

New Edition
MU

As per the New Revised Syllabus (REV- 2019 'C' Scheme)
of Mumbai University w.e.f. academic year 2020-21

Data Structure

(Code : CSC303)

Semester III

Computer Engineering / Computer Science and Engineering /
Artificial Intelligence & Data Science / Machine Learning / Cyber Security /
Internet of Things (IoT) / Data Engineering / Data Science /
Internet of Things and Cyber Security Including Block Chain Technology /
Computer Science Design

Dilip Kumar Sultania

Includes :

Solved Latest University Question Papers.



TechKnowledge
Publications

44

DATA STRUCTURE

(Code : CSC303)

Semester III – (Mumbai University)

Computer Engineering / Computer Science and Engineering
Artificial Intelligence and Data Science / Machine Learning
Cyber Security / Internet of Things (IOT) / Data Engineering
/ Data Science Internet of Things and Cyber Security Including
Block Chain Technology / Computer science Design

Strictly as per the New Revised Syllabus (Rev – 2019 'C' Scheme)
of Mumbai University w.e.f. academic year 2020-2021
(As per Choice Based Credit and Grading System)

Dilip Kumar Sultania

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



M0146C Price ₹ 485/-



DATA STRUCTURE (Code : CSC303)

Dilip Kumar Sultanta

(Semester III - Computer Engineering / Computer Science and Engineering / Artificial Intelligence and Data Science / Machine Learning / Cyber Security / Internet of Things (IOT) / Data Engineering / Data Science / Internet of Things and Cyber Security Including Block Chain Technology, Computer science Design MU)

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 (For Pune University)

First Edition : August 2020 (Rev – 2019 'C' Scheme)

Second Revised Edition : July 2021

Third Revised Edition : July 2022

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.

ISBN : 978-93-89889-98-7

Published By

TECHKNOWLEDGE PUBLICATIONS

Printed @

37/2, Ashtavinayak Industrial Estate,
Near Pari Company,
Narhe, Pune, Maharashtra State, India.
Pune - 411041

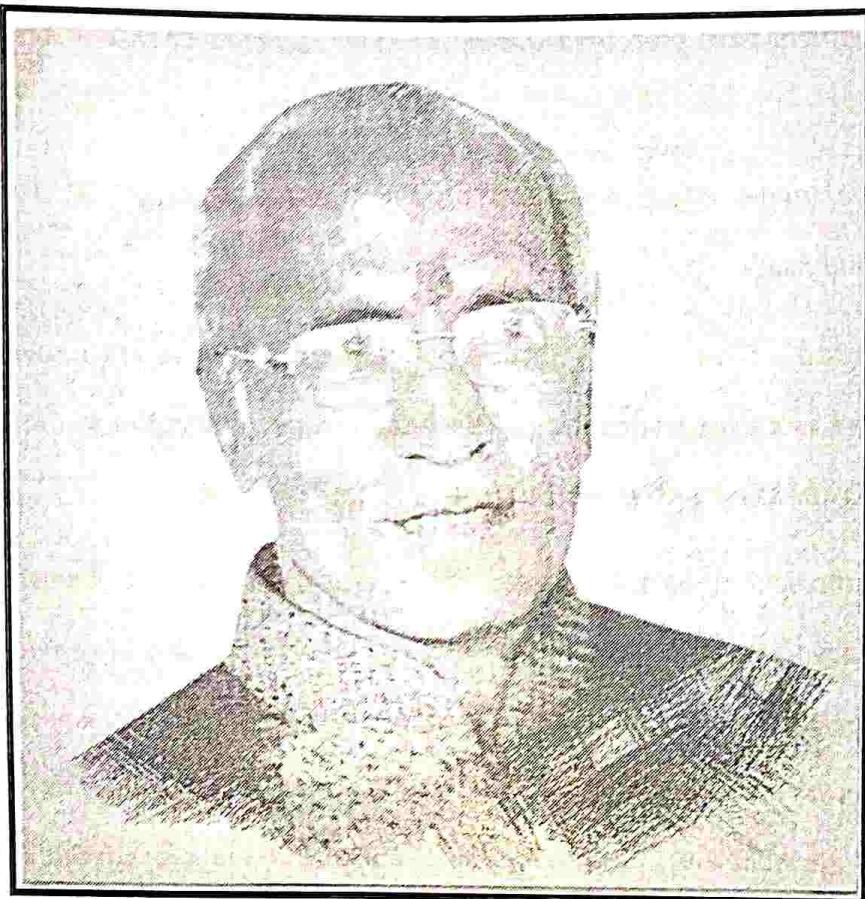
Head Office

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

Subject Code : CSC303

Book Code : M0146C

We dedicate this Publication soulfully and wholeheartedly, in loving memory of our beloved founder director,
Late Shri. Pradeepji Lalchandji Lunawat, who will always be an inspiration, a positive force and strong support behind us.



“My work is my prayer to God”

- *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 Structure" 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 programs and functions have been included. So, that this book will cater for all your needs.

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 thankful to Shri. J. S. Katre, Shri. Shital Bhandari, Shri. Arunoday Kumar and Shri. Chandrodai Kumar for the encouragement and support that they have extended. I am also thankful to Seema Lunawat for technology enhanced reading, E-books support and 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

Mumbai University Revised syllabus (Rev-2019 'C' Scheme) from Academic Year 2020-21

| Course Code | Course Name | Credits |
|-------------|----------------|---------|
| CSC303 | Data Structure | 03 |

Prerequisite: C Programming

Course Objectives:

1. *To understand the need and significance of Data structures as a computer Professional.*
2. *To teach concept and implementation of linear and Nonlinear data structures.*
3. *To analyze various data structures and select the appropriate one to solve a specific real-world problem.*
4. *To introduce various techniques for representation of the data in the real world.*
5. *To teach various searching techniques.*

Course Outcomes:

1. *Students will be able to implement Linear and Non-Linear data structures.*
2. *Students will be able to handle various operations like searching, insertion, deletion and traversals on various data structures.*
3. *Students will be able to explain various data structures, related terminologies and its types.*
4. *Students will be able to choose appropriate data structure and apply it to solve problems in various domains.*
5. *Students will be able to analyze and Implement appropriate searching techniques for a given problem.*
6. *Students will be able to demonstrate the ability to analyze, design, apply and use data structures to solve engineering problems and evaluate their solutions.*

| Module | | Detailed Content | Hours |
|--------|-----|---|-------|
| 01 | 1.1 | Introduction to Data Structures Introduction to Data Structures, Concept of ADT, Types of Data Structures-Linear and Nonlinear, Operations on Data Structures. (Refer Chapter 1) | 02 |
| 02 | 2.1 | Stack and Queues Introduction, ADT of Stack, Operations on Stack, Array Implementation of Stack, Applications of Stack-Well form-ness of Parenthesis, Infix to Postfix Conversion and Postfix Evaluation, Recursion. | 08 |
| | 2.2 | Introduction, ADT of Queue, Operations on Queue, Array Implementation of Queue, Types of Queue-Circular Queue, Priority Queue, Introduction of Double Ended Queue, Applications of Queue. (Refer Chapter 2) | |
| 03 | 3.1 | Linked List Introduction, Representation of Linked List, Linked List v/s Array, Types of Linked List - Singly Linked List, Circular Linked List, Doubly Linked List, Operations on Singly Linked List and Doubly Linked List, Stack and Queue using Singly Linked List, Singly Linked List Application-Polynomial Representation and Addition. (Refer Chapter 3) | 10 |
| 04 | 4.1 | Trees Introduction, Tree Terminologies, Binary Tree, Binary Tree Representation, Types of Binary Tree, Binary Tree Traversals, Binary Search Tree, Operations on Binary Search Tree, Applications of Binary Tree-Expression Tree, Huffman Encoding, Search Trees-AVL, rotations in AVL Tree, operations on AVL Tree, Introduction of B Tree, B+ Tree. (Refer Chapter 4) | 11 |
| 05 | 5.1 | Graphs Introduction, Graph Terminologies, Representation of Graph, Graph Traversals- Depth First Search (DFS) and Breadth First Search (BFS), Graph Application- Topological Sorting. (Refer Chapter 5) | 04 |
| 06 | 6.1 | Sorting and Searching Linear Search, Binary Search, Hashing-Concept, Hash Functions, Collision resolution Techniques. (Refer Chapter 6) | 04 |

□□□

Mumbai University
Revised syllabus (Rev-2019 'C' Scheme) from Academic Year 2020-21

| Lab Code | Lab Name | Credit |
|----------|---------------------|--------|
| CSL301 | Data Structures Lab | 01 |

Prerequisite: C Programming Language

Lab Objectives:

1. To implement basic data structures such as arrays, linked lists, stacks and queues
2. Solve problem involving graphs and trees
3. To develop application using data structure algorithms
4. Compute the complexity of various algorithms.

Lab Outcomes:

1. Students will be able to implement linear data structures and be able to handle operations like insertion, deletion, searching and traversing on them.
2. Students will be able to implement nonlinear data structures and be able to handle operations like insertion, deletion, searching and traversing on them
3. Students will be able to choose appropriate data structure and apply it in various problems
4. Students will be able to select appropriate searching techniques for given problems.

Suggested Experiments:

- Students are required to complete atleast 10 experiments.
 - Star (*) marked experiments are compulsory.
- 1* Implement Stack ADT using array.
 - 2* Convert an Infix expression to Postfix expression using stack ADT.
 - 3* Evaluate Postfix Expression using Stack ADT.
 - 4 Applications of Stack ADT.
 - 5* Implement Linear Queue ADT using array.
 - 6* Implement Circular Queue ADT using array.
 - 7 Implement Priority Queue ADT using array.
 - 8* Implement Singly Linked List ADT.
 - 9* Implement Circular Linked List ADT.
 - 10 Implement Doubly Linked List ADT.
 - 11* Implement Stack / Linear Queue ADT using Linked List.
 - 12* Implement Binary Search Tree ADT using Linked List.
 - 13* Implement Graph Traversal techniques : a) Depth First Search b) Breadth First Search
 - 14 Applications of Binary Search Technique.



Module 1**Chapter 1 : Introduction to Data Structures 1-1 to 1-27**

Syllabus : Introduction to Data Structures, Concept of ADT, Types of Data Structures-Linear and Nonlinear, Operations on Data Structures.

| | | |
|----------|--|------|
| 1.1 | Data..... | 1-1 |
| 1.1.1 | Data Types | 1-1 |
| 1.1.2 | Abstract Data Types (ADT) | 1-1 |
| 1.1.3 | Data Object | 1-3 |
| 1.2 | Data Structures..... | 1-3 |
| 1.2.1 | Types of Data Structures..... | 1-3 |
| 1.2.1(A) | Primitive and Non-Primitive | 1-3 |
| 1.2.1(B) | Linear and Non-Linear..... | 1-4 |
| 1.2.1(C) | Static and Dynamic..... | 1-5 |
| 1.3 | Relationship among Data Object, Data Type, Data Structure and Data Representation..... | 1-5 |
| 1.3.1 | Operations on Data Structure..... | 1-5 |
| 1.4 | Algorithm Analysis | 1-6 |
| 1.4.1 | Measuring the Running Time of a Program (Time Complexity)..... | 1-7 |
| 1.4.2 | Measurement of Growth Rate (Asymptotic Growth Rate) | 1-7 |
| 1.4.2(A) | Asymptotic Consideration..... | 1-7 |
| 1.4.2(B) | Constant Factor in Complexity Measure..... | 1-7 |
| 1.4.3 | Notation O : (Pronounced as Big-Oh), ($O(n^2)$ is Pronounced as Big-Oh of n^2) | 1-8 |
| 1.4.4 | Best Case, Worst Case and the Average Case Behaviour..... | 1-10 |
| 1.5 | Introduction to Arrays | 1-11 |
| 1.6 | Representation and Analysis..... | 1-11 |
| 1.7 | One-Dimensional Arrays | 1-12 |
| 1.8 | Operations with Arrays | 1-13 |
| 1.8.1 | Deletion | 1-14 |
| 1.8.2 | Insertion..... | 1-14 |
| 1.8.3 | Search..... | 1-16 |
| 1.8.4 | Merging of Sorted Arrays..... | 1-16 |
| 1.9 | Two-Dimensional Arrays | 1-18 |
| 1.9.1 | Initializing Two-Dimensional Arrays | 1-19 |
| 1.9.2 | Address Calculation..... | 1-19 |
| 1.10 | Multi-Dimensional Arrays..... | 1-21 |
| 1.11 | Application of Arrays..... | 1-21 |
| 1.11.1 | Addition of Two 2-D Matrices | 1-21 |
| 1.11.2 | Transpose of Square Matrix | 1-24 |

| | | |
|--------|--|------|
| 1.11.3 | Finding whether a given Square Matrix Is Symmetrical | 1-25 |
| 1.11.4 | Multiplication of Two Matrices $A_{m \times n}$ and $B_{n \times p}$ | 1-26 |

Module 2**Chapter 2 : Stack and Queues**

2-1 to 2-63

Syllabus : Introduction, ADT of Stack, Operations on Stack, Array Implementation of Stack, Applications of Stack – Well-formedness of Parenthesis, Infix to Postfix Conversion and Postfix Evaluation, Recursion, Introduction, ADT of Queue, Operations on Queue, Array Implementation of Queue, Types of Queue-Circular Queue, Priority Queue, Introduction of Double Ended Queue, Applications of Queue.

| | | |
|-------|--|------|
| 2.1 | Introduction | 2-1 |
| 2.2 | Operations on Stacks | 2-1 |
| 2.3 | Array Representation | 2-1 |
| 2.3.1 | 'C' Functions for Primitive Operations on a Stack | 2-2 |
| 2.3.2 | Program Showing Stack Operations | 2-2 |
| 2.3.3 | Well-Formedness of Parenthesis | 2-5 |
| 2.3.4 | Operations on Stack Considering Overflow and Underflow | 2-6 |
| 2.3.5 | Stack as an ADT | 2-6 |
| 2.4 | Applications of Stack..... | 2-7 |
| 2.4.1 | Expression Representation | 2-7 |
| 2.4.2 | Evaluation of a Postfix Expression using a Stack..... | 2-8 |
| 2.4.3 | Conversion of an Expression from Infix to Postfix | 2-12 |
| 2.5 | Expression Conversion (A Fast Method)..... | 2-22 |
| 2.5.1 | Infix to Postfix | 2-22 |
| 2.5.2 | Algorithm to Check Well-Formedness of Parenthesis | 2-23 |
| 2.6 | Introduction to Recursion | 2-27 |
| 2.7 | Converting a Recursive Function to an Equivalent C-Function | 2-27 |
| 2.7.1 | Finding Factorial of an Integer Number | 2-27 |
| 2.7.2 | Finding n^{th} Term of Fibonacci Sequence Recursive Definition | 2-28 |
| 2.7.3 | Finding GCD of given Numbers | 2-28 |
| 2.7.4 | Calculation of x^n using Recursion | 2-29 |
| 2.7.5 | Calculation of Sum of Digits | 2-29 |
| 2.8 | Examples of Recursion | 2-29 |



| | | | | | |
|-----------|--|------|----------|---|------|
| 2.8.1 | Finding Sum of the Elements Stored in an Array..... | 2-29 | 3.1 | Representation and Implementation of Singly Linked Lists..... | 3-1 |
| 2.8.1(A) | 'C' Function for Finding Sum of the Elements of an Array..... | 2-29 | 3.1.1 | Comparison between Array and Linked Lists | 3-1 |
| 2.8.2 | Finding Length of a String | 2-30 | 3.1.2 | Representation..... | 3-1 |
| 2.8.3 | Reversing a String..... | 2-30 | 3.1.3 | Implementation..... | 3-2 |
| 2.8.4 | Searching a Number in an Array | 2-30 | 3.1.4 | Types of Linked List | 3-3 |
| 2.8.5 | Finding Largest Element in an Array | 2-30 | 3.1.4(A) | Singly Linked List | 3-3 |
| 2.8.6 | Binary Search..... | 2-30 | 3.1.4(B) | Doubly Linked List..... | 3-3 |
| 2.8.7 | Tower of Hanoi Problem..... | 2-31 | 3.1.4(C) | A Circular Linked List | 3-3 |
| 2.9 | Solved Examples..... | 2-32 | 3.1.5 | Differences between Singly Linked List and Doubly Linked List | 3-3 |
| 2.10 | Removal of Recursion | 2-36 | 3.2 | Basic Linked List Operations | 3-4 |
| 2.11 | Tail Recursion..... | 2-37 | 3.2.1 | Creating a Linked List..... | 3-4 |
| 2.12 | Array and Linked Representation and Implementation of Queues | 2-38 | 3.2.2 | Traversing a Linked List | 3-5 |
| 2.12.1 | Definition | 2-38 | 3.2.3 | Counting Number of Nodes in a Linked List through Count Function..... | 3-5 |
| 2.12.2 | Application of Queues | 2-39 | 3.2.4 | Printing a List through Print Function | 3-6 |
| 2.12.3 | Array Representation and Implementation of Queues..... | 2-39 | 3.2.5 | Inserting an Item | 3-6 |
| 2.13 | Operations on Queue | 2-40 | 3.2.5(A) | Inserting an Item at the End of a Linked List..... | 3-7 |
| 2.13.1 | Operations on Queue Implemented using Array | 2-40 | 3.2.5(B) | Inserting a Data 'x' at a given Location 'LOC' in a Linked List, Referenced by 'head' | 3-8 |
| 2.14 | Circular Queues..... | 2-44 | 3.2.5(C) | Inserting an Element in a Priority Linked List..... | 3-10 |
| 2.14.1 | Queue using a Circular Array | 2-44 | 3.2.6 | Deleting an Item..... | 3-10 |
| 2.14.1(A) | Implementation of a Circular Movement Inside a Linear Array..... | 2-45 | 3.2.6(A) | Deletion of the Last Node of a Linked List..... | 3-11 |
| 2.15 | Applications of Queue | 2-52 | 3.2.6(B) | Deletion of a Node at Location 'LOC' from a Linked List..... | 3-12 |
| 2.15.1 | Categorizing Data..... | 2-52 | 3.2.6(C) | Delete a Linked List, Referenced by the Pointer Head | 3-13 |
| 2.15.2 | Job Scheduling..... | 2-52 | 3.2.7 | Concatenation of Two Linked Lists..... | 3-13 |
| 2.15.3 | Queue Simulation..... | 2-53 | 3.2.8 | Inversion of Linked List..... | 3-13 |
| 2.16 | Priority Queue..... | 2-53 | 3.2.9 | Searching a Data 'x' in a Linked List, Referenced by the Pointer Head | 3-15 |
| 2.16.1 | Implementation of Priority Queues | 2-54 | 3.2.10 | Searching an Element x in a Sorted Linked List..... | 3-16 |
| 2.16.1(A) | Implementation of a Priority Queue using a Circular Array..... | 2-54 | 3.2.11 | New Linear Linked List by Selecting Alternate Element..... | 3-16 |
| 2.16.2 | Dequeues | 2-58 | 3.2.12 | Handling of Records through Linked List..... | 3-17 |
| 2.16.3 | Implementation of Dequeue using a Circular Array..... | 2-60 | 3.2.13 | Merging of Sorted Linked Lists..... | 3-17 |

Module 3

Chapter 3 : Linked List

3-1 to 3-59

Syllabus : Introduction, Representation of Linked List, Linked List v/s Array, Types of Linked List - Singly Linked List, Circular Linked List, Doubly Linked List, Operations on Singly Linked List and Doubly Linked List, Stack and Queue using Singly Linked List, Singly Linked List Application-Polynomial Representation and Addition.



| | | |
|-------|--|------|
| 3.4 | Doubly Linked List..... | 3-28 |
| 3.4.1 | Creation of a Doubly Linked List..... | 3-29 |
| 3.4.2 | Deletion of a Node..... | 3-31 |
| 3.5 | Doubly Linked Circular List..... | 3-36 |
| 3.6 | Applications of Linked Lists | 3-37 |
| 3.6.1 | Polynomials as Linked Lists | 3-38 |
| 3.6.2 | Addition of Two Polynomials | 3-39 |
| 3.7 | Linked Representation of a Stack..... | 3-46 |
| 3.7.1 | Functions for Stack Operations | 3-47 |
| 3.8 | Linked Representation of a Queue..... | 3-50 |
| 3.8.1 | Comparison between Array Representation and the Linked Representation of a Queue..... | 3-50 |
| 3.8.2 | Operations on Queue Implemented using Linked Structure..... | 3-51 |
| 3.9 | Queue using a Circular Linked List..... | 3-55 |
| 3.9.1 | Implementation of a Priority Queue using a Linked List..... | 3-58 |

Module 4**Chapter 4 : Trees****4-1 to 4-102**

Syllabus : Introduction, Tree Terminologies, Binary Tree, Binary Tree Representation, Types of Binary Tree, Binary Tree Traversals, Binary Search Tree, Operations on Binary Search Tree, Applications of Binary Tree-Expression Tree, Huffman Encoding, Search Trees-AVL, rotations in AVL Tree, operations on AVL Tree, Introduction of B Tree, B+ Tree.

| | | |
|-------|---|-----|
| 4.1 | Basic Terminology..... | 4-1 |
| 4.1.1 | Introduction..... | 4-1 |
| 4.1.2 | Basic Terms | 4-1 |
| 4.2 | Binary Tree..... | 4-1 |
| 4.3 | Representation of a Binary Tree using an Array.. | 4-2 |
| 4.4 | Linked Representation of a Binary Tree | 4-3 |
| 4.4.1 | Program for Creation of a Sample Binary Tree | 4-4 |
| 4.4.2 | 'C' Function for Creation of a Binary Tree | 4-4 |
| 4.5 | A General Tree..... | 4-5 |
| 4.5.1 | Node Declaration for a Tree | 4-5 |
| 4.6 | Types of Binary Tree | 4-6 |
| 4.6.1 | Full Binary Tree | 4-6 |
| 4.6.2 | Complete Binary Tree..... | 4-7 |
| 4.6.3 | Skewed Binary Tree | 4-7 |
| 4.6.4 | Strictly Binary Tree | 4-7 |
| 4.6.5 | Extended Binary Tree (2-Tree)..... | 4-7 |

| | | |
|-----------|---|------|
| 4.7 | Binary Tree Traversal..... | 4-7 |
| 4.7.1 | Preorder Traversal (Recursive) | 4-8 |
| 4.7.1(A) | 'C' Function for Preorder Traversal | 4-8 |
| 4.7.2 | Inorder Traversal (Recursive)..... | 4-9 |
| 4.7.2(A) | 'C' Function for Inorder Traversal | 4-9 |
| 4.7.3 | Postorder Traversal (Recursive) | 4-10 |
| 4.7.3(A) | 'C' Function for Postorder Traversal..... | 4-10 |
| 4.7.4 | Non-Recursive Preorder Traversal..... | 4-10 |
| 4.7.4(A) | 'C' Function for Non-Recursive Preorder of Tree Along with the ADT Stack | 4-11 |
| 4.7.5 | Non-Recursive Inorder Traversal | 4-12 |
| 4.7.5(A) | 'C' Function for Non-Recursive Inorder Traversal of a Binary Tree..... | 4-12 |
| 4.7.6 | Non-Recursive Postorder Traversal | 4-12 |
| 4.7.6(A) | 'C' Function for Non-Recursive Postorder Traversal..... | 4-14 |
| 4.7.7 | Tree Traversal Examples | 4-15 |
| 4.8 | Basic Tree Operations..... | 4-18 |
| 4.8.1 | 'C' Function for Counting of Nodes in a Tree | 4-18 |
| 4.8.2 | 'C' Function for Counting of Leaf Nodes in a Tree (Recursive) | 4-18 |
| 4.8.3 | 'C' Function for Counting of Nodes of Degree 1 (Recursive) | 4-19 |
| 4.8.4 | 'C' Function for Counting of Nodes of Degree 2 (Recursive) | 4-19 |
| 4.8.5 | 'C' Function to Create an Exact Copy of a Tree (Recursive)..... | 4-19 |
| 4.8.6 | 'C' Function for Checking Equivalence of Two Binary Trees | 4-19 |
| 4.8.7 | 'C' Function for Finding Height of a Tree (Recursive)..... | 4-20 |
| 4.8.8 | 'C' Function for Swapping of Left and Right Children of Every Node (Mirror) | 4-20 |
| 4.8.9 | Finding Width of a Tree | 4-20 |
| 4.8.10 | Function to List the DATA Fields of the Node of a Binary Tree T by Level. Within Levels Nodes are Listed Left to Right..... | 4-21 |
| 4.8.11 | Non-Recursive Algorithm for Height of a Binary Tree..... | 4-22 |
| 4.8.11(A) | 'C' Function for Height of a Tree (Non-Recursive)..... | 4-22 |
| 4.9 | Creation of a Binary Tree from Traversal Sequence | 4-23 |
| 4.9.1 | Creation of Binary Tree from Preorder and Inorder Traversals | 4-23 |
| 4.9.2 | Creation of Tree from Postorder and Inorder Traversal | 4-23 |

| | | | | | |
|-----------|---|------|--------|--|-------|
| 4.9.3 | Examples on Tree Creation from Traversal Sequence | 4-23 | 4.13.2 | Representation of Binary Codes as a Binary Tree..... | 4-72 |
| 4.10 | Binary Search Tree (BST) | 4-27 | 4.13.3 | Huffman's Algorithm..... | 4-73 |
| 4.10.1 | Definition | 4-27 | 4.13.4 | Program for Huffman Tree | 4-77 |
| 4.10.2 | Operations on a Binary Search Tree | 4-27 | 4.14 | B-Trees | 4-78 |
| 4.10.2(A) | Initialize Operation..... | 4-27 | 4.14.1 | Insertion of a Key into a B-tree..... | 4-79 |
| 4.10.2(B) | Find Operation..... | 4-27 | 4.14.2 | Deleting a Value from a B-tree..... | 4-85 |
| 4.10.2(C) | Make Empty Operation..... | 4-28 | 4.14.3 | B-tree as an ADT..... | 4-89 |
| 4.10.2(D) | Insert Operation..... | 4-28 | 4.15 | B+ Trees..... | 4-95 |
| 4.10.2(E) | Example on Creation of a BST | 4-29 | 4.16 | Splay Tree..... | 4-97 |
| 4.10.2(F) | Delete Operation | 4-30 | 4.16.1 | Bottom up Splaying..... | 4-97 |
| 4.10.2(G) | Create..... | 4-32 | 4.16.2 | Top Down Splaying..... | 4-99 |
| 4.10.2(H) | Find Min..... | 4-32 | 4.17 | Trie Indexing | 4-101 |
| 4.10.2(I) | Find Max..... | 4-32 | 4.17.1 | Compact Trie..... | 4-102 |
| 4.10.3 | Program for Various Operations on BST | 4-32 | | | |
| 4.11 | AVL Trees | 4-36 | | | |
| 4.11.1 | Height Balanced Tree..... | 4-37 | | | |
| 4.11.2 | Balance Factor | 4-37 | | | |
| 4.11.3 | Structure of a Node in AVL Tree..... | 4-38 | | | |
| 4.11.4 | 'C' Function for Finding the Balance Factor of a Node..... | 4-38 | | | |
| 4.11.5 | Insertion of a Node into an AVL Tree | 4-38 | | | |
| 4.11.5(A) | Rotate Left..... | 4-39 | | | |
| 4.11.5(B) | Rotate Right | 4-40 | | | |
| 4.11.5(C) | Single Rotation and Double Rotation | 4-41 | | | |
| 4.11.5(D) | 'C' Function for Insertion of an Element into an AVL Tree | 4-64 | | | |
| 4.11.5(E) | 'C' Function to Find Height of AVL Tree | 4-64 | | | |
| 4.11.5(F) | 'C' Function to Rotate Right..... | 4-64 | | | |
| 4.11.5(G) | 'C' Function to Rotate Left..... | 4-65 | | | |
| 4.11.5(H) | 'C' Function for RR..... | 4-65 | | | |
| 4.11.5(I) | 'C' Function for LL | 4-65 | | | |
| 4.11.5(J) | 'C' Function for LR..... | 4-65 | | | |
| 4.11.5(K) | 'C' Function for RL..... | 4-65 | | | |
| 4.12 | Application of Trees..... | 4-65 | | | |
| 4.12.1 | Expression Trees..... | 4-65 | | | |
| 4.12.2 | Program on Expression Tree from Postfix Expression..... | 4-67 | | | |
| 4.12.3 | Conversion of an Expression into Binary Tree | 4-68 | | | |
| 4.12.4 | Construction of an Expression Tree from Infix Expression..... | 4-69 | | | |
| 4.13 | Huffman Algorithm..... | 4-71 | | | |
| 4.13.1 | Huffman Codes..... | 4-71 | | | |

Module 5**Chapter 5 : Graphs**

5-1 to 5-26

Syllabus : Introduction, Graph Terminologies, Representation of graph, Graph Traversals – Depth First Search (DFS) and Breadth First Search (BFS), Graph Application – Topological Sorting.

| | | |
|--------|--------------------------------------|------|
| 5.1 | Terminology and Representation | 5-1 |
| 5.1.1 | Definition | 5-1 |
| 5.1.2 | Undirected Graph..... | 5-1 |
| 5.1.3 | Directed Graph..... | 5-1 |
| 5.1.4 | A Complete Graph..... | 5-2 |
| 5.1.5 | Weighted Graph | 5-2 |
| 5.1.6 | Adjacent Nodes..... | 5-2 |
| 5.1.7 | Path..... | 5-2 |
| 5.1.8 | Cycle | 5-2 |
| 5.1.9 | Connected Graph..... | 5-2 |
| 5.1.10 | Subgraph..... | 5-3 |
| 5.1.11 | Component..... | 5-3 |
| 5.1.12 | Degree of a Vertex | 5-3 |
| 5.1.13 | Self Edges or Self Loops..... | 5-3 |
| 5.1.14 | Multigraph | 5-3 |
| 5.1.15 | Tree | 5-3 |
| 5.1.16 | Spanning Trees | 5-4 |
| 5.1.17 | Minimal Spanning Tree | 5-4 |
| 5.2 | Representation of Graphs | 5-4 |
| 5.2.1 | Adjacency Matrix..... | 5-4 |
| 5.2.2 | Adjacency List..... | 5-5 |
| 5.2.3 | Path Matrix | 5-11 |

| | | |
|----------|---|------|
| 5.3 | Traversal of Graphs..... | 5-12 |
| 5.3.1 | Depth First Search (DFS)..... | 5-12 |
| 5.3.1(A) | Algorithm for Depth First Search (Recursive) | 5-13 |
| 5.3.1(B) | Non-Recursive DFS Traversal..... | 5-15 |
| 5.3.2 | Breadth First Search(BFS) | 5-17 |
| 5.3.2(A) | Algorithm for BFS | 5-17 |
| 5.4 | Topological Sorting..... | 5-23 |
| 5.4.1 | Program for Topological Sorting..... | 5-23 |

Module 6

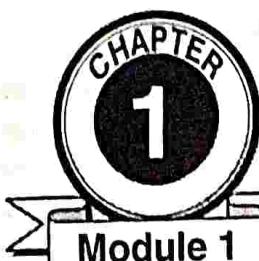
| | |
|-----------------------------------|-------------|
| Chapter 6 : Sorting and Searching | 6-1 to 6-49 |
|-----------------------------------|-------------|

Syllabus : Linear Search, Binary Search, Hashing-Concept, Hash Functions, Collision resolution Techniques.

| | | |
|----------|---|------|
| 6.1 | Searching | 6-1 |
| 6.2 | Sequential Search | 6-1 |
| 6.2.1 | Sequential Search on a Sorted Array | 6-2 |
| 6.3 | Binary Search..... | 6-3 |
| 6.4 | Sorting..... | 6-8 |
| 6.4.1 | Sort Stability | 6-8 |
| 6.4.2 | Sort Efficiency..... | 6-8 |
| 6.4.3 | Passes..... | 6-9 |
| 6.5 | Insertion Sort..... | 6-9 |
| 6.5.1 | Sorting an Array of Strings using Insertion Sort..... | 6-11 |
| 6.5.2 | Sorting an Array of Records on the given key using Insertion Sort | 6-12 |
| 6.6 | Bubble Sort..... | 6-13 |
| 6.7 | Selection Sort..... | 6-15 |
| 6.8 | Quick Sort..... | 6-16 |
| 6.8.1 | Picking a Pivot..... | 6-16 |
| 6.8.2 | Partitioning | 6-16 |
| 6.8.3 | Running Time of Quick Sort | 6-25 |
| 6.8.3(A) | Worst-Case Analysis | 6-25 |
| 6.8.3(B) | Best-Case Analysis | 6-25 |

| | | |
|-------------------------|---|------------|
| 6.8.3(C) | Average-Case Analysis..... | 6-25 |
| 6.8.4 | Role of Pivot in Efficiency of Quick Sort | 6-28 |
| 6.9 | Two-Way Merge Sort | 6-28 |
| 6.9.1 | Merging | 6-29 |
| 6.9.2 | Analysis of Merge Sort..... | 6-32 |
| 6.9.3 | Non-Recursive Merge Sort..... | 6-32 |
| 6.10 | Comparison of Sorting Algorithms..... | 6-33 |
| 6.11 | Best-Case, Worst-Case and Average-Case Analysis of Sorting Algorithm...6-34 | 6-34 |
| 6.12 | External Vs Internal Sorting..... | 6-34 |
| 6.13 | Hash Tables..... | 6-36 |
| 6.13.1 | What is Hashing ? | 6-36 |
| 6.13.2 | Hash Table Data Structure..... | 6-36 |
| 6.13.2(A) | Open Hashing Data Structure | 6-37 |
| 6.13.2(B) | Closed Hashing Data Structure..... | 6-37 |
| 6.13.3 | Hashing Functions | 6-37 |
| 6.13.3(A) | Characteristics of a Good Hash Function.... 6-37 | 6-37 |
| 6.13.3(B) | Division-Method | 6-38 |
| 6.13.3(C) | Midsquare Methods..... | 6-38 |
| 6.13.3(D) | Folding Method | 6-38 |
| 6.13.3(E) | Digit Analysis..... | 6-38 |
| 6.13.3(F) | Length Dependent Method..... | 6-38 |
| 6.13.3(G) | Algebraic Coding..... | 6-38 |
| 6.13.3(H) | Multiplicative Hashing | 6-39 |
| 6.13.4 | Collision Resolution Strategies (Synonym Resolution)..... | 6-39 |
| 6.13.4(A) | Separate Chaining | 6-39 |
| 6.13.4(B) | Open Addressing | 6-40 |
| 6.13.4(C) | Primary Clustering..... | 6-49 |
| • Lab Experiments | L-1 to L-33 | |
| • Appendix A : | Solved University Question Papers of Dec. 2017, May 2018 and Dec. 2018..... | A-1 to A-2 |
| • Appendix B : | Solved University Question Papers of May 2019 and Dec. 2019 | B-1 to B-2 |





Introduction to Data Structures

Syllabus

Introduction to Data Structures, Concept of ADT, Types of Data Structures-Linear and Nonlinear, Operations on Data Structures.

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** are non-decomposable 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.

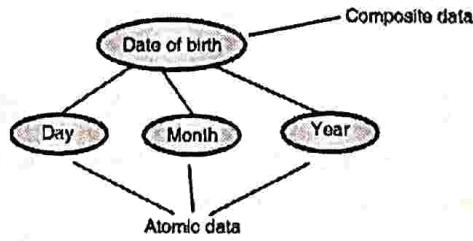


Fig. 1.1.1

- 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.

1.1.1 Data Types

- A data type is a term which refers to the kind of data that variables may hold in a programming language.
Example : 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 cannot 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.

1.1.2 Abstract Data Types (ADT)

MU - Dec. 13, May 16, Dec. 17

University Question

- Q. Define ADT with an example.

(Dec. 13, May 16, Dec. 17, 3 Marks)

- The concept of 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.
- When the program is hierarchical organized as shown in the Fig. 1.1.2, then the "main program" utilizes services at the functions appearing at level 1. Similarly, a function written at level 1 utilizes services of functions written at level 2. Main program uses the services of the next level function 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".

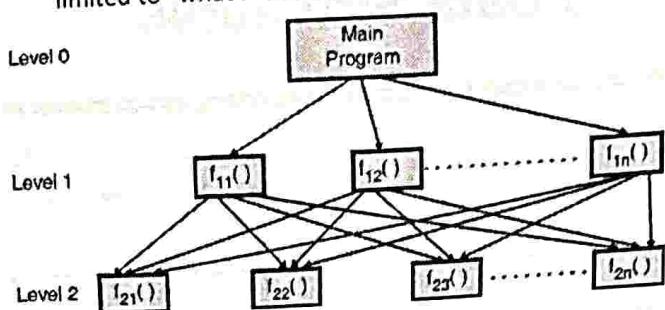


Fig. 1.1.2 : Hierarchical organized program

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 them :

- (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 "ARRAY"

- Arrays are stored in consecutive set of memory locations. An array can be thought of as a set of pair, index and value. For each index which is defined there is a value associated with that index. There are two operations permitted on 'ARRAY' data structure. These two operations are retrieve and store.

ADT ARRAY can be declared as below :

Structure ARRAY (value, index)

declare

CREATE() → array

RETRIEVE(array, index) → value

STORE(array, index, value) → array

- The function CREATE() produces an empty array. The function RETRIEVE() takes as input an array and an index, and either returns the appropriate value or an error.
- The function STORE() is used to enter new index-value pairs.

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, \dots, a_n

Where; n = Number of elements in the list

$a_i = i^{\text{th}}$ 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.

1.1.3 Data Object

- Data object refers to a set of variables used in a program.
- It is a place where the data values can be stored, retrieved and manipulated.
- Every data object is associated with data type (type of value that can be stored).
- A complex object (structure type) can have a number of attributes.
- Data object can also be viewed as a runtime instance of a data structure.

1.2 Data Structures

MU - May 15

University Question

Q. What is data structure ? (May 15, 2 Marks)

- A data structure is merely an instance of an ADT.
- An ADT or data structure is formally defined to be a triplet (D,F,A) where "D" stands for a set of Domains, "F" denotes the set of operations and "A" represents the axioms defining the functions in "F".
- An example of the data structure "Natural Number (NATNO)".

Structure of Natural Number (NATNO)

Operations

1. **ZERO() → natno**

2. **ISZERO(natno) → boolean**
3. **SUCC(natno) → natno**
4. **ADD(natno, natno) → natno**
5. **EQUAL(natno, natno) → boolean**

Axioms

For all $x, y \in \text{natno}$ let,
ISZERO(ZERO) is true.
ADD(ZERO,Y) is Y
EQUAL(x, ZERO), if **ISZERO(x)** then true else false.
D = {natno, boolean}
F = {ZERO, ISZERO, SUCC, ADD, EQUAL}
A = {Line no. 6 to 8 of the structure NATNO}

1.2.1 Types of Data Structures

MU - May 14, May 17, May 18

University Question

Q. Explain different types of data structures with example.
(May 14, May 17, May 18, 5 Marks)

1.2.1(A) Primitive and Non-Primitive

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 are Array, structure, union, linked-list, stack, queue, tree, and graph.
- Some of the most commonly used operations that can be performed on data structures are shown in Fig. 1.2.1.

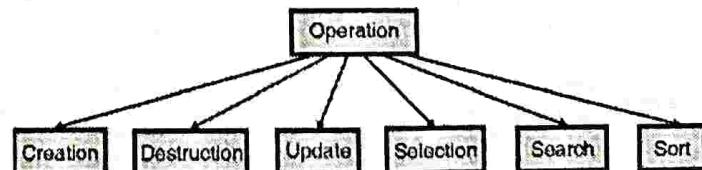


Fig. 1.2.1 : Data structure operations

**1.2.1(B) Linear and Non-Linear**

MU - Dec. 13, May 15, Dec. 16, Dec. 17,
May 19, Dec. 19

University Questions

Q. Explain linear and non-linear data structures with examples.

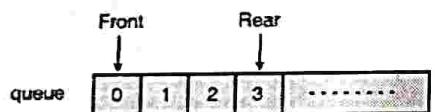
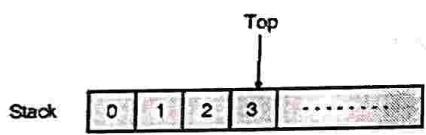
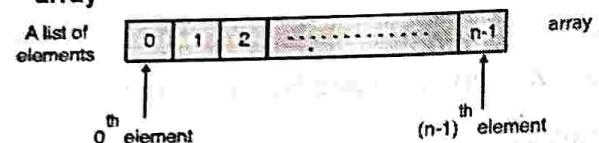
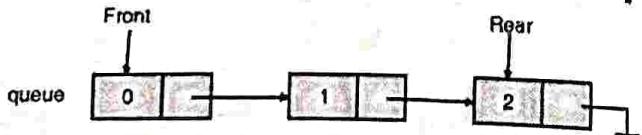
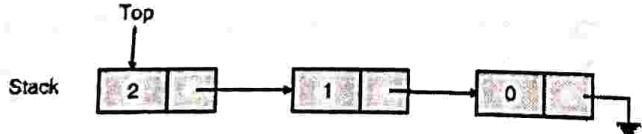
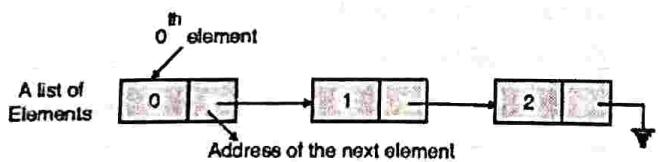
(Dec. 13, May 15, Dec. 16, Dec. 17, May 19, 3 Marks)

Q. Differentiate linear and non-linear data structures with example.

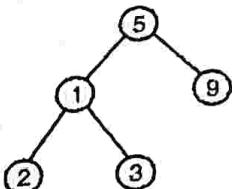
(Dec. 19, 4 Marks)

Linear

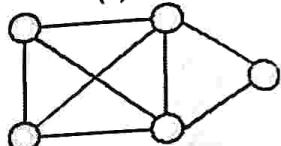
Elements are arranged in a Linear fashion (one dimension). All one-to-one relation can be handled through Linear data structures. Lists, stacks and queues are examples of linear data structure.

(I) Representation of Linear data structures in an array**(II) Representation of Linear data structures through Linked structure****Non-Linear**

All one to many, many to one or many to 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.



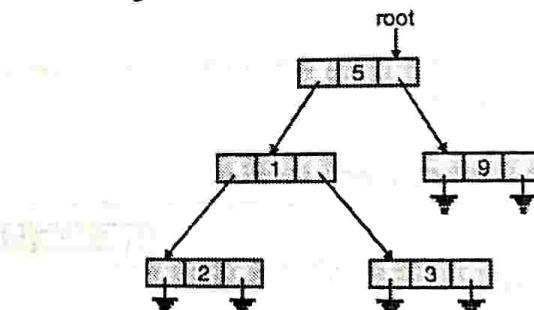
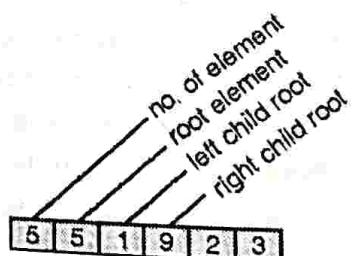
(a) Tree



(b) Graph

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

(c) Table

Fig. 1.2.2 : Non- Linear data structures**(a) Representation of the binary tree through linked structure****(b) Representation of the binary tree through an array****Fig. 1.2.3 : Representation of tree of Fig. 1.2.2(a)**

1.2.1(C) Static and Dynamic

Static Memory Allocation

In case of static data structure, memory for objects is allocated at the time of loading of the program. Amount of memory required is determined by the compiler during compilation.

Example :

```
int a[50];
```

Memory for the array 'a' of 50 element will be allocated at the time of loading of the program. It may not always be possible to fix the size of the array in advance. Amount of data to be handle is often determined by the user and not by the programmer. Our initial judgment of size, if wrong, may cause failure of program (due to overflow) or wastage of memory space.

- Static data structure causes under utilization of memory (in case of over allocation).
- Static data structure may cause overflow (under allocation).
- No reusability of allocated memory.
- Difficult to guess the exact size of data at the time of writing of program.

Dynamic

In case of dynamic data structures, the memory space required by variables is calculated and allocated during execution. Dynamic memory is managed in 'C' through a set of library function.

Allocating a block of memory in "C"

```
ptr = (cast_type) malloc(byte_size);
```

The "malloc()" returns a pointer of (cast type) to an area of memory with size, (byte_size)

Example :

```
x = (int *) malloc(100 * sizeof(int));
```

- A Linear data structure can be implemented either through static or dynamic data structures. Static data structured is preferred.
- All Linked structures are preferably implemented through dynamic data structures.
- Dynamic data structures provide flexibility in adding, deleting or rearranging data objects at run time.
- Additional space can be allocated at run time.
- Unwanted space can be released at run time.
- It gives re-usability of memory space.

1.3 Relationship among Data Object, Data Type, Data Structure and Data Representation

- A data type is a term which refers to the kind of data that variables may hold in a programming language. Every programming language has a method for declaring a set of variables of a particular type.
- A value stored in a variable cannot 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 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.
- Data object refers to a set of variables used in a program. Every data object is associated with data type. A complex object can have a number of attributes. Data object can also be viewed as a runtime instance of a data structure. Every type of data is converted into a stream of bits before it is stored in memory. Integers are represented as a signed binary numbers. Floating point numbers are represented in IEEE format.
- A data structure is merely an instance of an ADT. In most cases, a data structure is defined using structure and associated functions are written. A proper selection of data structure will determine the efficiency of the associated functions.

1.3.1 Operations on Data Structure

MU - Dec. 18

University Question

- Q. What are various operations possible on data structures? (Dec. 18, 5 Marks)**

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

- | | |
|---------------|--------------|
| 1. Traversing | 2. Searching |
| 3. Inserting | 4. Deleting |
| 5. Sorting | 6. Merging |

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.

1.4 Algorithm Analysis

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.
- Analysing 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 number 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 number of data to be added"); | 1 | t_2 |
| scanf("%d", &n) | 1 | t_3 |
| for($i = 1$; $i \leq n$; $i++$) | $n + 1$ | $(n + 1)t_4$ |
| 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, \dots, t_7 are computer dependent, on a faster computer, the execution time of an instruction will be less.
- (ii) $for(i = 1; i \leq n; i++)$, will be executed $n + 1$ times and not n times. For all values of i between 1 and n , the condition $i \leq n$ will evaluate to true and when i becomes $n+1$, the condition $i \leq n$ will evaluate to false. Thus the above instruction will be executed $n + 1$ times.

Example 1.4.1 : Determine the frequency count for all statements in the following program segment

1. $i = 1;$
2. $while(i \leq n)$
 - {
 3. $x = x + 1;$
 4. $i = i + 1;$

**Solution :**

| Statement No. | Frequency |
|---------------|-----------|
| 1 | 1 |
| 2 | $n + 1$ |
| 3 | n |
| 4 | n |

Example 1.4.2 : Determine the frequency counts for all statements in the following program segment.

1. `for(i = 1; i <= n; i++)`
2. `for(j = 1; j <= i; j++)`
3. `x = x + 1;`

Solution :

| | | Frequency | | |
|---------|-----------|---------------|-------|-----|
| | | Statement No. | | |
| | | 1 | 2 | 3 |
| $i = 1$ | $j = 1$ | 1 | 1 | 1 |
| | 2 | | 1 | 0 |
| $i = 2$ | $j = 1$ | 1 | 1 | 1 |
| | 2 | | 1 | 1 |
| | 3 | | 1 | 0 |
| | ... | 1 | 3 | 2 |
| $i = n$ | $j = 1$ | 1 | 1 | 1 |
| | 2 | | 1 | 1 |
| | $j = n$ | | 1 | 1 |
| | $j = n+1$ | | 1 | 0 |
| | | 1 | $n+1$ | n |

Fig. Ex. 1.4.2 : Program Trace**Frequency**

$$\text{Statement No. 1} = n + 1$$

$$\begin{aligned}\text{Statement No. 2} &= 2 + 3 + \dots + n + (n+1) \\ &= (1 + 2 + 3 + \dots + n) + n \\ &= n \frac{(n+1)}{2} + n = \frac{1}{2} (n^2 + 3n)\end{aligned}$$

$$\begin{aligned}\text{Statement No. 3} &= 1 + 2 + \dots + n = n \frac{(n+1)}{2} \\ &= \frac{1}{2} (n^2 + n)\end{aligned}$$

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

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.

1.4.2 Measurement of Growth Rate (Asymptotic Growth Rate)

1.4.2(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.

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

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 $n \geq 20$

1.4.2(B) Constant Factor In Complexity Measure

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

$$\begin{array}{ll} f_1(n) = 5n^2 & f_2(n) = 100n^2 \\ f_3(n) = 1000n^2 & f_4(n) = n^2 \end{array}$$
- Time complexity of $f_1(n)$ = Time complexity of $f_2(n)$
 $=$ Time complexity of $f_3(n)$ = Time complexity of $f_4(n)$

1.4.3 Notation O : (Pronounced as Big-Oh), ($O(n^2)$ is Pronounced as Big-Oh of n^2)

- 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 1.4.3 : $f(x) = x^2 + 5x$, $g(x) = x^2$ let us take $C = 2$

Solution :

| x | $x^2 + 5x$ | $2x^2$ |
|---|------------|--------|
| 1 | 5 | 2 |
| 2 | 14 | 8 |
| 5 | 50 | 50 |

$f(x) \leq Cx^2$
 $\text{For } x \geq 5$
 (asymptotic behavior)

$f(x) \leq Cg(x)$ for all $x \geq K$ where $C = 2$ and $K = 5$

Example 1.4.4 : For the function defined by $f(x) = 5x^3 + 6x^2 + 1$ show that $f(x) = O(x^3)$

Solution :

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

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

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

there exist $C = 12$ and $K = 1$ such that

$$f(x) \leq Cx^3 \text{ for all } x \geq K$$

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 1.4.5 : Write a timing complexity of $x = x + 1$.

Solution :

A program fragment without a loop

$$x = x + 1$$

The timing complexity is of constant order and it is represented by $O(1)$.

Example 1.4.6 : Calculate the running time of a given program

$$1 : \text{for}(i = 1; i \leq n; i++)$$

$$2 : x = x + 1;$$

Solution :

Step no. 1 will be executed $n + 1$ times whereas the step no. 2 will be executed n time. With the assumption that each step requires a constant time, the time required to execute the above program will be of the order of $2n + 1 = O(n)$.

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

$$f(n) \leq Cn \text{ for } C = 3 \text{ and}$$

$$n \geq 1$$

Conclusion :

A loop without nesting has a timing complexity $O(n)$.

Example 1.4.7 : Calculate the running time of a given program

$$\text{for}(i = 1; i \leq n; i++)$$

$$\text{for}(j = 1; j \leq n; j++)$$

$$x = x + 1;$$

**Solution :**

Since, the statement $x = x + 1$ will be executed n^2 times, the timing complexity of the above program segment is $O(n^2)$.

Example 1.4.8 : A large program

```

n = 500 ; } O(1) Timing Complexity
x = 1 ; } O(1)
y = 2 ; } O(1)
for(i = 1 ; i <= n ; i++)
    x = x + 1; } O(n)
for(j = 1 ; j <= n ; j++)
    y = y + 1; } O(n)
for(i = 1 ; i <= n ; i++)
    for(j = 1 ; j <= n ; j++)
        x = y + 1; } O(n^2)
    
```

Hence, the combined time complexity is;

$$O(\max(O(1), O(1), O(1), O(n), O(n), O(n^2))) = O(n^2)$$

$$\text{if } f(x) = 5x^3 + 6x^2 + 1$$

$$O(f(x)) = x^3 - \text{proved earlier}$$

Conclusion :

If $T_1(n)$ and $T_2(n)$ are the running times of two program fragments P_1 and P_2 and that $T_1(n)$ is $O(f(n))$ and $T_2(n)$ is $O(g(n))$. Then the running time of the complete program P_1 followed by P_2 is $O(\max(f(n), g(n)))$. For finding the timing complexity of a program, consider the program fragment having the highest timing complexity.

