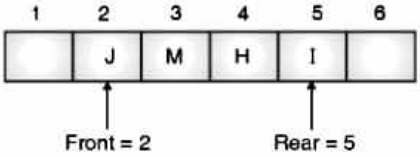
(d)

(iv) Shift towards left to bring all free spaces to the right.



(e)

(v) Insert M, H, I and delete one letter.



(f)

Fig. Ex. 4.2.1

Syllabus Topic : Types of Queue

4.3 Types of Queue

– There are mainly three types of queues

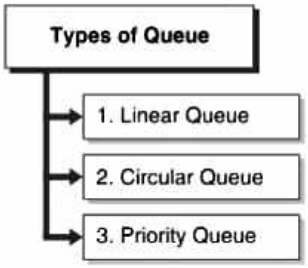


Fig. 4.3.1 : Types of queues

– Linear queue is already discussed in the section 4.2.

– Circular queue has been discussed in the section 4.3.

– Priority queue is discussed in section 4.4.

Syllabus Topic : Circular Queue (Array)

4.3.1 Circular Queue (Array)

➔ **(MSBTE : S-08, W-08, S-10, S-11, S-12, W-13, S-14, W-14, S-15, W-16, S-17)**

Q. 4.3.1 With suitable diagram, explain the principle of circular queue. (Refer section 4.3.1) **S-08**

Q. 4.3.2 Draw and explain circular queue in detail. (Refer section 4.3.1) **W-08, S-15**

Q. 4.3.3 Define circular queue. Also explain advantage of circular queue over linear queue with example. (Refer section 4.3.1) **S-10, W-17**

Q. 4.3.4 Explain the concept of circular queue with example. (Refer section 4.3.1) **S-11, S-14**

Q. 4.3.5 What is circular queue ? (Refer section 4.3.1) **S-12**

Q. 4.3.6 Describe circular queue with an example. (Refer section 4.3.1) **W-13**

Q. 4.3.7 Define circular queue. (Refer section 4.3.1) **W-14**

Q. 4.3.8 Describe circular queue. Give its advantage. (Refer section 4.3.1) **W-16**

☞ **Advantages of circular queue**

(i) Circular queue is a linear data structure in which last node is connected back to first node to form a circle.

(ii) It follows FIFO principle.

(iii) Elements are inserted at rear end deleted from front end.

(iv) It is also known as "Ring Buffer".

(v) Only one pointer is required to be maintained.

There is one potential problem with implementation of queue using a simple array. The queue may appear to be full although there may be some space in the queue.

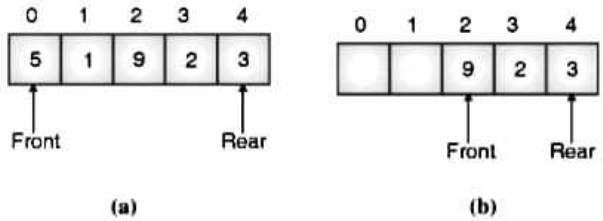


Fig. 4.3.2 : Queue is full, although locations 0 and 1 are vacant

– After insertion of five elements in the array (a queue) as shown in Fig. 4.3.2,

$$\left.\begin{array}{l}\text{rear} = 4\\ \text{front} = 0\end{array}\right\}\text{queue is full}$$

– After two successive deletions

$$\left.\begin{array}{l}\text{rear} = 4\\ \text{front} = 2\end{array}\right\}\text{queue is full}$$

queue in the Fig. 4.3.2 is full as there is no empty space ahead of rear.

The simple solution is that whenever rear gets to the end of the array, it is wrapped around to the beginning. Now, the array can be thought of as a circle. The first position follows the last element.

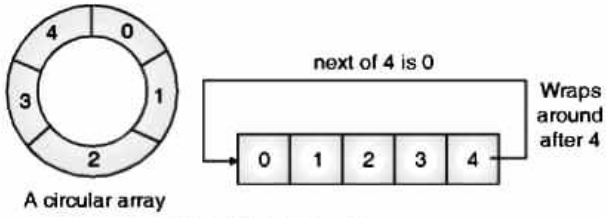


Fig. 4.3.3 : A circular array

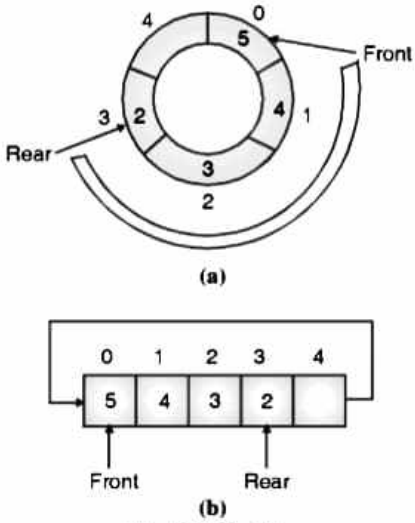


Fig. 4.3.4 Contd...

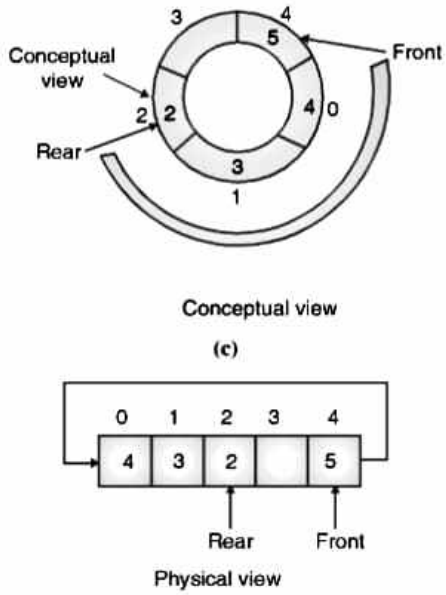


Fig. 4.3.4 : Queue at various places in an array

In a circular array, the queue is found somewhere around the circle in consecutive positions.

4.3.2 Implementation of a Circular Movement inside a Linear Array

➔ **(MSBTE : S-12, W-14)**

Q. 4.3.9 Explain insertion and deletion operation on circular queue. (Refer section 4.3.1) **S-12, W-14**

– To give a circular movement inside an array, whenever we go past the last element of the array, it should come back to the beginning of the array.

– Thus, when we move around an array of size five, following sequence for locations should be generated.

0 1 2 3 4 0 1 2 3 4 .....

i = (i + 1)%5 – Array of size 5

i = (i + 1)%MAX → Array of size MAX

expression used for circular movement

i	(i + 1)%5		
0	(0 + 1)%5	=	1
1	(1 + 1)%5	=	2
2	(2 + 1)%5	=	3
3	(3 + 1)%5	=	4
4	(4 + 1)%5	=	0 ← i wraps around to 0

```
i = 0;
while(1)
i =(i + 1)%5;
```

Above program segment will continue generating a circular sequence 0 1 2 3 4 0 1 2 3 .....

➢ **Data type for queue in a circular array.**

```
# define MAX 30
                    /* A queue with maximum of 30 elements */
typedef struct queue
{
    int data[MAX];
    int front, rear;
}queue;
```

☞ **Various operations on the queue in a circular array**

➢ **'C' function to initialize the queue by setting values of rear and front as – 1.**

```
void initialize(queue *P)
{
    P → rear = – 1;
    P → front = –1;
}
int empty(queue *P)
{
    if(P → rear == – 1)
        return(1);
    return(0);
}
int full(queue *P)
{
    if((P → rear + 1)% MAX == P → front)
        return(1);
    return(0);
}
```

Fig. 4.3.5 shows, queue is full.

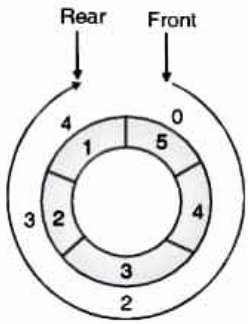


Fig. 4.3.5

When the queue becomes full, there will be no space ahead of rear in the circular array. Hence, front will be found, immediately ahead of rear in a queue that is full.

i.e. (p → rear + 1) % MAX == P → front

➢ **'C' function to insert the element in queue (enqueue)**

```
void enqueue(queue *P, int x)
{
    if(empty(P))                    /* empty queue */
    {
        P → rear = P → front==0;
            /* rear and front will point to the same element */
            P → data[P → rear] = x;
    }
    else
    {
        P → rear = (P → rear + 1)% MAX;
/* Advance P in circular array */
        P → data[P → rear] = x;
    }
}
```

➢ **'C' function to delete the element from queue (dequeue)**

```
int dequeue(queue *P)
{
    int x;
    x = P → data[P → front];
    if(P → rear == P → front)
                        /* deleted the last element*/
        initialize(P);
    else
        P → front =(P → front + 1)% MAX;
  /* front advances to the next position in circular array */
    return(x);
}
void print(queue *P)
{
  int i;
  i = P → front;
  while(i ! = P → rear)
  {
    printf("\n%d", P → data[i]);
    i = (i + 1) %MAX;
  }
  printf("\n%d", P → data[P → rear]);
}
```

Example 4.3.1 : Explain with suitable diagram queue full and queue empty condition for circular queue **W-09**

OR Explain 'Queue Full' and 'Queue Empty' condition with suitable example. **S-16**

Solution :

When the queue becomes full, there will be no space ahead of rear end in the circular array. Hence, front will be found immediately ahead of rear in the queue that is full. The Fig. 4.3.5 shows that the queue is full. The queue full condition is given by :

(rear + 1)% MAX == front, where MAX is the size of the array.

Queue empty condition is shown in Fig. Ex. 4.3.1.

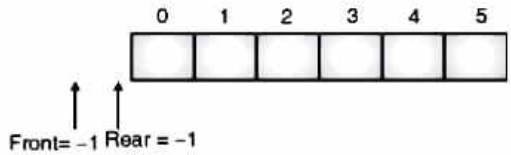


Fig. Ex. 4.3.1 : Queue empty condition

When the queue is empty, both rear and front will be equal to –1.

➢ **Program 4.3.1 : Program showing various operations on a circular queue.**

Q. 4.3.10 Write a program in 'C' language to implement 'circular queue'. (Refer Program 4.3.1) **S-13**

```
#include <stdio.h>
#include <conio.h>
#define MAX 30
typedef struct queue
{
    int data[MAX];
    int rear, front;
} queue;
void initialize (queue *p);
int empty (queue *p);
int full (queue *p);
void enqueue (queue *p, int x);
int dequeue (queue *p);
void print (queue *p);

void main ( )
{
    int x, op, n;
    queue q;
    initialize (&q);
    do
    {
```

```
printf ("\n1)create \n2)insert \n3)delete \n
        4)print \n5)Quit");
printf("\n enter your choice :");
scanf ("%d", &op);
switch(op)
{ case 1 : printf("\n enter no. of elements :");
        scanf("%d", &n);
        initialize (&q);
        printf("\n enter the data :");
        for(i =0; i < n; i++)
        {
            scanf("%d", &x);
            if (full(&q))
            {
                printf("\n queue is full ...");
                exit(0);
            }
            enqueue (&q, x);
        }
        break;
case 2 : printf("\n enter the element to be inserted :");
        scanf("%d", &x);
        if (full(&q))
        {
            printf("\n queue is full ...");
            exit(0);
        }
        enqueue (&q, x);
        break;
case 3 : if (empty(&q))
        {
            printf("\n queue is empty ...");
            exit (0);
        }
            x = dequeue (&q);
            printf("\n element =%d", x);
            break;
        case 4 : print (&q);
            break;
        default : break;
        }
    }while (op! = 5);
}
void initialize(queue *P)
{
    P → rear = – 1;
    P → front = –1;
}
int empty(queue *P)
{
    if(P → rear == – 1)
        return(1);
    return(0);
}
int full(queue *P)
{
    if((P → rear + 1)% MAX == P → front)
        return(1);
    return(0);
}
void enqueue(queue *P, int x)
{
    if(empty(P))                /* empty queue */
    {
        P → rear = P → front==0;
            /* rear and front will point to the same element */
            P → data[P → rear] = x;
    }
```

```
    else
    {
        P → rear = (P → rear + 1)% MAX;
    /* Advance P in the circular array */
        P → data[P → rear] = x;
    }
}
int dequeue(queue *P)
{
    int x;
    x = P → data[P → front];
    if(P → rear == P → front)  /* deleted the last element */
            initialize(P);
        else
            P → front =(P → front + 1)% MAX;
  /* front advances to the next position in the circular array */
        return(x);
}
void print(queue *P)
{
    int i;
    i = P → front;
    while(i != P → rear)
    {
        printf("\n%d", P → data[i]);
        i = (i + 1) %MAX;
    }
    printf("\n%d", P → data[P → rear]);
}
```

Example 4.3.2 : Consider the following queue of characters, where QUEUE is a circular array which is allocated six memory cells FRONT = 2, REAR = 4, QUEUE = ____, A, C, D, ____, ____

Describe queue as the following operations takes place.

(i) F is added to queue
(ii) Two letters are deleted
(iii) K, L, M are added to queue
(iv) Two letters are deleted
(v) R is added to queue
(vi) Two letters are deleted
(vii) S is added to queue
(viii) One letter is deleted
(ix) One letter is deleted **W -10**

Solution :

Sr. No.	Operation	QUEUE	REAR	FRONT
1.	Initially	–, A, C, D, –, –	4	2
2.	F is added	– , A, C, D, F, –	5	2
3.	Two letters are deleted	–, –, –, D, F, –	5	4
4.	K, L, M are added	L, M, –, D, F, K	2	4
5.	Two letters are deleted	L, M, –, –, – , K	2	6
6.	R is added	L, M, R, –, –, K	3	6
7.	Two letters are deleted	–, M, R, –, –	3	2
8.	S is added	–, M, R, S, –, –	4	2
9.	One letter is deleted	–, –, R, S, –, –	4	3
10.	One letter is deleted	–, –, –, S, –, –	4	4

Syllabus Topic : Concept of Priority Queue

4.4 Priority Queue

➔ **(MSBTE : S-09, W-10, S-11, S-12, W-12, S-13, W-13, S-14, W-14, S-15, S-16, W-16, S-17, W-17)**

Q. 4.4.1 Define priority queue. Describe the one-way list representation of a priority queue with suitable example and diagram.
(Refer section 4.4) **W-08**

Q. 4.4.2 Describe priority queues.
(Refer section 4.4) **S-09, S-13**

Q. 4.4.3 Describe priority queues along with an example. (Refer section 4.4) **W-10, W-12, S-14, W-14**

Q. 4.4.4 Explain priority queue with example. Explain its types. (Refer section 4.4) **S-11, W-13**

Q. 4.4.5 Explain the concept of priority queue with suitable example. (Refer section 4.4) **S-12**

Q. 4.4.6 Describe priority queue and list its advantages. (Refer section 4.4) **S-15, S-17**

Q. 4.4.7 With a neat sketch explain working of priority queue. (Refer section 4.4) **S-16**

Q. 4.4.8 What is priority queue ? Describe working of priority queue with suitable example. (Refer sections 4.4 and 4.4.1) **W-16**

Q. 4.4.9 Define following term : Proirity Queue. (Refer section 4.4) **W-17**

Priority queue is an ordered list of homogeneous elements. In a normal queue, service is provided on the basis of First-in-first-out. In a priority queue service is not provided on the basis of "first-come-first-served" but rather than each element has a priority based on the urgency of need.

- An element with higher priority is processed before other elements with lower priority.
- Elements with the same priority are processed on "first-come-first served" basis.
- An example of priority queue is a hospital waiting room. A patient having a more fatal problem will be admitted before other patients.
- Other applications of priority queues is found in long term scheduling of jobs processed in a computer. In practice, short processes are given a priority over long processes as it improves the average response of the system.

4.4.1 Implementation of Priority Queues

➔ **(MSBTE : W-16)**

Priority queue can be implemented using a circular array :

As the service must be provided to an element having highest priority, there could be a choice between :

(a) List is always maintained sorted on priority of elements with the highest priority element at the front. Here, deletion is trivial but insertion is complicated as the element must be inserted at the correct place depending on its priority.

(b) List is maintained in the "FIFO" form but the service is provided by selecting the element with highest priority. Deletion is difficult as the entire queue must be traversed to locate the element with highest priority. Here, insertion is trivial (at rear end).

☞ **Implementation of a Priority Queue using a circular array (Priority queue as an ADT)**

(a) Data type for priority queue in a circular array

```
# define MAX 30 /* A queue with maximum of 30 elements */
typedef struct pqueue
{ int data [MAX];
    int front, rear;
} pqueue;
```

(b) Operations on a priority queue

(i) initialize( ) : Make the queue empty.

(ii) empty( ) : Determine if the queue is empty.

(iii) full( ) : Determine if the queue is full.

(iv) enqueue( ) : Insert an element as per its priority.

(v) dequeue( ) : Delete the front element (front element will have the highest priority).

(vi) print( ) : Print elements of the queue.

(c) Prototype of functions used for various operations on the queue

- void initialize(pqueue *p);
- int empty(pqueue *p);
- int full(pqueue *p);
- void enqueue(pqueue *p, int x);
- int dequeue(pqueue *p);
- void print(pqueue *p);

- enqueue( ) operation will cause an overflow if the queue is full.
- Dequeue( ) operation will cause an underflow if the queue is empty.

➢ **C-implementation of functions**

Q. 4.4.10 Write a program in 'C' language to implement priority queue. (Refer section 4.4.1) **W-12**

```
void initialize (pqueue *p)
{   p → rear = – 1;
    p → front = – 1;
}
```

A value of rear or front as –1, indicates that the queue is empty.

```
int empty (pqueue *p)
{ if (P→ rear = = – 1)
        return (1);     /* queue is empty */
    return (0); /* queue is not empty */
}
int full (pqueue *p)
{   /* if front is next to rear in the circular array then the queue     is full */
      if (p → rear + 1)% MAX == p → front)
        return (1); /* queue is full */
    return (0);
}
void enqueue (pqueue *p, int x)
{ int i;
     if (full(p))
    printf("\n overflow …");
     else
    {   /* inserting in an empty queue */
        if (empty(p))
            {   p → rear = p → front = 0;
                p → data [0] = x;
            }
        else
        {
        /* move all lower priority data right by one place */
    i = p → rear ;
    while (x > p → data [i])
    {   p → data [(i + 1)%MAX] = p → data [i];
        /* position i on the previous element */
        i = (i – 1 + MAX) % MAX; /* anticlockwise
         movement inside the queue */
        if ((i + 1)% MAX = = p → front)
            break; /* if all elements have been moved */
    }
    /* insert x */
        i = (i + 1)% MAX;
        p → data [i] = x;
    /* re-adjust rear */
       p → rear = (p → rear + 1) % MAX;
    }
  }
}
int dequeue (pqueue *p)
{   int x;
    if (empty (p))
            printf("\n underflow …");
        else
        {   x = p → data [p → front];
            if (p → rear = = p → front) /* delete last element
                                                                    */
                initialize (p);
            else
                p → front = (p → front + 1) % MAX;
        }
    return (x);
```

```
    }
void print (pqueue *p)
{   int i, x;
    i = p → front;
    while ( i! = p → rear)
    {   x = p data [i];
        printf ("\n%d", x);
        i = (i + 1) % MAX;
    }
    /* print the last data */
        x = p → data [i];
        printf ("\n%d", x);
    }
```

➢ **Program 4.4.1 : Program showing various operations on a priority queue.**

```
#include <stdio.h>
#include <conio.h>
#define MAX 30
typedef struct pqueue
{   int data [MAX];
    int rear, front;
} pqueuel
void initialize(pqueue *p);
int empty(pqueue *p);
int full(pqueue *p);
void enqueue(pqueue *p, int x);
int dequeue(pqueue *p);
void print(pqueue *p);
void main( )
{ int x, op, n;
  pqueue q;
  initialize (&q);
do
{
  printf("\n1)create\n2)insert\n3)Delete\n4)print\n5)Quit");
  printf("\n enter your choice:");
  scanf ("%d", &op);
  switch (op)
  {
  case 1 : printf("\n enter no. of elements :");
        scanf("%d", &n);
        initialize (&q);
        printf("enter the data:");
        for (i = 0; i < n; i++)
        {
          scanf ("%d" &x);
          if (full (&q))
          {   printf("\n queue is full ...");
              exit(0);
          }
          enqueue (&q, x);
        }
        break;
  case 2 :  printf("\n enter the element to be inserted");
        scanf ("%d", &x);
        if (full(&q))
        {
          printf("\n queue is full ...");
          exit(0);
        }
        enqueue (&q, x);
        break;
  case 3 : if (empty (&q))
        {
              printf("\n queue is empty ...");
              exit(0);
        }
        x = dequeue (&q);
```

```
        printf("\n element = %d", x);
        break;
    case 4 : print(&q);
        break;
         default : break;
        }
    } while (op! = 5);
}
void initialize (pqueue *p)
{   p → rear = – 1;
    p → front = – 1;
}
/*A value of rear or front as –1,indicate that the queue is
empty.*/
int empty (pqueue *p)
{ if (p→ rear == – 1)
        return (1);     /* queue is empty */
    return (0);         /* queue is not empty */
}
int full (pqueue *p)
{/*if front is next rear in the circular array then the queue is
full*/
     if (p → rear + 1)% MAX == p → front)
        return (1);     /* queue is full */
    return (0);
}
void enqueue (pqueue *p, int x)
{ int i;
     if (full (p))
    printf("\n overflow …");
     else
    {   /* inserting in an empty queue */
        if (empty (p))
            {   p → rear = p → front = 0
                p → data [0] = x;
            }
        else
        {
        /* move all lower priority data right by one place */
    i = p → rear ;
    while (x > p → data [i])
    {   p → data [(i + 1)%MAX] = p → data [i];
        /* position i on the previous element */
        i = (i – 1 + MAX) % MAX; /* anticlockwise
movement inside the queue */
        if ((i + 1)% MAX = = p → front)
            break; /* if all elements have been moved */
    }
    /* insert x */
        i = (i + 1)% MAX;
        p → data [i] = x;
    /* re-adjust rear */
       p → rear = (p → rear + 1) % MAX;
    }
 }
}
int dequeue (pqueue *p)
{   int x;
    if (empty (p))
            printf("\n underflow …");
    else
        {   x = p → data [p → front];
            if (p → rear = = p → front) /* delete last element
*/
                initialize (p);
            else
                p → front = (p → front + 1) % MAX;
        }
    return (x);
    }
```

```
void print (pqueue *p)
{   int i, x;
    i = p → front;
    while ( i! = p → rear)
    {   x = p data[i];
        printf ("\n%d", x);
        i = (i + 1) % MAX;
    }
    /* print the last data */
        x = p → data[i];
        printf ("\n%d", x);
    }
```

☞ **Advantages of priority queue**

- Priority queue is an ordered list of homogeneous elements. In a normal queue, service is provided on basis of First-in-first-out.
- In a priority queue service is not provided on the basis of first come first served basis but rather than each element has a priority based on the urgency of need.

Syllabus Topic : Applications of Queue

4.5 Applications of Queue

➔ **(MSBTE : W-09, W-10, S-13, S-14)**

Q. 4.5.1	**State any two applications of queue (Refer section 4.5)**	**W-09 , W-10, S-13**
Q. 4.5.2	**Give applications of queue. (Refer section 4.5)**	**S-14**
Q. 4.5.3	**State any two application of queue. (Refer section 4.5)**	**S-17**

Queue is a very useful data structure. Various features of operating system are implemented using a queue.

(a) Scheduling of processes (Round Robin algorithm)

(b) Spooling (to maintain a queue of jobs to be printed)

(c) A queue of client processes waiting to receive the service from the server process.

(d) Various application software using non-linear data structure tree or graph requires a queue for breadth first traversal.

(e) Simulation of a real life problem with the purpose of understanding its behaviour. The probable waiting time of a person at a railway reservation counter can be found through the technique of computer simulation if the following concerned factors are known :

(1) Arrival rate

(2) Service time

(3) Number of service counters.

We will discuss some of the applications in detail :

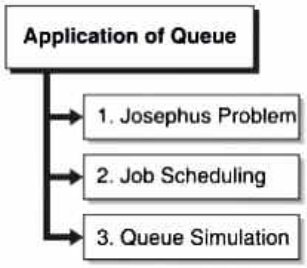


Fig. 4.5.1 : Applications of queue

4.5.1 Josephus Problem

This is an interesting programming problem known in the literature as Josephus problem. The problem is given as :

- Suppose there are n children standing in a queue.
- Students are numbered from 1 to n in the clockwise direction.
- Students choose a lucky number say m.

They start counting in clock-wise direction from the child designated as 1. The counting proceeds until the mth child is identified. m^{th} child is eliminated from the queue. Counting for the next round begins from the child next to the eliminated one and proceeds until the mth child is identified. This child is then eliminated and the process continues. After few rounds of counting only one child is left and this child is declared as winner.

☞ **Implementation**

It is required to write a program to identify a winner of the game. Input to the function are two parameters.

n : number of children

m : lucky number

Function returns an integer identifying the winner.

```
int winner(int m,int n)
{
    queue q;
    int i;
    init(&q);
        /*create a queue of n integers, numbered from 1 to n.
    Number i stands for ith child*/
    for(i=1;i<=n;i++)
        enqueue(&q,i);
  for(j=1;j<n;j++) /*n-1 iterations to eliminate n-1
children*/
        {
        for(i=1;i<m;i++) /* skip m-1 children*/
        {
            x=dequeue(&q);
            enqueue(&q,x);
        }
        x=dequeue(&q); /* remove m'th children*/
    }
    x=dequeue(&q); /*last child winner*/
    return(x);
}
```

4.5.2 Job Scheduling

Programs(jobs) entering a computer for execution are scheduled using some strategies. The purpose is to improve :

(i) CPU utilization

(ii) Response time

(iii) System utilization

The jobs entering the system are stored in the queue. These jobs can be selected for execution as per some pre-conceived strategy.

We will discuss one of the scheduling methods.

Round Robin technique

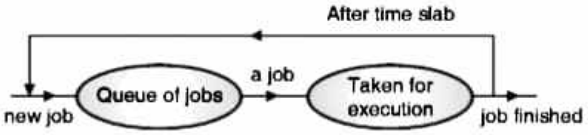


Fig. 4.5.2

Suppose there are n jobs waiting for execution. In Round Robin technique, CPU gives a fixed quantum of time to each job.

- A job is selected from the queue of jobs.
- It is executed for a fixed quantum of time. If the job is not completed by this time then it is removed and pushed back in queue of jobs.

Let us try to understand it with the help of an example.

The table below shows the jobs and the required run time.

Job no.	Run time
1	2 units
2	1 unit
3	2 units

Quantum of time = 1 unit

Initial status of queue

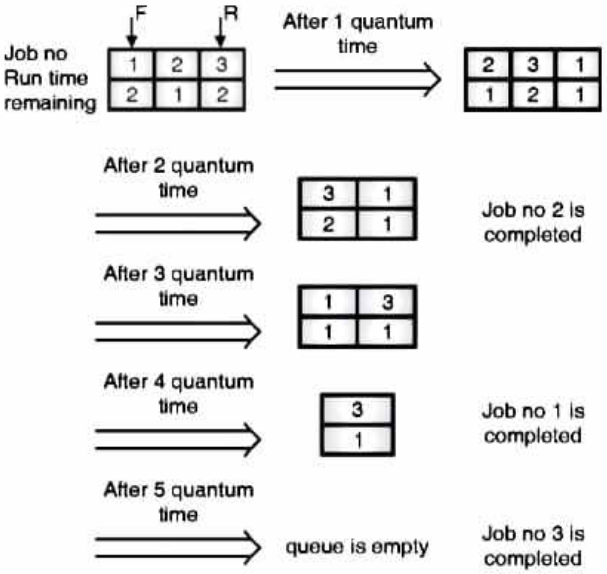


Fig. 4.5.3

```
void RR(Queue q)
{   /* q is a queue of jobs */
```

```
/* each job has two fields :1. jobno and 2. execution  time */
job t;
int quantum = 1; int time = 0;
while (! empty (q))
    {   t = dequeue (q);
        time = time + quantum;
        t.execution_time –=quantum;
        if (t.execution_time > 0)
            enqueue (q, t);
        else
            print t.jobno, time;
    }
}
```

4.5.3 Queue Simulation

Queuing system deals with computing probabilistically, how long users expect to wait on a line, how long the line gets. Waiting time and length of queue depends on :

(1) Arrival rate of users (2) Service time.

These parameters are given as probabilistic distribution function. We might be interested in finding how long on average a customer has to wait.

Computer can be used to simulate the behavior of a queue. A bank can utilize this information to determine how many cash counters are needed to ensure reasonable service time.

A typical queue simulation involves processing of events :

(1) Customer arriving (2) Customer departing

On arrival, a customer joins the queue and on departure he leaves the queue.

We can use the probabilistic function to generate a stream of ordered pairs. Each pair consists of :

(1) Arrival time (2) Service time.

These pairs are enqueued. A simulated clock can be used to give the concept of time. At each tick of the clock, all pending events are processed.

If the event is arrival, we check to see if a cash counter is free. If there is none, we place the arrival on the queue.

The waiting line of customers can be implemented as a queue. A cash counter can be implemented as a shared resource. Whenever the cash counter becomes free, a customer from the queue can be assigned to the shared resource (cash counter).

4.6 MSBTE Questions and Answers

➔ Summer 2008 – Total Marks 14

Q. 1 Explain any four terms in Queue. **(Section 4.2)** **(2 Marks)**

Q. 2 Write a menu driven program in 'C' language for Queue having menus : Store , Retrieve and display Queue. **(Program 4.2.1)** **(8 Marks)**

Q. 3 With suitable diagram, explain the principle of circular Queue. **(Section 4.3.1)** **(4 Marks)**

➔ Winter 2008 – Total Marks 12

Q. 4 Define priority queue. Describe the one-way list representation of a priority queue with suitable example and diagram. **(Sections 4.4 and 4.4.1)** **(8 Marks)**

Q. 5 Draw and explain circular queue in detail. **(Section 4.3)** **(4 Marks)**

➔ Summer 2009 – Total Marks 10

Q. 6 Describe priority queues. **(Section 4.5)** **(2 Marks)**

Q. 7 Write a procedure to insert an element into a queue and to delete an element from a queue. **(Sections 4.2)** **(8 Marks)**

➔ Winter 2009 – Total Marks 10

Q. 8 State any two application of queue. **(Section 4.5)** **(2 Marks)**

Q. 9 Distinguish between stack and queue. **(Section 4.1.3)** **(4 Marks)**

Q. 10 Explain with suitable diagram queue full and queue empty condition for circular queue. **(Example 4.3.1)** **(4 Marks)**

➔ Summer 2010 – Total Marks 16

Q. 11 Explain queue as an abstract data type. **(Section 4.2.1)** **(4 Marks)**

Q. 12 Write the procedure for inserting and deleting an element from queue. **(Section 4.2)** **(6 Marks)**

Q. 13 Define circular queue. Also explain advantage of circular queue over linear queue with example. **(Section 4.3.1)** **(6 Marks)**

➔ Winter 2010 – Total Marks 14

Q. 14 Define queue. State it's two applications. **(Sections 4.1 and 4.5)** **(2 Marks)**

Q. 15 Describe :

(i) Priority queues

along with an example.

(Sections 4.4) **(8 Marks)**

Q. 16 Consider the following queue of characters, where QUEUE is a circular array which is allocated six memory cells

FRONT = 2, REAR = 4, QUEUE = ____, A, C, D, ____, ____

Describe queue as the following operations takes place.

(i) F is added to queue

(ii) Two letters are deleted

(iii) K, L, M are added to queue

(iv) Two letters are deleted

(v) R is added to queue

(vi) Two letters are deleted

(vii) S is added to queue

(viii) One letter is deleted

(ix) One letter is deleted.

(Example 4.3.2) **(4 Marks)**

➔ Summer 2011 – Total Marks 20

Q. 17 Distinguish between stack and queue with minimum four points ? **(Section 4.1.3)** **(4 Marks)**

Q. 18 Explain the concept of circular queue with example. Explain their terms overflow and underflow with respect to queue. **(Section 4.3.1)** **(8 Marks)**

Ans. :

Overflow and underflow in a queue

Queue overflow is caused by insertion in a queue that is already full. Deletion from an empty queue will cause a queue underflow.

Q. 19 Explain priority queue with example. Explain its types. **(Section 4.4)** **(8 Marks)**

➔ Winter 2011 – Total Marks 10

Q. 20 Write a brief note on queue as an abstract data type **(Section 4.2.1)** **(10 Marks)**

➔ Summer 2012 – Total Marks 12

Q. 21 Explain the concept of priority queue with suitable example. **(Section 4.4)** **(4 Marks)**

Q. 22 What is circular queue ? Explain insertion and deletion operation on circular queue. **(Sections 4.3 and 4.3.1)** **(4 Marks)**

Q. 23 Write a 'C' program to implement a queue with insert and delete operation. **(Program 4.2.1)** **(4 Marks)**

➔ Winter 2012 – Total Marks 26

Q. 24 Explain the front and rear for queue. **(Section 4.1)** **(2 Marks)**

Q. 25 Write a program in 'C' language to implement priority queue. **(Section 4.4.1)** **(8 Marks)**

Q. 26 Describe priority queue with suitable example. **(Section 4.4)** **(4 Marks)**

Q. 27 Explain queues implementation using linked list. **(Section 4.2)** **(8 Marks)**

➔ Summer 2013 – Total Marks 18

Q. 28 Describe priority queues. **(Section 4.4)** **(2 Marks)**

Q. 29 Write a program in 'C' language to implement 'circular queue'. **(Program 4.3.1)** **(8 Marks)**

Q. 30 Explain queue as an abstract data type. **(Section 4.2.1)** **(4 Marks)**

Q. 31 Explain any two applications of queue. **(Section 4.5)** **(4 Marks)**

➔ Winter 2013 – Total Marks 16

Q. 32 Define queue ? Explain how two pointer front and rear related to queue with diagram. **(Sections 4.1 and 4.1.1)** **(4 Marks)**

Q. 33 Write an algorithm to insert and delete an element from queue. **(Section 4.2)** **(4 Marks)**

Q. 34 Describe priority queue with an example ? Explain its type. **(Section 4.4)** **(4 Marks)**

Q. 35 Describe circular queue with an example. **(Section 4.3.1)** **(4 Marks)**

➔ Summer 2014 – Total Marks 24

Q. 36 Describe priority queue with example. **(Section 4.4)** **(4 Marks)**

Q. 37 Explain the concept of circular queue. **(Section 4.3.1)** **(4 Marks)**

Q. 38 Explain queue as an abstract data type, also give applications of queue. **(Sections 4.2.1 and 4.5)** **(4 Marks)**

Q. 39 Write a menu driven program to insert, delete an element in a queue and display the queue. **(Program 4.2.1)** **(8 Marks)**

➔ Winter 2014 – Total Marks 16

Q. 40 Describe the queue as abstract data type. **(Section 4.2.1)** **(4 Marks)**

Q. 41 Define circular queue. Explain insertion and deletion operation on circular queue. **(Sections 4.3 and 4.3.1)** **(4 Marks)**

Q. 42 Describe priority queue with suitable example. **(Section 4.4)** **(4 Marks)**

Q. 43 Write a procedure for inserting an element in a queue. **(Section 4.2)** **(4 Marks)**

➔ Summer 2015 – Total Marks 16

Q. 1 (B)(c) Draw and explain circular queue. *(Ans. : Refer section 4.3.1)* **(4 Marks)**

Q. 2(e) Distinguish between stack and queue with minimum 4 points. *(Ans. : Refer section 4.1.3)* **(4 Marks)**

Q. 3(b) Describe priority queue and list its advantages. *(Ans. : Refer section 4.4)* **(4 Marks)**

Q. 4(c) Write a procedure to insert an element into a queue and to delete an element from a queue. *(Ans. : Refer section 4.2)* **(4 Marks)**

➔ Winter 2015 – Total Marks 16

Q. 1(B)(c) Define the terms FRONT and REAR of queue. Enlist any four operations of queue. *(Ans. : Refer sections 4.1 and 4.2)* **(4 Marks)**

Q. 2(e) Write a 'C' program to implement a queue with insert and delete operation. *(Ans. : Refer Program 4.2.1)* **(4 Marks)**

Q. 3(b) Describe how queue works as an abstract data type. *(Ans. : Refer section 4.2.1)* **(4 Marks)**

Q. 4(c) Write the procedure for inserting and deleting an element from queue. *(Ans. : Refer section 4.2)* **(4 Marks)**

➔ Summer 2016 – Total Marks 16

Q. 1(d) Explain 'Queue Full' and 'Queue Empty' condition with suitable example. *(Ans. : Refer Example 4.3.1)* **(4 Marks)**

Q. 4(c) Explain the terms front and rear with relevance to queue. List the operations to be performed at these ends. *(Ans. : Refer sections 4.1 and 4.2)* **(4 Marks)**

Q. 5(d) With a neat sketch explain working of priority queue. *(Ans. : Refer section 4.4)* **(4 Marks)**

➔ Winter 2016 – Total Marks 16

Q. 1(B)(3) Describe circular queue. Give its advantage. *(Ans. : Refer section 4.3.1)*

Q. 2(e) Write a program to insert element in queue. *(Ans. : Refer program 4.2.1)* **(4 Marks)**

Q. 3(b) What is priority queue ? Describe working of priority queue with suitable example. *(Ans. : Refer sections 4.4 and 4.4.1)* **(4 Marks)**

Q. 4(c) Write a code delete an element in queue. *(Ans. : Refer program 4.2.1)* **(4 Marks)**

➔ Summer 2017 – Total Marks 20

Q. 1(A)(e) Sketch representation of queue as an array. *(Ans. : Refer section 4.1.1)* **(4 Marks)**

Q. 2(c) Differentiate between stack and Queue. (Min. 4 points).
(Ans. : Refer section 4.1.3) **(4 Marks)**

Q. 3(d) Describe priority queue with its advantages.
(Ans. : Refer section 4.4) **(4 Marks)**

Q. 6(a) Write a program for insert and delete operation perform on queue. State any two application of queue.
(Ans. : Refer Program 4.2.1, Section 4.5) **(8 Marks)**

➔ Winter 2017 – Total Marks 18

Q. 1(B)(c) Define following terms :

(1) Proirity Queue *(Ans. : Refer section 4.4)*

(2) Queue as an Abstract Data Type *(Ans. : Refer section 4.2.1)*

(3) empty Queue.
(Ans. : Refer section 4.1.1) **(4 Marks)**

Q. 3(b) Define circular queue. Also describe advantage of circular queue over linear queue.
(Ans. : Refer section 4.3.1) **(4 Marks)**

❑❑❑

CHAPTER 5

Unit - IV

Linked List

Syllabus Topics

Introduction to Linked List Terminologies : node, Address, Pointer, Information field/Data field, Next pointer, Null Pointer, empty list.

Type of lists : Linear list, Circular list

Operations on a singly linked list :

Traversing a singly linked list, Searching a key in linked list, Inserting a new node in a linked list, Deleting a node from a linked list.

5.1 Representation and Implementation of Singly Linked Lists

5.1.1 Comparison between Array and Linked Lists

Array data structure is simple to use and it is supported by almost all programming languages. It is very simple to understand and time to access any element from an array is constant

- Simple to use
- Simple to define
- Constant access time
- Mapping by compiler

} Properties of array data structure

An array element can be accessed by a[i], where a is the name of the array and i is the index.

Compiler maps a[i] to its physical location in memory. Address of a[i] is given by starting address of a + i* size of array element in bytes. This mapping is carried out in constant time, irrespective of which element is accessed. Array data structure suffers from some severe limitations :

- Size of an array is defined at the time of programming.
- Insertion and deletion is time consuming.
- Requires contiguous memory.

First, the size of an array has to be defined when the program is being written and its space is calculated during compilation of the program. This means that the programmer has to take a decision regarding the maximum size of data. If the actual amount of data stored is less than the maximum size, a good amount of memory will be wasted and on the other hand a larger sample of data cannot be handled. To avoid the linear cost of insertion and deletion, we need to ensure that the list is not stored contiguously, since otherwise entire parts of list will need to be moved.

☞ **Advantages of linked lists**

i) Linked list is an example of dynamic data structure. They can grow and shrink during execution of the program.

ii) Representation of linear data structure. (linear data like polynomial, stack and queue can easily be represented using linked list.

iii) Efficient memory utilization. Memory is not pre-allocated like static data structure. Memory is allocated as per the need. Memory is deallocated when it is no longer needed.

iv) Insertion and deletions are easier and efficient. Insertion and deletion of a given data can be carried out in constant time.

5.1.2 Representation

➔ **(MSBTE : W-09, W-12, S-17)**

Q. 5.1.1 What is linked list ? (Refer section 5.1.2) **W-09, W-12**

Q. 5.1.2 Define linked list with example. (Refer section 5.1.2) **S-17**

The linked list consists of a series of structures. They are not required to be stored in adjacent memory locations. Each structure consists of a data field and address field. Address field contains the address of its successors. Fig. 5.1.1 shows the actual representation of the structure.

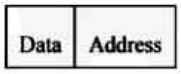


Fig. 5.1.1 : Representation of the structure

A variable of the above structure type is conventionally known as a **node**. Fig. 5.1.2 gives a representation of a linked list of three nodes.

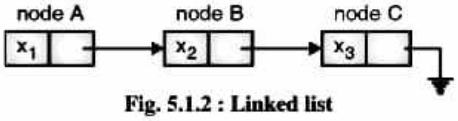


Fig. 5.1.2 : Linked list

A list consisting of three data x_1, x_2, x_3 is represented using a linked list. Node A stores the data x_1 and the address of the successor (next) node B. Node B stores the data x_2 and the address of its successor node C. Node C contains the data x_3 and its address field is grounded (NULL pointer), indicating it does not have a successor.

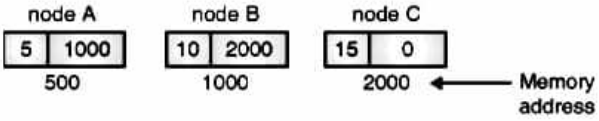


Fig. 5.1.3 : Memory representation of a linked list

Fig. 5.1.3 gives a memory representation of the linked list shown in Fig. 5.1.2. Nodes A, B and C happen to reside at memory locations 500, 1000 and 2000 respectively. $x_1 = 5$, $x_2 = 10$ and $x_3 = 15$. Node A resides at the memory location 500, its data field contains a value 5 and its address field contains 1000, which is the address of its successor node. Address field of node C contains 0 as it has no successor.

Syllabus Topic : Introduction to Linked List Terminologies

5.1.3 Implementation

➔ **(MSBTE : W-10, S-12, W-12, S-14, S-15, W-16, W-17)**

Q. 5.1.3 Define linked list terminology-node. (Refer section 5.1.3) **W-10**

Q. 5.1.4 Define the term address, and next pointer for linked list. (Refer section 5.1.3) **S-12**

Q. 5.1.5 Define the term next pointer of linked list. (Refer section 5.1.3) **W-12, S-14**

Q. 5.1.6 Define the terms pointer and NULL pointer. (Ans. : Refer sections 5.1.3 and 5.1.4) **W-17**

Structures in "C" can be used to define a node. Address of the successor node can be stored in a pointer type variable.

```
typedef struct node
{
        int data;
        struct node *next;
}node;
```

It is a **self referential structure** in "C". It is assumed that the type of data to be stored in each node is predefined to be of integer type. Next field of the structure is a pointer type variable, used for storing address of its successor node. Nodes are manipulated during run-time. A programming language must provide following facilities for run time manipulation of nodes.

- Acquiring memory for storage during run-time.
- Freeing memory during execution of the program, once the need for the acquired memory is over.

In "C" programming language, memory can be acquired through the use of standard library functions malloc( ) and calloc( ). Memory acquired during runtime can be freed through the use of library function free( ).

```
/* 1 */    node *P;
/* 2 */    P = (node *) malloc(sizeof(node));
/* 3 */    P → data = 5;
/* 4 */    P → next = NULL
```

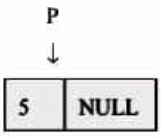


Fig. 5.1.4 : Memory for a node has been allocated during run-time. Its address is stored in the pointer P

In the program segment above, the line 1 declares a variable P with the storage class node *. It can store the address of a node that has been created dynamically. sizeof(node) in line 2 is storage requirement in number of bytes to store a node. malloc(sizeof(node)) returns the address of the allocated memory block and it is assigned to variable P. The address returned by malloc(sizeof(node)) is type casted using type casting operator(node *) before assigning it to the pointer P.

P → data = 5, in line 3 stores a value of 5 in the data field of node whose address is stored in pointer P. P → next = NULL stores a value of NULL in the next field of the node whose address is stored in the pointer P.

☞ A linked list with the address of the starting node

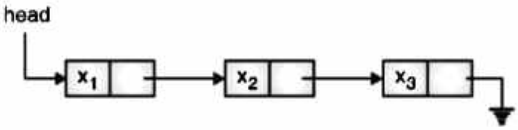


Fig. 5.1.5 : A linked list with the address of the head node

As an array is referenced by its starting address, a linked list is known by the address of its starting node. Address of the starting node is stored in a pointer variable (node * head) head. All manipulations on the linked list can be performed through the address of the starting node. Through the variable "head", first node can be accessed and through the address of the second node stored in the next field of the first node, second node can be accessed and so on.

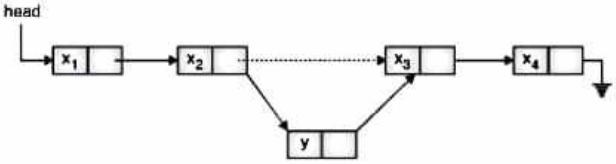


Fig. 5.1.6 : Insertion into a linked list

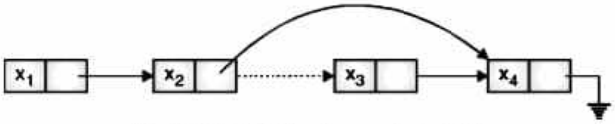


Fig. 5.1.7 : Deletion from a linked list

Insertion into a linked list requires obtaining a new node and then changing values of two pointers. The general idea is shown in Fig. 5.1.6. The dashed line represents the old pointer. It is changed to point to new node. Deletion of a node can be performed in one pointer change. Fig. 5.1.7 shows the result of deleting the node containing x_3. The next field of the node containing x_2 is changed to point to the node containing x_4. Node containing x_3 is freed using the library function free( ) subsequently.

Linked list is used to represent a list of data using dynamic data structure.

Structure of a node

```
typedef struct node
    {
        datatype data;
        struct node *next;
    }node ;
```

Set of representative function

```
Node * create () ;                  //create a linked list
int IsEmpty (node *head);           //check for empty
condition
void print (node *head);            //Display contents
node *Find (node*head, int x);      //find a value x
node *Insert (node *head, int position, int x);
                    // insert a value x
node *Delete (node * head, int position);
                    //Delete a data from the given position
Node *reverse (node *head);         // reverse a linked list
```

☞ Concept of information

Q. 5.1.7 Explain the concept of information, next, null pointer and empty list with respect to link list. (Refer section 5.1.3) **S-14**

1. Linked list is way to store the data. User can create a new place to store the data. When he writes the definition, it includes the variables containing some information. It has to be a pointer of same type.

2. For finding the information in linked list, it must start from head of list and also traverse the list sequentially until it finds the node.
3. Then it goes to the next node and accesses the information.

☞ **Definition of dynamic memory allocation**

Q. 5.1.8 Define dynamic memory allocation. State its importance. (Refer section 5.1.3) S-16

In 'C' programming also uses dynamic memory allocation. They can grow and shrink during execution of the program. Linked list is used to represent a list of data using dynamic data structure. 'C' uses standard library function malloc( ), calloc( ), realloc( ) and free( ).

☞ **Importance of dynamic memory allocation**

1. Memory gets assigned at run time hence it is called as dynamic memory allocation.
2. Memory is not allocated at the time of loading the program, it is allocated whenever required and can be de-allocated whenever not needed.

☞ **Distinguish between singly linked list and doubly linked list. (4 points)**

Q. 5.1.9 Distinguish between singly linked list and doubly linked list. (4 points) (Refer section 5.1.3) W-15

Sr. No.	Singly linked list	Doubly linked list
1.	In singly linked list we traverse in one direction.	In Doubly linked list we traverse in two directions.
2.	Each node holds one pointer fields.	Each node holds two pointer fields.
3.	In singly linked list each node contains data and link.	In doubly linked list, each node contains data link to next and link to previous one.
4.	**Application :** Stack can be implemented using SLL.	**Application :** Stack and hash table can be implemented using DLL.

Syllabus Topic : Types of Lists

5.1.4 Types of Linked List

➔ **(MSBTE : S-08, S-17)**

Q. 5.1.10 List types of linked list. (Refer section 5.1.4) S-08

Q. 5.1.11 Draw representation of linear linked list, circular linked list and doubly linked list. (Refer section 5.1.4) S-17

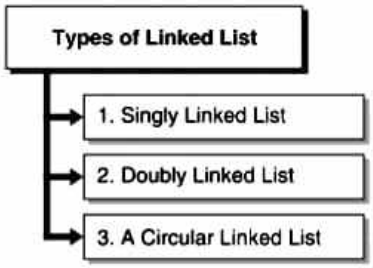


Fig. 5.1.8 : Types of linked list

Syllabus Topic : Linear List

5.1.4(A) Singly Linked List

In this type of linked list two successive nodes of the linked list are linked with each other in sequential linear manner. Movement in forward direction is possible.

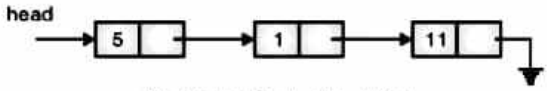


Fig. 5.1.9 : Singly linked list

5.1.4(B) Doubly Linked List

In this type of linked list each node holds two-pointer fields. In doubly linked list addresses of next as well as preceding elements are linked with current node.

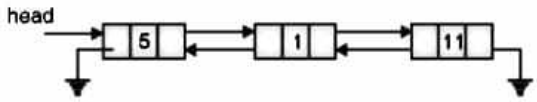


Fig. 5.1.10 : Doubly linked list

Syllabus Topic : Circular List

5.1.4(C) A Circular Linked List

➔ **(MSBTE : W-12)**

Q. 5.1.12 Describe the structure of circular linked list.
(Refer section 5.1.4(C)) W-12

In a circular list the first and the last elements are adjacent. A linked list can be made circular by storing the address of the first node in the next field of the last node.

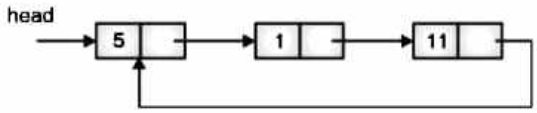


Fig. 5.1.11 : Circular linked list

Example 5.1.1 : **With diagram, define Two-way header list.** W-08

Solution :

The Fig. Ex. 5.1.1 of a two-way header list is shown below.

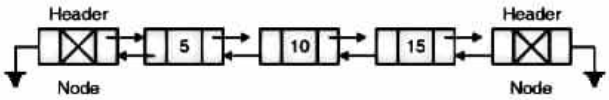


Fig. Ex. 5.1.1

A two-way header list is a doubly linked list with two header nodes, one at the beginning and other at the end. This list can be traversed easily in both forward and backward direction.

Example 5.1.2 : **Compare grounded header list and circular header list with diagram.** W -08

Solution :

A grounded header list is shown below.

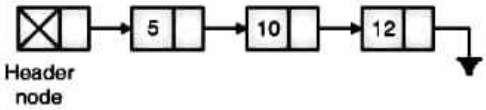


Fig. Ex. 5.1.2(a)

A circular header list is shown below :

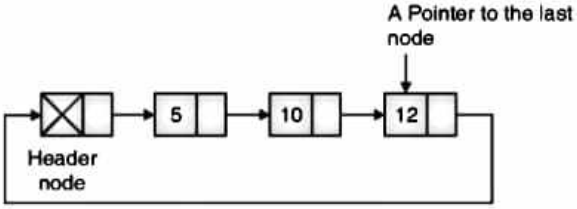


Fig. Ex. 5.1.2(b)

In a circular header list, the last node is connected back to the first node.

- In both grounded and circular list, insertion at the beginning can be carried out in constant time.
  Timing complexity = O(1).
- In a circular list, insertion at the end can be carried out in constant time. In a grounded list, insertion at the end is time consuming with timing complexity of O(n).

Syllabus Topic : NULL Pointer and Empty List

Example 5.1.3 : **Define NULL pointer and empty list.** S- 09, S-12, S-13, W-13, S-14, S-15, W-16, W-17

Solution :

A NULL pointer is a pointer containing 0 address.

P = NULL;

The above statement will set the pointer to NULL or 0 address.

An empty list has no elements in it.

L = (), represents an empty list

Syllabus Topic : Node and Pointer

Example 5.1.4 : **Define node and pointer.** W-09, S-12, W-13, S-15, W-16, W-17

OR

Define the term Node of linked list. W -12

Solution :

A node is a self referential structure. Structures in "C" can be used to define a node.

typedef struct node

```
{
    int data;
    struct node * next;
} node;
```

'data' field stores the data. For each data element, a node is acquired dynamically. Address of the successor node can be stored in the pointer type field "next".

Pointer in C-language is treated like a variable. It can store the address of any variable or address of a memory block, acquired dynamically.

☞ **Difference between Linear Linked List and Circular Linked List**

➔ **(MSBTE : S-17)**

Q. 5.1.13 Differentiate between linear linked list, circular linked list. (Refer section 5.1.4(C)) S-17

Sr. No.	Linear linked list	Circular linked list
1.	Next field of the last node is set to NULL.	Next field the last node is connected back to head node.
2.	Head pointer points to first node.	Head pointer points to last node.
3.	Insertion at the end has O(n) complexity.	Insertion at end has O(1) complexity.
4.	Searching algorithm terminates on finding the NULL pointer.	Searching terminates on coming to first node.
5.	Not suited for implementing queue data structure.	It can be used for queue data structure.

Syllabus Topic : Operations on a Singly Linked List

5.2 Basic Linked List Operations

➔ **(MSBTE : S-08, S-11, W-15, S-16)**

Q. 5.2.1 State the operations performed on linked list (Refer section 5.2) S-08

Q. 5.2.2 Explain linked list as an abstract data structure. (Refer section 5.2) S-11

Q. 5.2.3 List and define operations of linked list. (Refer section 5.2) W-15

Q. 5.2.4 Explain operations on singly linked list. (Refer section 5.2) S-16

We can treat a linked list as an abstract data type and perform following basic operations.

1. Creating a linked list.
2. Traversing the linked list.
3. Printing the link list.
4. Counting the nodes in the linked list.
5. Searching an item in the linked list.
6. Inserting an item.
7. Deleting an item.
8. Concatenating two lists.
9. Inversion.
10. Sorting of elements stored in a linked list.
11. Merging of two sorted linked list.
12. Separating a linked list in two linked lists.

5.2.1 Creating a Linked List

➢ **Program 5.2.1 : Program to create linked list.**

Q. 5.2.5 Write a 'C' program to insert a new node at the beginning into singly linked list. (Refer Program 5.2.1) W-15

```
#include<conio.h>
#include<stdio.h>
typedef struct node
{
    int data;
    struct node *next;
}node;
void main()
```

```
{
    node *HEAD,*P;
    int n,x,i;
    //no. of items to be inserted
    printf("\n no. of items :");
    scanf("%d",&n);
    //get the 1st node with its address in HEAD
    HEAD=(node*)malloc(sizeof(node));
    //read the data in 1st node
    scanf("%d",&(HEAD→ data));
    HEAD→ next=NULL;
    //HEAD points to the 1st node, while P points to the last
node
    P=HEAD; //in case of single node 1st and last node are
same
    //insert the remaining nodes
    for(i=1;i<n;i++)
    {
        P→ next=(node*)malloc(sizeof(node));
        //new node is inserted as the next node after P
        P=P→ next;
        P→ next=NULL;
        scanf("%d",&(P→ data));
    }
}
```

Let us trace the working of the above program manually. Assume n = 3 (3 nodes to be created) with inputs as 5, 1, 9.

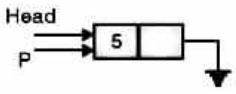


Fig. 5.2.1 : The linked list after insertion of the first item

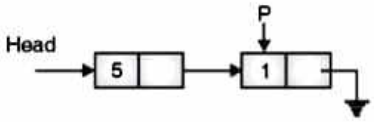


Fig. 5.2.2 : The linked list after first iteration of the "for" loop

The address of the newly acquired node is stored in the next field of the node pointed by P. Subsequently, P is moved to the next node.

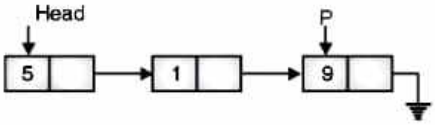


Fig. 5.2.3 : The status of the linked list after insertion of 3 elements

➢ **Program 5.2.2 : Program to create linked list through 'create' function.**

```
#include<conio.h>
#include<stdio.h>
typedef struct node
{
    int data;
    struct node *next;
}node;
node * create(int);
void main()
{
    node *HEAD;
    int n;
    HEAD = NULL;  //link list is empty
    printf("\n no. of items :");
    scanf("%d",&n);
    HEAD=create(n);  //create function returns the address of
first node
}
node * create(int n)
{
    node *head,*P;
    int i;
    head=(node*)malloc(sizeof(node));
```

```
    head→ next=NULL;
    scanf("%d",&(head→ data));
    P=head;
    //insert the remaining nodes
    for(i=1;i<n;i++)
    {
        P→ next=(node*)malloc(sizeof(node));
        //new node is inserted as the next node after P
        P=P→ next;
        scanf("%d",&(P→ data));
        P→ next=NULL;
    }
    return(head);
}
```

Syllabus Topic : Traversing a Singly Linked List

5.2.2 Traversing a Linked List

➔ **(MSBTE : W-15, W-16)**

Q. 5.2.6 Write an algorithm to traverse a singly linked list. (Refer section 5.2.2) **W-15, W-16**

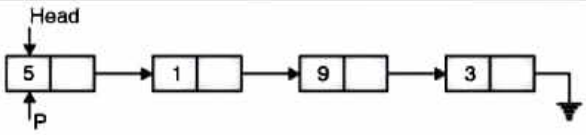


Fig. 5.2.4 : Traversal starts from the first node

Traversal of a linked list always starts from the first node. In order to traverse a linked list in the forward direction, a pointer type variable is assigned the address of the first node.

P = HEAD;

Entire list can be traversed through the following program segment.

```
P = HEAD;
while(P! = NULL)
    P = P → next; /* next field of the node pointed by P contains the address of the next node */
```

5.2.3 Counting Number of Nodes in a Linked List through Count Function

➔ **(MSBTE : S-16)**

Q. 5.2.7 Write an algorithm to count number of nodes in singly linked list.(Refer section 5.2.3) **S-16**

```
int count(node *P)
    {
        int i;
        i = 0;
        while(P ! = NULL)
        {
            i = i + 1;
            P = P → next;
        }
    return(i);
    }
```

➢ **Program 5.2.3 : Program to create linked list interactively and print the list and total number of items in the list.**

Q. 5.2.8 Write a 'C' code for creation of a linked list of 4 nodes. (Refer Program 5.2.3) **W-10**

```
#include<conio.h>
#include<stdio.h>
typedef struct node
{
    int data;
    struct node *next;
}node;
node * create(int);
void print(node *);
int count(node *);
void main()
{
    node *HEAD;
```

```
    int n,number;
    printf("\n no. of items :");
    scanf("%d",&n);
    HEAD=create(n); //create function returns the address of
first node
    print(HEAD);
    number=count(HEAD);
    printf("\n No of nodes = %d",number);
}
node * create(int n)
{
    node *head,*P;
    int i;
    head=(node*)malloc(sizeof(node));
    head→ next=NULL;
    scanf("%d",&(head→ data));
    P=head;
    //create subsequent nodes
    for(i=1;i<n;i++)
    {
        P→ next=(node*)malloc(sizeof(node));
        //new node is inserted as the next node after P
        P=P→ next;
        scanf("%d",&(P→ data));
        P→ next=NULL;
    }
    return(head);
}
void print(node *P)
{
    while(P!=NULL)
    {
        printf("← %d → ",P→ data);
        P=P→ next;
    }
}
int count(node *P)
{
    int i=0;
    while(P!=NULL)
    {
        P=P→ next;
        i++;
    }
    return(i);
}
```

Output

```
no. of items :4
12
13
14
15
← 12 → ← 13 → ← 14 → ←15 →
No of nodes = 4
```

5.2.4 Printing a List through Print Function

```
void print(node *P)
{
  while(P! = NULL)
      {
          printf("\n%d", P → data);
              P = P → next;
      }
}
```

Above function traverses a linked list and while traversing it prints the integer data stored in the node.

Syllabus Topic : Inserting a new Node in Linked List

5.2.5 Inserting an Item

➔ **(MSBTE : W-08, S-09, W-09, S-11, S-13, W-14, S-15, W-17)**

Q. 5.2.9 Write an algorithm to insert new node at the beginning, at middle position and at the end of a single linked list. (Refer section 5.2.5) **W-08, S-09, S-13**

Q. 5.2.10 Explain how insertion can be performed on linked list. (Refer section 5.2.5) **W-09**

Q. 5.2.11 Explain insertion operation in linked list with example. (Refer section 5.2.5) **S-11**

Q. 5.2.12 Write an algorithm to insert a new node at the beginning of a singly linked list. Give example. (Refer section 5.2.5) **W-14**

Q. 5.2.13 Write a 'C' program to insert a new node at the end of linear linked list. (Refer section 5.2.5) **W-17**

☞ **Inserting a new item, say x, has three situations**

1. Insertion at the front of the list.
2. Insertion in the middle of the list.
3. Insertion at the end of the list.

☞ **Algorithm for placing the new item at the beginning of a linked list**

1. Obtain space for new node.
2. Assign data to the data field of the new node.
3. Set the next field of the new node to the beginning of the linked list.
4. Change the reference pointer of the linked list to point to the new node.

☞ **Algorithm for inserting the new data after a node N1**

1. Obtain space for new node.
2. Assign value to its data field.
3. Search for the node N1.
4. Set the next field of the new node to point to N1 → next.
5. Set the next field of N1 to point to the new node.

➢ **'C' function to insert a data in a linked list after a given data**

```
node *insert(node *head, int x, int key)
 {
      /* data x is to be inserted after the key */
/* if key is – 1 then x is to be inserted as a front node */
      node *P, *q;
      /* obtain space for the new node */
      P = (node *) malloc(sizeof(node));
      /* store x in the new node */
      P → data = x;
      if(key = = – 1)
      {
          /* insert the node at the front of the list */
          P → next = head;
          head = P;
      }
      else
      {
          /* search for the key in the linked list */
          q = head;
          while(key! = q → data && q! = NULL)
              q = q → next;
          if(q ! = NULL)
          {
              /* if the key is found */
              P → next = q → next;
              q → next = P;
      }
 }
 return(head);
}
```

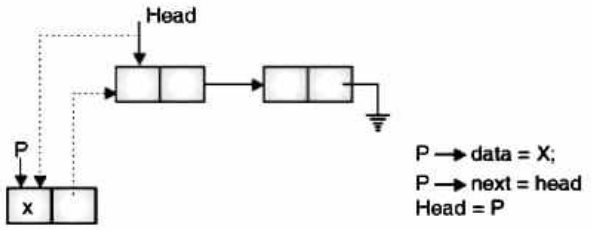


Fig. 5.2.5 : Insertion at the front

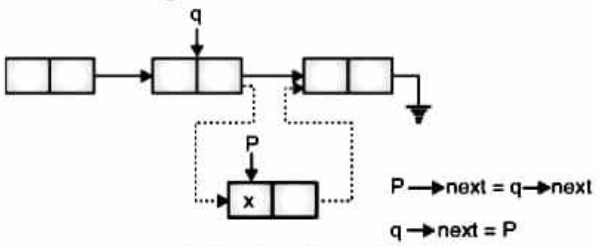


Fig. 5.2.6 : Insertion in between

5.2.5(A) Inserting an Item at the End of a Linked List

➔ **(MSBTE : S-12, S-15, W-16)**

Q. 5.2.14 Explain insertion of new node at end in singly linked list. (Refer section 5.2.5(A)) **S-12**

Q. 5.2.15 Explain insertion of new node at start and at end in singly linked list. (Refer sections 5.2.5 and 5.2.5(A)) **S-15**

Q. 5.2.16 Write an algorithm to insert a new node as the last of a singly linked list. Give example. (Refer section 5.2.5(A)) **W-16**

☞ **Algorithm**

[Insert an item 'x' in a linked list, referenced by the pointer 'head']

1. Acquire memory for new node

   i.e p = (node*) malloc(sizeof (node);

2. Assign value to the data filed and make its 'next' field 'NULL'

   i.e. P → data = x;

   P → next = NULL;

3. If 'head' is 'NULL'

   then

   [head = p ;

   goto step 6]

4. Position a pointer q on the last node by traversing the linked list from the first node and until it reaches the last node.

   i.e. q = head

   while (q → next ! = NULL)

   q = q → next ;

5. Store the address of the newly acquired node, pointed by P, in the next field of node pointed by q.

   i.e. q → next = p;

6. Stop.

➢ **'C' function for inserting an element 'x' at the end of linked list, referenced by head.**

```
node *insert_end(node *head,int x)
{
    node *p,*q;
    p=(node *)malloc(sizeof(node));
    p→ data=x;
    p→ next=NULL;
    if(head==NULL)
        return(p);
    q=head;
    while(q→ next!=NULL)
        q=q→ next;
    q→ next=p;
    return(head);
}
```

5.2.5(B) Inserting a Data 'x' at a Given Location 'LOC' in a Linked List, Referenced by 'Head'

➔ **(MSBTE : S-14)**

Q. 5.2.17 Write an algorithm to insert a node in between in a link list. (Refer section 5.2.5(B)) **S-14**

☞ **Algorithm**

1. Acquire memory for new node with its address in pointer P.

   i.e. P = (node *) malloc(sizeof(node));

2. Assign value to data field and make its 'next' field 'NULL'.

   i.e. P → data =x;

   P → next = NULL;

3. if (LOC = 1)

   then

   [insert the node, pointed by P, at the beginning of the linked list. This can be done by following steps.]

   (a) Store head in the next field of the node pointed by P

   i.e. P → next = head;

   (b) Move head to the newly connected node.

   i.e. head = P;

   go to step 7

4. [If LOC > 1]

   Position a pointer q on $(LOC - 1)^{th}$ node .

   i.e. q = head;

   for (i = 1; i < (LOC - 1); i++)

   q = q → next;

5. if q is NULL

   [then report " overflow" and terminate the algorithm.

   go to step 7]

6. Insert the node pointed by P, after the node pointed by q.

   i.e. P → next = q → next;

   q → next = P;

7. Stop.

➢ **'C' function for inserting an element x at the location 'LOC' in a linked list, referenced by head.**

```
node *insert_LOC(node *head,int x,int LOC)
{
    node *p,*q;
    int i;
    p=(node *)malloc(sizeof(node));
    p→ data = x;
    p→ next=NULL;
    if(LOC==1)
    {
        p→ next=head;
        return(p);
    }
    q = head;
    for(i=1;i<LOC-1;i++)
        if(q!=NULL)
            q=q→ next;
        else
        {
            printf("\nOverflow");
            return(head);
        }
    p→ next=q→ next;
    q→next=p;
    return(head);
}
```

5.2.5(C) Inserting an Element in a Priority Linked List

➢ **Inserting an element 'x' in a sorted list of integers, represented using a linked list. List should remain sorted after insertion.**

Or

➢ **Insert an element 'x' in a priority linked list. Elements are ordered on increasing priority.**

☞ **Algorithm**

[Existing linked list is referenced by the pointer 'head'. New data to be inserted is 'x'.]

1. Acquire memory for new node.

   i.e. P = (node *) malloc(sizeof(node));

2. Assign value to its data field and make its next field 'NULL'.

   i.e. P → data = x;

   P → next = NULL;

3. If head is NULL or x is less than the data stored in the first node (i. e. node pointed by 'head')

   then

   [insert the node P at the beginning of the linked list and terminate the algorithm]

   if (head = NULL || x < head → data)

{ P → next = head;

head = P;

go to step 5

}

4. [If head is not NULL and x > head → data]

Locate the point of insertion, as per the priority of x.

i.e.

q = head;

while (q → next ! = NULL && x > q → next → data)

q = q → next;

x should be inserted after the node pointed by q.

i.e. P → next = q → next;

q → next = P;

5. Stop.

➢ **'C' function to insert an element in priority linked list.**

```
node *insert_priority(node *head,int x)
{
    node *p,*q;
    p=(node *)malloc(sizeof(node));
    p→ data=x;
    p→ next=NULL;
    if(head==NULL || x<head→ data)
    {
        p→ next=head;
        head=p;
        return(head);
    }
    q=head;
    while(q→ next!=NULL && x>q→ next→ data)
        q=q→ next;
    p→ next=q→ next;
    q→ next=p;
    return(head);
}
```

Syllabus Topic : Deleting a Node from a Linked List

5.2.6 Deleting an Item

➔ **(MSBTE : W-09, S-10, S-11, W-13, W-14, W-17)**

Q. 5.2.18 Explain how deletion can be performed on linked list. (Refer section 5.2.6) **W-09**

Q. 5.2.19 Explain with suitable diagrams-how to delete a node from singly linked list at the beginning, in between and at end of the list. (Refer section 5.2.6) **S-10**

Q. 5.2.20 Explain deletion operation in linked list with example. (Refer section 5.2.6) **S-11**

Q. 5.2.21 Explain with suitable diagram how to delete a node from singly linked list at the start and at the end of list. (Refer section 5.2.6) **W-13**

Q. 5.2.22 Explain how to delete a node in linked list. (Refer section 5.2.6) **W-14**

Q. 5.2.23 Describe how to delete a node from linear linked list. (Refer section 5.2.6) **W-17**

Deleting a node from the list is even easier than insertion, as only one pointer value needs to be changed. Here again we have three situations.

1. Deleting the first item.
2. Deleting the last item.
3. Deleting from the middle of the list.

Algorithm for deleting the first item.

1. Store the address of the first node in a pointer variable, say P.
2. Move the head to the next node.
3. Free the node whose address is stored in the pointer variable P.

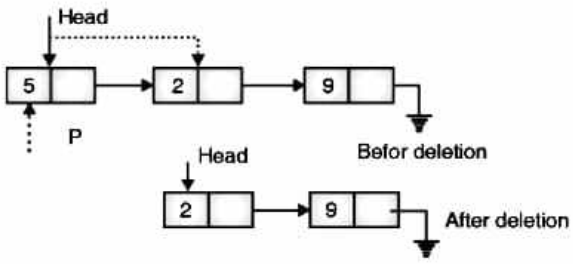


Fig. 5.2.7 : Deleting the first item

Algorithm for deleting a node from the middle of the linked list

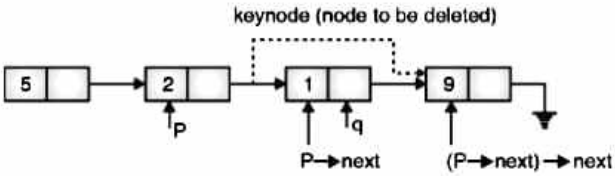


Fig. 5.2.8 : Deletion of middle element

1. Store the address of the preceding node in a pointer variable P. Node to be deleted is marked as key node.
2. Store the address of the key node in a pointer variable q, so that it can be freed subsequently.
3. Make the successor of the key node as the successor of the node pointed by P.
4. Free the node whose address is stored in the pointer variable q.

➢ **'C' function to delete a data from a linked list.**

```
node * delete(node *head, int x)/* x to be deleted from a
linked list */
{
    node *P, *q;

        if(x == head → data)
        {
            /* deleting the first item */
            P = head;
            head = head → next;
            free (P);
        }
        else
        }
    while(x ! =(P → next) → data && P → next ! = NULL)
            P = P → next;
        if(P → next ! = NULL)   /* if x exists */
        {
            q = P → next;
            P → next =(P → next) → next;
            free(q);     /* release space of the key node */
        }
    }
    return(head);
}
```

5.2.6(A) Deletion of the Last Node of a Linked List

☞ **Algorithm**

[Deleting last node of a linked list, referenced by the pointer head]

In order to delete the last node, we must position a pointer q on the last but one node. Address of the node to be deleted is stored in pointer P, So that the memory allocated to it can be freed.

1. If the first node itself is the last node then

   [make the linked list empty]

   i.e. if (head → next == NULL)

   { free (head);

   head = NULL; goto step 4

   }

2. [otherwise]

   position a pointer q on last but one node

   i.e. q = head;

   while (q → next → next ! = NULL)

   q = q → next;

3. Delete the last node

   i.e. p = q → next ;

   free (p);

   q → next = NULL;

4. Stop.

➢ **'C' function to delete last node of a linked list.**

```
node *delete_last(node *head)
{
    node *p,*q;
```

```
    if(head→ next==NULL)
    {
        free(head);
        head=NULL;
        return(head);
    }
    q=head;
    while(q→ next→ next!=NULL)
        q=q→ next;
    p=q→ next;
    free(p);
    q→ next=NULL;
    return(head);
}
```

5.2.6(B) Deletion of a Node at Location 'LOC' from a Linked List

➔ **(MSBTE : W-10)**

Q. 5.2.24 Write a 'C' code for deletion of an element from linked list. (Refer section 5.2.6(B)) **W-10**

➢ **Algorithm**

[Linked list is referenced by the pointer 'head']

1. if (LOC = =1)

   then

   [the node to be deleted is the first node This can be done by following steps]

   (a) Store the address of the first node in the pointer P

   i.e. P = head;

   (b) Move head to the next node

   i.e. head = head → next

   (c) Free the memory allocated to the node to be deleted

   i.e. free(P); goto step 5

2. [otherwise]

   position a pointer q on $(LOC - 1)^{th}$ node.

   i.e. q = head;

   for (i = 1; i<LOC-1; i++)

   q = q →next;

3. if 'q' is 'NULL'

   then

   [report underflow and terminate the algorithm goto step5]

4. Delete the desired node

   i.e. p = q → next;

   q → next = p → next;

   free (p);

5. Stop.

➢ **'C' function to delete a node at location 'LOC' from linked list.**

```
node *delete_LOC(node *head, int LOC)
{
    node *p,*q;
    int i;
    if(LOC==1)
    {
        p=head;
        head=head→ next;
        free(p);
        return(head);
    }
    q=head;
    for(i=1;i<LOC-1;i++)
        q=q→ next;
    if(q==NULL)
    {
        printf("\nUnderflow");
        return(head);
    }
    p=q→ next;
    q→ next=p→ next;
```

```
    free(p);
    return(head);
}
```

5.2.6(C) Delete a Linked List, Referenced by the Pointer Head

☞ **Algorithm**

1. if head == NULL
   then
   [go to step 3]
2. [otherwise] Delete the first node.
   i.e. P = head;
   head = head → next;
   free(P);
   goto step 1.
3. Stop.

➢ **'C' function to delete a linked list, referenced by the pointer head.**

```
node *delete_list(node *head)
{
    node *p;
    while(head!=NULL)
    {
        p=head;
        head=head→ next;
        free(p);
    }
    return(head);
}
```

5.2.7 Concatenation of Two Linked Lists

☞ **Algorithm for concatenation**

Let us assume that the two linked lists are referenced by head1 and head2 respectively.

(1) If the first linked list is empty then return head2.

(2) If the second linked list is empty then return head1.

(3) Store the address of the starting node of the first linked list in a pointer variable, say P.

(4) Move the P to the last node of the linked list through simple linked list traversal technique.

(5) Store the address of the first node of the second linked list in the next field of the node pointed by P. Return head1

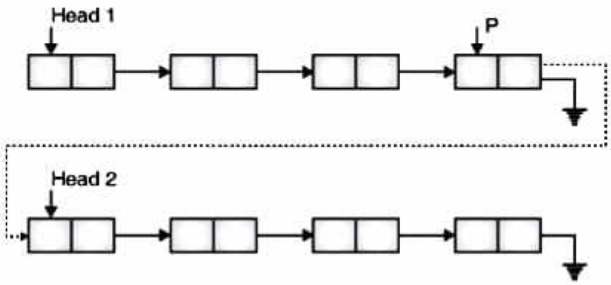


Fig. 5.2.9

➢ **'C' function to concatenate two linked lists.**

```
node * concatenate(node *head1, node *head2)
  {
      node *P;
      if(head1 == NULL)    /* if the first linked list is
empty */
          return(head2);
      if(head2 == NULL) /* if the second linked list is
empty */
          return(head1);
      P= head1;  /* place P on the first node of the first
linked list */
        while(P → next ! = NULL)    /* Move P to the last
node */
          P = P → next;
      P → next = head2;  /* address of the first node of the
second linked list stored in the last node of the first linked list
*/
      return(head1);
  }
```

5.2.8 Inversion of Linked List

A linked list can be reversed by changing the direction of the pointer, iteratively.

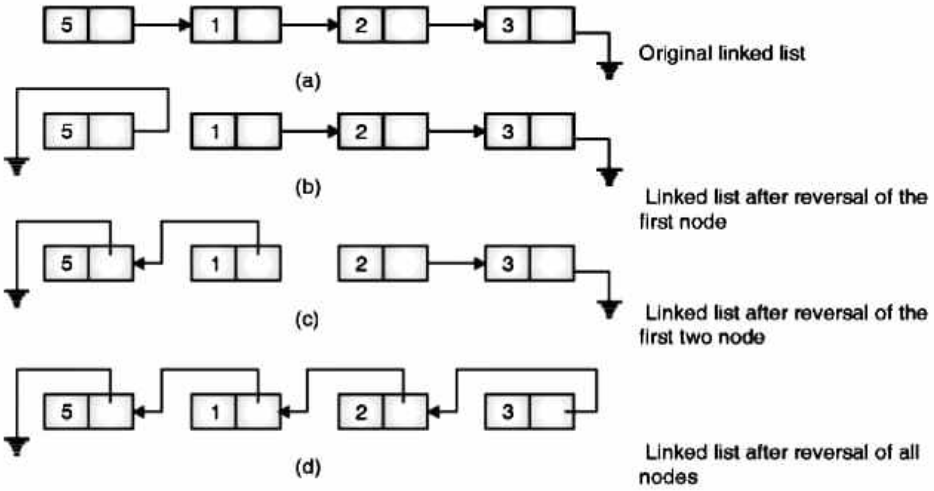


Fig. 5.2.10

☞ **Algorithm for reversing the linked list**

Let us take three pointer variables P, q and r.

P references the linked list reversed so far. q points to the linked list to be reversed. r points to the next node of q. Initially,

P = NULL; q = head; r = q → next;

```
while(all nodes have not been reversed)
{
    reverse the node pointed by q.
    q → next = P;
    move P, q, r forward by a node.
    P = q;
    q = r
    r = r → next;
}
```

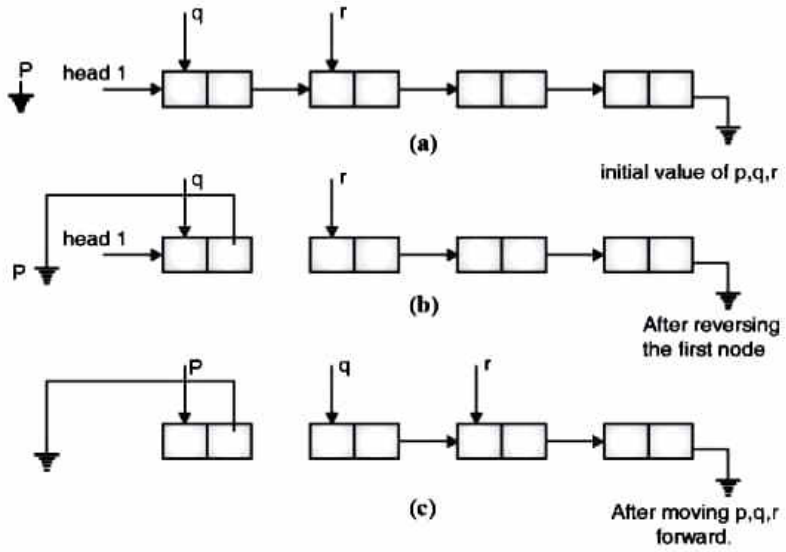


Fig. 5.2.11

➢ **'C' function for inversion.**

```
node * invert(node *head)        /* invert a linked list pointed
                                    by head */
{
    node *P ,*q ,*r;
    P = NULL; q = head; r = q → next;
                                /* initial values of P, q and r */
    while(q ! = NULL)
                            /* until all nodes have been reversed */
    {
        q → next = P;   /* move P, q, r forward by a node */
        P = q;
        q = r;
        if(r ! = NULL)
            r = r → next;
    }
  return(P);
}
```

Syllabus Topic : Searching a Key in Linked List

5.2.9 Searching a Data 'x' in a Linked List, Referenced by the Pointer Head

➔ **(MSBTE : W-09, S-12, W-12, S-13, S-14, W-14, S-17)**

Q. 5.2.25 Explain the operation on searching a desired node in the linked list. (Refer section 5.2.9) **W-09**

Q. 5.2.26 Write an algorithm for 'Search' operation in an unsorted linked list.(Refer section 5.2.9) **S-12 , W-12, W-13, W-14**

Q. 5.2.27 Explain the operation of searching a desired node in the singly linked list (Refer section 5.2.9) **S-13**

Q. 5.2.28 WAP to search an element in a link list. (Refer section 5.2.9) **S-14**

Q. 5.2.29 Write an algorithm for searching a node in linked list. (Refer section 5.2.9) **S-17**

➢ **Algorithm**

In order to search an element in a linked list, we start traversing the linked list from the first node. Traversal ends with a success if the element is found . If the element is not found then

search ends in a failure.

1. P = head;
2. if the data stored in node pointed by P is x
   then
   [end the search with success.
   i.e. return (1)]
3. Continue searching
   i.e. P = P → next.
4. if end of linked list
   then
   [end with failure
   i.e. return(0);
   ]
5. goto step 2
6. Stop.

➢ **'C' function for searching an element x in a linked list referenced by the pointer head. Function returns 1 if search ends in a success, otherwise, the function returns 0.**

```
int search1(node *head, int x)
{
   node *P;
   P=head;
   while(P!=NULL)
   {
      if(P→ data==x)
         return(1);
      P=P→ next;
   }
   return(0);
}
```

➢ **'C' function for searching an element x in a linked list referenced by the pointer head. Function returns address of the node if search ends in a success, otherwise, the function returns 'NULL'.**

```
node *search2(node *head, int x)
{
   node * P;
   P = head;
   while(P != NULL)
   {
      if (P→ data == x)
         return (P);
      P = P → next;
   }
   return(NULL);
}
```

➢ **'C' function for searching an element x in a linked list referenced by the pointer head. Function returns location of the node if search ends in a success, otherwise, the function returns '-1'.**

```
int search3(node *head, int x)
{
   int i=1;
   node *p;
   p=head;
   while(p!=NULL)
   {
      if(p→ data==x)
         return(i);
      i++;
      p=p→ next;
   }
   return(-1);
}
```

5.2.10 Searching an Element x in a Sorted Linked List

☞ **Algorithm**

[Linked list is referenced by the pointer head. Elements of the linked list are ordered in ascending order]

In order to search an element in a linked list of sorted elements, we start traversing the linked list from the first node, using a pointer P.

Search ends with success if the element is found in the node pointed by P.

Search ends in a failure under any of the given condition.

(a) P has become 'NULL'

(b) P → data is greater than x.

1. P = head
2. if P →data is equal to x

   then

   [report success and

   end the traversal]
3. if P is Null or P → data > x

   then

   [report failure and

   end the traversal]

Continue searching

i.e. P = P → next;

goto step 2.

4. Stop.

➢ **'C' function to search an element in sorted list.**

```
int search_sorted(node *head, int x)
{
    node *p;
    p=head;
    while(p!=NULL)
    {
        if(p→ data==x)
            return(1);
        if(p→ data>x)
            return(0);
        p=p→ next;
    }
    return(0);
}
```

5.2.11 Handling of Records through Linked List

Assume a singly linked list where each node contains student details like name, rollno and percentage of marks. Write a "C" function COUNT() to traverse the linked list and count how many students have obtained more than 60% marks.

Structure used to represent a node

```
typedef struct node
{
    char name [30];
    int rollno;
    float percent;
    struct node *next;
}node;
int COUNT (node *head)
{
    int n = 0;
    while (head! = NULL)
    {
        if (head → percent > 60.00)
            n++;
        head = head → next;
    }
    return (n);
}
```

5.2.12 Head Node in Linked List

Sometimes it is useful to keep an extra node at the beginning of a linked list. Such a node is also called sentinel node, a header node, or a dummy node. Fig. 5.2.12 shows a linked list with a header node.

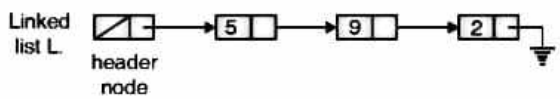


Fig. 5.2.12 : A linked list with a header node

Sentinel nodes are added to simplify algorithms/operations performed on the data structure.

Sentinel node simplifies insert operation

Insert at the front of the list is a special case. An insert operation at the beginning of list causes a change in the referencing pointer with the sentinel node, all cases of insertion become identical.

Sentinel node simplifies delete operation

Deletion from the front of a list is a special case. It changes the start of the list and hence the referencing pointer. With the sentinel node, all cases of deletion become identical.

5.3 Circular Linked List

(MSBTE : W-17)

Q. 5.3.1 With example, describe how circular linked list works when a node is deleted from beginning of list. (Refer section 5.3) **W-17**

In a circular linked list, last node is connected back to the first node. In some applications, it is convenient to use circular linked list. A queue data structure can be implemented using a circular linked list, with a single pointer "rear" as the front node can be accessed through the rear node.

Insertion of a node at the start or end of a circular linked list identified by a pointer to the last node of the linked list takes a constant amount of time. It is irrespective of the length of the linked list.

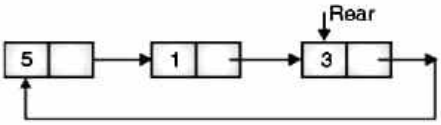
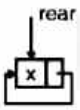


Fig. 5.3.1

Algorithm for traversing a circular list is slightly different than the same algorithm for a singly connected linked list because "NULL" is not encountered.

- **'C' function for inserting a number at the rear of a circular linked list.**

```
node * insert_rear(node * rear, int x)
{
    node *P;
    P =(node*) malloc(sizeof(node));
        /* acquire memory for the current data */
    P → data = x;
    if(rear = = NULL)
    {
        /* inserting in an empty linked list */
        rear = P;
        /* node is connected back to the same node */
        P → next = P;
        return(rear);
    }
    else
    {
        P → next = rear → next;
        rear → next = P;    /* node P is made a part of
the circle */
        rear =P
        return(rear)
    }
}
```



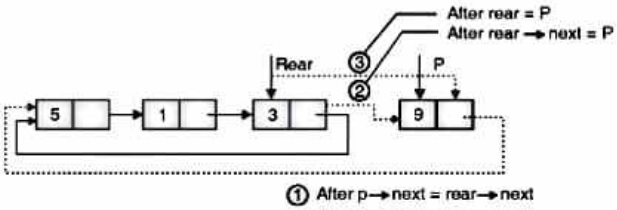


Fig. 5.3.2 : Insertion of a node at rear in circular linked list

- **'C' function for inserting a number at the front of the circular linked list.**

```
node * insert_front(node * rear, int x)
{
    node *P;
```

```
    P =(node *) malloc(sizeof(node));  /* acquire memory
*/
    P → data = x;
    if(rear == NULL) //linked list is empty
    {
        rear = P;
        P → next = P;
        return(rear);
    }
    else
    {
        P → next = rear → next;
        rear → next = P;
        return(rear);        /* rear is not moved after
  insertion */
    }
}
```

➢ **'C' function for traversing a circular linked list.**

```
void print(node * rear)
  {
    node *P;
    if(rear ! = NULL)
    {
      P = rear → next;
                        /* start traversing from the front */
    do
      {
        printf("\n%d", P → data);
        P = P → next;
      } while(P ! = rear → next);
    }
  }
```

In a circular linked list, starting and terminating cases for traversal are same. In such a case, do-while is the most suitable construct for traversing.