Example 1.4.9 : Calculate the time complexity for the following program segment

```

for(i = 1; i < n; i++)
    for(j = 0; j < n - i; j++)
        if(a[i] > a[j + 1])
            {
                temp = a[i];
                a[i] = a[j + 1];
                a[j + 1] = temp; } Statements
            for interchange
            of a[i] and a[j + 1]
        }
    
```

Solution :

For calculation of timing complexity, it is sufficient to express number of interchanges in terms of n (number of data).

| Program trace | Number of interchanges |
|--|------------------------|
| i = 1, j varies from 0 to $n - 2$ | $n - 1$ |
| i = 2, j varies from 0 to $n - 3$ | $n - 2$ |
| . | . |
| . | . |
| i = $n - 1$, j varies from $n - 1$ to $n - 1$ | 1 |

Total number of interchanges

$$\begin{aligned}
 &= 1 + 2 + \dots + (n - 2) + (n - 1) \\
 &= \frac{n(n - 1)}{2} = \frac{1}{2}(n^2 - n) = O(n^2)
 \end{aligned}$$

[Consider, the most significant term]

Example 1.4.10 : Calculate the time complexity for the following program segment

```

int A(int n)
{
    if(n <= 1)
        return(1);
    else
        return(A(n - 1) + A(n - 1));
}

```

Solution :

Recursion tree for $n = 4$

When the function A is called with the initial value of $n = 4$, it calls the same function recursively twice with $n = 3$ thus,

$$\begin{aligned}
 f(n) &= 2f(n - 1) && \text{if } n > 1 \\
 f(n) &= 1 && \text{if } n \leq 1 \\
 &&& (\text{Best case})
 \end{aligned}$$

$$\begin{aligned}
 \text{Since } f(n) &= 2f(n - 1) \\
 &= 2^2f(n - 2) = 2^{n-1}f(n - (n - 1)) \\
 &= 2^{n-1}f(1) = 2^{n-1} = O(2^n)
 \end{aligned}$$

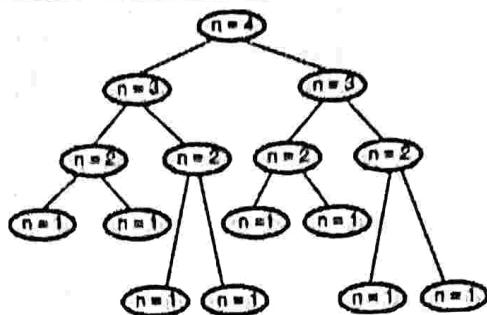


Fig. Ex. 1.4.10

1.4.4 Best Case, Worst Case and the Average Case Behaviour

Many programs do not produce same timing complexity in every case. Consider the problem of searching an element in an array with 10 elements.

| | | | | | | | | | |
|---|---|---|---|---|----|----|---|---|----|
| 5 | 1 | 9 | 6 | 2 | 11 | 13 | 6 | 7 | 16 |
|---|---|---|---|---|----|----|---|---|----|

Element 5 will be found in one attempt

Element 16 will require 10 comparisons to locate

Best case behaviour = 1 comparison (when the element to be searched is in the beginning)

Worst case behaviour = n comparisons (where n is the number of elements and the element to be searched is at the end of the array)

Average case behaviour = number of comparisons will be $n/2$ (from probability).

Example 1.4.11 : Calculate the worst case time complexity of the following program (Function for binary search) :

```

int binsearch(int a[], int i, int j, int x)
{
    int k;
    k = (i + j)/2;
    while(i <= j)
    {
        if(x > a[k])
            i = k + 1;
        else
            if(x < a[k])
                j = k - 1;
            else
                return(k);
    }
    k = (i + j)/2;
    return(-1);
}
  
```

Solution :

- Binary search requires that elements are sorted in ascending order.
- i is the starting index (i.e. 0), j is the index of the last element ($n - 1$).
- k gives the index of the centre element.

| | | | | | | | |
|---------|---|---------|---|---------------------|----|----|-----------|
| 2 | 5 | 9 | 8 | 11 | 13 | 15 | ← Element |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | ← Index |
| i | | k | | j | | | |
| ↑ | | ↑ | | ↑ | | | |
| $i = 0$ | | $j = 6$ | | $k = (0 + 6)/2 = 3$ | | | |

Element to be searched is x .

- Inside the while-loop, if the element x is found at the location k (centre), we return k .
- Inside the while-loop, if the element x is larger than the centre element; we select the second half of the array.
- Inside the while-loop, if the element x is smaller than the centre element; we select the first half of the array.

| | | | | | | |
|----------------------------------|------------|--------------|--------------------------------|-------------|----|----|
| 2 | 5 | 9 | 8 | 11 | 13 | 15 |
| ↑ $i = 0$ | First half | | ↑ Centre element $k = 3$ | Second half | | |
| Element to be searched, $x = 13$ | | | | | | |
| 11 | 13 | 15 | | | | |
| ↑ $i = 4$ | | ↑ $j = 6$ | | | | |

} Second half of the element

- In order to calculate time complexity, binary search for n elements must be expressed in terms of binary search of fewer numbers ($< n$) of elements.

$$f(n) = 1 + f(n/2)$$
- If there are 128 elements, then the maximum number of elements between i and j after every iteration will be 64, 32, 16, 8, 4, 2, 1, 0, -1

$$128 = 2^7$$



Number of iteration required (say h) = 9 = 7 + 2

$$\therefore 2^{h+2} = n$$

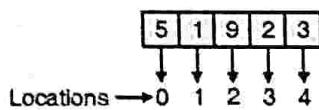
$$\text{or } \log(2^{h+2}) = \log n$$

$$\text{or } h+2 = \log n$$

$$\therefore h = \log n - 2 = O(\log n)$$

1.5 Introduction to Arrays

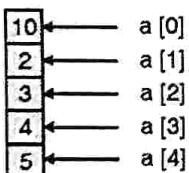
- Array is a collection of elements.
- All elements are of the same type.
- All elements are stored using contiguous memory.
- `int a[5]`, declares an integer array of five elements.
- Elements of an array can be initialised at the time of declaration.



`int a[5] = {10, 2, 3, 4, 5};`

- it is also possible to initialize individual elements as follows :

```
a[0] = 10;
a[1] = 2;
a[2] = 3;
a[3] = 4;
a[4] = 5;
```



- A particular value is accessed by writing a number called index number or subscript in brackets after the array name.

For example,

`a[3];`
/*represents the third element of the array,*/

- Index starts from 0 and goes upto n - 1, where n is the size of the array.

`int a[5];` /*an array of 5 elements*/

- In an array, element `a[i]` can be manipulated like a normal variable.

`a[5] = 0;`

`a[5] = a[4] + 2;`

1.6 Representation and Analysis

- Arrays must be declared before they are used. The general form of array declaration is

`type variable_name[size];`

- The type specifies the type of the element to be stored in the array. Type could be any of the primitive types like int, float, char, long, double etc. or user defined type.

`int number[10];`

- Declares an array of 10 integer constant with the name number. Any subscript from 0 to 9 is valid.

`float marks[15];`

- Declares marks to be an array containing 15 real numbers.

Note: C-language does not do any range checking on array subscript. Any reference to the array outside the declared limits may result in unpredictable program results.

`char name[20];`

- Declares the name as a character array that can hold a maximum of 20 characters. It may be noted that strings are stored in character array and a string is terminated by a null character ('\0').

"INDIA"

Each character of the string is treated as an array element and is stored in the memory as given below

| | | |
|------|---------|---|
| 'I' | name[0] | name[i] represents individual element of the string, whereas name without any subscript indicates the entire string |
| 'N' | name[1] | |
| 'D' | name[2] | |
| 'I' | name[3] | |
| 'A' | name[4] | |
| '\0' | name[5] | |

Array elements are stored in successive memory location.



Program 1.6.1 : Program to display array elements with their addresses.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a[3]={5, 4, 3};
    printf("\n a[0] , value=%d : address=%u", a[0],
    &a[0]);
    printf("\n a[1] , value=%d : address=%u", a[1],
    &a[1]);
    printf("\n a[2] , value=%d : address=%u", a[2],
    &a[2]);
    getch();
}
```

Output

```
a[0] , value=5 : address=65520
a[1] , value=4 : address=65522
a[2] , value=3 : address=65524
```

An integer requires 2 bytes of memory locations.
Addresses displayed have a difference of 2.

1.7 One-Dimensional Arrays

One-dimensional array is a list of elements or simply a row of elements. In mathematics, we often deal with variables that are simple scripted. For instance,

$$\sum_{i=0}^n x_i$$

x_i , refers to the i^{th} element of x . These variables can be expressed as $x[0], x[i], \dots x[n - 1]$

```
sum = 0;
for(i = 0; i < n; i++) /* loop for traversing */
sum = sum + x[i];
```

Elements of an array can be scanned from left to right, element by element through the use of a for-loop. Starting index is 0 and it goes upto $n - 1$ in step of 1.

Program 1.7.1 : Program to read and display the elements of an array along with the total, number of even and odd numbers.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int i, a[50], sum, n, even, odd;
    printf("\n Enter no. of elements :");
    scanf("%d", &n); // Reading values into Array
    printf("\n Enter the values :");
    for(i=0; i<n; i++)
        scanf("%d", &a[i]); /* computation of total */
    sum=0; even = odd = 0;
    for(i=0; i<n; i++)
    {
        sum = sum+a[i];
        if(a[i]%2==0)
            even++;
        else
            odd++;
    }
    /* printing of all elements of array */
    for(i=0; i<n; i++)
        printf("\n a[%d]=%d", i, a[i]);
    /* printing of total */
    printf("\n sum=%d\n even=%d\n odd=%d", sum,
    even, odd);
}
```

Output

```
Enter no of elements : 3
```

```
Enter the values : 1 2 3
```

```
a[0]=1
```

```
a[1]=2
```

```
a[2]=3
```

```
sum=6
```

```
Even=1
```

```
Odd=2
```

Array $a[50]$ can store up to maximum of 50 elements.
User can store any number of elements by reading and storing a value in the variable n .

Variable i is used to scan the array from the location 0 to $n - 1$. Sum of the elements of an array can be represented using the conventional mathematical notation.

 $n - 1$

$\sum_{i=0}^{n-1} x_i$ and same can be converted to an equivalent
for loop

```
sum = 0;
for(i = 0; i < n; i++)
    sum = sum + x[i];
```

Example 1.7.1 : Write a user defined function in 'C' to find the geometric average of n integers using the given formula :

$$G(\text{Avg}) = (x_1 * x_2 * x_3 \dots x_n)^{1/n}$$

Solution :

```
float Gmean(float a[], int n)
{
    //array contains x1...xn
    int i;
    float gavg=1;
    for(i=0; i < n; i++)
        gavg=gavg * a[i];
    gavg=Pow(gavg, 1.0/n);
    return(gavg);
}
```

Example 1.7.2 : Write output of the following program with explanation.

```
void main()
{
    int a[10];
    int i;
    a[0] = 0;
    a[1] = 1;
    for(i = 2; i < 10; i++)
        a[i] = a[i - 1] + a[i - 2];
    for(i = 0; i < 10; i++)
        printf("%d\n", a[i]);
}
```

Solution :

In the for loop

```
for(i = 2; i < 10; i++)
    a[i] = a[i - 1] + a[i - 2];
```

$a[i]$ is set to sum of the previous two elements for every i from 2 to 9.

$a[2] = a[1] + a[0] = 0 + 1 = 1$
 $a[3] = a[2] + a[1] = 1 + 1 = 2$
 $a[4] = a[3] + a[2] = 1 + 2 = 3$
 $a[5] = a[4] + a[3] = 2 + 3 = 5$
 $a[6] = a[5] + a[4] = 3 + 5 = 8$
 $a[7] = a[6] + a[5] = 5 + 8 = 13$
 $a[8] = a[7] + a[6] = 8 + 13 = 21$
 $a[9] = a[8] + a[7] = 13 + 21 = 34$

The output of the program will be

0
1
2
3
5
8
13
21
34

1.8 Operations with Arrays

Some of the frequently used operations with arrays are shown in Fig. 1.8.1.

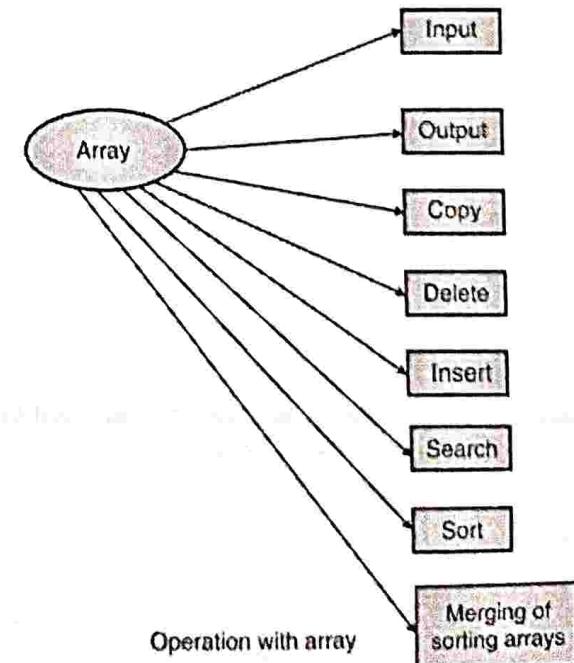
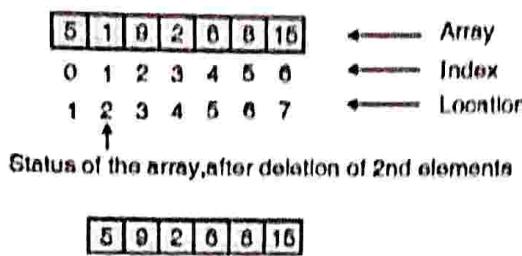


Fig. 1.8.1 : Operations with arrays



1.8.1 Deletion

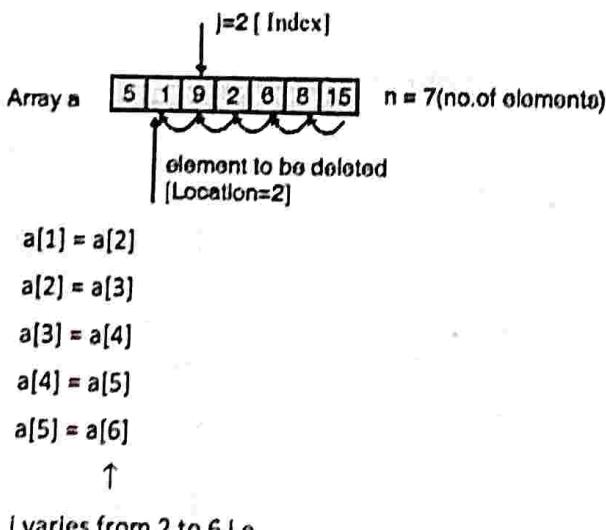
Deletion involves deleting of the specified element from the array.



In order to delete the second element, all elements ahead of second location should be moved left by one location.

```
/* delete the element from the location j */
index of the jth element = j - 1
while(j < n)
{
    a[j - 1] = a[j];
    j++;
}
n = n - 1;
```

loop for shifting of elements left by one location



j varies from 2 to 6 i.e.

Value of n becomes 6 after deletion

Program 1.8.1 : Program for deletion of an element from the specified location.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a[30], n, i, j;
    /* n : no. of elements stored in array
```

I : for scanning the array
j : location of the element to be deleted
*/
printf("\n Enter no. of elements :");
scanf("%d", &n);
/* read n elements in an array */
printf("\n Enter %d elements : ", n);
for(i=0; i<n; i++)
scanf("%d", &a[i]);
// read the location of the element to be deleted
printf("\n location of the element to be deleted : ");
scanf("%d", &j); /* loop for the deletion */
while(j<n)
{
 a[j-1]=a[j];
 j++;
}
n--; /* no. of elements reduced by 1 */
/* loop for printing */
for(i=0; i<n; i++)
printf("\n %d", a[i]);
getch();
}

Output

Enter no. of elements : 4

Enter 4 elements : 23 45 67 12

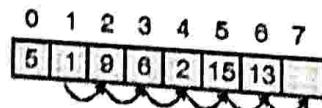
location of the element to be deleted : 3

23 45 12

1.8.2 Insertion

This operation can be used to insert an arbitrary element at specified location in an array.

array a 5 1 9 6 2 15 13 n = 7 Place of insertion = 2
before insertion



Creation of space at location 2 Index of location 2 is 1 i.e. loc-1

Movement of data to its right starts from last location(Index n-1)



```

for(i = n - 1; i >= loc-1; i--)
    a[i + 1] = a[i];

```

| | | | | | | |
|---|---|---|---|---|----|----|
| 5 | 1 | 9 | 6 | 2 | 15 | 13 |
|---|---|---|---|---|----|----|

Space is created at location 2 (index 1) Data to be inserted = 20
array [5 20 1 9 6 2 15 13] n = 8

after insertion

Program 1.8.2 : Program for Insertion of an element at the specified location.

```

#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

Program 1.8.3 : Program to copy all elements of an array into another array.

```

#include<stdio.h>
#include<conio.h>
void main()
{
    int a[30], b[30], i, n;
    /*
        Element of an array 'a' will be copied into array'b'
        i -> for scanning of an array
        x -> no. of elements in the array
    */
    printf("\n Enter no. of elements :");
    scanf("%d", &n);
    /* Reading values into Array */
    printf("\n Enter the values :");
    for(i=0; i<n; i++)
        scanf("%d", &a[i]);
    /* Copying data from array 'a' to array 'b'*/
    for(i=0; i<n; i++)
        b[i]=a[i];
    /* printing of all elements of array */
    printf("the copied array is :");
    for(i=0; i<n; i++)
        printf("b[%d]=%d \t", i, b[i]);
    getch();
}

```

Output

Enter no. of elements : 5

Enter the values : 12 22 32 42 52

the copied array is : b[0]=12 b[1]=22 b[2]=32
b[3]=42 b[4]=52



1.8.3 Search

The process of finding whether the specified element is an array is known as searching.

Program 1.8.4 : Program to search for an elements In an array.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a[30], x, n, i;
    /*
        a : for storing of data
        x : element to be searched
        n : no. of elements in the array
        i : scanning of the array
    */
    printf("\n Enter no. of elements :");
    scanf("%d", &n);
    /* Reading values into Array */
    printf("\n Enter the values :");
    for(i=0; i<n; i++)
        scanf("%d", &a[i]);
    // read the element to be searched
    printf("\n Enter the elements to be searched");
    scanf("%d", &x);
    /* search the element */
    i=0; // search starts from the location zero
    while(i<n && x!=a[i])
        i++;
    /* search until the element is not found i.e.x!=a[i]
    */
    /* search until the element could still be found i.e.
    i<n */
    if(i<n) /* Element is found */
        printf("found at the location =%d", i+1);
    else
        printf("\n not found");
    getch();
}
```

```
else
    printf("\n not found");
getch();
}
```

Output

Enter no. of elements : 3

Enter the values : 55 56 57

Enter the elements to be searched 56

found at the location =2

- In the above program, n elements are stored in the array a. Element to be searched x is searched by scanning the array from left to right using the loop.

```
i = 0;
while(i < n && x!= a[i])
    i++;
```

- The condition $i < n$ will fail if the element to be searched does not exist in the array. The second condition $x \neq a[i]$ will fail when the element x is found in the array. After termination of the loop, one can check the value of i, if the value of $i < n$ then the condition $x \neq a[i]$ has failed and the element x has been found in the array.

1.8.4 Merging of Sorted Arrays

Merging of two sorted array a and b is carried out in such way that the resultant array c is sorted after merging.

Array a [1 5 9 11 13] $n = 5$

Array b [2 4 5 6] $n = 4$

Merged array c [1 2 4 5 5 6 9 11 13] $n = 9$

Algorithm

Variable i : To scan the array a

Variable j : To scan the array b

Variable k : Number of elements copied into c



Initial conditions :

array a[1 5 9 11 13]

↑

i = 0

array b[2 4 5 6]

↑

j = 0

array c[]

↑

k = 0

Since the i^{th} element of the array a is smaller than the j^{th} element of the array b, it is copied into c. i and k are incremented by 1.

a[1 5 9 11 13]

↑

i = 1

array b[2 4 5 6]

↑

j = 0

array c[1]

↑

k = 1

Since

b[j] < a[i]

{ c[k] = b[j], j++, k++ }

a[1 5 9 11 13]

↑

i = 1

array b[2 4 5 6]

↑

array c[1 2]

↑

Since

b[j] < a[i]

{ c[k] = b[j], j++, k++ }

a[1 5 9 11 13]

↑

i = 1

array b[2 4 5 6]

↑

array c[1 2 4]

↑

Since

a[i] < b[j]

{ c[k] = a[i], i++, k++ }

a[1 5 9 11 13]

↑

i = 2

array b[2 4 5 6]

↑

array c[1 2 4 5]

↑

Since

b[j] < a[i]

{ c[k] = b[j], j++, i++ }

a[1 5 9 11 13]

↑

i = 2

array b[2 4 5 6]

↑

array c[1 2 4 5 5]

↑

since

b[j] < a[i]

{ c[k] = b[j], j++, k++ }

a[1 5 9 11 13]

↑

i = 2

array b[2 4 5 6]

↑

array c[1 2 4 5 5 6]

↑

j = 4

k = 6

Since, array b[] is exhausted, remaining elements of the array a[] are copied at the end of array c.

a[1 5 9 11 13]

↑

i = 5

b[2 4 5 6]

↑

j = 4

c[1 2 4 5 5 6 9 11 13]

↑

k = 9

**Program 1.8.5 : Program for merging of two array.**

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a[30], b[30], c[30], i, j, k, n1, n2;
    /*
        n1 : no. of elements in the array 'a'
        n2 : no. of elements in the array 'b'
    */
    printf("\n Enter no. of elements in 1'st array :");
    scanf("%d", &n1);
    for(i=0; i<n1; i++)
        scanf("%d", &a[i]);
    printf("\n Enter no. of elements in 2'nd array :");
    scanf("%d", &n2);
    for(i=0; i<n2; i++)
        scanf("%d", &b[i]);
    i=0; j=0; k=0; /* merging starts */
    while(i<n1 && j<n2)
    {
        if(a[i]<=b[j])
        {
            c[k]=a[i];
            i++; k++;
        }
        else
        {
            c[k]=b[j];
            k++; j++;
        }
    }
    /* Some statements in array a are still remaining
       whereas the array b is exhausted */
    while(i<n1)
    {
        c[k]=a[i];
        i++; k++;
    }
    /* some elements in array b are still remaining
       whereas the array 'a' is exhausted */
    while(j<n2)
    {
        c[k]=b[j];
        k++; j++;
    }
}
```

```
}
/* Displaying elements of array 'c' */
printf("\n Merged array is :");
for(i=0; i<n1+n2; i++)
    printf("\n %d", c[i]);
}
```

Output

```
Enter no. of elements in 1'st array : 5 1 5 9 11 13
Enter no. of elements in 2'nd array : 4 2 4 5 6
Merged array is : 1 2 4 5 5 6 9 11 13
```

1.9 Two-Dimensional Arrays

- Two-dimensional arrays can be thought of as a table consisting of rows and columns.
- int a[3][4], declares an integer array of three rows and four columns. Row index starts from 0 and will go upto 2.
- Similarly, column index will start from 0 and will go upto 3.

Table 1.9.1 : Arrangement of elements and their indices

| Row | Column 0 | Column 1 | Column 2 | Column 3 |
|-------|----------|----------|----------|----------|
| Row 0 | a[0][0] | a[0][1] | a[0][2] | a[0][3] |
| Row 1 | a[1][0] | a[1][1] | a[1][2] | a[1][3] |
| Row 2 | a[2][0] | a[2][1] | a[2][2] | a[2][3] |

Program 1.9.1 : Program to show the use of two dimensional array.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a[10][10], m, n, i, j;
    /* physical size of the array is 10*10
       whereas a specified size of the array
       of the size m*n can be stored in array a */
    /*
        m : number of rows
        n : number of columns
    */
    printf("\n Enter the rows and columns :");
```



```

scanf("%d %d", &m, &n);
/* Reading m*n elements */
for(i=0; i<m; i++) /* m rows */
for(j=0; j<n; j++)
/* inside each row there are n columns */
{
    printf("\n Enter the value of(%d)(%d) = ", i, j);
    scanf("%d", &a[i][j]);
}
/* printing of all elements of 2-D array */
for(i=0; i<m; i++)
{
    printf("\n ");
    /* each row starts from newline */
    for(j=0; j<n; j++)
        printf("%d\t", a[i][j]);
}
getch();
}

```

Output

```

Enter the rows and columns : 3 4
Enter the value of(0)(0)=2
Enter the value of(0)(1)=1
Enter the value of(0)(2)=0
Enter the value of(0)(3)=5
Enter the value of(1)(0)=9
Enter the value of(1)(1)=3
Enter the value of(1)(2)=6
Enter the value of(1)(3)=1
Enter the value of(2)(0)=4
Enter the value of(2)(1)=6
Enter the value of(2)(2)=7
Enter the value of(2)(3)=8

```

| | | | |
|---|---|---|---|
| 2 | 1 | 0 | 5 |
| 9 | 3 | 6 | 1 |
| 4 | 6 | 7 | 8 |

- A nested loop is required to scan all elements of a two dimensional array. Outer loop is for rows and the inner loop is for columns.

```

for(i = 0; i < m; i++) /* m rows */
for(j = 0; j < n; j++) /* n columns */

```

- Above nested loop is required for all simple operations like reading, printing, addition of two arrays.

1.9.1 Initializing Two-Dimensional Arrays

- Similar to one dimensional arrays, two-dimensional arrays may be initialized by providing a list of initial values.

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

- Above statement can be equivalently written as

```
int a[2][3] = {{1, 1, 1}, {2, 2, 2}};
```

1.9.2 Address Calculation

- An $m \times n$ matrix ($a[1..m][1..n]$) where the row index varies from 1 to m and column index varies from 1 to n can be written as (please note that in C language the index will start from 0 and go upto $m - 1$).

$$a = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}$$

- a_{ij} denotes the entry in the i^{th} row and the j^{th} column. In computer memory, all elements are stored linearly using contiguous addresses. Hence, in order to store a two dimensional matrix a two dimensional address space must be mapped to one dimensional (linear) address space.
- In the computer's memory matrices are stored in either **row major form** or **column major form**.

Row major form

Elements are stored row wise i.e. row 1, row 2, ... row m .

Column major form

Elements are stored column wise i.e. column 1, column 2, ... column n .



Mapping of a two dimensional (3×4) array

| | | | | | | | | | | | |
|--------|--------|--------|--------|---|---|----|----|---|---|---|----|
| [0][0] | [0][1] | [0][2] | [0][3] | 0 | 1 | 2 | 3 | 0 | 3 | 6 | 9 |
| [1][0] | [1][1] | [1][2] | [1][3] | 4 | 5 | 6 | 7 | 1 | 4 | 7 | 10 |
| [2][0] | [2][1] | [2][2] | [2][3] | 8 | 9 | 10 | 11 | 2 | 5 | 8 | 11 |

Arrangement of Indices

Row major mapping

Column major mapping

[C uses row-major scheme and FORTRAN uses column major scheme]

To locate the elements a_{ij} in a 2-dimensional array $a[L_1 \dots U_1][L_2 \dots U_2]$ where L_1 and U_1 are the lower and upper bounds respectively for row and L_2 and U_2 are the lower and upper bounds respectively for columns.

| | | | | |
|-------------------|-----------------------|-----------------------|---------|-------------------|
| $a_{L_1 L_2}$ | $a_{L_1 L_2 + 1}$ | $a_{L_1 L_2 + 2}$ | \dots | $a_{L_1 U_2}$ |
| $a_{L_1 + 1 L_2}$ | $a_{L_1 + 1 L_2 + 1}$ | $a_{L_1 + 1 L_2 + 2}$ | \dots | $a_{L_1 + 1 U_2}$ |
| : | | | | |
| $a_{U_1 L_2}$ | $a_{U_1 L_2 + 1}$ | $a_{U_1 L_2 + 2}$ | \dots | $a_{U_1 U_2}$ |

Arrangement of indices for the array $a[L_1 \dots U_1][L_2 \dots U_2]$ (A) Location of an element a_{ij} (row major form) = $B + (i - L_1) \times (U_2 - L_2 + 1) \times S + (j - L_2) \times S$

B : Base address

U2 : $L_2 + 1$ – Number of columns, i.e. number of elements in each row

S : Number of bytes taken to store each element (an integer requires 2 bytes of memory)

$(i - L_1) \times (U_2 - L_2 + 1) \times S$ – i^{th} row will start after $i - L_1$ rows, each row has $U_2 - L_2 + 1$ elements and the size of each element is S bytes.

Application of above formula for an array declared in C-program.

int a[3][4]

Let the array is stored in memory at a base location 100. Let each element be stored in memory using $S = 2$ bytes.

| | | | |
|-----------------------|-----|-----------------------|--------|
| Lower bound $L_1 = 0$ | row | Lower bound $L_2 = 0$ | column |
| Upper bound $U_1 = 2$ | | Upper bound $U_2 = 3$ | |

location of $[2][1] = 100 + (2 - 0) \times (3 - 0 + 1) \times 2 + (1 - 0) \times 2$

$$= 100 + 16 + 2 = 118$$

| Elements | a_{00} | a_{01} | a_{02} | a_{03} | a_{10} | a_{11} | a_{12} | a_{13} | a_{20} | a_{21} | a_{22} | a_{23} |
|-----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|
| Locations | 100 | 102 | 104 | 106 | 108 | 110 | 112 | 114 | 116 | 118 | 120 | 122 |

Elements stored in contiguous memory locations in row major form.

(B) Location of an element a_{ij} (column major form) = $B + (j - L_2) \times (U_1 - L_1 + 1) \times S + (i - L_1) \times S$ location of $[2][1] = 100 + (1 - 0) \times (2 - 0 + 1) \times 2 + (2 - 0) \times 2$

$$= 100 + 6 + 4 = 110$$

| Elements | a_{00} | a_{10} | a_{20} | a_{01} | a_{11} | a_{21} | a_{02} | a_{12} | a_{22} | a_{03} | a_{13} | a_{23} |
|-----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|
| Locations | 100 | 102 | 104 | 106 | 108 | 110 | 112 | 114 | 116 | 118 | 120 | 122 |



Example 1.9.1 : Each element of an array Data [20][50] requires 4 bytes of storage. Base address of Data is 2000. Determine the location of Data [10][10] when the array is stored as :
 (i) Row major (ii) Column major.

Solution :

- (i) Location of an element a_{ij} (row major form)
- $$= B + (i - L_1) \times (U_2 - L_2 + 1) \times S + (j - L_2) \times S$$
- Here $L_1 = 0$, $U_1 = 19$, $L_2 = 0$, $U_2 = 49$ and $B = 2000$
- $$\text{Required location} = 2000 + (10 - 0) \times (49 - 0 + 1) \times 4 + (10 - 0) \times 4$$
- $$= 2000 + 2000 + 40 = 4040$$
- (ii) Location of an element a_{ij} (column major form)
- $$= B + (j - L_2) \times (U_1 - L_1 + 1) \times S + (i - L_1) \times S$$
- $$= 2000 + (10) \times (20) \times 4 + 10 \times 4$$
- $$= 2000 + 800 + 40 = 2840$$

Example 1.9.2 : Given an array

`int a[] = {69, 78, 63, 98, 67, 75, 66, 90, 81}`

Calculate address of $a[5]$ if base address is 1600.

MU - Dec. 18, 2 Marks

Solution :

Address of $a[5]$ is given by,

$$\begin{aligned} &= \text{Base address} + \text{Size of each elements} * 5 \\ &= 1600 + 2 \times 5 = 1610 \end{aligned}$$

Example 1.9.3 : Given an array `int a[] = {23, 55, 63, 89, 45, 67, 85, 99}`. Calculate address of $a[5]$ if base address is 5100.

MU - Dec. 19, 2 Marks

Solution :

Address of $a[5]$ is given by,

$$\begin{aligned} &= \text{Base address} + \text{Size of each elements} * 5 \\ &= 5100 + 5 \times 2 = 5110 \end{aligned}$$

1.10 Multi-Dimensional Arrays

C allows arrays of 1 or more dimensional. Exact limit is compiler dependent.

General form of a multi-dimensional array is

`type array_name[S1][S2] ...[Sn];`

Where S_i is the size of the i^{th} dimension.

Some examples are :

`int a[3][4][2];`

`float b[9][6][3];`

A three dimensional array of the size $m \times n \times p$.

$m \rightarrow$ Number of planes (from plate number 0 to $m - 1$)

$n \rightarrow$ Number of rows in each plane (from row number 0 to $n - 1$)

$p \rightarrow$ Number of columns in each plane (from column number 0 to $p - 1$)

Number of elements in each plane = $n \times p$.

Element $a[i][j][k]$, will be found in the i^{th} plane.

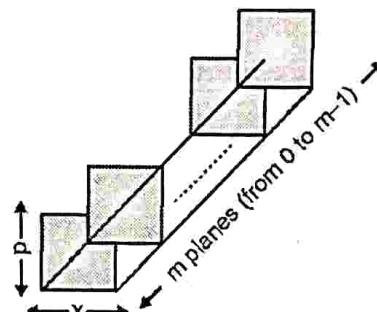


Fig. 1.10.1

Offset of i^{th} plane = $i \times \text{size of plane} = i \times n \times p$

Offset of j^{th} row in a plane = $j \times \text{number of elements in a row}$
 $= j \times p$

Offset of k^{th} element in a row = k

$\therefore a[i][j][k]$ will be found at the address

$$\begin{aligned} &= \text{Starting address of the array} + (i \times n \times p + j \times p + k) \times \text{Size of each element} \end{aligned}$$

1.11 Application of Arrays

1.11.1 Addition of Two 2-D Matrices

- If we add two matrices $a_{m \times n}$ and $b_{m \times n}$ producing $c_{m \times n}$ then $(i, j)^{\text{th}}$ element of the resultant matrix $c[i][j] = a[i][j] + b[i][j]$.
- Two matrices can be added using the following program segment :

```
for(i = 0; i < m; i++) } calculate all elements of
for(j = 0; j < n; j++) } the array c
  c[i][j] = a[i][j] + b[i][j];
```

- Two matrices to be added must be addition compatible i.e. they must be of the same size.

**Program 1.11.1 : Program for addition of two matrices.**

```
#include<stdio.h>
#include<conio.h>
void add(int[ ][10], int[ ][10], int[ ][10], int, int);
void read(int[ ][10], int, int);
void print(int[ ][10], int, int);

void main()
{
    int a[10][10], b[10][10], c[10][10], m, n;
    printf("\n Enter the size of matrix :");
    scanf("%d%d", &m, &n);
    printf("\n Enter the elements of 1'st matrix");
    read(a, m, n);
    printf("\n Enter the elements of 2'nd matrix");
    read(b, m, n);
    add(a, b, c, m, n);
    printf("\n 1'st matrix is :");
    print(a, m, n);
    printf("\n 2'nd matrix is :");
    print(b, m, n);
    printf("\n result matrix is :");
    print(c, m, n);
}

void add(int x[ ][10], int y[ ][10], int z[ ][10],
int m, int n)
{
    int j, i;
    for(i=0; i<m; i++)
        for(j=0; j<n; j++)
            z[i][j]=x[i][j]+y[i][j];
}

void read(int x[ ][10], int m, int n)
{
    int i, j;
    for(i=0; i<m; i++)
        for(j=0; j<n; j++)
    {
        scanf("%d", &x[i][j]);
    }
}

void print(int x[ ][10], int m, int n)
```

```
{
    int i, j;
    for(i=0; i<m; i++)
    {
        printf("\n ");
        for(j=0; j<n; j++)
            printf("%d\t", x[i][j]);
    }
}
```

Output

```
Enter the size of matrix : 3 4
Enter the elements of 1'st matrix
0 0 0
1 1 1
2 2 2
Enter the elements of 2'nd matrix
1 2 3 4
1 2 3 4
1 2 3 4
1'st matrix is :
0 0 0 0 .
1 1 1 1
2 2 2 2
2'nd matrix is :
1 2 3 4
1 2 3 4
1 2 3 4
result matrix is :
1 2 3 4
2 3 4 5
3 4 5 6
```

1. Program segment to count non-zero elements of a matrix

```
count = 0;
for(i = 0; i < m; i++)
    for(j = 0; j < n; j++)
        if(a[i][j] != 0)
            count++;
printf("\n %d", count);
```

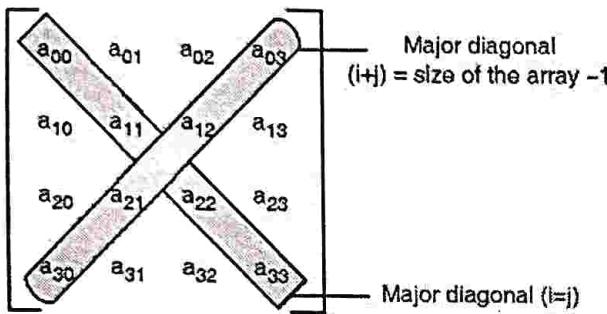


A matrix of $m \times n$ elements is scanned using the nested loop [row wise and inside the row column wise] and whenever a non-zero value is found, the value of count is incremented by 1.

2. Program segment for finding the sum of all elements of;

- (a) Diagonal
- (b) Lower triangle [minus main diagonal]
- (c) Upper triangle [minus main diagonal]

In order to understand the above; let us consider the case of a 4×4 matrix $a[4][4]$.



(a) Main diagonal

For any element a_{ij} , $i = j$

(row index of the element = column index)

(b) Lower triangle

For any element a_{ij} , $i > j$ i.e. row index > column index

(c) Upper triangle

For any element a_{ij} , $i < j$ i.e. row index is less than column index.

3. Program segment for scanning and finding the sum of all elements of lower triangle (Minus main diagonal)

```
sum = 0;
for(i = 1; i < m; i++)
    for(j = 0; j < i; j++)
        sum = sum + a[i][j];
```

- In the outer loop $\text{for}(i = 1; i < m; i++)$, i starts from 1 as the first element of the lower triangle is a_{10} .
- Inner loop $\text{for}(j = 0; j < i; j++)$ is used to scan the elements of the i^{th} row, j varies from 0 to $i - 1$ as $i > j$ for lower triangle.

4. Program segment for finding the sum of diagonal elements

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

5. Program segment for finding the sum of all elements of the upper triangle (Minus main diagonal)

```
sum = 0;
for(i = 0; i < m - 1; i++)
    for(j = i + 1; j < m; j++)
        sum = sum + a[i][j];
```

Algorithm to find product of diagonal elements

```
PROD = 1;
for(i = 0; i < N; i++)
    PROD = PROD * A[i][i];
printf("\n %d", PROD);
```

Program 1.11.2 : Write a C program to find the sum of major and minor diagonal of $m \times n$ matrix.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a[10][10], n, i, j, major, minor;
    printf("\n Enter size of the matrix(n for n x n) : ");
    scanf("%d", &n);
    printf("\n Enter matrix data : ");
    for(i = 0; i < n; i++)
        for(j = 0; j < n; j++)
            scanf("%d", &a[i][j]);
    /* For any element on major diagonal i = j and for
     an element on minor diagonal i + j = n */
    major = minor = 0;
    for(i = 0; i < n; i++)
    {
        major = major + a[i][i];
        minor = minor + a[i][n - i - 1];
    }
    printf("\n Sum of elements of major diagonal
          = %d", major);
```



```

printf("\n Sum of element of minor diagonal
      = %d", minor);
getch();
}

```

Output

Enter size of the matrix(n for n × n) : 4

Enter matrix data :

| | | | |
|---|---|---|---|
| 5 | 4 | 9 | 6 |
| 4 | 1 | 0 | 3 |
| 2 | 2 | 5 | 4 |
| 6 | 1 | 0 | 5 |

Sum of elements of major diagonal = 16

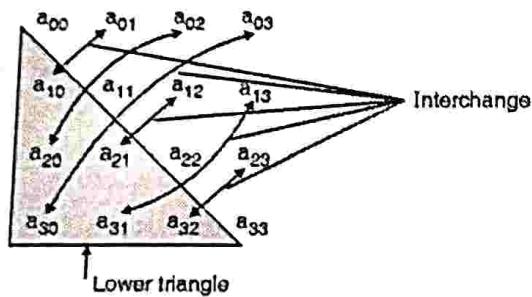
Sum of element of minor diagonal = 14

1.11.2 Transpose of Square Matrix

Transpose of a square matrix is found by interchanging rows and columns.

$$A = \begin{bmatrix} 5 & 1 & 9 & 6 \\ 2 & 0 & 8 & 4 \\ 9 & 6 & 5 & 2 \\ 1 & 0 & 0 & 1 \end{bmatrix}_{4 \times 4}$$

$$A^T = \begin{bmatrix} 5 & 2 & 9 & 1 \\ 1 & 0 & 6 & 0 \\ 9 & 8 & 5 & 0 \\ 6 & 4 & 2 & 1 \end{bmatrix}$$



Transpose of a square matrix can be found by interchanging all elements a_{ij} of the lower triangular matrix with the corresponding elements a_{ji} .

for($i = 1; i < m; i++$) } Loop to scan elements of
 for($j = 0; j < i; j++$) } the lower triangle.
 Interchange($a[i][j], a[j][i]$)

Program 1.11.3 : Program to find the transpose of square matrix.

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

```

void main()
{
  int a[10][10], m, i, j, temp;
  /* actual size of matrix is m*n
  i, j - for scanning of array
  temp - for interchanging of  $a[i][j]$  and  $a[j][i]$  */
  printf("\n Enter the size of matrix :");
  scanf("%d", &m);
  /* Reading elements of matrix */
  printf("\n Enter the values :");
  for(i=0; i<m; i++)
    for(j=0; j<m; j++)
      scanf("%d", &a[i][j]);
  //To print original square matrix
  printf("\n original square matrix is");
  for(i=0; i<m; i++)
  {
    printf("\n ");
    for(j=0; j<m; j++)
      printf("%d\t", a[i][j]);
  }
  /* Finding transpose */
  for(i=1; i<m; i++)
    for(j=0; j<i; j++)
    {
      temp=a[i][j];
      a[i][j]=a[j][i];
      a[j][i]=temp;
    }
  /* printing of all elements of final matrix */
  printf("\n Transpose matrix is :");
  for(i=0; i<m; i++)
  {
    printf("\n ");
    for(j=0; j<m; j++)
      printf("%d\t", a[i][j]);
  }
  getch();
}

```

Output

Enter the size of matrix : 4

Enter the values :

5 1 9 6

2 0 8 4

9 6 5 2



1 0 0 1
original square matrix is

| | | | |
|---|---|---|---|
| 5 | 1 | 9 | 6 |
| 2 | 0 | 8 | 4 |
| 9 | 6 | 5 | 2 |
| 1 | 0 | 0 | 1 |

Transpose matrix is :

| | | | |
|---|---|---|---|
| 5 | 2 | 9 | 1 |
| 1 | 0 | 6 | 0 |
| 9 | 8 | 5 | 0 |
| 6 | 4 | 2 | 1 |

1.11.3 Finding whether a given Square Matrix is Symmetrical

- A square matrix is said to be symmetrical if the matrix $A = A^T$ (A^T = transpose of A).
- A square matrix is said to be symmetrical if for all $a_{ij} \in$ lower triangle, $a_{ij} = a_{ji}$.

Program 1.11.4 : Write a program to print major and minor diagonal elements of square matrix.

```
# include <stdio.h>
# include <conio.h>
void main()
{ int a [10][10], n, i, j;
  printf("\n Enter size of the matrix(n of n×n):");
  scanf("%d", &n);
  printf("\n Enter matrix data :");
  for(i = 0; i < n; i++)
    for(j = 0; j < n; j++)
      scanf("%d", &a[i][j]);
  /* Elements of major diagonal */
  printf("\n Elements of major diagonal\n ");
  for(i=0; i < n; i++)
    printf(" %d", a[i][i]);
  /* Elements of minor diagonal */
  printf("\n \n Elements of minor diagonal\n ");
  for(i=0; i < n; i++)
    printf(" %d", a[i][n-i-1]);
  getch();
}
```

Output

Enter size of the matrix(n of n * n) : 4

Enter matrix data :

5 1 9 6

2 0 8 4

9 6 5 2

1 0 0 1

Elements of major diagonal

5 0 5 1

Elements of minor diagonal

6 8 6 1

Program 1.11.5 : Program for checking of symmetry.

```
#include<stdio.h>
#include<conio.h>
int symmetry(int a[ ][10], int m);
void main()
{ int a[10][10], m, i, j, result;
  printf("\n Enter the size of matrix :");
  scanf("%d", &m);
  /* Reading matrix elements */
  printf("\n Enter the values :");
  for(i=0; i<m; i++)
    for(j=0; j<m; j++)
      scanf("%d", &a[i][j]);
  result=symmetry(a, m);
  if(result==0)
    printf("\n Not symmetrical");
  else
    printf("\n Symmetrical");
  getch();
}
int symmetry(int a[ ][10], int m)
{ int i, j;
  for(i=1; i<m; i++)
    for(j=0; j<i; j++)
      if(a[i][j] != a[j][i])
        return(0);
  return(1);
}
```

**Output**

Enter the size of matrix : 4

Enter the values :

1 2 3 4

2 1 4 3

3 4 1 3

4 3 3 1

Symmetrical

Enter the size of matrix : 4

Enter the values :

1 2 3 4

1 2 3 4

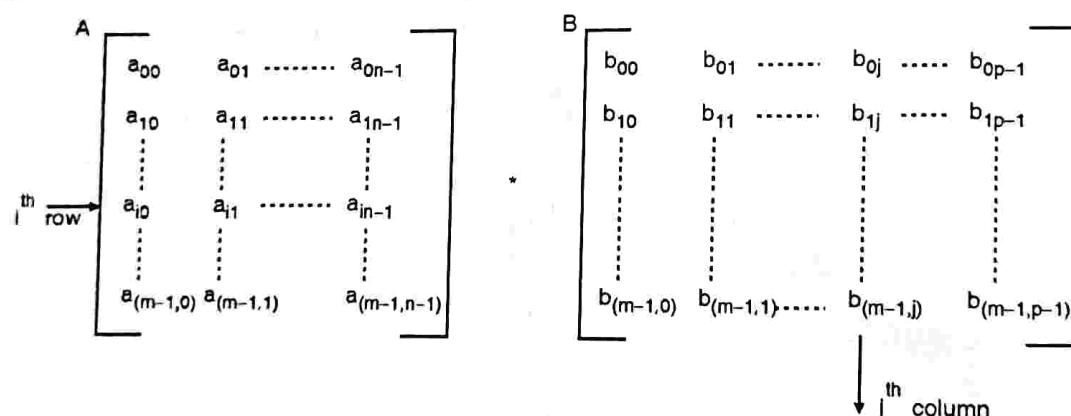
1 2 3 4

1 2 3 4

Not symmetrical

1.11.4 Multiplication of Two Matrices $A_{m \times n}$ and $B_{n \times p}$

In order that two matrices A and B are multiplication compatible, number of columns of matrix A should be equal to the number of rows of the matrix B. The resultant matrix C will have a size of $m \times p$ method for finding the $(i, j)^{th}$ element of the matrix C.



$$C_{ij} = \text{dot product of the } i^{\text{th}} \text{ row of A and the } j^{\text{th}} \text{ column of B}$$

$$\therefore C_{ij} = a_{i0} \times b_{0j} + a_{i1} \times b_{1j} + a_{i2} \times b_{2j} + \dots + a_{in-1} \times b_{n-1j} = \sum_{k=0}^{n-1} a_{ik} \times b_{kj}$$

equivalent loop for finding C_{ij} in C-language

```
C[i][j] = 0;
```

```
for(k = 0; k < n; k++)
```

```
    C[i][j] = C[i][j] + A[i][k] * B[k][j];
```

Once we know how to calculate C_{ij} , we can calculate all elements of the resultant matrix C by varying i from 0 to $m - 1$ and j from 0 to $p - 1$.



Program 1.11.6 : Program for multiplication of two matrices.

```
#include<stdio.h>
#include<conio.h>
void read(int[ ][10], int, int);
void print(int[ ][10], int, int);
void multiply(int[ ][10], int[ ][10], int[ ][10], int, int,
int);
void main()
{
    int a[10][10], b[10][10], c[10][10], m, n, p;
    /* first matrix of size m*n (a)
       second matrix of size n*p (b)
       resultant matrix of size m*p (c)
    */
    printf("\n Enter the size of 1'st matrix :");
    scanf("%d%d", &m, &n);
    printf("\nEnter no. of columns in the 2'nd matrix :");
    scanf("%d", &p);
    printf("\nEnter the data of 1'st matrix :");
    read(a, m, n);
    printf("\nEnter the data of 2'nd matrix :");
    read(b, n, p);
    multiply(a, b, c, m, n, p);
    printf("\n Result is :");
    print(c, m, p);
}
void read(int a[ ][10], int m, int n)
{
    int i, j;
    for(i=0; i<m; i++)
        for(j=0; j<n; j++)
            scanf("%d", &a[i][j]);
}
void print(int a[ ][10], int m, int n)
{
    int i, j;
    for(i=0; i<m; i++)
    {
        printf("\n ");
        for(j=0; j<n; j++)
            printf("%d\t", a[i][j]);
    }
}
void multiply(int a[ ][10], int b[ ][10], int c[ ][10], int
m, int n, int p)
```

```
{
    int i, j, k;
    for(i=0; i<m; i++)
        for(j=0; j<p; j++)
    {
        c[i][j]=0;
        for(k=0; k<n; k++)
            c[i][j]=c[i][j]+a[i][k]*b[k][j];
    }
}
```

Output

```
Enter the size of 1'st matrix :2 2
Enter no. of columns in the 2'nd matrix :2
Enter the data of 1'st matrix :
2 2
2 2
Enter the data of 2'nd matrix :
3 3
3 3
Result is :
12 12
12 12
```

Example 1.11.1 : Write a function in 'C' to find maximum of each row for a given matrix of order $M \times N$ having integer elements and print them.

Solution :

```
void maxrow(int a [10][10], int M, int N)
{
    int i, j, k;
    //K point to the largest element
    for(i=0; i<M; i++)
    {
        k=0;
        for(j=1; j<N; j++)
            if(a[i][j] > a[i][k])
                k=j;
        printf("\n largest of row %d=%d", i,
a[i][k]);
    }
}
```



Module 2

Stack and Queues

Syllabus

Introduction, ADT of Stack, Operations on Stack, Array Implementation of Stack, Applications of Stack – Well-formness of Parenthesis, Infix to Postfix Conversion and Postfix Evaluation, Recursion. Introduction, ADT of Queue, Operations on Queue, Array Implementation of Queue, Types of Queue-Circular Queue, Priority Queue, Introduction of Double Ended Queue, Applications of Queue.

2.1 Introduction

- 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. 2.1.1 shows expansion and shrinking of a stack. Initially stack is empty.

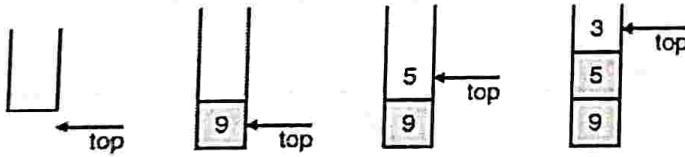


Fig. 2.1.1 : Insertion of 9,5,3 in a stack

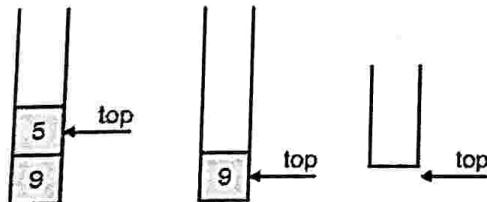


Fig. 2.1.2 : Deletion of 3 elements from the stack

- A variable `top`, points to the top element of the list.

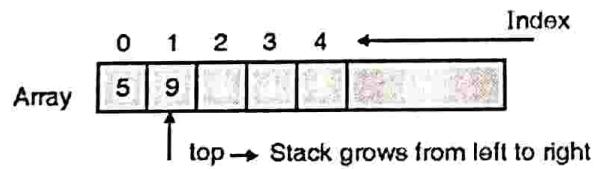
2.2 Operations on Stacks

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)
6. `display_top()` : Returns the top element.

2.3 Array Representation

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[MAX];
    int top;
}stack;
/*Max is a constant, maximum number of elements
that can be stored */
```

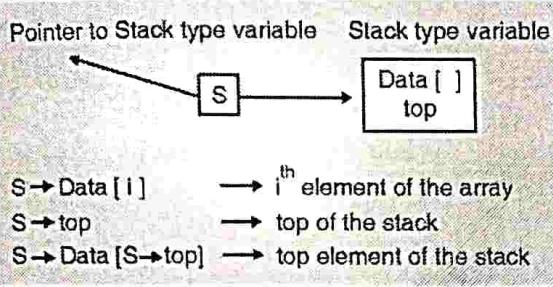
- Initially, `top` is set to `-1`
- A stack with `top` as `-1` is an empty stack.
- When the value of `top` becomes `MAX - 1` after a series of insertions, it is full.
- After “push” operation $\text{top} = \text{top} + 1$ (Stack Growing)
- After “pop” operation $\text{top} = \text{top} - 1$ (Stack Shrinking)



2.3.1 'C' Functions for Primitive Operations on a Stack

```
void initialize(stack *S)
{
    S-> top = -1;
}
```

Initially, stack is empty.



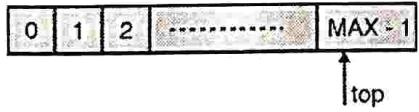
Accessing elements of a stack using a pointer

```
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.