➢ **'C' function for counting of nodes in a circular linked list.**

```
int count (node * rear)
  {  node p;
     int n = 0;
     if (rear == NULL)
        return (0);
     p = rear→ next;
     do
     {
        n = n + 1;
        p = p → next;
     }  while (p! = rear → next);
  return (n);
  }
```

➢ **Write a 'C' function to insert and delete a Nth element in circular linked list ? Give the pictorial representation of the same.**

Insertion

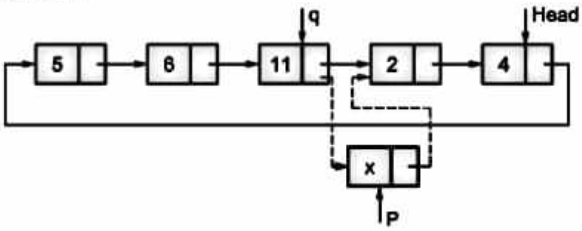


Fig. 5.3.3

q points to $(N-1)^{th}$ element. Element to be inserted is pointed by P.

➢ **'C' function for insertion.**

```
node * insert_cir_loc(node *head,int x,int loc)
{
  node *p,*q;
  int i;
```

```
    p=(node *)malloc(sizeof(node));
    p→ data=x;
    p→ next=NULL;
    if(loc==1)
        if(head==NULL)
        {
            p→ next=p;
            return(p);
        }
        else
        {
            p→ next=head→ next;
            head→ next=p;
            return(head);
        }
    q=head→ next;
    for(i=1;i<loc-1;i++)
        if(q!=head)
            q=q → next;
        else
        {
            printf("\nOverflow");
            return(head);
        }
    p→next=q→next;
    q→next=p;
    return(head);
}
```

Deletion

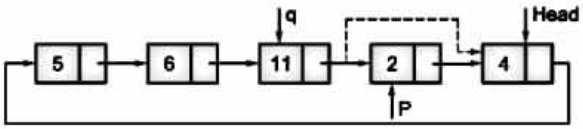


Fig. 5.3.4

q points to $(N-1)^{th}$ element. Element to be deleted is pointed by p

➢ **'C' function for deletion.**

```
node *delete_cir_loc(node *head,int loc)
{
    node *p,*q;
    int i;
    if(loc==1)
    {
        if(head→ next==head)
        {
            free(p);
            return(NULL);
        }
        else
        {
            p=head→ next;
            head→ next=p→ next;
            free(p);
            return(head);
        }
    }
    q=head→ next;
    for(i=1;i<loc-1;i++)
        if(q!=head)
            q=q→ next;
        else
        {
            printf("\nUnderflow");
            return(head);
        }
```

```
    p=q→ next;
    q→ next=p→next;
    free(p);
    return(head);
}
```

Example 5.3.1 : Write an algorithm that will concatenate two circular linked lists, producing a circular linked list.

Solution :

Two linked lists l_1 and l_2 can be Combined by manipulating the following pointers :

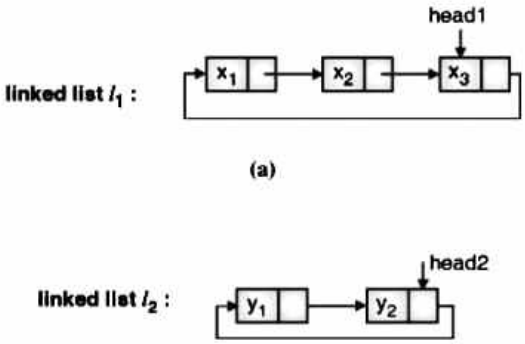
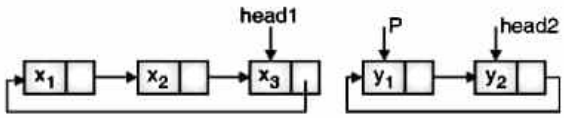
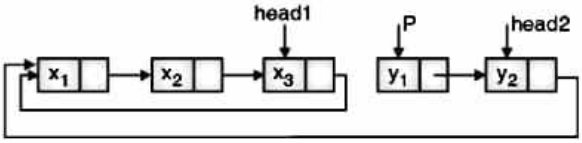
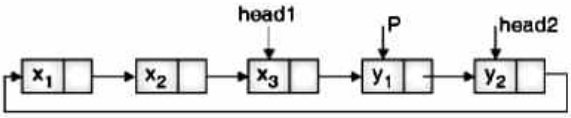
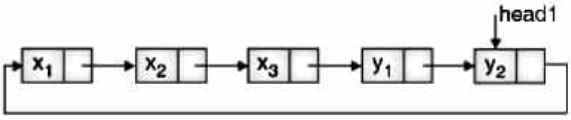


Fig. Ex. 5.3.1

Step No.	Operation	Status of l_1 and l_2
1.	P = head2 → next ;	
2.	head2 → next = head1 → next ;	
3.	head1 → next = P	
4.	head1 = head2	

The concatenated list is being referenced through head1.

5.4 MSBTE Questions and Answers

➔ Summer 2008 – Total Marks 02

Q. 1 List types of linked list and state the operations performed on linked list. **(Sections 5.1.4 and 5.2)** **(2 Marks)**

➔ Winter 2008 – Total Marks 12

Q. 2 With diagram, define Two-way Header list. **(Example 5.1.1)** **(2 Marks)**

Q. 3 Write an algorithm to insert new node at the beginning, at middle position and at the end of a single linked list. **(Section 5.2.5)** **(6 Marks)**

Q. 4 Compare a Grounded Header list and Circular Header list with diagram. **(Example 5.1.2) (4 Marks)**

➔ Summer 2009 – Total Marks 08

Q. 5 Define NULL pointer and Empty list. **(Example 5.1.3)** **(2 Marks)**

Q. 6 Write an algorithm to insert new node at the beginning, at middle position and at the end of a single linked list. **(Section 5.2.5)** **(6 Marks)**

➔ Winter 2009 – Total Marks 10

Q. 7 Define node and pointer. **(Example 5.1.4) (2 Marks)**

Q. 8 What is linked list ? Explain how insertion and deletion can be performed on linked list. **(Sections 5.1.2, 5.2.5 and 5.2.6)** **(4 Marks)**

Q. 9 Explain the operation on searching a desired node in the linked list. **(Section 5.2.9)** **(4 Marks)**

➔ Summer 2010 – Total Marks 08

Q. 10 Explain with suitable diagrams-how to delete a node from singly linked list at the beginning, in between and at end of the list. **(Section 5.2.6)** **(8 Marks)**

➔ Winter 2010 – Total Marks 12

Q. 11 Define linked list terminology-Node. Null pointer. **(Section 5.1.3 and Example 5.1.3)** **(2 Marks)**

Q. 12 Write a 'C' code for creation of a linked list of 4 nodes. **(Program 5.2.3)** **(6 Marks)**

Q. 13 Write a 'C' code for deletion of an element from linked list. **(Section 5.2.6(B))** **(4 Marks)**

➔ Summer 2011 – Total Marks 12

Q. 14 Explain linked list as an abstract data structure? **(Section 5.2)** **(4 Marks)**

Q. 15 Explain insertion and deletion operations in linked list with example. **(Sections 5.2.5 and 5.2.6)** **(8 Marks)**

➔ Summer 2012 – Total Marks 14

Q. 16 Define the term node, address, null pointer and next pointer for linked list. **(Section 5.1.3, Examples 5.1.3 and 5.1.4)(2 Marks)**

Q. 17 Write algorithm for 'Search' operation in an unsorted linked list. **(Section 5.2.9)** **(4 Marks)**

Q. 18 Explain insertion of new node at start and at end in singly linked list. **(Sections 5.2.5 and 5.2.5(A))** **(4 Marks)**

➔ Winter 2012 – Total Marks 10

Q. 19 What is linked list ? Define Node and Next pointer terminologies of linked list. **(Sections 5.1.2, 5.1.3 and Example 5.1.4)(2 Marks)**

Q. 20 Describe the structure of circular linked list. **(Section 5.1.4(C))** **(4 Marks)**

Q. 21 Write algorithm for search operation in an unsorted linked list. **(Section 5.2.9)** **(4 Marks)**

➔ Summer 2013 – Total Marks 14

Q. 22 Define NULL pointer and Empty list. **(Example 5.1.3)** **(2 Marks)**

Q. 23 Write an algorithm to insert new node at the beginning, at the middle position and at the end of a single linked list. **(Section 5.2.5)** **(8 Marks)**

Q. 24 Explain the operation of searching a desired node in the singly linked list. **(Section 5.2.9)** **(4 Marks)**

➔ Winter 2013 – Total Marks 12

Q. 25 Define the term node, address, null pointer and empty list for linked list. **(Examples 5.1.3 and 5.1.4)** **(4 Marks)**

Q. 26 Explain with suitable diagram how to delete a node from singly linked list at the start and at the end of list. **(Section 5.2.6)** **(4 Marks)**

Q. 27 Write an algorithm for search operation in unsorted singly linked list ? **(Section 5.2.9)** **(4 Marks)**

➔ Summer 2014 – Total Marks 12

Q. 28 Explain the concept of information, next, null pointer and empty list with respect to link list.

(Section 5.1.3 and Example 5.1.3) **(4 Marks)**

Q. 29 Write an algorithm to insert a node in between in a link list. **(Section 5.2.5(B))** **(4 Marks)**

Q. 30 WAP to search an element in a link list.
(Section 5.2.9) **(4 Marks)**

➔ Winter 2014 – Total Marks 12

Q. 31 Write an algorithm to insert a new node at the beginning of a singly linked list. Give example. **(Section 5.2.5)** **(4 Marks)**

Q. 32 Write an algorithm for 'search' operation in an unsorted linked list. **(Section 5.2.9)** **(4 Marks)**

Q. 33 Explain how to delete a node in linked list.

(Section 5.2.6) **(4 Marks)**

➔ Summer 2015 – Total Marks 12

Q. 2(c) Define the term :

i) Node ii) Address

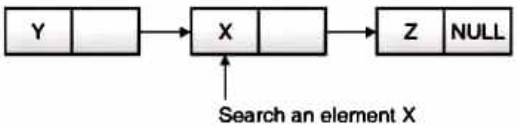
iii) Null pointer

iv) Next pointer for linked list.

(Ans. : Refer Example 5.1.4, Example 5.1.3 and section 5.1.3) **(4 Marks)**

Q. 3(c) With suitable diagram, explain 'searching' of a node in a linked list. **(4 Marks)**

Ans. : Searching an element



Q. 4(e) Explain insertion of new node at start and at end in singly linked list.
(Ans. : Refer sections 5.2.5 and 5.2.5(A))
(4 Marks)

➔ Winter 2015 – Total Marks 16

Q. 2(c) List and define operations of linked list.
(Ans. : Refer section 5.2) **(4 Marks)**

Q. 3(c) Distinguish between singly linked list and doubly linked list. (4 points)
(Ans. Refer section 5.1.3) **(4 Marks)**

Q. 4(d) Write an algorithm to traverse a singly linked list. *(Ans. : Refer section 5.2.2)* **(4 Marks)**

Q. 4(e) Write a 'C' program to insert a new node at the beginning into singly linked list.
(Ans. : Refer program 5.2.1) **(4 Marks)**

➔ Summer 2016 – Total Marks 16

Q. 1(e) Define dynamic memory allocation. State its importance.
(Ans. Refer section 5.1.3) **(4 Marks)**

Q. 2(d) Write an algorithm to count number of nodes in singly linked list.*(Ans. : Refer section 5.2.3)*
(4 Marks)

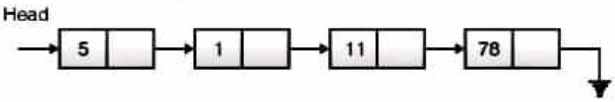
Q. 6(a) Explain operations on singly linked list.
(Ans. : Refer section 5.2) **(8 Marks)**

➔ Winter 2016 – Total Marks 12

Q. 2(c) Write an algorithm to insert a new node as the last of a singly linked list. Give example.
(Ans. : Refer section 5.2.5(A)) **(4 Marks)**

Ans. : For example : Consider linked list as

After inserting 78 at the end of list it becomes



Q. 4(d) Define following terms :

i) Node ii) Null pointer

iii) Empty list iv) Information

(Ans. : Refer Examples 5.1.4 and 5.1.3)
(4 Marks)

Ans. :

iv) Information

Information is stored in the data field of a node.

Q. 4(e) Write an algorithm to traverse a singly linked list. *(Ans. : Refer section 5.2.2)* **(4 Marks)**

➔ Summer 2017 – Total Marks 18

Q. 1(A)(f) Define linked list with example.
(Ans. : Refer section 5.1.2) **(2 Marks)**

Q. 2(d) Draw representation of linear linked list, circular linked list and doubly linked list.
(Ans. : Refer section 5.1.4) **(4 Marks)**

Q. 3(e) Write an algorithm for searching a node in linked list. *(Ans. : Refer section 5.2.9)* **(4 Marks)**

Q. 5(c) Differentiate between linear linked list, circular linked list.
(Ans. : Refer section 5.1.4(C)) **(8 Marks)**

➔ Winter 2017 – Total Marks 18

Q. 1(A)(d) Define the terms pointer and NULL pointer.
(Ans. : Refer section 5.1.3 and 5.1.4) **(2 Marks)**

Q. 3(c) Describe how to delete a node from linear linked list. *(Ans. : Refer section 5.2.6)* **(8 Marks)**

Q. 4(d) With example, describe how circular linked list works when a node is deleted from beginning of list. *(Ans. : Refer section 5.3)* **(4 Marks)**

Q. 4(e) Write a 'C' program to insert a new node at the end of linear linked list.
(Ans. : Refer section 5.2.5) **(4 Marks)**

❑❑❑

CHAPTER 6

Unit - V

Trees

Syllabus Topics

Terminologies : tree, degree of a node, degree of a tree, level of a node, leaf node. Depth/Height of a tree, In-degree and Out-Degree, Path, Ancestor and descendant nodes

Tree Types and Traversal methods Types of Trees : General tree, Binary tree, Binary search tree (BST).

Binary tree traversal : In order traversal, Preorder traversal, Post order traversal

Expression tree.

6.1 Introduction to Trees

➔ **(MSBTE : S-12, S-13, S-15, W-15, W-17)**

Q. 6.1.1	**What is tree ? (Refer Section 6.1)**	**S-12, S-15**
Q. 6.1.2	**Define the term tree (Refer Section 6.1)**	**S-13, W-15**
Q. 6.1.3	**Define tree. (Refer section 6.1)**	**W-17**

A tree is a collection of elements called "nodes", one of which is distinguished as a root say r, along with a relation "parenthood" that places a hierarchical structure on the nodes. The root can have zero or more nonempty subtrees T_1, T_2, ... T_k each of whose roots are connected by a directed edge from r.

The root of each subtree is said to be a child of r and r is parent of each subtree root. Fig. 6.1.1 is a typical tree using the recursive definition.

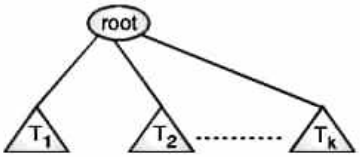


Fig. 6.1.1 : Generic tree

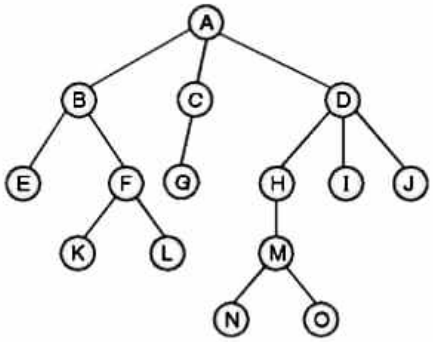


Fig. 6.1.2 : A tree

Syllabus Topic : Degree of a Node, Degree of a Tree, Level of a Node, Leaf Node, Depth/Height of a Tree, In-degree and Out-Degree, Path, Ancestor and Descendant Nodes

6.1.1 Basic Terms

➔ **(MSBTE : W-08, S-09, W-09, S-11, S-12, S-13, W-13, S-14, W-14, S-15, W-15, S-16, S-17, W-17)**

Q. 6.1.4	**Define the terms related to Binary Tree : Level, Depth, Leaf Node, Root Node. (Refer Section 6.1.1)**	**W- 08, W-14**
Q. 6.1.5	**Describe in brief, the terms related to Binary Tree: Level, Depth, Leaf Node, Root Node, Siblings. (Refer Section 6.1.1)**	**S-09, S-13, W-15**

Q. 6.1.6 Explain the terms with the help of diagram.
(i) Siblings (ii) Leaf Node
(Refer Section 6.1.1) **W- 09, W-14, S-15**

Q. 6.1.7 Explain any eight terms of tree terminology.
(Refer Section 6.1.1) **S-11**

Q. 6.1.8 Define any four terminologies related to tree.
(Refer Section 6.1.1) **S-12, S-15**

Q. 6.1.9 Define the given tree terminology :
(1) Degree of node
(Refer Section 6.1.1) **S-14, W-17**
(2) Leaf node
(Refer Section 6.1.1) **W-13, S-14**

Q. 6.1.10 Explain level of a node w.r.t tree.
(Refer Section 6.1.1) **S-14**

Q. 6.1.11 Explain height of a tree.
(Refer Section 6.1.1) **S-14, W-14, W-17**

Q. 6.1.12 Define the following with respect to tree :
(i) Ancestor
(ii) Depth (Refer section 6.1.1)
(iii) Degree (Refer section 6.1.1)
(iv) Path (Refer section 6.1.1) **S-16**

Q. 6.1.13 Define :
i) Sibling (Refer section 6.1.1)
ii) Depth of tree (Refer section 6.1.1)
iii) Degree of tree. (Refer section 6.1.1) **S-17**

- **Root :** It is a special node in a tree structure and the entire tree is referenced through it. This node does not have a parent. In the tree of Fig. 6.1.2, the root is A.

- **Parent :** It is an immediate predecessor of a node. In the Fig. 6.1.2, A is the parent of B, C, D.

- **Child :** All immediate successors of a node are its children. In the Fig.6.1.2, B, C and D are the children of A.

- **Siblings :** Nodes with the same parent are siblings. H, I and J are siblings as they are children of the same parent D.

- **Path :** It is number of successive edges from source node to destination node. The path length from node D to node N in Fig.6.1.2 is 3. Node D is connected to N through three edges D-H, H-M, M-N.

- **Degree of a node**

The degree of a node is a number of children of a node. In Fig. 6.1.2, A and D are nodes of degree 3, B, F and M are nodes of degree 2, C and H are nodes of degree 1 and E, K, L, N, O, I, J are nodes of degree 0. A node of degree 0 is also known as the **leaf node.** A **leaf node** is a terminal node and it has no children.

- **Leaf node**

A node of degree 0 is also known as a leaf node. A leaf node is a terminal node and it has no children. In the Fig. 6.1.2 E, K, L, G, N, O, I, J are leaf nodes.

- **Level**

The level of a node in a binary tree is defined as :

1. The root of the tree has level 0
2. The level of any other node in the tree is one more than the level of its father.

The levels of various nodes in the tree of Fig. 6.1.2 is given below.

Nodes	Level
A	0
B, C, D	1
E, F, G, H, I, J	2
K, L, M	3
N, O	4

- **Depth**

The depth of a binary tree is the maximum level of any node in the tree. This is equal to the node in the tree. This is equal to the longest path from the root to any leaf node. The depth of the tree in Fig. 6.1.2 is 4.

☞ **Degree of a tree**

The degree of a tree is the maximum degree of a node in the tree. For example, the degree of the tree shown in the Fig. 6.1.2 is 3. The node D and node A have maximum degree and it is 3.

☞ **Depth/Height of a tree**

Height of a tree is the distance of the root node from its farthest descendant. Height of the tree shown in the Fig. 6.1.2 is 4. The distance of the root node A from its farthest descendant N is 4.

☞ **Ancestor/descendant node**

All the nodes on the path leadning to root node are ancestors of a node. In the Fig. 6.1.2, ancestors of the node N are (M, H, D, A). all the nodes in the subtress of a node are its descendants. In the Fig. 6.1.2, the descendants of node D are (H, I, J, M, N, O).

Syllabus Topic : Binary Tree

6.2 Binary Tree

➔ **(MSBTE : W-08, W-10, S-11, S-12, W-12, W-13, S-14, S-15, W-16, W-17)**

Q. 6.2.1	**Explain the binary tree with suitable example and diagram. (Refer Section 6.2)**	**W-08, W-12**
Q. 6.2.2	**Write an algorithm for binary tree along with an example. (Refer Section 6.2)**	**W-10**
Q. 6.2.3	**Explain Binary tree with an example ? (Refer Section 6.2)**	**S-11**
Q. 6.2.4	**What is Binary tree ? (Refer Section 6.2)**	**S-12, W-13, S-15, W-13, W-16, W-17**
Q. 6.2.5	**Describe the concept of binary tree and its application. (Refer Section 6.2)**	**S-14, W-16**

A tree is binary if each node of the tree can have maximum of two children. Moreover, children of a node of binary tree are ordered. One child is called the "left" child and the other is called the "right" child. An example of binary tree is shown in Fig. 6.2.1.

Node A has two children B and C. Similarly, nodes B and C, each have one child namely G and D respectively.

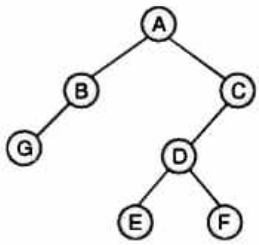


Fig. 6.2.1 : A binary tree

6.2.1 Representation of a Binary Tree using an Array

➔ **(MSBTE : W-10, W-13, S-16)**

Q. 6.2.6	**Describe with an example sequential representation of trees in memory. (Refer Section 6.2.1)**	**W-10**
Q. 6.2.7	**Describe representation of binary tree. (Refer Section 6.2.1)**	**W-13, S-16**

In order to represent a tree in a single one-dimensional array, the nodes are numbered sequentially level by level left jto right. Even empty nodes are numbered.

When the data of the tree is stored in an array then the number appearing against the node will work as indices of the node in an array.

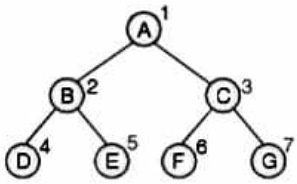


Fig. 6.2.2 : Numbering of nodes

0	1	2	3	4	5	6	7
7	A	B	C	D	E	F	G

Location number zero of the array can be used to store the size of the tree in terms of total number of nodes (existing or not existing).

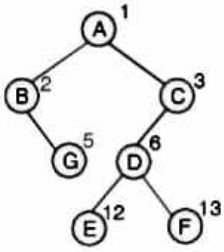


Fig. 6.2.3 : Numbering of nodes with some non-existing children

0	1	2	3	4	5	6	7	8	9	10	11	12	13
13	A	B	C	\0	G	D	\0	\0	\0	\0	\0	E	F

Fig. 6.2.4 : Array representation of the tree shown in Fig. 6.2.3

All non-existing children are shown by "\0" in the array.

Index of the left child of a node $i = 2 \times i$

Index of the right child of a node $i = 2 \times i + 1$

Index of the parent of a node $i = i/2$

Sibling of a node i will be found at the location $i + 1$, if i is a left child of its parent. Similarly, if i is a right child of its parent then its sibling will be found at $i - 1$.

Example 6.2.1 : Explain array representation of binary trees using the following.

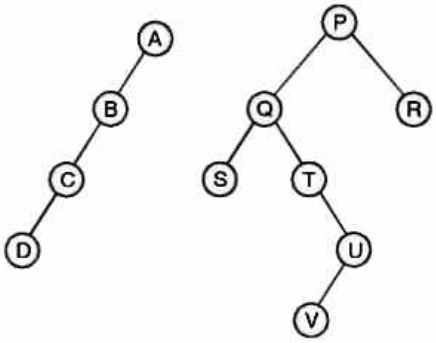


Fig. Ex. 6.2.1 : State and explain limitations of this representation

Solution :

In order to represent a tree in a single one-dimensional array, the nodes are numbered sequentially level by level from left to right. Even empty nodes are numbered. Location number zero of the array can be used to store the size of the tree in terms of total number of nodes (existing or not existing).

Step 1 : Numbering of nodes

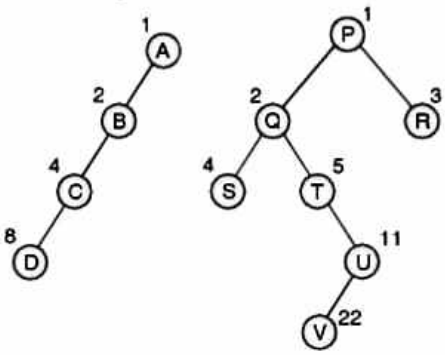


Fig. Ex. 6.2.1(a)

Step 2 :

Representation :

Left tree

0	1	2	3	4	5	6	7	8	
15	A	B		C				D	.....

Right tree

0	1	2	3	4	5		11		22	
31	P	Q	R	S	T	....	U	.....	V	.....

Fig. Ex. 6.2.1(b)

Array representation is less efficient for a sparse tree. In array representation, memory is allocated even for empty nodes. Array representation also suffers from the problem of underflow. The size of the array is fixed during compile time.

Example 6.2.2 : Define a binary tree. Show the sequential representation of the binary tree given.

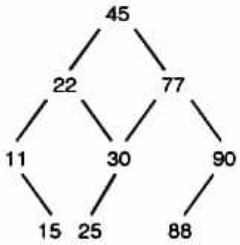


Fig. Ex. 6.2.2

Solution :

A tree is binary if each node of the tree can have maximum of two children. Moreover, children of a node of binary tree are ordered.

One child is called "left" child and the other is called the "right" child.

Array representation of the given tree

Step 1 : Numbering of nodes :

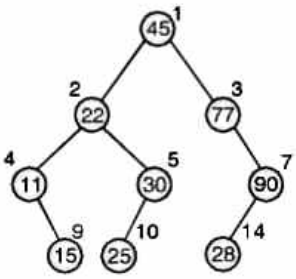


Fig. Ex. 6.2.2(a)

Step 2 : Representation

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
15	45	22	77	11	30		90		15	25				28	

Fig. Ex. 6.2.2(b)

6.2.2 Linked Representation of a Binary Tree

➔ **(MSBTE : W-10, W-13)**

Q. 6.2.8	**Describe with an example linked representation of trees in memory. (Refer Section 6.2.2)**	**W-10**
Q. 6.2.9	**Describe representation of binary tree. (Refer Section 6.2.2)**	**W-13**

Linked representation of a binary tree is more efficient than array representation. A node of a binary tree consists of three fields as shown below :

- Data
- Address of the left child
- Address of the right child

Left	Data	Right

Left and right are pointer type fields. Left holds the address of left child and right holds the address of right child.

In "C" language, the following structure can be defined to represent one node of a binary tree. It is assumed that a node contains an integer data.

```
typedef struct node
  {  int data;
     struct node *left;
     struct node *right;
  }node;
```

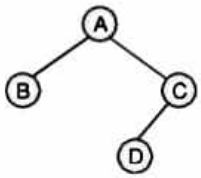


Fig. 6.2.5 : A sample binary tree

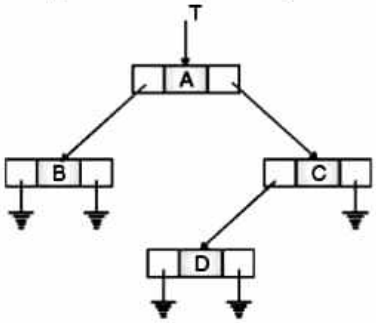


Fig. 6.2.6 : Linked representation of the binary tree of Fig. 6.2.5

The level of root node is always 0. The immediate children of the root are at level 1 and their immediate children at level 2 and so on.

Height of a node : Height of a node is the distance of the node from its farthest descendant. Height of the node C is 2 as its distance from the descendant H is 2.

Nodes	Height
A	3
C	2
B, F	1
D, E, H, G	0

Height of various nodes of the tree shown in Fig. 6.2.7.

Height of a tree : Height of a tree is height of the root node. Height of the tree shown in Fig. 6.2.7 is 3 as the height of the root node A is 3.

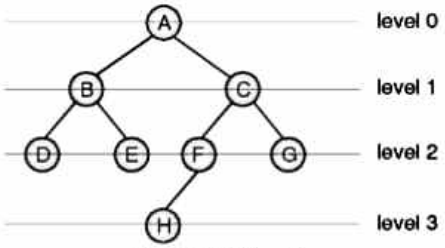


Fig. 6.2.7 : Levels

Syllabus Topic : General Tree

6.3 General Tree

➔ **(MSBTE : W-13, W-16)**

Q. 6.3.1	**Define the term general tree. (Refer Section 6.3)**	**W-13**
Q. 6.3.2	**Describe general tree (Refer section 6.3)**	**W-16**

In a general tree, number of children per node is not limited to two. Since, the number of children per node can vary greatly, it might not be feasible to make the children direct links in the node. Children of a node can be stored in a linked list with parent node storing the address of its leftmost child.

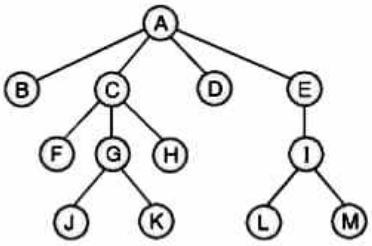


Fig. 6.3.1 : A tree

6.3.1 Node Declaration for a Tree

➔ **(MSBTE : S-13, W-15, W-17)**

Q. 6.3.3	**Define the forest. (Refer Section 6.3.1)**	**S-13**
Q. 6.3.4	**Compare between General Tree and Binary Tree (any 4 points). (Refer Section 6.3.1)**	**W-15, W-17**

```
typedef struct tnode
  { int data; /* assuming tree stores integer data */
    struct tnode * firstchild;
    struct tnode * nextsibling;
  } tnode;
```

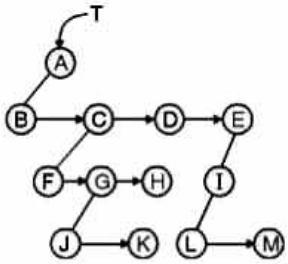


Fig. 6.3.2 : Leftmost child right sibling representation of the tree shown in Fig. 6.3.1

Each node in Fig. 6.3.2 has exactly two links. The first pointer points to its leftmost child and the other pointer points to its next sibling. For example, the first pointer of the node A, points to its leftmost child B. As the root does not have a sibling, its next pointer is NULL. As the node B does not have a child, its first pointer is NULL and its next pointer points to its sibling C.

The tree of Fig. 6.3.2 is obtained from the general tree of Fig.6.3.1 using 'leftmost child right sibling' relation. The tree of Fig. 6.3.2, does not look like a binary tree, but, if it is rotated by 45°, it will certainly look like a binary tree. The tree of Fig. 6.3.2(a) is obtained by rotating the tree of Fig. 6.3.2 by 45°.

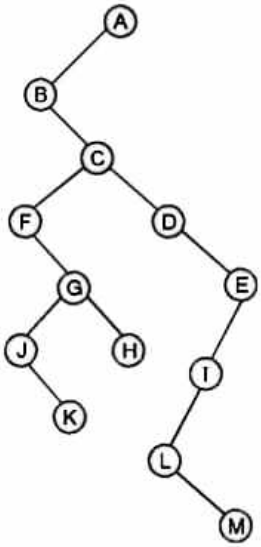


Fig. 6.3.2(a) : Tree of Fig. 6.3.2 is redrawn after it is rotated by 45°

A **forest** is defined as a set of trees. A forest can be represented by a binary tree. It should be clear that the right child of an equivalent binary tree (see Fig. 6.3.2(a)) will always be null. A root does not have a sibling. When a forest is transformed into a binary tree, root will have a right child. Right child of the root will be next tree in a forest. Consider a forest with three trees, as shown in the Fig. 6.3.2(b).

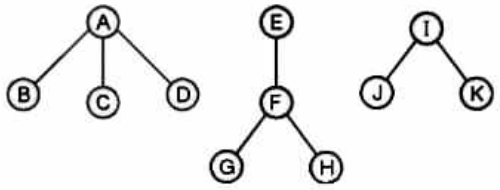


Fig. 6.3.2(b) : A forest containing three trees

Each tree in forest is converted to a binary tree using 'leftmost child right sibling relation'. It is shown in Fig. 6.3.2(c)

Each tree of forest in Fig. 6.3.2(b) is represented by its corresponding binary tree.

Three binary trees of Fig. 6.3.2(c) can be combined by :

(1) Tree with root node E is the right child of node A.

(2) Tree with root node I is the right child of node E.

Final tree is shown in Fig. 6.3.2(d).

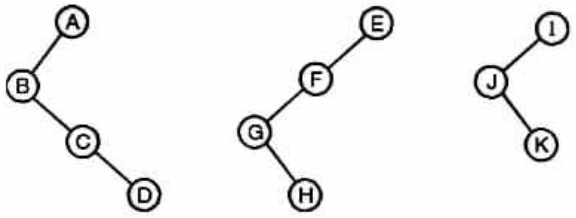


Fig. 6.3.2(c)

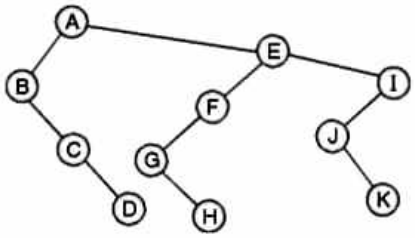


Fig. 6.3.2(d) : Binary tree representation of forest in Fig. 6.3.2(b)

Example 6.3.1 : What is the necessity of converting a tree into binary tree ? Given the following tree :

Fig. Ex. 6.3.1

Covert it into a binary tree and list down the steps for the same.

Solution :

Algorithms for binary tree are simple and widely used. Thus it is easy to give a formal treatment to a binary tree. A tree can be converted into a binary tree using leftmost child right sibling relation.

- The left pointer points to the leftmost child.
- The right pointer points to its next sibling.

Equivalent binary tree

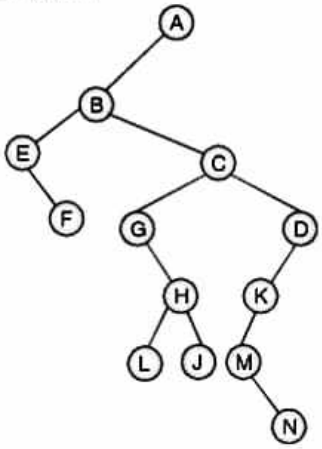


Fig. Ex. 6.3.1(a)

☞ **Comparison between general tree and binary tree**

Sr. No.	General Tree	Binary Tree
1.	Any number of children for a node.	Maximum of two children per node.
2.	May not be possible to store addresses of children directly in a node.	It is possible to store address of children in a node.
3.	Tree representation is difficult.	Tree representation is easy.
4.	Difficult to write algorithms for general tree.	There are standard algorithms.

Q. 6.3.5 Write a program to count the number of node in a binary search tree. W-14

```
int count(mode * T)
{
```

```
    if(T==NULL)
        return(0)
    return(1 + count (T → left) + count (T → right));
}
```

6.4 Types of Binary Tree

➔ (MSBTE : S-09, S-14)

Q. 6.4.1 Enlist the types of Binary Tree and give their meaning. (Refer Section 6.4) S-09

Q. 6.4.2 List types of trees and explain any one. (Refer Section 6.4) S-14

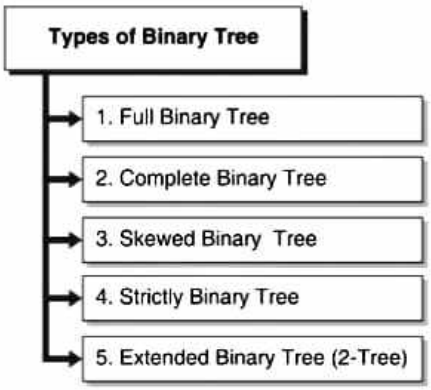


Fig. 6.4.1 : Types of binary tree

6.4.1 Full Binary Tree

➔ (MSBTE : S-13)

Q. 6.4.3 Define the term fully binary tree. (Refer Section 6.4.1) S-13

A binary tree is said to be full binary tree if each of its node has either two children or no child at all. Every level is completely filledup. Number of node at any level i in a full binary tree is given by 2^i. A full binary tree is shown in Fig. 6.4.2(a).

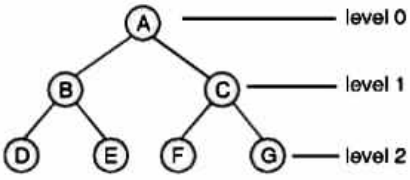


Fig. 6.4.2(a) : Full binary tree of depth 3

Total number of nodes in a full binary tree of height $h = 2^0 + 2^1 + 2^2 \ldots + 2^h = 2^{h+1} - 1$

The height of the tree shown is Fig. 6.4.2(a) is 2 and number of nodes is given by $2^{2+1} - 1 = 8 - 1 = 7$.

Hence number of nodes $n = 2^{h+1} - 1$

Or $2^{h+1} = n + 1$

Taking log on both sides.

$$h + 1 = \log_2 (n + 1)$$

$$\therefore h = (\log_2 (n + 1)) - 1$$

Tournament tree is an example of full binary tree.

6.4.2 Complete Binary Tree

➔ (MSBTE : W-10, S-13, S-17)

Q. 6.4.4 Describe complete binary tree along with an example. (Refer Section 6.4.2) W-10

Q. 6.4.5 Define the term 'complete binary tree' (Refer Section 6.4.2) S-13, S-17

A complete binary tree is defined as a binary tree where

(i) All leaf nodes are on level n or n – 1.

(ii) Levels are filled from left to right.

Examples of a complete binary tree are shown in Fig. 6.4.2(b).

Heap is an example of complete binary tree.

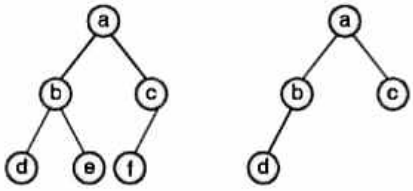


Fig. 6.4.2(b) : Complete binary trees

6.4.3 Skewed Binary Tree

Trees in of Fig. 6.4.2(c) are examples of skewed binary trees. A skewed binary tree could be skewed to the left or it could be skewed to the right.

In a left skewed binary tree, most of the nodes have the left child without corresponding right child.

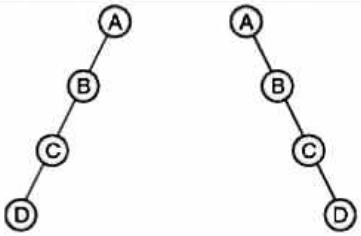


Fig. 6.4.2(c) : Skewed Binary trees

Similarly, in a right skewed binary tree, most of the nodes have the right child without a corresponding left child.

Binary search tree could be an example of a skewed binary tree.

6.4.4 Strictly Binary Tree

If every non-terminal node in a binary tree consists of non-empty left subtree and right subtree, then such a tree is called strictly binary tree. In other words, a node will have either two children or no child at all. Fig. 6.4.2(d) shows a strictly binary tree.

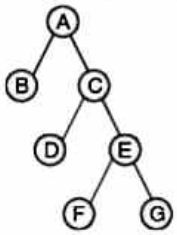


Fig. 6.4.2(d) : Strictly binary tree

6.4.5 Extended Binary Tree (2-Tree)

➔ **(MSBTE : W-10)**

Q. 6.4.6 Describe extended binary tree along with an example. (Refer Section 6.4.5) **W-10**

In an extended tree, each empty subtree is replaced by a failure node. A failure node is represented by □. Nodes with 2 children are called internal nodes, and the nodes with 0 children are called external nodes. Any binary tree can be converted into a extended binary tree by replacing each empty subtree by a failure node.

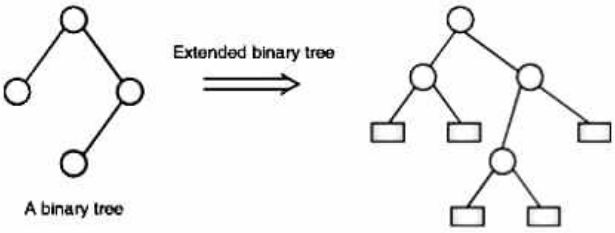


Fig. 6.4.2(e) : Extended binary tree

Syllabus Topic : Binary Tree Traversal

6.5 Binary Tree Traversal

➔ **(MSBTE : S-08, W-10, S-11, W-11, S-15, W-15)**

Q. 6.5.1 Give recursive algorithms for pre-order, in-order and post-order, traversal of Binary Tree. (Refer Section 6.5) **S-08**

Q. 6.5.2 Enlist operations on trees. (Refer Section 6.5) **W-10**

Q. 6.5.3 Define tree traversal along with Preorder, Inorder and Post order algorithms . (Refer Section 6.5) **S-11**

Q. 6.5.4 Explain Inorder, Preorder and Postorder traversal. (Refer Sections 6.5, 6.5.1, 6.5.2 and 6.5.3) **W-11, S-15**

Q. 6.5.5 List operations on trees. (Refer section 6.5) **S-15**

Q. 6.5.6 Define tree traversal. List different types of tree traversal. (Refer sections 6.5, 6.5.1, 6.5.2 and 6.5.3) **W-15**

Most of the tree operations require traversing a tree in a particular order. Traversing a tree is a process of visiting every node of the tree and exactly once. Since, a binary tree is defined in a recursive manner, tree traversal too could be defined recursively. For example, to traverse a tree, one may visit the root first, then the left subtree and finally traverse the right subtree. If we impose the restriction that left subtree is visited before the right subtree then three different combination of visiting the root, traversing left subtree, traversing right subtree is possible.

1. Visit the root, traverse, left subtree, traverse right subtree.
2. Traverse left subtree, visit the root, traverse right subtree.
3. Traverse left subtree, traverse right subtree, visit the root.

These three techniques of traversal are known as preorder, inorder and postorder traversal of a binary tree.

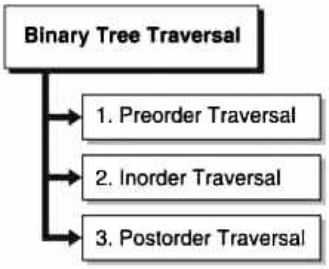


Fig. 6.5.1 : Binary Tree Traversal

Syllabus Topic : Preorder Traversal

6.5.1 Preorder Traversal (Recursive)

➔ **(MSBTE : S-08, S-10, S-11, S-15, W-15, S-17, W-17)**

Q. 6.5.7 State the principles of Pre-order traversal of Binary Tree. (Refer Section 6.5.1) **S-08**

Q. 6.5.8 Write an algorithm to traverse the tree in preorder with example. (Refer Section 6.5.1) **S-10**

Q. 6.5.9 Define tree traversal along with Preorder algorithms. (Refer Section 6.5.1) **S-11, W-17**

Q. 6.5.10 With suitable example, describe preorder traversal of tree. Also write algorithm for preorder traversal. (Refer section 6.5.1) **W-17**

Q. 6.5.11 Describe the process of pre-order traversal of binary tree. Give one example. (Refer section 6.5.1) **W-15**

Q. 6.5.12 Write algorithm for preorder traversal of binary tree. (Refer section 6.5.1) **S-17**

The functioning of preorder traversal of a non-empty binary tree is as follows :

1. Firstly, visit the root node (visiting could be as simple as printing the data stored in the root node).
2. Next, traverse the left subtree in preorder.
3. At last, traverse the right-subtree in preorder.

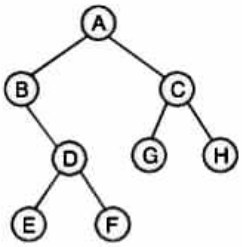
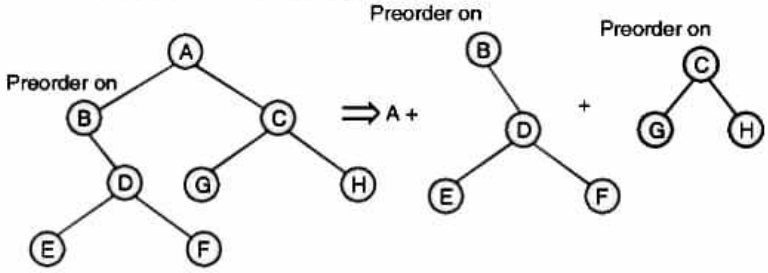


Fig. 6.5.2 : A sample binary tree

Stepwise preorder traversal of tree is shown in Fig. 6.5.3 is given below :



⟹ A + (B + Preorder on D) + (C + Preorder on G + Preorder on H)

⟹ A + (B + (D + Preorder on E + Preorder on F)) + (CGH)

⟹ A + (BDEF) + (CGH)

⟹ ABDEFCGH

Fig. 6.5.3

➢ **'C' Function for Preorder Traversal**

```
void preorder (node * T)  /* address of the root node is passed in T */
{
    if (T! = NULL)
    {
        printf ("\n%d", T → data);  /* visit the root */
        preorder (T → left);        /* preorder traversal on left subtree  */
        preorder (T → right);       /* preorder traversal on right subtree  */
    }
}
```

In words we could say, "visit" a node, traverse left and continue again. When you cannot continue, move right and begin again or move back until you can move right and resume.

Syllabus Topic : Inorder Traversal

6.5.2 Inorder Traversal (Recursive)

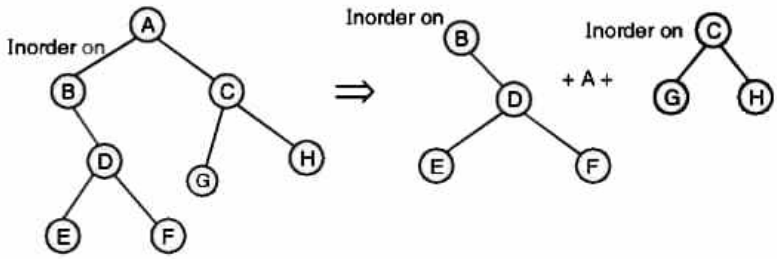
➔ **(MSBTE : S-08, W-09, S-11, W-13, S-15, W-15, W-16)**

Q. 6.5.13	**State the principles of In-order traversal of Binary Tree. (Refer Section 6.5.2)**	**S-08**
Q. 6.5.14	**With the help of example, describe the process of in order traversal of binary tree. (Refer Section 6.5.2)**	**W-09**
Q. 6.5.15	**Define tree traversal along with inorder algorithms. (Refer Section 6.5.2)**	**S-11**
Q. 6.5.16	**Write an algorithm for inorder traversal of binary tree. (Refer Section 6.5.2)**	**W-13**
Q. 6.5.17	**Write a 'C' program to traverse the tree in in-order. (Refer section 6.5.2)**	**W-15**
Q. 6.5.18	**Write algorithm for In-order traversal for Binary tree. Demonstrate with suitable example. (Refer section 6.5.2)**	**W-16**

The functioning of inorder traversal of a non-empty binary tree is as follows :

1. Firstly, traverse the left subtree in inorder.
2. Next, visit the root node.
3. At last, traverse the right subtree in inorder.

Stepwise inorder traversal of tree shown in Fig. 6.5.4.



= (B + Inorder on D) + A + ((Inorder on G) + C + (Inorder on H))

= (B + ((Inorder on E) + D + ((Inorder on F)) + A + (GCH)

= BEDFAGCH

Fig. 6.5.4

➢ **'C' Function for Inorder Traversal**

```
void inorder (node *T)        /* address of the root node is  passed in T */
{
    if (T! =NULL)
    {
        inorder (T → left);              /* inorder traversal on left subtree */
        printf ("\n%d", T → data);   /* visit the root */
        inorder (T → right);          /* inorder traversal on
                                         right subtree */
    }
}
```

Syllabus Topic : Postorder Traversal

6.5.3 Postorder Traversal (Recursive)

➔ **(MSBTE : S-10, S-11, S-14, S-15, W-15)**

Q. 6.5.19 Write an algorithm to traverse the tree in and post order with example. (Refer Section 6.5.3)	**S-10**
Q. 6.5.20 Define tree traversal along with Post order algorithms. (Refer Section 6.5.3)	**S-11**
Q. 6.5.21 Write an algorithm for post order traversal in a tree. (Refer Section 6.5.3)	**S-14**

The functioning of postorder traversal of a non-empty binary tree is as follows :

1. Firstly, traverse the left subtree in postorder.
2. Next, traverse the right subtree in postorder.
3. At last, visit the root node.

Stepwise postorder traversal of tree shown in Fig. 6.5.5.

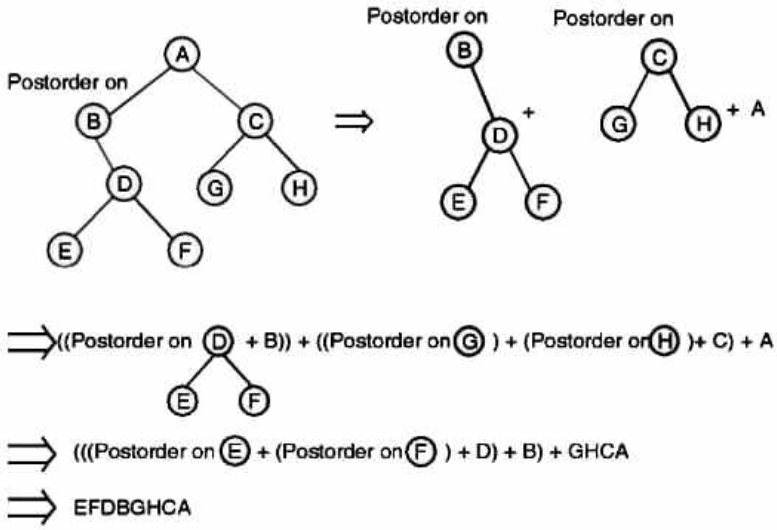


Fig. 6.5.5

➢ **'C' Function for Postorder Traversal**

Q. 6.5.22 Write a 'C/C++' program for postorder traversal of binary tree. (Refer Section 6.5.3) **W-08**

```
void postorder (node * T)          /* address of the root node passed in T */
{
    if (T! = NULL)
    {
        postorder (T → left);          /* postorder traversal on left subtree */
        postorder (T → right);         /* postorder traversal on right subtree */
        printf ("\n%d", T → data);  /* visit the root */
    }
}
```

6.6 Tree Traversal Examples

Example 6.6.1 : Traverse the following binary tree into preorder and inorder and postorder with reason.

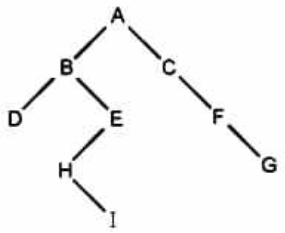


Fig. Ex. 6.6.1

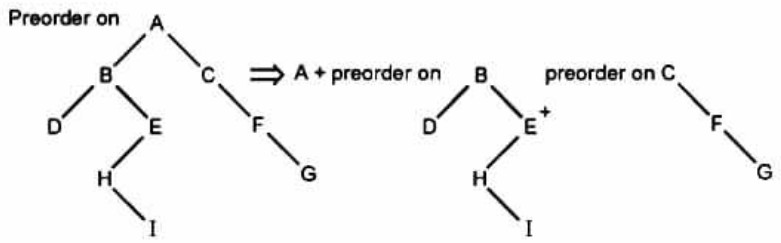
Solution :

Preorder traversal

The following of preorder traversal of a non-empty binary tree is as follows :

1. Firstly, visit the root node.
2. Next, traverse the left subtree in preorder.
3. At last, traverse the right subtree in preorder.

Stepwise preorder traversal on the above tree is given below.



= A + (B + preorder on D + preorder on E)

+ C + (preoder on F)

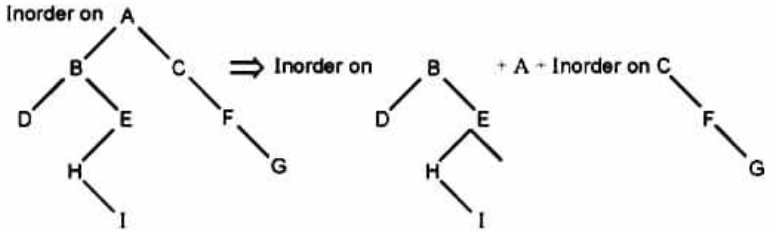
= ABD + (E + Preorder on H) + C + (F + Preorder on G)

= ABDE + (H + Preorder on I) + CFG

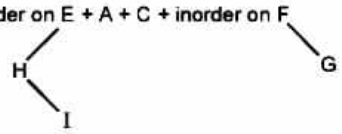
= ABDEHICFG

(a)

Inorder traversal



⇒ Inorder on D + B + inorder on E + A + C + inorder on F



⇒ DB + inorder on H + E + AC + F + inorder on G

= DB + H + inorder on I + EACF + G

= DBHIEACFG

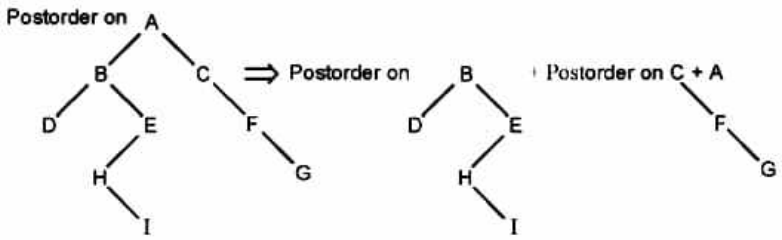
(b)

Fig. Ex. 6.6.1

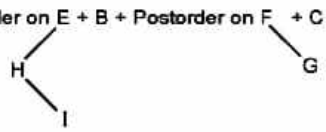
The functioning of inorder traversal of a non-empty binary tree is as follows :

1. Firstly, traverse the left subtree in inorder.
2. Next, visit the root node.
3. At last, traverse the right subtree in inorder.

Postorder traversal



⇒ Postorder on D + Postorder on E + B + Postorder on F + C + A



⇒ D + Postorder on H + E + B + Postorder on G + F + CA

⇒ D + Postorder on I + H + EB + G + FCA

⇒ D + I + HEBGFCA

⇒ DIHEBGFCA

Fig. Ex. 6.6.1 (c)

The functioning of postorder traversal of a non-empty binary tree is as follows :

1. Firstly, traverse the left subtree in postorder.
2. Next, traverse the right subtree in postorder.
3. At last, visit the root node.

Example 6.6.2 : Traverse the following binary tree into preorder and inorder.

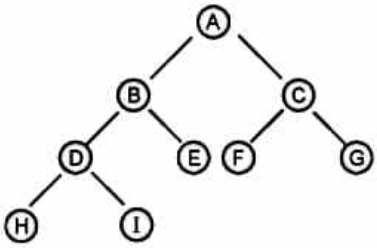
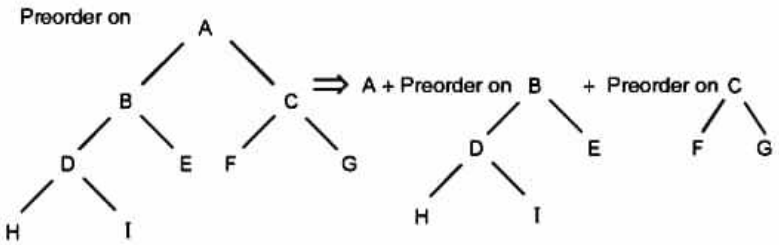


Fig. Ex. 6.6.2

Solution :

Preorder Traversal



⇒ A + B + Preorder on D + Preorder on E + C + Preorder on F + Preorder on G

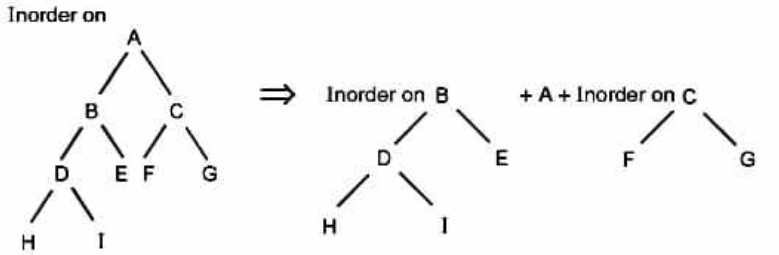
H I

⇒ AB + D + Preorder on H + Preorder on I + EC + F + G

⇒ ABD + H + I + ECFG ⇒ ABDHIECFG

Fig. Ex. 6.6.2 (a)

Inorder Traversal



⇒ Inorder on D + B + Inorder on E + A + Inorder on F + C + Inorder on G

H I

⇒ AB + D + Preorder on H+ Preorder on I + EC + F + G

⇒ Inorder on H + D + inorder on I + B + E + A + F + C + G

⇒ H + D + I + BEAFCG

⇒ HDIBEAFCG

Fig. Ex. 6.6.2 (b)

Example 6.6.3 : Perform inorder, preorder and postorder traversal of the following binary tree. **S-08, S-13**

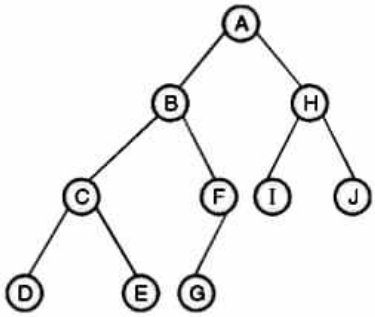


Fig. Ex. 6.6.3

Solution :

Preorder sequence : A B C D E F G H I J

Inorder sequence : D C E B G F A I H J

Postorder sequence : D E C G F B I J H A

Example 6.6.4 : For the given binary tree, perform inorder, preorder and postorder traversal. **W -08**

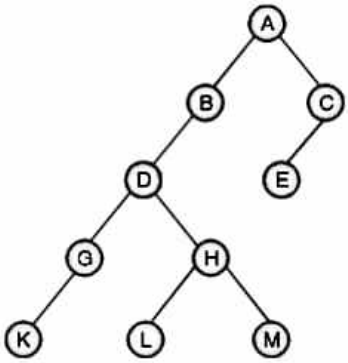


Fig. Ex. 6.6.4

Solution :

Preorder sequence : A B D G K H L M C E

Inorder sequence : K G D L H M A E C

Postorder sequence : K G L M H D B E C A

Example 6.6.5 : For the given binary tree, perform inorder, preorder and postorder traversal. **S-09, W-10**

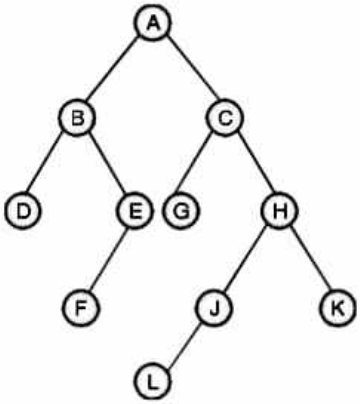


Fig. Ex. 6.6.5

Solution :

Preorder sequence : A B D E F C G H J L K

Inorder sequence : D B F E A G C L J H K

Postorder sequence : D F E B G L J K H C A

Example 6.6.6 : For the given binary tree perform inorder, preorder and postorder traversal . Refer Fig. Ex. 6.6.6. **S-12**

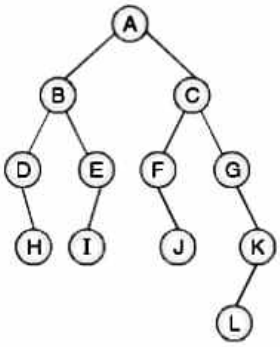


Fig. Ex. 6.6.6

Solution :

Preorder sequence : A B D H E I C F J G K L

Inorder sequence : D H B I E A F J C G L K

Postorder sequence : H D I E B J F L K G C A

Example 6.6.7 : Draw the tree structure of the expression given below. Z* (A + B) * C/D **S-11**

Solution :

Expression tree for Z * (A + B) * C/D

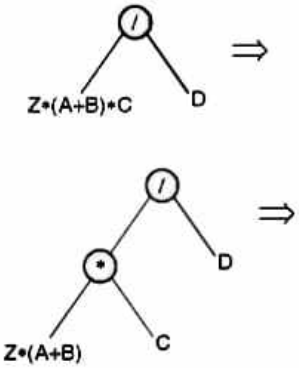


Fig. Ex. 6.6.7

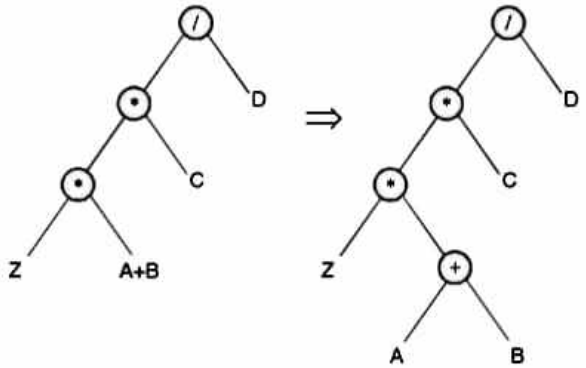


Fig. Ex. 6.6.7(a)

Example 6.6.8 : Consider the following tree. Write its

(i) In-order traversal sequence.

(ii) Pre-order traversal sequence.

(iii) Post order traversal sequence. **W -12**

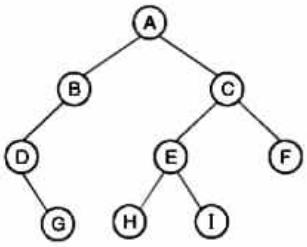


Fig. Ex. 6.6.8

Solution :

(i) In-order traversal sequence : DGBAHEICF

(ii) Pre-order traversal sequence : ABDGCEHIF

(iii) Post order traversal sequence : GDBHIEFCA

Example 6.6.9 : Consider the following tree. Write its :

a) In-order traversal sequence

b) Pre-order traversal sequence

c) Post-order traversal sequence. **S -15**

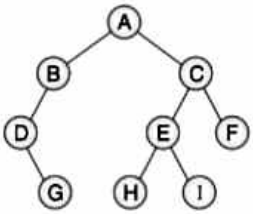


Fig. Ex. 6.6.9

Ans. :

Preorder : A B D G C E H I F

Inorder : D G B A H E I C F

Postorder : G D B H I E F C A

Example 6.6.10 : Differentiate between tree and graph. Traverse the following tree in Inorder, Preorder and Postorder. **S -16**

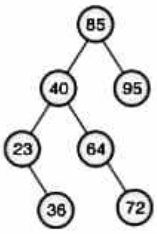
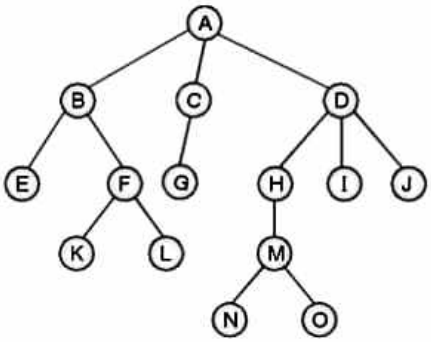
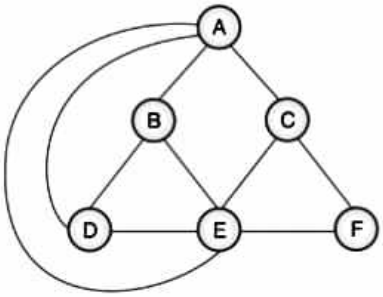


Fig. Ex. 6.6.10

Solution :

Difference between tree and graph

Sr. No.	Tree	Graph
1.	A tree is a collection of elements called "nodes", one of which is distinguished as a root say r, along with a relation "parenthood" that places a hierarchical structure on the nodes.	A graph G is a set of vertices (V) and set of edges (E). The set V is a finite, nonempty set of vertices. The set E is a set of pair of vertices representing edges.
2.	In tree, there is exactly one root node, every child have only one parent.	In graph there is no such root node.
3.	Tree is a special graph which have no loops, no circuits and no self loops,	Graph can have loop, cycle and self loops.
4.	Tree can be traversed by Depth First Search and Breadth First Search.	Graph can be traversed by Depth First Search and Breadth First Search.
5.	**Types :** 1. Binary Search Tree 2. AVL Tree	**Types :** 1. Directed Graph 2. Undirected Graph
6.	Tree is hierarchical model.	Graph is network model.
7.		

Preorder : 85, 40, 23, 36, 64, 72, 95

Inorder : 23, 36, 40, 64, 72, 85, 95

Postorder : 36, 23, 72, 64, 40, 95, 85

Example 6.6.11 : Traverse the following tree in inorder, preorder and postorder. **W -17**

Solution :

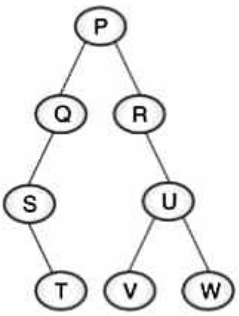


Fig. Ex. 6.6.11

Preorder : P Q S T R U V W

Inorder : S T Q P V U W R

Postorder : T S Q V W U R P

Example 6.6.12 : For given tree perform preorder postorder and inorder traversal. Refer Fig. Ex. 6.6.12. **W -13**

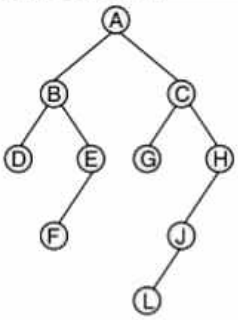


Fig. Ex. 6.6.12

Solution :

Preorder sequence : A B D E F C G H J L

Inorder sequence : D B F E A G C L J H

Postorder sequence : D F E B G L J H C A

Example 6.6.13 : Write the preorder and postorder traversal of a tree. **W -13**

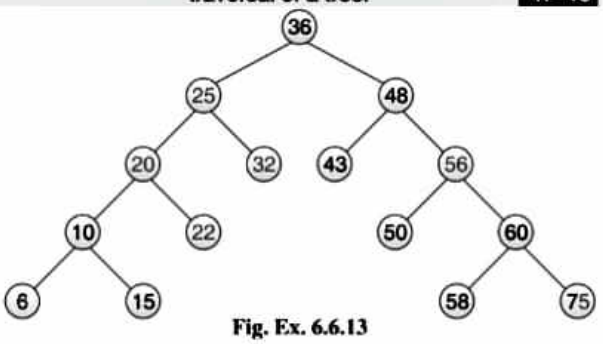


Fig. Ex. 6.6.13

Solution :

Postorder : 6, 15, 10, 22, 20, 32, 25, 58, 75, 50, 60, 43, 56, 48, 36

Preorder : 36, 25, 20, 10, 6, 15, 22, 32, 48, 43, 56, 50, 60, 58, 75

Example 6.6.14 : Perform in-order, post-order and pre-order traversal of binary tree, Refer Fig. Ex. 6.6.14. **W -14**

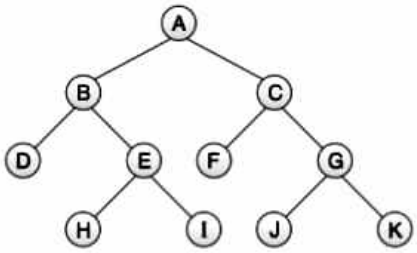


Fig. Ex. 6.6.14

Solution :

(1) **Inorder :** D, B, H, E, I, A, F, C, J, G, K

(2) **Preorder :** A, B, D, E, H, I, C, F, G, J, K

(3) **Postorder :** D, H, I, E, B, E, J, K, G, C, A

Syllabus Topic : Binary Search Tree (BST)

6.7 Binary Search Tree (BST)

➔ **(MSBTE : S-14)**

Q. 6.7.1 What is meant by binary search tree with example ? (Refer Sections 6.7 and 6.7.1) **S-14**

6.7.1 Definition

A binary search tree is a binary tree, which is either empty or in which each node contains a key that satisfies the following conditions :

(1) All keys are distinct.

(2) For every node, X, in the tree, the values of all the keys in its left subtree are smaller than the key value in X.

(3) For every node, X, in the tree, the values of all the keys in its right subtree are larger than the key value in X.

Binary search tree finds its application in searching. In Fig. 6.7.1, the tree on the left is a binary search tree. Tree on the right is not a BST. Left subtree of the node with key 8 has a key with value 9.

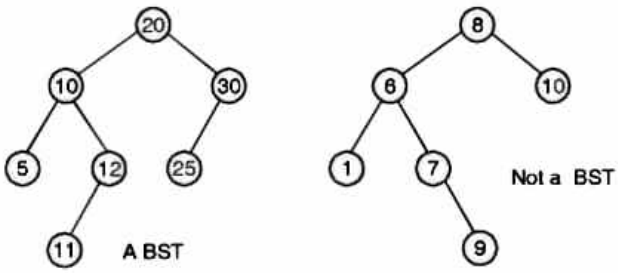


Fig. 6.7.1 : Left tree is a BST, right tree is not a binary search tree

6.7.2 Operations on a Binary Search Tree

➔ **(MSBTE : W-10)**

Q. 6.7.2 Enlist operations on trees. (Refer Section 6.7.2) W-10

(1) Initialise (2) Find

(3) Makeempty (4) Insert

(5) Delete (6) Create

(7) Findmin (8) Findmax

☞ **Structure of node of a binary search tree**

```
typedef struct BSTnode
{
    int data;
    struct BSTnode *left, *right;
} BSTnode;
```

6.7.2(A) Initialize Operation

Initially the tree is empty and hence the referencing pointer root should be set to NULL.

➢ **'C' function for initialise.**

```
BSTnode* initialize()
{
    return(NULL) ;
}
```

6.7.2(B) Find Operation

➔ **(MSBTE : S-08)**

Q. 6.7.3 Explain searching a value in Binary search tree with example. (Refer Section 6.7.2(B)) S-08

It is often required to find whether a key is there in the tree. If the key, X, is found in the node T, the function returns the address of T or NULL if there is no such node.

☞ **Recursive algorithm for find**

	return value	condition
Find (root, x)	NULL	if root == NULL
	root	if root → data == x
	return (Find (root → right, x))	if x > root → data
	return (Find (root → left, x))	if x < root → data

If the key, X, is found to be larger than the value stored in node T, we make a recursive call to function with the right subtree. Otherwise, we make a recursive call to function with the left subtree.

➢ **'C' Function for find() Recursive**

```
BSTnode *find(BSTnode * root ,int x)
{
    if((root==NULL)
        return(NULL);
    if(root→ data==x)
        return(root);
    if(x>root→ data)
        return(find(root→ right),x));
    return(find(root→ left),x));
}
```

➢ **'C' Function for find() Non-Recursive**

Q. 6.7.4 Write a program for 'Binary Search in 'C' language. (Refer Section 6.7.2(B)) S-08

```
BSTnode *find(BSTnode *root,int x)
{
    while(root!=NULL)
```

```
    {
        if(x==root→ data)
            return(root);
        if(x>root→ data)
            root=root→ right;
        else
        root=root→ left;
    }
    return(NULL);
}
```

6.7.2(C) Makeempty Operation

This function deletes every node of the tree. It also releases the memory acquired by the node.

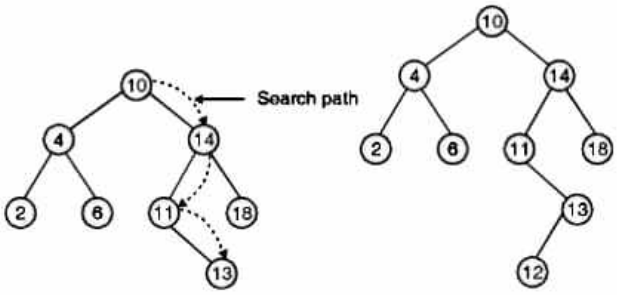
➢ **'C' Function to Release Memory**

```
BSTnode *makeempty (BSTnode *root)
{
    if (root ! = NULL)
        {
            makeempty (root → left);
            makeempty (root → right);
            free (root);
        }
        return (NULL);
}
```

6.7.2(D) Insert Operation

Q. 6.7.5	**Explain insertion of node on binary search tree. (Refer Section 6.7.2(D))**	**W-11**
Q. 6.7.6	**Describe following binary search tree operation : Insertion of node (Refer Section 6.7.2(D))**	**W-12**

The function insert (T, x), adds element x to an existing binary search tree. T is tested for NULL, if so, a new node is created to hold x. T is made to point to the new node. If the tree is not empty then we search for x as in find() operation.



(a) Before insertion of the new key 12 (b) After inserting 12
Fig. 6.7.2 Insertion operation into a binary search tree

If x is already there in the then insert() operation terminates without insertion as a BST is not allowed to have duplicate keys. If we find a NULL pointer during the find() operation. We replace it by a pointer to a new node holding x. Fig. 6.7.2 shows the insert operation.

➢ **'C' Function for insert() – Recursive**

```
BSTnode *insert(BSTnode *T,int x)
{
    if(T==NULL)
    {
        T=(BST *)malloc(sizeof(BSTnode));
        T→ data=x;
        T→ left=NULL;
        T→ right=NULL;
        return(T);
    }
    if(x > T→ data)          // insert in right subtree
    {
        T→ right=insert(T→ right,x);
        return(T);
    }
    T→ left=insert(T→ left,x);   //insert in left subtree
    return(T);
}
```

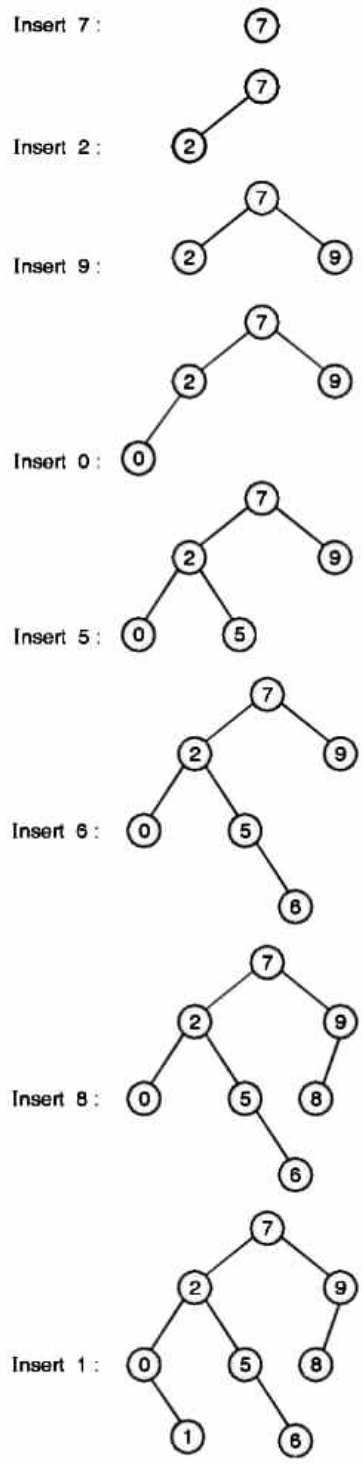
➢ **'C' Function for Insert() - Non Recursive**

```
BSTnode *insert(BSTnode *T,int x)
{
    BSTnode *p,*q,*r;
    // acquire memory for the new node
    r=(BSTnode*)malloc(sizeof(BSTnode));
    r→ data=x;
    r→ left=NULL;
    r→ right=NULL;
    if(T==NULL)
        return(r);
    // find the leaf node for insertion
    p=T;
    while(p!=NULL)
    {
        q=p;  //save p before advancing
        if(x>p→ data)
            p=p→ right;
        else
            p=p→ left;
    }
    if(x>q→ data)
        q→ right=r;  // x as right child of q
    else
        q→ left=r;  //x as left child of q
    return(T);
}
```

6.7.2(E) Example on Creation of a BST

Example 6.7.1 : Insert the integers 7, 2, 9, 0, 5, 6, 8, 1 into a binary search tree by repeated application of insert operation.

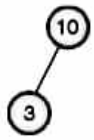
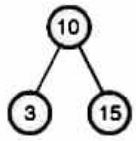
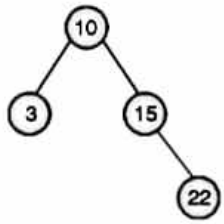
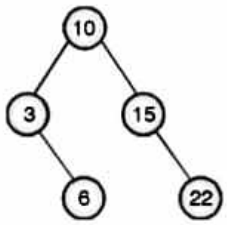
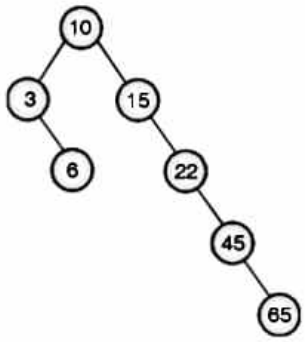
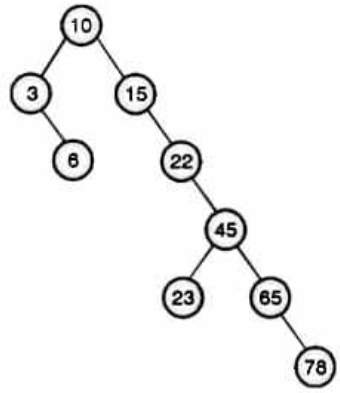
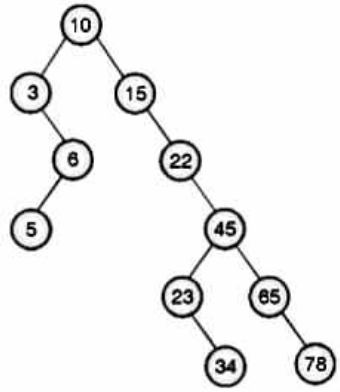
Solution :



Example 6.7.2 : Construct binary search tree for the following data :

10, 3, 15, 22, 6, 45, 65, 23, 78, 34, 5

Solution :

Sr. No.	Value to be inserted	Tree
1.	10	
2.	3	
3.	15	
4.	22	
5.	6	
6.	45,65	
7.	23,78	
8.	34,5	

Preorder : 10, 3, 6, 5, 15, 22, 45, 23, 34, 65, 78

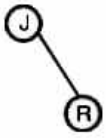
Inorder : 3, 5, 6, 10, 15, 22, 23, 34, 45, 65, 78

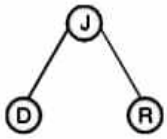
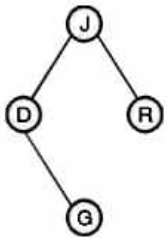
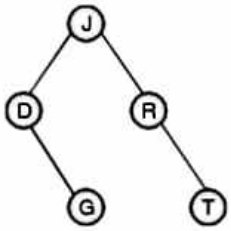
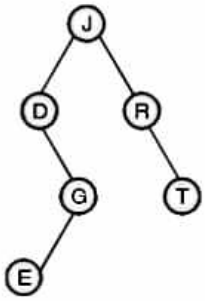
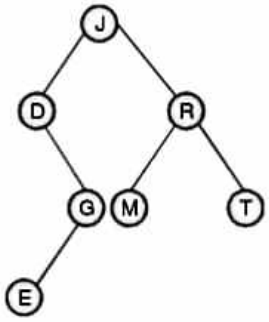
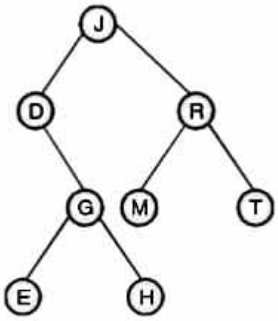
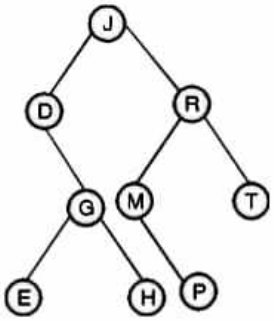
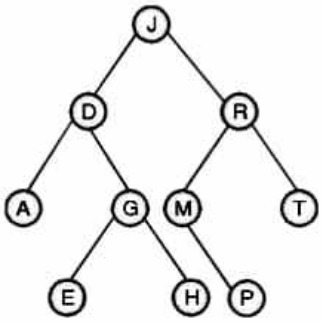
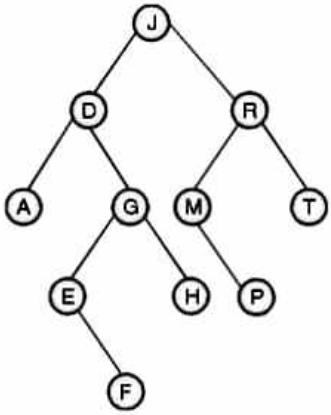
Postorder : 5, 6, 3, 34, 23, 78, 65, 45, 22, 15, 10

Example 6.7.3 : Construct a Binary search tree from the given list of letters inserted in order into all empty binary search tree J, R, D, G, T, E, M, H, P, A, F, Q

S-09, S-13

Solution :

Sr. No.	Letter to be inserted	Tree
1	J	
2	R	

Sr. No.	Letter to be inserted	Tree
3	D	
4	G	
5	T	
6	E	
7	M	
8	H	
9	P	
10	A	
11	F	

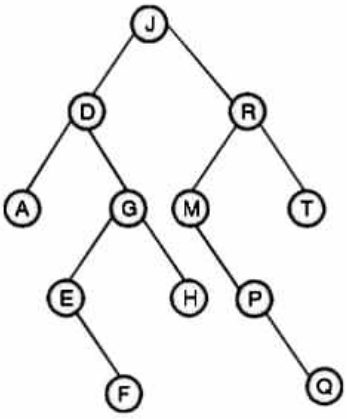
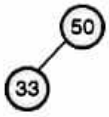
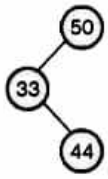
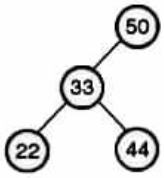
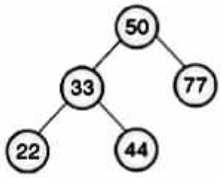
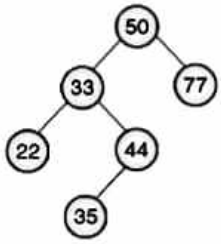
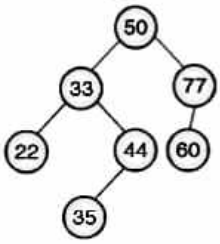
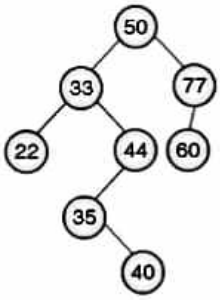
Sr. No.	Letter to be inserted	Tree
12	Q	

Fig. Ex. 6.7.3

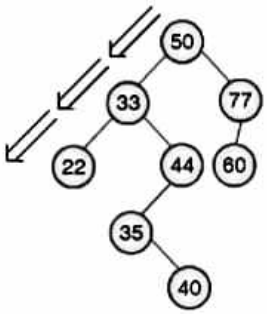
Example 6.7.4 : Suppose following eight numbers are added in order into an empty binary search tree T.50, 33, 44, 22, 77, 35, 60, 40 Draw the tree T and search an item 20 in the tree. **W-10, W-13**

Solution :

Sr. No.	Insert	Tree
1.	50	
2.	33	
3.	44	
4.	22	

Sr. No.	Insert	Tree
5.	77	
6.	35	
7.	60	
8.	40	

During the search operation, the algorithm will follow the path as shown below. The search will end in a failure as the element 20 is not in the tree.



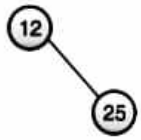
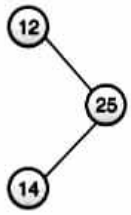
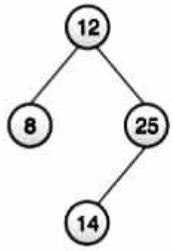
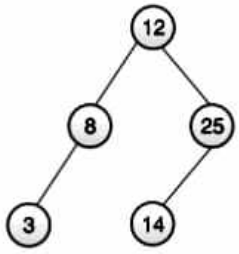
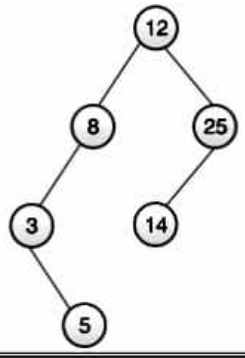
Path traced during search for the element 20.

$50 \Rightarrow 33 \Rightarrow 22 \Rightarrow$ NULL

Example 6.7.5 : Create a binary tree for following data : 12, 25, 14, 8, 3, 5. **W-15**

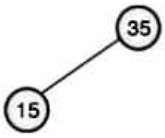
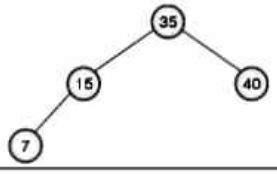
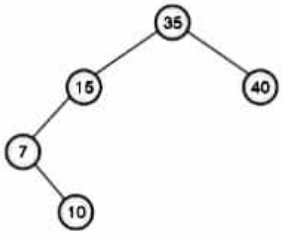
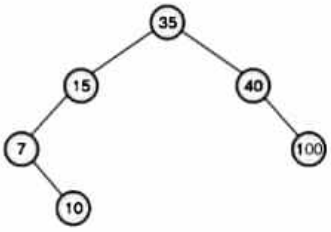
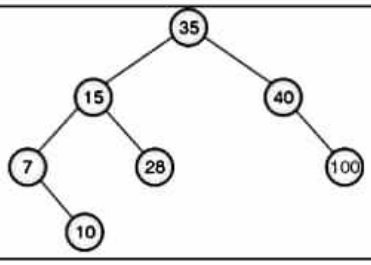
Solution :

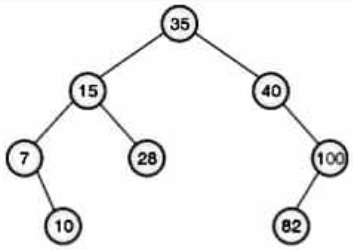
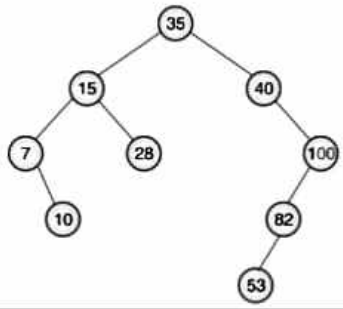
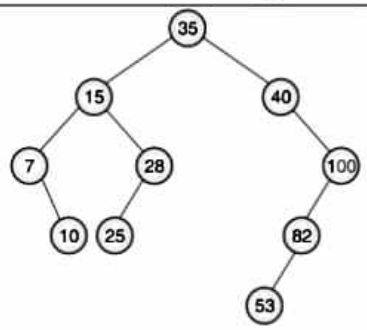
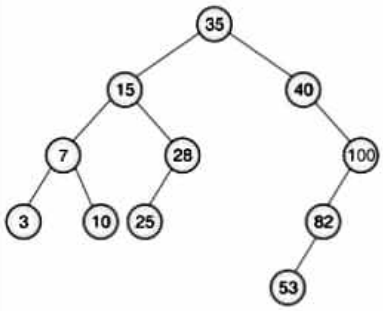
Given data : 12, 25, 14, 8, 3, 5

Data to be inserted	Tree
12	
25	
14	
8	
3	
5	

Example 6.7.6 : Draw binary search tree using following elements :
35,15, 40, 7, 10, 100, 28, 82, 53, 25, 3. **S-16**

Solution :

Step 1	**Insert 35**	
Step 2	**Insert 15**	
Step 3	**Insert 40**	
Step 4	**Insert 7**	
Step 5	**Insert 10**	
Step 6	**Insert 100**	
Step 7	**Insert 28**	

Step 8	**Insert 82**	
Step 9	**Insert 53**	
Step 10	**Insert 25**	
Step 11	**Insert 3**	

Example 6.7.7 : **Draw a binary search tree for given sequence and write postorder traversal of tree.**

10 5 8 9 7 6 2 15. **S-17**

Solution :

BST for the sequeuce is given below :

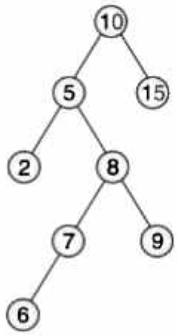


Fig. Ex. 6.7.7

Postorder sequence : 2 6 7 9 8 5 15 1

6.7.2(F) Delete Operation

➔ **(MSBTE : S-08, W-11, S-12, W-12, S-13)**

Q. 6.7.7 **Give stepwise procedure for deletion of a node in a Binary Tree.**
(Refer Section 6.7.2(F)) **S-08, S-13**

Q. 6.7.8 **Explain deletion of node on binary search tree.**
(Refer Section 6.7.2(F)) **W-11**

Q. 6.7.9 **Give the algorithm for deletion of a node in binary search tree.**
(Refer Section 6.7.2(F)) **S-12**

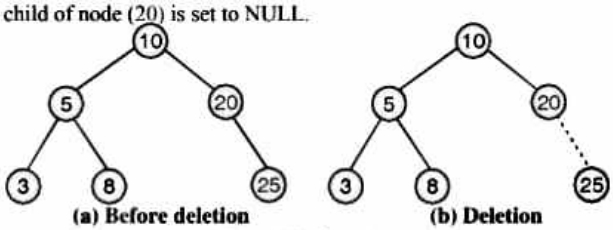
Q.6.7.10 **Describe following binary search tree operation :**
Deletion of node. (Refer Section 6.7.2(F)) **W-12**

In order to delete a node, we must find the node to be deleted. The node to be deleted may be :

(a) A leaf node (b) A node with one child

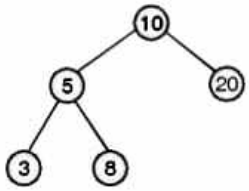
(c) A node with two children.