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;
}
```

Note : 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);
}
```

```
int display_top(stack *S)
{
    if(! empty(S))
        return(S-> data [S-> top]);
    return(-1); // if stack is empty
}
```

Note : Main program should ensure that the stack is not empty before making a call to pop() function. It may cause an under flow, otherwise.

The function push(), increase the value of the top by 1 and then stores the element at the top location in "data" array.

The function pop() returns the top elements from the stack. It also, reduces the value of top by 1 (one element is deleted from the stack).

2.3.2 Program Showing Stack Operations

Convert a decimal number to binary using a stack.

Example :

Input number = 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 |

$$(21)_{10} = (10101)_2$$

A decimal number can be converted to binary from 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 2.3.1 : Program for conversion of decimal number to binary form.

OR

Write a 'C' program to convert decimal to binary using any appropriate data structure you have studied.

MU - Dec. 13, 7 Marks

```
#include<stdio.h>
#include<conio.h>
#define MAX 20
typedef struct stack
{
    int data[MAX];
    int top;
}stack;
void init(stack *s);
int empty(stack *s);
int full(stack *s);
int pop(stack *s);
void push(stack *s, int);
void main()
{
    stack s;
    int x;
    init(&s);
```

```
printf("\n Enter decimal number:");
scanf("%d", &x);
while((x != 0))
```

```
{ if(!full(&s))
{
    push(&s, x%2);
    x = x/2;
}
```

```
else
{
    printf("\n stack 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

```

Program 2.3.2 : Program to check whether a given string is a palindrome.

```

#include <stdio.h>
#include <conio.h>
typedef struct stack
{
    char data[30];
    int top;
} stack;
void init(stack *p)
{
    p->top = -1;
}
void push( stack *p, char x)
{
    p->top = p->top + 1;
    p->data[p->top] = x;
}
char POP(stack *p)
{
    char x;
    x = p->data[p->top];
    p->top = p->top - 1;
    return(x);
}
int empty(stack *p)
{

```

```

{
    if(p->top == -1)
        return(1);
    return(0);
}
void palindrome(char[ ]);
void main( )
{
    stack s;
    char text [20];
    int i;
    init(&s);
    printf("\n Enter a string:");
    gets(text);
    palindrome(text);
}
void palindrome(char text[ ])
{
    stack S;
    int i;
    init(&S);
    /* Stack is being used to reverse the sting. Original
    string and the reversed string are compared character
    by character */
    for(i = 0; text[i] != '\0'; i++)
        push(&S, text[i]);
    for(i = 0; text[i] != '\0'; i++)
        if(text[i] != POP(&S))
            break;
        if(text[i] != '\0')
            /* A mismatch found before end of the string */
            printf("\n Not a palindrome");
        else
            printf("\n A Palindrome");
    getch();
}

```

Output

```

Enter a string : MADAM
A palindrome.

```



Program 2.3.3 : Write a program to perform the following operations :

- Push the number 10 and 20 on first stack.
- Push the number 100 and 200 on second stack.
- Pop all elements from both stack one by one and display.

```
# include <stdio.h>
# include <conio.h>
# define MAX 10
typedef struct stack
{
    int data[MAX];
    int top;
} stack;
void init(stack *s)
{
    s-> top = -1;
}
int empty(stack *s)
{
    if(s-> top == -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 main()
{
    int x;
    stack st1, st2;
    init(&st1);
```

```
init(&st2);
push(&st1, 10);
push(&st1, 20);
push(&st2, 100);
push(&st2, 200);
printf("\n contents of first stack :");
while(! empty(&st1))
{
    x = pop(&st1);
    printf("%d\t", x);
}
printf("\n contents of second stack :");
while(! empty(&st2))
{
    x = pop(&st2);
    printf("%d\t", x);
}
```

2.3.3 Well-Formedness of Parenthesis

MU - Dec. 15

University Question

Q. Write a function in C to maintain 2 stacks in a single array.
(Dec. 15, 10 Marks)

An expression is said to be well formed if every opening bracket has a corresponding closing bracket and there is no extra bracket.

Examples of well formed Expressions :

1. (())
2. ((()))
3. ((())) ()

Algorithm to read in a parenthesised infix expression and check well-formedness of parenthesis

(I) Check whether parentheses match

s : stack of characters

x : character type

Step 1 : $x \leftarrow$ read the next token

Step 2 : if($x == '('$)

Push(s,x);

Step 3 : if($x == ')'$)

if(top element of the stack is '(')

pop(s);

else



Print "mismatch"

Step 4 : if(more tokens)

 goto step1

Step 5 : if(stack is not empty)

 Print "mismatch"

 else

 Print "match"

Step 6 : stop

Program 2.3.4 : Program for checking well-formedness of parenthesis.

OR Use stack data structure to check well-formedness of parentheses in an algebraic expression. Write C program for the same. **MU - Dec. 18, 10 Marks**

OR Write a program in 'C' to check for balanced parenthesis in an expression using stack. **MU - May 19, 10 Marks**

```
void main()
{
    char x;
    stack s;
    init(&s);
    printf("\n enter a parenthesized expression:");
    while((x = getchar()) != '\n')
    {
        switch(x)
        {
            case '(' : push(&s, '(');
                        break;
            case ')' : if(!empty(&s))
                        pop(&s);
                        else
                        {
                            printf("\n mismatch ...");
                            exit(0);
                        }
                        break;
            default : break;
        }
    }
    if(!Empty(&s))
        printf("\n mismatch:");
    else
        printf("\n well formed:");
}
```

2.3.4 Operations on Stack Considering Overflow and Underflow

'C' functions for operations on stack

```
void push()
{
    if(p->top == N-1)
        printf("\n overflow !! 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("\n Underflow !!! 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]);
}
```

2.3.5 Stack as an ADT

MU - Dec. 15, Dec. 16

University Question

Q. Explain STACK as ADT. (Dec. 15, Dec. 16, 5 Marks)

A stack is an ordered list with the restriction that insertions and deletions can be performed at only one position, namely, the front end of the list, called the top. The fundamental operation on the stack are as follows.



- Push, equivalent to an insert.
- POP, equivalent to deleting the most recently inserted element.
- A POP on an empty stack is considered to be an error.
- A push operation on stack that is full is considered to be an error.

Structure used for stack

```
typedef struct stack
{
    int data[SIZE];
    /* int has been taken for convenience, it could be
       any data type */
    int top;
} stack;
```

Operation on stack

| | |
|----------------------|---|
| initialize(stack *) | /* Make a stack empty by setting top equal to -1 */ |
| boolean empty(stack) | /* Determine, whether the stack is empty */ |
| boolean full(stack) | /* Determine, whether the stack is full */ |
| int POP(stack *) | /* If the stack is not empty then POP the top element */ |
| push(stack*, int) | /* If the stack is not full then PUSH a new data on top of the stack */ |
| int gettop(stack*) | /* If the stack is not empty then retrieve the top element */ |

2.4 Applications of Stack

MU - Dec. 16

University Question

Q. Give application of stack. (Dec. 16, 2 Marks)

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

2.4.1 Expression Representation

MU - May 15

University Question

Q. Explain infix, postfix and prefix expressions with an example. (May 15, 6 Marks)

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 | $+ x * y z$ |
| Postfix | $x y z * +$ |

(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 .

Note : 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 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.

$$+ 5 * 3 2$$

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 expression

5 3 2 * +

- Find the first operator from left to right and perform the operation.
- First operator is * and therefore, 3 and 2 are multiplied expression becomes 5 6 +
- First operator is + and therefore, 5 and 6 are added expression becomes : 11

Note : Prefix and postfix expressions are free from any precedence. They are more suited for mechanization. Computer uses postfix form for representation of an expression.

2.4.2 Evaluation of a Postfix Expression using a Stack

MU - Dec. 15

University Question

Q. Write a function in 'C' to convert prefix expression to postfix expression. (Dec. 15, 5 Marks)

Given expression Stack

6 5 3 + 9 * +



Initially stack is empty.

- First token is an operand, push 6 on the stack.

5 3 + 9 * +



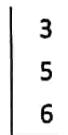
- Next token is an operand, push 5 on the stack.

3 + 9 * +



- Next token is an operand, push 3 on the stack.

+ 9 * +



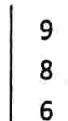
- Next token is an operator, pop two operands 3 and 5, add them and push the result on the stack.

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 78 ← 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 2.4.1 : Program for evaluation of a postfix expression.

OR Write program in 'C' to evaluate a postfix expression using stack ADT.

MU - Dec. 13, May 18, 10 Marks

```

/* 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("\n Enter the expression(i.e 59+3*) \n
single digit operand and operators only:");
    while((x = getchar()) != '\n')
    {
        if(isdigit(x))
            push(&s, x-48);
        else
        {
            op2 = pop(&s);
            op1 = pop(&s);
            val = evaluate(x, op1, op2);
            push(&s, val);
        }
    }
    val = pop(&s);
    printf("\n value 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:
value of expression = 78

Example 2.4.1 : 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\-

Solution :

| Step | Input | Stack |
|------|--------------|---------------|
| 1. | ABC + * DE/- | |
| 2. | BC + *DE/- | 5 |
| 3. | C + * DE/- | 6 5 |
| 4. | + * DE/- | 2 6 5 |
| 5. | * DE/- | 8 5 |
| 6. | DE/- | 40 |
| 7. | E/- | 12 40 |
| 8. | /- | 4 12 40 |
| 9. | - | 3 40 |
| 10. | End | 37 |

∴ Value of the expression = 37

Example 2.4.2 : Compare stacks and queues.**Solution : Comparison of stack and queue**

| Sr. No. | Stack | Queue |
|---------|---|---|
| 1. | Stack is last in first out (LIFO) i.e. which is entered last will be retrieved firstly. | Queue is first out (FIFO) i.e. which is entered first will be served first. |
| 2. | Stack is Linear data structure which follows LIFO. | Queue is Linear data structure which follows FIFO. |
| 3. | Insertions and deletions are possible through one end called top. | Insertions are at the rare end and deletions are from the front end in a queue. |

| Sr. No. | Stack | Queue |
|---------|-----------------------------|----------------------------------|
| 4. | Example : Books in library. | Example : Cinema ticket counter. |

Example 2.4.3 : Evaluate the following postfix expression.

Show all steps :

ab * c + d - e + where a = 5, b = 4,
c = 10, d = 15 and e = 6.

Solution :

| Step | Expression | Stack |
|------|------------|----------|
| 1. | ab*c+d-e+ | |
| 2. | b*c+d-e+ | 5 |
| 3. | *c+d-e+ | 4 5 |
| 4. | c+d-e+ | 20 |
| 5. | +d-e+ | 10 20 |
| 6. | d-e+ | 30 |
| 7. | -e+ | 15 30 |
| 8. | e+ | 15 |
| 9. | + | 6 15 |
| 10. | End | 21 |

∴ Value of the expression = 21

Example 2.4.4 : Find the value of expression :

5 6 2 + * 12 4 / -

Show the contents of stack and variables operand 1, operand 2 and value.

Solution :

| Step | Input | Stack | Operand 1 | Operand 2 | Value |
|------|--------------------|---------------|-----------|-----------|-------|
| 1. | 5 6 2 + * 12 4 / - | Empty | - | - | - |
| 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 12 40 | 4 | 3 | - |
| 10. | End | 37 | 40 | 3 | 37 |

∴ Value of the expression = 37



Example 2.4.5 : Evaluate the following postfix expression using stack.

$623 + -382 / *2\$3 +$

Solution :

$623 + -382 / *2\$3 +$  ← Initially stack is empty.

- First token is an operand, push 6 on the stack.

$23 + -382 / *2\$3 +$ 

- Next token is an operand, push 2 on the stack.

$3 + -382 / *2\$3 +$ 

- Next token is an operation, push 3 on the stack.

$+ -382 / *2\$3 +$ 

- Next token is an operator, POP two operands 3 and 2, add them and push the result on the stack.

$-382 / *2\$3 +$ 

- Next token is an operator, POP two operands 6 and 5, subtract them and push the result on the stack.

$382 / *2\$3 +$ 

- Next token is an operand, push 3 on the stack.

$82 / *2\$3 +$ 

- Next token is an operand, push 8 on the stack.

$2 / *2\$3 +$ 

- Next token is an operand, push 2 on the stack.

$/ *2\$3 +$ 

- Next token is an operator, POP two operands 8 and 2, divide them and push the result in the stack.

$+ *2\$3 +$ 

3
 1

- Next token is an operator, POP two operands 3 and 4, add them and push the result in the stack.

$*2\$3 +$ 

1

- Next token is an operator, POP two operands 1 and 7, multiply them and push the result in the stack.

$2\$3 +$ 

7

- Next token is an operand, push 2 on the stack.

$\$3 +$ 

7

- Next token is an operand, POP two operand 7 and 2, find the power and push the result on the stack.

$3 +$ 

49

- Next token is an operand, push 3 on the stack.

$+ 3$ 

49

- Next token is an operator, POP two operands 49 and 3, add them and push the result on the stack.

52

∴ Value of the expression = 52

2.4.3 Conversion of an Expression from Infix to Postfix

Example of an infix expression : $(A + B \wedge C) * D + E \wedge 5$ where A, B, C, D and E are integer constants and $B \wedge C$ means B^C .

Manual method :

Step 1 :

In manual evaluation of the above expression, $B \wedge C$ must be calculated first. Hence, we convert $B \wedge C$ to its equivalent postfix equivalent i.e. $BC\wedge$.

$(A + B \wedge C) * D + E \wedge 5$: Original expression

$(A + BC\wedge) * D + E \wedge 5$: $BC\wedge$ should treated as a single integer number (single token) and $A + BC\wedge$ should be the next operation to be performed.

- ABC $\wedge + * D + E \wedge 5$: ABC $\wedge + * D$ is the next operation to be performed.
ABC $\wedge + D * + E \wedge 5$: E $\wedge 5$ is the next operation to be performed.
ABC $\wedge + D * + E 5 \wedge$: + is the last operation to be performed.
ABC $\wedge + D * E 5 \wedge +$: Final expression.

Algorithmic approach :

A + B * C

- In the above example, evaluation of * precedes evaluation of + as * has higher precedence over +.

Note : Perform an operation if the current operator has equal or higher precedence over the succeeding operator.

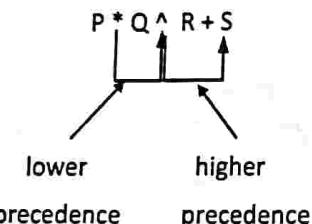
- Thus an operator coming on the right of the current operator will determine if the current operation should be performed.

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 $\wedge, *, +$
- All operators must be saved on top of the stack until we get an equal or higher precedence operator.

Conversion of P * Q \wedge R + S

| Expression | Stack | Output | Remark |
|----------------------|------------|--------------------|--|
| P * Q \wedge R + S | NULL | - | - |
| * Q \wedge R + S | NULL | P | Operand must be printed |
| Q \wedge R + S | * | P | * operation will be performed if the next operator is of lower or equal precedence |
| \wedge R + S | * | PQ | Operand must be printed |
| R + S | * \wedge | PQ | * cannot be performed as \wedge has higher precedence |
| + S | * \wedge | PQR | Operand must be printed |
| + S | NULL | PQR \wedge * | All higher precedence operators (compare to +) are popped and printed and finally the current operator is pushed on top of the stack |
| S | + | PQR \wedge * | - |
| NULL | + | PQR \wedge * S | Operand must be printed |
| Null | Null | PQR \wedge * S + | Finally, all operators are popped and printed |

Algorithm for conversion of an expression from Infix to postfix

```

s : stack
while(more tokens)
{
    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 2.4.6 : Convert the following expression from infix to postfix using a stack.

a && b || c || !(e > f)

Solution :

| Sr. No. | Expression | Stack | Output | Comment |
|---------|-------------------------|-------|------------------|---|
| 1. | a && b c !(e > f) | NULL | - | Initial condition |
| 2. | && b c !(e > f) | NULL | a | Print a |
| 3. | b c !(e > f) | && | a | Push && |
| 4. | c !(e > f) | && | ab | Print b |
| 5. | c !(e > f) | NULL | ab && | Pop and print higher precedence operators |
| 6. | c !(e > f) | | ab && | Push the current operator |
| 7. | !(e > f) | | ab && c | Print c |
| 8. | !(e > f) | NULL | ab && c | Pop and print equal or higher precedence operators |
| 9. | !(e > f) | | ab && c | Push the current operator |
| 10. | ! (e > f) | | ab && c | Pop and print equal or higher precedence operators |
| 11. | (e > f) | | ab && c | Push the current operator |
| 12. | e > f | (| ab && c | '(' should always be pushed. |
| 13. | > f) | (| ab && c e | Print e |
| 14. | > f) | (| ab && c e | Pop and print equal or higher precedence operators. Remember '(' has lowest precedence |
| 15. | f) | (> | ab && c e | Push the current operator |
| 16. |) | (> | ab && c e f | Print f |
| 17. | NULL | | ab && c e f > | When the next input is ')' then all operators until '(' should be popped and printed |
| 18. | NULL | NULL | ab && c e f > | Pop and print all operations. |

Example 2.4.7 : Convert the following arithmetic expression into postfix and show stack status after every step in tabular forms : A + (B * C - (D / E - F) * G) * H.

**Solution :**

| Sr. No. | Stack | Input | Output |
|---------|---------|-----------------------------------|-----------------|
| 1. | Empty | $A + (B * C - (D/E - F) * G) * H$ | - |
| 2. | Empty | $+ (B * C - (D/E - F) * G) * H$ | A |
| 3. | + | $(B * C - (D/E - F) * G) * H$ | A |
| 4. | + (| $B * C - (D/E - F) * G) * H$ | A |
| 5. | + (| $* C - (D/E - F) * G) * H$ | AB |
| 6. | + (* | $C - (D/E - F) * G) * H$ | AB |
| 7. | + (* | $- (D/E - F) * G) * H$ | ABC |
| 8. | + (- | $(D/E - F) * G) * H$ | ABC* |
| 9. | + (- (| $D/E - F) * G) * H$ | ABC* |
| 10. | + (- (| $/E - F) * G) * H$ | ABC*D |
| 11. | + (- (/ | $E - F) * G) * H$ | ABC*D |
| 12. | + (- (/ | $- F) * G) * H$ | ABC*DE |
| 13. | + (- (- | $F) * G) * H$ | ABC*DE/ |
| 14. | + (- (- | $) * G) * H$ | ABC*DE/F |
| 15. | + (- | $* G) * H$ | ABC*DE/F- |
| 16. | + (- * | $G) * H$ | ABC*DE/F- |
| 17. | + (- * | $) * H$ | ABC*DE/F-G |
| 18. | + | $* H$ | ABC*DE/F-G*- |
| 19. | + * | H | ABC*DE/F-G*- |
| 20. | + * | End | ABC*DE/F-G*-H |
| 21. | Empty | End | ABC*DE/F-G*-H*+ |

Example 2.4.8 : Convert the following infix expression to postfix using stack. $A/B\$C + D^*E - A^*C$ **Solution :**

| Sr. No. | Expression | Stack | Output | Comment |
|---------|------------------------|-------|--------|-------------------|
| 1. | $A/B\$C + D^*E - A^*C$ | NULL | - | Initial condition |
| 2. | $/B\$C + D^*E - A^*C$ | NULL | A | Print operand |
| 3. | $B\$C + D^*E - A^*C$ | / | A | Push operand |
| 4. | $\$C + D^*E - A^*C$ | / | AB | Print operand |
| 5. | $C + D^*E - A^*C$ | /\$ | AB | Push operand |



| Sr. No. | Expression | Stack | Output | Comment |
|---------|-------------|-------|-----------------|---------------------------------------|
| 6. | + D*E - A*C | /\$ | ABC | Print operand |
| 7. | D*E - A*C | + | ABC\$/ | POP and print '/\$' and then push '+' |
| 8. | *E - A*C | + | ABC\$/D | Print operand |
| 9. | E - A*C | +* | ABC\$/D | Push operand |
| 10. | - A*C | +* | ABC\$/DE | Print operand |
| 11. | A*C | - | ABC\$/DE*+ | POP and print '+*' and then push '-' |
| 12. | *C | - | ABC*\$/DE*+A | Print operand |
| 13. | C | -* | ABC*\$/DE*+A | Push operator |
| 14. | End | -* | ABC*\$/DE*+AC | Print operand |
| 15. | End | NULL | ABC*\$/DE*+AC*- | POP and print all operator. |

Result = ABC*\$/DE*+A*-

Example 2.4.9 : Convert the following expression into postfix. Show all steps : $a + b * c/d - e$

Solution :

| Sr. No. | Expression | Stack | Output |
|---------|-------------------|-------|---------------|
| 1. | $a + b * c/d - e$ | NULL | - |
| 2. | $+ b * c/d - e$ | NULL | a |
| 3. | $b * c/d - e$ | + | a |
| 4. | $* c/d - e$ | + | ab |
| 5. | $c/d - e$ | +* | ab |
| 6. | $/d - e$ | +* | abc |
| 7. | $d - e$ | +/ | abc * |
| 8. | $- e$ | +/ | abc * d |
| 9. | e | - | abc * d/+ |
| 10. | end | - | abc * d/+ e |
| 11. | end | NULL | abc * d/+ e - |

Postfix : abc * d/+ e -

Example 2.4.10 : Convert the following infix expression into postfix expression using stack.

$A/B^{**}C+D^*E-A*C$

Solution :

| Sr. No. | Expression | Stack | Output | Comment |
|---------|----------------------|-------|--------|---|
| 1. | $A/B^{**}C+D^*E-A*C$ | NULL | - | Initial condition |
| 2. | $/B^{**}C+D^*E-A*C$ | NULL | A | Print operand |
| 3. | $B^{**}C+D^*E-A*C$ | / | A | POP and print higher and equal precedence operators and then push the current operator. |



| Sr. No. | Expression | Stack | Output | Comment |
|---------|---------------|-------|----------------|---|
| 4. | $**C+D*E-A*C$ | / | AB | Print the operand |
| 5. | $C+D*E-A*C$ | /** | AB | POP and print higher and equal precedence operators and then push the current operator. |
| 6. | $+D*E-A*C$ | /** | ABC | Print the operand |
| 7. | $D*E-A*C$ | + | ABC**/ | POP and print higher and equal precedence operators and then push current operator. |
| 8. | $*E-A*C$ | + | ABC**/D | Print the operand |
| 9. | $E-A*C$ | + | ABC**/D | POP and print higher and equal precedence operators and then push the current operator |
| 10. | $-A*C$ | +* | ABC**/DE | Print the operand |
| 11. | $A*C$ | - | ABC**/DE*+ | POP and print higher and equal precedence operators and then push the current operator. |
| 12. | $*C$ | - | ABC**/DE*+A | Print the operand |
| 13. | C | -* | ABC**/DE*+A | POP and print equal and higher precedence operators and then push the current operator. |
| 14. | End | -* | ABC**/DE*+AC | Print operand |
| 15. | End | NULL | ABC**/DE*+AC*- | POP and print all operators from the stack. |

Output : ABC**/DE*+AC*-

Example 2.4.11 : Convert the following expression from infix to postfix using a stack.

$$[a + (b - c)] * [(d - e) / (f - g + h)].$$

Solution :

| Sr. No. | Expression | Stack | Output |
|---------|---|--------|--------|
| 1. | $[a + (b - c)] * [(d - e) / (f - g + h)]$ | NULL | - |
| 2. | $a + (b - c)] * [(d - e) / (f - g + h)]$ | [| - |
| 3. | $+ (b - c)] * [(d - e) / (f - g + h)]$ | [| a |
| 4. | $(b - c)] * [(d - e) / (f - g + h)]$ | [+ | a |
| 5. | $b - c)] * [(d - e) / (f - g + h)]$ | [+ (| a |
| 6. | $-c)] * [(d - e) / (f - g + h)]$ | [+ (- | ab |
| 7. | $c)] * [(d - e) / (f - g + h)]$ | [+ (- | ab |
| 8. | $)] * [(d - e) / (f - g + h)]$ | [+ (- | abc |
| 9. | $] * [(d - e) / (f - g + h)]$ | [+ | abc - |



| Sr. No. | Expression | Stack | Output |
|---------|---------------------|-----------|---------------------------|
| 10. | * [(d-e)/ (f-g +h)] | NULL | abc - + |
| 11. | [(d-e)/ (f-g +h)] | * | abc - + |
| 12. | (d-e)/ (f-g +h)] | *[| abc - + |
| 13. | d-e)/ (f-g +h)] | *[(| abc - + |
| 14. | -e)/ (f-g +h)] | *[(| abc - + d |
| 15. | e)/ (f-g +h)] | * [(- | abc - + d |
| 16. |)/ (f-g +h)] | * [(- | abc - + de |
| 17. | / (f-g +h)] | *[| abc - + de - |
| 18. | (f-g +h)] | *[/ | abc - + de - |
| 19. | f-g +h)] | * [/ (| abc - + de - |
| 20. | -g +h)] | * [/ (| abc - + de - f |
| 21. | g +h)] | * [/ (- | abc - + de - f |
| 22. | +h)] | * [/ (- | abc - + de - fg |
| 23. | h)] | * [/ (+ | abc - + de - fg - |
| 24. |)] | * [/ (+ | abc - + de - fg - h |
| 25. |] | *[/ | abc - + de - fg - h + |
| 26. | NULL | * | abc - + de - fg - h + / |
| 27. | NULL | NULL | abc - + de - fg - h + / * |

Example 2.4.12 : Convert the following infix expressions into postfix expression using stack :

- (i) $((A + B)^* C - (D - E)) \$ (F + G)$, where \$-Exponent
- (ii) $A\$B^*C - D + E/F/(G + H)$.

Solution :

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

| Sr. No. | Stack | Infix expression | Postfix expression |
|---------|-------|--------------------------------------|--------------------|
| 1. | Empty | $((A + B)^* C - (D - E)) \$ (F + G)$ | - |
| 2. | (| $(A + B)^* C - (D - E)) \$ (F + G)$ | - |
| 3. | ((| $A + B)^* C - (D - E)) \$ (F + G)$ | - |
| 4. | ((| $+ B)^* C - (D - E)) \$ (F + G)$ | A |
| 5. | ((+ | $B)^* C - (D - E)) \$ (F + G)$ | A |
| 6. | ((+) | $)^* C - (D - E)) \$ (F + G)$ | AB |
| 7. | (| $* C - (D - E)) \$ (F + G)$ | AB+ |
| 8. | (* | $C - (D - E)) \$ (F + G)$ | AB+ |
| 9. | (* | $- (D - E)) \$ (F + G)$ | AB + C |
| 10. | (- | $(D - E)) \$ (F + G)$ | AB + C* |
| 11. | (-(| $D - E)) \$ (F + G)$ | AB + C * |



| Sr. No. | Stack | Infix expression | Postfix expression |
|---------|-----------|------------------|------------------------|
| 12. | (- (| - E) \$ (F + G) | AB + C * D |
| 13. | (- (- | E) \$ (F + G) | AB + C * D |
| 14. | (- (-) |) \$ (F + G) | AB + C * DE |
| 15. | (-) |) \$ (F + G) | AB + C * DE - |
| 16. | Empty | \$ (F + G) | AB + C * DE -- |
| 17. | \$ | (F + G) | AB + C * DE -- |
| 18. | \$ (| F + G) | AB + C * DE -- |
| 19. | \$ (| + G) | AB + C * DE -- F |
| 20. | \$ (+ | G) | AB + C * DE -- F |
| 21. | \$ (+) | | AB * C * DE -- FG |
| 22. | \$ | End | AB + C * DE -- FG + |
| 23. | Empty | End | AB + C * DE -- FG + \$ |

(ii) A \$ B * C - D + E/F/(G + H)

| Sr. No. | Stack | Infix expression | Postfix expression |
|---------|-------|------------------------------|-----------------------------|
| 1. | Empty | A \$ B * C - D + E/F/(G + H) | - |
| 2. | Empty | \$ 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 + / + |



Example 2.4.13 : Convert following infix expression to an equivalent postfix expression. $A + B * C - D/F$.

Solution :

| Sr. No. | Expression | Stack | Output | Comment |
|---------|-------------------|-------|----------------|--|
| 1. | $A + B * C - D/F$ | NULL | - | Initial condition |
| 2. | $+ B * C - D/F$ | NULL | A | Print |
| 3. | $B * C - D/F$ | + | A | Pop equal and higher precedence operators from the stack and then push the current operator. |
| 4. | $* C - D/F$ | + | AB | Print |
| 5. | $C - D/F$ | + * | AB | Pop equal and higher precedence operators from the stack and then push the current operator |
| 6. | $- D/F$ | + * | ABC | Print |
| 7. | D/F | - | ABC * + | Pop equal and higher precedence operators from the stack and then push the current operator |
| 8. | $/F$ | - | ABC * + D | Print |
| 9. | F | - / | ABC * + D | Pop equal and higher precedence operators from the stack and then push the current operator |
| 10. | $-$ | - / | ABC * + DF | Print |
| 11. | $-$ | NULL | ABC * + DF / - | Pop all operators, one by one from the stack and print them. |

Postfix equivalent = ABC * + DF / -

Example 2.4.14 Convert the expression to postfix : $(f - g) * ((a + b) * (c - d))/e$

MU - May 17, 5 Marks

Solution :

| Sr. No. | Stack | Infix expression | Postfix expression |
|---------|-------------|-----------------------------------|--------------------|
| 1. | Empty | $(f - g) * ((a + b) * (c - d))/e$ | - |
| 2. | (| $f - g) * ((a + b) * (c - d))/e$ | - |
| 3. | (| $- g) * ((a + b) * (c - d))/e$ | f |
| 4. | (- | $g) * ((a + b) * (c - d))/e$ | f |
| 5. | (- | $) * ((a + b) * (c - d))/e$ | fg |
| 6. | Empty | $* ((a + b) * (c - d))/e$ | fg- |
| 7. | * | $((a + b) * (c - d))/e$ | fg- |
| 8. | *((| $a + b) * (c - d))/e$ | fg- |
| 9. | *((+ | $+ b) * (c - d))/e$ | fg - a |
| 10. | *((+ b) | $)* (c - d))/e$ | fg - a |
| 11. | *((+ b) * | $(c - d))/e$ | fg - ab |



| Sr. No. | Stack | Infix expression | Postfix expression |
|---------|-------|------------------|-----------------------|
| 12. | *(| $*(c - d)) / e$ | $fg - ab +$ |
| 13. | *(* | $(c - d)) / e$ | $fg - ab +$ |
| 14. | *(*() | $c - d)) / e$ | $fg - ab +$ |
| 15. | *(*() | $- d)) / e$ | $fg - ab + c$ |
| 16. | *(*(- | $d)) / e$ | $fg - ab + c$ |
| 17. | *(*(- | $)) / e$ | $fg - ab + cd$ |
| 18. | *(* | $) / e$ | $fg - ab + cd -$ |
| 19. | * | $/ e$ | $fg - ab + cd - *$ |
| 20. | */ | e | $fg - ab + cd - *$ |
| 21. | */ | $-$ | $fg - ab + cd - *e$ |
| 22. | Empty | $-$ | $fg - ab + cd - *e/*$ |

∴ Postfix Expression = $fg - ab + cd - *e/*$

Program 2.4.2 : Program for conversion of infix to its postfix form operators supported '+,-,*,/,%,&,(,) operands supported - all single character operands

OR Write a program to convert an expression from infix to postfix using stack.

MU - May 14, Dec. 16, Dec. 17, Dec. 19, 10 Marks

```
/* 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("\n Enter an infix expression : ");
    gets(infix);
    infix_to_postfix(infix, postfix);
    printf("\n Postfix : %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 +

2.5 Expression Conversion (A Fast Method)

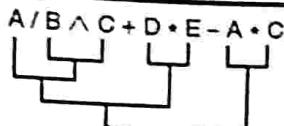
2.5.1 Infix to Postfix

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 :

$$(((A / (B \wedge C)) + (D * E)) - (A * C))$$

After moving operators to replace their corresponding right parentheses, we get :

$$ABC^ / DE^ * + AC^ * -$$

2.5.2 Algorithm to Check Well-Formedness of Parenthesis

An expression is said to well formed with respect to parenthesis :

- If every opening parenthesis has a closing parenthesis.
- If we count number of left parenthesis and right parenthesis then at no time, count of right parenthesis should exceed the count of left parenthesis.

Some well formed expression are given below :

$(())$

$(()) (())$

- A stack can be used to validate an expression using two simple rules. We scan the expression from left to right.
- If a left parenthesis is found, we perform a push operation on stack.
- If a right parenthesis is found, we perform a POP operation on stack.
- An extra right parenthesis will mean a POP operation on an empty stack.
- An extra left parenthesis will mean a non-empty stack at the end of algorithm.

Pseudo code

Validate(char expression[])

```

{
    stack S;
    initialize S;
    for each element x in expression[ ]
    {
        if(x is a '(');
            Push(S, '(');
        else
            if(x is a ')');
                if stack S is empty
                {
                    report "error";
                    return;
                }
                else
                    POP(S);
            }
        if stack is not empty
            report "error";
        else
            report "a well formed expression";
    }
}

```

Example 2.5.1 : Convert the following expression into other two forms.

(i) $((A-(B+C))^D) \$ (E+F)$ where

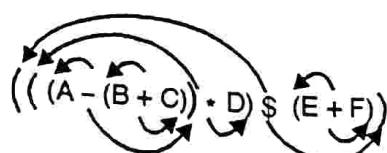
$\$ = \text{exponentiation}$

(ii) $\text{Imn} \$ q P \$ y \$ / - rs^ +$

Solution :

(i) Expression $((A-(B+C))^D) \$ (E+F)$ is in infix form.

Step 1 : After fully parenthesizing the expression, we get



Step 2 :

Expression can be converted to postfix form by moving operators to replace their corresponding right parentheses.

Postfix form : $ABC + - D^* EF + \$$

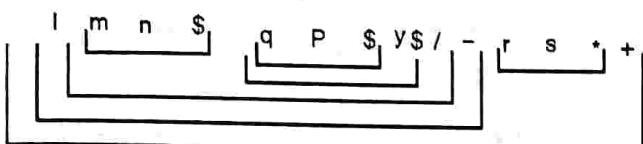


Expression can be converted to postfix form by moving operators to replace their corresponding left parentheses.

Prefix form : \$*-A+BCD +EF

(ii) Expression $lmn \ $ q P \$y\$/-rs^*+$ is in postfix form.

Step 1 : After grouping elements in the order of evaluation, we get



Step 2 : Expression can be converted to infix form by moving operators at the center of the group.

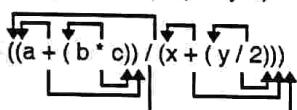
infix form : $l - m \$ n / q \$ P \$ y + r * s$

expression can be converted of prefix form by moving operators at the beginning of the group.

prefix form : $+ - l / \$mn\$qPy *rs$

Example 2.5.2 : Give the postfix and prefix forms of the infix expression given below.

$(a + b * c) / (x + y/2)$



Solution :

Step 1 : After fully parenthesizing the expression, we get

Step 2 : Expression can be converted to postfix form by moving operators to replace their corresponding right parentheses.

postfix form : $a b c * + x y 2 / + /$

Expression can be converted to prefix form by moving operators to replace their corresponding left parentheses.

prefix form : $/ + a * b c + x / y 2$

Example 2.5.3 : Explain the necessity of representing expression in prefix and postfix notion. For the given postfix expression, evaluate it for the values given. Show stepwise stack contents.

$ABC*DEF^G*-H^*+$

A = 6, B = 1, C = 4, D = 16, E = 2, F = 3, G = 2, H = 5, where $^$ = exponential operator.

Solution : Prefix and Postfix expressions are free from any precedence. They are more suited for mechanization.

| Input | Stack | Comment |
|-----------------|------------------------|-----------|
| ABC*DEF^G*-H^*+ | | Initially |
| BC*DEF^G*-H^*+ | 6 | PUSH A |
| C*DEF^G*-H^*+ | 1 6 | PUSH B |
| DEF^G*-H^*+ | 4 1 6 | PUSH C |
| DEF^G*-H^*+ | 4 6 | 4*1 |
| EF^G*-H^*+ | 16 4 6 | PUSH D |
| F^G*-H^*+ | 2 16 4 6 | PUSH E |
| ^G*-H^*+ | 3 2 16 4 6 | PUSH F |
| /G*-H^*+ | 8 16 4 6 | 2^3 |
| G*-H^*+ | 2 4 6 | 16/8 |



Input

 $* - H * +$

Stack

| |
|---|
| 2 |
| 2 |
| 4 |
| 6 |

Comment

PUSH G

 $- H * +$

| |
|---|
| 4 |
| 4 |
| 6 |

 $2 * 2$ $H * +$

| |
|---|
| 0 |
| 6 |

 $4 - 4$ $* +$

| |
|---|
| 5 |
| 0 |
| 6 |

PUSH H

 $+$

| |
|---|
| 0 |
| 6 |

 $5 * 0$

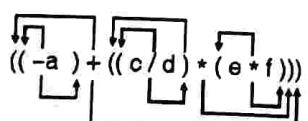
End

| |
|---|
| 6 |
|---|

 $6 + 0$ $\therefore \text{Value of the expression} = 6$ **Example 2.5.4 :** Consider an infix expression. $- a + (c/d)^*(e^f)$

Convert it to postfix and prefix form.

Evaluate the postfix expression for the given values.

 $a = 2, c = 4, d = 2, e = 3, f = 5$ **Solution :****(1) Conversion****Step 1 :** Fully parenthesizing the given infix expression, we get :**Step 2 :** Operators are moved to replace their corresponding right parentheses to convert the expression into postfix form.Postfix expression : $a - cd/ef^{**+}$ **Step 3 :** Expression is converted into prefix form by moving operators to replace their corresponding left parentheses.Prefix expression : $+ - a * /cd*ef$ **(2) Evaluation of postfix expression**

Expression (Input)

 $a - cd/ef^{**+}$

| |
|--|
| |
|--|

Initially

 $- cd/ef^{**+}$

| |
|---|
| 2 |
|---|

push

 cd/ef^{**+}

| |
|----|
| -2 |
|----|

It is a unary minus

 d/ef^{**+}

| |
|---|
| 4 |
|---|

Push

 $/ef^{**+}$

| |
|----|
| 2 |
| 4 |
| -2 |

Push

 ef^{**+}

| |
|---|
| 2 |
|---|

Perform 4/2

 f^{**+}

| |
|----|
| 3 |
| -2 |

Push

 $^{**+}$

| |
|----|
| 5 |
| 3 |
| -2 |

Push

 $^{*+}$

| |
|----|
| 5 |
| 3 |
| -2 |

 $^{*+}$

| |
|----|
| 15 |
| 2 |
| -2 |

Perform 5*3

 $^{*+}$

| |
|----|
| 15 |
| 2 |
| -2 |

 $^{*+}$

| |
|----|
| 30 |
| -2 |

Perform $15 * 2$

| Expression (Input) | Stack | Comment |
|---|-------|---------------------|
| End | 28 | Perform $30 + (-2)$ |
| \therefore Value of the expression = 28 | | |

- Example 2.5.5 :** Given the following Infix notations, find their equivalent Prefix and Postfix notations :
- $(A + B)^*C$
 - $(A - B) / (C - D)$
 - $((A + B)^* (C - D)) / E$
 - $(A + B^*C) / (X + Y/Z)$

Solution :

| Sr. No. | Infix | Prefix | Postfix |
|---------|---------------------------|---------------------|-------------------|
| (i) | $(A + B)^*C$ | $* + ABC$ | $AB + C *$ |
| (ii) | $(A - B) / (C - D)$ | $/ - AB - CD$ | $AB - CD - /$ |
| (iii) | $((A + B)^* (C - D)) / E$ | $/* + AB - CDE$ | $AB + CD - *E /$ |
| (iv) | $(A + B * C) / (X + Y/Z)$ | $/ + A * BC + X/YZ$ | $ABC **+ XYZ / +$ |

- Example 2.5.6 :** Given following infix expression, find their prefix and postfix notations.
- $a * b/c*d - e/f$
 - $(a+b)/(c+d)$
 - $a+b+c+d$

Solution :

| Sr. No. | Infix expression | Prefix | Postfix |
|---------|---------------------|----------------|----------------|
| (i) | $a*b/c*d-e/f$ | $-*/ *abcd/ef$ | $ab*c/d*ef/-$ |
| (ii) | $(a + b) / (c + d)$ | $/ + ab + cd$ | $ab + cd + /$ |
| (iii) | $a + b + c + d$ | $+++abcd$ | $ab + c + d +$ |

- Example 2.5.7 :** Convert the following expression in other two forms, where \$ stands for unary minus.
- $ab + cd - *$
 - $$a + (b - c) \uparrow D$
 - $/ - *abc + ef$
 - $$a + p \uparrow q \uparrow r$

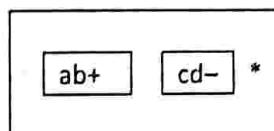
Solution :

(i) $ab + cd - *$

Given expression is in postfix form.

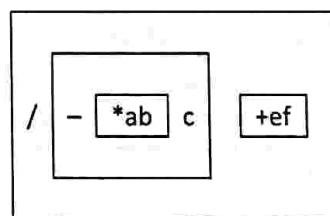
Elements can be grouped as shown below.

Grouping is done left to right and in the sequence of evaluation.



- Infix form : Operators are moved between the two operands of the group, parentheses is used wherever it is necessary.
 $(a + b)^*(c - d)$
- Prefix form : Operators are moved at the beginning of the group.
 $* + ab - cd$
- (ii) $\$a + (b - c) \uparrow D$
Given expression is in infix form.
Fully parenthesizing the expression, we get :
 $((\$a) + ((b - c) \uparrow D))$
- Postfix form (operator moved to its corresponding right parentheses) : $a\$bc - D \uparrow +$.
- Prefix form (operator moved to its corresponding left parentheses) : $+ \$a \uparrow - bcD$
- (iii) $/ - * abc + ef$

Given expression is in prefix form. Elements can be grouped as shown below. Grouping is done right to left.



- Postfix form (operator at the end of the group) :
 $ab * c - ef + /$
- Infix form (operator at the center of the group) :
 $(a * b - c) / (e + f)$
- (iv) $\$a + p \uparrow q \uparrow r$
Given expression is in infix form.
Fully parenthesizing the expression, we get :
 $((\$a) + ((p \uparrow q) \uparrow r))$

Data Structure (MU-Comp.)

- Postfix form (operator moved to its corresponding right parentheses) : $a \$ p q \uparrow r \uparrow +$
- Prefix form (operator moved to its corresponding left parentheses) : $+ \$ a \uparrow \uparrow p q r$.

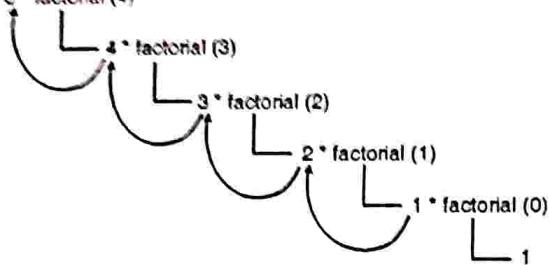
Recursion is a fundamental concept in mathematics. When a function is defined in terms of itself then it is called a recursive function. Consider the definition of factorial of a positive integer n .

$$\text{Factorial}(n) = \begin{cases} 1 & \text{if } (n=0) \\ n * \text{factorial}(n-1), & \text{otherwise} \end{cases}$$

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 :

$$\text{factorial}(5) = 5 * \text{factorial}(4)$$



Recursion expands when $n > 0$ and it starts winding up on hitting the base ($n = 0$).

2.6 Introduction to Recursion

MU - Dec. 14, May 15, May 17

University Questions

Q. What is recursion ? (Dec. 14, May 17, 2 Marks)

Q. Explain the term recursion with an example. (May 15, 5 Marks)

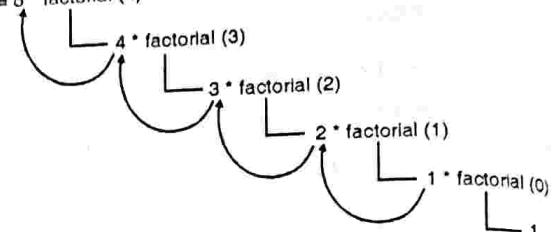
Recursion is a fundamental concept in mathematics. When a function is defined in terms of itself then it is called a recursive function. Consider the definition of factorial of a positive integer n .

$$\text{Factorial}(n) = \begin{cases} 1 & \text{if } (n=0) \\ n * \text{factorial}(n-1), & \text{otherwise} \end{cases}$$

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 :

$$\text{factorial}(5) = 5 * \text{factorial}(4)$$



Recursion expands when $n > 0$ and it starts winding up on hitting the base ($n = 0$).

2.7 Converting a Recursive Function to an Equivalent C-Function

2.7.1 Finding Factorial of an Integer Number

Recursive definition of factorial of an integer

$$f(n) = \begin{cases} 1 & \text{if } (n == 0) \\ n * f(n-1), & \text{otherwise} \end{cases}$$

'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. 2.7.1 to give a better insight into recursion.

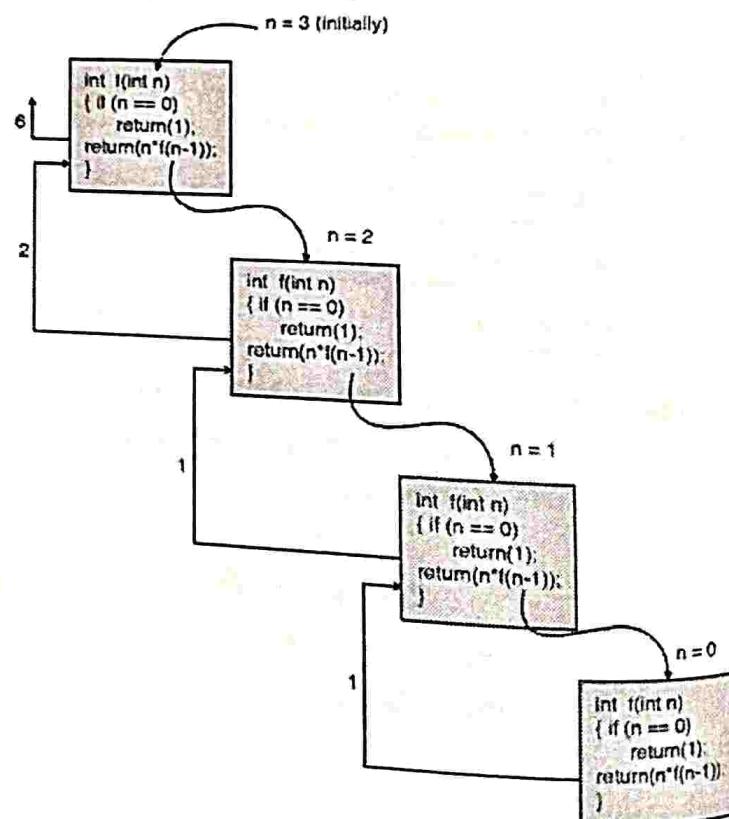


Fig. 2.7.1

Sequence of calls could also be shown through a recursion tree.

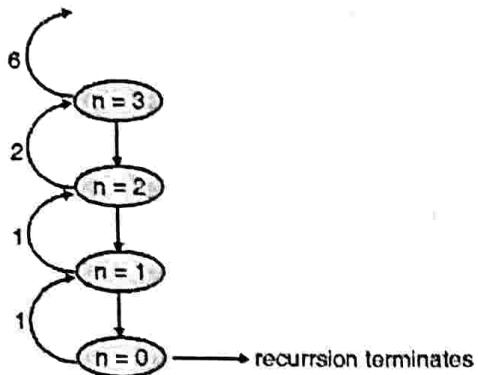


Fig. 2.7.2 : Recursion tree for finding factorial (3)

2.7.2 Finding n^{th} Term of Fibonacci Sequence Recursive Definition

$$T_n = T_{n-1} + T_{n-2} \quad \text{if } n > 1$$

$$T_1 = 1$$

$$T_0 = 0$$

'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 $\text{fib}(n) = \text{fib}(n - 1) + \text{fib}(n - 2)$ for $n > 1$ is a case of binary recursion. The first term on the right $\text{fib}(n - 1)$ is known as left recursion and the second term on the right $\text{fib}(n - 2)$ is known as right recursion.

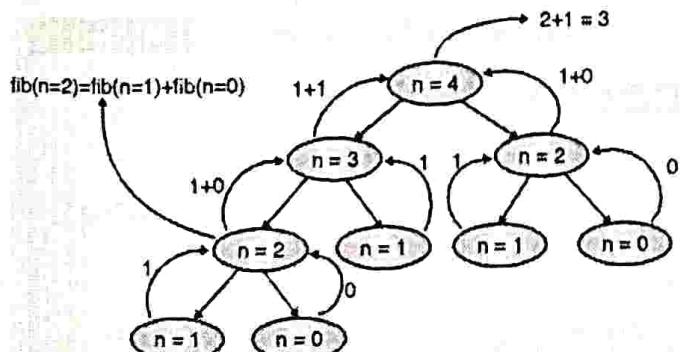


Fig. 2.7.3 : A recursion tree, showing calculation of T_4

Fibonacci series :

| | | | | |
|-------|-------|-------|-------|-------|
| 0 | 1 | 1 | 2 | 3 |
| | | | | |
| T_0 | T_1 | T_2 | T_3 | T_4 |

Program 2.7.1 : Program for finding factorial of a number through the use of recursive function.

```

#include<stdio.h>
#include<conio.h>
int fact(int);
void main()
{
    int x, n;
    printf("\nEnter 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

2.7.3 Finding GCD of given Numbers

MU - May 14, May 15

University Questions

- Q. Write recursive functions to calculate GCD of 2 numbers. (May 14, 5 Marks)
Q. Explain example of recursion. (May 15, 2 Marks)

Recursive definition of GCD of two numbers

$$\begin{aligned} f(x, y) &= y && \text{if } x \text{ is divisible by } y \\ &= f(y, x \% y) && \text{otherwise} \end{aligned}$$

Note : It is assumed that $x \geq y$



C function for finding GCD

```
int GCD(int x, int y)
{
    int temp;
    if(x < y)
        /* y interchange x and y if x < y */
    {
        temp = x;
        x = y;
    }
    if(x % y) == 0
        return(y);
    return(GCD(y, x % y));
}
```

2.7.4 Calculation of x^n using Recursion

Recursive Definition of x^n

$$\begin{aligned} f(x, n) &= 1 && \text{if}(n \text{ is equal to } 0) \\ &= x * f(x, n - 1) && \text{otherwise} \end{aligned}$$

C function for finding x^n

```
int Power(int x, int n)
{
    if(n == 0)
        return(1);
    return(x * (Power(x, n - 1)));
}
```

2.7.5 Calculation of Sum of Digits

MU - May 17

University Question

Q. Write a recursive function in 'C' to find sum of digits of a number. (May 17, 5 Marks)

C function for finding sum of digits of a number

```
int Sum(int x, int t)
{
    if(x == 0)
        return(0);
    t = t + x % 10;
    return(t);
}
```

2.8 Examples of Recursion

2.8.1 Finding Sum of the Elements Stored in an Array

$$\begin{aligned} \text{sum}(a, n) &= a[n-1] + \text{sum}(a, n-1) \dots \text{if } n > 1 \\ &= a[0] \dots \text{if } n = 1 \end{aligned}$$

Where, a = Array

n = Number of elements

$$\begin{aligned} \text{sum}(a[5, 9, 2, 6], 4) &= 6 + \text{sum}(a[5, 9, 2], 3) \\ &= 6 + 16 \end{aligned}$$

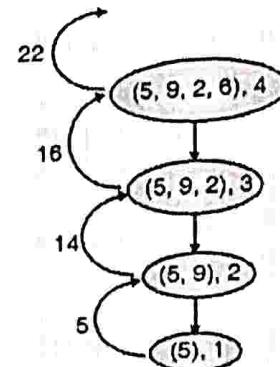


Fig. 2.8.1 : Recursion tree showing calculation of sum of 4 elements 5, 9, 2, 6

Definition of finding sum of array elements is declared in terms of sum itself.

- On the right hand side of the recursive definition $\text{sum}(a, n) = a(n-1) + \text{sum}(a, n-1)$, the value of n reduces by 1. Thus it has moved closer to the point of termination ($n = 1$) and the recursion will definitely terminate.
- Every recursion should have at least one case for termination. Recursion in the above example terminates when n reaches 1. Sum of the elements of an array having one element will be the element itself.

2.8.1(A) 'C' Function for Finding Sum of the Elements of an Array

MU - Dec. 14

University Question

Q. Write a 'C' program to calculate sum of 'n' natural numbers using recursion. (Dec. 14, 3 Marks)

```
int sum(int a[], int n)
{
    if(n == 1)
        return(a[0]);
```



```

else
    return(a[n - 1] + sum(a, n - 1));
}

```

2.8.2 Finding Length of a String

$\text{length}(s) = 0$ if $s[0] = '\text{O}'$
 $1 + \text{length}(s + 1)$ otherwise

Example :

$$\begin{aligned}
 \text{length}("xyz") &= 1 + \text{length}("yz") \\
 &= 1 + 1 + \text{length}("y") \\
 &= 2 + 1 + \text{length}("") = 3
 \end{aligned}$$

'C' function for finding length of a string

```

int length(char s[])
{
    if(s[0] == '\0')
        return(0);
    return(1 + length(s + 1));
}

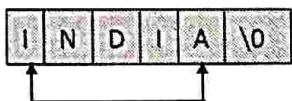
```

2.8.3 Reversing a String

$\text{Reverse}(s, i, j) = (\text{interchange}(s[i], s[j]) \text{ reverse}(s, i + 1, j - 1))$
 If $i < j$.

$s \rightarrow$ character array
 $i \rightarrow$ index of the first character
 $j \rightarrow$ index of the last character

Example :



interchange the characters 'I' and 'A' and then reverse the string "NDI".

'C' function for reversing a string

```

void reverse(char *s, int i, int j)
{
    char temp;
    if(i < j)
    {
        temp = s[i];
        s[i] = s[j];
        s[j] = temp;
    }
}

```

```

    reverse(s, i + 1, j - 1);
}
}

```

2.8.4 Searching a Number in an Array

$a \rightarrow$ array
 $i \rightarrow$ index of the starting element
 $j \rightarrow$ index of the last element
 $\text{key} \rightarrow$ element to be searched
 $\text{search}(a, i, j, \text{key}) = -1$ if $i > j$
 i if $a[i] == \text{key}$
 $\text{search}(a, i + 1, j, \text{key})$ otherwise

A return value of -1 indicates that the element is not found otherwise it gives the index of the location of occurrence of the search element.

'C' function for searching an element

```

int search(int *a, int i, int j, int key)
{
    if(i > j)
        return(-1);
    if(a[i] == key)
        return(i);
    return(search(a, i + 1, j, key));
}

```

2.8.5 Finding Largest Element in an Array

$\text{largest}(a, i, j) = a[i]$ if $(i = j)$

$\max(a[i], \text{largest}(a, i + 1, j))$ otherwise

'C' function for finding the largest element of an array

```

int largest(int *a, int i, int j)
{
    if(i == j)
        return(a[i]);
    if(a[i] > largest(a, i + 1, j))
        return(a[i]);
    return(largest(a, i + 1, j));
}

```

2.8.6 Binary Search

$\text{binsearch}(a, i, j, \text{key}) = -1$ if $(i > j)$

$$\frac{i+j}{2} \text{ Key} = a\left[\frac{i+j}{2}\right]$$



$\text{binsearch}(a, i, \frac{i+j}{2} - 1, \text{key}) \dots \text{key} < a[\frac{i+j}{2}]$

$\text{binsearch}(a, \frac{i+j}{2} + 1, j, \text{key}) \dots \text{key} > a[\frac{i+j}{2}]$

'C' function for binary search

```
int binsearch(int *a, int i, int j, int key)
{
    int k;
    k = (i + j)/2;
    if(i > j)
        return(-1);
    if(key < a[k])
        return(binsearch(a, i, k - 1, key));
    return(binsearch(a, k + 1, j, key));
}
```

2.8.7 Tower of Hanoi Problem

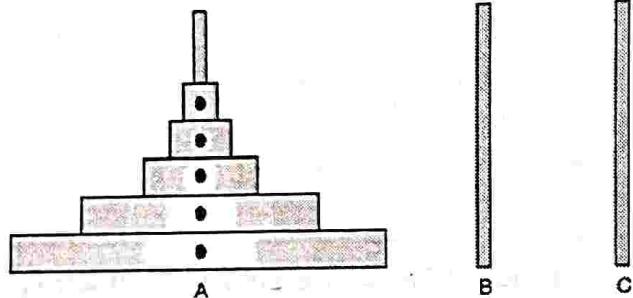


Fig. 2.8.2 : Disks are stacked on peg A in decreasing order of size

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 its solution 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.

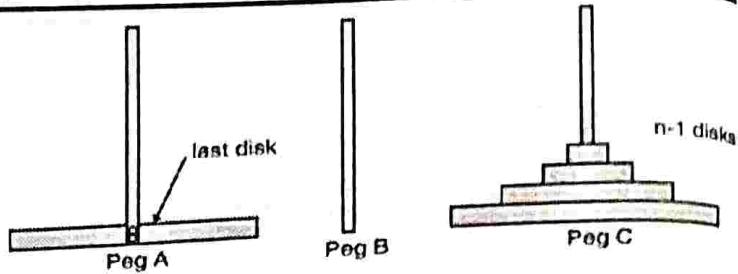


Fig. 2.8.3

Step 2 : Move the only disk from peg A to peg B.

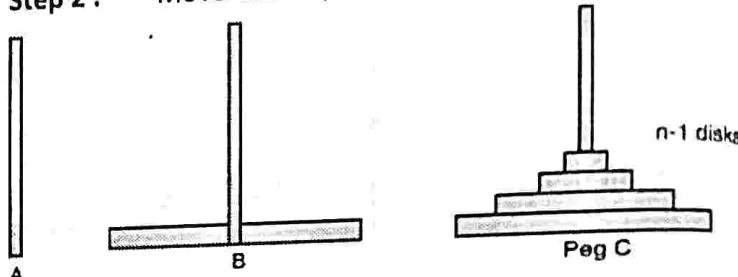


Fig. 2.8.4

Step 3 : Move the $n - 1$ disks from peg C to peg B.

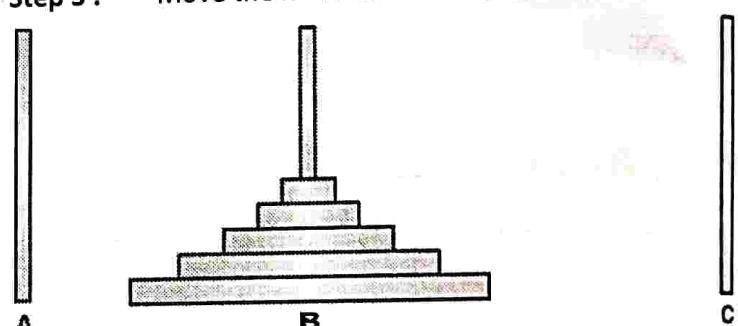


Fig. 2.8.5

Recursive definition

$$\text{ToH}(n, x, y, z) = \begin{cases} \text{ToH}(n-1, x, z, y) \\ \text{move a disk from } x \text{ to } y \\ \text{ToH}(n-1, z, y, x) \end{cases} \text{ if } n \geq 1$$

Where, n = Number of disks

x = Source peg

y = Destination peg

z = Empty peg to be used for transfer

- $\text{ToH}(n, x, y, z) = \text{transfer } n \text{ plates from peg } x \text{ to peg } y \text{ using peg } z.$
- $\text{ToH}(n-1, x, z, y) = \text{transfer } n-1 \text{ plates from peg } x \text{ to peg } z \text{ using peg } y.$
- $\text{ToH}(n-1, z, y, x) = \text{transfer } n-1 \text{ plates from } z \text{ to } y \text{ using peg } x.$

Program 2.8.1 : 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, z, y, x);
    }
}
```

Output

Enter number of plates:3

A -> B
A -> C
B -> C
A -> B
C -> A
C -> B
A -> B

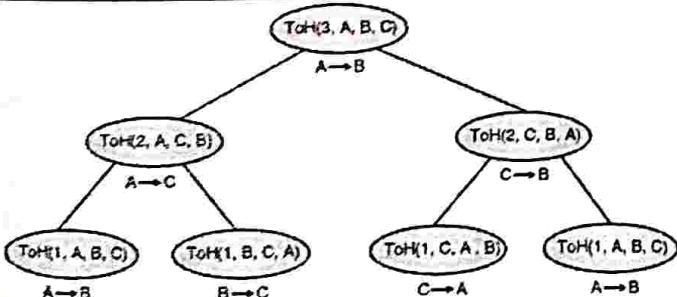


Fig. 2.8.6 : Recursion tree for the function ToH() for n = 3 plates

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”.

2.9 Solved Examples

Example 2.9.1 : Print a string in the reverse order using a recursive function.

Solution :

```
void print(char *S)
{
    if(*S! = '0')
    {
        print(S + 1);
        printf("%c", *S);
    }
}
```

Example 2.9.2 : Write a recursive function for the recursive definition given below :

Ackermann's function A (m, n) is defined as follows :

$$A(m, n) = \begin{cases} (n+1) & \text{if } m = 0 \\ A(m-1, 1) & \text{if } n = 0 \\ A(m-1, A(m, n-1)) & \text{otherwise} \end{cases}$$

Solution :

```
int A(int m, int n)
{
    int x, y;
    if(m == 0)
        return(n + 1);
    else
    {
        if(n == 0)
        {
            ...
        }
    }
}
```



```

        x = A(m - 1, 1);
        return(x);
    }
    else
    {
        x = A(m, n - 1);
        y = A(m - 1, x);
        return(y);
    }
}

```

Example 2.9.3 : Write a recursive function to compute the binomial coefficient.

$${}^n C_m \text{ where } {}^n C_0 = {}^n C_n = 1.$$

Solution :

$$\text{We know, } {}^n C_m = {}^{n-1} C_m + {}^{n-1} C_{m-1}$$

```

int binomial(int n, int m)
{
    int x;
    if(n == m || m == 0)
        return(1);
    x = binomial(n - 1, m) + binomial(n - 1, m - 1);
    return(x);
}

```

Example 2.9.4 : Write a recursive algorithm to read a string of digit characters and produce the value of integer they represent.

Solution :

Let us assume that string is stored in an array $a[]$ from location i to location j .

$$\begin{aligned} \text{value}(a, i, j) &= a[i] - 48 \dots \dots \dots \text{if } (i == j) \\ &= 10 * \text{value}(a, i, j-1) + a[j] - 48 \dots \dots \dots \text{otherwise} \end{aligned}$$

Example 2.9.6 : A recursive function f is shown below. What is the value of $f(5)$?

```

int f(int x)
{
    if(x < 2)
        return(1);
    else
        return f(x - 1) + f(x - 2);
}

```

```

int value(char *a, int i, int j);
void main()
{
    char s[50];
    int val, i, j;
    printf("\n enter a number :");
    gets(s);
    j = strlen(s) - 1;
    val = value(s, 0, j);
    printf("\n %d", val);
}

int value(char *a, int i, int j)
{
    if(i == j)
        return(a[i] - 48);
    return(10 * value(a, i, j - 1) + a[j] - 48);
}

```

Example 2.9.5 : Let a and b positive integers. Suppose a function Q is defined recursively as follows :

$$Q(m, n) = \begin{cases} 0 & \text{if } (a < b) \\ Q(a - b, b) + 1 & \text{if } b \leq a \end{cases}$$

- (i) Find the value of $Q(2, 3)$ and $Q(14, 3)$
- (ii) Find $Q(5861, 7)$

Solution :

$$\begin{aligned} (i) \quad Q(2, 3) &= 0 \text{ as } a < b \\ Q(14, 3) &= Q(14 - 3, 3) + 1 = Q(11, 3) + 1 \\ &= Q(8, 3) + 2 = Q(5, 3) + 3 \\ &= Q(2, 3) + 4 = 0 + 4 = 4 \\ (ii) \quad Q(5861, 7) &= Q(817 \times 7 + 2, 7) \\ &= Q(2, 7) + 817 = 0 + 817 = 817 \end{aligned}$$

Solution :

Recursion tree for $n = 5$ is given below.

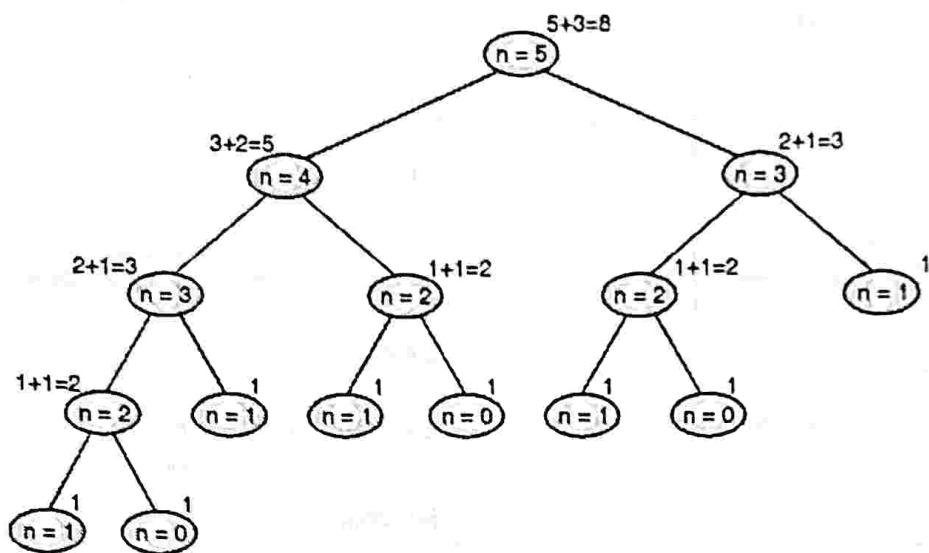


Fig. Ex. 2.9.6

Hence $f(5) = 8$

Example 2.9.7 : Write non-recursive version of the function factorial() using a stack.

Solution :

```

int factorial(int n)
{
    stack s;
    int val;
    init(&s);
    while(n > 0)
    {
        push(&s, n);
        n = n - 1;
    }
    val = 1;
    while(! empty(&s))
    {
        val = val * pop(&s);
    }
    return(val);
}
  
```

Simulation of the function factorial() for $n = 4$

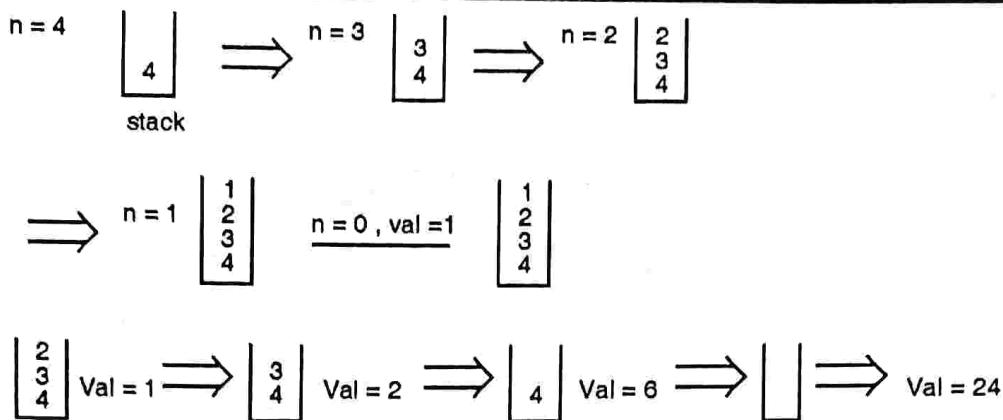


Fig. Ex. 2.9.7

Example 2.9.8 : What will be output of the following program ? Explain.

```
int f(int);
void main()
{
    int s;
    s = f(1234);
    printf("%d", s);
}
int f(int n)
{
    if(n == 0)
        return (0);
    else
        return (n%10+f(n/10));
}
```

Solution :

Recursion Tree

Function will find the sum of the digits of a number. It will display a value = 10.

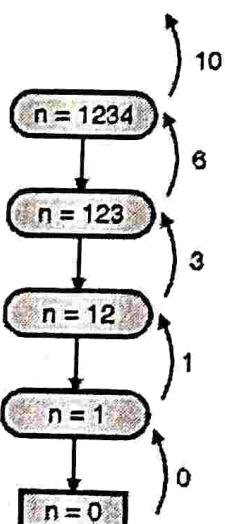


Fig. Ex. 2.9.8

Example 2.9.9 : What is the use of stack in function call ?

Solution :

Nested calls of functions are managed through stack. When a function is invoked, memory is allocated on stack for :

- Local variables
- Return address

On termination of execution of the function, memory is de-allocated. Details regarding the called Function are stored in a specific structure known as Activation record. An activation record consists of :

- Storage space for all local variables
- Definition of function
- Return address
- A pointer to activation record.

Activation records are stored in the stack. When a function is called, its activation record is pushed into the stack and control is transferred to the called Program. When the function execution completes, the control returns to the caller function. It obtains the return address from the return address field of the activation record.

Example 2.9.10 : Consider the following program :

```
void main()
{
    =
    A();
    =
}
A()
{ = }
```

```
B();
=
C();
}
```

Solution :

Stack operation for above program

| Action | Stack of activation records |
|------------------------------|-----------------------------|
| 1. Function main() is called | |
| 2. Function A() is called | |
| 3. Function B() is called | |
| 4. Function B() completes | |
| 5. Function C() is called | |
| 6. Function C() completes | |
| 7. Function A() completes | |
| 8. main() completes | |

2.10 Removal of Recursion

- Any recursive function can be converted to non-recursive function through use of a stack.
- A recursive call is similar to a call to another function.
- Any call to a function requires that the function has storage area where it can store its local variables and actual parameters.
- Return address must be saved before a call is made to a function.
- Storage area for local variables, actual parameters and the return address can be provided through a stack.
- In case of a recursive call, the value of local variables, parameters and the return address must be saved on the stack.
- While returning from a nested call, the previous outer call must be recalled with resetting all the local variables and operation must resume from where it was suspended.

Rules for converting a recursive algorithm to non-recursive one

Initialization

1. Declare stack – It will hold local variables, parameters, return address etc.
2. The first statement after the stack initialization must have a label.

Steps required to replace a recursive call

1. Push all local variables and parameters into the stack.
2. Push an integer “i” into the stack. It gives the return address.
3. Set the value of formal parameters.
4. Transfer the control to the beginning of the function (i.e. first label immediately after initialization of stack) using goto.
5. There should always be a label statement immediately following the recursive call. This label is the return address.

Steps required at the end of recursion function

1. If the stack is empty, then the recursion is finished.
2. Otherwise, pop the stack to restore the values of all local variables and parameters called by value.
3. Pop the return address.

2.11 Tail Recursion

Tail recursion refers to a recursive call at the last line. Tail recursion can be eliminated by changing the recursive call to a goto preceded by a set of assignments per function call. This simulates the recursive call because nothing needs to be saved after the recursive call finishes. We can just goto the top of the function with the values that would have been used in a recursive call.

Recursive function TOH with tail recursion

```
void TOH(int n, char x, char y, char z)
{
    if(n > 0)
    {
        TOH(n - 1, x, z, y);
        printf("%c %c → %c", x, y);
        TOH(n - 1, z, y, x); /* tail recursion */
    }
}
```

Without tail recursion

```
void TOH(int n, char x, char y, char z)
{
    char temp;
    label 0 :
    if(n > 0)
    {
        TOH(n - 1, x, z, y);
        printf("%c %c → %c", x, y);
        temp = x; x = z; z = temp;
        n = n - 1;
        goto label 0
    }
}
```

Example 2.11.1 : Explain use of stack in function call.

MU - May 16, 10 Marks

OR

Explain recursion as an application of stack with examples.

MU - May 16, 10 Marks

Solution :

Nested calls of functions are managed through stack. When a function is invoked, memory is allocated on stack for :

- Local variables
- Return address

On termination of execution of the function, memory is de-allocated. Details regarding the called functions are stored in a specific structure known as activation record. An activation record consists of :

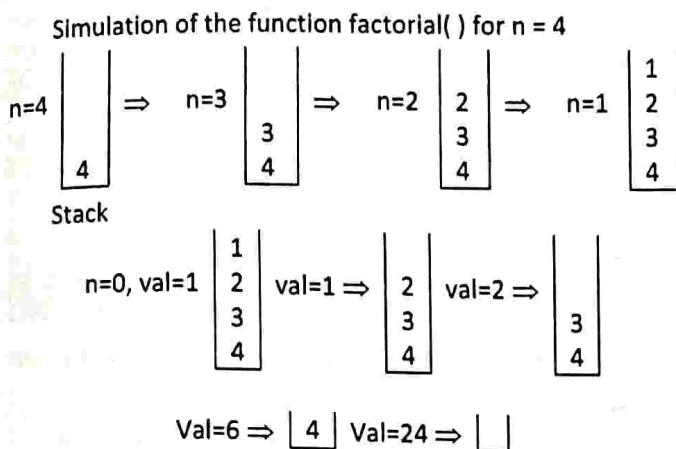
- Storage space for local variables
- Definition of function
- Return address
- A pointer to activation record.

Activation records are stored on the stack. When a function is called, its activation record is pushed into the stack and control is transferred to the called program. When the function execution completes, the control returns to the caller function. It obtains the return address from the return address field of the activation record.

Example 2.11.2 : Write non-recursive function for finding the factorial of a number using a stack.

Solution :

```
int factorial(int n)
{
    stack S;
    int val;
    init(&S);
    while(n > 0)
    {
        push(&S, n);
        n = n - 1;
    }
    val = 1;
    while(!empty(&S))
    {
        val = val * POP(&S);
    }
    return(val);
}
```



Example 2.11.3 : Give the stack operation for the following program.

```
void main()
{
```

```
-  
-  
-  
A();
```

```
-  
-  
-  
-  
}
```

```
A()  
{  
-  
-  
-  
B();  
-  
-  
-  
C();  
}
```

Solution :

| Step | Action | Stack of activation records |
|------|----------------------------|-----------------------------|
| 1. | Function main() is called | main() |
| 2. | Function A() is called | A() main() |

| Step | Action | Stack of activation records |
|------|-------------------------|-----------------------------|
| 3. | Function B() is called | B() A() main() |
| 4. | Function B() completes | A() main() |
| 5. | Function C() is called | C() A() main() |
| 6. | Function C() completes | A() main() |
| 7. | Function A() completes | main() |
| 8. | main() completes | |

2.12 Array and Linked Representation and Implementation of Queues

MU - Dec. 14, May 17

University Question

Q. Give ADT for the queue data structure.

(Dec. 14, May 17, 5 Marks)

2.12.1 Definition

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. 2.12.1 : Insertion of elements is done at the rear end and deletion from the front end

2.12.2 Application of Queues

MU - Dec. 14, May 17.

University Question

Q. Discuss in brief any two applications of Queue data structure. (Dec. 14, May 17, 5 Marks)

- 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.
- Various application software using non-linear data structure tree or graph requires a queue for breadth first traversal.
- 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

2.12.3 Array Representation and Implementation of Queues

MU - Dec. 15

University Question

Q. Explain Queue as ADT. (Dec. 15, 5 Marks)

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.

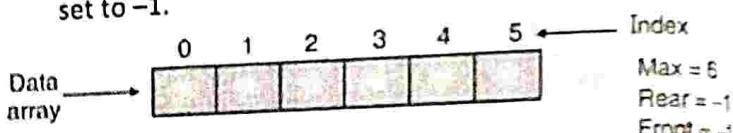


Fig. 2.12.2 : An empty queue after Initialization

- Fig. 2.12.3 shows the status of an queue after insertion of the element '5'.

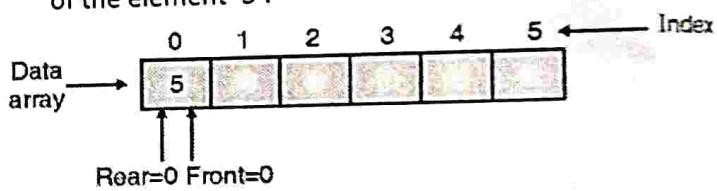


Fig. 2.12.3 : A queue after insertion of the first element '5'

- On subsequent insertions, front remains at the same place, where rear advances.

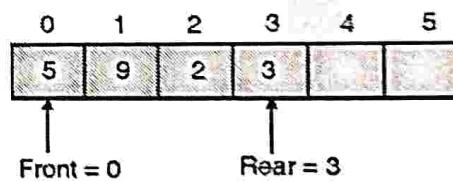


Fig. 2.12.4 : A queue after insertion of four elements

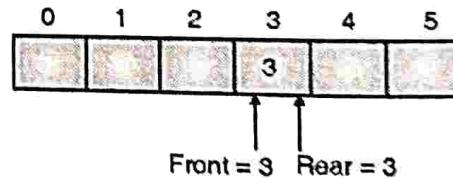


Fig. 2.12.5 : A queue after 3 successive deletions

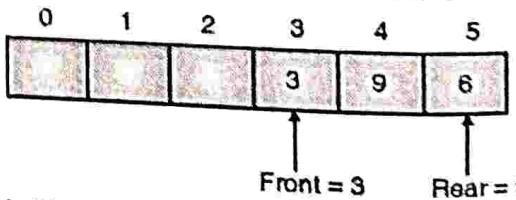
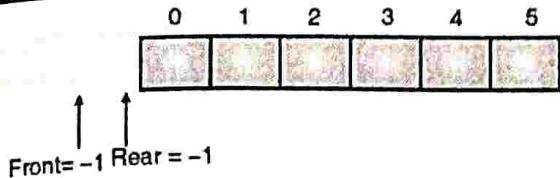
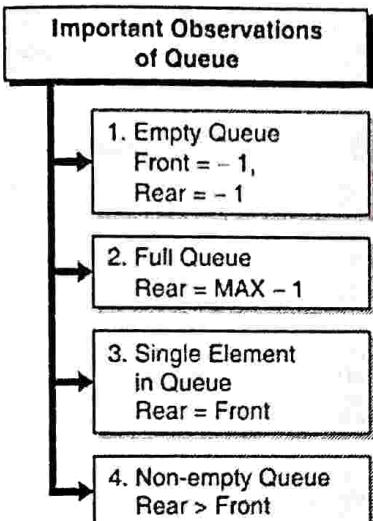


Fig. 2.12.6 : A queue after 2 successive Insertions

**Fig. 2.12.7 : A queue after 3 successive deletions**

Following points can be observed : Following points can be observed :

**Fig. 2.12.8**

- (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 :

Note : Problem of overflow : Refer to the Fig. 2.12.6, the queue has become full as rear = MAX - 1. There are three vacant spaces (location 0, 1, 2) but these spaces can not 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.

2.13 Operations on Queue

2.13.1 Operations on Queue Implemented using Array

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 return 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 q1, q2;
```

q₁ and q₂ are queue type variables

- (d) Declaring a queue type pointer variable.

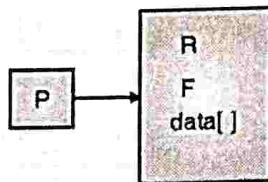
```
Q q1, *P;
```

```
P = &q1
```

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. 2.13.1 : Elements of the structure 'Q' can be accessed through the pointer P
($P \rightarrow R$, $P \rightarrow F$, $P \rightarrow \text{data}[]$)**

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. 2.13.1. 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, $P \rightarrow R$ (i.e. rear field (R) of the queue type variable whose address is stored in P) or $P \rightarrow F$ 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);
}
    
```

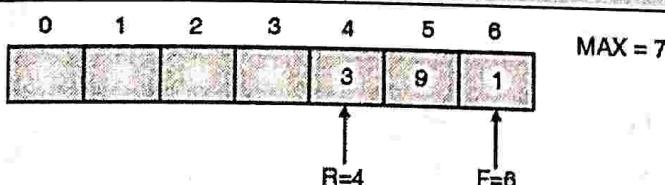


Fig. 2.13.2 : The queue is full as there is no space left for further insertion

enqueue()

Algorithm for Insertion In a queue

- Inserting in an empty queue.
(a) $R = F = 0;$

- (b) $\text{data}[R] = x$ /* x is the element to be inserted */
/* Both R and F will point to the only element of the queue */

- Inserting in a non-empty queue.

- (a) $R = R + 1;$
 $\text{data}[R] = x$
/* 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.

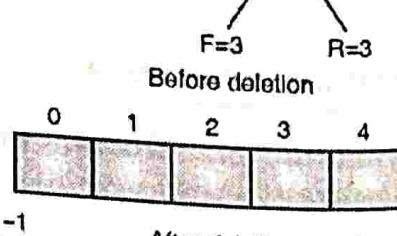
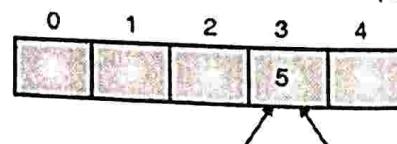
```

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;
    }
}
    
```

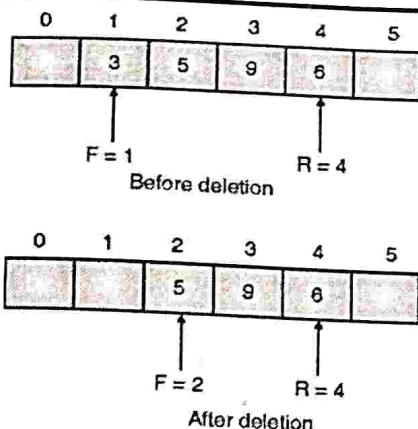
dequeue()

Algorithm for deletion from a queue :

- Deletion of the last element or only element($R = F$)
(a) $x = \text{data}[F]$ (b) $R = F = -1$; (c) $\text{return}(x)$
- Deletion of the element when the queue length > 1
(a) $x = \text{data}[F]$ (b) $F = F + 1$ (c) $\text{return}(x)$



(a) Deletion of the last element



(b) Deletion of an element from the queue when the queue length > 1

Fig. 2.13.3

Note : Deletion of an element from the queue should be carried out after checking that it is not empty.

Program 2.13.1 : A sample program for queue.

Show the status of the queue after every operation.

(a) Insert 5 elements

(b) Delete 2 elements

OR

Write a program in 'C' to implement linear queue using array.

MU - Dec. 17, May 19, 10 Marks

```
#include<conio.h>
#include<stdio.h>
#define MAX 10
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 x, i;
    initialise(&q);
```

```
printf("\n Enter 5 elements :");
for(i = 1; i <= 5; i++)
{
    scanf("%d", &x);
    if(!full(&q))
        enqueue(&q, x);
    else
    {
        printf("\n Queue is full .....exiting");
        exit(0);
    }
}
print(&q);
for(i = 1; i <= 2; i++)
{
    if(!empty(&q))
        x = dequeue(&q);
    else
    {
        printf("\n cannot delete...Queue is empty");
        exit(0);
    }
}
print(&q);
}
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;
        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

```

Enter 5 elements : 5 4 3 2 1
5 4 3 2 1
3 2 1

```

Example 2.13.1 : 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 one letter.

Solution :

Initial queue :

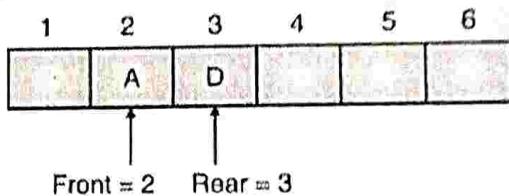


Fig. Ex. 2.13.1(a)

- (i) Add 'S'

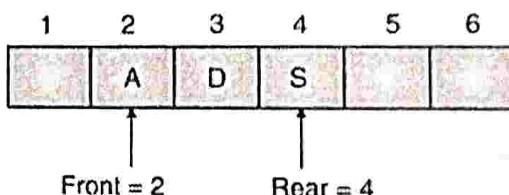


Fig. Ex. 2.13.1(b)

- (ii) Add 'J'

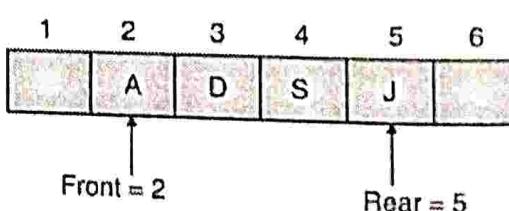


Fig. Ex. 2.13.1(c)

- (iii) Delete two letters

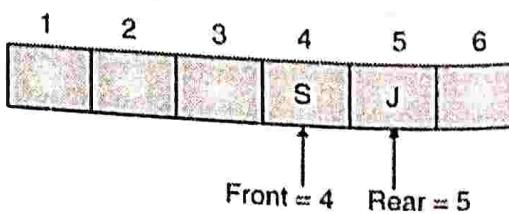


Fig. Ex. 2.13.1(d)

- (iv) Shift towards left to bring all free spaces to the right.

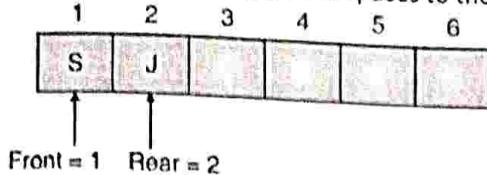


Fig. Ex. 2.13.1(e)

- (v) Insert M, H, I and delete one letter.

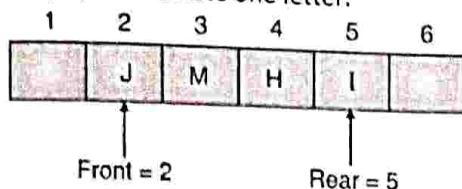


Fig. Ex. 2.13.1(f)

2.14 Circular Queues

MU - May 15, Dec. 16

University Questions

- Q. What is circular queue ? (May 15, 7 Marks)
 Q. Explain circular Queue. (Dec. 16, 5 Marks)

2.14.1 Queue using a Circular Array

(Advantage of circular queue)

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.

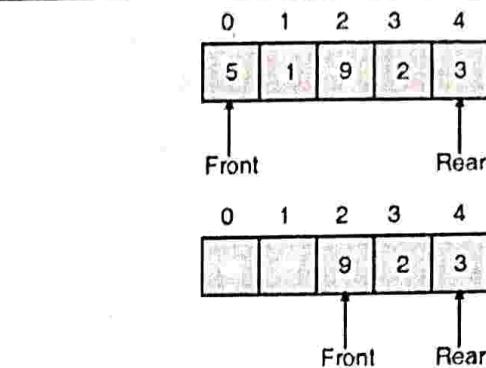


Fig. 2.14.1 : Queue is full, although locations 0 and 1 are vacant

- After insertion of five elements in the array (a queue) as shown in Fig. 2.14.1,

$$\left. \begin{array}{l} \text{rear} = 4 \\ \text{front} = 0 \end{array} \right\} \text{queue is full}$$

- 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. 2.14.1 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.

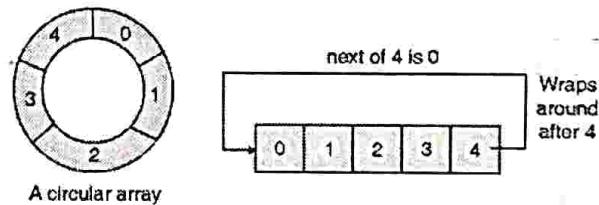


Fig. 2.14.2 : A circular array

In a circular array, the queue is found somewhere around the circle in consecutive positions.

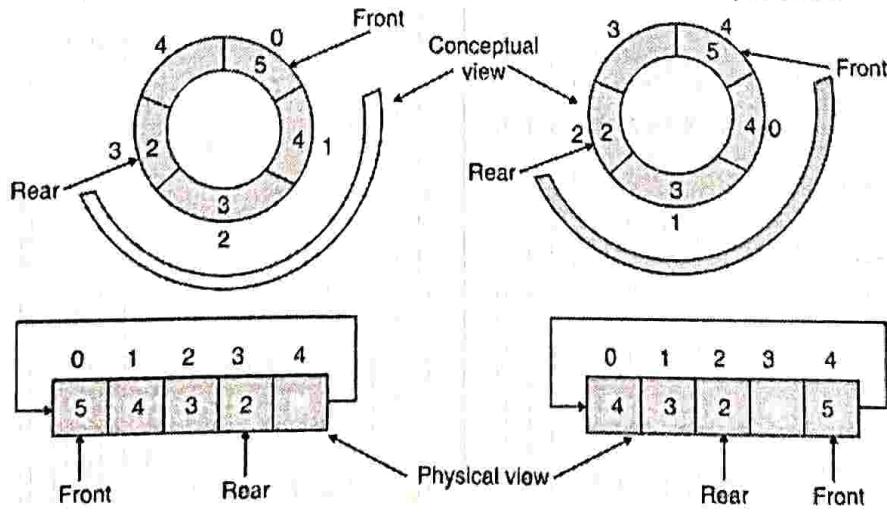


Fig. 2.14.3 : Queue at various places in an array



2.14.1(A) Implementation of a Circular Movement Inside a Linear Array

- 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

$i = (i + 1) \% 5 \rightarrow$ Array of size 5
 $i = (i + 1) \% \text{MAX} \rightarrow$ Array of size MAX expression used for circular movement

$$\begin{array}{rcl} i & (i + 1) \% 5 \\ \hline 0 & (0 + 1) \% 5 & = 1 \\ 1 & (1 + 1) \% 5 & = 2 \\ 2 & (2 + 1) \% 5 & = 3 \\ 3 & (3 + 1) \% 5 & = 4 \\ 4 & (4 + 1) \% 5 & = 0 \leftarrow i \text{ wraps around to } 0 \end{array}$$

$i = 0$
while(1)
 $i = (i + 1) \% 5;$

Above program segment will continue
 \leftarrow 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 operation 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. 2.14.4 shows, queue is full.

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. $(P \rightarrow \text{rear} + 1) \% \text{MAX} == P \rightarrow \text{front}$

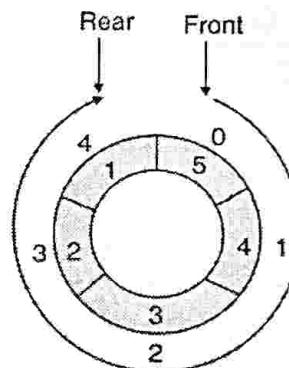


Fig. 2.14.4

```
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]);
}

```

Program 2.14.1 : Showing various operations on a circular queue.

OR Write a function in C to Insert delete and display elements in circular Queue.

OR Write a program in 'C' to implement a circular queue. The following operations should be performed by the program :

- (i) Creating the queue
- (ii) Deleting from the queue
- (iii) Inserting in the queue
- (iv) Displaying all the elements of the queue.

MU - Dec. 13; Déc. 15, 12 Marks

OR Write a program in 'C' to implement a circular queue

MU - May 15, 5 Marks

OR

Write a menu driven program in 'C' to implement QUEUE ADT. The program should perform the following operations.

- (I) Inserting element in the beginning
- (II) Deleting an Element from the Queue
- (III) Displaying the Queue
- (IV) Exiting the program

MU - Dec. 16, 12 Marks

OR

Write a program in 'C' to implement Circular Queue using arrays.

MU - May 18, 10 Marks

```

#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("\n 1)create \n 2)insert \n 3)delete \n
4)print \n 5)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 2.14.2 : Consider a circular queue of size 5 having initial status as :

| Front | Rear | Circular queue | | | | |
|-------|------|----------------|---|---|---|---|
| | | 0 | 1 | 2 | 3 | 4 |
| 1 | 2 | P | Q | | | |

Show the value of front, rear and the contents of circular queue after every step in tabular form for the following operations :

- (1) R is added
- (2) Delete 2 letters
- (3) S, T, U are added
- (4) Three letters are deleted
- (5) V is added

Solution :

| Operation | Front | Rear | Circular queue |
|---------------------------|-------|------|----------------------|
| Initial | 1 | 2 | 0 1 2 3 4 P Q |
| R is added | 1 | 3 | 0 1 2 3 4 P Q R |
| Delete 2 letters | 3 | 3 | 0 1 2 3 4 R |
| S,T,U are added | 3 | 1 | 0 1 2 3 4 T U R S |
| Three letters are deleted | 1 | 1 | 0 1 2 3 4 U |
| V is added | 1 | 2 | 0 1 2 3 4 U V |

Example 2.14.3 : Write necessary functions in C to implement circular queue in an array.
Assume
rear = front = 0, initially.

Solution :

Structure used for queue

```

#define MAX 30
typedef struct queue
{
    int data[MAX];
    int rear, front;
} queue;

```

Functions

```

void initialize(queue *p)
{
    p->rear = p->front = 0;
}

int empty(queue *p)
{
    if(p->rear == p->front)
        return(1);
    return(0);
}

int full(queue *p)
{
    if((p->rear + 1) % MAX == p->front)
        return(1);
    return(0);
}

void insert(queue *p, int x)
{
    if(full(p))
        printf("\n overflow:");
    else
    {
        p->rear = (p->rear + 1) % MAX;
        p->data[p->rear] = x;
    }
}

int Delete(queue *p)
{
    if(empty(p))
        printf("\n underflow:");
}

```

```

else
{
    p->front = (p->front + 1) % MAX;
    return(p->data[p->front]);
}
}

```

In the above scheme, front will always point one position counterclockwise from the first element in the queue. An empty condition is indicated by front = rear. Initially, front = rear = 0

In order to add an element, it is necessary to move rear one position clockwise i.e.

$$\text{rear} = (\text{rear} + 1) \% \text{MAX}$$

Similarly, it will be necessary to move front one position clockwise each time a deletion is made.

i.e. $\text{front} = (\text{front} + 1) \% \text{MAX}$

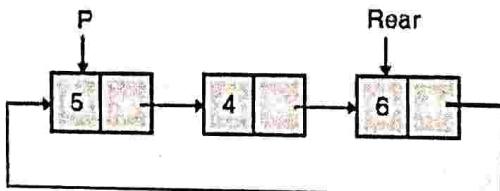
'C' function for insertion of an element in a queue represented using a circular linked list.

```

void enqueue(node **R, int x)
//queue is referenced by address of rear node.
{
    node *P;
    P = (node *) malloc(sizeof(node));
    P-> data = x;
    if(*R == NULL)
    {
        P-> next = P; *R = P;
    }
    else
    {
        P-> next = (*R)-> next;
        (*R)-> next = P;
        *R = P;
    }
}

```

Steps for deletion of an element from a queue represented using a circular linked list.



- (a) $P = \text{rear} \rightarrow \text{next};$
P Point to the node to be deleted
- (b) $x = P \rightarrow \text{data};$
if (last node is being deleted)
{
- (a) release the memory of the node
being deleted.
`free(P);`
- (b) Make the queue empty
`rear = NULL;`
- (c) Return the value stored in the
front node
`return(x);`

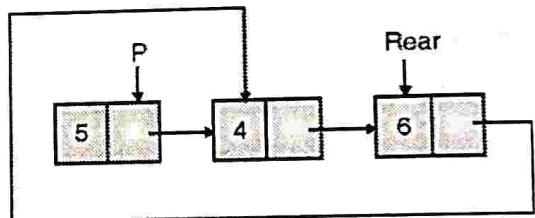
}

else

{

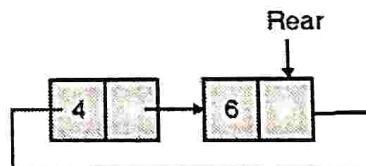
- (c) Remove the front node from the queue

$$\text{rear} \rightarrow \text{next} = P \rightarrow \text{next}$$



- (d) release the memory of the node being deleted

$$\text{free}(P)$$



- (e) return the value stored in the front node

$$\text{return}(x)$$

}

'C' function for deletion of the front node from a queue represented using a circular linked list.

```

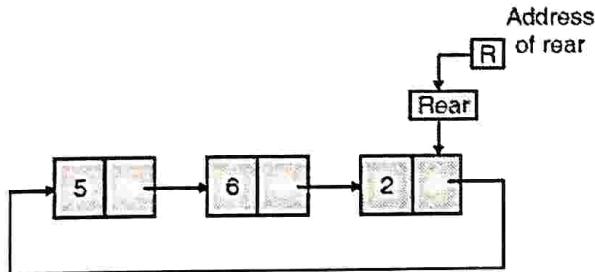
int dequeue(node **R)
//pointer rear is passed by reference
{
    node *P;
    int x;
    P = (*R)-> next;
}

```

```

/* 'R' is same as 'rear' as R contains the address
of rear*/
x = P -> data;
if(P -> next == P)
/* deleting the last node */
{
    *R = NULL;
    free(P);
    return(x);
}
(*R) -> next = P -> next;
free(P);
return(x);
}

```



In the above function, rear is passed by reference as after deletion, rear may change. Receiving variable in function int dequeue(node **R) is declared as node **R. R contains the address of rear and hence, *R can be used in place of 'rear'.

'C' function for printing elements of a queue represented using a circular linked list.

```

void Print(node * rear)
{
    node P;
    P = rear -> next;
    /* start printing from the front */
    do
    {
        printf("\n %d", P -> data);
        P = P -> next;
    } while(P != rear -> next);
}

```

In case of a circular linked list, the starting case and the termination case for the loop used for traversal of the linked list are identical.

- (1) We start printing from the front node.
 - (2) We terminate printing on reaching the front node.
Such cases are best handled through do-while loops.
- Program 2.14.4 :** Program for showing various operations on a queue represented using circular linked list.

```

#include<stdio.h>
#include<conio.h>
typedef struct node
{
    int data;
    struct node *next;
}node;
void init(node **R);
void enqueue(node **R, int x);
int dequeue(node **R);
int empty(node *rear);
void print(node *rear);
void main()
{
    int x, option;
    int n = 0, i;
    node *rear;
    init(&rear);
    clrscr();
    do
    {
        printf("\n 1. Insert\n 2. Delete\n 3. Print\n 4.
Quit");
        printf("\n your option: ");
        scanf("%d", &option);
        switch(option)
        {
            case 1 :
                printf("\n Element to be inserted");
                scanf("%d", &n);
                for(i = 0; i < n; i++)
                {
                    scanf("\n %d", &x);
                    enqueue(&rear, x);
                }
        }
    } while(option != 4);
}

```



```

    }
    break;
case 2 : if(!empty(rear))
{
    x = dequeue(&rear);
    printf("\n Element deleted = %d", x);
}
else
    printf("\n Underflow..... Cannot deleted");
break;
case 3 : print(rear);
break;
}
} while(option != 4);
getch();
}

void init(node **R)
{
    *R = NULL;
}

void enqueue(node **R, int x)
{
    node *p;
    p = (node *)malloc(sizeof(node));
    p->data = x;
    if(empty(*R))
    {
        p->next = p;
        *R = p;
    }
    else
    {
        p->next = (*R)->next;
        (*R)->next = p;
        (*R) = p;
    }
}

int dequeue(node **R)
{
    int x;
}

```

```

node *p;
p = (*R)->next;
p->data = x;
if(p->next == p)
{
    *R = NULL;
    free(p);
    return(x);
}
(*R)->next = p->next;
free(p);
return(x);
}

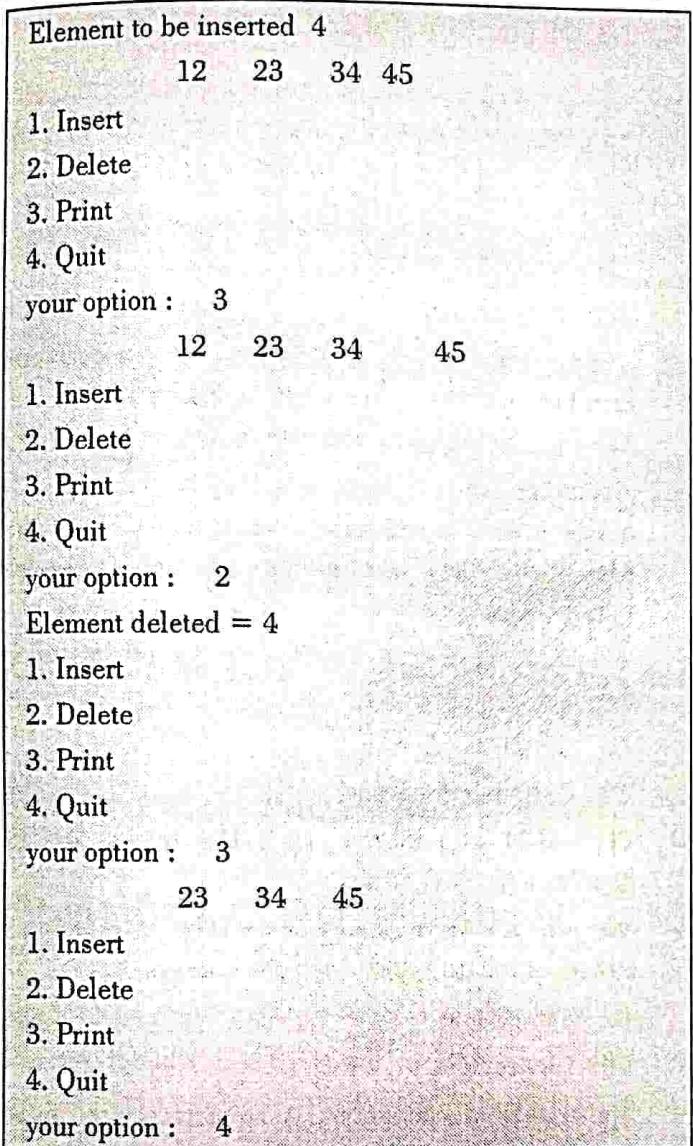
void print(node *rear)
{
    node *p;
    if(!empty(rear))
    {
        p = rear->next;
    }
    p = p->next;
    do
    {
        printf("\n %d", p->data);
        p = p->next;
    }while(p != rear->next);
}

int empty(node *P)
{
    if(P->next == -1)
        return(1);
    return(0);
}

```

Output

1. Insert
 2. Delete
 3. Print
 4. Quit
- your option : 1



2.15 Applications of Queue

Queue is a very useful data structure. Various features of operating system are implemented using a queue.

- Scheduling of processes (Round Robin Algorithm)
- Spooling (to maintain a queue of jobs to be printed)
- A queue of client processes waiting to receive the service from the server process.
- Various application software using non-linear data structure tree or graph requires a queue for breadth first traversal.
- 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 :

(1) Josephus problem (2) Job scheduling

(3) Queue simulation

2.15.1 Categorizing Data

The queue can be used to categorize the data. One example of categorizing data is college library. A typical college has several departments such civil engineering, mechanical engineering, computer engineering and so on. A library has various sections in which the books from each stream are arranged. These sections are like multiple queues in which appropriate data (books) are stored. Thus categorization of data can be possible using multiple queues. These multiple queues can be represented in a single array.

2.15.2 Job Scheduling

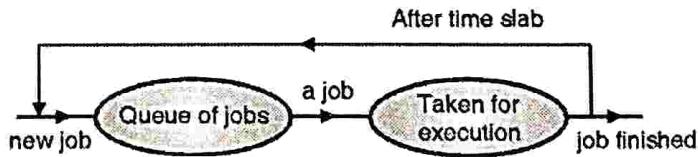
Programs (jobs) entering a computer for execution are scheduled using some strategies. The purpose is to improve :

- CPU utilization
- Response time
- 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



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 2.15.1 shows the jobs and the required run time.

Table 2.15.1

| Job no. | Run time | Quantum of time = 1 unit |
|---------|----------|-----------------------------|
| 1 | 2 units | |
| 2 | 1 unit | |
| 3 | 2 units | |

Initial status of queue

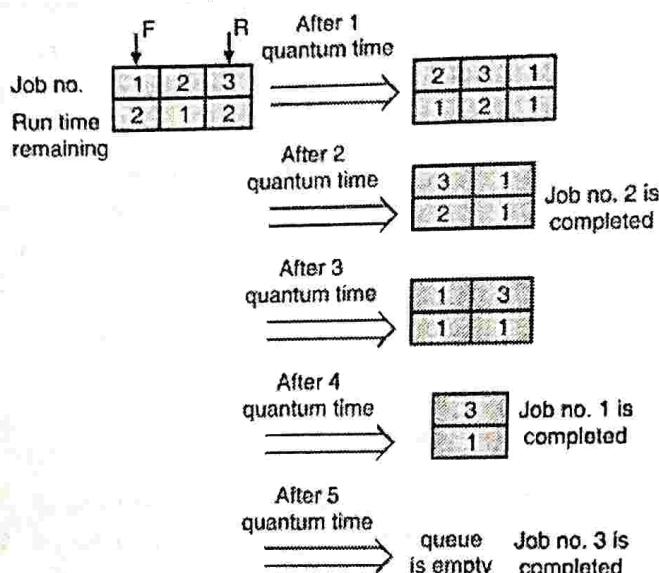


Fig. 2.15.1

```
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;
    }
}
```

2.15.3 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).

2.16 Priority Queue

MU - May 19

University Question

Q. Explain Priority Queue with example.

(May 19, 5 Marks)

- 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" basis 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 application 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.

2.16.1 Implementation of Priority Queues

Priority queue can be implemented both using :

(a) Linked list or (b) 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).

2.16.1(A) Implementation of a Priority Queue using a Circular Array

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;
```

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.