If the node is a leaf node, it can be deleted immediately by setting the corresponding parent pointer to NULL. For example, consider the tree of Fig. 6.7.3(a). Assume that the node (25) is to be deleted. node (25) is a right child of its parent node (20). Right child of node (20) is set to NULL.



(a) Before deletion **(b) Deletion**

Fig. 6.7.3 Contd...

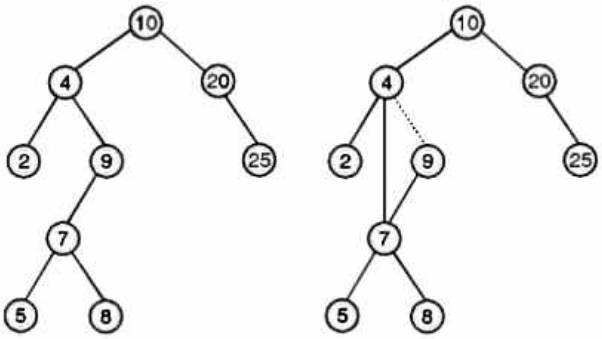


(c) After deletion

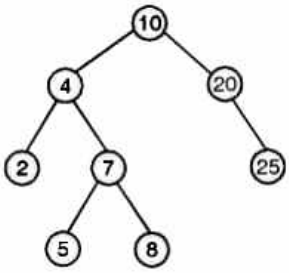
Fig. 6.7.3 : Deletion of node (25), a leaf node

Even when the node to be deleted has one child, it can be deleted easily. If a node q is to be deleted and it is right child of its parent node p. The only child of q will become the right child of p after deletion of q. Similarly, if a node q is to be deleted and it is left child of its parent node p. The only child of q will become the left child of p after deletion.

For example, consider the tree of Fig. 6.7.4(a). Assume that node (9) is to be deleted. Since node (9) is a right child of its parent node (4). The only child subtree of node (9), with node (7) will become the right subtree of node (4) after deletion. Fig. 6.7.5 shows deletion of node (15). It is left child of its parent node (20).

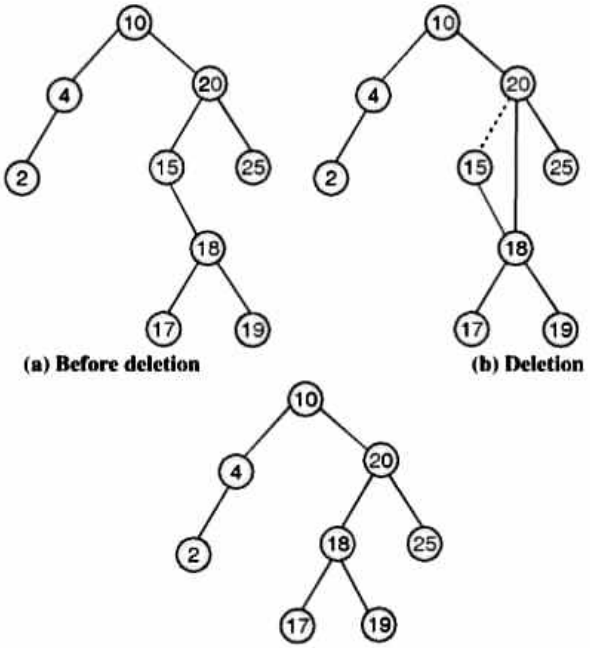


(a) Before deletion **(b) Deletion**



(c) After deletion

Fig. 6.7.4 : Deletion of a node (9) with one child

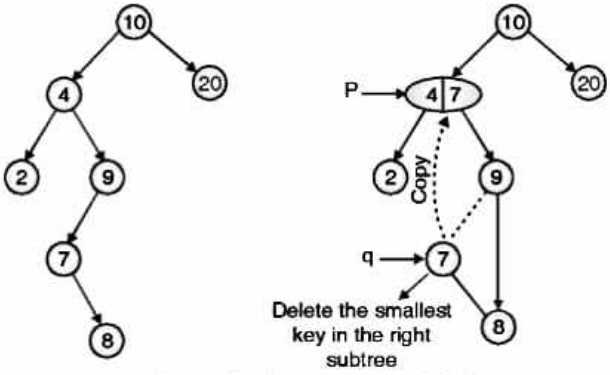


(a) Before deletion **(b) Deletion**

(c) After deletion

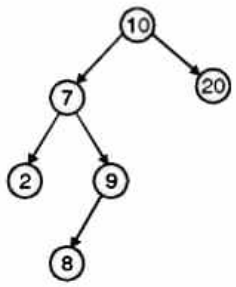
Fig. 6.7.5 : Deletion of a node (15) with one child

The case in which the node to be deleted has two children is a bit complicated. The general strategy is to replace the data of this node with the smallest data of the right subtree (inorder successor) and then delete the smallest node in the right subtree. The smallest node in right subtree will either be a leaf node or a node of degree 1. As a first step, the node with smallest value in the right subtree of P (address of node (4)) is found and its address is stored in q. Content of node q is copied in node P. As a second step, the node q is deleted, a node with one child. We have already discussed how to delete a leaf node or node with one child. For example, consider the tree of Fig. 6.7.6. Assume that node (4) is to be deleted. Node (7) is the smallest node in the right subtree of node (4). Value 7 is copied in the node P (earlier node (4)) as shown in the Fig. 6.7.6(b). Now, the node q is deleted.



(a) Before deletion **(b) Deletion**

Fig. 6.7.6 Contd...



(c) After deletion

Fig. 6.7.6 : Deletion of a node (4) with two children

➢ **'C' Function for Deletion**

```
BSTnode *delete(BSTnode *T,int x)
{
    if(T==NULL)
    {
        printf("\nElement not found :");
        return(T);
    }
    if(x < T→ data)            // delete in left subtree
    {
        T→ left=delete(T→ left,x);
        return(T);
    }
    if(x > T→ data)            // delete in right subtree
    {
        T→ right=delete(T→ right,x);
        return(T);
    }
    // element is found
    if(T→ left==NULL && T→ right==NULL)   // a leaf
node
    {
        temp=T;
        free(temp);
        return(NULL);
    }
    if(T→ left==NULL)
    {
        temp=T;
        T=T→ right;
        free(temp);
        return(T);
    }
    if(T→ right==NULL)
    {
        temp=T;
        T=T→ left;
        free(temp);
        return(T);
    }
    // node with two children
    temp=find_min(T→ right);
    T→ data=temp→ data;
    T→ right=delet(T→ right, temp → data);
    return(T);
}
```

6.7.2(G) Create

A binary search tree can be created by making repeated calls to insert operation.

➢ **'C' Function for Tree Creation**

```
BSTnode *create()
{
    int n,x,i;
    BSTnode *root;
    root=NULL;
    printf("\nEnter no. of nodes :");
    scanf("%d",&n);
    printf("\nEnter tree values :");
```

```
    for(i=0;i<n;i++)
    {
        scanf("%d",&x);
        root=insert(root,x);
    }
    return(root);
}
```

6.7.2(H) FindMin

This function returns to address of the node with smallest value in the tree.

➢ **'C' function for finding the smallest value is a BST.**

```
BSTnode *findmin (BSTnode *T)
{
        while (T→ left ! = NULL)
            T = T → left ;
        return (T);
}
```

6.7.2(I) FindMax

This function returns the address of the node with largest value in the tree.

➢ **'C' function for finding the largest value in a BST.**

```
BSTnode *findmax (BSTnode * T)
{
        while (T → right ! = NULL)
            T = T → right ;
        return (T);
}
```

6.7.3 Program for Various Operations on BST

➢ **Program 6.7.1 : Program showing various operations on a binary search tree.**

```
#include<conio.h>
#include<stdio.h>
#include<stdlib.h>
typedef struct BSTnode
{
    int data;
    struct BSTnode *left,*right;
}BSTnode;

BSTnode *initialise();
BSTnode *find(BSTnode *,int);
BSTnode *insert(BSTnode *,int);
BSTnode *delet(BSTnode *,int);
BSTnode *find_min(BSTnode *);
BSTnode *find_max(BSTnode *);
BSTnode *create();
void inorder(BSTnode *T);
void main()
{
    BSTnode *root,*p;
    int x;
    clrscr();
    initialise();
    root=create();
    printf("\n**** BST created ****");
    printf("\ninorder traversal on the tree ");
    inorder(root);
    p=find_min(root);
    printf("\n smallest key in the tree = %d",p->data);
    p=find_max(root);
    printf("\nlargest key in the tree = %d",p->data);
    printf("\n **** delete operation ****");
    printf("\nEnter the key to be deleted :");
    scanf("%d",&x);
    root=delet(root,x);
    printf("inorder traversal after deletion :");
```

```
    inorder(root);
}
void inorder(BSTnode *T)
{
    if(T!=NULL)
    {
        inorder(T→ left);
        printf("%5d",T→ data);
        inorder(T→ right);
    }
}
BSTnode *initialise()
{
    return(NULL);
}
BSTnode *find(BSTnode *root,int x)
{
    while(root!=NULL)
    {
        if(x==root→ data)
            return(root);
        if(x>root→ data)
            root=root→ right;
        else
        root=root→ left;
    }
    return(NULL);
}
BSTnode *insert(BSTnode *T,int x)
{
    BSTnode *p,*q,*r;
    // acquire memory for the new node
    r=(BSTnode*)malloc(sizeof(BSTnode));
    r→ data=x;
    r→ left=NULL;
    r→ right=NULL;
    if(T==NULL)
        return(r);
    // find the leaf node for insertion
    p=T;
    while(p!=NULL)
    {
        q=p;
        if(x>p→ data)
            p=p→ right;
        else
            p=p→ left;
    }
    if(x>q→ data)
        q→ right=r;  // x as right child of q
    else
        q→ left=r;   //x as left child of q
    return(T);
}
BSTnode *delete(BSTnode *T,int x)
{
    BSTnode *temp;
    if(T==NULL)
    {
        printf("\nElement not found :");
        return(T);
    }
    if(x < T→ data)          // delete in left subtree
    {
        T→ left=delete(T→ left,x);
        return(T);
    }
    if(x > T→ data)          // delete in right subtree
```

```
    {
        T→ right=delete(T→ right,x);
        return(T);
    }
    // element is found
    if(T→ left==NULL && T→ right==NULL) //a leaf node
    {
        temp=T;
        free(temp);
        return(NULL);
    }
    if(T→ left==NULL)
    {
        temp=T;
        T=T→ right;
        free(temp);
        return(T);
    }
    if(T→ right==NULL)
    {
        temp=T;
        T=T→ left;
        free(temp);
        return(T);
    }
    // node with two children
    temp=find_min(T→ right);
    T→ data=temp→ data;
    T→ right=delete(T→ right,x);
    return(T);
}
BSTnode *create()
{
    int n,x,i;
    BSTnode *root;
    root=NULL;
    printf("\nEnter no. of nodes :");
    scanf("%d",&n);
    printf("\nEnter tree values :");
    for(i=0;i<n;i++)
    {
        scanf("%d",&x);
        root=insert(root,x);
    }
    return(root);
}
BSTnode *find_min(BSTnode *T)
{
    while(T→ left!=NULL)
        T=T→ left;
    return(T);
}
BSTnode *find_max(BSTnode *T)
{
    while(T→ right!=NULL)
        T=T→ right;
    return(T);
}
```

Output

```
Enter no. of nodes : 5
Enter tree values : 34 11 2   99 6
**** BST created ****
inorder traversal on the tree    2   6   11   34   99
 smallest key in the tree = 2
largest key in the tree = 99
 **** delete operation ****
Enter the key to be deleted :11
inorder traversal after deletion :   2   6   34   99
```

Syllabus Topic : Expression Trees

6.8 Application of Trees

6.8.1 Expression Trees

➔ **(MSBTE : W-14)**

Q. 6.8.1 Describe expression tree with an example. (Refer Section 6.8.1) W-14

When an expression is represented through a tree, it is known as an expression tree. The leaves of an expression tree are operands, such as constants or variables names and all internal nodes contain operations. Fig. 6.8.1 gives an example of an expression tree.

(a + b * c) * e + f

A preorder traversal on the expression tree gives prefix equivalent of the expression. A postorder traversal on the expression tree gives postfix equivalent of the expression.

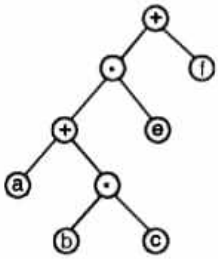


Fig. 6.8.1

Prefix (expression tree of 6.8.1) = + * + a * b c e f

Postfix (expression tree of 6.8.1) = a b c * + e * f +

☞ **Constructing an expression tree**

Here, we will discuss an algorithm for constructing an expression tree from a postfix expression. In case an expression tree is to be converted from infix expression, infix expression should be converted to postfix.

☞ **Algorithm**

We read our expression one symbol at a time. If the symbol is an operand, we create one node tree and push a pointer to it onto a stack. If the symbol is an operator, we pop pointer to two trees T_2 and T_1 from the stack and form a new tree whose root is the operator and whose left and right children point to T_1 and T_2 respectively. A pointer to this new tree is then pushed onto the stack.

As an example, suppose the input is ,

abc * + e * f +

The first three symbols are operands, so we create one-node trees push pointers to them onto a stack.

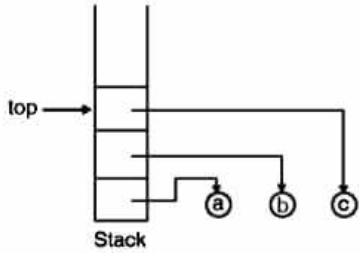


Fig. 6.8.2

Next, 4 * is read, so two pointers to tree are popped, a new tree is formed, and a pointer to it is pushed onto the stack.

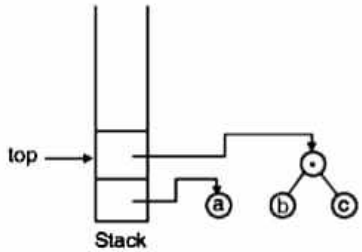


Fig. 6.8.3

Next, a + is read, so two pointers to tree are popped, a new tree is formed, and a pointer to it is pushed onto the stack.

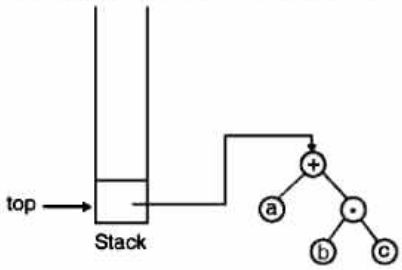


Fig. 6.8.4

Next, e is read, a one node tree is created and a pointer to it is pushed onto the stack.

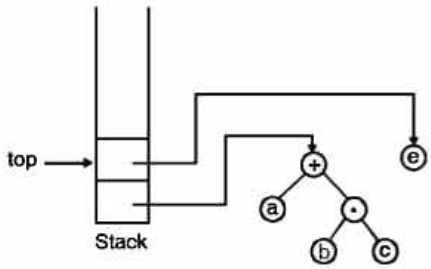


Fig. 6.8.5

Next, a * is read, so two pointers to tree are popped, a new tree is formed, and a pointer to it is pushed onto the stack.

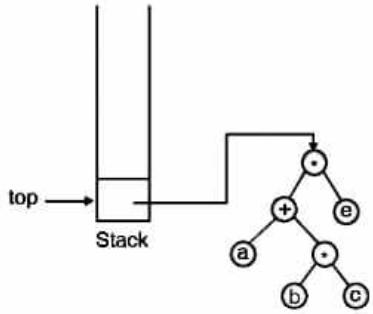


Fig. 6.8.6

Continuing, f is read, one node tree is created and a pointer to it is pushed onto the stack.

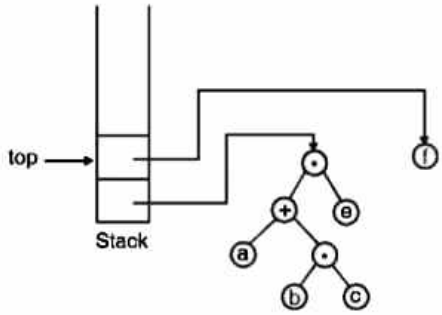


Fig. 6.8.7

Finally, a + is read, two trees are merged, and a pointer to the final tree is pushed on the stack.

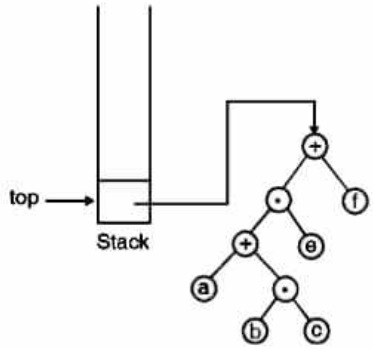


Fig. 6.8.8

Example 6.8.1 : State any two applications of binary tree. S-12

Solution :

Two applications of binary tree :

1. Expression tree 2. Heap sort

6.8.2 Conversion of an Expression into Binary Tree

Example 6.8.2 : Construct the binary tree for the following expression.

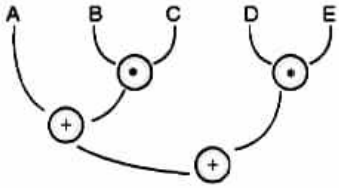
A + B * C + D * E

Solution :

Step 1 : Group elements as per the sequence of evaluation. [This step is similar to fully parenthesizing an expression].

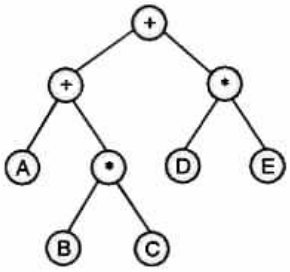
(a)

Step 2 : Move the operator at the center of the group.



(b)

Step 3 : Invert the structure.



(c)

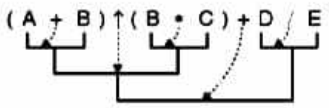
Fig. Ex. 6.8.2

Example 6.8.3 : Construct the binary tree for the following expression

(A + B) ↑ (B * C) + D / E

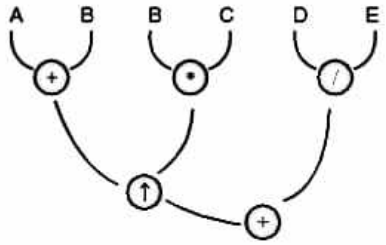
Solution :

Step 1 : Group elements as per the sequence of evaluation. [This step is similar to fully parenthesizing an expression].



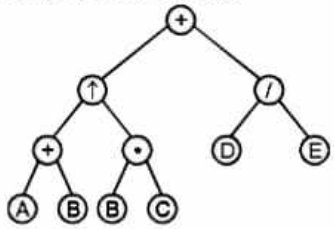
(a)

Step 2 : Move the operator at the center of the group.



(b)

Step 3 : Inverting the structure, we get



(c)

Fig. Ex. 6.8.3

Example 6.8.4 : Draw the tree structure for the following expressions

(i) $(2a + 5b)^3 (x - 7y)^4$ **W - 08, W -10, S-15**

(ii) $(a - 3b) (2x - y)^3$ **W-08, W-13, S-15**

(iii) $E_1 = (a - 3b) (2x - y)^3$ **S-09**

(iv) $E_2 = (2a + 5b)^3 (x - 7y)^4$ **S-09**

(v) $(- x + 2y + 3z + 4a + 5b + 6c)^4 * (7d + 4Z)^2$ **W -09**

(vi) $(x + 2y + 3z - 4 - 4a + 5b + 6c)^4 * (7b + 4z)^2$ **S-10, S-13**

(vii) $E = ((a - b) / (C * D) + e)$ **W -10**

(viii) $(11a^2 + 7b^3 + 5c)^4 + (3a^3 + 4b^2 + 8c)^3$ **S-12, S-15**

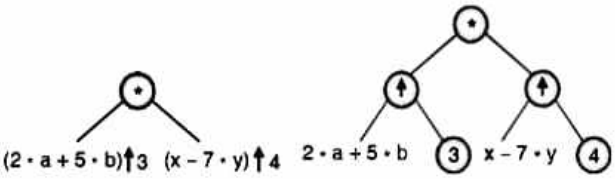
Solution :

(i) $(2a + 5b)^3 (x - 7y)^4$

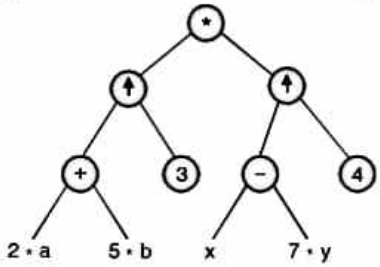
Re- writing the expression after specifying every operator, we get

$((2 * a + 5 * b) \uparrow 3) * ((x - 7 * y) \uparrow 4)$

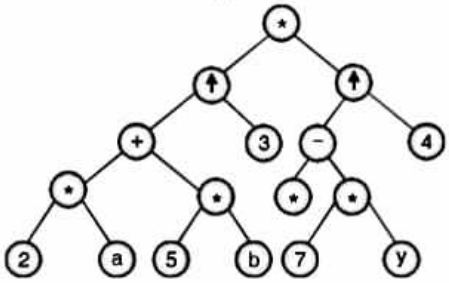
Stepwise construction of tree :



(a) (b)



(c)



(d)

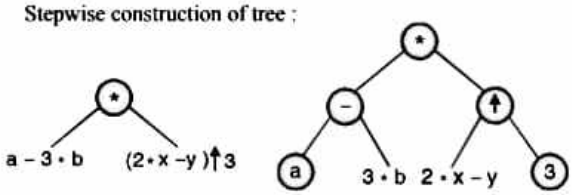
Fig. Ex. 6.8.4(i)

(ii) $(a - 3b) (2 x - y)^3$

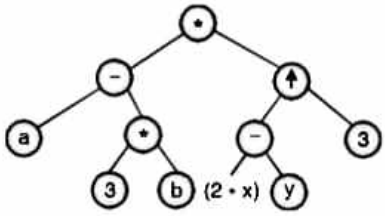
Fully specifying every operator, we get,

$(a - 3 * b) * (2 * x - y) \uparrow 3$

Stepwise construction of tree :

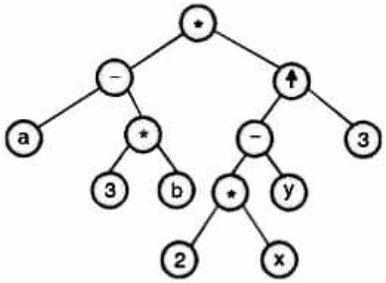


(a) (b)



(c)

Fig. Ex. 6.8.4 (ii) Contd...



(d)

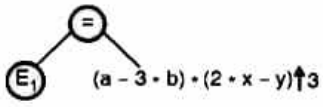
Fig. Ex. 6.8.4(ii)

(iii) $E_1 = (a - 3b)\,(2x - y)^3$

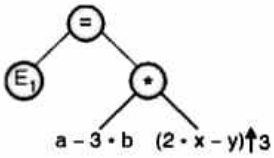
Fully specifying every operator, we get

$E_1 = (a - 3 * b) * (2 * x - y) \uparrow 3$

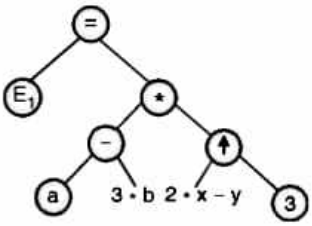
Stepwise construction of tree :



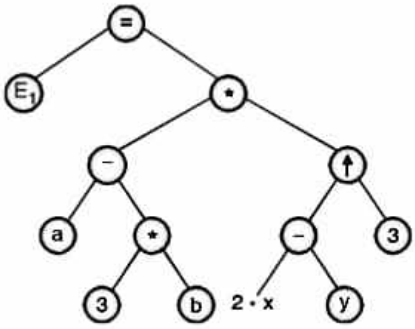
(a)



(b)

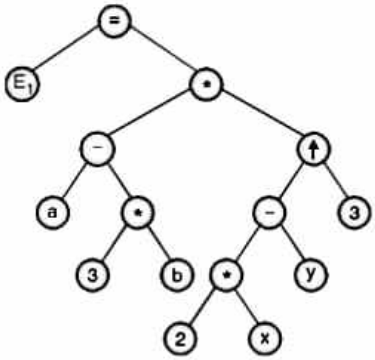


(c)



(d)

Fig. Ex. 6.8.4(iii) Contd...



(e)

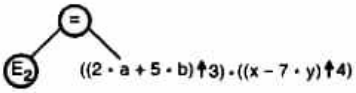
Fig. Ex. 6.8.4(iii)

(iv) $E_2 = (2a + 5b)^3\,(x - 7y)^4$

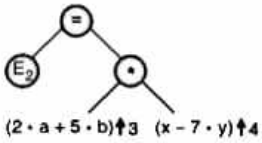
Fully specifying every operator, we get

$E_2 = ((2 * a + 5 * b) \uparrow 3) * ((x - 7 * y) \uparrow 4)$

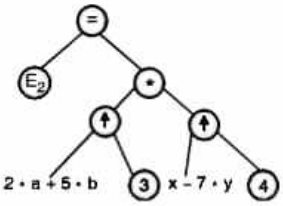
Stepwise construction of tree :



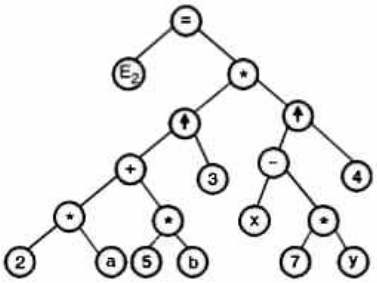
(a)



(b)



(c)



(d)

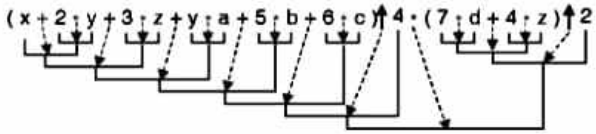
Fig. Ex. 6.8.4(iv)

(v) $(x + 2y + 3z + 4a + 5b + 6c)^4 * (7d + 4z)^2$

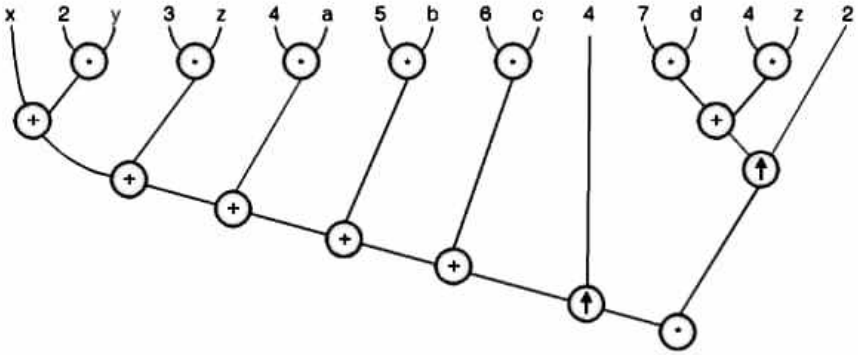
Step 1 : Fully specifying every operator, we get,

$(x + 2 * y + 3 * z + 4 * a + 5 * b + 6 * c) \uparrow 4 * (7 * d + 4 * z) \uparrow 2$

Step 2 : Group elements as per the sequence of evaluation. (This step is similar to fully parenthesizing an expression)



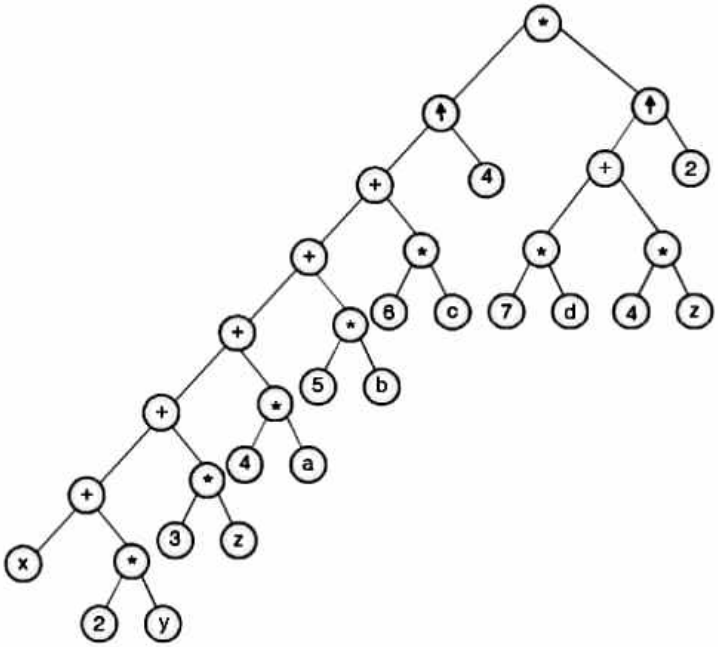
(a)



(b)

Fig. Ex. 6.8.4 (v) Contd..

Step 3 : invert the structure.



(c)

Fig. Ex. 6.8.4(v)

(vi) $(x + 2y + 3z - 4 - 4a + 5b + 6c)^4 * (7d + 4z)^2$

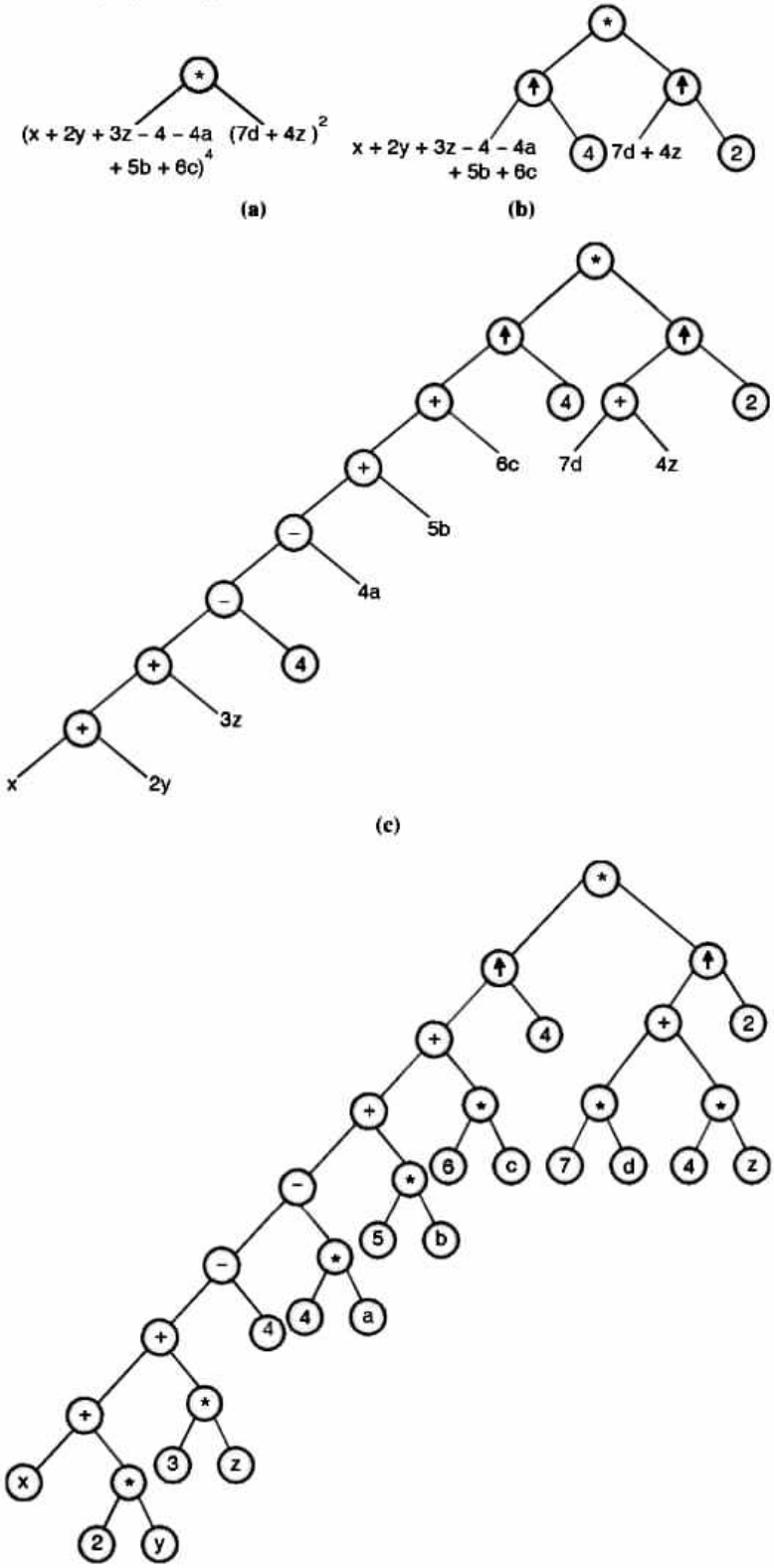
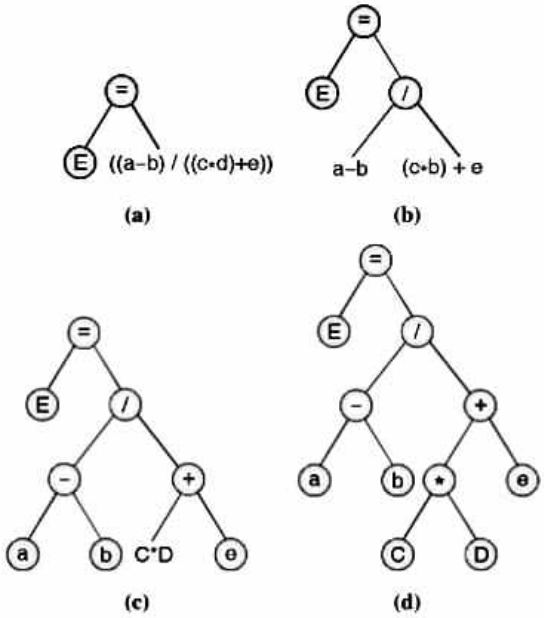


Fig. Ex. 6.8.4(vi)

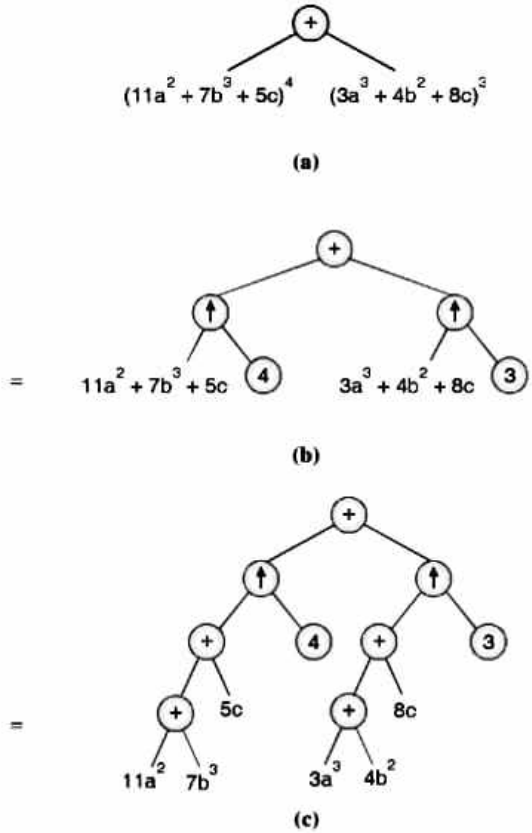
(vii) E : ((a – b) / (C * D) + e)



(a) (b) (c) (d)

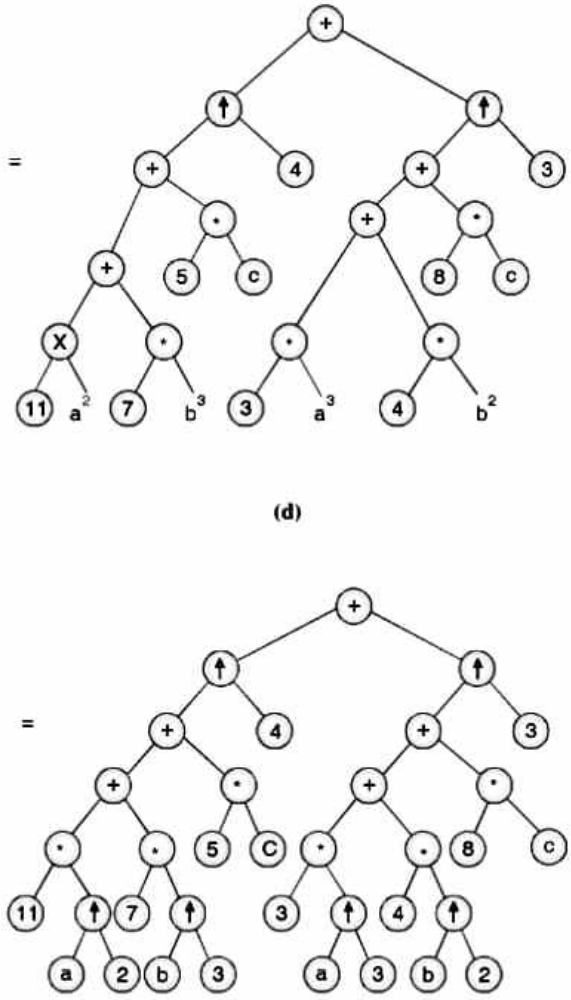
Fig. 6.8.4(vii)

(viii) $(11a^2 + 7b^3 + 5c)^4 + (3a^3 + 4b^2 + 8c)^3$



(a) (b) (c)

Fig. 6.8.4(viii) contd…

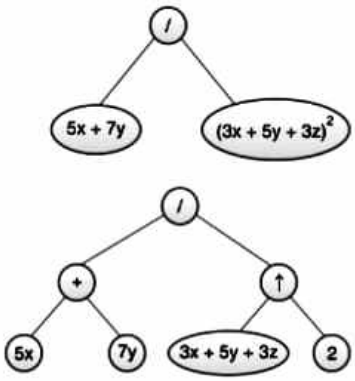


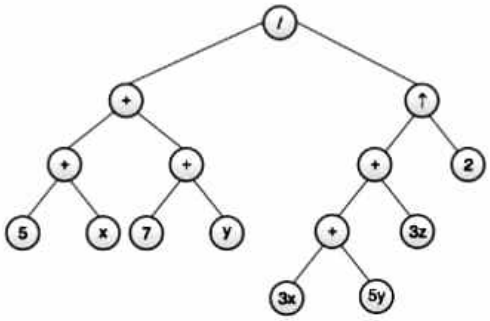
(d) (e)

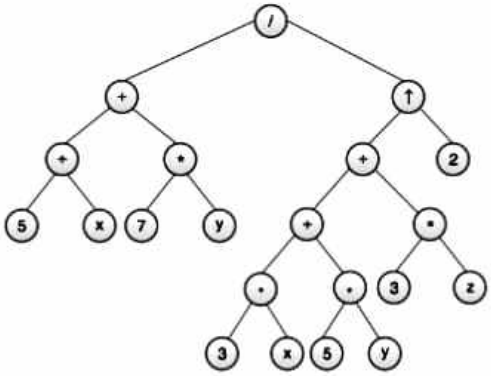
Fig. Ex. 6.8.4(viii)

Example 6.8.5 : Draw the tree structure for the following expression :
$(5x + 7y) / (3x + 5y + 3z)^2$ **W-15**

Solution :







Example 6.8.6 : Draw expression tree for the following :

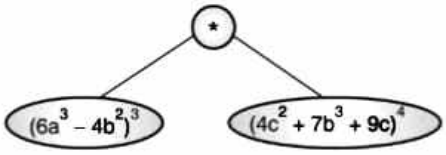
(i) $(6a^3 - 4b^2)^3 * (4c^2 + 7b^3 + 9c)^4$

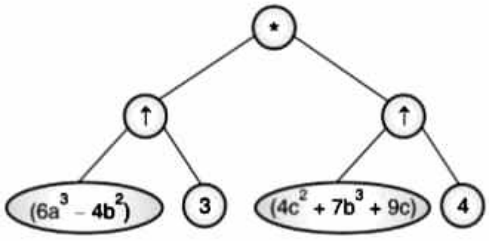
(ii) $(a^2 + 2ab + b^2)^3 + (c^2 - 2cd + d^2)^3$

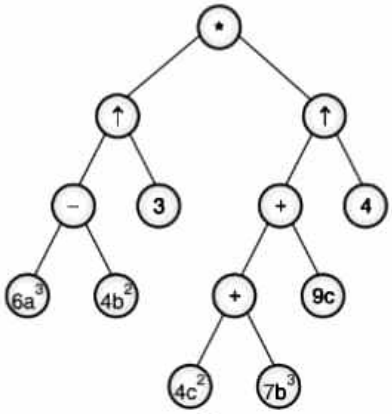
S-16

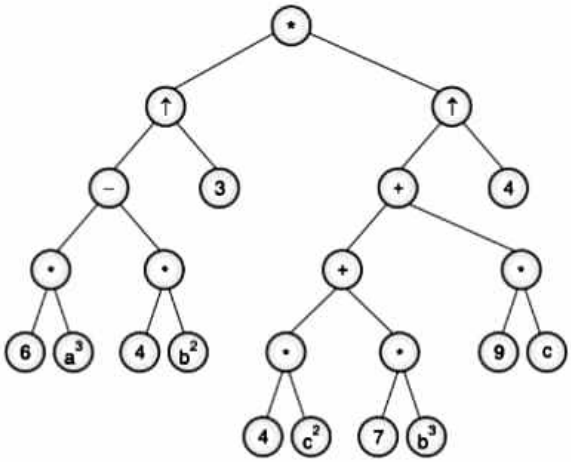
Solution :

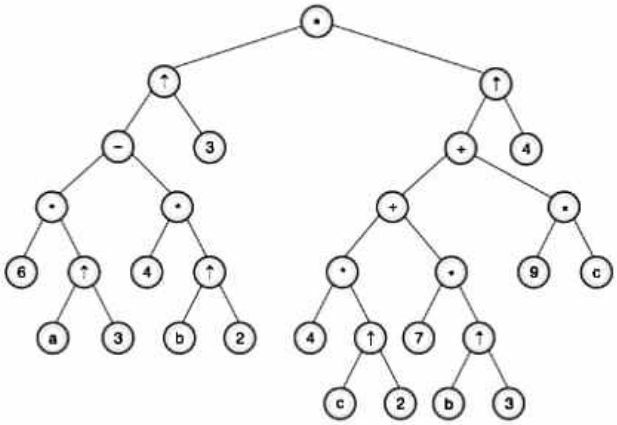
(i) $(6a^3 - 4b^2)^3 * (4c^2 + 7b^3 + 9c)^4$