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

```
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;
                /* anticlock wise 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 2.16.1 : Program showing various operations on a priority queue.

OR

OR

Write a 'C' program to implement a priority queue. MU - Dec. 14, 8 Marks

Write a 'C' program to implement priority queue using arrays. The program should perform the following operations:

- Inserting in a priority queue.
- Deletion from a queue.
- Displaying contents of the queue. MU - Dec. 18, 12 Marks

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

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

```

#define MAX 30
typedef struct pqueue
{
    int data [MAX];
    int rear, front;
} pqueue;

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("\n 1)create\n 2)insert\n 3)Delete\n
        4)print\n 5)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);
                }
        }
    } while (op != 5);
}

```

```

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;
            /* anticlock wise 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);
}

```

Example 2.16.1 : What do you mean by priority queue?

Explain any one application of priority queue with suitable Example.

Solution : 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 basis but rather each element has a priority based on the urgency of need.

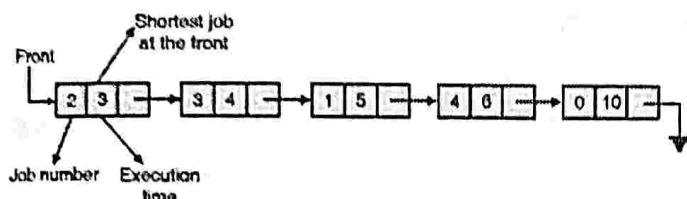
- An element with higher priority is processed before other elements with lower priority.

- Elements with the same priority are processed on first-come-first-served basis.
- An example of priority queue is hospital waiting room. A patient having a more fatal problem will be admitted before other patients.

Other application of priority queue 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. Let us consider a list of jobs with the execution-time requirements as given below :

| Job No | 0 | 1 | 2 | 3 | 4 |
|----------------|----|---|---|---|---|
| Execution time | 10 | 5 | 3 | 4 | 6 |

Job numbers 0 to 4, enter a priority queue as they arrive. Shortest job will have the highest priority and it will be serviced first.



- Job no. 2 will be completed in 3 time units.
- Job no. 3 will require additional 4 time units, it will be completed in 7 time units.
- Job no. 1 will require additional 5 units of time, it will be completed in 12 time units.
- Job no. 4 will require additional 6 units of time, it will be completed in 18 time units.
- Job no. 0 will require additional 10 units of time, it will be completed in 28 time units.

Structure of node

```

typedef struct node
{
    int time;
    int jobno;
    struct node *next;
} node;

```

Pseudo code for calculation of turnaround time

```

Priority queue q; /* q is a priority queue */
initialize q;
Read n /* enter no. of jobs */
for(i = 0; i < n; i++)

```

```

{
    read execution_time, jobno;
    insert(&q, execution_time, jobno);
}

elapsed_time = 0;
while(!empty(q))
{
    elapsed_time = elapsed_time + time(front node);
    print jobno(front node), elapsed_time;
    delete(&q);
}

```

Example 2.16.2 : Compare stacks and queues.

Solution : Comparison of stack and queues

| Sr. No. | Stack | Queue |
|---------|---|---|
| 1. | Stack is last in first out (LIFO) i.e. which is entered last will be retrieved firstly. | Queue is first out (FIFO) i.e. which is entered first will be served first. |
| 2. | Stack is Linear data structure which follows LIFO. | Queue is Linear data structure which follows FIFO. |
| 3. | Insertion and deletions are possible through one end called top. | Insertions are at the rare end and deletions are from the front end in a queue. |
| 4. | Example : Books in library. | Example : Cinema ticket counter. |

2.16.2 Dequeues

MU - May 16, Dec. 16, May 18, Dec. 19

University Questions

Q. Explain Double ended queue.

(May 16, Dec. 19, 4 Marks)

Q. Explain Double ended queue with example.

(Dec. 16, May 18, 10 Marks)

The word dequeue is a short form of double ended queue. It is general representation of both stack and queue and can be used as stack and queue. In a dequeue, insertion as well deletion can be carried out either at the rear end or the front end. In practice, it becomes necessary to fix the type of operation to be performed on front and rear end. Dequeue can be classified into two types :



Input restricted Dequeue

The following operations are possible in an input restricted dequeue :

- Insertion of an element at the rear end
- Deletion of an element from front end
- Deletion of an element from rear end

Output restricted Dequeue

The following operations are possible in an output restricted dequeue.

- Deletion of an element from front end
- Insertion of an element at the rear end
- Insertion of an element at the front end

There are various methods to implement a dequeue.

- Using a circular array
- Using a singly linked list.
- Using a singly circular linked list.
- Using a doubly linked list.
- Using a doubly circular linked list.

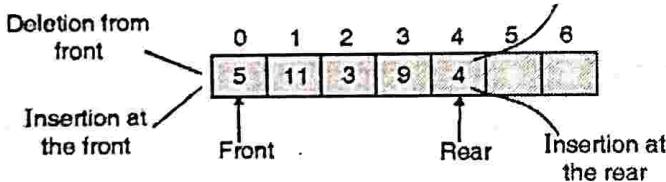


Fig. 2.16.1 : A dequeue in a circular array

Operations associated with dequeue :

- empty() : Whether the queue is empty ?
- full() : Whether the queue is full ?
- initialize() : Make the queue empty
- enqueueR() : Add item at the rear end of the queue.
- enqueueF() : Add item at the front end of the queue.
- dequeueR() : Delete item from the rear end of the queue.
- dequeueF() : Delete item from the front end of the queue.

Timing complexity of various dequeue operations :

enqueue R() - O(1) - constant time.

enqueue F() - O(1) - constant time.

dequeue R() - O(1) - constant time.

dequeue F() - O(1) - constant time.

Advantage of dequeue : The dequeue is a general representation of both stack and queue it can be used both as stack or a queue.

Dequeue as an ADT

Data type for dequeue in an array

```
#define MAX 30
/* A queue with maximum of 30 elements */
typedef struct DQ
{
    int data [MAX];
    int rear, front;
} DQ;
```

Operations on a dequeue

- initialize() : Make the queue empty.
- empty() : Determine if queue is empty.
- full() : Determine if queue is full.
- enqueueF() : Insert an element at the front end of the queue.
- enqueueR() : Insert an element at the rear end of the queue.
- dequeueR() : Delete the rear element.
- dequeueF() : Delete the front element.
- print() : Print elements of the queue.

Prototype of functions used for various operations on queue

- void initialize (DQ *p);
- int empty (DQ *p);
function returns 1 or 0, depending on whether the queue pointed by p is empty or not.
- int full (DQ *p);
function returns 1 or 0, depending on whether the queue pointed by p is full or not.
- void enqueueF (DQ *p, int x);
- void enqueueR (DQ *p, int x);
- int deleteR (DQ *p);
- int deleteF (DQ *p);
- void print (DQ *p);

- enqueueR() and enqueueF() will cause an overflow if the queue is full.
- dequeueR() and dequeueF() will cause an underflow if the queue is empty.

Example 2.16.3 : (i) Consider a dequeue given below which has LEFT=1, RIGHT=5

_ A B C D E _ _ _ .

Now perform the following operations on the dequeue

1. Add F on the left.
2. Add G on the right.
3. Add H on the right.
4. Delete two alphabets from left
5. Add I on the right

(ii) Differentiate peep() and pop() functions

Solution :

(i)

| Sr. No. | Operation | Dequeue | Left | Right |
|---------|------------------------------|------------|------|-------|
| 1. | Initially | -ABCDE -- | 1 | 5 |
| 2. | Add F on the left | FABCDE -- | 0 | 5 |
| 3. | Add G on the right | FABCDEG - | 0 | 6 |
| 4. | Add H on the right | FABCDEGH | 0 | 7 |
| 5. | Delete 2 alphabets from left | -- BCDEGH | 2 | 7 |
| 6. | Add I on the right | I - BCDEGH | 2 | 0 |

(ii) **peep()** : Seeing the value of the top element of stack without removing it.

Pop() : Removing the top element of a stack.

2.16.3 Implementation of Dequeue using a Circular Array

MU - Dec. 19

University Questions

Q. Write a C program to implement Double Ended Queue.
(Dec. 19, 8 Marks)

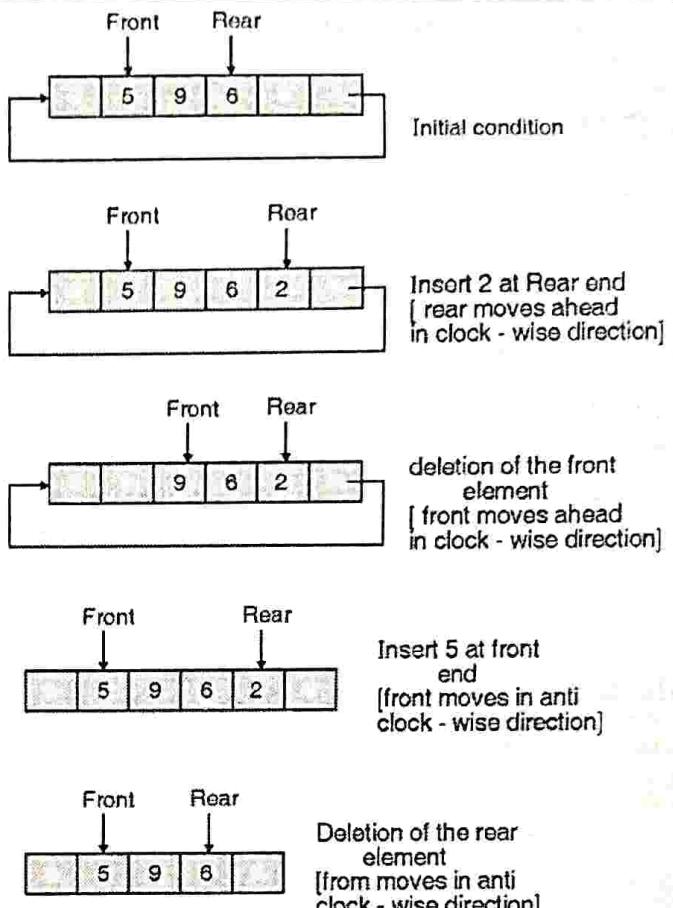


Fig. 2.16.2 : Various operations on dequeue and movement of front and rear in clock-wise or anti-clockwise direction

Expression for clock-wise movement

rear = (rear + 1) % MAX where MAX is the size of the array.

front = (front + 1) % MAX

Expression for anti-clockwise movement

rear = (rear - 1 + MAX) % MAX

front = (front - 1 + MAX) % MAX

'C' functions for various operations on a dequeue represented using a circular array.

A dequeue can be represented using the following data type.

```
typedef struct dequeue
{
    int data[MAX];
    int front, rear;
} dequeue;
```



- (1) Initialize the queue by setting values of rear and front as -1.

```
void initialize(dequeue *P)
{
    P → rear = -1;
    P → front = -1;
}
```

- (2) Test, whether the dequeue is empty ?

```
int empty(dequeue *P)
{
    if(P → rear == -1)
        return(1);
    return(0);
}
```

- (3) Test, whether the dequeue is full ?

```
int full(dequeue *P)
{
    if((P → rear + 1) % MAX == P → front)
        return(1);
    return(0);
}
```

- (4) Add item at the rear end of the dequeue.

```
void enqueueR(dequeue *P, int x)
{
    if(empty(P))
    {
        P → rear = 0;
        P → front = 0;
        P → data[0] = x;
    }
    else
    {
        P → rear = (P → rear + 1) % MAX;
        P → data[P → rear] = x;
    }
}
```

- (5) Add item at the front end of the dequeue : on addition of item at the front end, front will move in anti-clockwise direction.

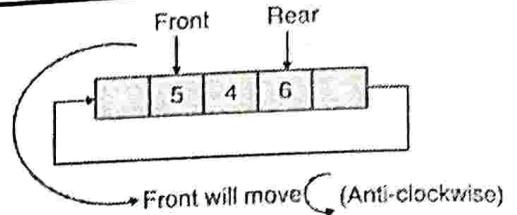


Fig. 2.16.3

void enqueueF(dequeue *P, int x)

```
{
    if(empty(P))
    {
        P → rear = 0;
        P → front = 0;
        P → data[0] = x;
    }
    else
    {
        P → front = (P → front - 1 + MAX) % MAX;
        P → data[P → front] = x;
    }
}
```

- (6) Delete an item from the front end of the dequeue :

```
int dequeueF(dequeue *P)
{
    int x;
    x = P → data[P → front];
    if(P → rear == P → front)
        /* delete the last element */
        initialize(P);
    else
        P → front = (P → front + 1) % MAX;
    return(x);
}
```

- (7) Delete an item from the rear end of the dequeue :

```
int dequeueR(dequeue *P)
{
    int x;
    x = P → data[P → rear];
    if(P → rear == P → front)
        initialize(P)
    else
```

```

P → rear = (P → rear - 1 + MAX)%MAX;
return(x);
}

```

Program 2.16.2 : A program showing various operations on a deque represented using a circular queue.

```

#include <stdio.h>
#include <conio.h>
#define MAX 30
typedef struct deque
{
    int data [MAX];
    int rear, front;
} deque;
void initialize(deque *p);
int empty(deque *p);
int full(deque *p);
void enqueueR(deque *p, int x)
void enqueueF(deque *p, int x)
int dequeueF(deque *p);
int dequeueR(deque *p);
int print(deque *p);
void main()
{
    int x, op, n;
    deque q;
    initialize(&q);
    do
    {
        printf("\n 1)create\n 2)insert(rear)\n
3)insert(front)\n 4)Delete(rear)\n 5)Delete(front)");
        printf("\n 6)print\n 7)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 ...");
                    }
                }
            case 2 : P → rear = (P → rear - 1 + MAX)%MAX;
                return(x);
        }
    }
}

```

```

exit(0);
}
enqueueR(&q, x);
}
break;
case 2 : printf("\n enter element to be inserted:");
scanf("%d", &x);
if(full(&q))
{
    printf("\n queue is full ...");
    exit(0);
}
enqueueR(&q, x);
break;
case 3 :
printf("\n enter the elment to be inserted : ");
scanf("%d", &x);
if(full(&q))
{
    printf("\n queue is full ...");
    exit(0);
}
enqueueF(&q, x);
break;
case 4 : if(empty(&q))
{
    printf("\n queue is empty ...");
    exit(0);
}
x = deleteR(&q);
printf("\n element = %d", x);
break;
case 5 : if(empty(&q))
{
    printf("\n queue is empty ...");
    exit(0);
}
x = deleteF(&q);
printf("\n element = %d", x);
break;
case 6 : printf(&q); break;
default : break;
}while(op != 7);
}
void initialize(dequeue *P)

```

```

{
    P → rear = -1;
    P → front = -1;
}

int empty(dequeue *P)
{
    if(P → rear == -1)
        return(1);
    return(0);
}

int full(dequeue *P)
{
    if((P → rear + 1) % MAX == P → front)
        return(1);
    return(0);
}

void enqueueR(dequeue *P, int x)
{
    if(empty(P))
    {
        P → rear = 0;
        P → front = 0;
        P → data[0] = x;
    }
    else
    {
        P → rear = (P → rear + 1) % MAX;
        P → data[p → rear] = x;
    }
}

void enqueueF(dequeue *P, int x)
{
    if(empty(P))
    {
        P → rear = 0;
        P → front = 0;
        P → data[0] = x;
    }
}

```

```

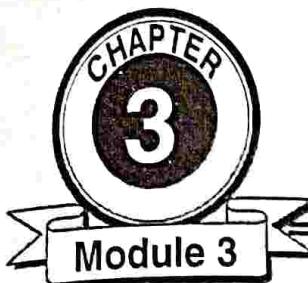
else
{
    P → front = (P → front - 1 + MAX) % MAX;
    P → data[P → front] = x;
}

int dequeueF(dequeue *P)
{
    int x;
    x = P → data[P → front];
    if(P → rear == P → front)
        /* delete the last element */
        initialize(P);
    else
        P → front = (P → front + 1) % MAX;
    return(x);
}

int dequeueR(dequeue *P)
{
    int x;
    x = P → data[P → rear];
    if(P → rear == P → front)
        initialize(P);
    else
        P → rear = (P → rear - 1 + MAX) % MAX;
    return(x);
}

void print(dequeue *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]);
}

```



Linked List

Syllabus

Introduction, Representation of Linked List, Linked List v/s Array, Types of Linked List - Singly Linked List, Circular Linked List, Doubly Linked List, Operations on Singly Linked List and Doubly Linked List, Stack and Queue using Singly Linked List, Singly Linked List Application-Polynomial Representation and Addition.

3.1 Representation and Implementation of Singly Linked Lists

3.1.1 Comparison between Array and Linked Lists

MU - May 16, Dec. 17, May 18

University Question

Q. What are the advantages of using linked lists over arrays ? (May 16, Dec. 17, May 18, 5 Marks)

- 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.

1. Simple to use
 2. Simple to define
 3. Constant access time
 4. 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 $a + i * \text{size of array element}$ in bytes. This mapping is carried out in constant time, irrespective of which element is accessed. Array data structure suffers from some severe limitations :

1. Size of an array is defined at the time of programming.
 2. Insertion and deletion is time consuming.
 3. 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. (Inline 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.

3.1.2 Representation

MU - Dec. 15

University Question

Q. Explain Linked list as an ADT. (Dec. 15, 5 Marks)

- 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. 3.1.1 shows the actual representation of the structure.



Fig. 3.1.1 : Representation of the structure

- A variable of the above structure type is conventionally known as a **node**. Fig. 3.1.2 gives a representation of a linked list of three nodes.

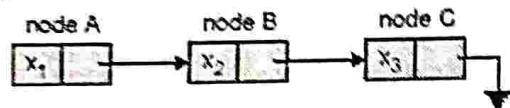


Fig. 3.1.2 : Linked list

- 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.

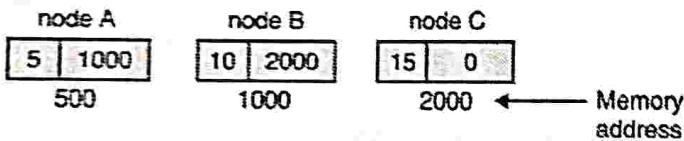


Fig. 3.1.3 : Memory representation of a linked list

- Fig. 3.1.3 gives a memory representation of the linked list shown in Fig. 3.1.2. 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.

3.1.3 Implementation

MU - Dec. 15

University Question

Q. Explain Linked list as an ADT. (Dec. 15, 5 Marks)

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;
}
```

- 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
```

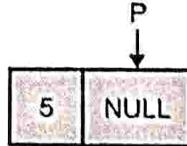


Fig. 3.1.4 : Memory for a node has been allocated during run-time. Its address is stored in the pointer P

- 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 head node

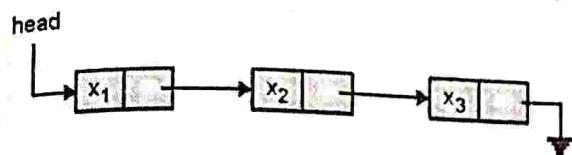


Fig. 3.1.5 : A linked list with the address of the head node

- As an array is referenced by its starting address, a linked list is known by the address of its head node. Address of the head 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.

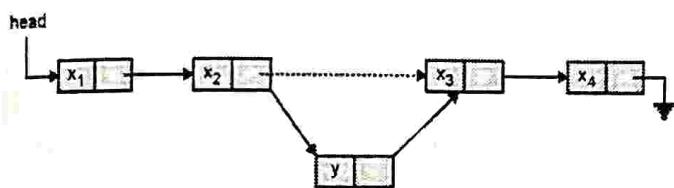


Fig. 3.1.6 : Insertion into a linked list

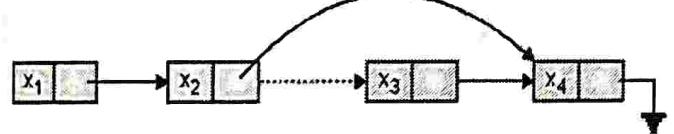


Fig. 3.1.7 : Deletion from a linked list

- Insertion into a linked list requires obtaining a new node and then changing values of two pointers. The general idea is shown in Fig. 3.1.6.
- 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. 3.1.7 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.

3.1.4 Types of Linked List

3.1.4(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.

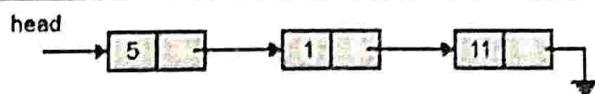


Fig. 3.1.8 : Singly linked list

3.1.4(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.

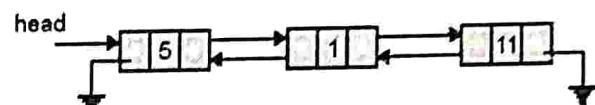


Fig. 3.1.9

3.1.4(C) A Circular Linked List

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.

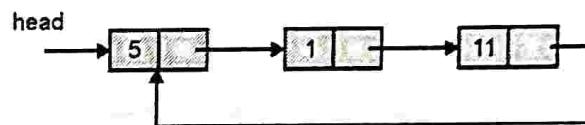


Fig. 3.1.10

3.1.5 Differences between Singly Linked List and Doubly Linked List

MU - May 15

University Question

- Q. State differences between Singly Linked List and Doubly Linked list data structures along with their applications.
(May 15, 5 Marks)

| Sr. No. | Singly linked list | Doubly linked list |
|---------|--|---|
| 1. | It has one pointer, pointing to successor. | It has two pointers, one pointing to successor and another pointing to predecessor. |
| 2. | Traversal is possible only in the forward direction. | One can traverse in both forward and backward directions. |
| 3. | Less memory is required by a node. | More memory required by a node. |
| 4. | Fewer pointer adjustment in delete and insert operation. | More pointer adjustment in delete and insert operation. |



| Sr. No. | Singly linked list | Doubly linked list |
|---------|---|--|
| 5. | In singly linked list, each node contains data and link. | In doubly linked list, each node contains data, link to next node and link to previous node. |
| 6. | Applications : <ul style="list-style-type: none"> (i) Representation of linear data structure. (ii) Representation of Stack. (iii) Representation of Queue. (iv) Representation of Polynomial. (v) Representation of Sparse matrix. | Applications : <ul style="list-style-type: none"> (i) Representation of dequeue. (ii) Memory management and garbage collection. |

3.2 Basic Linked List Operations

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.

3.2.1 Creating a Linked List

Program 3.2.1 : Program to create linked list.

```
#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 1'st node with its address in HEAD
    HEAD = (node*)malloc(sizeof(node));
    //read the data in 1'st 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.

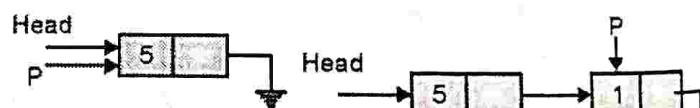


Fig. 3.2.1 : The linked

list after insertion of
the first item

Fig. 3.2.2 : 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.

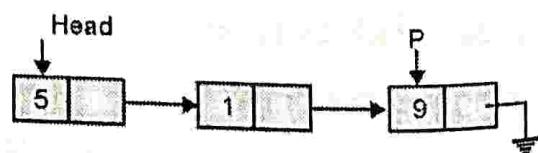


Fig. 3.2.3 : The status of the linked list after
insertion of 3 elements

Program 3.2.2 : 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);
}
```

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 */
```

3.2.3 Counting Number of Nodes in a Linked List through Count Function

```
int count(node *P)
{
    int i;
    i = 0;
    while(P != NULL)
    {
        i = i + 1;
        P = P -> next;
    }
    return(i);
}
```

Program 3.2.3 : Program to create linked list interactively and print the list and total number of items in the list.

MU - Dec. 16, 10 Marks

OR

Write a program in 'C' to create a singly linked list and perform the following operations :

- (I) Insert into list
- (II) Search for data
- (III) Delete from list
- (IV) Display data

MU - Dec. 16, 10 Marks

```
#include<conio.h>
#include<stdio.h>
typedef struct node
{
    int data;
    struct node *next;
}node;
```

3.2.2 Traversing a Linked List

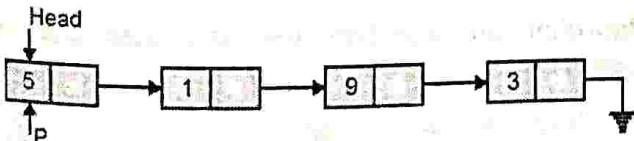


Fig. 3.2.4 : Traversal starts from the first node



```

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

```

3.2.4 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.

3.2.5 Inserting an Item

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.

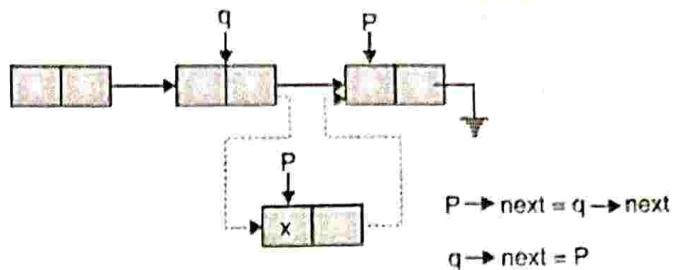
Algorithm for inserting the new data after a node N1

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 $N1 \rightarrow \text{next}$.
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);
}
```



3.2.5(A) Inserting an Item at the End of a Linked List

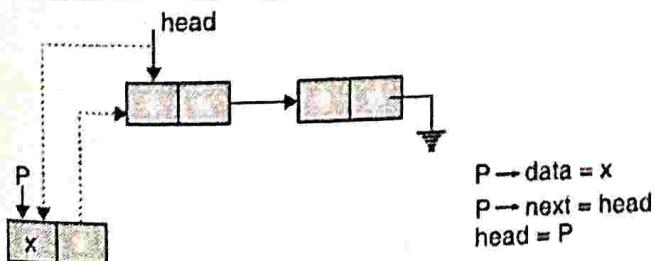
Algorithm

[Insert an item 'x' in a linked list, referenced by the pointer 'head']

1. Acquire memory for new node
i.e. $p = (\text{node}^*) \text{malloc}(\text{sizeof}(\text{node}))$;
2. Assign value to the data filed and make its 'next' field 'NULL'
i.e. $p \rightarrow \text{data} = x$;
 $p \rightarrow \text{next} = \text{NULL}$;
3. If 'head' is 'NULL'
then
[$\text{head} = p$;
goto step 6]
4. Position a pointer q on the last node by traversing the linked list form the first node and until it reaches the last node.
i.e. $q = \text{head}$
 $\text{while}(q \rightarrow \text{next} \neq \text{NULL})$
 $q = q \rightarrow \text{next}$;
5. Store the address of the newly acquired node, pointed by p, in the next field of node pointed by q.
i.e. $q \rightarrow \text{next} = p$;
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 = head;
    head = p;
}
```





```

q = head;
while(q->next != NULL)
    q = q->next;
q->next = p;
return(head);
}

```

3.2.5(B) Inserting a Data 'x' at a given Location 'LOC' In a Linked List, Referenced by 'head'

Algorithm

- Acquire memory for new node with its address in pointer p.
i.e. p = (node *) malloc(sizeof(node));
- Assign value to data field and make its 'next' field 'NULL'.
i.e. p → data = x;
p → next = NULL;
- 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
- [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;
- if q is NULL
[then report " overflow" and terminate the algorithm.
go to step 7]
- Insert the node pointed by p, after the node pointed by q.
i.e. p → next = q → next;
q → next = p;
- 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);
}

```

Program 3.2.4 : Write a C program to create singly linked list of integers, display the same and count no. of even elements in the list (use functions).

```

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

typedef struct node
{
    int data;
    struct node *next;
} node;

node *create ();
void print (node *head);
int count_even (node *head);

```

```

void main()
{
    node *head;
    int result;
    head = create();
    print(head);
    result = count_even(head);
    printf("\n no. of even data = %d", result);
    getch();
}

node *create()
{
    node *head, *p;
    int i, n;
    head = NULL;
    printf("\nEnter 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);
}

void print(node *head)
{
    node *p;
    printf("\n\n");
    for(p = head; p != NULL; p = p->next)
        printf("%d ", p->data);
}

int count_even(node * head)
{
    int s = 0;
    for(; head != NULL; head = head->next)
        if(head->data%2 == 0)
            s++;
    return(s);
}

```

Example 3.2.1 : Explain role of stack in recursion and implement the following using a recursive functions.

- Display singly linked list reverse
- Count no. of nodes in singly linked list.

Solution :

Nested calls of functions are managed through stack. When a function is invoked, memory is allocated on stack for;

- Local variables
- Return address

On termination of execution of the function, memory is de-allocated. Details regarding the called function are stored in a specific structure known as Activation record.

An activation record consists of :

- Storage space for all local variables
- Definition of function
- Return address
- A pointer to activation record.

Activation records are stored on the stack. When a function is called, its activation record is pushed into the stack and control is transferred to the called Program. When the function execution completes, the control returns to the caller function. It obtains the return address from the return address field of the activation record.

```

void display_reverse (node *h)
{
    if(h != NULL)
    {
        display_reverse (h-> next);
        printf("%d", h->data);
    }
}

int count(node *h)
{
    if(h == NULL)
        return(0);
    return(1 + count(h->next));
}

```



3.2.5(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. $p = (\text{node} *) \text{malloc}(\text{sizeof}(\text{node}))$;

2. Assign value to its data field and make its next field 'NULL'.

i.e. $p \rightarrow \text{data} = x$;

$p \rightarrow \text{next} = \text{NULL}$;

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($\text{head} == \text{NULL} || x < \text{head} \rightarrow \text{data}$)

{

$p \rightarrow \text{next} = \text{head}$;

$\text{head} = p$;

goto step 5

}

4. [If head is not NULL and $x > \text{head} \rightarrow \text{data}$]

Locate the point of insertion, as per the priority of x.

i.e.

$q = \text{head}$;

while($q \rightarrow \text{next} != \text{NULL} \&& x > q \rightarrow \text{next} \rightarrow \text{data}$)

$q = q \rightarrow \text{next}$;

x should be inserted after the node pointed by q.

i.e. $P \rightarrow \text{next} = q \rightarrow \text{next}$;

$q \rightarrow \text{next} = p$

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);
}
```

3.2.6 Deleting an Item

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.

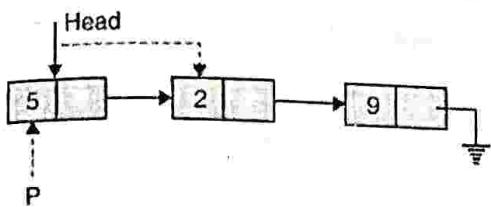
3. Deleting from the middle of the list.

Algorithm for deleting the first item.

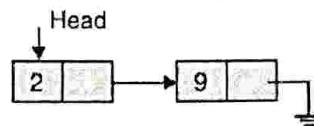
1. Store the address of the first node in a pointer variable, say P.

2. Move the head to the next node.

3. Free the node whose address is stored in the pointer variable P.



(a) Before deletion



(b) After deletion

Fig. 3.2.7 : Deleting the first item

Algorithm for deleting a node from the middle of the linked list.

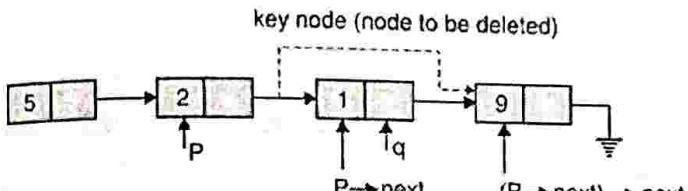


Fig. 3.2.8 : Deletion of middle element

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.

```
/* x to be deleted from a linked list */
node * delete(node *head, int x)
{
    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);
}
```

3.2.6(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);
}
```

3.2.6(B) Deletion of a Node at Location 'LOC' from a Linked List

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 head 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);
```

```
}
```

3.2.6(C) Delete a Linked List, Referenced by the Pointer Head

Algorithm

1. if head == NULL
then
 [goto 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);
}
```

3.2.7 Concatenation of Two Linked Lists

Algorithm for concatenation

Let us assume that the two linked lists are referenced by head 1 and head 2 respectively.

1. If the first linked list is empty then return head 2.
2. If the second linked list is empty then return head 1.
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 and return head 1.

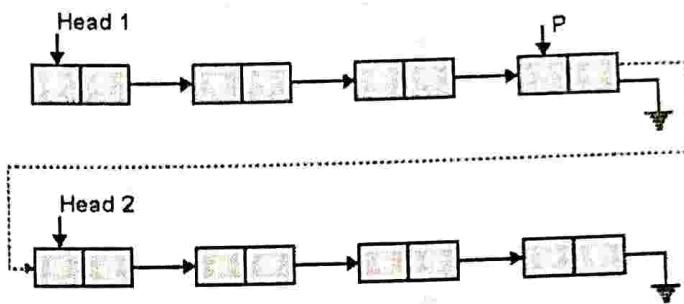


Fig. 3.2.9

'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);
}
```

3.2.8 Inversion of Linked List

A linked list can be reversed by changing the direction of the pointer, iteratively.

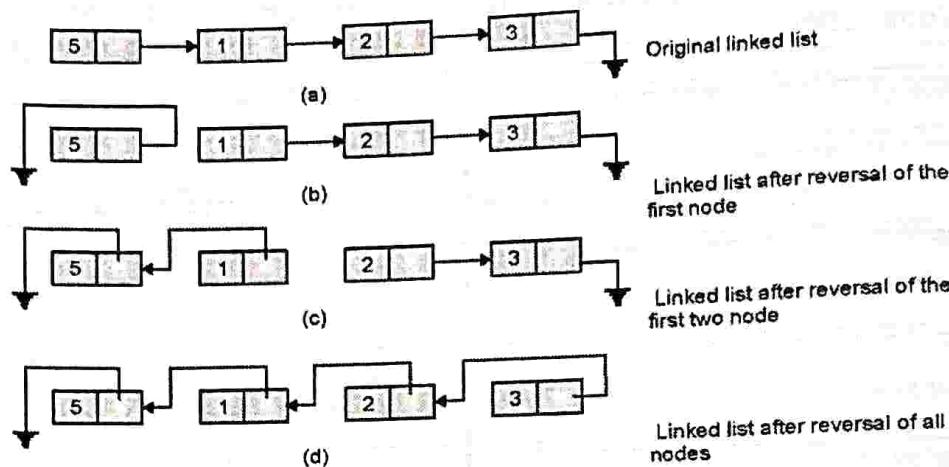


Fig. 3.2.10

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.

```

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;
}
  
```

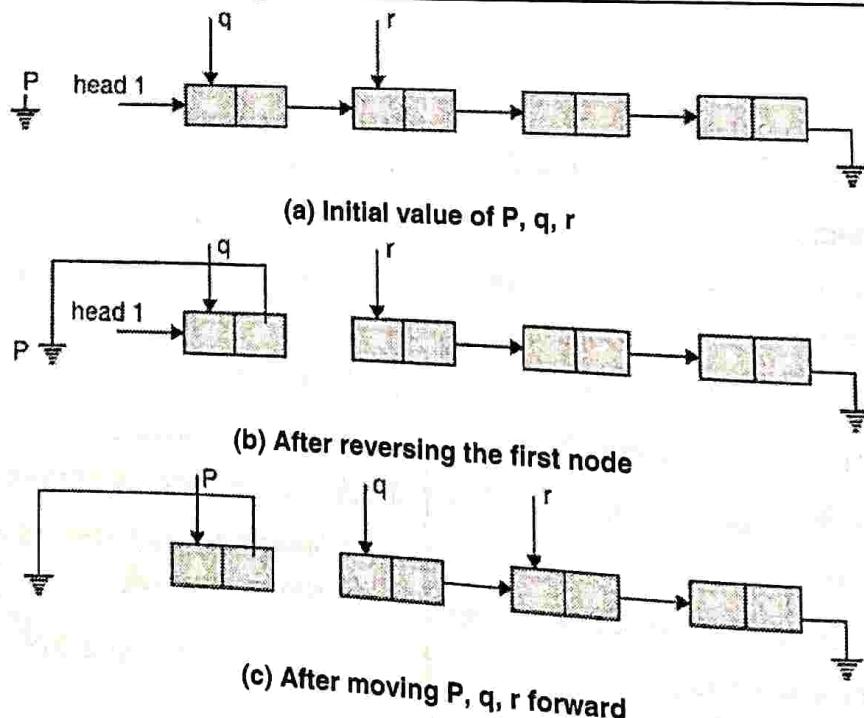


Fig. 3.2.11

'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);
}

```

3.2.9 Searching a Data 'x' in a Linked List, Referenced by the Pointer Head**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 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.

Search Functions

- (I) '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);
}

```

- (II) '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);
}

```

- (III) '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);
}

```

3.2.10 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) $P \rightarrow \text{data}$ is greater than x .
- 1. $P = \text{head}$
- 2. if $P \rightarrow \text{data}$ is equal to x
then
[report success and
end the traversal]

3. if P is Null or $P \rightarrow \text{data} > x$
then
[report failure and
end the traversal]

4. Continue searching
i.e. $P = P \rightarrow \text{next};$
goto step 2.

5. 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);
}

```

3.2.11 New Linear Linked List by Selecting Alternate Element

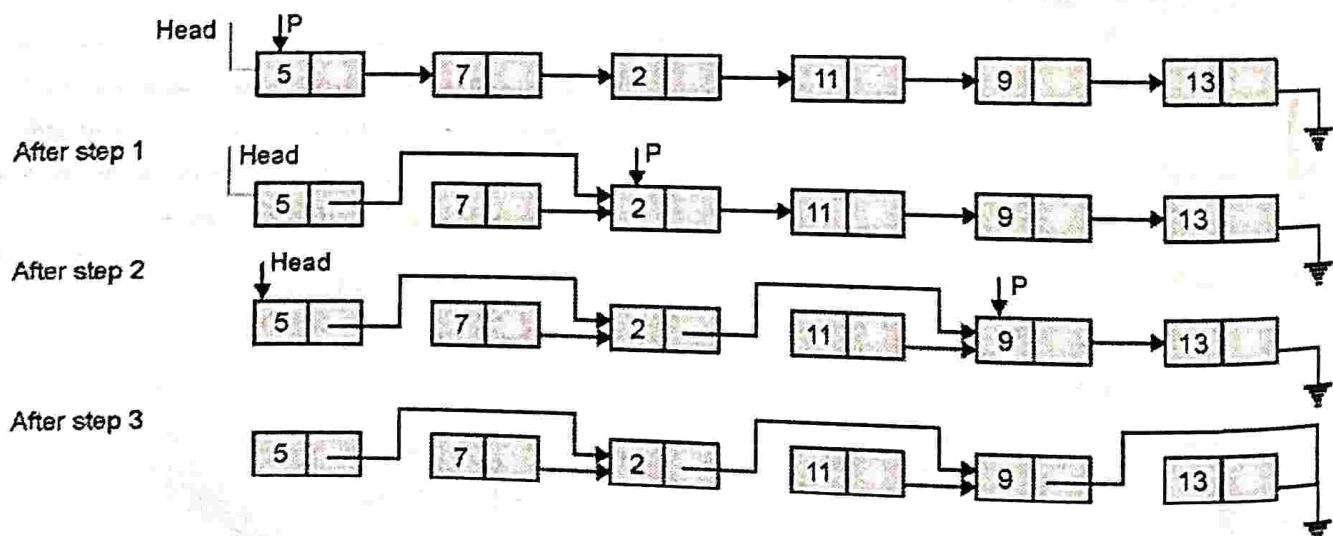


Fig. 3.2.12

Algorithm

[Linked list is referenced by the pointer head]

1. $P = \text{head}$
2. If ($P \rightarrow \text{next}$ is equal to NULL)
 - then go step 5
3. $P \rightarrow \text{next} = P \rightarrow \text{next} \rightarrow \text{next}$
 $P = P \rightarrow \text{next};$
4. goto step 2
5. Stop.

```
void create_alternate(node *head)
{
    node *P;
    P = head;
    while(P != NULL && P->next != NULL)
    {
        P->next = P->next->next;
        P = P->next;
    }
}
```

3.2.12 Handling of Records through Linked List

Assume a singly linked list where each node contains student details like name, rollno and percentage of marks. Following is a COUNT() function 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;

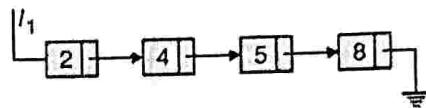
int COUNT(node *head)
{
    int n = 0;
    while(head != NULL)
    {
        if(head -> percent > 60.00)
            n++;
    }
}
```

```
    head = head -> next;
}
return(n);
}
```

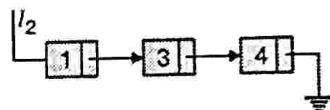
3.2.13 Merging of Sorted Linked Lists

Merging of two sorted linked l_1 and l_2 with resultant list as l is being explained with help of an example.

First list l_1 (given) :



Second list l_2 (given) :

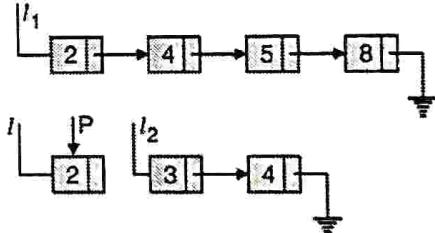


Final list l (initially NULL) :

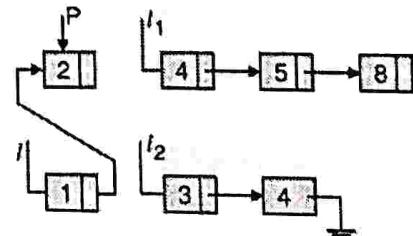


Merging

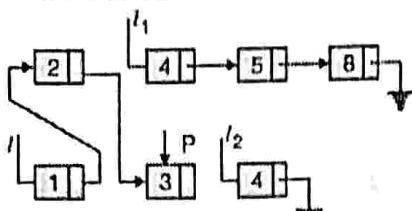
Step 1 : Smaller of the two elements pointed by l_1 and l_2 is copied to l . A pointer P is positioned on the last node of l .



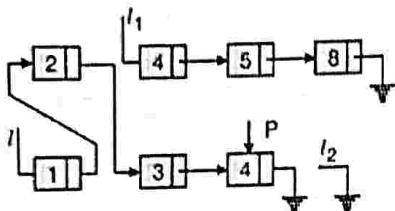
Step 2 : Smaller of the two elements pointed by l_1 and l_2 is appended to l . P is positioned on the last node of l .



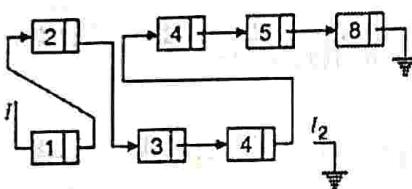
Step 3 : Smaller of the two elements pointed by l_1 and l_2 is appended to l . P is positioned on the last node of l .



Step 4 : Smaller of the two elements pointed by l_1 and l_2 is appended to l . P is positioned on the last node.



Step 5 : Since, l_2 has become NULL, the list l_1 is appended at the end of l .



'C' function for merging of two sorted lists pointed by l_1 and l_2

```
node * merge(node * l1, node * l2)
{
    node *l = NULL, *P;
    if(l1 == NULL) //first list is empty
        return (l2);
    if(l2 == NULL) //second list is empty
        return (l1);
    // copy the first element
    if(l1->data < l2->data)
    {
        l = P = l1;
        l1 = l1->next;
    }
    else
    {
        l = P = l2;
        l2 = l2->next;
    }
    // Append remaining data to produce l.
    while(l1 != NULL && l2 != NULL)
    {
        if(l1->data < l2->data)
        {
            P->next = l1;
            l1 = l1->next;
            P = P->next;
        }
        else
    }
```

```
{
    P->next = l2;
    l2 = l2->next;
    P = P->next;
}

// one of the lists l1 or l2 will have some elements
if(l1 != NULL)
    P->next = l1;
else
    P->next = l2;
return(l);
}
```

3.2.14 Splitting a Linked List at the Middle and Merge with Second Half as First Half

'C' program to create a single linked list and split it at the middle and make the second half as the first and vice versa. Display the final list.

```
#include<stdio.h>
#include<conio.h>
typedef struct node
{
    int data;
    struct node *next;
}node;
void main()
{
    node *head = NULL, *p, *q;
    int n, i, x;
    printf("\n Enter no. of nodes :");
    scanf("%d", &n);
    /* create a linked list of n nodes */
    for(i = 0; i < n; i++)
    {
        printf("\n Enter the next element :");
        scanf("%d", &x);
        p = (node*)malloc(sizeof(node));
        p->data = x;
        p->next = NULL;
        if(head == NULL)
            head = q = p;
        else
        {
            q->next = p;
            q = p;
        }
    }
    /* locate the centre of linked list */
    p = q = head;
    while(q->next != NULL)
    {
        p = p->next;
        q = q->next;
    }
```

```

if(q->next != NULL)
    q = q->next;
}
q->next = head;
head = p->next;
p->next = NULL;
/* Display the final list */
for(p = head; p != NULL; p->next)
{
    printf("\n%d", p->data);
    p = p->next;
}
}

```

Output

```

Enter no. of nodes : 3
Enter the next element : 12
Enter the next element : 23
Enter the next element : 34
34
12
23

```

3.2.15 Removing Duplicate Elements from a Linked List

```

void delete_duplicates(node *head)
{
    int x;
    node *P;
    while(head != NULL)
    {
        x = head->data;
        /* delete all subsequent nodes with x in its
           data field*/
        P = head;
        while(P->next != NULL)
        {
            if(P->next->data == x) //delete
                P->next = P->next->next;
            else
                P = P->next;
        }
        head = head->next;
        /* find the duplicates of the next element*/
    }
}

```

Example 3.2.2 : Write recursive functions for :

- (i) Display SLL forward
- (ii) Display SLL reverse

Solution :**(i) Display SLL forward**

```

void display_forward(node*h)
{
    if(h != NULL)

```

```

    {
        printf("%d", h->data);
        display_forward(h->next);
    }
}

```

(ii) Display SLL reverse

```

void display_reverse(node*h)
{
    if(h != NULL)
    {
        display_reverse(h->next);
        printf("%d", h->data);
    }
}

```

Example 3.2.3 : Write a 'C' program to implement a singly Linked List which supports the following operations :

- (i) Insert a node in the beginning
- (ii) Insert a node in the end
- (iii) Insert a node after a specific node
- (iv) Deleting a specific node
- (v) Displaying the list.

MU - Dec. 14, May 17, May 18, 10 Marks

Solution :

```

/*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);
node *copy(node *);
int count(node *);
node *concatenate(node *, node *);

```

```

if(q->next != NULL)
    q = q->next;
}
q->next = head;
head = p->next;
p->next = NULL;
/* Display the final list */
for(p = head; p != NULL; p->next)
{
    printf("\n%d", p->data);
    p = p->next;
}
}

```

Output

```

Enter no. of nodes : 3
Enter the next element : 12
Enter the next element : 23
Enter the next element : 34
34
12
23

```

3.2.15 Removing Duplicate Elements from a Linked List

```

void delete_duplicates(node *head)
{
    int x;
    node *P;
    while(head != NULL)
    {
        x = head->data;
        /* delete all subsequent nodes with x in its
         data field*/
        P = head;
        while(P->next != NULL)
        {
            if(P->next->data == x) //delete
                P->next = P->next->next;
            else
                P = P->next;
        }
        head = head->next;
        /* find the duplicates of the next element*/
    }
}

```

Example 3.2.2 : Write recursive functions for :

- (i) Display SLL forward
- (ii) Display SLL reverse

Solution :

(i) Display SLL forward

```

void display_forward(node*h)
{
    if(h != NULL)

```

```

    {
        printf("%d", h->data);
        display_forward(h->next);
    }
}

```

(ii) Display SLL reverse

```

void display_reverse(node*h)
{
    if(h != NULL)
    {
        display_reverse(h->next);
        printf("%d", h->data);
    }
}

```

Example 3.2.3 : Write a 'C' program to implement a singly Linked List which supports the following operations :

- (i) Insert a node in the beginning
- (ii) Insert a node in the end
- (iii) Insert a node after a specific node
- (iv) Deleting a specific node
- (v) Displaying the list.

MU - Dec. 14, May 17, May 18, 10 Marks

Solution :

```

/*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);
node *copy(node *);
int count(node *);
node *concatenate(node *, node *);

```

```

void split(node *);
void main()
{
    int op, op1, x;
    node *head = NULL;
    node *head1 = NULL, *head2 = NULL,
    *head3 = NULL;
    clrscr();
    do
    {
        printf("\n\n 1)create\n 2)Insert\n 3)Delete\n
4)Search");
        printf("\n 5)Reverse\n 6)Print\n 7)Count\n 8)Copy\n
9)Concatenate");
        printf("\n 10)Split\n 11)Quit");
        printf("\nEnter your Choice:");
        scanf("%d", &op);
        switch(op)
        {
            case 1: head = create(); break;
            case 2: printf("\t 1)Beginning\n\t 2)End\n\t 3)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("\t 1)Beginning\n\t 2)End\n\t
3)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;
    case 7: printf("\nNo.of node = %d", count(head));
        break; //count
    case 8: head1 = copy(head); //copy
        printf("\nOriginal Linked List :");
        print(head);
        printf("\nCopied Linked List :");
        print(head1);
        break;
    case 9:printf("\nEnter the first linked list:");
        head1 = create();
        printf("\nEnter the second linked list:");
        head2 = create();
        head3 = concatenate(head1, head2);
        printf("\nConcatenated Linked List :");
        print(head3);
        break; //concatenate
    case 10:printf("\nEnter a linked list : ");
        head1 = create();
        split(head1);
        break; //split
    }
}
}while(op!=11);
}

node *create()
{
    node *head, *p;
    int i, n;
    head = NULL;
    printf("\nEnter 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("\n Insert after which number ? : ");
    scanf("%d", &y);
    //locate the 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("\n Data 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("\n Underflow....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("\n Underflow....Empty Linked List");
        return(head);
    }
}

```

```

printf("\n Enter 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("\n Underflow....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("\n Enter the data to be searched: ");
    scanf("%d", &data);
    p = head;
    while(p != NULL && p->data != data)
    {
        loc++;
        p = p->next;
    }
    if(p == NULL)
        printf("\n Not found:");
    else
        printf("\n Found 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);
}

node *copy(node *h)
{
    node *head = NULL, *p;
    if(h == NULL)
        return head;
    //Copy the first node
    head = p = (node*)malloc(sizeof(node));
    p->data = h->data;
    while(h->next != NULL)
    {
        p->next = (node*)malloc(sizeof(node));
        p = p->next;
        h = h->next;
        p->data = h->data;
    }
    p->next = NULL;
    return (head);
}

int count(node *h)
{
    int i;
    for(i = 0; h != NULL; h = h->next)
        i++;
    return(i);
}

node *concatenate( node *h1, node * h2)
{
    node *p;
    if(h1 == NULL)
        return(h2);
}

```

```

if(h2 == NULL)
    return(h1);
p = h1;
while(p->next != NULL)
    p = p->next;
p->next = h2;
return(h1);
}

void split(node *h1)
{
    node *p, *q, *h2;
    printf("\n Linked list to be split : ");
    print(h1);
    /* linked list will be broken from the centre using
    the //pointers p and q */
    if(h1 == NULL)
        return;
    p = h1;
    q = h1->next;
    while(q != NULL && q->next != NULL)
    {
        q = q->next->next;
        p = p->next;
        /* When q reaches the last node, p will reach the
        centre node */
    }
    h2 = p->next;
    p->next = NULL;
    printf("\nFirst half : ");
    print(h1);
    printf("\nSecond half : ");
    print(h2);
}

```

3.3 Circular Linked List

MU - May 19

University Question

Q. Write a program to implement Circular Linked List.

Provide the following operations :

- (i) Insert a node .
- (ii) Delete a node
- (iii) Display the list

(May 19, 10 Marks)

- 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.

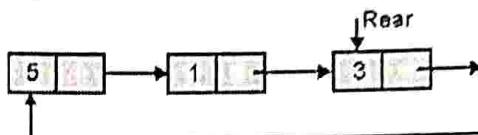


Fig. 3.3.1

- 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.
- 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);
    }
}

```

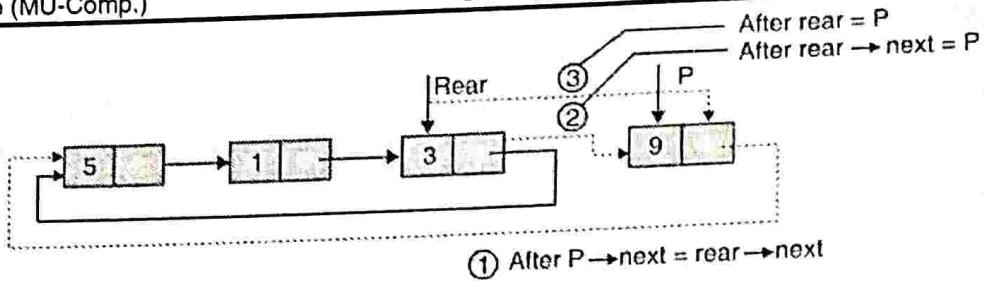


Fig. 3.3.2 : Insertion of a node at rear in circular linked list

'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)
    {
        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 the terminating case for traversal are same. In such a case, do-while is the most suitable construct for traversing.

Example 3.3.1 : Write a 'C' function to insert and delete a N^{th} element in circular linked list ? Give the pictorial representation of the same.

Solution :

Insertion

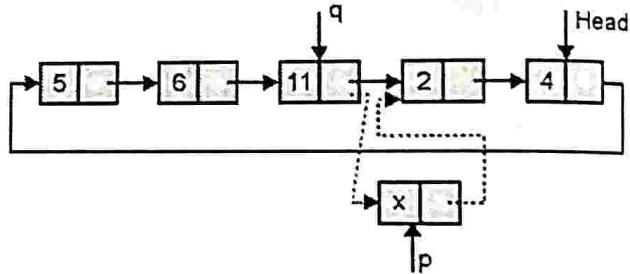


Fig. Ex. 3.3.1(a) : Insertion

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;
        for(i = 1; i < loc - 1; i++)
            head = head → next;
        p → next = head → next;
        head → next = p;
    }
}
```

$p \rightarrow \text{next} = \text{head} \rightarrow \text{next};$

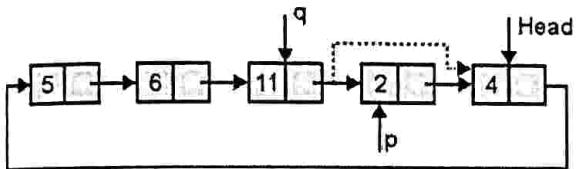
```

    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. Ex. 3.3.1(b)**

q points to $(N - 1)^{\text{th}}$ 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 3.3.2 : Write a C program to implement circular linked list that performs following functions.

- (i) Insert a node in the beginning
- (ii) insert a node in the end
- (iii) Count the number of nodes
- (iv) Display the list

MU - Dec. 19, 12 Marks**Solution :**

```

#include<studio.h>
#include<conio.h>
typedef struct node
{
    int data;
    struct node * next;
}node;

node * insert_beginning(node *h, int x);
node * insert_end(node * h, int x);
int count(node * h);
void display(node * h);

void main()
{
    node *head = NULL;
    int op, x;
    do
    {
        printf("\n 1) Insert [Beginning] \n 2) Insert [End]\n 3) Count\n 4) Display\n 5) Quit");
        printf("\n enter a choice :");

```



```
scanf("%d", &op);
switch(op)
{
    Case 1 :
        printf("\n enter a data :");
        scanf("%d", &x);
        head = insert_beginning(head,x);
        break;

    Case 2 :
        printf("\n enter a data:");
        scanf("\%d", &x);
        head = insert_end(head, x);
        break;

    Case 3 :
        x = count(head);
        printf("\n No. of nodes = %d",x);
        break;

    case 4 : display(head);
        break;
}

} while(op != 5);

}

node * insert_beginning(node * h, int x)
{
    node * p;
    p = (node *) malloc(sizeof(node));
    p → data = x;
    if(h == NULL)
    {
        p → next = p;
        return(p);
    }
    p → next = h → next;
    h → next = p;
    return(h);
}

node * insert_end(node * h, int x)
{
    node * p;
    p = (node *) malloc(sizeof(node));
```

```
p → data = x;
if(h == NULL)
{
    p → next = p;
    return(p);
}
p → next = h → next;
h → next = p;
h = p;
return(h);

int count(node * h)
{
    int x = 0;
    node * p;
    p = h;
    if(h == NULL)
        return(0);
    do
    {
        x = x + 1;
        p = p → next;
    } while(p != h);
    return(x);
}

void display(node * h)
{
    node * p;
    if(h == NULL)
        return;
    p = h;
    do
    {
        printf("%d \t", p → data);
        p = p → next;
    } while(p != h);
}
```

Example 3.3.3 : Write a program in 'C' to implement circular queue using Link list.

MU - May 14, 10 Marks

Solution :

```

/*To implement circular queue using linked list.*/
#include<stdio.h>
#include<conio.h>
typedef struct node
{
    int data;
    struct node *next;
}node;
void init(node **R);
void enqueue(node **R, int x);
int dequeue(node **R);
int empty(node *rear);
void print(node *rear);
void main()
{
    int x, option;
    int n = 0, i;
    node *rear;
    init(&rear);
    clrscr();
    do
    {
        printf("\n 1. Insert\n 2. Delete\n 3. Print\n 4. Quit");
        printf("\n your option:   ");
        scanf("%d", &option);
        switch(option)
        {
            case 1 :
                printf("\n Number of Elements to be inserted");
                scanf("%d", &n);
                for(i = 0; i < n; i++)
                {
                    scanf("\n %d", &x);
                    enqueue(&rear, x);
                }
                break;
            case 2 : if(! empty(rear))
            {
                x = dequeue(&rear);
                printf("\n Element deleted = %d", x);
            }
            else
                printf("\n Underflow..... Cannot deleted");
        }
    }while(option != 4);
    getch();
}

```

```

        break;
    case 3 : print(rear);
        break;
    }
}while(option != 4);
getch();
}

void init(node **R)
{
    *R = NULL;
}
void enqueue(node **R, int x)
{
    node *p;
    p = (node *)malloc(sizeof(node));
    p->data = x;
    if(empty(*R))
    {
        p->next = p;
        *R = p;
    }
    else
    {
        p->next = (*R)->next;
        (*R)->next = p;
        (*R) = p;
    }
}
int dequeue(node **R)
{
    int x;
    node *p;
    p = (*R)->next;
    p->data = x;
    if(p->next == p)
    {
        *R = NULL;
        free(p);
        return(x);
    }
    (*R)->next = p->next;
    free(p);
    return(x);
}
void print(node *rear)

```



```

{
    node *p;
    if(!empty(rear))
    {
        p = rear->next;
    }
    p = p->next;
    do
    {
        printf("\n %d", p->data);
        p = p->next;
    }while(p != rear->next);
}

int empty(node *P)
{
    if(P->next == -1)
        return(1);
    return(0);
}

```

Output

```

1. Insert
2. Delete
3. Print
4. Quit
your option : 1
Element to be inserted 4
      12 23 34 45
1. Insert
2. Delete
3. Print
4. Quit
your option : 3
      12 23 34 45
1. Insert
2. Delete
3. Print
4. Quit
your option : 2
Element deleted = 4
1. Insert
2. Delete
3. Print
4. Quit
your option : 3

```

23 34 45

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

your option : 4

3.3.1 Applications of Circular Linked List

Circular linked list can be used for storing a list of elements. It allows insertion in constant time at both ends of the list. In general, it can be used for following applications :

- | | |
|------------------|-------------------|
| 1. Stack | 2. Queue |
| 3. Dequeue | 4. Priority queue |
| 5. Polynomial | 6. List |
| 7. Sparse matrix | |

3.4 Doubly Linked List

MU Dec. 18

University Question

Q. What is a doubly linked list ? (Dec. 18, 1 Mark)

In a singly linked list, we can easily move in the direction of the link. Finding a node, preceding any node is a time consuming process. The only way to find the node which precedes a node is to start back at the beginning of the list. If we have a problem where moving in either direction is often necessary, then it is useful to have doubly linked lists. Each node has two link fields, one linking in the forward direction and one in the backward direction.

A node in a doubly linked list has at least 3 fields, say "data", "next" and "previous". A doubly linked list is shown in Fig. 3.4.1.

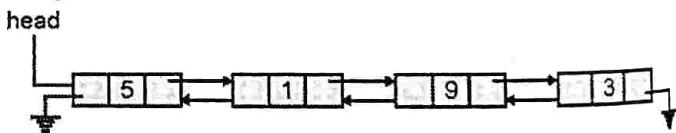


Fig. 3.4.1 : A doubly linked list

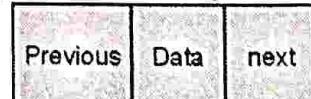


Fig. 3.4.2 : Structure of the node

A node of a doubly linked list can be defined using the following structure.

```

typedef struct dnode
{
    int data;
    struct dnode *next, *prev;
} dnode;

```

3.4.1 Creation of a Doubly Linked List

Fig. 3.4.3 shows stepwise insertion of 3 elements, namely 5, 2 and 6. Pointer P is used to help insertion at the rear of the linked list.

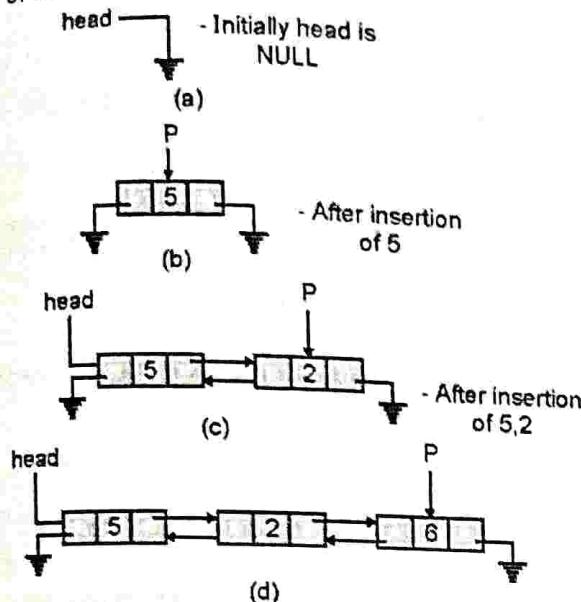


Fig. 3.4.3

Algorithm for insertion of x at the rear

```
dnode *P, *q;
```

Step 1 : Acquire memory for the new data

```
q = (dnode*) malloc(sizeof(dnode));
```

Step 2 : Store x in the newly acquired node

```
q → data = x;
```

Step 3 : Make previous and next field as NULL;

```
q → prev = q → next = NULL;
```

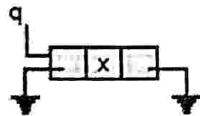


Fig. 3.4.4 : Status of node after step 1 to step 3

Case I : Inserting in an empty list (head = NULL)

```
if(head == NULL)
{
    head = P = q;
}
```

Case II : Inserting in a non-empty list

```
else
{
    P → next = q;
    q → prev = P;
    P = q;
}
```

Note : A linked list can be created through a repeated application of reading a new value of n and then inserting at rear.

Program 3.4.1 : A sample program for creation of a doubly linked list and printing its elements in forward and reverse direction.

OR Write a program in 'C' to implement Doubly Link list with methods insert, delete and search.

MU - May 14, Dec. 18, 5/10 Marks

```
#include<conio.h>
#include<stdlib.h>
#include<stdio.h>
typedef struct dnode
{
    int data;
    struct dnode *next, *prev;
}dnode;

dnode * create();
void print_forward(dnode *);
void print_reverse(dnode *);
void main()
{
    dnode *head;
    head = NULL; // initially the list is empty
    head = create();
    printf("\nElements in forward direction :");
    print_forward(head);
    printf("\nElements in reverse direction :");
    print_reverse(head);
}

dnode *create()
{
    dnode *h, *P, *q;
    int i, n, x;
    h = NULL;
    printf("\nEnter no. of elements :");
    scanf("%d", &n);
    for(i = 0; i < n; i++)
    {
        printf("\nEnter next data: ");
        scanf("%d", &x);
        q = (dnode*)malloc(sizeof(dnode));
        q → data = x;
        if(h == NULL)
            head = P = q;
        else
        {
            P → next = q;
            q → prev = P;
            P = q;
        }
    }
}
```



```

q->data = x;
q->prev = q->next = NULL;
if(h == NULL)
    P = h = q;
else
    P->next = q;
    q->prev = P;
    P = q;
}
return(h);
}

void print_forward(dnode *h)
{
    while(h != NULL)
    {
        printf("<- %d ->", h->data);
        h = h->next;
    }
}

void print_reverse(dnode *h)
{
    while(h->next != NULL)
        h = h->next;
    while(h != NULL)
    {
        printf("<- %d ->", h->data);
        h = h->prev;
    }
}

```

Output

Enter no. of elements : 4

Enter next data: 1

Enter next data: 2

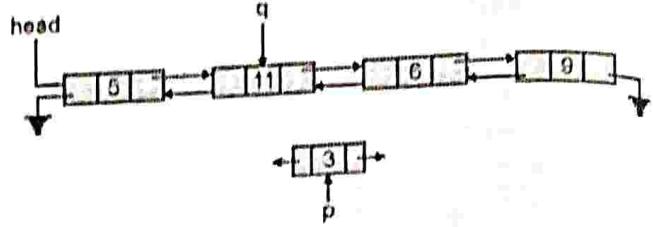
Enter next data: 3

Enter next data: 4

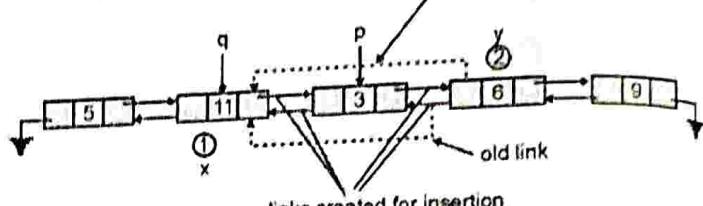
Elements in forward direction : <-1-> <-2-> <-3-> <-4->

Elements in reverse direction : <-4-> <-3-> <-2-> <-1->

Insertion and deletion are two basic operations on such lists. Consider that a new node pointed to by "p" is to be inserted after the node pointed to by q in a doubly linked list as shown in the Fig. 3.4.5.



(a) Insertion of a node in a doubly linked list



(b) A doubly linked list after the insertion of the node

Fig. 3.4.5

Links of two nodes are altered during insertion. In the Fig. 3.4.5(b), a new node pointed to by p is inserted between the two nodes x and y. Following links must be changed for proper insertion of node pointed by p between x and y.

1. Right link of x
2. Left link of y
3. Left and right links of node pointed to by p

Instructions for making above changes

| | |
|--|----------------------------|
| $p \rightarrow next = q \rightarrow next;$ | //right link of p set to y |
| $p \rightarrow prev = q;$ | //left link of p set to x |
| $q \rightarrow next = P;$ | //right link of x set to p |
| $(p \rightarrow next) \rightarrow prev = p;$ | //left link of y set to p |

'C' function for inserting a node pointed to by p after a node pointed to by q.

```

void insert(dnode *p, dnode *q)
{
    p->next = q->next;
    p->prev = q;
    q->next = p;
    if(p->next != NULL) /*insertion at the end*/
        p->next->prev = p;
}

```

If the right pointer of x is NULL then the insertion is being performed at the end. Hence, there is no question of modifying the left pointer of y.

'C' function for inserting a node pointed to by p before a node pointed to by q.

```
dnode * insert (dnode * head, dnode * p, dnode * q)
```

```
{
    if (q->prev == NULL)
        { // insert at the beginning
            p->next = q;
            p->prev = NULL;
            q->prev = p;
            return (p);
        }
    else
        { p->prev = q->prev;
            p->next = q;
            p->prev->next = p;
            q->prev = p;
            return (head);
        }
}
```

'C' function for inserting a value x, at the beginning of doubly linked list.

```
dnode * insert2(dnode * head, int x)
{
    dnode *p;
    p = (dnode *) malloc(sizeof(dnode));
    /* get memory for the new node */
    p->data = x;
    p->prev = NULL;
    p->next = head;
    if(head != NULL) /* not an empty list */
        head->prev = p;
    return(p);
}
```

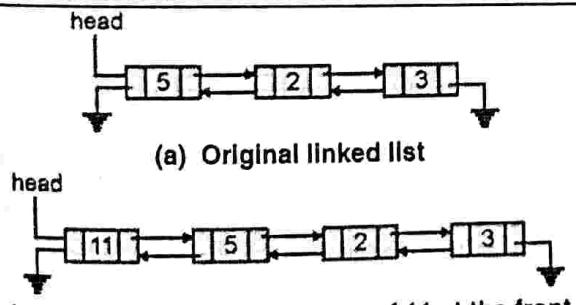


Fig. 3.4.6

'C' function for inserting a value x, at the end of a doubly linked list.

```
dnode * insert3(dnode *head, int x)
```

```
dnode *p, *q;
p = (dnode *) malloc(sizeof(dnode));
p->data = x;
p->prev = p->next = NULL;
if(head == NULL) /* empty list */
    return(p);
/* go to the last node */
q = head;
while(q->next != NULL)
    q = q->next;
p->prev = q;
q->next = p;
return(head);
}
```

3.4.2 Deletion of a Node

MU - Dec. 15

University Question

Q. Write a function for deletion of a node from Doubly linked list.
(Dec. 15, 5 Marks)

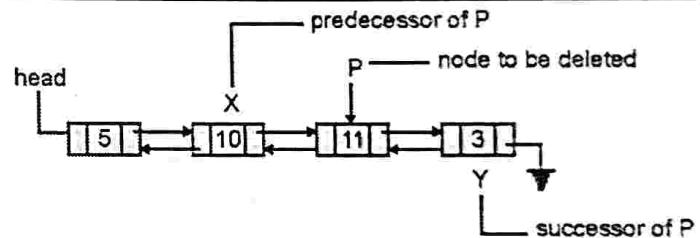


Fig. 3.4.7(a) : Node pointed to by P is to be deleted

When a node, pointed to by P is to be deleted than its predecessor node x and it's successors node y (as shown in Fig. 3.4.7(a)) will be affected.

- Right link at x should be set to y.
- Left link at y should be set to x.
- Release the memory allocated to the node pointed to by P.

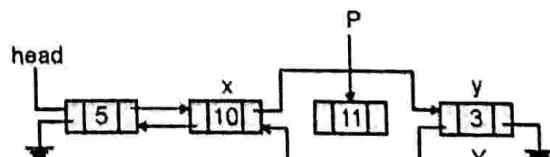


Fig. 3.4.7(b) : Links to be modified as shown above

C-instructions for deletions :

```
P->prev->next = P->next; // right at x set to y
P->next->prev = P->prev; // left link at y set to x
free(P);
```



'C' function for deletion of a node pointed by P in a doubly linked list.

```
dnode * delete_double(dnode *head, dnode *P)
{
    if(P == head) /* deleting the head node */
    {
        P->next->prev = NULL;
        head = P->next;
        free(P);
        return(head);
    }
    P->prev->next = P->next;
    if(P->next != NULL) /* not the last node */
        P->next->prev = P->prev;
    free(P);
    return(head);
}
```

- Special care should be taken to delete the first or the last node.
- If the node to be deleted is the first node then head should be advanced to the next node.
- If the node to be deleted is the last node then there is no node to its right.

Program 3.4.2 : Give node structure to represent a list names using DLL and write C functions for the following :

- (I) Display list forward
- (II) Display list reverse
- (III) Display names starting with letter S or s.

```
//Node structure
typedef struct node
{
    char data[30];
    struct node *next, *prev;
} node;
void display_forward(node *P)
{
    while(P != NULL)
    {
        printf("\n%s", P->data);
        P = P->next;
    }
}
```

```
}
void display_reverse(node *P)
{
    if(P != NULL)
    {
        display_reverse(P->next);
        printf("\n%s", P->data);
    }
}
void display_start_s(node *P)
{
    while(P != NULL)
    {
        if(P->data[0] == 'S' || P->data[0] == 's')
            printf("\n%s", P->data);
        P = P->next;
    }
}
```

Program 3.4.3 : Write a C program to implement a Doubly Linked List which performs the following operations :

- (I) Inserting element in the beginning
- (II) Inserting element in the end
- (III) Inserting element after an element
- (IV) Deleting a particular element
- (V) Displaying the list

MU - May 16, Dec. 17, 10/12 Marks

/*Accept input as a string and construct a doubly linked list for the input string with each node containing as a data one character from the string and perform :

- a. Insert
- b. Delete
- c. Display forward
- d. Display backward */

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
typedef struct student
{
    int rollno, marks;
    char name[15];
}student;
typedef struct node
{
    student data;
    struct node *next, *prev;
}node;
node *create();
```

```

node *insert_b(node *head, student x);
node *insert_e(node *head, student x);
node *insert_in(node *head, student x);
node *delete_in(node *head);
void displayforward(node *head);
void displaybackward(node *head);
void modify(node *head);
student read()
{
    student x;
    printf("\n Enter name :");
    scanf("%s", x.name);
    printf("\n Enter roll no. and Marks :");
    scanf("%d%d", &x.rollno, &x.marks);
    return x;
}
void print(student x)
{
    printf("\n %s\t%d\t%d", x.name, x.rollno, x.marks);
}
void main()
{
    int op, op1;
    student x;
    node *head = NULL;
    clrscr();
    do
    {
        flushall();
        printf("\nSelect Option :\n 1)Create\n 2)Insert\n 3)Delete\n 4)Modify");
        printf("\n 5)Display forward\n 6)Display backward\n 7)Quit");
        printf("\nEnter your Choice: ");
        scanf("%d", &op);
        switch(op)
        {
            case 1:head = create();
            break;
            case 2:printf("\n\t 1)Beginning\n\t 2)End\n\t 3)In between");
            printf("\nEnter your choice : ");
            scanf("%d", &op1);
            printf("\nEnter the data to be inserted : ");
            flushall();
            x = read();
            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: head = delete_in(head);
            break;
            case 4:modify(head);
            break;
            case 5:displayforward(head);
            break;
            case 6:displaybackward(head);
            break;
        }
    }while(op!= 7);
}
node *create()
{
    node *head, *p;
    student x;
    int n, i;
    head = NULL;
    printf("\nEnter no. of students : ");
    scanf("%d", &n);
    printf("\nEnter records of students : ");
    flushall();
    for(i = 1; i<= n; i++)
    {
        x = read();
        if(head == NULL)
        {
            head = p = (node*)malloc(sizeof(node));
            head->next = head->prev = NULL;
        }
        else
        {
            p->next = (node*)malloc(sizeof(node));
            p->next->prev = p;
            p = p->next;
            p->next = NULL;
        }
        p->data = x;
    }
    return(head);
}

```



```

node *insert_b(node *head, student x)
{
    node *p;
    p = (node*)malloc(sizeof(node));
    p->data = x;
    if(head == NULL)
    {
        head = p;
        head->next = head->prev = NULL;
    }
    else
    {
        p->next = head;
        head->prev = p;
        p->prev = NULL;
        head = p;
    }
    return(head);
}

node *insert_e(node *head, student x)
{
    node *p, *q;
    p = (node*)malloc(sizeof(node));
    p->data = x;
    if(head == NULL)
    {
        head = p;
        head->next = head->prev = NULL;
    }
    else
    {
        for(q = head; q->next != NULL; q = q->next);
        q->next = p;
        p->prev = q;
        p->next = NULL;
    }
    return(head);
}

node *insert_in(node *head, student x)
{
    node *p, *q;
    int rollno;
    p = (node*)malloc(sizeof(node));
    p->data = x;
    printf("\nEnter after which student ? : ");
    printf("\nEnter roll no. : ");
    scanf("%d", &rollno);
}

```

```

flushall();
for(q = head ; q != NULL && q->data.rollno != rollno; q = q->next);
if(q->data.rollno == rollno)
{
    p->next = q->next;
    p->prev = q;
    p->next->prev = p;
    p->prev->next = p;
}
else
    printf("\nData not found ");
return(head);
}

node *delete_in(node *head)
{
    node *p, *q;
    int rollno;
    if(head == NULL)
    {
        printf("\nUnderflow....Empty Linked List");
        return(head);
    }
    printf("\nEnter the roll no. of student to be deleted:");
    flushall();
    scanf("%d", &rollno);
    for(p = head ; p != NULL && p->data.rollno != rollno ; p = p->next);
    if(p->data.rollno != rollno)
    {
        printf("\nUnderflow.....data not found");
        return(head);
    }
    if(p == head)
    {
        head = head->next;
        if(head != NULL)
            head->prev = NULL;
        free(p);
    }
    else
    {
        p->prev->next = p->next;
        p->next->prev = p->prev;
        free(p);
    }
    return(head);
}

```

```

} void displayforward(node *head)
{
    node *p;
    printf("\n");
    if(head != NULL)
    {
        for(p = head ; p != NULL ; p = p->next)
            print(p->data);
    }
}

} void displaybackward(node *head)
{
    node *p;
    printf("\n");
    if(head != NULL)
    {
        // goto the last node
        for(p = head ; p->next != NULL ; p = p->next);
        for( ; p != NULL ; p = p->prev)
            print(p->data);
    }
}

} void modify(node *head)
{
    int rollno;
    node *p;
    printf("\nEnter Roll no of the student : ");
    scanf("%d", &rollno);
    for(p = head; p != NULL & p->data.rollno != rollno; p = p->next);
    if(p == NULL)
    {
        printf("\nEnter a new record : ");
        p->data = read();
    }
    else
        printf("Wrong roll no ....");
}

```

Program 3.4.4 : Write a program in 'C' to implement Doubly Link-list with methods Insert, delete and search.

MU - May 14, May 17, 10 Marks

Solution :

```

/*Doubly Linked List :*/
#include <stdio.h>
#include <conio.h>

```

```

typedef struct dnode
{
    int data;
    struct dnode *next, *prev;
}dnode;
dnode *create()
{
    int i, n, x;
    dnode *head, *p, *q;
    head = NULL;
    printf("\nEnter no. of data : ");
    scanf("%d", &n);
    printf("\nEnter data : ");
    for(i = 1; i <= n; i++)
    {
        printf("\nEnter next data : ");
        scanf("%d", &x);
        q = (dnode*)malloc(sizeof(dnode));
        q->data = x;
        q->next = q->prev = NULL;
        if(head == NULL)
        {
            p = head = q;
        }
        else
        {
            p->next = q;
            q->prev = p;
            p = q;
        }
    }
    return(head);
}

void display(dnode *head)
{
    printf("\n");
    for(; head != NULL; head = head->next)
        printf("%d ", head->data);
}

```

```

dnode *Delete(dnode *head, int x)
{
    dnode *p, *q;
    if(head == NULL)
        return(head);
    if(head->data == x)
    {

```



```

    p = head;
    head = head->next;
    head->prev = NULL;
    free(p);
    return(head);
}

p = head;
while(p != NULL && p->data != x)
{
    p = p->next;
    if(p != NULL)
    {
        if(p->next == NULL)
        {
            p->prev->next = NULL;
            free(p);
        }
        else
        {
            p->prev->next = p->next;
            p->next->prev = p->prev;
            free(p);
        }
    }
}
return(head);
}

int search(dnode *head, int x)
{
    while(head != NULL)
    {
        if(head->data == x)
            return(1);
        head = head->next;
    }
    return(0);
}

void main()
{
    dnode *head;
    int x;
    head = create();
    printf("\nEnter the data to be searched : ");
    scanf("%d", &x);
    if(search(head, x))
        printf("\nfound");
    else
        printf("\nNot found");
}

```

```

printf("\nEnter the data to be deleted : ");
scanf("%d", &x);
head = Delete(head, x);
printf("\nLinked list after deletion : ");
display(head);
getch();
}

```

3.5 Doubly Linked Circular List

- In a doubly linked circular list, the previous pointer of the first node points to the last node.
- Next pointer of the last node points to the first node.

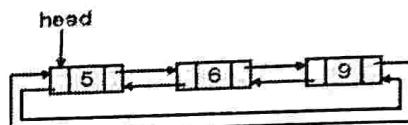


Fig. 3.5.1 : A doubly linked circular list

- In a doubly linked circular list, insertion at the front as well as at the rear can be done in constant time order ($O(1)$).
- If the list is not circular then insertion at the front can be done in constant time order ($O(1)$) but the insertion at the rear requires traversing of entire linked list to locate the last node. Time complexity of insertion at the end is of the order of ($O(n)$).

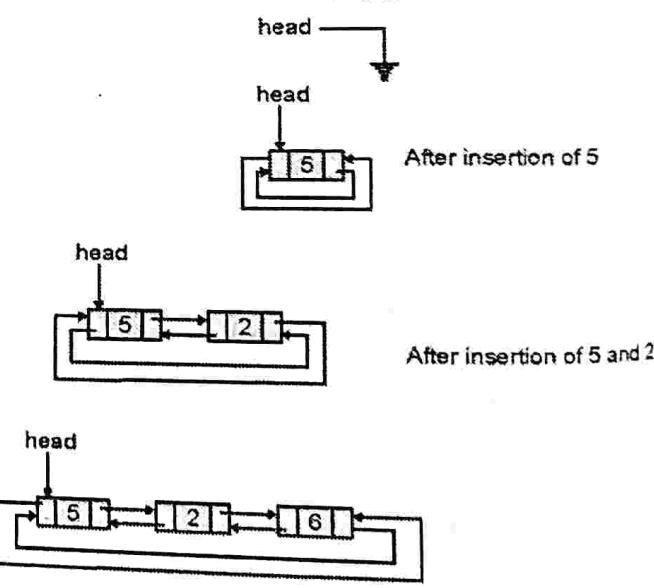


Fig. 3.5.2

Fig. 3.5.2 showing stepwise insertion of 3 elements namely 5, 2 and 6 in a doubly linked circular list.

Program 3.5.1 : A sample program for insertion of 5 elements in a doubly circular linked list and subsequently printing of elements stored in the list.

```

#include <conio.h>
#include <stdlib.h>
#include <stdio.h>
typedef struct dnode
{
    int data;
    struct dnode *next, *prev;
}
dnode;
dnode * insert(dnode *head, int x);
void print(dnode *);
void main()
{
    dnode *head;
    int i, x;
    head = NULL; // initially the list is empty
    printf("\nEnter 5 elements : ");
    for(i = 1; i <= 5; i++)
    {
        scanf("%d", &x);
        head = insert(head, x);
    }
    print(head);
    getch();
}

dnode *insert(dnode *head, int x)
{
    dnode *P;
    P = (dnode*)malloc(sizeof(dnode));
    P->data = x;
    P->prev = P->next = NULL;
    if(head == NULL) // inserting in an empty link list
    {
        P->prev = P->next = P;
        return(P);
    }
    P->prev = head->prev;
    P->next = head;
    head->prev->next = P;
    head->prev = P;
    return(head);
}

void print(dnode *head)
{
    dnode *P;
    P = head;
}

```

```

do
{
    printf("<- %d ->", P->data);
    P = P->next;
} while(P != head);
}

```

Output

Enter 5 elements : 5 4 3 2 1

<- 5 -><- 4 -><- 3 -><- 2 -><- 1 ->

Example 3.5.1 : Explain importance of header node in a linked list.

Solution :

Header node is an additional node, added at the beginning of a linked list. It does not contain any valid data. A list (5, 3, 2, 9) is shown below using a linked list with header node.

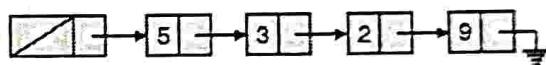


Fig. Ex. 3.5.1

Advantages

1. The address of the first node does not change and hence the address of the header node can always be passed by value.
2. In all cases, the insertion at the beginning requires the same steps. We do not have to consider whether the linked list is empty or not.
3. In all cases, the deletion of the first data requires the same steps. We do not have to consider whether the list will become empty or not after deletion of the first data.

Disadvantage

1. Additional memory is needed for the header node.

3.6 Applications of Linked Lists

MU- May 16, Dec. 17, May 18

University Question

- Explain any one application of linked list with an example. (May 16, 8 Marks)
- Write short note on Application of Linked list - Polynomial Addition. (Dec. 17, 10 Marks)
- Explain different applications of linked list. (May 18, 5 Marks)



3.6.1 Polynomials as Linked Lists

Representation of polynomial

A polynomial $P(x)$ of degree n is defined by the following expression.

$$P(x) = a_0 + a_1x^1 + a_2x^2 + \dots + a_nx^n$$

- Where a_n is any real number and n is an integer.
- A polynomial can be thought as the ordered list of non-zero terms. Each term is a 2-tuple containing power and coefficient.
- Thus, the polynomial $5 + 6x^2 - 9x^4$ is represented as $((0, 5), (2, 6), (4, -9))$. Tuple $(0, 5)$ stands for $5x^0$; the second tuple $(2, 6)$ indicates $6x^2$ and the last tuple denotes $-9x^4$. The linked representation of the said polynomial is shown below.

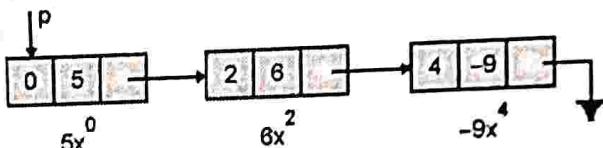


Fig. 3.6.1

Each term of the polynomial can be defined by the following structure.

```
typedef struct pnode
{
    float coeff;
    int pow;
    struct pnode * next;
} pnode;
```

We can treat a polynomial an abstract data type and perform following basic operations.

1. Creating a polynomial
2. Printing a polynomial
3. Addition of two polynomials
4. Multiplication of two polynomials
5. Evaluation of a polynomial.

Program 3.6.1 : A program to create and print a polynomial.

```
#include<conio.h>
#include<stdio.h>
#include<stdlib.h>
typedef struct pnode
{
    int pow;
    float coeff;
```

```
struct pnode *next;
} pnode;
pnode create(int); //create polynomials of n terms
void print(pnode *);
/* print the polynomial referenced by address of first
node */
void main()
{
    pnode *HEAD;
    int n;
    printf("\n Enter no. of terms : ");
    scanf("%d", &n);
    HEAD = create(n);
    print(HEAD);
}
```

'C' function to create a polynomial.

Creation of a linked list representing a polynomial is similar to that of a normal linked list. Each node of the linked list contains two data field, namely power and coefficient. Since, polynomial is an ordered list, data for each term must be entered in ascending order on power.

```
pnode *create(int n)
{
    pnode *head, *p;
    int i;
    // create the first node
    head = (pnode *)malloc(sizeof(pnode));
    //acquire memory for the first node
    p = head;
    head->next = NULL;
    printf("\n Enter power and coefficient : ");
    scanf("%d %f", &(p->pow), &(p->coeff));
    for(i = 1; i < n; i++)
    {
        //create subsequent nodes
        p->next = (pnode *)malloc(sizeof(pnode));
        p = p->next;
        p->next = NULL;
        printf("\n Enter power and coefficient : ");
        scanf("%d %f", &(p->pow), &(p->coeff));
    }
    return(head);
}
```

'C' function to print a polynomial.

Printing of a polynomial is similar to printing a linked list.

```
void print(pnode *p)
{
    printf("\n");
    while(p != NULL)
    {
        printf("%5.2fx ^ %d\t", p->coeff, p->pow);
        p = p->next; //goto the next node
    }
}
```

3.6.2 Addition of Two Polynomials

MU - May 19, Dec. 19

University Questions

- Q. Application of linked list - Polynomial addition.
(May 19, 10 Marks)
- Q. Explain Polynomial representation and addition using linked list with suitable example. (Dec. 19, 10 Marks)

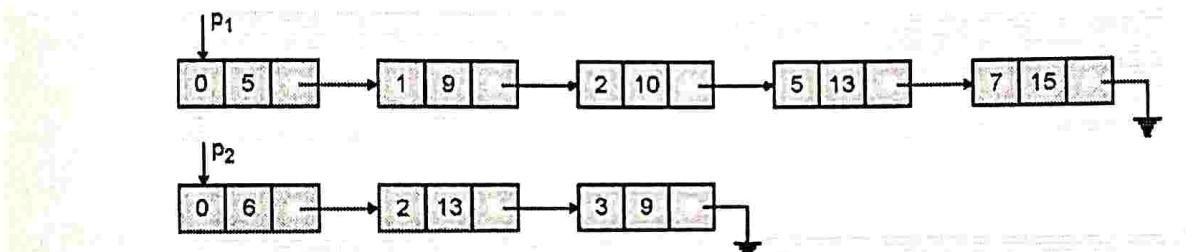


Fig. 3.6.2 : Representation of two polynomial using linked list

- Since, the power of the terms pointed by p_1 and p_2 are same, coefficients are added and the new term is inserted into the resultant polynomial p_3 . Pointer r_3 helps in inserting, subsequent terms at the rear end of the resultant polynomial.

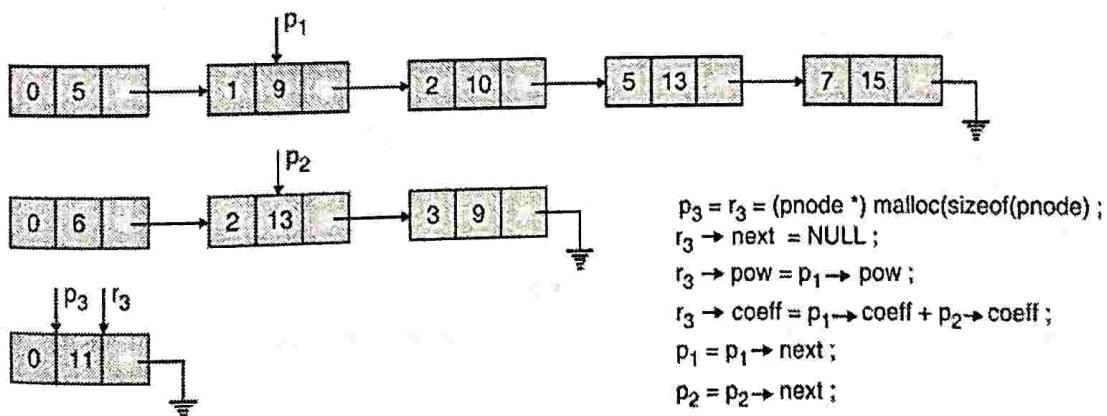


Fig. 3.6.2(a)

- Since, the power of the term pointed by p_1 is less than the power of the term pointed by p_2 , term pointed by p_1 is added to p_3 . Term is added as a next node of r_3 . p_1 and r_3 are advanced by a node.

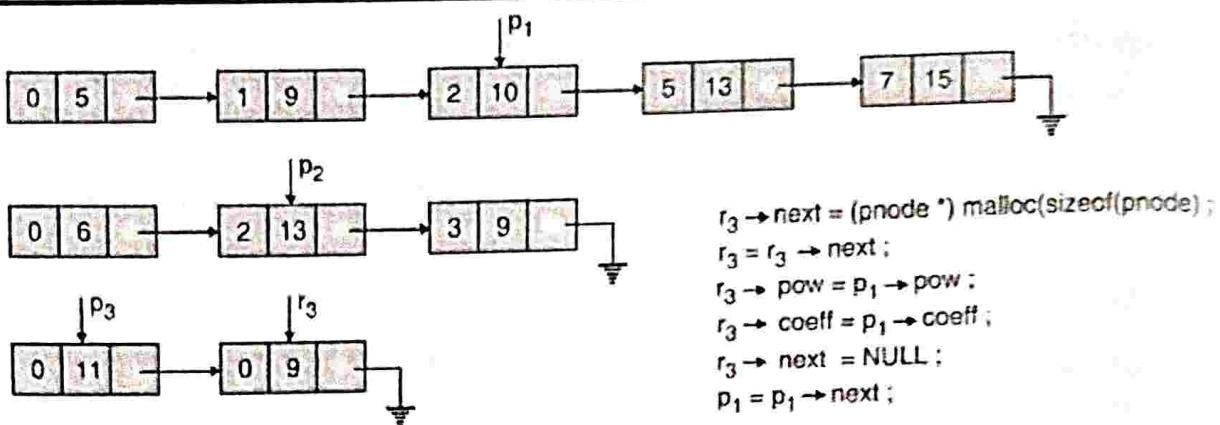


Fig. 3.6.2 (b)

- Coefficients of the terms pointed by p_1 and p_2 are added and the new term is inserted into p_3 .

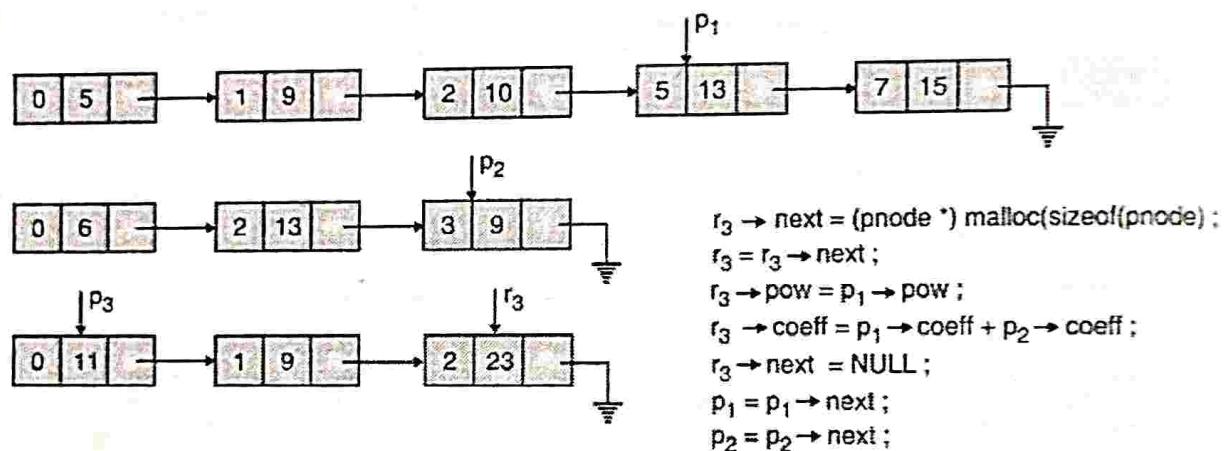


Fig. 3.6.2 (c)

- Since the power of the term pointed by p_2 is less than the power of the term pointed by p_1 , term pointed by p_2 is inserted into p_3 .

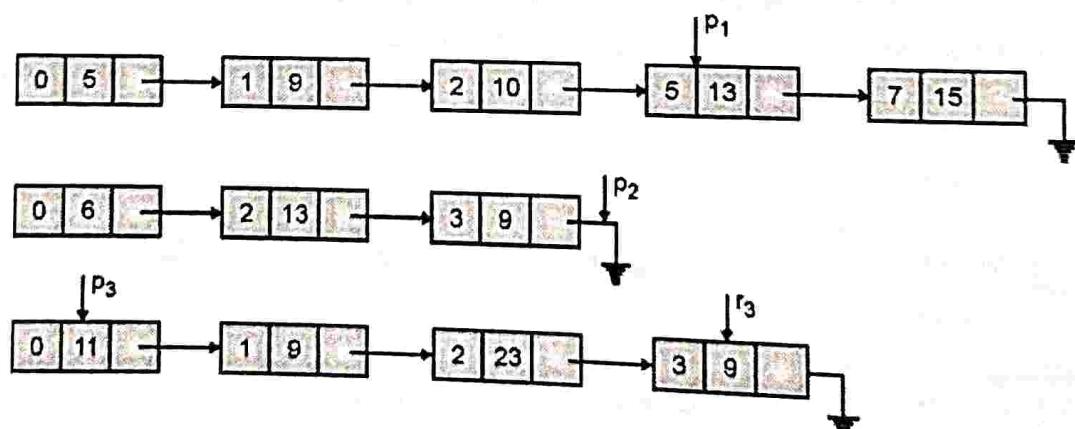


Fig. 3.6.2 (d)

- Since no more terms are left in p_2 , all the terms of p_1 are inserted at the end of p_3 .

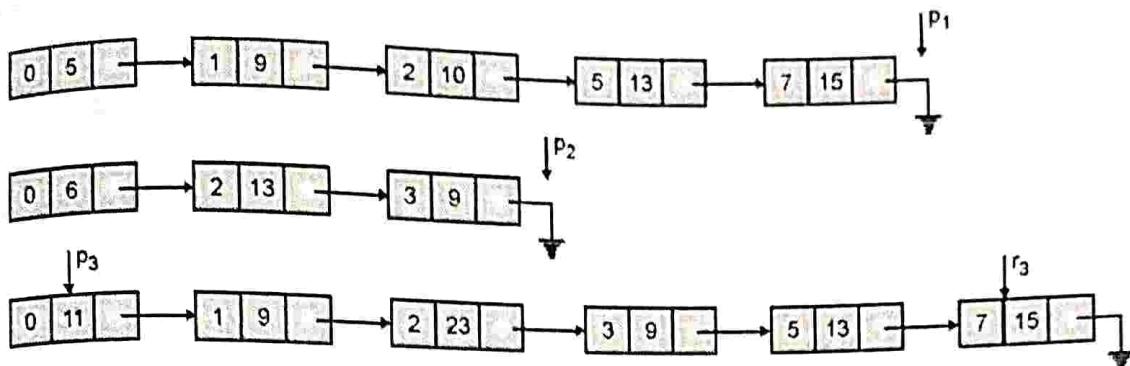


Fig. 3.6.2(e)

```

while(p1 != NULL)
{
    r3->next = (pnode*) malloc(sizeof(pnode));
    r3 = r3->next;
    r3->pow = p1->pow;
    r3->coeff = p1->coeff;
    r3->next = NULL;
    p1 = p1->next;
}

```

'C' function for addition of two polynomials.

```

pnode *addpoly(pnode *p1, pnode *p2)
{
    pnode *p3, *r3;
    p3 = NULL;
    while(p1 != NULL && p2 != NULL)
    {
        if(p3 == NULL)
        {
            p3=r3 = (pnode*)malloc(sizeof(pnode));
            r3->next = NULL;
        }
        else
        {
            r3->next=(pnode*)malloc(sizeof(node));
            r3 = r3->next;
            r3->next = NULL;
        }
        if(p1->pow<p2->pow)
        {
            r3->pow = p1->pow;
            r3->coeff = p1->coeff;
            p1 = p1->next;
        }
        else
    }

```

```

        if(p2->pow<p1->pow)
        {
            r3->pow = p2->pow;
            r3->coeff = p2->coeff;
            p2 = p2->next;
        }
        else
        {
            r3->pow = p1->pow;
            r3->coeff = p1->coeff + P2->coeff;
            p1 = p1->next;
            p2 = p2->next;
        }
    }
    //insert the remaining terms of P1 into P3
    while(p1 != NULL)
    {
        if(p3 == NULL)
        {
            p3 = r3 = (pnode*)malloc(sizeof(pnode));
            r3->next = NULL;
        }
        else
        {
            r3->next = (pnode*)malloc(sizeof(pnode));
            r3 = r3->next;
            r3->next = NULL;
        }
        r3->pow = p1->pow;
        r3->coeff = p1->coeff;
        p1 = p1->next;
    }
}

```



```

// insert the remaining terms of p2 into p3
while(p2 != NULL)
{
    if(p3 == NULL)
    {
        p3 = r3 = (pnode*)malloc(sizeof(pnode));
        r3->next = NULL;
    }
    else
    {
        r3->next = (pnode*)malloc(sizeof(pnode));
        r3 = r3->next;
        r3->next = NULL;
    }
    r3->pow = p2->pow;
    r3->coeff = p2->coeff;
    p2 = p2->next;
}
return(p3);
}

```

Program 3.6.2 : Write a C program to represent and add two polynomials using linked list.

MU - Dec. 18, 12 Marks

```

#include<math.h>
#include<stdio.h>
#include<conio.h>
typedef struct term
{
    int power;
    float coeff;
}term;
typedef struct polynomial
{
    term a[20];
    int n;
}polynomial;
void init(polynomial *ptr);
void read(polynomial *ptr);
void print(polynomial *ptr);
polynomial add(polynomial *p1, polynomial *p2);
void insert(polynomial *, term);
void main()
{
    polynomial p1, p2, p3;
    int option;
    float x, value;
    do

```

```

    {
        printf("\n1 : Create 1'st polynomial");
        printf("\n2 : Create 2'nd polynomial");
        printf("\n3 : Add polynomials");
        printf("\n4 : Quit");
        printf("\nEnter your choice : ");
        scanf("%d", &option);
        switch(option)
        {
            case 1: read(&p1);
            break;
            case 2: read(&p2);
            break;
            case 3: p3 = add(&p1, &p2);
            printf("\n1'st polynomial ->\n");
            print(&p1);
            printf("\n2'nd polynomial ->\n");
            print(&p2);
            printf("\n Sum = ");
            print(&p3);
            break;
        }
    }while(option != 4);
}

```

void read(polynomial *Ptr)

```

{
    int n, i, power;
    float coeff;
    term t;
    init(Ptr);
    printf("\nEnter number of terms : ");
    scanf("%d", &n);
    /* read n terms */
    for (i = 0; i < n; i++)
    {
        printf("\nEnter a term(power coeff.)");
        scanf("%d%f", &power, &coeff);
        t.power = power;
        t.coeff = coeff;
        insert(Ptr, t);
    }
}

```

void print(polynomial *Ptr)

```

{
    int i;
    printf("\n");
    for(i = 0; i < Ptr->n; i++)
        printf("%5.2fX ^ %d\n", (Ptr->a[i]).coeff,
               (Ptr->a[i]).power);
}

```

polynomial add(polynomial *p1, polynomial *p2)

```

polynomial p3;
term t;
int i, j;
i = j = 0;
init(&p3);
while(i < p1->n && j < p2->n)
{
    if(p1->a[i].power == p2->a[j].power)
    {
        t.power = p1->a[i].power;
        t.coeff = p1->a[i].coeff + p2->a[j].coeff;
        insert(&p3, t);
        i++; j++;
    }
    else
    if(p1->a[i].power < p2->a[j].power)
    {
        insert(&p3, p1->a[i]);
        i++;
    }
    else
    {
        insert(&p3, p2->a[j]);
        j++;
    }
}
while(i < p1->n)
{
    insert(&p3, p1->a[i]);
    i++;
}
while(j < p2->n)
{
    insert(&p3, p2->a[j]);
    j++;
}
return(p3);
}

void init(polynomial *Ptr)
{
    Ptr->n = 0;
}

void insert(polynomial *Ptr, term t)
{
    int i;
    /* move all higher power term by 1 place movement
       should start with the last term */
    for(i = Ptr->n-1; (Ptr->a[i]).power > t.power
        && i >= 0; i--)
        Ptr->a[i+1] = Ptr->a[i];
    /* insert the term t */
    Ptr->a[i+1] = t;
    (Ptr->n)++;
}

```

Output

1 : create 1'st polynomial

2 : create 2'nd polynomial

3 : Add polynomials

4 : Quit

Enter your choice :1

Enter number of terms :3

enter a term(power coeff.)2 3

enter a term(power coeff.)4 6

enter a term(power coeff.)5 7

1 : create 1'st polynomial

2 : create 2'nd polynomial

3 : Add polynomials

4 : Quit

Enter your choice :2

Enter number of terms :2

enter a term(power coeff.)2 3

enter a term(power coeff.)6 8

1 : create 1'st polynomial

2 : create 2'nd polynomial

3 : Add polynomials

4 : Quit

Enter your choice :3

1'st polynomial -> 3.00X ^ 2 6.00X ^ 4

7.00X ^ 5

2'nd polynomial -> 3.00X ^ 2 8.00X ^ 6

Sum = 6.00X ^ 2 6.00X ^ 4 7.00X ^ 5

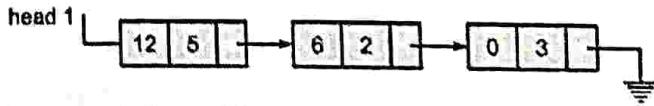
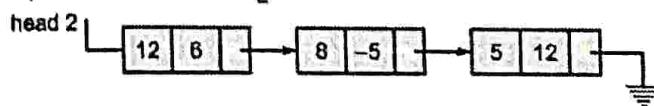
8.00X ^ 6

Example 3.6.1 : Consider the following polynomials represented using linked lists.

$$C_1 = 5x^{12} + 2x^6 + 3$$

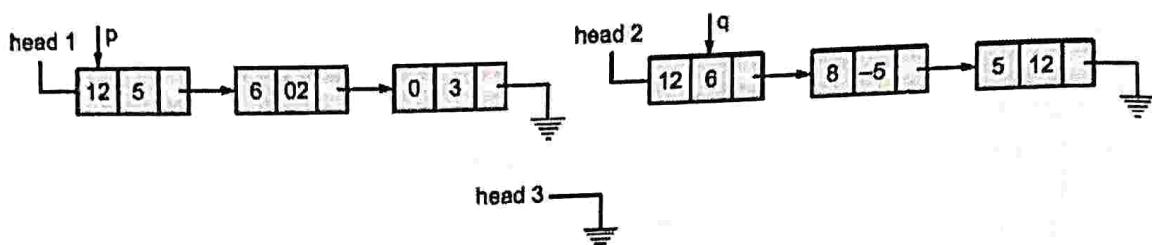
$$C_2 = 6x^{12} - 5x^8 + 12x^5$$

Show the addition process of above polynomials diagrammatically.