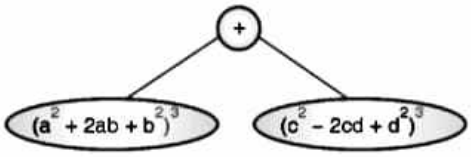


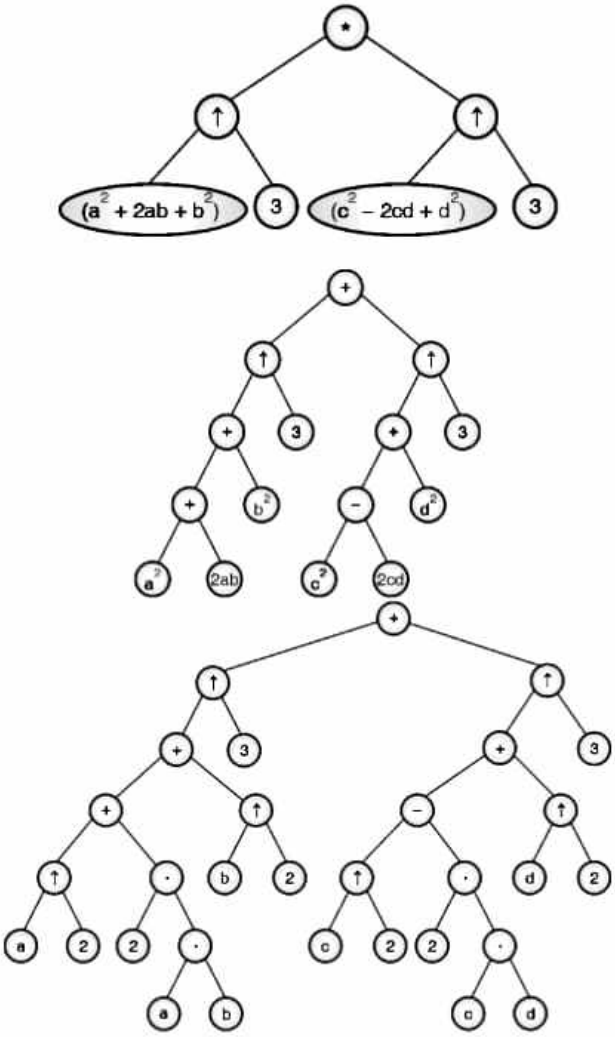






(ii) $(a^2+2ab+b^2)^3 + (c^2-2cd+d^2)^3$





Example 6.8.7 : Draw tree structure for following expression.
[3A + 7B] – [(6D – 4E) ∧ 6C] **W-16**

Solution :

Expression is

(3 * A + 7 * B) – [(6 * D – 4 * E) ∧ 6 * C]

Tree for above expression is :

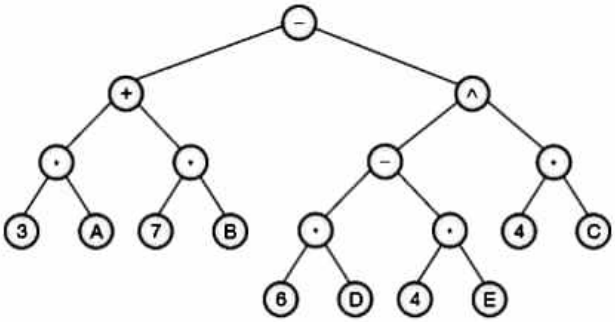
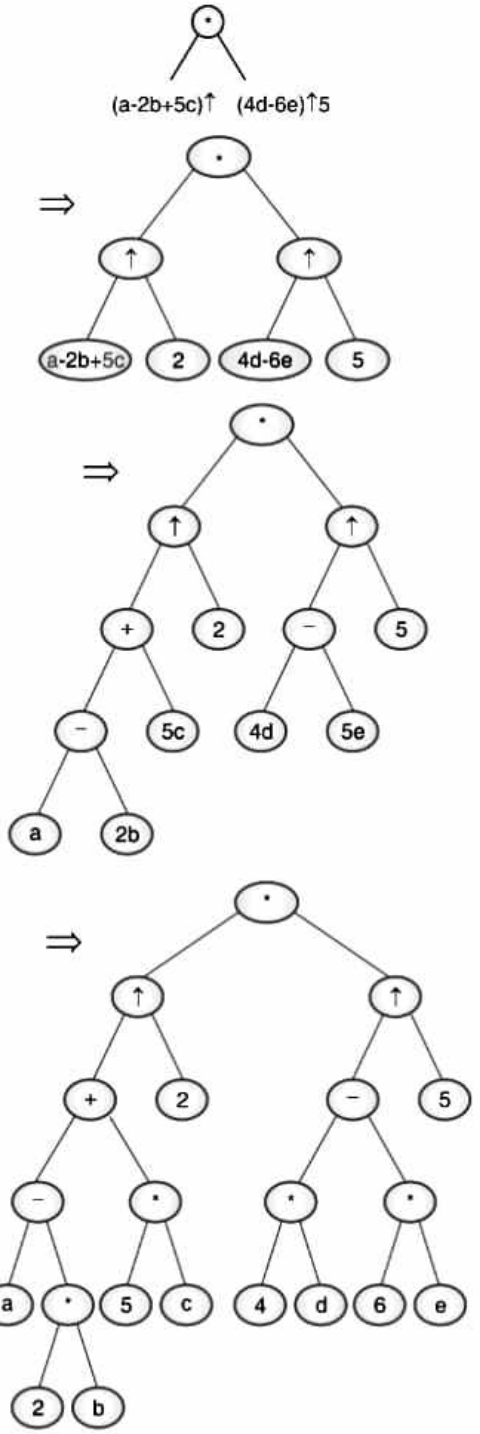


Fig. Ex. 6.8.7

Example 6.8.8 : Draw tree for given expression and find inorder, preorder and postorder traversal. $(a - 2b + 5c)^2 (4d - 6e)^5$. **S-17**

Solution : Expressions tree :



Preorder : *↑+–a*2b*5c

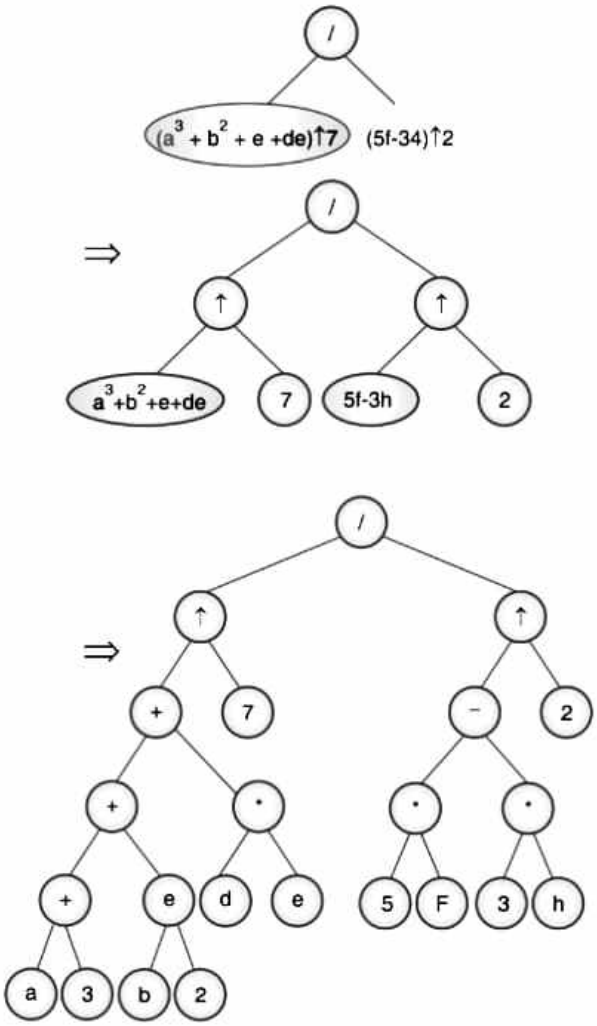
2↑–*4d*6e5

Inorder : a – 2*b + 5c ↑2 * 4*d –6*e↑5

Postorder : a 2 b * – 5c * + 2↑ 4d * 6 e * – 5 ↑ *

Example 6.8.9 : Draw an expression tree for following expression.
$(a^3 + b^2 + c + de)^7/(5f - 3h)^2$ **W-17**

Solution :



6.9 MSBTE Questions and Answers

➔ Summer 2008 – Total Marks 24

Q. 1 State the principles of pre-order and In-order traversal of binary tree.
(Sections 6.5.1 and 6.5.2) **(2 Marks)**

Q. 2 Perform inorder, preorder and postorder traversal of the following Binary Tree. (Refer Fig. Q. 2)
(Example 6.6.3) **(6 Marks)**

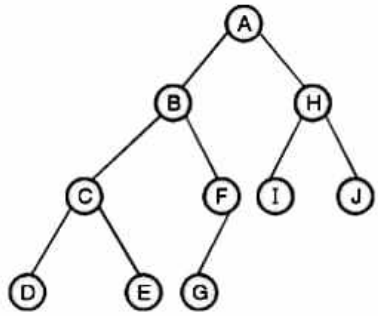


Fig. Q. 2

Q. 3 Explain searching a value in binary search tree with example. **(Section 6.7.2(B))** **(4 Marks)**

Q. 4 Write a program for 'binary Search in 'C' language.
(Section 6.7.2(B)) **(4 Marks)**

Q. 5 Give stepwise procedure for deletion of a node in a binary tree.
(Section 6.7.2(F)) **(4 Marks)**

Q. 6 Give recursive algorithms for pre-order, in-order and post-order, traversal of binary tree.
(Section 6.5) **(4 Marks)**

➔ Winter 2008 – Total Marks 19

Q. 7 Define the terms related to binary tree : Level, Depth, Leaf node, Root node.
(Section 6.1.1) **(2 Marks)**

Q. 8 Explain the binary tree with suitable example and diagram. **(Section 6.2)** **(6 Marks)**

Q. 9 For the given binary tree in Fig. Q. 9, perform inorder, preorder and postorder traversal.
(Example 6.6.4) **(3 Marks)**

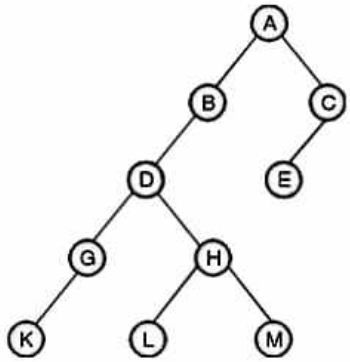


Fig. Q. 9

Q. 10 Write a 'C/C++' program for postorder traversal of binary tree. **(Section 6.5.3)** **(4 Marks)**

Q. 11 Draw the tree structure for the following expressions.

(i) $(2a + 5b)^3 (x - 7y)^4$

(ii) $(a - 3b) (2x - y)^3$

(Example 6.9.4) **(4 Marks)**

➔ Summer 2009 – Total Marks 26

Q. 12 For the given binary tree, perform Inorder, preorder and post order traversal. **(Example 6.6.5)(6 Marks)**

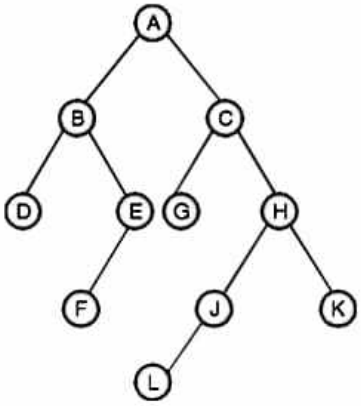


Fig. Q. 12

Q. 13 Enlist the types of binary tree and give their meaning. **(Section 6.4)** **(4 Marks)**

Q. 14 Construct a binary search tree from the given list of letters inserted in order into an empty binary search tree :

J,R,D,G,T,E,M,H,P,A,F,Q.

(Example 6.7.3) **(4 Marks)**

Q. 15 Explain threaded trees with diagram.

(Section 6.8) **(4 Marks)**

Q. 16 Draw the 2-tree corresponding to each of the following algebraic expressions :

$E_1 = (a - 3b) (2x - y)^3$

$E_2 = (2a + 5b)^3 (x - 7y)^4$

(Example 6.8.4) **(4 Marks)**

Q. 17 Describe in brief, the terms related to Binary Tree: Level, Depth, Leaf Node, Root Node.

(Section 6.1.1) **(4 Marks)**

➔ Winter 2009 – Total Marks 10

Q. 18 Explain the terms with the help of diagram.

(i) Siblings (ii) Leaf Node

(Section 6.1.1) **(2 Marks)**

Q. 19 With the help of example, describe the process of in order traversal of binary tree.

(Section 6.5.2) **(4 Marks)**

Q. 20 Draw tree structure for the following expression.

$(x + 2y + 3z + 4a + 5b + 6c)^4 * (7d + 4z)^2$

(Example 6.8.4) **(4 Marks)**

➔ Summer 2010 – Total Marks 28

Q. 21 Write an algorithm to traverse the tree in preorder and post order with example.

(Sections 6.5.1 and 6.5.3) **(6 Marks)**

Q. 22 Draw the tree structure for the following expression :

$(x + 2y + 3z - 4 - 4a + 5b + 6c)^4 * (7d + 4z)^2$

(Example 6.8.4) **(6 Marks)**

➔ Winter 2010 – Total Marks 28

Q. 23 Enlist operations on trees.

(sections 6.7.2 and 6.5) **(2 Marks)**

Q. 24 Describe complete binary tree and extended binary tree along with an example.

(Sections 6.4.2 and 6.4.5) **(6 Marks)**

Q. 25 For the following arithmetic expression draw the binary tree.

(i) E : ((a – b) / (C C * D) + e)

(ii) E : $(2a + 5b)^3 (x - 7y)^4$

(Example 6.9.4) **(4 Marks)**

Q. 26 Suppose following eight numbers are added in order into an empty binary search tree T.

50, 33, 44, 22, 77, 35, 60, 40

Draw the tree T and search an item 20 in the tree.

(Example 6.7.4) **(4 Marks)**

Q. 27 Describe with an example sequential representation and linked representation of trees in memory.

(Sections 6.2.1 and 6.2.2) **(4 Marks)**

Q. 28 Write an algorithm for binary tree along with an example. **(Section 6.2)** **(4 Marks)**

Q. 29 Describe weight balanced trees and height balanced trees along with an example.

(Sections 6.10 and 6.11) **(4 Marks)**

Q. 30 Consider the following tree. Write it's

- Preorder traversal sequence
- Inorder traversal sequence
- Post order traversal sequence

Refer Fig. Q. 30 **(Example 6.6.5)** **(4 Marks)**

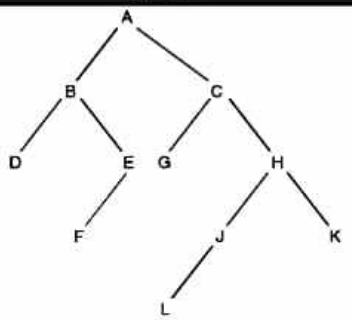


Fig. Q. 30

➔ Summer 2011 – Total Marks 20

Q. 31 Explain binary tree with an example ? **(Section 6.2)** **(4 Marks)**

Q. 32 Explain any eight terms of tree terminology. **(Section 6.1.1)** **(8 Marks)**

Q. 33 Define tree traversal along with preorder, inorder and post order algorithms. Draw the tree structure of the expression given below.

Z * (A + B) * C/D

(Sections 6.5, 6.5.1, 6.5.2 and 6.5.3, Example 6.6.7) **(8 Marks)**

➔ Winter 2011 – Total Marks 20

Q. 34 Explain insertion and deletion of node on binary search tree. **(Sections 6.7.2(D) and 6.7.2(F))** **(10 Marks)**

Q. 35 Explain Inorder, Preorder and Postorder traversal. **(Section 6.5)** **(10 Marks)**

➔ Summer 2012 – Total Marks 22

Q. 36 What is Binary tree ? **(Section 6.2)** **(2 Marks)**

Q. 37 For the given binary tree perform inorder, preorder and postorder traversal and state any two applications of binary tree. Refer Fig. Q. 38. **(Example 6.6.6 and Example 6.8.1)** **(8 Marks)**

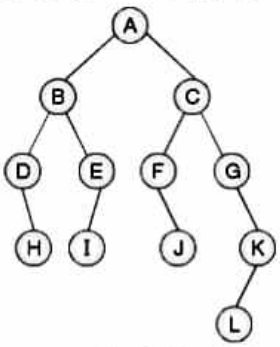


Fig. Q. 37

Q. 38 What is tree ? Define any four terminologies related to tree and draw the tree structure for following expression.

$(11a^2 + 7b^3 + 5c)^4 + (3a^3 + 4b^2 + 8c)^3$

(Sections 6.1 and 6.1.1, Example 6.8.4) **(8 Marks)**

Q. 39 Give the algorithm for deletion of a node in binary search tree. **(Section 6.7.2(F))** **(4 Marks)**

➔ Winter 2012 – Total Marks 16

Q. 40 Describe following binary search tree operations.

(a) Insertion of node

(b) Deletion of node

(Sections 6.7.2(D) and 6.7.2(F)) **(4 Marks)**

Q. 41 Explain binary tree with suitable example and diagram. **(Section 6.2)** **(4 Marks)**

Q. 42 Consider the following tree. Write its

(a) In-order traversal sequence.

(b) Pre-order traversal sequence.

(c) Post order traversal sequence.

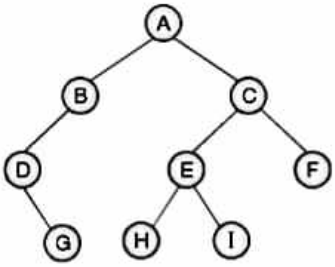


Fig. Q. 44

(Example 6.6.8) **(4 Marks)**

➔ Summer 2013 – Total Marks 28

Q. 43 Define the term 'complete binary tree' and 'fully binary tree'. **(Sections 6.4.1 and 6.4.2)** **(2 Marks)**

Q. 44 Define the term tree and forest. **(Sections 6.1 and 6.3.1)** **(2 Marks)**

Q. 45 raw the tree structure for the following expression.

$(x + 2y + 3z - 4 - 4a + 5b + 6c)^4 * (7d + 4z)^2$

(Example 6.8.4(vi)) **(8 Marks)**

Q. 46 Construct the 'binary search tree' using given list of letters and search in order into an empty binary search tree.

J, R, D, G, T, E, M, H, P, A, F, Q

(Example 6.7.3) **(4 Marks)**

Q. 47 Give stepwise procedure for deletion of a node in a Binary tree. **(Section 6.7.2(F))** **(4 Marks)**

Q. 48 Explain the term with the help of diagram siblings, leaf node, level of tree, depth of a tree.

(Section 6.1.1) **(4 Marks)**

Q. 49 Perform inorder, preorder and postorder traversal of the following Binary tree. Ref. Fig. Q. 49.

(Example 6.6.3) **(4 Marks)**

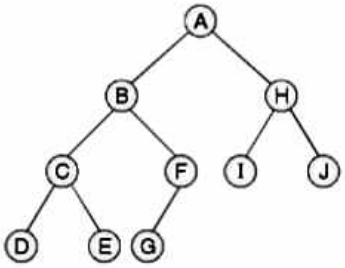


Fig. Q. 49

➔ Winter 2013 – Total Marks 28

Q. 50 Define the given tree terminology :

(1) Degree of node

(2) Leaf node

(Section 6.1.1) **(2 Marks)**

Q. 51 Define the term General tree and Binary tree.

(Sections 6.1 and 6.2) **(2 Marks)**

Q. 52 Describe representation of binary tree.

(Sections 6.2.1 and 6.2.2) **(4 Marks)**

Q. 53 Write an algorithm for inorder traversal of binary tree.

(Section 6.5.2) **(4 Marks)**

Q. 54 Draw the tree structure for expression given below:

$(a - 3b)(2x - y)^3$ **(Example 6.8.4(ii))** **(4 Marks)**

Q. 55 Draw a binary search tree for the number given:

50, 33, 44, 22, 77, 35, 60, 40.

(Example 6.7.4) **(4 Marks)**

Q. 56 What is binary tree ? For given tree perform preorder postorder and inorder traversal. Refer Fig. 56. **(Section 6.2, Example 6.6.12)**

(8 Marks)

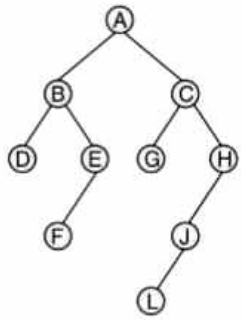


Fig. Q. 56

➔ Summer 2014 – Total Marks 29

Q. 57 List types of trees and explain any one.

(Section 6.4) **(4 Marks)**

Q. 58 Describe the concept of binary tree and its application. **(Section 6.2)** **(4 Marks)**

Q. 59 What is meant by binary search tree with example ?

(Section 6.7) **(4 Marks)**

Q. 60 Explain the following terms w.r.t. tree with diagram :

(i) Degree of a node **(Section 6.1.1)**

(ii) Degree of a tree

(iii) Level of a node **(Section 6.1.1)**

(iv) Leaf node **(Section 6.1.1)**

(v) Height of a tree. **(Section 6.2.2)** **(5 Marks)**

Ans. :

(ii) Degree of a tree :

It is the maximum degree of any of its nodes.

Q. 61 Write an algorithm for post order traversal in a tree.

(Section 6.5.3) **(8 Marks)**

Q. 62 Write the preorder and postorder traversal of a tree.

(Example 6.6.13) **(4 Marks)**

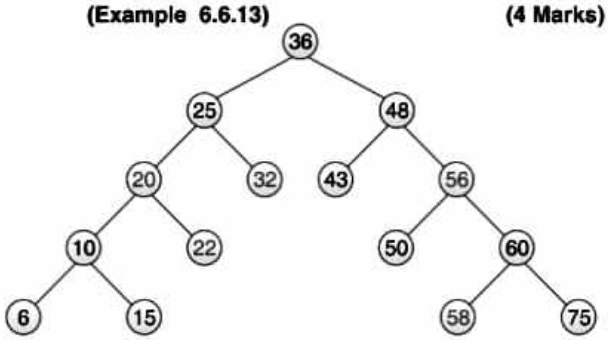


Fig. Q. 62

➔ Winter 2014 – Total Marks 22

Q. 63 Explain following tree terminology with the help of diagram.

(1) Siblings **(Section 6.1.1)**

(2) Height of Tree **(Section 6.2.2)** **(2 Marks)**

Q. 64 Define the term related to binary tree : root node, leaf node, level and depth. **(Section 6.1.1)(4 Marks)**

Q. 65 Perform in-order, post-order and pre-order traversal of binary tree, Refer Fig. Q. 65.
(Example 6.6.14) **(4 Marks)**

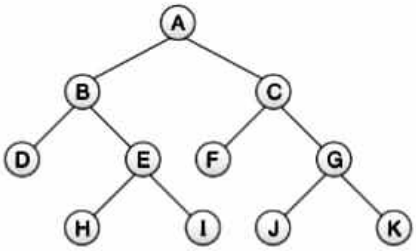


Fig. Q. 65

Q. 66 Describe expression tree with an example.
(Section 6.8.1) **(4 Marks)**

Q. 67 Write a program to count the number of node in a binary search tree.**(Section 6.3.1)** **(8 Marks)**

➔ Summer 2015 – Total Marks 28

Q. 1(A)(f) List operations on trees.
(Ans. : Refer section 6.5) **(2 Marks)**

Q. 1(A)(g) What is binary tree ?
(Ans. : Refer section 6.2) **(2 Marks)**

Q. 2(d) Explain the terms with the help of diagram :

i) Siblings ii) Leaf Node

(Ans. : Refer section 6.1.1) **(4 Marks)**

Q. 3(d) Explain inorder, preorder and postorder traversal.
(Ans. : Refer sections 6.5.1, 6.5.2 and 6.5.3) **(4 Marks)**

Q. 3(e) Draw the tree structure for the following expressions :

i) $(2a + 5b)^3 (x - 7y)^4$

ii) $(a - 3b) (2x - y)^3$.

(Ans. : Refer Example 6.8.4(i) and (ii)) **(4 Marks)**

Q. 4(f) Consider the following tree. Write its :

a) In-order traversal sequence

b) Pre-order traversal sequence

c) Post-order traversal sequence.

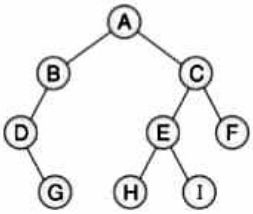


Fig. 1-Q. 4(f)

(Ans. : Refer Example 6.6.9) **(4 Marks)**

Q. 6(b) What is tree ? Define any four terminologies related to tree and draw the tree structure for following expression

$(11a^2 + 7b^3 + 5c)^4 + (3a^3 + 4b^2 + 8c)^3$.

(Ans. : Refer sections 6.1, 6.1.1 and Example 6.8.4(viii)) **(8 Marks)**

➔ Winter 2015 – Total Marks 27

Q. 1(A)(e) Define the terms : root-node, leaf, sibling.
(Ans. : Refer section 6.1.1) **(1 Mark)**

Q. 1(A)(f) Define tree traversal. List different types of tree traversal.
(Ans. : Refer sections 6.5, 6.5.1, 6.5.2 and 6.5.3) **(2 Marks)**

Q. 2(d) Describe the process of pre-order traversal of binary tree. Give one example.
(Ans. : Refer section 6.5.1) **(4 Marks)**

Q. 3(d) Define tree. Create a binary tree for following data : 12, 25, 14, 8, 3, 5.
(Ans. : Refer section 6.1 and Example 6.7.5) **(4 Marks)**

Q. 3(e) Write a 'C' program to traverse the tree in in-order. *(Ans. : Refer section 6.5.2)* **(4 Marks)**

Q. 4(f) Draw the tree structure for the following expression :
$(5x + 7y) / (3x + 5y + 3z)^2$
(Ans. : Refer Example 6.8.5) **(4 Marks)**

Q. 6(b) Compare between General Tree and Binary Tree (any 4 points).
(Ans. : Refer section 6.3.1) **(8 Marks)**

➔ Summer 2016 – Total Marks 28

Q. 1(f) Define the following with respect to tree :

(i) Ancestor

(ii) Depth *(Ans. : Refer section 6.1.1)*

(iii) Degree *(Ans. : Refer section 6.1.1)*

(iv) Path *(Ans. : Refer section 6.1.1)* **(4 Marks)**

Ans. :

(i) **Ancestor :** Every node, starting from the root node and going upto the father of the current node are ancestor nodes.

Q. 2(e) Draw binary search tree using following elements :

35,15, 40, 7, 10, 100, 28, 82, 53, 25, 3.

(Ans. : Refer Example 6.7.6) **(4 Marks)**

Q. 3(c) Differentiate between tree and graph. Traverse the following tree in Inorder, Preorder and Postorder.

(Ans. : Refer Example 6.6.10) **(8 Marks)**

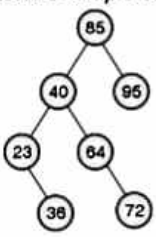


Fig. 1-Q. 3(c)

Q. 4(e) Describe binary tree representation.

(Ans. : Refer sections 6.2 and 6.2.1) **(4 Marks)**

Q. 6(b) Draw expression tree for the following :

(i) $(6a^3 - 4b^2)^3 * (4c^2 + 7b^3 + 9c)^4$

(ii) $(a^2 + 2ab + b^2)^3 + (c^2 - 2cd + d^2)^3$

(Ans. : Refer section 6.8.6) **(8 Marks)**

➔ Winter 2016 – Total Marks 26

Q. 1(A)(5) Define following w.r.t. tree :

(a) Ancestor (b) Descendant nodes **(2 Marks)**

Ans. :

(a) **Ancestor :** Every node, starting from the root node and going upto the father of the current node are ancestor nodes.

(b) **Descendant node :** All immediate successors of a node are its descendants.

Q. 2(d) Describe concept of Binary tree. State its application.

(Ans. : Refer section 6.2) **(4 Marks)**

Ans. :

Applications of Binary tree :

1. Implementing routing table in router.
2. Sorting
3. Searching
4. Expression Evaluation

Q. 3(d) Write algorithm for In-order traversal for Binary tree. Demonstrate with suitable example.

(Ans. : Refer section 6.5.2) **(4 Marks)**

Q. 3(e) Draw tree structure for following expression.

[3A + 7B] – [(6D – 4E) ∧ 6C]

(Ans. : Refer Example 6.8.7) **(4 Marks)**

Q. 4 (f) Describe general tree and binary tree.

(Ans. : Refer sections 6.3 and 6.2) **(4 Marks)**

Q. 6 (b) Write a program to calculate number node in binary search tree.

*(Ans. : Refer Q. 67 from Winter 2014)***(8 Marks)**

➔ Summer 2017 – Total Marks 28

Q. 1(A)(g) Differentiate between tree and graph (Min. 2 points).

(Ans. : Refer Q. 3(c) from Summer 2016) **(8 Marks)**

Q. 2(e) Write algorithm for preorder traversal of binary tree. *(Ans. : Refer section 6.5.1)* **(4 Marks)**

Q. 3(f) Draw a binary search tree for given sequence and write postorder traversal of tree.

10 5 8 9 7 6 2 15.

(Ans. : Refer Example 6.7.7) **(4 Marks)**

Q. 4(e) Define : **(4 Marks)**

i) Sibling *(Ans. : Refer section 6.1.1)*

ii) Depth of tree *(Ans. : Refer section 6.1.1)*

iii) Complete binary tree *(Ans. : Refer section 6.4.2)*

iv) Degree of tree. *(Ans. : Refer section 6.1.1)*

Q. 6 (b) Draw tree for given expression and find inorder, preorder and postorder traversal.

$(a - 2b + 5c)^2 (4d - 6e)^5$.

(Ans. : Refer Example 6.8.8) **(8 Marks)**

➔ Winter 2017 – Total Marks 18

Q. 1(A)(e) Define tree. State its two types.
(Ans. : Refer section 6.1) **(2 Marks)**

Ans. : Type of tree : 1. General tree 2. Binary tree.

Q. 1(A)(h) Define following terms :

(i) Height of tree *(Ans. : Refer section 6.1.1)*

(ii) Degree of a node
(Ans. : Refer section 6.1.1) **(2 Marks)**

Q. 2(d) With suitable example, describe preorder traversal of tree. Also write algorithm for preorder traversal.
(Ans. : Refer section 6.5.1) **(2 Marks)**

Q. 3(d) Differentiate between general tree and binary tree. (any four points).
(Ans. : Refer section 6.3.1) **(4 Marks)**

Q. 3(e) Define binary tree. Traverse the following tree in inorder, preorder and postorder.
(Ans. : Refer section 6.2 and Example 6.6.11) **(4 Marks)**

Q. 4(f) Draw an expression tree for following expression.
$(a^3 + b^2 + c + de)^7/(5f - 3h)^2$
(Ans. : Refer Example 6.8.8) **(4 Marks)**

❑❑❑

CHAPTER 7

Unit - V

Graphs

Syllabus Topics

Introduction to Graph terminologies : graph, node (Vertices), arcs (edge), directed graph, undirected graph, in-degree, out-degree, adjacent, successor, predecessor, relation, path, sink, articulation point. Adjacency List, Adjacency Matrix of directed/undirected graph.

Syllabus Topic : Introduction to Graph Terminologies - Graph, Node (Vertices), Arcs (Edge)

7.1 Terminology and Representation

7.1.1 Definition

➔ **(MSBTE : S-08, S-11, S-12)**

Q. 7.1.1	**Describe in brief, the terms related to graph : nodes, edges, in-degree, out-degree. (Refer section 7.1.1)**	**S-08**
Q. 7.1.2	**Define Graph? (Refer section 7.1.1)**	**S-11**
Q. 7.1.3	**What is graph ? (Refer section 7.1.1)**	**S-12**

A graph G is a set of vertices (V) and set of edges (E). The set V is a finite, nonempty set of vertices. The set E is a set of pair of vertices representing edges.

G = (V, E)

V(G) = Vertices of graph G

E(G) = Edges of graph G

An example of graph is shown in Fig. 7.1.1.

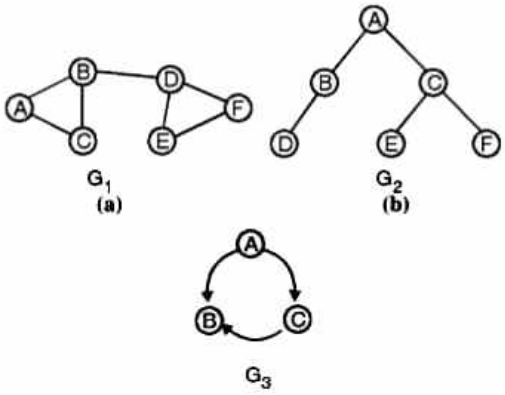


Fig. 7.1.1 : Graphs

The set representation for each of these graphs is given by

$V(G_1)$ = {A, B, C, D, E, F}

$V(G_2)$ = {A, B, C, D, E, F}

$V(G_3)$ = {A, B, C}

$E(G_1)$ = {(A, B),(A, C),(B, C),(B, D),(D, E),(D, F),(E, F)}

$E(G_2)$ = {(A,B),(A, C),(B, D),(C, E),(C, F)}

$E(G_3)$ = {(A, B),(A, C),(C, B)}

Syllabus Topic : Undirected Graph

7.1.2 Undirected Graph

➔ **(MSBTE : S-11, W-13)**

Q. 7.1.4 Enlist and explain two types of graph with examples. (Refer section 7.1.2) **S-11**

Q. 7.1.5 Describe undirected graph with diagram. (Refer section 7.1.2) **W-13**

A graph containing unordered pair of vertices is called an undirected graph. In an undirected graph, pair of vertices(A, B) and(B, A) represent the same edge.

The set of vertices V = {1, 2, 3, 4, 5}. The set of edges E = {(1, 2), (1, 3), (1, 5), (2, 3), (2, 4), (3, 4), (4, 5)}.

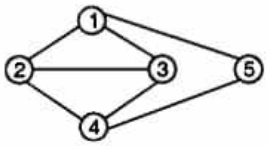


Fig. 7.1.2 : Example of an undirected graph

Syllabus Topic : Directed Graph

7.1.3 Directed Graph

➔ **(MSBTE : S-10, S-11, S-12, W-13, W-14)**

Q. 7.1.6 Define following terms with respect to graph :

(i) Directed graph (ii) Predecessor

(iii) Succesor

(Refer sections 7.1.3 and 7.1.19) **S-10**

Q. 7.1.7 Enlist and explain two types of graph with examples (Refer section 7.1.3) **S-11**

Q. 7.1.8 Explain following term with respect to graph using suitable example : Directed graph (Refer section 7.1.3) **S-12**

Q. 7.1.9 Describe directed graph with diagram. (Refer section 7.1.3) **W-13**

Q. 7.1.10 Define directed edge of a tree. (Refer section 7.1.3) **W-14**

A graph containing ordered pair of vertices is called a directed graph. If an edge is represented using a pair of vertices (V_1, V_2) then the edge is said to be directed from V_1 to V_2.

V_1 is called predecessor of V_2 and V_2 is successor of V_1

The first element of the pair, V_1 is called the start vertex and the second element of the pair, V_2 is called the end vertex. In a directed graph, the pairs (V_1, V_2) and (V_2, V_1) represent two different edges of a graph. Example of a directed graph is shown in Fig. 7.1.3.

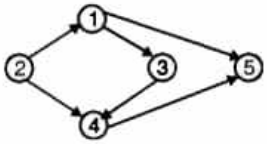


Fig. 7.1.3 : Example of a directed graph

The set of vertices V = {1, 2, 3, 4, 5, 5}.

The set of edges E = {(1, 3), (1, 5), (2, 1), (2, 4), (3, 4), (4, 5)}.

7.1.4 Complete Graph

An undirected graph, in which every vertex has an edge to all other vertices is called a complete graph.

A complete graph with N vertices has $\frac{N(N-1)}{2}$ edges.

Example of a complete graph is shown in Fig. 7.1.4.

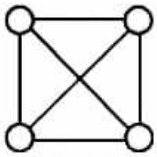


Fig. 7.1.4 : A complete graph

7.1.5 Weighted Graph

➔ **(MSBTE : S-10, S-12)**

Q. 7.1.11 Explain following term with respect to graph using suitable example : Weighted graph (Refer section 7.1.5) **S-10, S-12**

A weighted graph is a graph in which edges are assigned some value. Most of the physical situations are shown using weighted graph. An edge may represent a highway link between two cities.

The weight will denote the distance between two connected cities using highway. Weight of an edge is also called its cost. The graph of Fig. 7.1.5 is an example of a weighted graph.

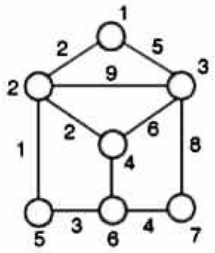


Fig. 7.1.5 : A weighted graph

Syllabus Topic : Adjacent Nodes

7.1.6 Adjacent Nodes

Two vertices V_1 and V_2 are said to be adjacent if there is an edge between V_1 and V_2.

Syllabus Topic : Path

7.1.7 Path

➔ **(MSBTE : W-15)**

Q. 7.1.12 Define the following terminologies with respect to graph : Path W-15

A path from vertex V_0 to V_n is a sequence of vertices V_0, V_1, V_2 ... V_{n-1}, V_n. Here, V_0 is adjacent to V_1, V_1 is adjacent to V_2 and V_{n-1} is adjacent to V_n. The **length** of a path is the number of edges on the path. A path with n vertices has a length of n – 1. A path is simple if all vertices on the path, except possibly the first and last, are distinct.

7.1.8 Cycle

A cycle is a simple path that begins and ends at the same vertex. Fig. 7.1.6 is an example of a graph with cycle.

A B D A is a cycle of length 3

B D C B is a cycle of length 3

A B C D A is a cycle of length 4

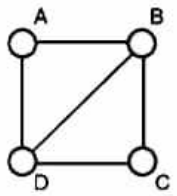


Fig. 7.1.6 : A graph with cycles

7.1.9 Connected Graph

A graph is said to be connected if there exists a path between every pair of vertices V_i and V_j.

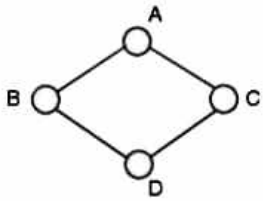


Fig. 7.1.7 : A connected graph

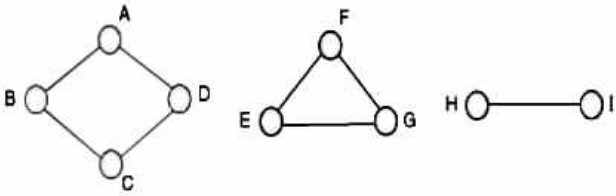


Fig. 7.1.8 : A disconnected graph with 3 components

The graph in Fig. 7.1.7 is a connected graph. But the graph in Fig. 7.1.8 is not connected as there is no path between D and E or between G and H. The graph of Fig. 7.1.8 consists of 3 connected components.

7.1.10 Subgraph

A subgraph of G is a graph G_1 such that V (G_1) is a subset of V (G) and E (G_1) is a subset of E (G).

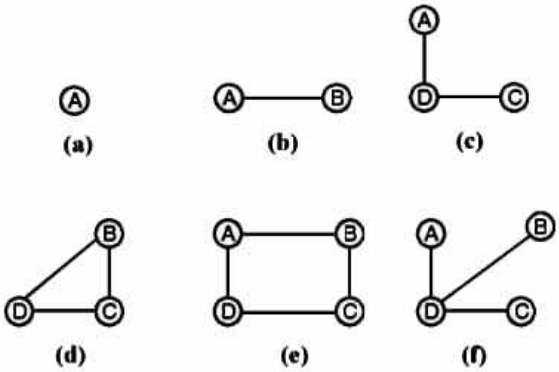


Fig. 7.1.9 : Some sub graphs of the graph of Fig. 7.1.6

7.1.11 Component

A component H of an undirected graph is a maximal connected subgraph. The graph of Fig. 7.1.10 has three components H_1, H_2 and H_3.

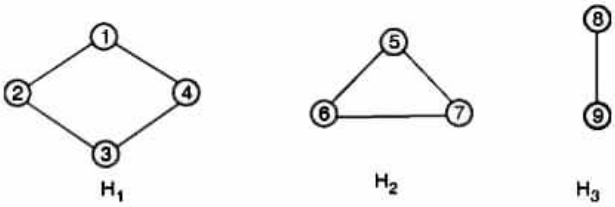


Fig. 7.1.10 : A graph with three components

Syllabus Topic : In-degree, Out-degree

7.1.12 Degree of a Vertex

➔ **(MSBTE : S-08, W-09, S-10, S-12, S-13, S-14, W-14, W-15, W-16, S-16, S-17)**

Q. 7.1.13 Describe in brief, the terms related to graph : nodes , edges , in-degree, out- degree.
(Refer section 7.1.12) **S-08, W-16**

Q. 7.1.14 Explain in-degree and out-degree of a node with example.
(Refer section 7.1.12) **W-09, S-13, S-14**

Q. 7.1.15 Define following term with respect to graph : In-degree of a node
(Refer section 7.1.12) **S-10, W-15**

Q. 7.1.16 Explain following terminologies with respect to graph using suitable example.
(i) in-degree (ii) Out-degree
(Refer section 7.1.12) **S-12, S-14, W-14**

Q. 7.1.17 Define the following terms in graph :
i) Source, ii) Isolated node
(Refer sections 7.1.12) **S-16**

Q. 71.18 Define following W.R.T. tree
(a) In-degree (b) Out-degree
(Refer section 7.1.12) **S-17**

The total number of edges linked to a vertex is called its degree. The **indegree** of a vertex is the total number of edges coming to that node. The **outdegree** of a node is the total number of edges going out from that node. A vertex, which has only outgoing edges and no incoming edges, is called a **source**. A vertex having only incoming edges and no outgoing edges is called a **sink**. When indegree of a vertex is one and outdegree is zero then such a vertex is called a **pendant** vertex. When the degree of a vertex is 0, it is an **isolated** vertex.

7.1.13 Self Edges or Self Loops

An edge of the form (V, V) is known as self edge or self loop. An example of self edge is shown in the Fig. 7.1.11.

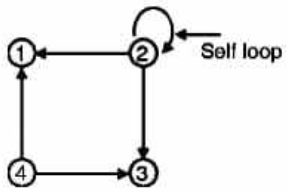


Fig. 7.1.11 : A graph with self loop

7.1.14 Multigraph

A graph with multiple occurrences of the same edge is known as a multigraph. An example of multigraph is shown in the Fig. 7.1.12.

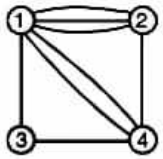


Fig. 7.1.12 : A multigraph

7.1.15 Tree

A tree is a connected graph without any cycle. The graphs of Fig. 7.1.13 are not trees as they contain cycles. The graphs of Fig. 7.1.14 are example of trees.

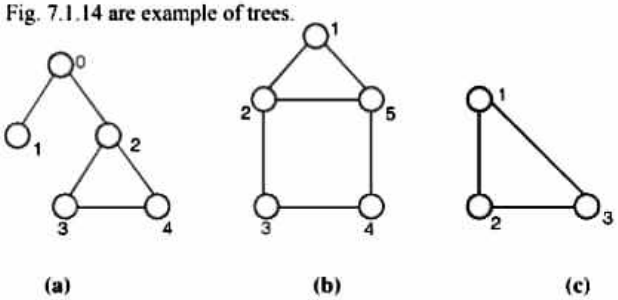


Fig. 7.1.13 : Graphs are not trees

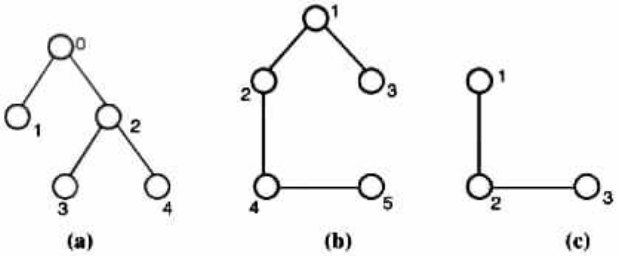


Fig. 7.1.14 : Example of trees

Note : Tree is a special case of a graph. When a tree is treated as a graph, it need not have a special vertex called the root.

7.1.16 Spanning Trees

A spanning tree of a graph G = (V, E) is a connected subgraph of G having all vertices of G and no cycles in it. If the graph G is not connected then there is no spanning tree of G. A graph may have multiple spanning trees. Fig. 7.1.16 gives some of the spanning trees of the graph shown in Fig. 7.1.15.

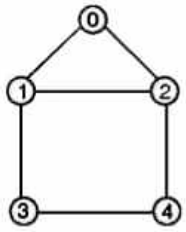
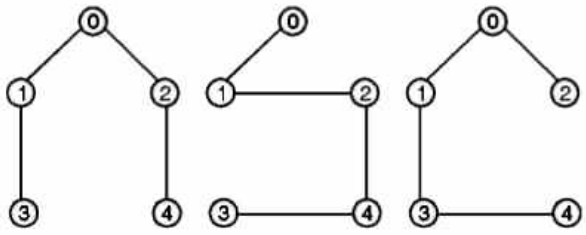


Fig. 7.1.15 : A sample connected graph

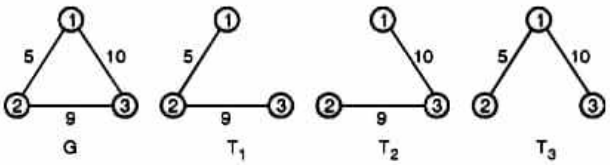


(a) (b) (c)

Fig. 7.1.16 : Spanning trees of the graph of Fig. 7.1.15

7.1.17 Minimal Spanning Tree

The cost of a graph is the sum of the costs of the edges in the weighted graph. A spanning tree of a group G = (V, E) is called a minimal cost spanning tree or simply the minimal spanning tree of G if its cost is minimum.