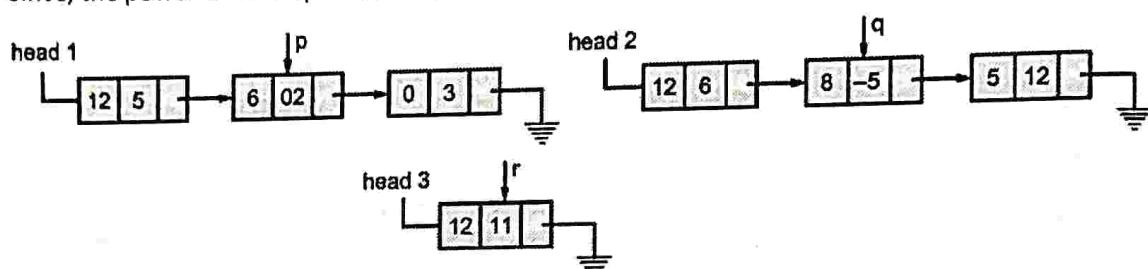
Solution :**Representation of C₁****Representation of C₂**

**Addition :**

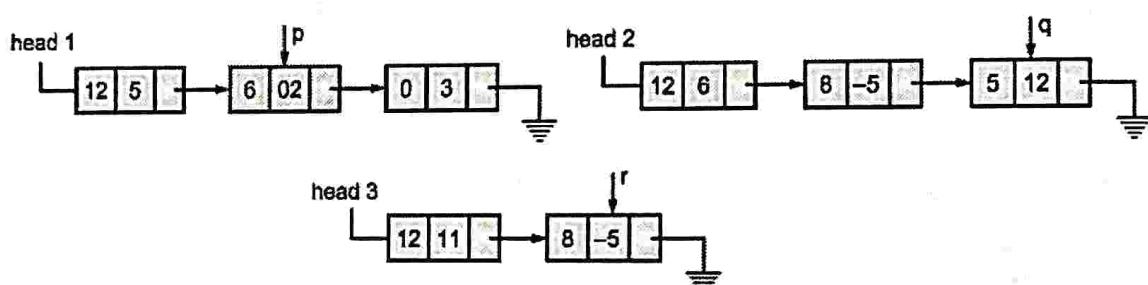
Final result is stored in a linked list reference by head 3



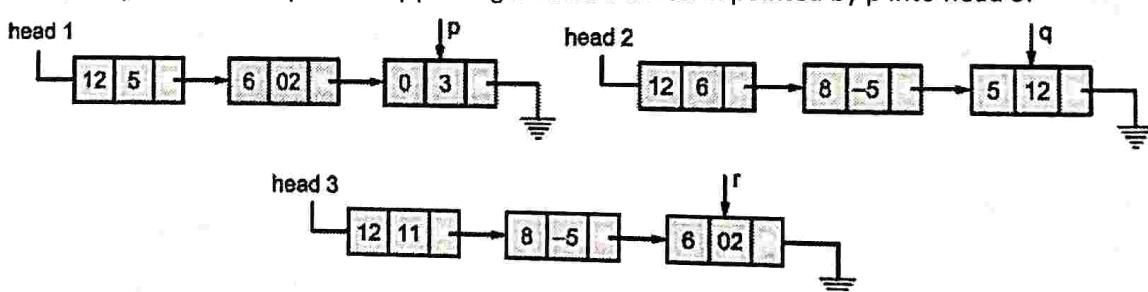
Step 1 : Since, the power of terms pointed by p and q are same : Add the terms and insert the resultant term in head 3.



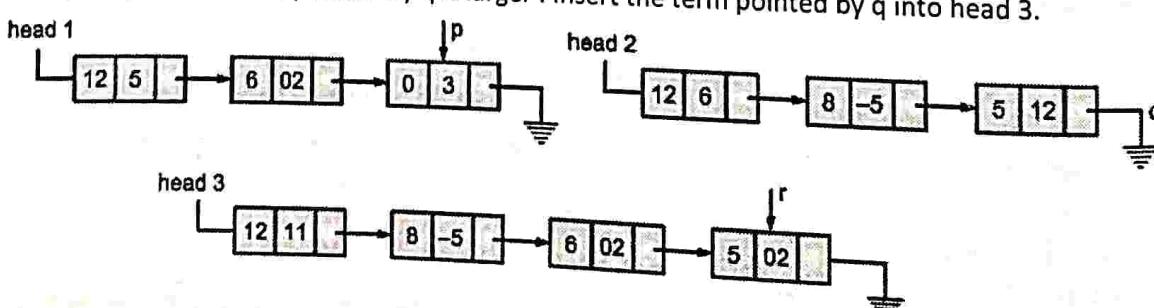
Step 2 : Since, the power of term pointed by q is larger : Insert the term pointed by q into head 3.



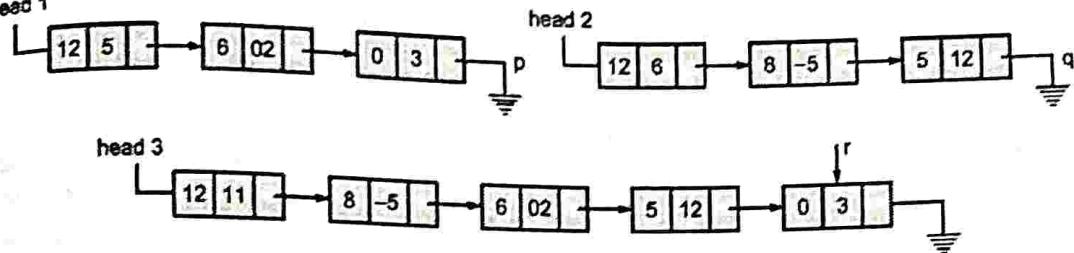
Step 3 : Since, the power of term pointed by p is larger : Insert the term pointed by p into head 3.



Step 4 : Since, the power of term pointed by q is larger : Insert the term pointed by q into head 3.



Step 5: Since, the pointer q has become NULL : remaining terms of head 1 are inserted into head 3.



$$\text{Final polynomial} = 11x^{12} + (-5)x^8 + 02x^6 + 12x^5 + 3$$

Multiplication of two polynomials

| | |
|---|--|
| $5 + 2x + 9x^3 + 3x^5$ | p_1 |
| $3 + x^2 + 2x^4$ | p_2 |
| $15 + 6x + 27x^3 + 9x^5$ | p_3 [p_1 is multiplied with 3] |
| $5x^2 + 2x^3 + 9x^5 + 3x^7$ | p_4 [p_1 is multiplied with x^2] |
| $10x^4 + 4x^5 + 18x^7 + 6x^9$ | p_5 [p_1 is multiplied with x^4] |
| p_3, p_4 and p_5 are added to produce the resultant polynomial. | |

Algorithm for multiplication of two polynomials p_1 and p_2 :

R3 and temp are of polynomial type.

$R3 \leftarrow \text{NULL}$

$\text{Temp} \leftarrow \text{NULL}$

$\text{while}(p2 \neq \text{NULL})$

{

 multiply the current term of p_2 with the polynomial p_1 with its result in temp

 temp \leftarrow (polynomial p_1) \times (term pointed by p_2)

 add the polynomial "temp" to the final polynomial "R3"

$R3 \leftarrow R3 + \text{temp};$

$p2 \leftarrow (p2 \rightarrow \text{next})$

 /* skip the current term of p_2 */

}

'C' function for multiplication of two polynomials.

$\text{pnode} * \text{multiply}(\text{pnode} * p1, \text{pnode} * p2)$

```

pnode * temp, *r3;
r3 = NULL;
while(p2 != NULL)
{
    temp = multiterm(p1, p2);
    /* function multiterm multiplies the polynomial
       p1 with the current term of p2 */
    r3 = addpoly(r3, temp); ]
    p2 = p2 → next;
}
return(r3);
}

```

$\text{pnode} * \text{multiterm}(\text{pnode} * p1, \text{pnode} * x)$

{

 pnode * new, * r1;

 new = NULL;

 while(p1 != NULL)

{

 if(new == NULL)

 {

 r1 = new = (pnode *) malloc(sizeof(pnode));

 r1 → next = NULL;

 }

 else

 {

 r1 → next = (pnode *) malloc(sizeof(pnode));

 r1 = r1 → next;

 r1 → next = NULL;

 }

 r1 → coeff = p1 → coeff * x → coeff;

 r1 → pow = p1 → pow + x → pow;

 p1 = p1 → next;

}



```

    return(new);
}

```

'C' function for evaluation of a polynomial.

```

float evaluate (pnode * p1, float val)
{
    float x;
    x = 0.00;
    while(p1 != NULL)
    {
        x = x + p1->coeff * pow(val, p1->pow);
        p1 = p1->next;
    }
    return(x);
}

```

3.7 Linked Representation of a Stack

- Stack can be represented efficiently through Linked lists.
- When a stack is represented using a Linked list, it is never full as long as system has memory for dynamic allocation of space for a node. A stack can be represented through a singly connected Linked list.

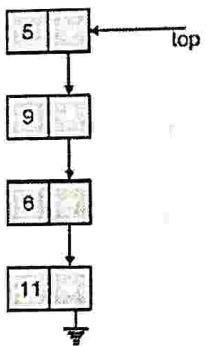


Fig. 3.7.1 : Linked list representation of a stack

Structure of a node

```

typedef struct node
{
    int data ;
    struct node *next ;
}node ;

```

top is a pointer type variable, pointing to the top node

Declaration of top

```
node *top ;
```

- Memory for a new node can be allocated through malloc()


```
(node *) malloc(sizeof(node))
```

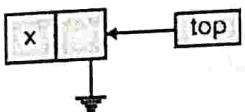
sizeof(node) gives number of bytes required to store a node. Function malloc() returns the address of the memory block allocated to store a node. Address returned by malloc() is type casted through (node*) so that the address can be treated as an address of a node.

- Initially top is NULL (Empty Stack) ;
- Inserting an element x in an empty stack

$$\text{top} = (\text{node}^*) \text{ malloc}(\text{sizeof}(\text{node})) ;$$

$$\text{top} \rightarrow \text{data} = x ;$$

$$\text{top} \rightarrow \text{next} = \text{NULL} ;$$

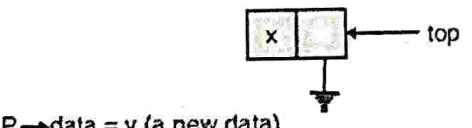


- Subsequent elements can be inserted in the stack through the following operations.

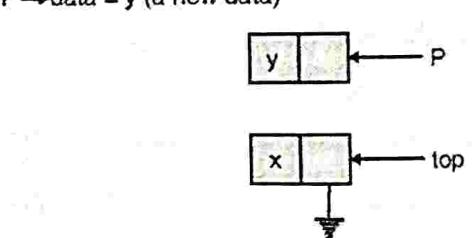
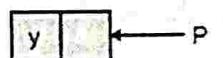
```

node * P ;
//A pointer for acquiring memory for new node
P = (node * ) malloc(sizeof(node)) ;

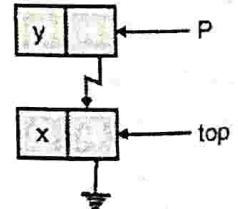
```



$P \rightarrow \text{data} = y$ (a new data)

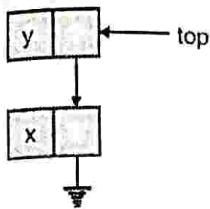


$P \rightarrow \text{next} = \text{top}$



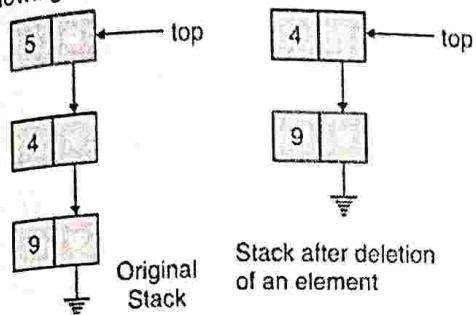
- Address of the top node (address is stored in the pointer type variable top) is copied in the next field of node whose address is stored in P

$$\text{top} = P ;$$



Address of the new node is copied in the pointer top.

An element from the stack can be deleted through the following operations.



```

P = top /* save the address of the top node in a
pointer P. So that the memory occupied by the node
can be freed */
top = top → next ; // top node is deleted
free(P); // memory freed
  
```

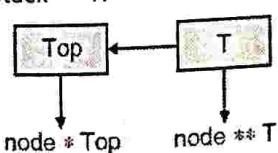
3.7.1 Functions for Stack Operations

(a) init Function

```

void init(stack **T) ;
calling the function from the main program
void main(c)
{
    stack * top ;
    init(& top) ;
}
  
```

Since, the function init() modifies the value of top, address of the pointer type variable 'top' must be passed to the function. If the contents of a variable (defined in calling program) is to be modified by the called function then the calling program must pass its address. Pointer variable "top" is passed by reference and the corresponding parameter in function init is declared as stack **T.



Contents of the variable top can be accessed through T and it is simply *T.

(b) push Function

```
void push(stack ** T, int x)
```

Insert a data in a stack, referenced by *T(Top)

(c) pop Function

```
int pop(stack ** T)
```

Delete the top element of a stack referenced by *T(Top) and return it to the calling program.

(d) empty Function

```
int empty(stack * top)
```

Check, whether the stack is empty. top is passed by value as the function empty() will not change the variable top.

'C' functions for stack operations.

```

void init(stack **T)
{
    * T = NULL ; /* make the stack empty) */
}
void push(stack ** T, int x)
{
    stack * P ;
    /* get a new node */
    P = (stack *) malloc(sizeof(stack)) ;
    P → data = x ;
    /* attach the node at the front */
    P → next = *T ;
    * T = P ;
}
int pop(stack **T)
{
    int x ;
    stack * P ;
    P = * T ;
    *T = P → next ;
    x = P → data ;
    free(P) ;
    return(x) ;
}
int empty(stack *Top)
{
    if(Top == NULL)
        return(1) ;
    return(0) ;
}
  
```



A program showing usage of stack

Reverse a string using a stack represented using a linked list

Program 3.7.1 : Program to reverse a string represented using a stack.

OR Write a program to implement stack using linked list.

MU - Dec. 17, 5 Marks

```
#include<stdio.h>
#include<conio.h>
typedef struct stack
{
    char data;
    struct stack *next;
} stack;
void init(stack **);
int empty(stack *);
char pop(stack **);
void push(stack **, char);
void main()
{
    stack *TOP;
    char x;
    init(&TOP);
    printf("\nEnter the string :");
    while((x = getchar()) != '\n')
        push(&TOP, x);
    printf("\n");
    while(!empty(TOP))
    {
        x = pop(&TOP);
        printf("%c", x);
    }
}
void init(stack **T)
{
    *T = NULL;
}
int empty(stack *TOP)
{
    if(TOP == NULL)
        return(1);
    return(0);
}
```

```
void push(stack **T, char x)
{
    stack *P;
    P = (stack *)malloc(sizeof(stack));
    P->data = x;
    P->next = *T;
    *T = P;
}
char pop(stack **T)
{
    char x;
    stack *P;
    P = *T;
    *T = P->next;
    x = P->data;
    free(P);
    return(x);
}
```

Output

Enter the string : structure
erutcurts

Program 3.7.2 :

Write a program In 'C' to implement Stack using Linked-List. Perform the following operations :

- (I) Push
- (II) Pop
- (III) Peek
- (IV) Display the stack contents

MU - May 19, 10 Marks

```
#include<studio.h>
#include<conio.h>
typedef struct stack
{
    int data;
    struct stack * next;
} stack;
void init(stack **);
int empty(stack *);
char pop(stack **);
void push(stack **, int);
```

```

int peep(stack *);
void display(stack *);

void main()
{
    int x, option;
    node *top;
    init(& top);
    do
    {
        printf("\n 1) Push \n 2) Pop \n 3) Peek \n
4) Display");
        printf("\n your option :");
        scanf("%d", &option);
        switch(option)
        {
            case 1 :
                printf("\n element to be inserted : ");
                scanf("%d", &x);
                push(&top, x);
                break;
            case 2 :
                if(! empty(top))
                {
                    x = pop(&top);
                    printf("\n Data = %d", x);
                }
                break;
            case 3 :
                if(! empty(top))
                {
                    x = peep(top);
                    printf("\n Data = %d", x);
                }
                break;
            case 4 : display(top);
                break;
        }
    }while(option != 5);
}

```

```

}

void init(stack **T)
{
    * T = NULL; /* make the stack empty) */
}

void push(stack ** T, int x)
{
    stack * P;
    /* get a new node */
    P = (stack *) malloc(sizeof(stack));
    P->data = x;
    /* attach the node at the front */
    P->next = *T;
    * T = P;
}

int pop(stack **T)
{
    int x;
    stack * P;
    P = * T;
    *T = P->next;
    x = P->data;
    free(P);
    return(x);
}

int empty(stack *Top)
{
    if(Top == NULL)
        return(1);
    return(0);
}

int peep(node *top)
{
    return(top->data);
}

void display(node *top)
{
    while(top != NULL)
    {

```



```

    printf("%d \t", top->data);
    top = top->next;
}
}

```

3.8 Linked Representation of a Queue

A queue can be represented by a linked structure of elements. Each element is stored in a node. Memory area for a node can be acquired during runtime through the C function "malloc".

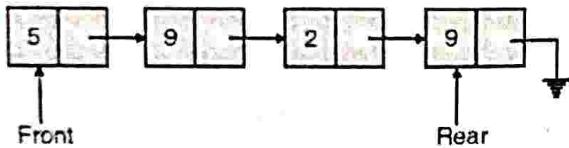


Fig. 3.8.1 : Linked representation of a queue

A queue can be declared as follows.

```

typedef struct node
{
    int data;
    struct node * next;
} node;
typedef struct Q
{
    node *R;
    node *F;
}Q;

```

Structure node, has two fields :

- data → Stores queue element
- next → Stores the address of the next node holding the next element of the queue.

Structure queue (Q) has two fields :

R → address of the rear node.

F → address of the front node.

When a queue is initialized, both its pointers "R" and "F" should be set to NULL.

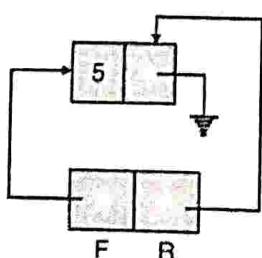


Fig. 3.8.2 : Status of the queue after insertion of the element 5

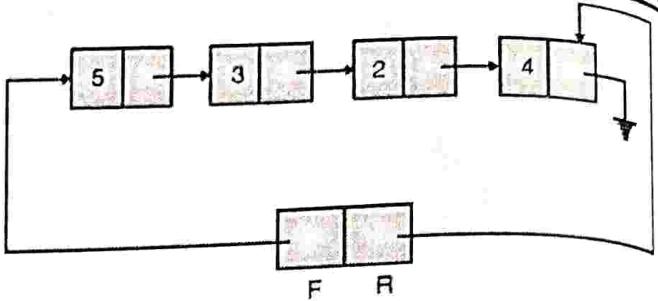


Fig. 3.8.3 : Status of the queue after further insertions of 3, 2 and 4

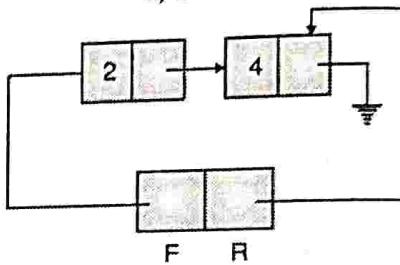


Fig. 3.8.4 : Status of the queue after two successive deletions

3.8.1 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.

3.8.2 Operations on Queue Implemented using Linked Structure

A set of useful operations on a queue includes :

1. **initialize()** : Initializes a queue by setting the value of rear and front pointer to "NULL".
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. **Print()** : It prints the queue elements from front to rear.

(a) Data structure declaration.

```
typedef struct node
{
    int data;
    struct node *next;
}node;
typedef struct Q
{
    node *R;
    node *F;
} Q;
```

(b) Declaring a queue type variable

`Q q1, q2;`

Element q_1 of the type Q(queue)

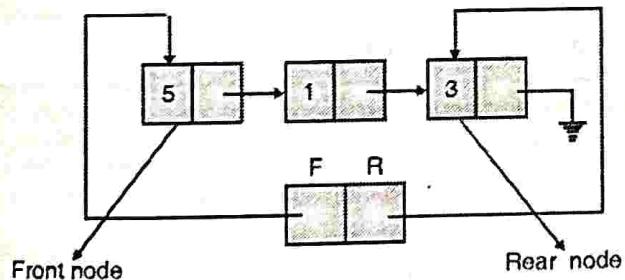


Fig. 3.8.5 : Memory representation of a queue

Data field of the front node is ($q_1.F \rightarrow$ data)

Next field of the front node is ($q_1.F \rightarrow$ next)

Data field of the rear node is ($q_1.R \rightarrow$ data)

Next field of the rear node is ($q_1.R \rightarrow$ next);

(c) Declaring a queue type pointer

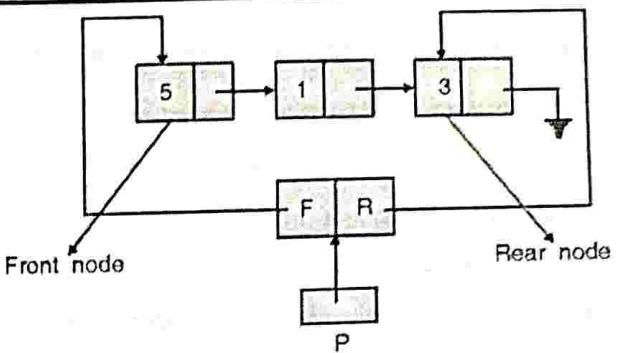
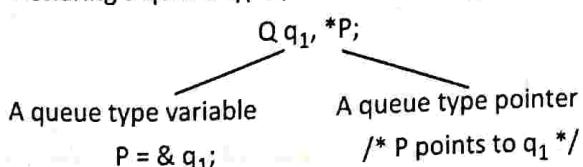


Fig. 3.8.6 : Memory representation of a queue with its address in P

Data field of the front node is ($P \rightarrow F \rightarrow$ data);

Next field of the front node is ($P \rightarrow F \rightarrow$ next);

Data field of the rear node is ($P \rightarrow R \rightarrow$ data);

Next field of the rear node is ($P \rightarrow R \rightarrow$ next);

'C' function to initialize queue.

```
void initialize(Q *P)
{
    P->R = NULL;
    P->F = NULL;
}
```

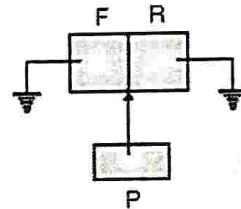


Fig. 3.8.7 : Initial state of a queue.

Queue is addressed through the pointer P

'C' function to check whether queue is empty or not.

```
int empty(Q *P)
{
    if(P->R == NULL) /* empty queue */
        return(1);
    return(0);
}
```

enqueue()

enqueue function inserts a value (say x) at the rear end of the queue.



Steps required for insertion of an element x in a queue :

- Memory is acquired for the new node.
- Value x is stored in the new node.
- New node is inserted at the rear end.
- Special care should be taken for insertion into an empty queue. Both rear and front pointers will point to the only element of the queue.

```
Node *p; Q q;
p = (node *) malloc(sizeof(node));
/* acquire memory */
p->data = x; /* store x in the node */
p->next = NULL;
if(empty(&q))
/* inserting element in an empty queue */
{
    q.R = q.F = p;
}
else
{
    (q.R)->next = p;
    q.R = p;
}
```

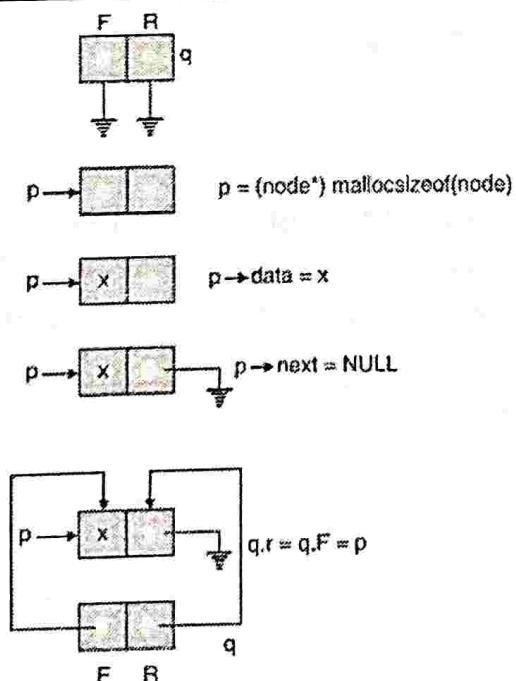


Fig. 3.8.8(a) : Insertion In an empty queue (stepwise)

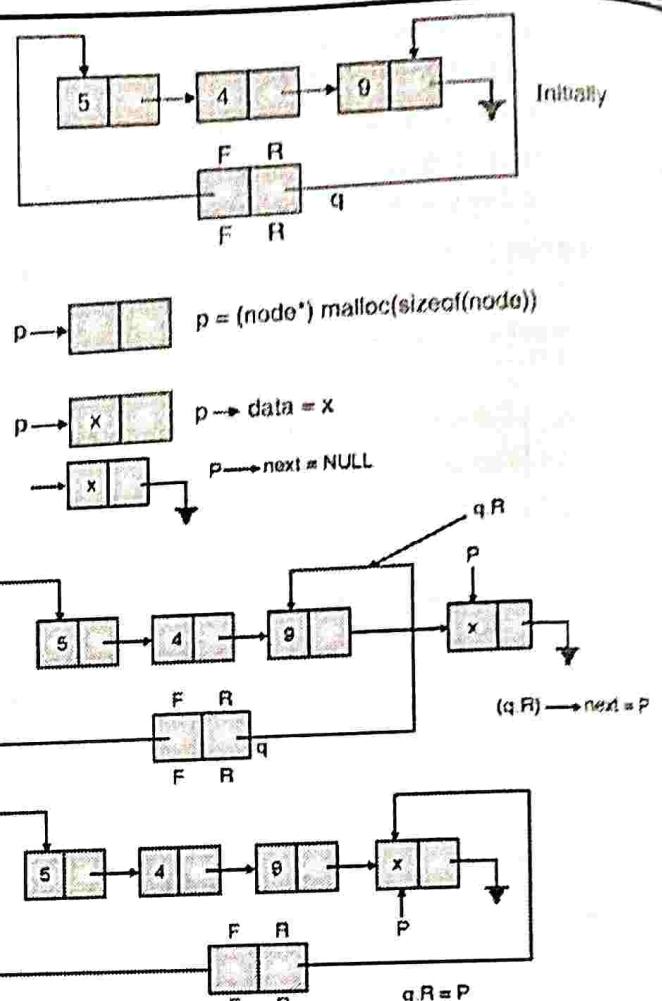


Fig. 3.8.8(b) : Insertion Into a non-empty queue
(stepwise)

Queue type variable "q" should be passed by address for insertion() operation value of q (rear field) changes after insertion.

'C' function for insertion.

```
void insert(Q *qP, int x)
{
    node *P;
    P = (node *) malloc(sizeof(node));
    P->data = x;
    P->next = NULL;
    if(empty(qP))
    {
        qP->R = qP->F = P;
    }
    else
    {
        qP->R->next = P;
        qP->R = P;
    }
}
```

'C' function dequeue()

`dequeue()` function deletes the front node of the queue and returns the value stored in the node, to the calling program. Front pointer of the queue is advanced to point to the next node. Special care should be taken while deleting the last node. As it will make the queue empty after deletion. Memory used by the deleted node should be released.

Steps required for deletion of a node from the queue :

```
Node *P; Q q;
P = q.F;
/* store the address of the front node in P */
x = P->data
if(q.R == q.F)
/* deleting the last node */
{
    initialize(&q);
    /* make the queue empty */
    free(P);
    return(x);
}
else
{
    q.F = P->next;
    /* Advance the front pointer */
    free(P);
    return(x);
}
```

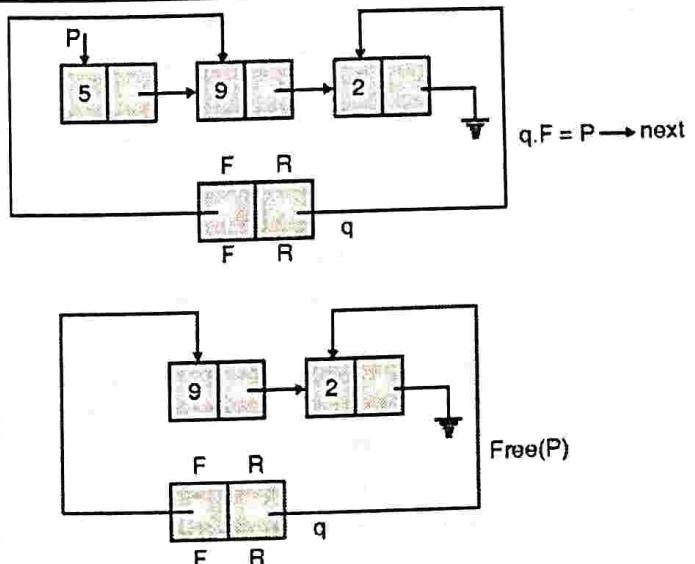


Fig. 3.8.9 : Deletion of the front node from a queue (stepwise)

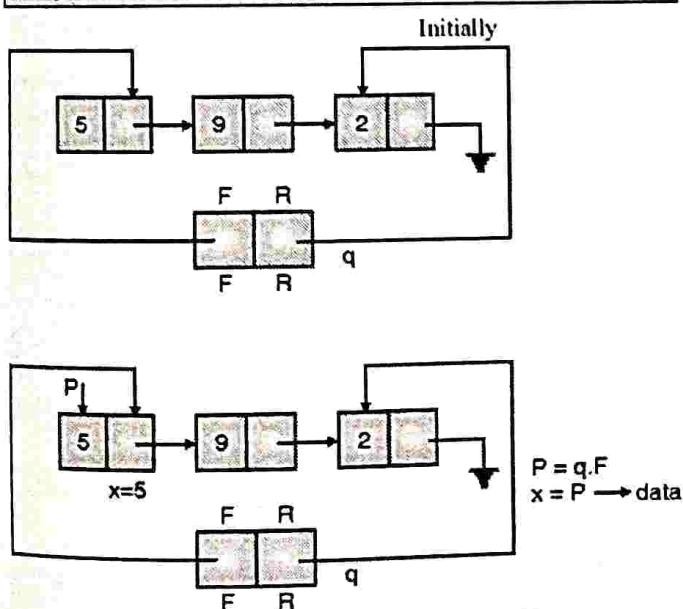
'C' function for deletion.

```
int dequeue(Q *qP)
{
    int x;
    node *P;
    P = qP->F;
    x = P->data;
    if(qP->R == qP->F) /* last element */
        initialize(qP);
    else
        qP->F = P->next;
        free(P);
    return(x);
}
```

'C' function for printing a queue.

Elements of the queue can be printed by traversing the underlying linked list starting from the front node.

```
void print(Q *qP)
{
    node *P;
    P = qP->F; /* start from front */
    while(P != NULL)
    {
        printf("\n%d", P->data);
        P = P->next;
    }
}
```



Program 3.8.1 : A sample program showing various operations on a queue represented using a linked list.

- (I) Insert five values in the queue.
- (II) Print elements of the queue.
- (III) Delete 2 elements from the queue.
- (IV) Print elements of the queue.
- (V) Delete the remaining elements of the queue.

OR
Write a program in 'C' to implement QUEUE ADT using Linked-List. Perform the following operations :

- (I) Insert a node in the Queue.
- (II) Delete a node from the Queue.
- (III) Display Queue elements

MU – May 18, 10 Marks

```
#include<conio.h>
#include<stdio.h>
#define MAX 10
typedef struct node
{
    int data;
    struct node *next;
}node;

typedef struct Q
{
    node *R, *F;
}Q;

void initialise(Q *);
int empty(Q *);
int full(Q *);
void enqueue(Q *, int);
int dequeue(Q *);
void print(Q *);
void main()
{
    Q q;
    int x, i;
    initialise(&q);
    printf("\nEnter 5 elements :");
    for(i = 1; i <= 5; i++)
    {
```

```
        scanf("%d", &x);
        enqueue(&q, x);
    }
    printf("\nDisplaying queue :");
    print(&q);
    x = dequeue(&q);
    x = dequeue(&q);
    printf("\nAfter deletion of two elements :");
    print(&q);
    //Delete remaining elements
    while(!empty(&q))
    {
        x = dequeue(&q);
        getch();
    }
}

void initialise(Q *qP)
{
    qP->R = NULL;
    qP->F = NULL;
}

void enqueue(Q *qP, int x)
{
    node *P;
    P = (node*)malloc(sizeof(node));
    P->data = x;
    P->next = NULL;
    if(empty(qP))
    {
        qP->R = P;
        qP->F = P;
    }
    else
    {
        (qP->R)->next = P;
        qP->R = P;
    }
}

int dequeue(Q *qP)
{
    int x;
    node *P;
    P = qP->F;
    x = P->data;
    if(qP->F == qP->R) //deleting the last element
```

```

initialise(qP);
else
    qP->F = P->next;
free(P);
return(x);
}

void print(Q *qP)
{
    int i;
    node *P;
    P = qP->F;
    while(P != NULL)
    {
        printf("\n%od \n", P->data);
        P = P->next;
    }
}
int empty(Q *qp)
{
    if(qp->R == NULL)
        return 1;
    return 0;
}

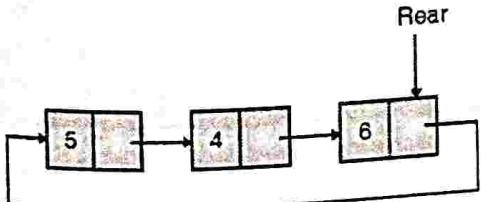
```

Output

```

Enter 5 elements : 4      5   34  21  90
displaying queue :
4
5
34
21
90
34
21
90

```

3.9 Queue using a Circular Linked List

In a circular linked list, rear node points back to the front node. Address of the front node can be found through the rear node.

Address of the front node = rear → next

When a queue is implemented through a singly circular linked list, only one pointer i.e. address of the rear node is required to be maintained.

In contrast, when a queue is maintained using a linked list, two pointers must be maintained :

- (a) Address of the front node (for deletion)
- (b) Address of the rear node (for insertion)

Special care should be taken, while inserting an element in an empty queue. Similarly, after deletion of the last element, the queue should become empty.

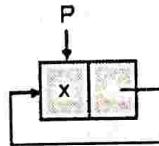
Steps for insertion of element x in queue represented using a circular linked list :

- (a) Acquire memory for the node
 $P = (\text{node} *) \text{malloc}(\text{sizeof}(\text{node}))$
- (b) Store the data x in the node
 $P \rightarrow \text{data} = x;$

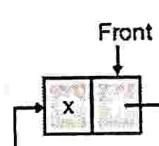
If the queue is empty

- (c) newly acquired node should be connected back to itself.

$$P \rightarrow \text{next} = P;$$

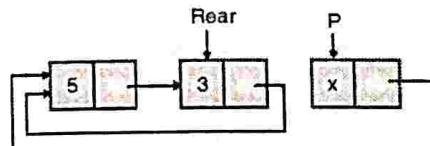


- (d) front should point to the only element
 $\text{front} = P$

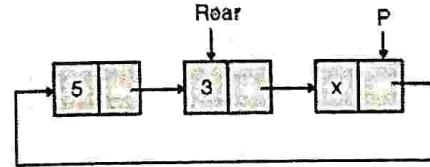


If the queue is not empty

- (e) $P \rightarrow \text{next} = \text{rear} \rightarrow \text{next}$

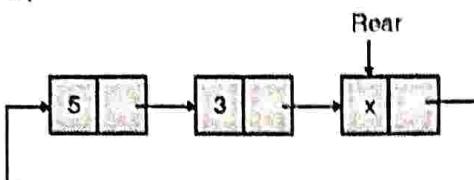


- (f) $\text{rear} \rightarrow \text{next} = P$





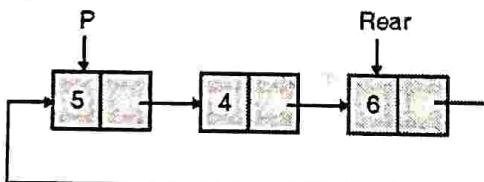
(g) rear = P



'C' function for insertion of an element in a queue represented using a circular linked list.

```
void enqueue(node **R, int x)
//queue is referenced by address of rear node.
{
    node *P;
    P = (node *) malloc(sizeof(node));
    P->data = x;
    if(*R == NULL)
    {
        P->next = P; *R = P;
    }
    else
    {
        P->next = (*R)->next;
        (*R)->next = P;
        *R = P;
    }
}
```

Steps for deletion of an element from a queue represented using a circular linked list.

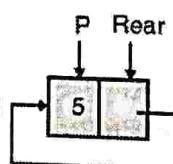


(a) P = rear → next;

P Point to the node to be deleted

(b) x = P → data;

if (last node is being deleted)



(a) release the memory of the node being deleted.

free(P);

(b) Make the queue empty
rear = NULL;

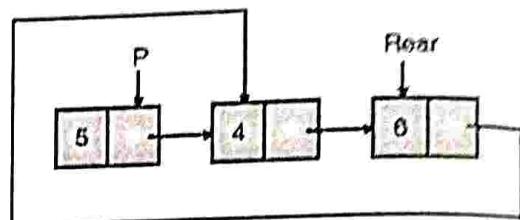
(c) Return the value stored in the front node

return(x);

}
else
{

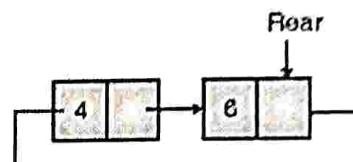
(c) Remove the front node from the queue.

rear → next = P → next



(d) release the memory of the node being deleted.

free(P)



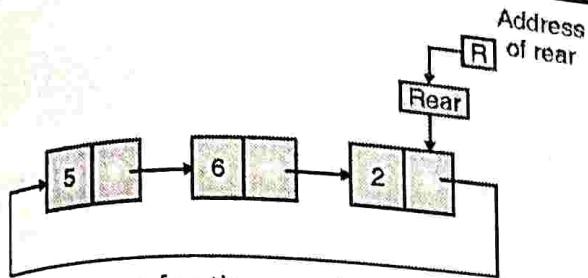
(e) return the value stored in the front node

return(x)

}

'C' function for deletion of the front node from a queue represented using a circular linked list.

```
int dequeue(node **R)
//pointer rear is passed by reference
{
    node *P;
    int x;
    P = (*R)->next;
    /* *R is same as rear as R contains the address of
       rear */
    x = P->data;
    if(P->next == P) /* deleting the last node */
    {
        *R = NULL;
        free(P);
        return(x);
    }
    (*R)->next = P->next;
    free(P);
    return(x);
}
```



In the above function, rear is passed by reference as after deletion, rear may change. Receiving variable in function int dequeue(node **R) is declared as node **R. R contains the address of rear and hence, *R can be used in place of 'rear'.

'C' function for printing elements of a queue represented using a circular linked list.

```
void Print(node * rear)
{
    node P;
    P = rear -> next; /* start printing from the front */
    do
    {
        printf("\n%d", P -> data);
        P = P -> next;
    } while(P != rear -> next);
}
```

In case of a circular linked list, the starting case and the termination case for the loop used for traversal of the linked list are identical.

1. We start printing from the front node.
2. We terminate printing on reaching the front node.

Such cases are best handled through do-while loops.

Program 3.9.1 : Program for showing various operations on a queue represented using circular linked list.

```
#include<stdio.h>
#include<conio.h>
typedef struct node
{
    int data;
    struct node *next;
}node;
void init(node **R);
void enqueue(node **R, int x);
int dequeue(node **R);
int empty(node *rear);
void print(node *rear);
void main()
```

```
{
    int x, option;
    int n = 0, i;
    node *rear;
    init(&rear);
    clrscr();
    do
    {
        printf("\n1. Insert\n2. Delete\n3. Print\n4. Quit");
        printf("\n your option: ");
        scanf("%d", &option);
        switch(option)
        {
            case 1 :
                printf("\n Element to be inserted");
                scanf("%d", &n);
                for(i = 0; i < n; i++)
                {
                    scanf("\n %d", &x);
                    enqueue(&rear, x);
                }
                break;
            case 2 :
                if(! empty(rear))
                {
                    x = dequeue(&rear);
                    printf("\n Element deleted = %d", x);
                }
                else
                    printf("\n Underflow..... Cannot deleted");
                break;
            case 3 :
                print(rear);
                break;
        }
    }while(option != 4);
    getch();
}

void init(node **R)
{
    *R = NULL;
}

void enqueue(node **R, int x)
{
    node *p;
    p = (node *)malloc(sizeof(node));
    p->data = x;
```



```

if(empty(*R))
{
    p->next = p;
    *R = p;
}
else
{
    p->next = (*R)->next;
    (*R)->next = p;
    (*R) = p;
}
int dequeue(node **R)
{
    int x;
    node *p;
    p = (*R)->next;
    p->data = x;
    if(p->next == p)
    {
        *R = NULL;
        free(p);
        return(x);
    }
    (*R)->next = p->next;
    free(p);
    return(x);
}
void print(node *rear)
{
    node *p;
    if(!empty(rear))
    {
        p = rear->next;
    }
    p = p->next;
    do
    {
        printf("\n %d", p->data);
        p = p->next;
    }while(p != rear->next);
}
int empty(node *P)
{
    if(P->next == -1)
        return(1);
}

```

```

    return(0);
}

```

Output

1. Insert

2. Delete

3. Print

4. Quit

your option: 1

Element to be inserted 4

12 23 34 45

1. Insert

2. Delete

3. Print

4. Quit

your option : 3

12 23 34 45

1. Insert

2. Delete

3. Print

4. Quit

your option : 2

Element deleted = 4

1. Insert

2. Delete

3. Print

4. Quit

your option : 3

23 34 45

1. Insert

2. Delete

3. Print

4. Quit

your option: 4

3.9.1 Implementation of a Priority Queue using a Linked List

Assumptions

- Elements will be inserted as per its priority.
- All deletions from the front.
- Elements of the priority queue are assumed to contain their priority values and hence, the type of elements is assumed to be integer.

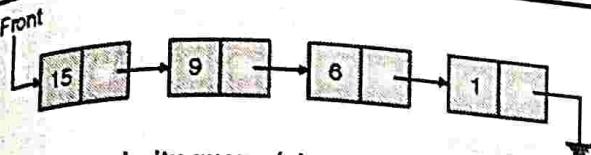


Fig. 3.9.1 : A priority queue (element at the front end has the highest priority with elements arranged in descending order of priority)

Data structure to be used for representation is similar to one used for representation of linked list.

Declaration in 'C' :

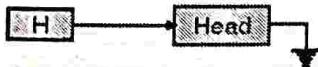
```
typedef struct node
{
    int data;          /* priority */
    struct node * next;
} node;
```

Operations on a priority queue

1. **initialize()** : Make the queue empty.
2. **empty()** : Determine if the queue is empty.
3. **enqueue()** : Insert an element as per its priority.
4. **dequeue()** : Delete the front element (front element will have the highest priority).

C-implementation of various operations on a priority queue.

```
void initialize(node **H)
{
    *H = NULL;
}
```



A singly linked list is referenced through the address of its front node. Address of the front node is stored in a pointer variable head (declared as node *head). If the value of head is to be changed in any called function, head should be passed by reference. Receiving variable in called function must be declared as 'node **' type. If the receiving variable is declared as node '**H' then '**H' can be used in place of 'head'.

```
int empty(node *head)
{
    if(head == NULL)
        return(1);
}
```

```

return(0);
}

void enqueue(node **H, int x)
{
    node *P, *q;
    P = (node *) malloc(sizeof(node));
    P-> data = x;
    P-> next = NULL;
    /* if the queue is empty */
    if(*H == NULL)
        *H = P
    else
        if(x > *H-> data)
            /* highest priority element must be inserted at
               front */
        {
            P-> next = *H;
            *H = P;
        }
        else
        {
            q = *H; /* locate the point of insertion */
            while(q-> next != NULL && x < q->
next-> data)
                q = q-> next;
            P-> next = q-> next;
            q-> next = p; //insert the node
        }
    }

int dequeue(node **H)
{
    int x; node *P;
    P = *H;
    x = P-> data;
    (*H) = P-> next;
    free(P);
    return(x);
}
```

CHAPTER 4

Module 4

Trees

Syllabus

Introduction, Tree Terminologies, Binary Tree, Binary Tree Representation, Types of Binary Tree, Binary Tree Traversals, Binary Search Tree, Operations on Binary Search Tree, Applications of Binary Tree-Expression Tree, Huffman Encoding, Search Trees-AVL, rotations in AVL Tree, operations on AVL Tree, Introduction of B Tree, B+ Tree.

4.1 Basic Terminology

4.1.1 Introduction

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, \dots, 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. 4.1.1 is a typical tree using the recursive definition.

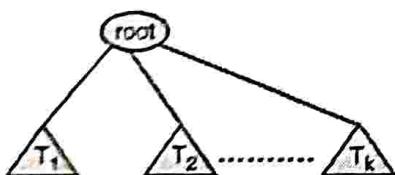


Fig. 4.1.1 : Generic tree

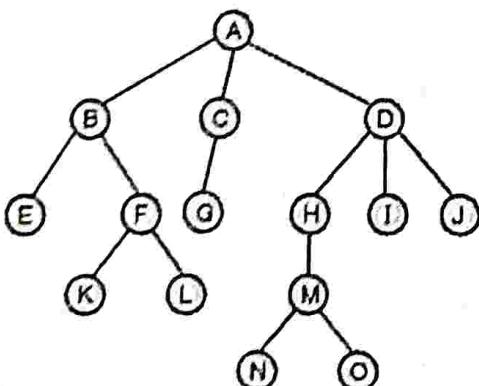


Fig. 4.1.2 : A tree

4.1.2 Basic Terms

- 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. 4.1.2, the root is A.
- Parent** : It is an immediate predecessor of a node. In the Fig. 4.1.2, A is the parent of B, C, D.
- Child** : All immediate successors of a node are its children. In the Fig. 4.1.2, 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. 4.1.2 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. 4.1.2, 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.

4.2 Binary Tree

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. 4.2.1.

Node A has two children B and C. Similarly, nodes B and C, each have one child name G and D respectively.

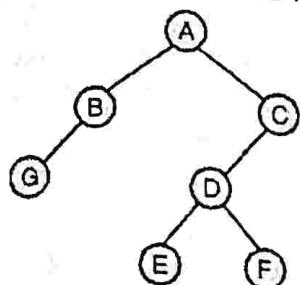


Fig. 4.2.1 : A binary tree

4.3 Representation of a Binary Tree using an Array

In order to represent a tree in a single one-dimensional array, the nodes are numbered sequentially level by level left to right. Even empty nodes are numbered.

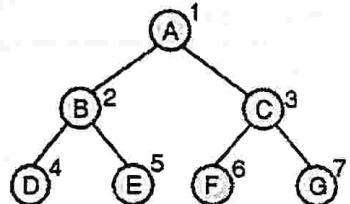


Fig. 4.3.1 : Numbering of nodes

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.

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

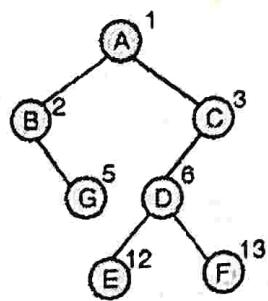


Fig. 4.3.2 : Numbering of nodes with some non-existing children

| | | | | | | | | | | | | | |
|----|---|---|---|----|---|---|----|----|----|----|----|----|----|
| 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. 4.3.3 : Array representation of the tree shown in Fig. 4.3.2

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 4.3.1 : Explain array representation of binary trees using the Fig. Ex. 4.3.1.

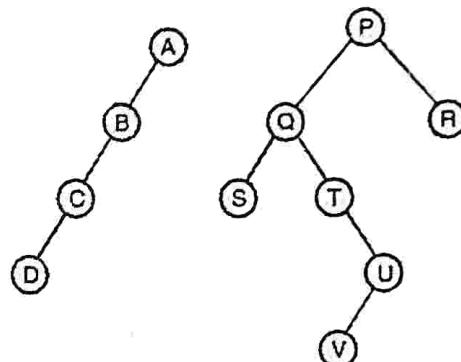


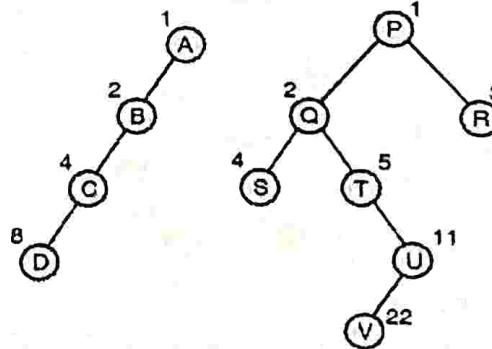
Fig. Ex. 4.3.1

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



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 |

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 4.3.2 : Define a binary tree. Show the sequential representation of the binary tree given.

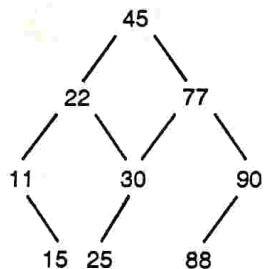
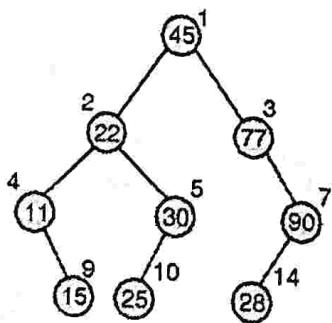


Fig. Ex. 4.3.2

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****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 |

4.4 Linked Representation of a Binary Tree

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 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;
  
```

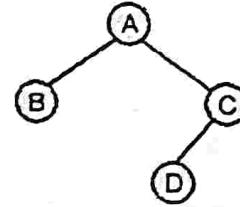


Fig. 4.4.1 : A sample binary tree

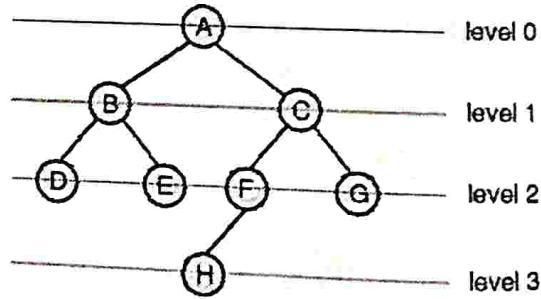


Fig. 4.4.2 : Levels

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. 4.4.2.

Height of a tree : Height of a tree is height of the root node. Height of the tree shown in Fig. 4.4.2 is 3 as the height of the root node A is 3.

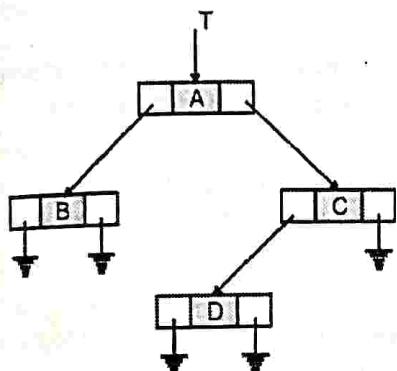


Fig. 4.4.3 : Linked list representation of the binary tree of Fig. 4.4.1

The following C-program can be used to construct the binary tree of Fig. 4.4.1.

4.4.1 Program for Creation of a Sample Binary Tree

Program 4.4.1 : A program for construction of binary tree of Fig. 4.4.1

```

#include<stdio.h>
#include<conio.h>
typedef struct node
{
    char data;
    struct node *left, *right;
}node;
void main()
{
    node *T; // Root of the tree
}
    
```

```

node *P, *q;
// Address of the first node in T
T = (node*)malloc(sizeof(node));
T->left = NULL;
T->right = NULL;
T->data = 'A';
/* Get a new node with its address in P and make it left node of T */
P = (node*)malloc(sizeof(node));
P->left = NULL;
P->right = NULL;
P->data = 'B'; // Make it left node of T
T->left = P;
/* Get a new node with its address in P and make it 'right' node of T */
P = (node*)malloc(sizeof(node));
P->left = NULL;
P->right = NULL;
P->data = 'C';
T->right = P; // Make it right node of T
/* Insert a new node as left child of node pointed by P*/
q = (node*)malloc(sizeof(node));
q->left = NULL;
q->right = NULL;
q->data = 'D';
P->left = q;
}
    
```

4.4.2 'C' Function for Creation of a Binary Tree

A binary tree can be created recursively. The function works as follow :

1. Read a new data in x.
2. Acquire memory for a new node with its address in pointer p.
3. Store the data x in the node P.
4. Recursively create the left subtree of p and make it the left child of p.
5. Recursively create the right subtree of p and make it the right child of p.