(a) (b) (c) (d)

Fig. 7.1.17 : An example of minimal spanning tree

G → A sample weighted graph

T_1 → A spanning tree of G with cost 5 + 9 = 14

T_2 → A spanning tree of G with cost 10 + 9 = 19

T_3 → A spanning tree of G with cost 5 + 10 = 15

Therefore, T_1 with cost 14 is the minimal cost spanning tree of the graph G.

7.1.18 Applications of Spanning Tree

Spanning trees have many applications. Some of them are :

- **Routing of a packet in a network :** A node can represent a router, located in a city and the link between routers can be represented using an edge. A spanning tree can represent a network with the minimum number of links.
- **Electrical network :** A spanning tree can be used to obtain an independent set of circuit equations for an electric network.

Syllabus Topic : Successor, Predecessor, Relation

7.1.19 Successor, Predecessor, Relation

➔ **(MSBTE : W-15)**

Q. 7.1.19 Define the following terminologies with respect to graph : Successor (Refer section 7.1.19) **W-15**

If a node V is reachable from the node u, then u is a predecessor of V and V is a successor of u. If there is a directed edge from u to v, then u is a direct predecessor of v and v is a direct successor of u.

Syllabus Topic : Sink

7.1.20 Sink

➔ **(MSBTE : S-16)**

Q. 7.1.20 Define the following term in graph :

i) Sink (Refer section 7.1.20) **S-16**

A sink vertex is a vertex with outdegree as zero.

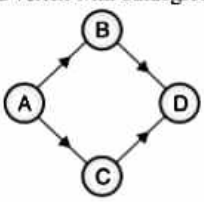


Fig. 7.1.18

In the above graph, the vertex D is a sink vertex as there are no outgoing lines from the vertex D.

Syllabus Topic : Articulation Point

7.1.21 Articulation Point

➔ **(MSBTE : S-16)**

Q. 7.1.21 Define the following terms in graph :

i) Articulation point. (Refer section 7.1.21) **S-16**

A vertex in an undirected connected graph is an articulation point iff removing it disconnects the graph.

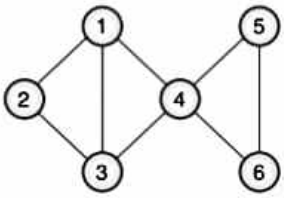


Fig. 7.1.19

In the graph above, the vertex 4 is an articulation point.

Syllabus Topic : Adjacency List, Adjacency Matrix of Directed/ Undirected Graph

7.2 Representation of Graphs

➔ **(MSBTE : S-17)**

Q. 7.2.1 Explain representation of graph in detail. (Refer section 7.2) **S-17**

7.2.1 Adjacency Matrix

➔ **(MSBTE : W-11, S-12, W-12)**

Q. 7.2.2 Describe application of graph in data structures. (Refer section 7.2.1) **W-11**

Q. 7.2.3 Explain Sequential Representation of Graphs. (Refer section 7.2.1) **W-11**

Q. 7.2.4 List applications of graph. (Refer section 7.2.1) **S-12**

Q. 7.2.5 Explain adjacency matrix representation of graph with suitable example. (Refer section 7.2.1) **S-12**

Q. 7.2.6 Compare the sequential representation method of graphs with linked representation method of graph. (Refer section 7.2.1) **W-12**

A two dimensional matrix can be used to store a graph. A graph G = (V, E) where V = {0, 1, 2, … n – 1} can be represented using a two dimensional integer array of size n × n.

int adj[20][20]; can be used to store a graph with 20 vertices.

adj[i][j] = 1, indicates presence of edge between two vertices i and j = 0, indicates absence of edge between two vertices i and j

- A graph is represented using a square matrix.
- Adjacency matrix of an undirected graph is always a symmetric matrix, i.e. an edge (i, j) implies the edge (j, i).
- Adjacency matrix of a directed graph is never symmetric adj[i][j] = 1, indicates a directed edge from vertex i to vertex j.

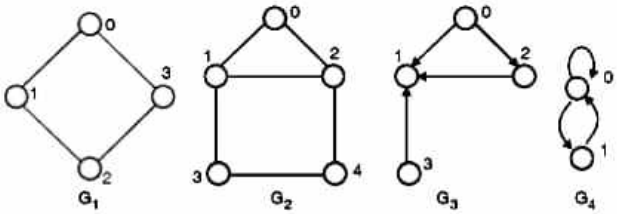


Fig. 7.2.1 : Graphs G_1, G_2, G_3 and G_4

	0	1	2	3
0	0	1	0	1
1	1	0	1	0
2	0	1	0	1
3	1	0	1	0

G_1 (Undirected graph)

	0	1	2	3	4
0	0	1	1	0	0
1	1	0	1	1	0
2	1	1	0	0	1
3	0	1	0	0	1
4	0	0	1	1	0

G_2 (Undirected graph)

Fig. 7.2.2 Contd…

	0	1	2	3
0	0	1	1	0
1	0	0	0	0
2	0	1	0	0
3	0	1	0	0

G_3 (Directed graph)

	0	1
0	1	1
1	1	0

G_4 (With self loop)

Fig. 7.2.2 : Adjacency matrix representation of graphs G_1, G_2, G_3 and G_4 of Fig. 7.2.1

Adjacency matrix representation of a weighted graph

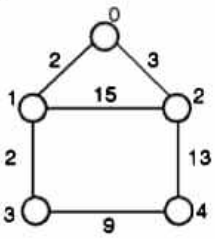
For weighted graph, the matrix adj[ ][ ] is represented as :

If there is an edge between vertices i and j then

Adj[i][j] = weight of the edge (i, j) otherwise,

Adj[i][j] = 0

Fig. 7.2.3 is an example of representation of a weighted graph.



(a)

	0	1	2	3	4
0	0	2	3	0	0
1	2	0	15	2	0
2	3	15	0	0	13
3	0	2	0	0	9
4	0	0	13	9	0

(b)

Fig. 7.2.3 : A weighted graph and its adjacency matrix

- Adjacency matrix representation of graphs is very simple to implement.
- Memory requirement : Adjacency matrix representation of a graph wastes lot of memory space. Such matrices are found to be very sparse. Above representation requires space for n^2 elements for a graph with n vertices. If the graph has e number of edges then $n^2 - e$ elements in the matrix will be 0.
- Presence of an edge between two vertices V_i and V_j can be checked in constant time.

  if(adj[i][j] = = 1)

  edge is present between vertices i and j

  else

  edge is absent between vertices i and j.
- Degree of a vertex can easily be calculated by counting all non-zero entries in the corresponding row of the adjacency matrix.

7.2.2 Adjacency List

➔ **(MSBTE : W-08, W-11, W-12, S-13)**

Q. 7.2.7 Explain the link representation of a Graph with suitable example. (Refer section 7.2.2) **W-08, S-13**

Q. 7.2.8 Explain linked representation of graphs. (Refer section 7.2.2) **W-11**

Q. 7.2.9 Compare the sequential representation method of graphs with linked representation method of graph. (Refer section 7.2.2) **W-12**

A graph can be represented using a linked list. For each vertex, a list of adjacent vertices is maintained using a linked list. It creates a separate linked list for each vertex V_i in the graph G = (V, E)

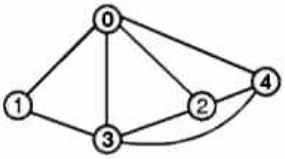


Fig. 7.2.4 : A graph

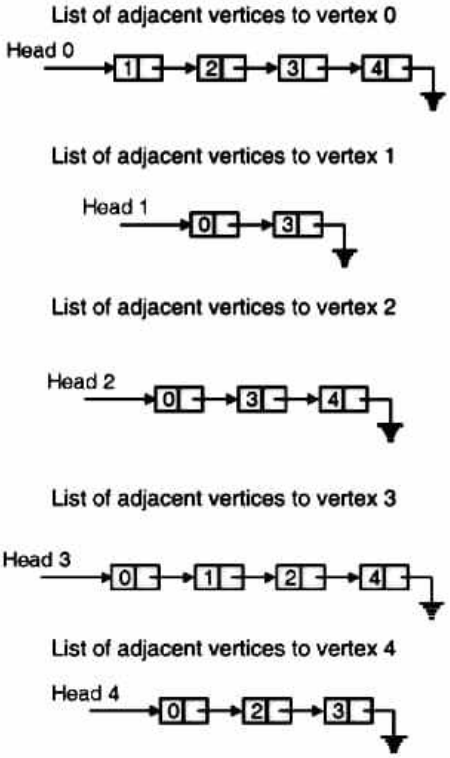


Fig. 7.2.5 : Adjacency list for each vertex of graph of Fig. 7.2.4

Adjacency list of a graph with n nodes can be represented by an array of pointers. Each pointer points to a linked list of the corresponding vertex. Fig. 7.2.6 shows the adjacency list representation of graph of Fig. 7.2.4.

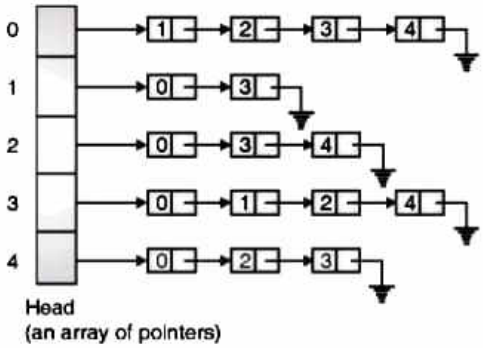


Fig. 7.2.6 : Adjacency list representation of the graph of Fig. 7.2.4

- Adjacency list representation of a graph is very memory efficient when the graph has a large number of vertices but very few edges.
- For an undirected graph with n vertices and e edges, total number of nodes will be n + 2e.
- If e is large then due to overhead of maintaining pointers, adjacency list representation does not remain cost effective over adjacency matrix representation of a graph.
- Degree of a node in an undirected graph is given by the length of the corresponding linked list.
- Finding in-degree of a directed graph represented using adjacency list will require O(e) comparisons. Lists pointed by all vertices must be examined to find the indegree of a node in a directed graph.
- Checking the existence of an edge between two vertices i and j is also time consuming. Linked list of vertex i must be searched for the vertex j.

A graph can be represented using a structure as defined below

```
# define MAX 30     /* graph has maximum of 30 nodes */
typedef struct node
{
    struct node * next;
    int vertex;
}node;
node * head[MAX];
```

If a weighted graph is to be represented using a adjacency list, then structure "node" should be modified to include the weight of a edge.

Thus the definition of 'struct node' is modified as below :

```
typedef struct node
{
    struct node * next;
    int vertex;
    int weight;
}node;
```

7.2.3 Examples on Graph Representation

Example 7.2.1 : For the following graph obtain :
(i) The in degree and out degree of each vertex,
(ii) Its adjacency matrix
(iii) Its adjacency list representation.
W-09, S-14

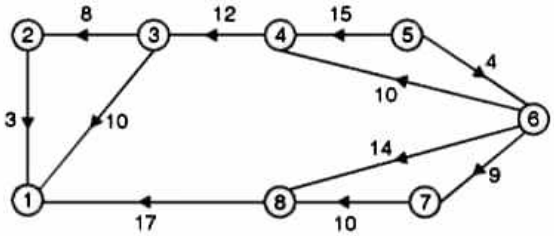


Fig. Ex. 7.2.1

Solution :

(i)

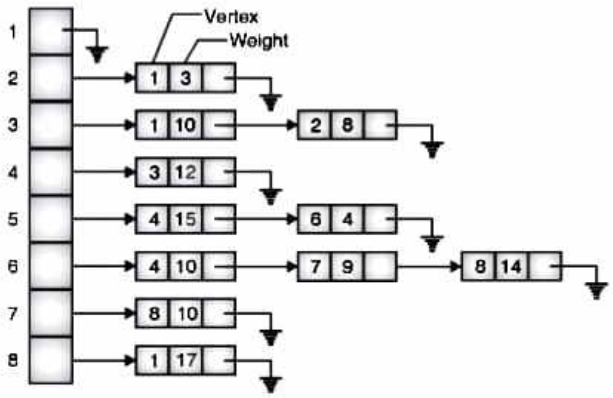
Vertex No.	Indegree	Outdegree
1	3	0
2	1	1
3	1	2
4	2	1
5	1	1
6	1	3
7	1	1
8	2	1

(ii)

	1	2	3	4	5	6	7	8
1	0	0	0	0	0	0	0	0
2	3	0	0	0	0	0	0	0
3	10	8	0	0	0	0	0	0
4	0	0	12	0	0	0	0	0
5	0	0	0	15	0	4	0	0
6	0	0	0	10	0	0	9	14
7	0	0	0	0	0	0	0	10
8	17	0	0	0	0	0	0	0

(a)

(iii) Adjacency list



(b)

Fig. Ex. 7.2.1

Example 7.2.2 : An Euler tour of a connected graph G = (V,E) is a cycle that traverses each edge of G exactly once, although it may visit a vertex more than once. For an Euler cycle to exist, the degree of each node must be even. Write an algorithm to determine whether the given graph is Euler when :

(i) Adjacency matrix is used to represent the graph.

(ii) Adjacency list is used to represent the graph.

Solution :

1. Using adjacency matrix representation.

For the graph to be Euler, the degree of each node should be even. The function returns 1 if the graph is Euler, 0 otherwise.

```
int euler_matrix (int G[][MAX], int n)
  {  //n is number of vertices
     int i, j, count;
     for (i=0; i <n; i ++)
     {  count = 0;
        for (j=0; j<n; j++)
           if (G[i][j]==1)
              count =count + 1;
        if (count%2==1)
        return(0);
     }
     return(1);
  }
```

2. Using adjacency list.

```
int euler_list (node *G[], int n)
  {  node *P;
     int count, i;
     for (i=0; i<n; i++)
        {  P=G[i]; //head of the linked list
           count=0;
           while (P! = NULL)
              {  count = count + 1;
                 P = P→ next;
              }
           if (count%2==1)
              return(0);
        }
     return(1);
  }
```

Example 7.2.3 : For the adjacency list given below, draw the corresponding graph.

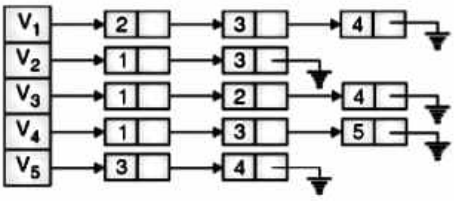


Fig. Ex. 7.2.3

Solution :

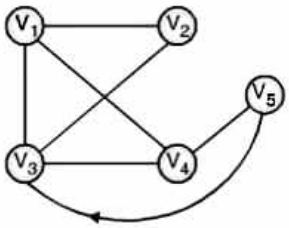


Fig. Ex. 7.2.3(a)

Example 7.2.4 : Find the adjacency matrix for the graph shown below.

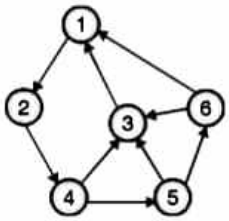


Fig. Ex. 7.2.4

Solution :

Adjacency matrix is given below :

	1	2	3	4	5	6
1	0	1	0	0	0	0
2	0	0	0	1	0	0
3	1	0	0	0	0	0
4	0	0	1	0	1	0
5	0	0	1	0	0	1
6	1	0	1	0	0	0

Fig. Ex. 7.2.4(a)

Example 7.2.5 : For the given graph, give adjacent list, edge list and storage representation for adjacency list. **S-08**

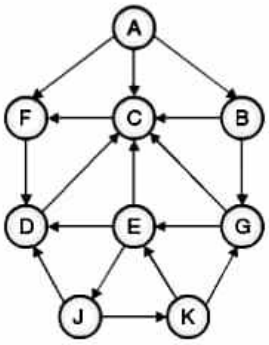


Fig. Ex. 7.2.5

Solution :

Adjacent list

Vertex	Adjacent vertices
A	F, C, B
B	C, G
C	F
D	C
E	C, D, J
F	D
G	C, E
J	D, K
K	E, G

Edge list

(A, F), (A, C), (A, B), (B, C), (B, G), (C, F), (D, C)

(E, C), (E, D), (E, J), (F, D), (G, C), (G, E), (J, D), (J, K), (K, E), (K, G)

Adjacency List representation

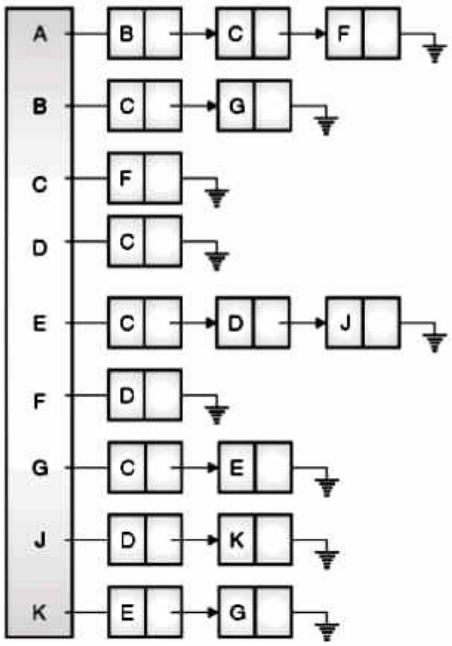


Fig. Ex. 7.2.5(a)

Example 7.2.6 : Consider the following graph. Find it's adjacency matrix and path matrix by using sequential representation of graph. Refer Fig. Ex. 7.2.6.DATA : X, Y, Z, W **W -10**

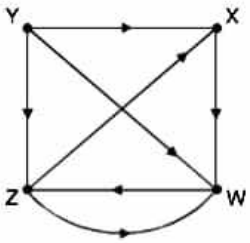


Fig. Ex. 7.2.6

Solution :

1. **Adjacency matrix :**

	w	x	y	z
w	0	0	0	1
x	1	0	0	0
y	1	1	0	1
z	1	1	0	0

2. **Path matrix**

	w	x	y	z
w	1	1	0	1
x	1	1	0	1
y	1	1	0	1
z	1	1	0	1

Example 7.2.7 : Consider the graph given in Fig. Ex. 7.2.7(i)Give adjacency matrix representation.(ii)Give adjacency list representation of the graph. **S-11**

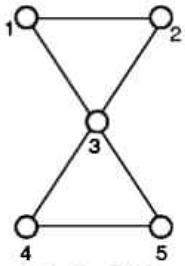


Fig. Ex. 7.2.7

Solution :

(i) **Adjacency matrix**

	1	2	3	4	5
1	0	1	1	0	0
2	1	0	1	0	0
3	1	1	0	1	1
4	0	0	1	0	1
5	0	0	1	1	0

(ii) **Adjacency list**

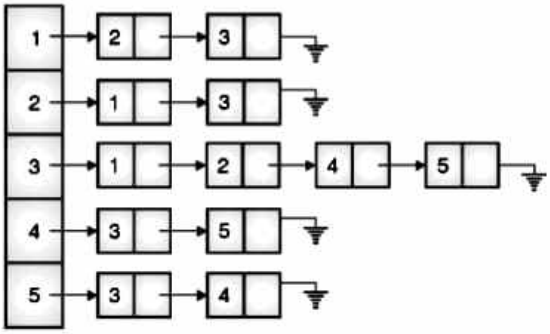


Fig. Ex. 7.2.7(a)

Example 7.2.8 : Consider the graph shown in Fig. Ex. 7.2.8. (i) Give adjacency matrix representation.(ii)Give adjacency list representation of the graph. **W-14**

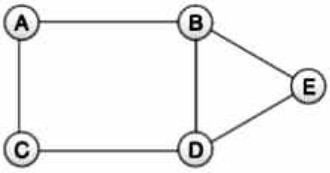


Fig. Ex. 7.2.8

Solution :

(i) Give adjacency matrix representation

	A	B	C	D	E
A	0	1	1	0	0
B	1	0	0	1	1
C	1	0	0	1	0
D	0	1	1	0	1
E	0	1	0	1	0

(ii) Give adjacency list representation of the graph

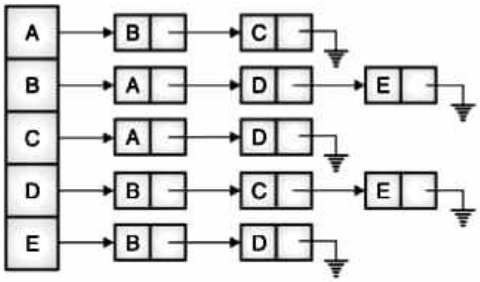


Fig. Ex. 7.2.8(a)

Example 7.2.9 : Consider the graph given below :

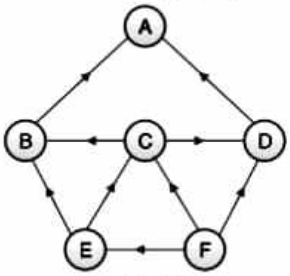


Fig. Ex. 7.2.9

(i) Give adjacency matrix representation.

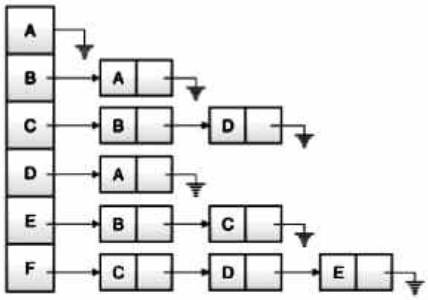
(ii) Give adjacency list representation of graph. **W-15**

Solution :

(i) Adjacency matrix :

	A	B	C	D	E	F
A	0	0	0	0	0	0
B	1	0	0	0	0	0
C	0	1	0	1	0	0
D	1	0	0	0	0	0
E	0	1	1	0	0	0
F	0	0	1	1	1	0

(ii) Adjacency list :



Example 7.2.10 : For the following graph :

(i) Give adjacency matrix representation,

(ii) Give adjacency list representation. **S-16**

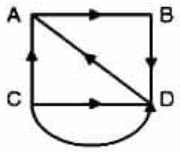


Fig. Ex. 7.2.10

Solution :

(i) Adjacency matrix representation

	A	B	C	D
A	0	1	0	0
B	0	0	0	1
C	1	0	0	1
D	1	0	0	0

(ii) Adjacency list representation

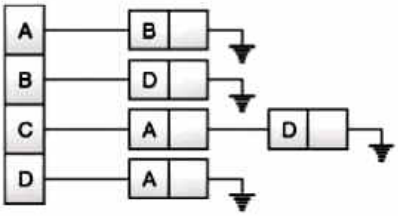


Fig. Ex. 7.2.10(a)

Example 7.2.11 : Consider the graph 'G' in Fig. Ex. 7.2.11.

(i) Find all simple paths from C – A.

(ii) Find all simple paths from D – B.

(iii) Find indeg [B] and outdeg [C].

(iv) Find the adjacency matrix A for graph.

(v) Give adjacency list representation of graph. **W-16**

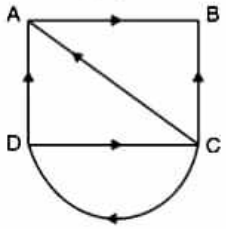


Fig. Ex. 7.2.11

Solution :

i) All simple paths from C – A are

C – A

C – D – A

ii) All simple paths from D – B are

D – A – B

D – C – B

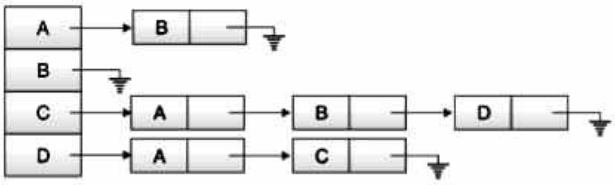
D – C – A – B

iii)

Indegree [B] = 2

Outdegree[C] = 3

iv) **Adjacency list :**



Example 7.2.12 : Consider the graph G in Fig. Ex. 7.2.12.

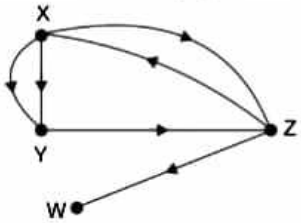


Fig. Ex. 7.2.12

i) Write adjacency matrix representation.

ii) Find all simple path from X to W.

iii) Find indegree (X) and otudegree (W). **S-17**

Solution :

i) **Adjacency matrix**

	X	Y	Z	W
X	0	1	1	0
Y	1	0	1	0
Z	1	0	0	1
W	0	0	0	0

ii) $X \rightarrow Z,\ X \rightarrow Y \rightarrow Z$

iii) indegree (X) = 2

Outdegree (W) = 0

Example 7.2.13 : Consider the graph 'G' in following Fig. Ex. 7.2.13.

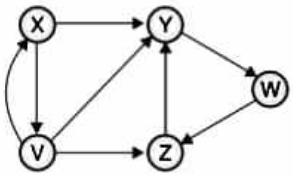


Fig. Ex. 7.2.13

i) Find all simple path from X to Z

ii) Find indegree, and outdegree of nodes Y and Z.

iii) Find adjacency matrix A for the above graph.

iv) Give adjacency list representation of above graph. **W-17**

Solution :

i) $X \rightarrow V \rightarrow Z$, $X \rightarrow V \rightarrow Y \rightarrow W \rightarrow Z$

$X \rightarrow Y \rightarrow W \rightarrow Z$

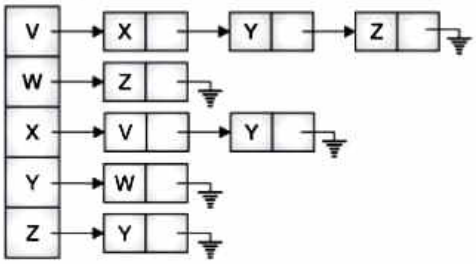
ii)

	Indegree	Outdegree
Y	3	1
Z	2	1

iii) **Adjacency matrix**

	V	W	X	Y	Z
V	0	0	1	1	1
W	0	0	0	0	1
X	1	0	0	1	0
Y	0	1	0	0	0
Z	0	0	0	1	0

(iv) **Adjacency list**



7.2.4 Programs on Graph Representation

➢ **Program 7.2.1 : Program to compute the in degree and out degree of every vertex of a directed graph when the graph is represent by an adjacency matrix.**

```
#include<conio.h>
#include<stdio.h>
void main()
{
    int G[10][10],n,i,j,i_degree,o_degree;
    printf("\n Enter no of nodes : ");
    scanf("%d",&n);
    // read the adjacency matrix
    printf("\n Enter adjacency matrix of the graph :");
    for(i=0;i<n;i++)
        for(j=0;j<n;j++)
        {
            scanf("%d", &G[i][j]);
        }
    // indegree of node i=no. of 1's in ith column
    // outdegree if node i=no. of 1's in ith row
    for(i=0;i<n;i++)
    {
        i_degree=0;
        for(j=0;j<n;j++)
            if(G[j][i]!=0)
                i_degree++;
        o_degree=0;
        for(j=0;j<n;j++)
            if(G[i][j]!=0)
                o_degree++;
        printf("\n node number
%d\tindegree=%d\toutdegree=%d",i,i_degree,o_degree);
    }
}
```

Output

```
Enter no of nodes : 6
Enter adjacency matrix of the graph :0 0 1 0 1 0
0 0 1 1 0 0
0 0 0 0 0 0
0 0 0 0 0 1
0 1 0 1 0 0
1 0 0 0 0 0
node number 0  indegree=1    outdegree=2
node number 1  indegree=1    outdegree=2
node number 2  indegree=2    outdegree=0
```

```
node number 3  indegree=2    outdegree=1
node number 4  indegree=1    outdegree=2
node number 5  indegree=1    outdegree=1
```

- **Program 7.2.2 : Program to compute the indegree and outdegree of every vertex of a directed graph when the graph is represented by an adjacency list.**

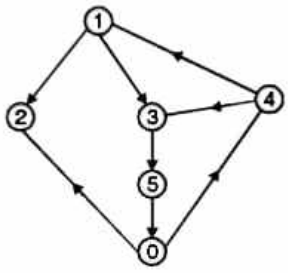
```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
typedef struct node
{
    struct node *next;
    int vertex;
}node;
node * G[20];
/* An array of pointers G[i] is the head of the list of adjacency
  vertices of node i represented using a linked list */
int n;
void read_graph();
int in_degree(int v);
int out_degree(int v);
void insert(int vi,int vj);
void main()
{
    int i_degree,o_degree,i;
    read graph();
    for(i=0;i<n;i++)
    {
        i_degree=in_degree(i);
        o_degree=out_degree(i);
        printf("\nNode
No=%d\tindegree=%d\toutdegree=%d",i,i_degree,o_degree);
    }
}
void read_graph()
{
    int i,vi,vj,no_of_edges;
    printf("\nEnter no of vertices : ");
    scanf("%d",&n);
    // initialise G[] with a null
    for(i=0;i<n;i++)
    {
        G[i]=NULL;
        // read edges and insert them in G[]
        printf("\nEnter no of edges : ");
        scanf("%d",&no_of_edges);
        for(i=0;i<no_of_edges;i++)
        {
            printf("\nEnter an edge (u,v) :");
            scanf("%d%d",&vi,&vj);
            insert(vi,vj);
        }
    }
}
void insert(int vi,int vj)
{
    node *p,*q;
    // acquire memory for the new node
    q=(node *)malloc(sizeof(node));
    q→ vertex=vj;
    q→ next=NULL;
    //insert the node in the linked list number vi
    if(G[vi]==NULL)
        G[vi]=q;
    else
    {
        // go to end of linked list
        p=G[vi];
```

```
        while(p→ next!=NULL)
            p=p→ next;
        p→ next=q;
    }
}
int out_degree(int v)
{
    node *p;
    int o_degree=0;
    p=G[v];
    //count number of nodes in the linked list p
    while(p!=NULL)
    {
        o_degree++;
        p=p→ next;
    }
    return(o_degree);
}
int in_degree(int v)
{
    node *p;
    int in_degree,i;
    //all linked list must be searched for the vertex
    in_degree=0;
    for(i=0;i<n;i++)
    {
        p=G[i];
        while(p!=NULL)
        {
            if(p→ vertex==v)
                in_degree++;
            p=p→ next;
        }
    }
    return(in_degree);
}
```

Output

```
Enter no of vertices : 6
Enter no of edges : 8
Enter an edge (u,v) :1 2
Enter an edge (u,v) :1 3
Enter an edge (u,v) :4 1
Enter an edge (u,v) :0 2
Enter an edge (u,v) :5 0
Enter an edge (u,v) :0 4
Enter an edge (u,v) :4 3
Enter an edge (u,v) :3 5
Node No=0    indegree=1    outdegree=2
Node No=1    indegree=1    outdegree=2
Node No=2    indegree=2    outdegree=0
Node No=3    indegree=2    outdegree=1
Node No=4    indegree=1    outdegree=2
Node No=5    indegree=1    outdegree=1
```

Function readgraph(), creates an adjacency list for the given graph. Edges are entered as a pair of vertices (V_i, V_j). Edge (V_i, V_j) is entered in the ith linked list and the "vertex" field of the node is set to V_j working of readgraph() is shown below :



Input graph

(0, 4), (0, 2)

(1, 2), (1, 3)

(3, 5)

(4, 1), (4, 3)

(5, 0)

Fig. 7.2.7(a)

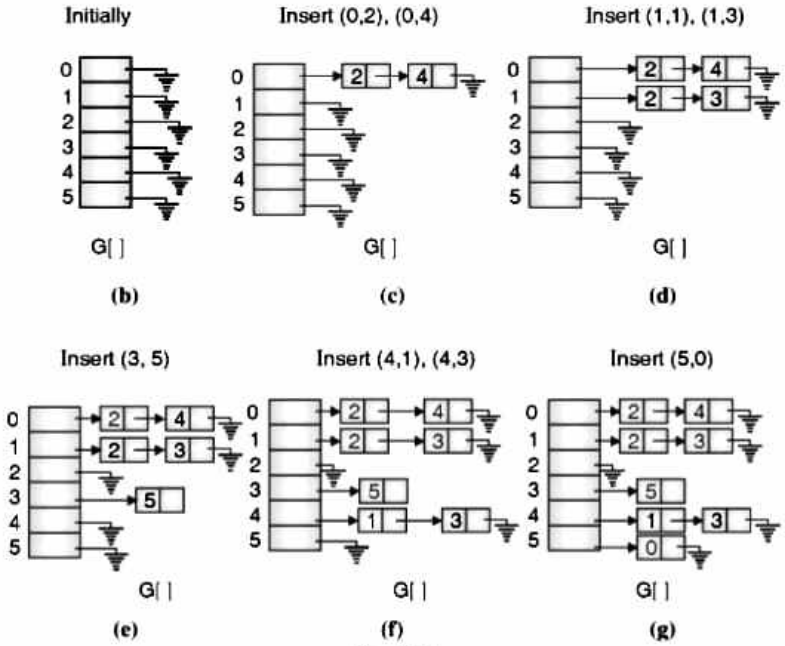


Fig. 7.2.7

7.3 MSBTE Questions and Answers

➔ Summer 2008 – Total Marks 12

Q. 1 For the given graph in Fig. Q. 1 give adjacent list, edge list and storage representation for adjacency list. **(Example 7.2.5)** **(8 Marks)**

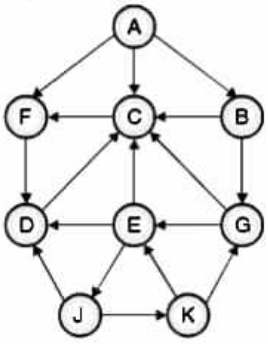


Fig. Q. 1

Q. 2 Describe in brief, the terms related to graph : nodes, edges, in-degree, out – degree.
(Sections 7.1.1 and 7.1.12) **(4 Marks)**

➔ Winter 2008 – Total Marks 04

Q. 3 Explain the link representation of a Graph with suitable example. **(Section 7.2.2)** **(4 Marks)**

➔ Winter 2009 – Total Marks 04

Q. 4 Explain in-degree and out-degree of a node with example.
(Section 7.1.12 and Example 7.2.1) **(4 Marks)**

➔ Summer 2010 – Total Marks 04

Q. 5 Define following terms with respect to graph :

(i) In-degree of a node **(Section 7.1.12)**

(ii) Directed graph **(Section 7.1.3)**

(iii) Weighted graph **(Section 7.1.5)**

(iv) Predecessor **(Section 7.1.3)** **(4 Marks)**

➔ Winter 2010 – Total Marks 04

Q. 6 Consider the following graph. Find it's adjacency matrix and path matrix by using sequential representation of graph. Refer Fig. Q. 6.

DATA : X, Y, Z, W

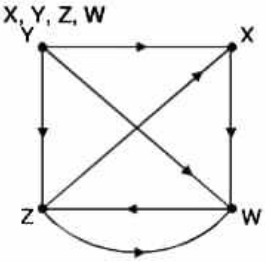


Fig. Q. 6

(Example 7.2.6) **(4 Marks)**

➔ Summer 2011 – Total Marks 16

Q. 7 Define Graph? Enlist and explain two types of graph with examples.

(Sections 7.1.1, 7.1.2 and 7.1.3) **(8 Marks)**

Q. 8 Consider the graph given in Fig. Q.8

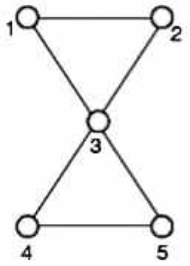


Fig. Q. 8

(i) Give adjacency matrix representation.

(ii) Give adjacency list representation of the graph.

(Example 7.2.7) **(8 Marks)**

➔ Winter 2011 – Total Marks 20

Q. 9 Describe application of graph in data structures. **(10 Marks)**

Ans. :

Some of the applications of the graph include:

1. Representation of many-many relation. **(Section 7.2.1)**
2. Finding shortest distance.
3. Routing using spanning tree **(Sections 7.1.17 and 7.1.18)**

Q. 10 Explain Sequential and Linked Representation of Graphs.**(Sections 7.2.1 and 7.2.2)** **(10 Marks)**

➔ Summer 2012 – Total Marks 10

Q. 11 List applications of graph. **(2 Marks)**

Ans. :

Some of the applications of the graph include :

1. Representation of many-many relation **(Section 7.2.1)**
2. Finding shortest distance
3. Routing using spanning tree. **(Sections 7.1.17 and 7.1.18)**

Q. 12 What is graph ? Explain adjacency matrix representation of graph with suitable example. **(Sections 7.1.1 and 7.2.1)** **(4 Marks)**

Q. 13 Explain following terminologies with respect to graph using suitable example. **(4 Marks)**

(i) Directed graph **(Section 7.1.3)**

(ii) in-degree **(Section 7.1.12)**

(iii) Out-degree **(Section 7.1.12)**

(iv) Weighted graph **(Section 7.1.5)**

➔ Winter 2012 – Total Marks 04

Q. 14 Compare the sequential representation method of graphs with linked representation method of graph.

(Sections 7.2.1 and 7.2.2) **(4 Marks)**

➔ Summer 2013 – Total Marks 08

Q. 15 Explain the link representation of a graph using suitable example. **(Section 7.2.2)** **(4 Marks)**

Q. 16 Explain in degree and out degree of a node with example.

(Section 7.1.12) **(4 Marks)**

➔ Winter 2013 – Total Marks 04

Q. 17 Describe directed graph and undirected graph with diagram.

(Sections 7.1.2 and 7.1.3) **(4 Marks)**

➔ Summer 2014 – Total Marks 06

Q. 18 Explain indegree and outdegree of a graph with example.

(Section 7.1.12 and Example 7.2.1) **(4 Marks)**

Q. 19 Explain the following terms w.r.t. tree with diagram :

(vi) In degree **(Section 7.1.12)**

(vii) Out degree. **(Section 7.1.12)** **(2 Marks)**

➔ Winter 2014 – Total Marks 14

Q. 20 Define directed edge of a tree.

(Section 7.1.3) **(2 Marks)**

Q. 21 Explain the following term with respect to graph using suitable example.

(i) In-degree (ii) Out-degree

(Section 7.1.12) **(4 Marks)**

Q. 22 Consider the graph shown in Fig. Q. 22.

(i) Give adjacency matrix representation.

(ii) Give adjacency list representation of the graph. **(Example 7.2.8)** **(8 Marks)**

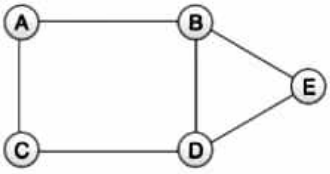


Fig. Q. 22

➔ Winter 2015 – Total Marks 13

Q. 1(A)(e) Define : In-degree.

(Ans. : Refer section 7.1.12) **(1 Mark)**

Q. 3(f) Define the following terminologies with respect to graph :

(i) Direct graph (ii) Successor

(iii) Predecessor (iv) Path

(Ans. : Refer sections 7.1.3 and 7.1.7)

(4 Marks)

Q. 6(c) Consider the graph given below :

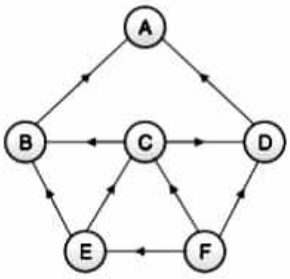


Fig. 1-Q. 6(c)

(i) Give adjacency matrix representation.

(ii) Give adjacency list representation of graph. *(Ans. : Refer Example 7.2.9)*

(8 Marks)

➔ Summer 2016 – Total Marks 12

Q. 4(f) Define the following terms in graph :

i) Source, ii) Isolated node, iii) Sink,

iv) Articulation point.

(Ans. : Refer sections 7.1.12, 7.1.20 and 7.1.21) **(4 Marks)**

Q. 6(c) For the following graph :

(i) Give adjacency matrix representation,

(ii) Give adjacency list representation.

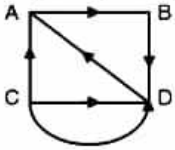


Fig. 1-Q. 6(c)

(Ans. : Refer Example 7.2.10) **(8 Marks)**

➔ Winter 2016 – Total Marks 12

Q. 1(A)(6) Define following W.R.T. tree

(a) In-degree (b) Out-degree

(Ans. : Refer section 7.1.12) **(2 Marks)**

Q. 1(A)(8) List any four application of graph.

(Ans. : Refer Q.11 Summer 2012) **(2 Marks)**

Q. 6 (c) Consider the graph 'G' in Fig. 1-Q. 6(c).

(i) Find all simple paths from C – A.

(ii) Find all simple paths from D – B.

(iii) Find indeg [B] and outdeg [C].

(iv) Find the adjacency matrix A for graph.

(v) Give adjacency list representation of graph. *(Ans. : Refer Example 7.2.11)*

(8 Marks)

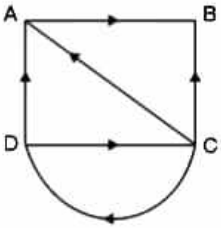


Fig. 1-Q. 6(c)

➔ Summer 2017 – Total Marks 14

Q. 1(A)(h) What is indegree and outdegree of a node in graph ?

(Ans. : Refer section 7.1.12) **(2 Marks)**

Q. 2(f) Explain representation of graph in detail.

(Ans. : Refer section 7.2) **(4 Marks)**

Q. 6(c) Consider the graph G in Fig. 1-Q. 6(c)

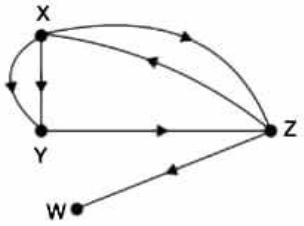


Fig. 1-Q. 6(c)

i) Write adjacency matrix representation.

ii) Find all simple path from X to W.

iii) Find indegree (X) and otudegree (W).

(Ans. : Refer Example 7.2.12) **(8 Marks)**

➔ Winter 2017 – Total Marks 08

Q. 6(c) Consider the graph 'G' in following Fig. 1-Q. 6(c)

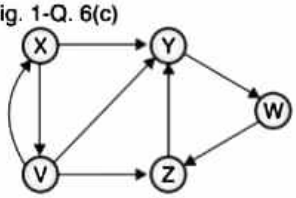


Fig. 1-Q. 6(c)

(i) Find all simple path from X to Z

(ii) Find indegree, and outdegree of nodes Y and Z.

(iii) Find adjacency matrix A for the above graph.

(iv) Give adjacency list representation of above graph.

(Ans. : Refer Example 7.2.13) **(8 Marks)**

❑❑❑

Note

Lab Manual

List of Practicals

L.1 Program Based on Array

- **Program 1 : Implement a 'C' program for performing following operations on Array : Creation, Insertion, Deletion, display.**

```
/*Program based on array: operation for insertion and deletion
*/

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

void main()
{
  int a[30],n=0,op,i,x,loc;
  clrscr();
  do
    {
      printf("\n\n1)Read initial Data in the array");
      printf("\n2)Insert\n3)Delete\n4)Quit");
      printf("\nEnter your choice : ");
      scanf("%d",&op);
      switch(op)
        {
         case 1: printf("\nEnter no. of data : ");
              scanf("%d",&n);
              printf("\nEnter %d elements : ",n);
              for(i=0;i<n;i++)
                  scanf("%d",&a[i]);
              break;
case 2: printf("\nEnter a location from %d to %d :
          ",1,n+1);
        scanf("%d",&loc);
        if(loc>=1 & loc<= n+1)
          {
            printf("\nEnter the data to be inserted : ");
            scanf("%d",&x);
            for(i=n-1;i>=loc-1;i--)
                a[i+1]=a[i];
                a[loc-1]=x;
            n++;
            printf("\nData after insertion :");
            for(i=0;i<n;i++)
                printf("%d ",a[i]);
          }
        else
            printf("\n Invalid location ");
        break;
case 3: printf("\nEnter a location from %d to
          %d : ",1,n);
        scanf("%d",&loc);
        if(loc>=1 & loc<= n)
          {
            for(i=loc;i<n;i++)
                a[i-1]=a[i];
            n--;
            printf("\nData after deletion :");
            for(i=0;i<n;i++)
```

```
                printf("%d ",a[i]);
            }
            else
              printf("\n Invalid location ");
            break;
        }
    }while(op!=4);
}
```

Output

```
1)Read initial Data in the array
2)Insert
3)Delete
4)Quit
Enter Your Choice : 2
Enter a location from 1 to 1 : 1
Enter the data to be inserted : 12

Data after insertion :12
1)Read initial Data in the array
2)Insert
3)Delete
4)Quit
Enter Your Choice : 2
Enter a location from 1 to 2 : 2
Enter the data to be inserted : 34

Data after insertion :12 34
1)Read initial Data in the array
2)Insert
3)Delete
4)Quit
Enter Your Choice :2
Enter a location from 1 to 3 : 3
Enter the data to be inserted :6

Data after insertion :12   34  6
1)Read initial Data in the array
2)Insert
3)Delete
4)Quit
Enter Your Choice : 3
Enter a location from 1 to 3 : 2

Data after deletion :12    6
1)Read initial Data in the array
2)Insert
3)Delete
4)Quit
Enter Your Choice :4
```

L.2 Various Searching Methods

- **Program 2 : Implement a 'C' program to search a particular data from the given Array using : (i) Linear Search, (ii) Binary Search.**

```
/*Program for various searching methods */

#include <stdio.h>
#include <conio.h>
int binsearch(int a[],int i ,int j,int key);   /*Recursive binary
                                                search*/
int linsearch(int a[],int n , int key);
void main()
 {
  int a[30],key,n,i,result,op;
  clrscr();
  do
    {
     printf("\n1)Linear Search\n2)Binary Search\n3)Quit");
     printf("\nEnter Your Choice : ");
```

```
    scanf("%d",&op);
    switch(op)
      {
       case 1: printf("\n Enter No. of elements : ");
           scanf("%d" ,&n);
           printf("\n Enter a  list of %d elements : ",n);
           for(i=0;i<n;i++)
              scanf("%d",&a[i]);
           printf("\n Enter the the element to be searched : ");
           scanf("%d",&key);
           result=linsearch(a,n,key);
           if(result==-1)
               printf("\n Not found : ");
           else
               printf("\n Found at location= %d",result+1);
           break;
       case 2: printf("\n Enter No. of elements : ");
           scanf("%d" ,&n);
           printf("\n Enter a sorted list of %d elements : ",n);
           for(i=0;i<n;i++)
              scanf("%d",&a[i]);
           printf("\n Enter the the element to be searched : ");
           scanf("%d",&key);
           result=binsearch(a,0,n-1,key);
           if(result==-1)
               printf("\n Not found : ");
           else
               printf("\n Found at location= %d",result+1);
           break;
      }
    }while(op!=3);
}
int binsearch(int a[],int i, int j,int key)
{
  int c;
  if(i>j)
   return(-1);
  c=(i+j)/2;
  if(key==a[c])
     return(c);
  if(key>a[c])
     return(binsearch(a,c+1,j,key));
  return(binsearch(a,i,c-1,key));
 }
int linsearch(int a[],int n , int key)
{
  int i;
  for(i=0;i<n;i++)
    {
   if(a[i]==key)
       return(i);
    }
  return(-1);
}
```

Output

```
1)Linear Search
2)Binary Search
3)Quit

Enter Your Choice : 1
Enter No. of elements : 3

Enter a  list of 3 elements : 1 45 89
Enter the the element to be searched : 0

Not found :
1)Linear Search
2)Binary Search
```

```
3)Quit
Enter Your Choice : 1
Enter No. of elements : 3
Enter a  list of 3 elements : 34 78 6
Enter the the element to be searched : 6
Found at location= 3

1)Linear Search
2)Binary Search
3)Quit
Enter Your Choice : 2
Enter No. of elements : 4

Enter a sorted list of 4 elements : 23 60 89 90
 Enter the the element to be searched : 90

Found at location= 4

1)Linear Search
2)Binary Search
3)Quit
Enter Your Choice : 3
```

L.3 Sorting Methods

- **Program 3 : Implement a 'C' program to sort an array using following methods : (i) Bubble Sort, (ii) Selection Sort (iii) Insertion Sort.**

```
/*Implement sorting methods - bubble sort, selection sort
insertion sort.*/

void insertion_sort(int [],int);
void selection_sort(int a[],int n);
void bubble_sort(int a[],int n);
void main()
{
    int a[50],n,i,op;
    clrscr();
    do
     {
      printf("\n1)Insertion\n2)Selection\n3)Bubble\n4)Quit");
      printf("\nEnter your choice : ");
      scanf("%d",&op);
      switch(op)
       {
        case 1: printf("\nEnter no. of elements :");
            scanf("%d",&n);
            printf("\nEnter array elements :");
            for(i=0;i<n;i++)
               scanf("%d",&a[i]);
            insertion_sort(a,n);
            break;
        case 2: printf("\nEnter no. of elements :");
            scanf("%d",&n);
            printf("\nEnter array elements :");
            for(i=0;i<n;i++)
               scanf("%d",&a[i]);
            selection_sort(a,n);
            break;
        case 3: printf("\nEnter no of elements :");
            scanf("%d",&n);
            printf("\nEnter array elements :");
            for(i=0;i<n;i++)
               scanf("%d",&a[i]);
            bubble_sort(a,n);
            break;
       }
    } while(op!=4);
}
void insertion_sort(int a[],int n)
{
```

```
    int i,j,temp,k;
    printf("\n Unsorted Data:");
    for(k=0;k<n;k++)
      printf("%5d",a[k]);
    for(i=1;i<n;i++)
    {
        temp=a[i];
        for(j=i-1;j>=0 && a[j]>temp; j--)
            a[j+1]=a[j];
            a[j+1]=temp;
      printf("\n After pass   %d", i);
        for(k=0;k<n;k++)
         printf("%5d",a[k]);
    }
}
void selection_sort(int a[],int n)
{
    int i,j,temp,k;
    printf("\n Unsorted Data:");
    for(k=0;k<n;k++)
      printf("%5d",a[k]);
    for(i=0;i<n-1;i++)
    {      k=i;
        for(j=i+1;j<n;j++)
          if(a[j]<a[k])
            k=j;
         temp=a[i];
         a[i]=a[k];
         a[k]=temp;
      printf("\n After pass   %d",i+1);
        for(k=0;k<n;k++)
         printf("%5d",a[k]);
    }
}
```

```
void bubble_sort(int a[],int n)
{
    int i,j,k,temp;
    printf("\n Unsorted Data:");
    for(k=0;k<n;k++)
      printf("%5d",a[k]);
    for(i=1;i<n;i++)
     {
      for(j=0;j<n-1;j++)
        if(a[j]>a[j+1])
          {
            temp=a[j];
            a[j]=a[j+1];
            a[j+1]=temp;
          }
        printf("\nAfter pass   %d",i);
        for(k=0;k<n;k++)
         printf("%5d",a[k]);
      }
}
```

Output

```
1)Insertion
2)Selection
3)Bubble
4)Quit
Enter your choice : 1
Enter no of elements :4
Enter array elements :5
7
33
2
Unsorted Data:  5   7  33   2
After pass 1=   5   7  33   2
After pass 2=   5   7  33   2
```

```
After pass 3=   2   5   7  33

1)Insertion
2)Selection
3)Bubble
4)Quit
Enter your choice : 2
Enter no of elements :4
Enter array elements :55   2  1
88
Unsorted Data:  55   2   1  88
After pass  1   1   2  55  88
After pass  2   1   2  55  88
After pass  3   1   2  55  88

1)Insertion
2)Selection
3)Bubble
4)Quit
Enter your choice : 3
Enter no of elements :4
Enter array elements :2
6
33
11
Unsorted Data:   2   6  33  11
After pass  1   2   6  11  33
After pass  2   2   6  11  33
After pass  3   2   6  11  33
1)Insertion
2)Selection
3)Bubble
4)Quit
Enter your choice : 4
```

L.4 Operations on Stack

➢ **Program 4 : Write C program to perform PUSH and POP operations on stack using array.**

Q. Explain the concept of PUSH, POP and top of the stack with example. **S-14**

```
/* Simulation of stack using  an array */

#include<stdio.h>
#include<conio.h>
#define MAX 6

typedef struct stack
{
    int data[MAX];
    int top;
}stack;
void init(stack *);
int empty(stack *);
int full(stack *);
int pop(stack *);
void push(stack *,int);
void print(stack *);
void main()
{
    stack s;
    int x,op;
    init(&s);
    clrscr();
    do {
        printf("\n\n1)Push\n2)Pop\n3)Print\n4)Quit");
        printf("\nEnter Your choice: ");
        scanf("%d",&op);
        switch(op)
         {
```

```
        case 1:printf("\n enter a number :");
            scanf("%d",&x);
            if(!full(&s))
             push(&s,x);
            else
             printf("\nStack is full.....");
            break;
        case 2:if(!empty(&s))
            { x=pop(&s);
             printf("\npopped value= %d",x);
            }
            else
           printf("\nStack is empty.....");
            break;
        case 3:print(&s);break;
        }
    }while(op!=4);
}
void init(stack *s)
{
    s→ top=-1;
}
int empty(stack *s)
{
    if(s→ top==-1)
        return(1);
    return(0);
}
int full(stack *s)
{
    if(s→ top==MAX-1)
        return(1);
    return(0);
}
void push(stack *s,int x)
{
    s→ top=s→ top+1;
    s→ data[s→ top]=x;
}
int pop(stack *s)
{
    int x;
    x=s→ data[s→ top];
    s→ top=s→ top-1;
    return(x);
}
void print(stack *s)
 {
    int i;
    printf("\n");
    for(i=s→ top;i>=0;i--)
        printf("%d  ",s→ data[i]);
 }
```

Output

```
enter a number :9

1)Push
2)Pop
3)Print
4)Quit
Enter Your choice: 1
enter a number :15

1)Push
2)Pop
3)Print
4)Quit
Enter Your choice: 3
```

```
15 9

1)Push
2)Pop
3)Print
4)Quit
Enter Your choice: 2

popped value= 15
1)Push
2)Pop
3)Print
4)Quit
Enter Your choice: 2
popped value= 9

1)Push
2)Pop
3)Print
4)Quit
Enter Your choice: 2
Stack is empty.....

1)Push
2)Pop
3)Print
4)Quit
Enter Your choice: 4
```

L.5 Implementation of Linear Queue

- **Program 5 : Write C program to perform INSERT and DELETE operations on Linear Queue using array.**

```
/* Simulation of queue using an array */
#include<conio.h>
#include<stdio.h>
#define MAX 5

typedef struct Q
{
    int R,F;
    int data[MAX];
}Q;
void initialise(Q *P);
int empty(Q *P);
int full(Q *P);
void enqueue(Q *P,int x);
int dequeue(Q *P);
void print(Q *P);
void main()
{
    Q q;
    int op,x;
    initialise(&q);
    clrscr();
    do
      {
        printf("\n\n1)Insert\n2)Delete\n3)Print\n4)Quit");
        printf("\nEnter Your Choice:");
        scanf("%d",&op);
        switch(op)
          {
            case 1: printf("\n Enter a value:");
                scanf("%d",&x);
                if(!full(&q))
                    enqueue(&q,x);
                else
                    printf("\nQueue is full !!!!");
                break;
            case 2: if(!empty(&q))
```

```
                {
                  x=dequeue(&q);
                  printf("\Deleted Data=%d",x);
                }
              else
              printf("\nQueue is empty !!!!");
              break;
          case 3: print(&q);break;
        }
    }while(op!=4);
}
void initialise(Q *P)
{
    P→ R=-1;
    P→ F=-1;
}
int empty(Q *P)
{
    if(P→ R==-1)
        return(1);
    return(0);
}
int full(Q *P)
{
    if(P→ R==MAX-1)
        return(1);
    return(0);
}
void enqueue(Q *P,int x)
{
    if(P→ R==-1)
    {
        P→ R=P→ F=0;
        P→ data[P→ R]=x;
    }
    else
    {
        P→ R=P→ R+1%MAX;
        P→ data[P→ R]=x;
    }
}
int dequeue(Q *P)
{
    int x;
    x=P→ data[P→ F];
    if(P→ R==P→ F)
    {
        P→ R=-1;
        P→F=-1;
    }
    else
        P→ F=P→ F+1;
    return(x);
}
void print(Q *P)
{
    int i;
    if(!empty(P))
    {
        printf("\n");
        for(i=P→ F;i<=P→ R;i++)
            printf("%d\t",P→ data[i]);
    }
}
```

Output

```
1)Insert
2)Delete
3)Print
4)Quit
Enter Your Choice:1
Enter a value:3

1)Insert
2)Delete
3)Print
4)Quit
Enter Your Choice:1
Enter a value:6

1)Insert
2)Delete
3)Print
4)Quit
Enter Your Choice:1
Enter a value:8
```

```
1)Insert
2)Delete
3)Print
4)Quit
Enter Your Choice:3
3     6     8

1)Insert
2)Delete
3)Print
4)Quit
Enter Your Choice:2
Deleted Data=3

1)Insert
2)Delete
3)Print
4)Quit
Enter Your Choice:4
```

L.6 Implementation of Circular Queue

➢ **Program 6 : Write C program to perform INSERT and DELETE operations on Circular Queue using array.**

Q. Write a program in 'C' language to implement 'circular queue'. **S-13**

```
#include <stdio.h>
#include <conio.h>
#define MAX 30
typedef struct queue
{
    int data[MAX];
    int rear, front;
} queue;
void initialize (queue *p);
int empty (queue *p);
int full (queue *p);
void enqueue (queue *p, int x);
int dequeue (queue *p);
void print (queue *p);

void main ( )
{
    int x, op, n;
    queue q;
    initialize (&q);
    do
    {
        printf ("\n1)create \n2)insert \n3)delete \n4)print
          \n5)Quit");
        printf("\n enter your choice :");
        scanf ("%d", &op);
        switch(op)
        { case 1 : printf("\n enter no. of elements :");
                scanf("%d", &n);
                initialize (&q);
                printf("\n enter the data :");
                for(i =0; i < n; i++)
                {
                    scanf("%d", &x);
                    if (full(&q))
                    {
                        printf("\n queue is full ...");
                        exit(0);
                    }
                    enqueue (&q, x);
                }
                break;
        case 2 : printf("\n enter the element to be inserted :");
                scanf("%d", &x);
                if (full(&q))
                {
                    printf("\n queue is full ...");
```

```
                exit(0);
            }
            enqueue (&q, x);
            break;
        case 3 : if (empty(&q))
            {
                printf("\n queue is empty ...");
                exit (0);
            }
            x = dequeue (&q);
            printf("\n element =%d", x);
            break;
        case 4 : print (&q);
            break;
        default : break;
        }
    }while (op! = 5);
}
void initialize(queue *P)
{
    P → rear = – 1;
    P → front = –1;
}
int empty(queue *P)
{
    if(P → rear == – 1)
        return(1);
    return(0);
}
int full(queue *P)
{
    if((P → rear + 1)% MAX == P → front)
        return(1);
    return(0);
}
void enqueue(queue *P, int x)
{
    if(empty(P))                    /* empty queue */
    {
        P → rear = P → front==0; /* rear and front will point
                                    to the same element */
            P → data[P → rear] = x;
    }
    else
    {
        P → rear = (P → rear + 1)% MAX;
                        /* Advance P in the circular array */
        P → data[P → rear] = x;
    }
}
int dequeue(queue *P)
{
        int x;
        x = P → data[P → front];
        if(P → rear == P → front)
                        /* deleted the last element */
            initialize(P);
        else
            P → front =(P → front + 1)% MAX;
        /* front advances to the next position in the circular
            array */
        return(x);
}
void print(queue *P)
{
    int i;
    i = P → front;
    while(i != P → rear)
```

```
    {
        printf("\n%d", P → data[i]);
        i = (i + 1) %MAX;
    }
    printf("\n%d", P → data[P → rear]);
}
```

L.7 Operations on Singly Linked List

- **Program 7 : Write C program to perform the operations (Insert, Delete, Traverse, and Search) on Singly Linked List.**

```
/*Operations on SLL(singly linked list) */

#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
typedef struct node
  { int data;
    struct node *next;
  }node;

node *create();
node *insert_b(node *head,int x);
node *insert_e(node *head,int x);
node *insert_in(node *head,int x);
node *delete_b(node *head);
node *delete_e(node *head);
node *delete_in(node *head);
node *reverse(node *head);
void search(node *head);
void print(node *head);
void main()
{ int op,op1,x;
 node *head=NULL;
 clrscr();
 do
  {
   printf("\n\n1)create\n2)Insert\n3)Delete\n4)Search");
   printf("\n5)Reverse\n6)Print\n7)Quit");
   printf("\nEnter your Choice:");
   scanf("%d",&op);
   switch(op)
   { case 1:head=create();break;
   case 2:printf("\n\t1)Beginning\n\t2)End\n\t3)In between");
     printf("\nEnter your choice : ");
     scanf("%d",&op1);
     printf("\nEnter the data to be inserted : ");
     scanf("%d",&x);
     switch(op1)
      { case 1: head=insert_b(head,x);
          break;
        case 2: head=insert_e(head,x);
          break;
        case 3: head=insert_in(head, x);
          break;
      }
     break;
   case 3:printf("\n\t1)Beginning\n\t2)End\n\t3)In between");
     printf("\nEnter your choice : ");
     scanf("%d",&op1);
     switch(op1)
      { case 1:head=delete_b(head);
          break;
        case 2:head=delete_e(head);
          break;
        case 3:head=delete_in(head);
          break;
      }
      break;
   case 4:search(head);break;
   case 5:head=reverse(head);
     print(head);
     break;
   case 6:print(head);break;
   }
 }while(op!=7);
```

```
}
node *create()
{ node *head,*p;
 int i,n;
 head=NULL;
 printf("\n Enter no of data:");
 scanf("%d",&n);
 printf("\nEnter the data:");
 for(i=0;i<n;i++)
  {
   if(head==NULL)
    p=head=(node*)malloc(sizeof(node));
   else
    {
    p->next=(node*)malloc(sizeof(node));
    p=p->next;
    }
    p->next=NULL;
    scanf("%d",&(p->data));
  }
 return(head);
}
node *insert_b(node *head,int x)
{  node *p;
   p=(node*)malloc(sizeof(node));
   p→ data=x;
   p→ next=head;
   head=p;
   return(head);
}
node *insert_e(node *head,int x)
{  node *p,*q;
   p=(node*)malloc(sizeof(node));
   p→ data=x;
   p→ next=NULL;
   if(head==NULL)
     return(p);
   //locate the last node
   for(q=head;q→ next!=NULL;q=q→ next)
   ;
   q→ next=p;
   return(head);
}
node *insert_in(node *head,int x)
{  node *p,*q;
   int y;
   p=(node*)malloc(sizeof(node));
   p→ data=x;
   p→ next=NULL;
   printf("\Insert after which number ? : ");
   scanf("%d",&y);
   //locate the lthe data 'y'
   for(q=head ; q != NULL && q→ data != y ;
    q=q→ next)
   ;
   if(q!=NULL)
    {
    p→ next=q→ next;
    q→ next=p;
    }
   else
     printf("\nData not found ");
   return(head);
}
node *delete_b(node *head)
{
 node *p,*q;
 if(head==NULL)
```

```
    {
     printf("\n Underflow....Empty Linked List");
     return(head);
    }
  p=head;
  head=head→ next;
  free(p);
  return(head);
}
node *delete_e(node *head)
{
  node *p,*q;
  if(head==NULL)
    {
     printf("\nUnderflow....Empty Linked List");
     return(head);
    }
  p=head;
  if(head→ next==NULL)
    { // Delete the only element
      head=NULL;
      free(p);
      return(head);
    }
//Locate the last but one node
  for(q=head;q→ next→ next !=NULL;q=q→ next)
  ;
  p=q→ next;
  q→ next=NULL;
  free(p);
  return(head);
}
node *delete_in(node *head)
{
  node *p,*q;
  int x,i;
  if(head==NULL)
    {
     printf("\nUnderflow....Empty Linked List");
     return(head);
    }
  printf("\nEnter the data to be deleted : ");
  scanf("%d",&x);
  if(head→ data==x)
    { // Delete the first element
      p=head;
      head=head→ next;
      free(p);
      return(head);
    }
//Locate the node previous to one to be deleted
  for(q=head;q→ next→ data!=x && q→ next
!=NULL;q=q→ next)
  ;
  if(q→ next==NULL)
    {
      printf("\nUnderflow.....data not found");
      return(head);
    }
  p=q→ next;
  q→ next=q→ next→ next;
  free(p);
  return(head);
}
void search(node *head)
{ node *p;
  int data,loc=1;
  printf("\nEnter the data to be searched: ");
  scanf("%d",&data);
  p=head;
```

```
 while(p!=NULL && p→ data != data)
  { loc++;
    p=p→ next;
  }
 if(p==NULL)
  printf("\nNot found:");
 else
  printf("\nFound at location=%d",loc);
}
void print(node *head)
{ node *p;
 printf("\n\n");
 for(p=head;p!=NULL;p=p→ next)
  printf("%d  ",p→ data);
}
node *reverse(node *head)
  { node *p,*q,*r;
   p=NULL;
   q=head;
   r=q→ next;
   while(q!=NULL)
     {
    q→ next=p;
    p=q;
    q=r;
    if(q!=NULL)
      r=q→ next;
     }
  return(p);
  }
```

Output

```
1)create
2)Insert
3)Delete
4)Search
5)Reverse
6)Print
7)Quit
Enter your Choice:1
Enter no of data:3
Enter the data:5
6
9

1)create
2)Insert
3)Delete
4)Search
5)Reverse
6)Print
7)Quit
Enter your Choice:2
      1)Beginning
      2)End
      3)In between
Enter your choice : 1
Enter the data to be inserted : 3

1)create
2)Insert
3)Delete
4)Search
5)Reverse
6)Print
7)Quit
Enter your Choice:4
Enter the data to be searched: 6
Found at location=3
```

```
1)create
2)Insert
3)Delete
4)Search
5)Reverse
6)Print
7)Quit
Enter your Choice:3
      1)Beginning
      2)End
      3)In between
Enter your choice : 3
Enter the data to be deleted : 6

1)create
2)Insert
3)Delete
4)Search
5)Reverse
6)Print
7)Quit
Enter your Choice:5
9 5 3

1)create
2)Insert
3)Delete
4)Search
5)Reverse
6)Print
7)Quit
Enter your Choice:7
```

L.8 Singly Circular Linked List

- **Program 8 : Write C program to perform the operations (Insert, Delete, Traverse, and Search) on Circular Singly Linked List.**

```
/*Implement singly CLL(circular linked list) */
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
typedef struct node
  { int data;
   struct node *next;
  }node;

node *create();
node *insert_b(node *head,int x);
node *insert_e(node *head,int x);
node *insert_in(node *head,int x);
node *delete_b(node *head);
node *delete_e(node *head);
node *delete_in(node *head);
void search(node *head);
void print(node *head);

void main()
{ int op,op1,x;
 node *head=NULL;
 clrscr();
 do
  {
   printf("\n\n1)create\n2)Insert\n3)Delete\n4)Search");
   printf("\n5)Print\n6)Quit");
   printf("\nEnter your Choice:");
   scanf("%d",&op);
   switch(op)
```

```
    { case 1:head=create();break;
    case 2:printf("\n\t1)Beginning\n\t2)End\n\t3)Anywhere");
        printf("\nEnter your choice : ");
        scanf("%d",&op1);
        printf("\nEnter the data to be inserted : ");
        scanf("%d",&x);
        switch(op1)
         { case 1: head=insert_b(head,x);
                break;
          case 2: head=insert_e(head,x);
                break;
          case 3: head=insert_in(head, x);
                break;
         }
        break;
    case 3:printf("\n\t1)Beginning\n\t2)End\n\t3)Anywhere");
        printf("\nEnter your choice : ");
        scanf("%d",&op1);
        switch(op1)
         { case 1:head=delete_b(head);
                break;
          case 2:head=delete_e(head);
                break;
          case 3:head=delete_in(head);
                break;
         }
        break;
    case 4:search(head);break;
    case 5:print(head);break;
    }
  }while(op!=6);
}

node *create()
{ node *head,*p;
 int i,n,x;
 head=NULL;
 printf("\n Enter no of data:");
 scanf("%d",&n);
 printf("\nEnter the data:");
 for(i=0;i<n;i++)
  {
   scanf("%d",&x);
   if(head==NULL)
    {
      head=(node*)malloc(sizeof(node));
      head->next=head;
      head->data=x;
    }
   else
    {
      p=(node*)malloc(sizeof(node));
      p->data=x;
      p->next=head->next;
      head->next=p;
      head=p;
    }
  }
 return(head);
}

node *insert_b(node *head,int x)
{  node *p;
   p=(node*)malloc(sizeof(node));
   p->data=x;
   if(head==NULL)
    {
      p->next=p;
```

```
        head=p;
    }
  else
    {
        p->next=head->next;
        head->next=p;

    }
  return(head);
}

node *insert_e(node *head,int x)
{  node *p;
  p=(node*)malloc(sizeof(node));
  p->data=x;
  if(head==NULL)
    {
        p->next=p;
        head=p;
    }
  else
    {
        p->next=head->next;
        head->next=p;
        head=p;
    }
  return(head);
}


node *insert_in(node *head,int x)
{  node *p,*q;
   int loc,i;
   p=(node*)malloc(sizeof(node));
   p->data=x;
   p->next=p;
   printf("\Enter the location : ");
   scanf("%d",&loc);
   if(loc==1) // inserting as a first node
     {  if(head==NULL)
          return(p);
        else
          {
            p->next=head->next;
            head->next=p;
          }
     }
   else
     {
       q=head->next;
       for(i=1; i<loc-1;i++)
          if(q->next != head->next)
             q=q->next;
          else
            {
              printf("\nOverflow ****\n ");
              return head;
            }

   //insert a node as next node of q
       p->next=q->next;
       q->next=p;
       if(q==head)
         head=p;
     }
 return(head);
 }
node *delete_b(node *head)
```

```
{
 node *p,*q;
 if(head==NULL)
   {
   printf("\nUnderflow....Empty Linked List");
   return(head);
   }
 p=head->next;
 if(head->next==head)
   head=NULL;
 else
   head->next=p->next;
 free(p);
 return(head);

}
node *delete_e(node *head)
{
 node *p,*q;
 if(head==NULL)
   {
   printf("\nUnderflow....Empty Linked List");
   return(head);
   }
 p=head->next;
 if(head->next==head)
   { // Delete the only element
    head=NULL;
    free(p);
    return(head);
   }
//Locate the last but one node
 for(q=head->next;q->next!=head;q=q->next)
 ;
 p=head;
 head=q;
 head->next=p->next;
 free(p);
 return(head);
}
node *delete_in(node *head)
{
 node *p,*q;
 int loc,i;
 if(head==NULL)
   {
   printf("\nUnderflow....Empty Linked List");
   return(head);
   }
 printf("\nEnter the location of the data to be deleted : ");
 scanf("%d",&loc);
 if(loc==1)
   { // Delete the first element
    p=head->next;
    if(head->next==head)
    head=NULL;
    else
    head->next=p->next;
    free(p);
    return(head);
   }
 else
   {
   q=head->next;
   for(i=1; i<loc-1;i++)
       if(q->next->next!=head->next)
           q=q->next;
       else
```

```
            {
              printf("\nunderflow **** ");
              return head;
            }
    p=q->next;
    q->next=p->next;
    if(p==head)
      head=q;
    free(p);
    }
 return(head);
}

void search(node *head)
{ node *p;
 int data,loc=1;
 printf("\nEnter the data to be searched: ");
 scanf("%d",&data);
 p=head->next;
 do
   {
   if(data==p->data)
     {
     printf("\nFound at location=%d",loc);
     return;
     }
   loc++;
   p=p->next;
  }while(p!=head->next);
  printf("\nNot found:");
}

void print(node *head)
 {
   node *p;
   printf("\n\n");
   if(head==NULL)
       return;
   p=head->next;
   do
    {
       printf("%d  ",p->data);
       p=p->next;
    } while(p!=head->next);
 }
```

Output

```
1)create
2)Insert
3)Delete
4)Search
5)Print
6)Quit
Enter your Choice:1

 Enter no of data:5

Enter the data:26 54 3 14 52



1)create
2)Insert
3)Delete
4)Search
5)Print
6)Quit
Enter your Choice:5
```

```
26 54 3 14 52

1)create
2)Insert
3)Delete
4)Search
5)Print
6)Quit
Enter your Choice:2

    1)Beginning
    2)End
    3)Anywhere
Enter your choice : 2

Enter the data to be inserted : 96

1)create
2)Insert
3)Delete
4)Search
5)Print
6)Quit
Enter your Choice:3

    1)Beginning
    2)End
    3)Anywhere
Enter your choice : 1

1)create
2)Insert
3)Delete
4)Search
5)Print
6)Quit
Enter your Choice:5

54 3 14 52 96

1)create
2)Insert
3)Delete
4)Search
5)Print
6)Quit
Enter your Choice:4

Enter the data to be searched: 52

Found at location=4
```

L.9 Operations on BST

- **Program 9 : Write C program to implement BST (Binary Search Tree) and traverse the tree (Inorder, Preorder, Post order).**

```
/* A menu driven program for binary search tree creation and inorder,
  preorder,postorder recursive traversals of all nodes.Modify the program
  for performing following operations:
  (a) Search a value (b) Insertion of a value (c) Deletion of a value
*/

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

```
typedef struct BSTnode
{
    int data;
    struct BSTnode *left,*right;
}BSTnode;

BSTnode *find(BSTnode *,int);
BSTnode *insert(BSTnode *,int);
BSTnode *delet(BSTnode *T,int x);
BSTnode *create();
void inorder(BSTnode *T);
void preorder(BSTnode *T);
void postorder(BSTnode *T);

void main()
 {
   BSTnode *root=NULL,*p;
   int x,op;
   clrscr();
   do
    {
printf("\n\n1)Create\n2)Insert\n3)Search\n4)Preorder(recursiv
e)");
      printf("\n5)Inorder(recursive)\n6)Posorder(Recursive)");
      printf("\n7)Delete\n8)Quit");
      printf("\nEnter Your Choice :");
      scanf("%d",&op);
       switch(op)
        {
       case 1: root=create();break;
       case 2:printf("\nEnter the key to be inserted :");
           scanf("%d",&x);
           root=insert(root,x);
           break;
       case 3:printf("\nEnter the key to be searched :");
           scanf("%d",&x);
           p=find(root,x);
           if(p==NULL)
            printf("\n ***** Not Found****");
           else
            printf("\n ***** Found*****");
           break;
       case 4: preorder(root);break;
       case 5: inorder(root);break;
       case 6: postorder(root);break;
       case 7:printf("\nEnter the key to be deleted :");
           scanf("%d",&x);
           root=delet(root,x);
           break;
        }
    }while(op!=8);
}
void inorder(BSTnode *T)
{
   if(T!=NULL)
   {
      inorder(T->left);
      printf("%d\t",T->data);
      inorder(T->right);
   }
}

void preorder(BSTnode *T)
{     if(T!=NULL)
     { printf("%d\t",T->data);
      preorder(T->left);
      preorder(T->right);
     }
```

```
}
void postorder(BSTnode *T)
{       if(T!=NULL)
      {
        postorder(T->left);
        postorder(T->right);
        printf("%d\t",T->data);
      }
}
BSTnode *find(BSTnode *root,int x)
{
    while(root!=NULL)
    {
        if(x==root->data)
            return(root);
        if(x>root->data)
            root=root->right;
        else
        root=root->left;
    }
    return(NULL);
}

BSTnode *insert(BSTnode *T,int x)
{
    BSTnode *r;
    // acquire memory for the new node
    if(T==NULL)
     {
        r=(BSTnode*)malloc(sizeof(BSTnode));
        r->data=x;
        r->left=NULL;
        r->right=NULL;
        return(r);
     }



    if(x>T->data)
     {
        T->right=insert(T->right,x);
        return(T);
     }
    else
        if(x<T->data)
         {
            T->left=insert(T->left,x);
            return(T);
         }
        else //duplicate data
            return(T);
}
BSTnode *create()
{
    int n,x,i;
    BSTnode *root;
    root=NULL;
    printf("\nEnter no. of nodes :");
    scanf("%d",&n);
    printf("\nEnter tree values :");
    for(i=0;i<n;i++)
    {
        scanf("%d",&x);
        root=insert(root,x);
    }
    return(root);
}

BSTnode *delet(BSTnode *T,int x)
```

```
{
    BSTnode *temp;
    if(T==NULL)
    {
        printf("\nElement not found :");
        return(T);
    }
    if(x < T->data)             // delete in left subtree
    {
        T->left=delet(T->left,x);
        return(T);
    }
    if(x > T->data)             // delete in right subtree
    {
        T->right=delet(T->right,x);
        return(T);
    }
    // element is found
    if(T->left==NULL && T->right==NULL)   // a leaf
node
    {
        temp=T;
        free(temp);
        return(NULL);
    }
    if(T->left==NULL)
    {
        temp=T;
        T=T->right;
        free(temp);
        return(T);
    }
    if(T->right==NULL)
    {
        temp=T;
        T=T->left;
        free(temp);
        return(T);
    }
    // node with two children
    //go to the inorder successor of the node
    temp=T->right;
    while(temp->left !=NULL)
     temp=temp->left;
    T->data=temp->data;
    T->right=delet(T->right,temp->data);
    return(T);
}
```

Output

```
1)Create
2)Insert
3)Search
4)Preorder(recursive)
5)Inorder(recursive)
6)Posorder(Recursive)
7)Delete
8)Quit
Enter Your Choice :1

Enter no. of nodes :3

Enter tree values :562 84 213


1)Create
2)Insert
3)Search
4)Preorder(recursive)
```

```
5)Inorder(recursive)
6)Posorder(Recursive)
7)Delete
8)Quit
Enter Your Choice :2

Enter the key to be inserted :22



1)Create
2)Insert
3)Search
4)Preorder(recursive)
5)Inorder(recursive)
6)Posorder(Recursive)
7)Delete
8)Quit
Enter Your Choice :3

Enter the key to be searched :84

***** Found*****
1)Create
2)Insert
3)Search
4)Preorder(recursive)
5)Inorder(recursive)
6)Posorder(Recursive)
7)Delete
8)Quit
Enter Your Choice :4
562   84   22   213

1)Create
2)Insert
3)Search
4)Preorder(recursive)
5)Inorder(recursive)
6)Posorder(Recursive)
7)Delete
8)Quit
Enter Your Choice :5
22   84   213   562

1)Create
2)Insert
3)Search
4)Preorder(recursive)
5)Inorder(recursive)
6)Posorder(Recursive)
7)Delete
8)Quit
Enter Your Choice :6
22   213   84   562
```

❑❑❑

Appendix A

Solved MSBTE Question Paper of Winter 2018

Data Structures Using C

Winter 2018

Q. 1(a) Define the term algorithm.
(Section 1.6) **(02 Marks)**

Q. 1(b) List any 4 applications of queue.
(Section 4.5) **(02 Marks)**

Q. 1(c) Describe following terms w.r.to tree: **(02 Marks)**
(i) leaf node **(Section 6.1.1)**
(ii) level of node **(Section 6.1.1)**

Q. 1(d) Differentiate between stack and queue.
(any two points) **(Section 4.1.3)** **(02 Marks)**

Q. 1(e) Describe undirected graph with suitable example.
(Section 7.1.2) **(02 Marks)**

Q. 1(f) Define the terms : linear data structure and non-linear data structure.**(Section 1.3.2) (02 Marks)**

Q. 1(g) Convert infix expression into prefix expression : (A + B)*(C / G) +F **(02 Marks)**

Ans. :

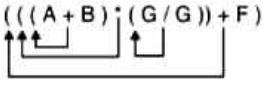
Step 1 :

Fully parenthesizing the infix expression, we get

(((A + B) * (C/G)) + F)

Step 2 :

Moving operators to replace their corresponding left parenthesis



+ * + AB/CGF : Required prefix expression

Q. 2(a) Describe working of linear search with example.
(Section 2.8.1) **(04 Marks)**

Q. 2(b) Describe the concept of linked list with the terminologies : node, next pointer, null pointer and empty list. **(Section 5.1.2)** **(04 Marks)**

Q. 2(c) Describe queue full and queue empty operation conditions on linear queue with suitable diagrams.
(Section 4.2) **(04 Marks)**

Q. 2(d) Differentiate between general tree and binary tree.
(any four points) **(Section 6.3.1)** **(04 Marks)**

Q. 3(a) Write 'c' program for deletion of an element from an array.
(Program 1 from Lab manual) **(04 Marks)**

Q. 3(b) Convert following expression into postfix form. Give stepwise procedure.
A + B ↑ C * (D / E) – F / G **(04 Marks)**

Ans. :

Step 1 :

Reversing the expression, we get

G/F – (E/D) * C ↑ B + A

Step 2 :

Converting to postfix form

Sr. No.	Stack	Infix expression	Postfix expression
1.	Empty	G/F – (E/D) *C↑B + A	–
2.	Empty	/F – (E/D) *C↑B + A	G
3.	/	F – (E/D) * C↑B + A	G
4.	/	– (E/D) * C↑B + A	GF
5.	–	(E/D) *C↑B + A	GF/
6.	– (	E/D) * C↑B + A	GF/
7.	– (	/D) * C↑B + A	GF/E
8.	– (/	D) * C↑B + A	GF/E
9.	– (/	) * C↑B + A	GF/ED
10.	–	*C↑B + A	GF/ED/
11.	– *	C↑B + A	GF/ED/

Sr. No.	Stack	Infix expression	Postfix expression
12.	– *	↑B + A	GF /ED/C
13.	– * ↑	B + A	GF/Ed/C
14.	– * ↑	+ A	GF/ED/CB
15.	– +	A	GF/ED/CB↑*
16.	– +	–	GF/ED/CB↑ * A
17.	Empty	–	GF/ED/CB↑ * A + –

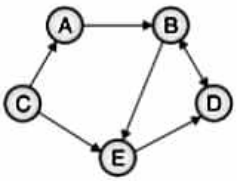
Step 3 :

Reversing the expression, we set the prefix form

– + A * ↑ BC/DE/FG

Q. 3(c) Find the position of element 29 using binary search method in an array 'A' given below. Show each step. A = {11, 5, 21, 3, 29, 17, 2, 43} **(Example 2.8.2)** **(04 Marks)**

Q. 3(d) Give adjacency list and adjacency matrix for given graph : **(04 Marks)**

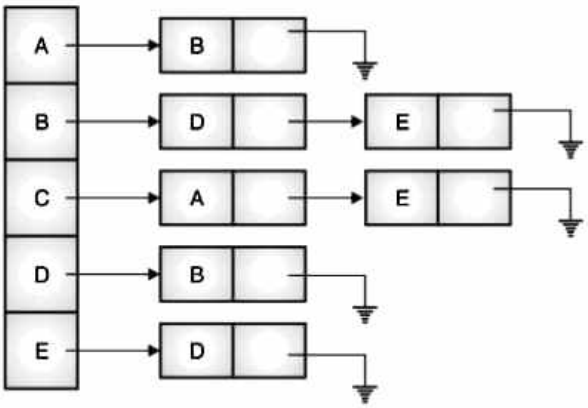


Ans. :

(i) Adjacency Matrix

	A	B	C	D	E
A	0	1	0	0	0
B	0	0	0	1	1
C	1	0	0	0	1
D	0	1	0	0	0
E	0	0	0	1	0

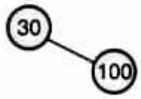
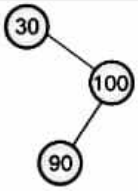
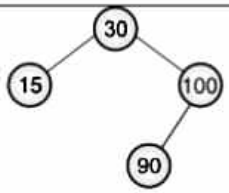
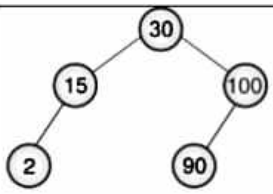
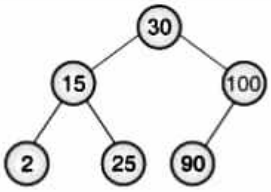
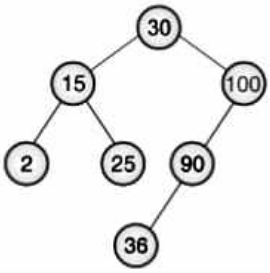
(ii) Adjacency List

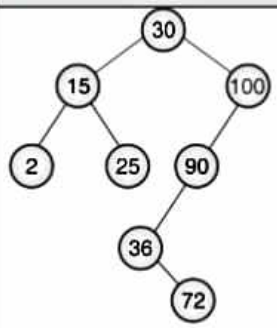
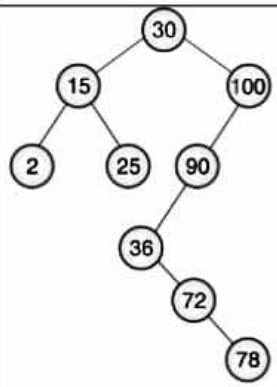
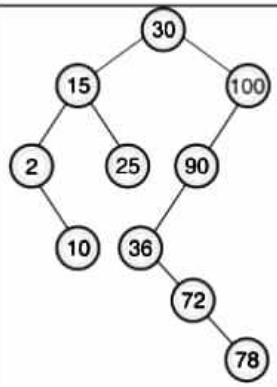


Q. 4(a) Describe working of bubble sort with example. **(Section 2.2)** **(04 Marks)**

Q. 4(b) Construct a binary search tree for following elements : 30, 100, 90, 15, 2, 25, 36, 72, 78, 10 show each step of construction of BST. **(04 Marks)**

Ans. :

Sr. No.	Element to be inserted	Binary Search Tree
1.	30	
2.	100	
3.	90	
4.	15	
5.	2	
6.	25	
7.	36	

Sr. No.	Element to be Inserted	Binary Search Tree
8.	72	
9.	78	
10.	10	

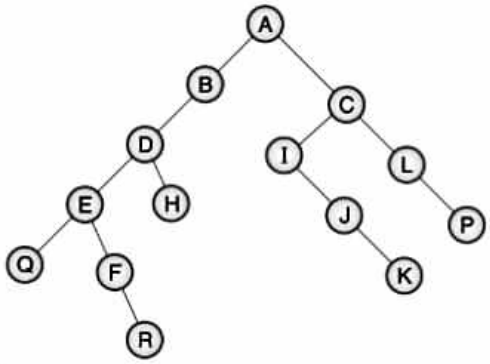
Q. 4(c) Write an algorithm to count number of nodes in singly linked list. **(Section 5.2.3)** **(04 Marks)**

Q. 4(d) Write a program in 'C' to insert an element in a linear queue. **(Program 4.2)** **(04 Marks)**

Q. 4(e) Describe circular linked list with suitable diagram. Also state advantage of circular linked list over linear linked list. **(Section 5.3)** **(04 Marks)**

Q. 5(a) Write algorithm for performing push and pop operations on stack.
(Section 3.4.1) **(06 Marks)**

Q. 5(b) For given binary tree write in-order, pre-order and post-order traversal. **(06 Marks)**



Ans. :

Preorder : ABDEQFRHCIJKLP

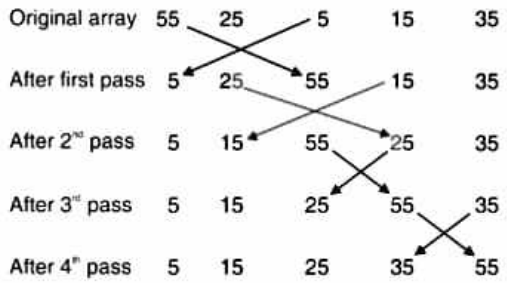
Inorder : QEFRDHBAIJKCLP

Postorder : QRFEHDBKJIPCA

Q. 5(c) Write an algorithm to insert an element at the beginning and at end of linked list.
(Sections 5.2.5 and 5.2.5(A)) **(06 Marks)**

Q. 6(a) Describe working of selection sort method. Also sort given input list in ascending order using selection sort input list : 55, 25, 5, 15, 35.
(Section 2.3) **(06 Marks)**

Ans. :



Q. 6(b) Define the term recursion. Write a program in C to display factorial of a entered number using recursion.
(Section 3.8 and Program 3.8.1) **(06 Marks)**

Q. 6(c) Describe procedure to delete an element from singly linked list using diagram.
(Section 5.2.6) **(06 Marks)**

❑❑❑

Note

Note

Note