```
#include <stdio.h>
#include <conio.h>
typedef struct node
{
    int data;
    struct node *left, *right;
}node;
node *create( )
{
    node *p;
    int x;
    printf("\n Enter a data(-1 for no data) : ");
    scanf("%d", &x);
    if(x == -1)
        return NULL;
    p = (node*)malloc(sizeof(node));
    p->data = x;
    printf("\n Enter left child of %d : ", x);
    p->left = create();
    printf("\n Enter right child of %d : ", x);
    p->right = create();
    return p;
}
```

4.5 A General Tree

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.

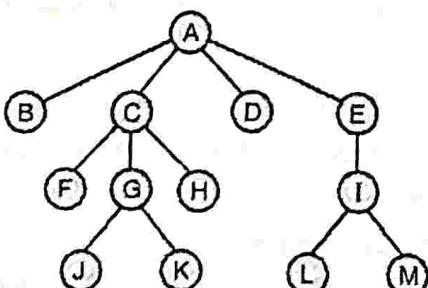


Fig. 4.5.1 : A tree

4.5.1 Node Declaration for a Tree

```
typedef struct tnode
{
    int data; // Assuming tree stores integer data
    struct tnode *firstchild;
    struct tnode *nextsibling;
}tnode;
```

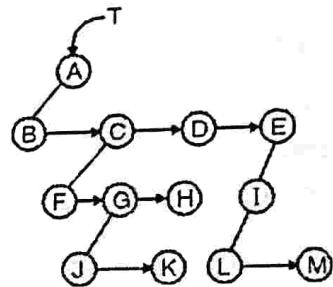


Fig. 4.5.2 : Leftmost child right sibling representation of the tree shown in Fig. 4.5.1

Each node in Fig. 4.5.2 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. 4.5.2 is obtained from the general tree of Fig. 4.5.1 using 'leftmost child right sibling' relation. The tree of Fig. 4.5.2, 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. 4.5.2(a) is obtained by rotating the tree of Fig. 4.5.2 by 45°.

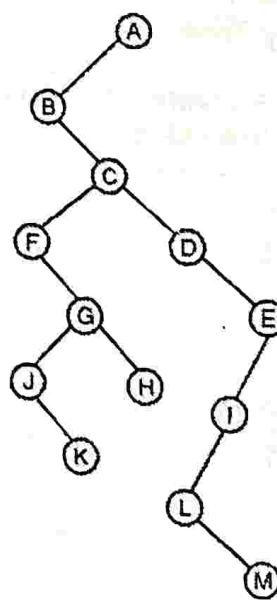


Fig. 4.5.2(a) : Tree of Fig. 4.5.2 is redrawn after it is rotated by 45°

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. 4.5.2(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. 4.5.2(b).

Fig. 4.5.2(b)

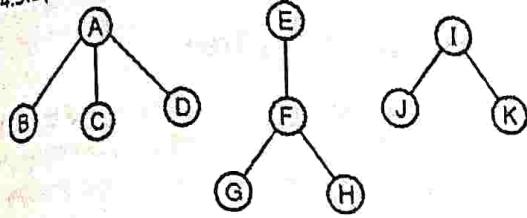


Fig. 4.5.2(b) : A forest containing three trees

Each tree in forest is converted to a binary tree using 'leftmost child right sibling relation'. It is shown in Fig. 4.5.2(c).

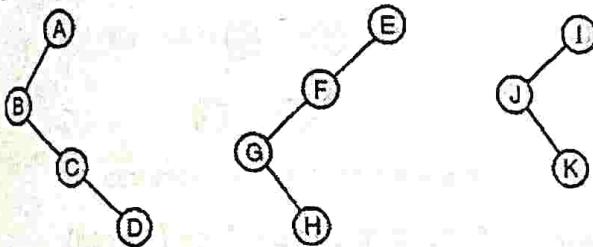


Fig. 4.5.2(c)

Each tree of forest in Fig. 4.5.2(b) is represented by its corresponding binary tree.

Three binary trees of Fig. 4.5.2(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. 4.5.2(d).

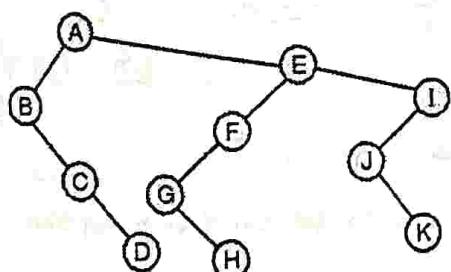


Fig. 4.5.2(d) : Binary tree representation of forest in Fig. 4.5.2(b)

Example 4.5.1 : What is the necessity of converting a tree into binary tree ? Given the following tree :

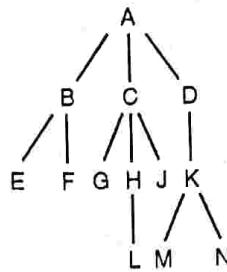


Fig. Ex. 4.5.1

Convert 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

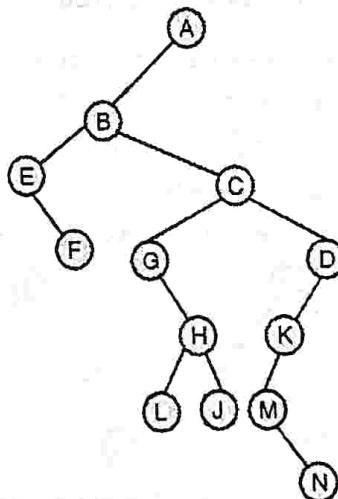


Fig. Ex. 4.5.1(a)

4.6 Types of Binary Tree

4.6.1 Full Binary Tree

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 filled. 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. 4.6.1(a).

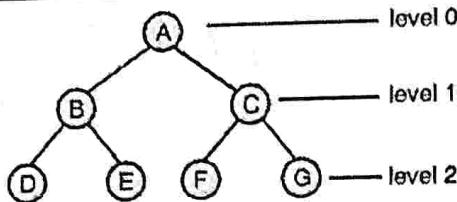


Fig. 4.6.1(a) : Full binary tree of depth 3

Total number of nodes in a full binary tree of height

$$h = 2^0 + 2^1 + 2^2 \dots + 2^h = 2^{h+1} - 1$$

The height of the tree shown in Fig. 4.6.1(a) is 2 and number of nodes is given by

$$2^{h+1} - 1 = 8 - 1 = 7.$$

Hence no. 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.

4.6.2 Complete Binary Tree

A complete binary tree is defined as a binary tree where

- All leaf nodes are on level n or $n - 1$.
- Levels are filled from left to right.

Examples of a complete binary tree are shown in Fig. 4.6.1(b). Heap is an example of complete binary tree.

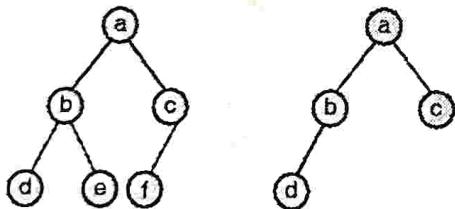


Fig. 4.6.1(b) : Complete binary trees

4.6.3 Skewed Binary Tree

Fig. 4.6.1(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. 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.

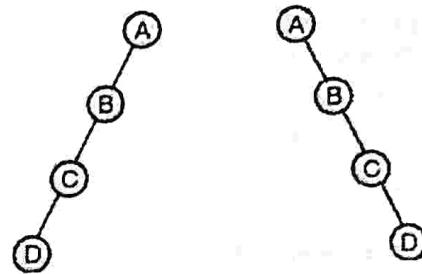


Fig. 4.6.1(c) : Skewed Binary trees

4.6.4 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. 4.6.1(d) shows a strictly binary tree.

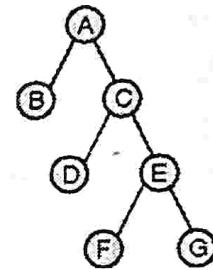


Fig. 4.6.1(d) : Strictly binary tree

4.6.5 Extended Binary Tree (2-Tree)

In an extended tree, each empty subtree is replaced by a failure node. A failure node is represented by \square . 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.

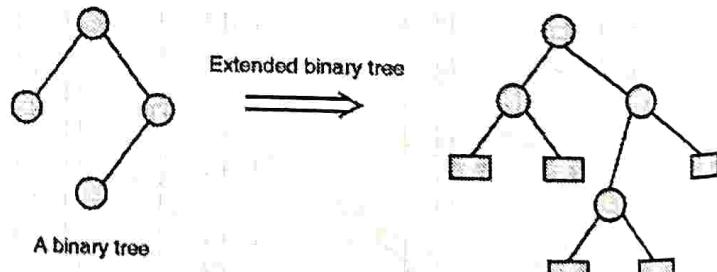


Fig. 4.6.1(e) : Extended binary tree

4.7 Binary Tree Traversal

MU - May 15, May 18

University Questions

Q. Define traversal of binary tree. Explain different types of traversals of Binary tree with example.

(May 15, 5 Marks)

Q. Explain different types of tree traversals techniques with example. Also write recursive function for each traversal technique. (May 18, 10 Marks)

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.

4.7.1 Preorder Traversal (Recursive)

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.

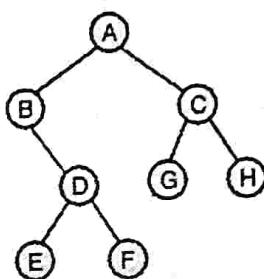


Fig. 4.7.1 : A sample binary tree

Stepwise preorder traversal of tree is shown in Fig. 4.7.2 is given below :

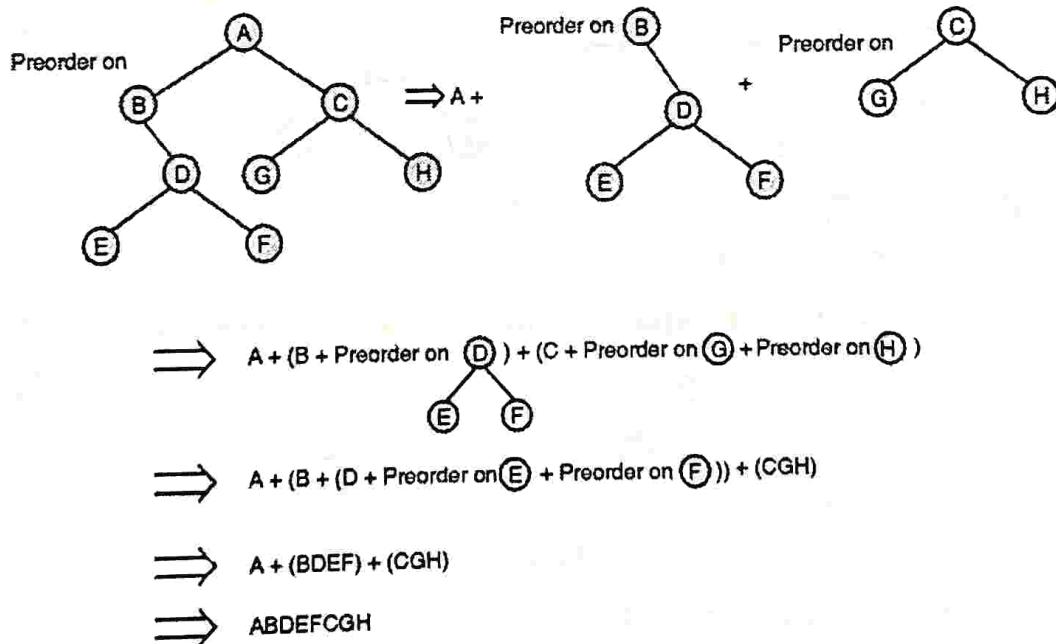


Fig. 4.7.2

4.7.1(A) 'C' Function for Preorder Traversal

MU - Dec. 19

University Question

Q. Write a recursive function to perform pre-order traversal of a binary tree.

(Dec. 19, 8 Marks)

```

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.

4.7.2 Inorder Traversal (Recursive)

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. 4.7.3.

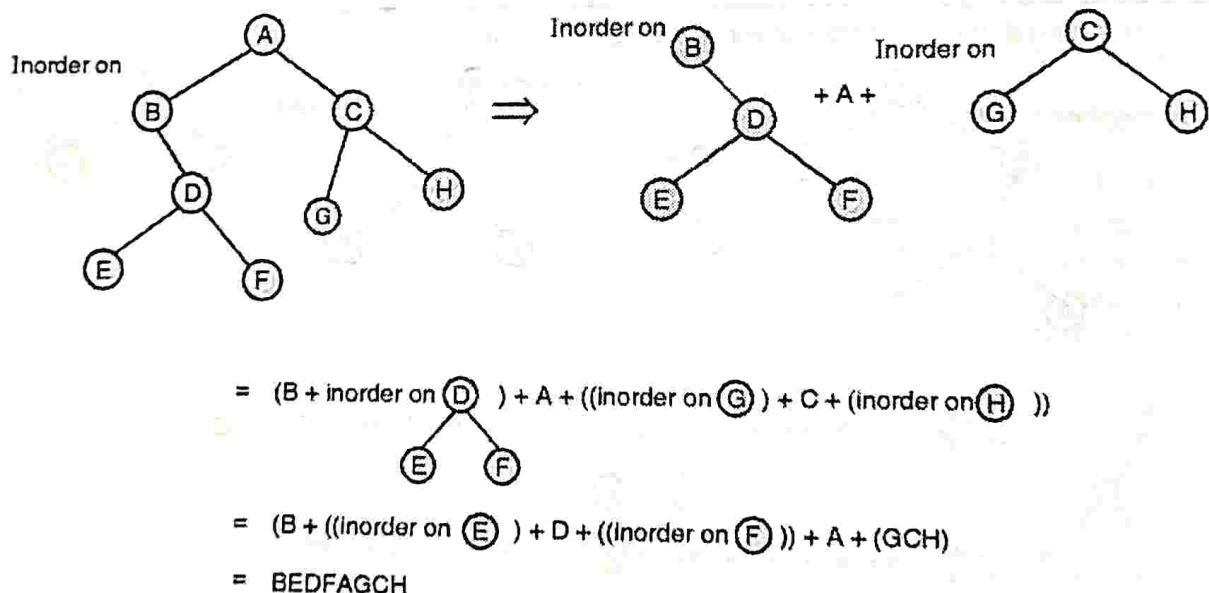


Fig. 4.7.3

4.7.2(A) '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 */
    }
}

```

```

    inorder(T → right);
}
}
/* Inorder traversal on right subtree */

```

4.7.3 Postorder Traversal (Recursive)

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. 4.7.4 is given below.

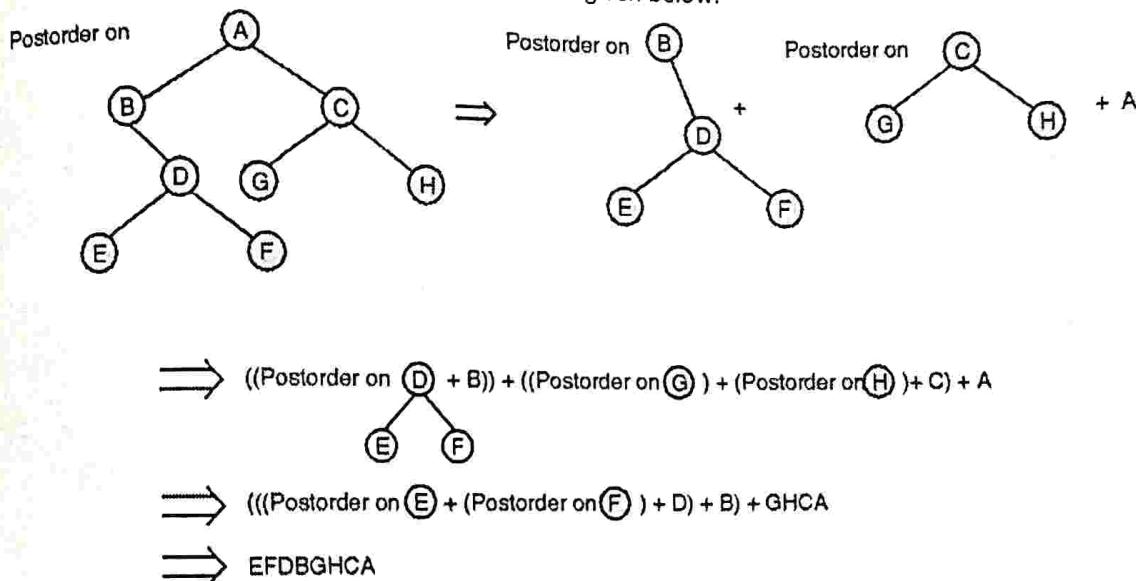


Fig. 4.7.4

4.7.3(A) 'C' Function for Postorder Traversal

```

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 */
    }
}

```

4.7.4 Non-Recursive Preorder Traversal

Algorithm :

Recursion from the function for preorder traversal in Section 4.7.1 can be removed through the use of a stack. Algorithm for non-recursive traversal is given below :

Step1: Start traversing from the root (say T), traverse left and continue traversing left. All nodes traversed are pushed into a stack (say S)

```

while(T! = NULL)
{
    visit(T);
    push(&S, T);
    T = T → left;
}

```



[nodes are saved in the stack for the purpose of backtracking i.e. traversal of right subtrees of all nodes visited so far]

Step 2

```

if the stack S is empty
then
    traversal is finished
else
{
    [ visit the right subtree of the node popped
    from the stack ]
    T = POP(&S);
    T = T → right ;
    Start traversing from T, traverse left and
    continue traversing left.
    All nodes traversed are pushed into the stack
    S.
    while(T! = NULL)
    {
        visit(T)
        push(&S, T);
        T = T → left;
    }
}

```

Step 3 : goto step 2

4.7.4(A) 'C' Function for Non-Recursive Preorder of Tree Along with the ADT Stack

```

typedef struct node //structure of tree node
{
    int data;
    struct node *left;
    struct node *right;
}node;
typedef struct stack
{
    node *data[30]; //maximum of 30 nodes
    int top;
}stack;
void preorder_non_recursive(node *T)

```

```

{
    stack S; //initialise the stack
    S.top = -1; //traverse left
    while(T! = NULL)
    {
        printf("\n %d", T->data);
        push(&S, T);
        T = T->left;
    }
    while(!empty(&S))
    {
        // pop a node and traverse its right subtree
        T = pop(&S);
        T = T->right;
        while(T! = NULL)
        {
            printf("\n %d", T->data);
            push(&S, T);
            T = T->left;
        }
    }
}

void push(stack *S, node *T)
{
    S->top = S->top + 1;
    (S->data)[S->top] = T
}

node *pop(stack *S)
{
    node *T;
    T = (S->data)[S->top];
    S->top = S->top - 1;
    return(T);
}

int empty(stack *S)
{
    if(S->top == -1)
        return(1);
    return(0);
}

```

4.7.5 Non-Recursive Inorder Traversal

Algorithm :

Algorithm for non-recursive inorder traversal of binary tree too works similar to non-recursive preorder traversal. In non-recursive preorder traversal, a node is visited before it is pushed into the stack. In non-recursive inorder traversal, a node is visited immediately after it is popped from the stack.

Step 1: Start traversing from the root (say T), traverse left and continue traversing left. All nodes traversed are pushed into the stack (say S).

```
while(T != NULL)
{ push(&S, T);
  T = T -> left;
}
```

[nodes are saved in the stack for the purpose of backtracking i.e. traversal of right subtrees of all nodes, visited so far.]

Step 2

```
if the stack S is empty
then traversal is finished
else
{ Visit the right subtree of the node popped from the
stack
  T = pop(&S); visit(T);
  T = T -> right;
  Start traversing from T, traverse left and continue
  traversing left. All nodes traversed are pushed into
  the stack S.
  while(T != NULL)
  { push(&S, T);
    T = T -> left;
  }
}
```

Step 3: goto step 2.

4.7.5(A) 'C' Function for Non-Recursive Inorder Traversal of a Binary Tree

```
void inorder_non_recursive(node *T)
{ stack s;
```

```
s.top = -1; //Initialising a stack
while(T != NULL)
{
  push(&s, T);
  T = T -> left;
}
while(!empty(&s))
{
  //Pop a node and traverse its right subtree
  T = pop(&s);
  printf("\n %d", T->data);
  T = T -> right;
  while(T != NULL)
  {
    push(&s, T);
    T = T -> left;
  }
}
```

4.7.6 Non-Recursive Postorder Traversal

Non-recursive postorder traversal works in slightly different way. In postorder traversal, an element is visited after the right subtree is traversed. Thus, the address of the node should be preserved in stack until it has been printed. Non-recursive postorder traversal algorithm encounters a node three times :

1. While going to the left.
2. During backtracking to traverse the right subtree.
3. While returning from the right.

An additional field 'flag' in stack is used to differentiate between the two situations.

- While going to left, address of the treenode along with flag = 0 is pushed onto the stack.
- During backtracking to traverse the right subtree, an element is popped from the stack and if flag field is found to be 0, it is pushed back with flag set to 1.
- When we return from the right subtree and pop an element from the stack, the flag field will be 1. This is the time when we visit the node.

Example 4.7.1 : Simulate working of non-recursive postorder traversal on the following tree

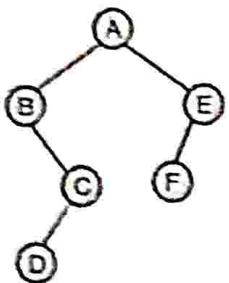


Fig. Ex. 4.7.1

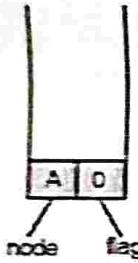
Solution :

Initially stack will be empty. We start traversing from the root.

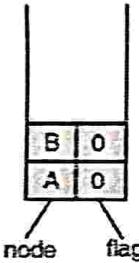
Step 1: Start traversing from the root, traverse and continue traversing left.

Every node encountered is pushed onto the stack with flag set to 0.

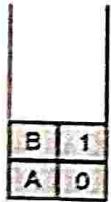
goto node A



goto node B



Step 2: Pop an element from stack. Node B will be popped. Since, flag is equal to 0, it is pushed back with flag equal to 1. Now, we traverse the right subtree of B.

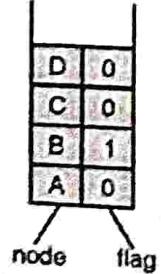


Step 3: Start traversing from the root of the subtree rooted at C [right subtree of B]. Continue traversing left. Every node encountered is pushed onto the stack with flag set to 0.

goto node C



goto node D

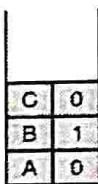


Step 4: Pop an element from stack. Node D will be popped. Since, flag is equal to 0, it is pushed back with flag equal to 1. Now, we traverse the right subtree of D. Right subtree of node D is a NULL tree.



Step 5: Pop an element from stack. Node D will be popped. Since, flag is equal to 1, it is visited.

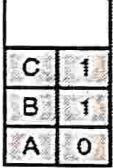
First node visited is D



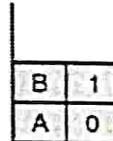
Status of stack after subsequent steps is given below:

Step 6: Goto the right of node C

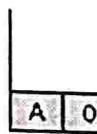
node C



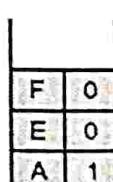
Step 7: Visit C



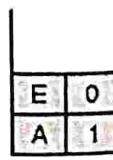
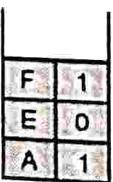
Step 8: Visit B



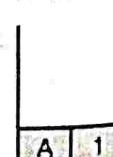
Step 9: Goto the right of node A



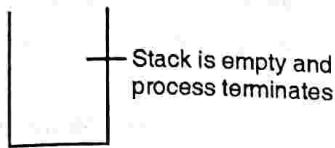
Step 10: Goto the right of F **Step 11:** Visit F



Step 12: Goto the right of E **Step 13:** Visit E



Step 14: Visit A



Postorder traversal sequence : D C B F E A.

Algorithm

Step 1: Start traversing from the root (say T), traverse left and continue traversing left. Every node encountered is pushed onto the stack with flag equal to 0.

```
while(T != NULL)
{
    push(&S, &T, 0);
    T = T->left;
}
```

Step 2 :

```
if the stack S is empty
then
    traversal is finished
else
{
    (T, flag) = pop(&S)      // pop node and flag
    if flag is equal to 1
    then
        visit the node T
    else
        { // visit its right subtree
            push(&S, T, 1);
            T = T->right;
            while(T != NULL)
            {
                push(&S, T, 0)
                T = T->left;
            }
        }
}
```

Step 3 : Goto step 2.

4.7.6(A) 'C' Function for Non-Recursive Postorder Traversal

```
typedef struct treenode //structure of tree node
{
    int data;
    struct node *left;
    struct node *right;
}treenode;

typedef struct stacknode
{
    treenode *data;
    int flag;
}stacknode;

typedef struct stack
{
    stacknode data[30];
    int top;
} stack;

void postorder_non_recursive(treenode *T)
{
    stack S;
    stacknode stnode;
    treenode *P;
    S.top = -1;           //initialise the stack;
    while(T != NULL) //traverse left
    {
        /* explore until leftmost child each tree node
        is saved in the stack with the flag=0. flag 0
        indicates that the right subtree has not been
        traversed */
        stnode.data = T;
        stnode.flag = 0;
        push(&S, stnode);
        T = T->left;
    }
    while(!empty(&S))
    {
        stnode = pop(&S)
        /* if the flag = 0 then
        visit right subtree */
        if(stnode.flag == 0)
        {
            stnode.flag = 1;
            push(&S, stnode);
            T = stnode.data;
        }
    }
}
```



```

T = T->right;
while(T != NULL)
{
    stnode.data = T;
    stnode.flag = 0;
    push(&S, T);
    T = T->left;
}
else
{
/* Right subtree has been traversed and the
current node must be visited now */
T = stnode.data;
printf("\n %d", T->data);
}
}

/* Functions for ADT stack used in non recursive post
order traversal */
void push(stack *S, stacknode t)
{
    S->top = S->top + 1;
    (S->data)[S->top] = t;
}

stacknode pop(stack *S)
{
    stacknode t;
    t = (S->data)[S->top];
    S->top = S->top - 1;
    return(t);
}

int empty(stack *S)
{
    if(S->top == -1)
        return(1);
    return(0);
}

```

Example 4.7.2 : Write non-recursive postorder and inorder traversals and compare their complexities.

Solution :

Algorithm for non-recursive postorder traversal is discussed in Section 4.7.6.

Algorithm for non-recursive inorder traversal is discussed in Section 4.7.5.

Time complexity of inorder = $O(n)$

Time complexity of postorder = $O(n)$

4.7.7 Tree Traversal Examples

Example 4.7.3 : Traverse the following binary tree into preorder and inorder and postorder with reason.

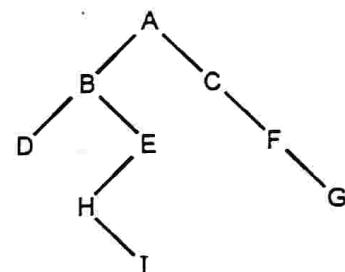


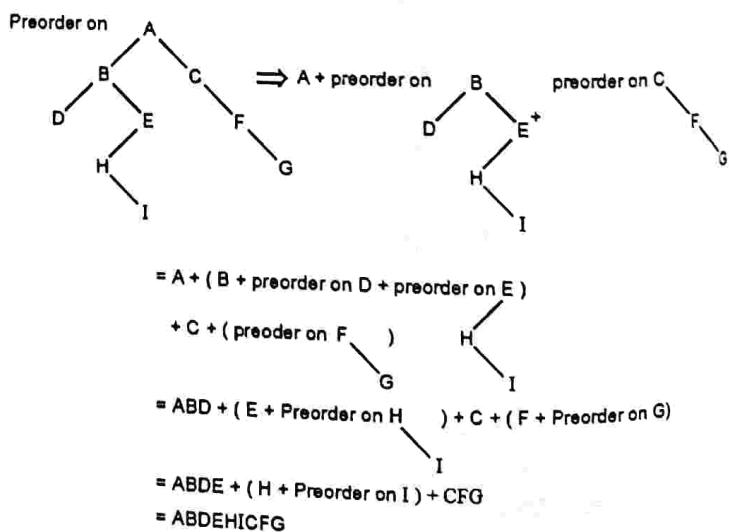
Fig. Ex. 4.7.3

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.

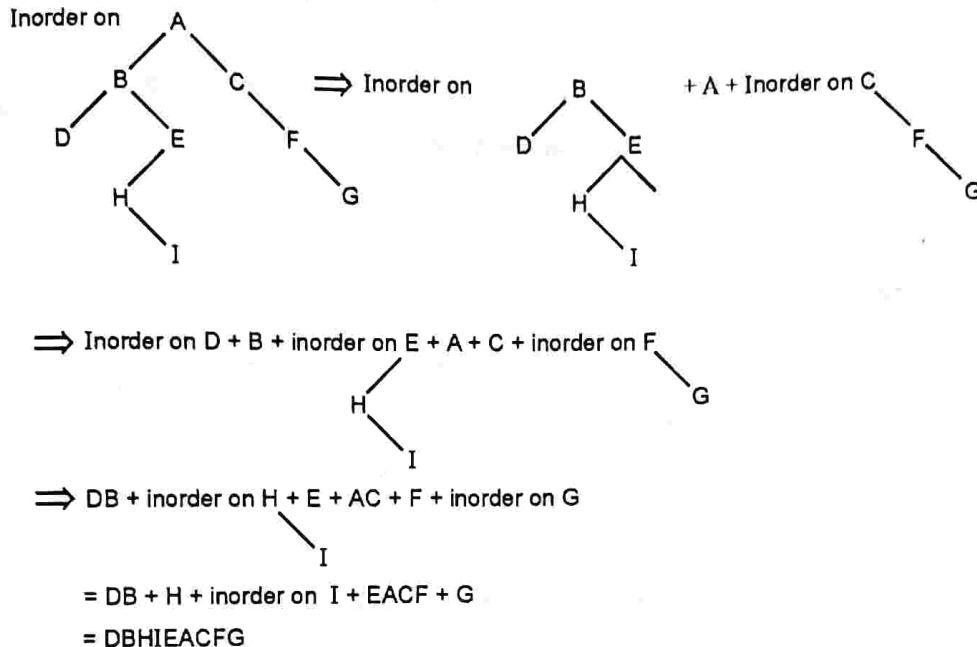
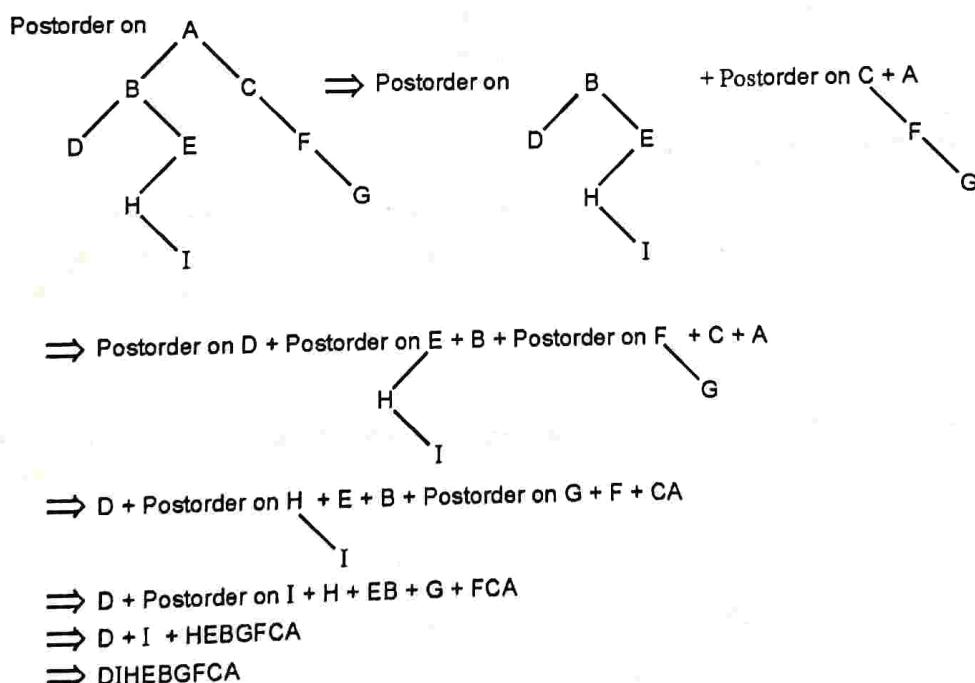
Stepwise preorder traversal on the above tree is given as follows :



Inorder traversal :

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**

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 4.7.4 : Traverse the following binary tree into preorder and inorder with reason.

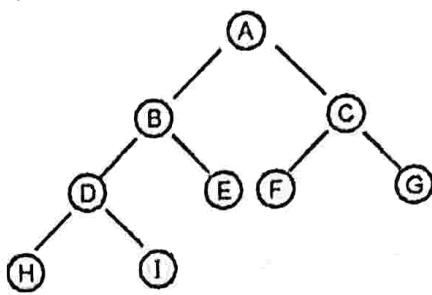
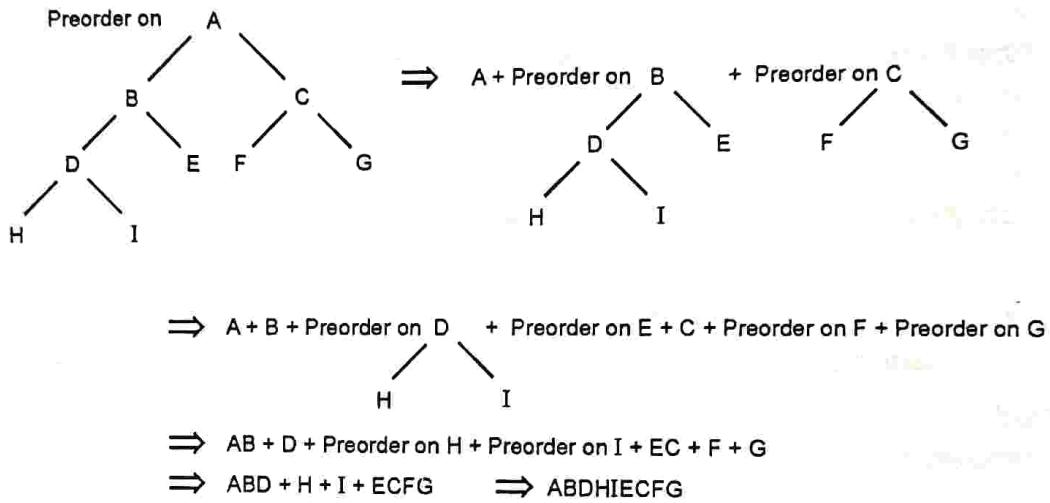


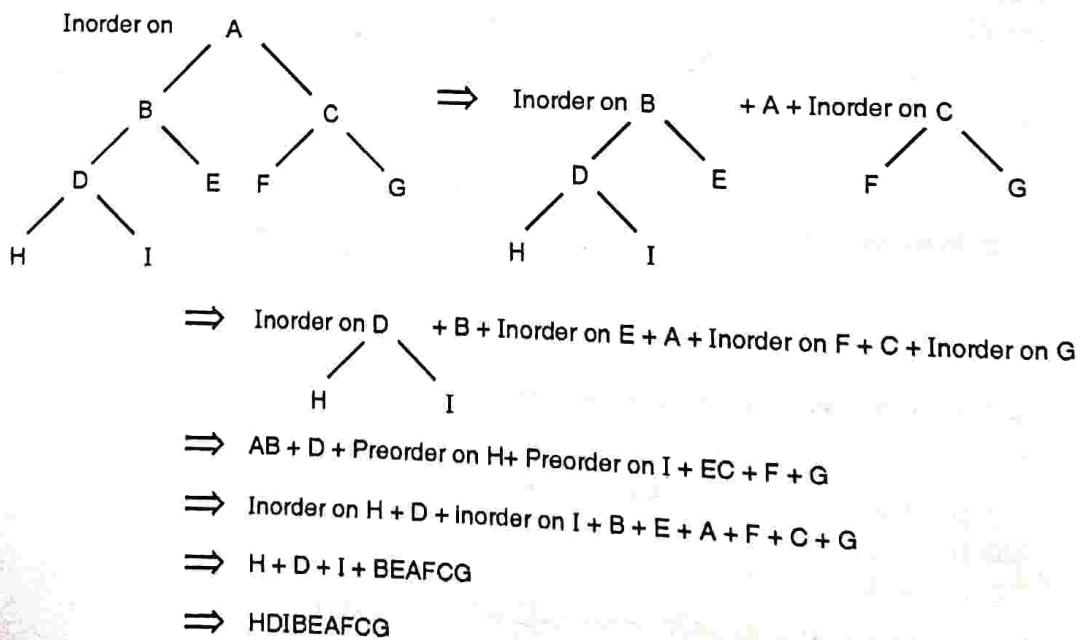
Fig. Ex. 4.7.4

Solution :

Preorder traversal



Inorder traversal



4.8 Basic Tree Operations

Using the definition of a binary tree and the recursive version of the traversal, we can easily perform most of the operations on binary tree. We can slightly modify the tree traversal functions to perform most of the operations.

4.8.1 'C' Function for Counting of Nodes in a Tree

Recursive

```
int count(node *T)
//Address of root node is passed in T
{
    int i;
    if(T == NULL)
        return(0); // A null tree with 0 nodes
    i = 1+count(T->left)+count(T->right);
    /* count(T->left) given no of nodes in the left
    subtree. count(T->right) given no. of nodes in the
    right subtree. Hence the total no. of nodes in a tree
    = 1(current node) + no. of nodes in the left
    subtree + no of nodes in the right subtree */
    return(i);
}
```

Non-recursive

Any of the tree traversal algorithms like :

- (i) Non-recursive preorder traversal,
- (ii) Non-recursive inorder traversal,
- (iii) Non-recursive postorder traversal,
- (iv) Non-recursive level-wise traversal

Can be used to visit every node of a tree. Each algorithm ensures that each node will be visited and visited only once.

Number of nodes in a tree is exactly equal to number of nodes visited.

Algorithm given below is based on preorder traversal of tree.

```
// Non-recursive function for counting of nodes
int count_non_recursive(node *T)
{
    stack S; int count = 0;
```

```
s.top = -1; // Initialize the stack
while(T != NULL)
{
    count = count + 1; //Visit
    push(&S, T);
    T = T->left;
}
while(!empty(&S))
{
    T = pop(&S);
    // Pop a node to traverse its right subtree
    T = T->right;
    while(T != NULL)
    {
        count = count + 1; //visit
        push(&S, T);
        T = T->left;
    }
}
return(count);
}
```

4.8.2 'C' Function for Counting of Leaf Nodes in a Tree (Recursive)

```
int count0(node *T)
{
    int i;
    if(T == NULL)
        return(0);
    if(T->left == NULL && T->right == NULL)
        return(1); // A leaf node
    i = count0(T->left)+count0(T->right);
    /* no. of leaf nodes in the left subtree and right sub
    tree */
    return(i);
}
```



4.8.3 'C' Function for Counting of Nodes of Degree 1 (Recursive)

```
int count1(node **T)
{
    int i;
    if(T == NULL)
        return(0);
    if(T->left == NULL && T->right == NULL)
        return(0);
    if(T->left == NULL || T->right == NULL)
    {
        // A node of degree 1
        i = 1 + count1(T->left)+count1(T->right);
        return(i);
    }
    i = count1(T->left)+count1(T->right);
    return(i); // A node of degree 2
}
```

4.8.4 'C' Function for Counting of Nodes of Degree 2 (Recursive)

```
int count2(node **T)
{
    int i;
    if(T == NULL)
        return(0); //a null tree
    if(T->left == NULL && T->right == NULL)
        return(0);
    if(T->left == NULL || T->right == NULL)
    {
        i = count2(T->left)+count2(T->right);
        return(i);
    }
    i = 1+count2(T->left)+count2(T->right);
    return(i);
}
```

4.8.5 'C' Function to Create an Exact Copy of a Tree (Recursive)

```
node *copy(node *T)
{
    node *P;
    P = NULL;
    if(T1 == NULL)
    {
        //Create a copy of current node in P
        P = (node*)malloc(sizeof(node));
        P->data = T->data;
        P->left = copy(T->left);
        //copy left sub tree
        P->right = copy(T->right);
        //Copy right sub tree
    }
    return(P);
}
```

4.8.6 'C' Function for Checking Equivalence of Two Binary Trees

```
int equivalence(node *T1, node *T2)
{
    /* Function returns 1 if the tree pointed by T1 and T2 are equivalent */
    if(T1 == NULL && T2 == NULL)
        return(1); //Two null trees are equivalent
    if(T1 != NULL && T2 != NULL)
    {
        if(T1->data == T2->data &&
            equivalence(T1->left, T2->left) &&
            equivalence(T1->right, T2->right))
            return(1);
    }
    return(0);
}
```

4.8.7 'C' Function for Finding Height of a Tree (Recursive)

Height of tree starting with root node
 $A = \text{Maximum}(\text{height of the left subtree of } A, \text{Height of the right subtree of } A) + 1 = \text{Maximum}(1, 2) + 1 = 3$

The recursive approach for finding the height of a binary tree is explained in the Fig. 4.8.1.

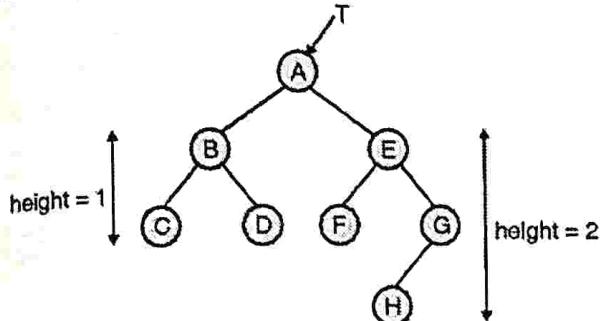


Fig. 4.8.1

```

int height(node *T)
{
    int hl, hr;
    /* hl - height of left subtree
       h2 - height of right subtree */
    if(T == NULL)
        return(0);
    if(T->left == NULL && T->right == NULL)
        return(0);
    //height of leaf node is 0
    hl = height(T->left);
    hr = height(T->right);
    if(hl>hr)
        return(hl+1);
    return(hr+1);
}
  
```

4.8.8 'C' Function for Swapping of Left and Right Children of Every Node (Mirror)

```

node * swapper(node * T)
{
    node * P;
    if(T!=NULL)
    {
        P = T->left;
        T->left = T->right;
        T->right = P;
        swapper(T->left);
        swapper(T->right);
    }
}
  
```

```

    T->left = swapper(T->right);
    T->right = swapper(P);
}
return(T);
}
  
```

4.8.9 Finding Width of a Tree

Width of a tree is defined by maximum number of nodes at any level. In the tree shown above, maximum number of nodes are at level 3. Hence the width of the tree is 3.

We can find the width of a tree through level-wise traversal.

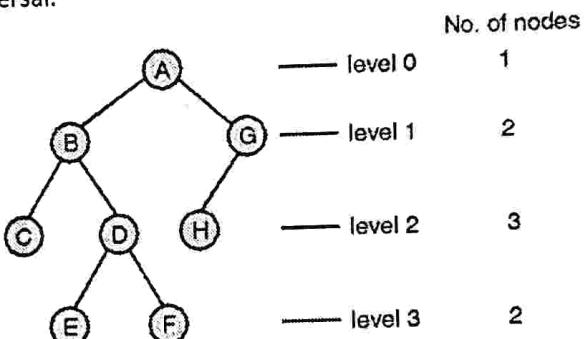


Fig. 4.8.2

'C' function for finding width of a tree.

```

typedef struct node           //Structure of tree node
{
    char data;
    struct node *left, *right;
} node;

typedef struct Q
// Queue used for level wise traversal
{
    node * data [50];
    int R, F;
} Q;

int width(node * T)
{
    Q q;
    node *p1, *p2;
    int maxwidth = 0, newwidth;
    if(T == NULL)
        return(0);
    q.R = -1;                                // Initialize the queue
    q.F = -1;
    p1 = T;
    while(q.R < q.F)
    {
        p2 = p1->left;
        if(p2 != NULL)
            q.data[q.F] = p2;
        p2 = p1->right;
        if(p2 != NULL)
            q.data[q.F] = p2;
        p1 = q.data[q.R];
        q.R++;
        if(q.R == q.F)
            newwidth = q.F - q.R + 1;
        if(newwidth > maxwidth)
            maxwidth = newwidth;
    }
    return(maxwidth);
}
  
```



```

insert(&q, T);
maxwidth = 1;
while(!empty(&q))
{
    p2 = q.R;
    /*p2 points to the last element of the current
    level*/
    newwidth = 0;
    do
    {
        p1 = delete(&q);
        if(p1->left != NULL)
        {
            insert(&q, p1->left);
            newwidth++;
        }
        if(p1->right != NULL)
        {
            insert(&q, p1->right);
            newwidth++;
        }
    }
    while(p1 != p2);
    if(newwidth > maxwidth)
        maxwidth = newwidth;
}
return(maxwidth);
}

```

4.8.10 Function to List the DATA Fields of the Node of a Binary Tree T by Level. Within Levels Nodes are Listed Left to Right

Levels by level traversal of a binary tree is also known as breadth first traversal. Level by level traversal on a binary tree is implemented through a queue. Level by level traversal of the binary tree using a queue is shown in Fig. 4.8.3 :

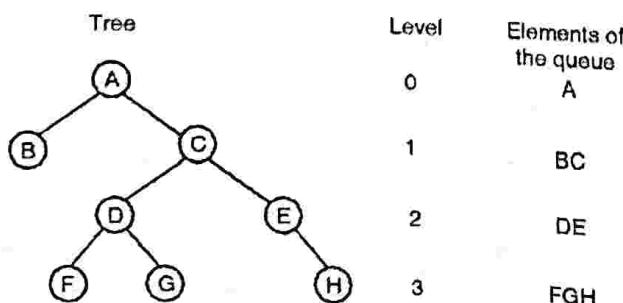


Fig. 4.8.3

Traversal starts with root node A in the queue. It is replaced by its children nodes to reach the next level. Nodes B and C are replaced by their respective children to reach the nodes at level 2 and so on.

'C' function for level by level traversal of a binary tree along with ADT queue.

```

typedef struct node //Structure of tree node
{
    char data;
    struct node *left, *right;
}node;
typedef struct Q
{
    node *data[30];
    int R, F;
}Q;
void BFT(node *T)
{
    Q q;
    node *p1, *p2;
    q.R = -1;
    q.F = -1; /* Initialize the queue */
    insert(&q, T); //Start traversal at level 0
    printf("\n %c", T->data);
    while(!empty(&q))
    {
        printf("\n ");
        /* Replace all nodes of the queue with the nodes
        at next level */
        p2 = q.R
        /* P2 points to the last element of the current level */
        do
        {
            p1 = delete(&q);
            if(p1->left != NULL)
            {
                insert(&q, p1->left);
            }
        }
        while(p1 != p2);
    }
}

```

```

        printf("%c", (p1->left)->data);
    }
    if(p1->right != NULL)
    {
        insert(&q, p1->right);
        printf("%c", (p1->right)->data);
    }
}while(p1 != p2);
}
}

```

4.8.11 Non-Recursive Algorithm for Height of a Binary Tree

Level by level traversal can be used to find the height of a tree. Level by level traversal is implemented with the help of a queue.

Algorithm

Let us assume that the tree is reference by a tree pointer T.

1. If T is NULL or T is a leaf node then
height = 0
goto step 6
2. Insert T in a queue (q)
height = 0
3. If q is empty then
goto step 6
4. Replace every node in the queue with its children i.e.
 $T \leftarrow$ Delete the front node from the queue
 $q \leftarrow$ Insert left and right children of T in queue q.
5. goto step 3.
6. Print height
7. End

4.8.11(A) 'C' Function for Height of a Tree (Non-Recursive)

```

typedef struct node // Structure of a tree node
{
    char data;
    struct node *left, *right;
} node;

```

```

typedef struct Q
{
    node * data [30];
    int R, F;
} Q;
int height(node * T)
{ Q q;
    node *p1, *p2;
    int ht; //ht for storing height
    if(T == NULL)
        return(0);
    if(T->left == NULL && T->right == NULL)
        return(0);
    ht = 0;
    initialize q;
    insert(&q, T); // start traversal at level 0
    while(! empty(&q))
    {
        /* Replace nodes of the queue with nodes at next
        level */
        p2 = q . R
        /* p2 points to the last element of the current level
        */
        do
        {
            p1 = delete(&q);
            /* delete the front element from q */
            if(p1->left != NULL)
                insert(&q, p1->left);
            if(p1->right != NULL)
                insert(&q, p1->right);
        } while(p1 != p2);
        ht = ht + 1; /* No. of levels covered*/
    }
    return(ht);
}

```

4.9 Creation of a Binary Tree from Traversal Sequence

4.9.1 Creation of Binary Tree from Preorder and Inorder Traversals

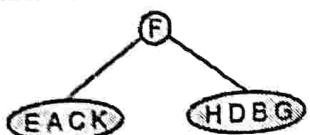
Inorder E A C K F H D B G

Preorder F A E K C D H G B

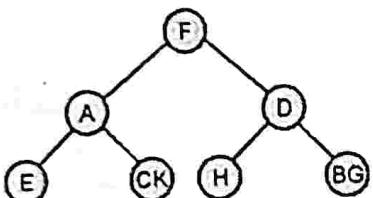
1. In preorder traversal root is the first node. Hence F is the root of the binary tree.



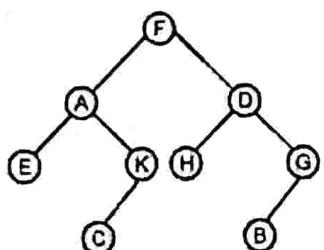
2. From inorder traversal, we can find left and right descendants of the root.



3. Among nodes E A C K, A comes first in preorder traversal. Therefore, A is root of the subtree with nodes E A C K. Among nodes H D B G, D comes first in preorder traversal. Therefore, D is root of the subtree with nodes H D B G.



4. Among nodes C K, K comes first in preorder traversal. Therefore, K is root of the subtree with nodes C K. Among nodes B G, G comes first in preorder traversal. Therefore, G is root of the subtree with nodes B G.



4.9.2 Creation of Tree from Postorder and Inorder Traversals

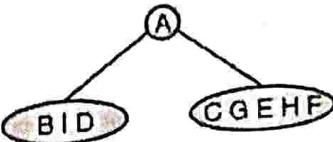
Inorder B I D A C G E H F

Postorder I D B G C H F E A

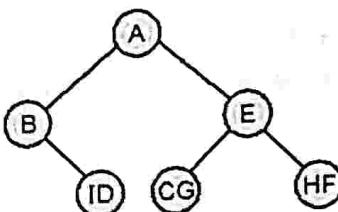
1. In postorder traversal, root is the last node. Hence A is the root of the binary tree.



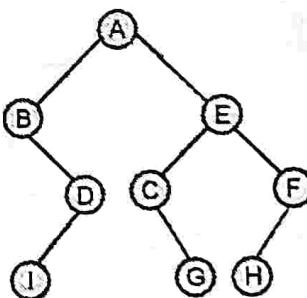
2. From inorder traversal, we can find left and right descendants of the root.



3. Among nodes B I D, B comes last in postorder traversal. Therefore, B is the root of the subtree with nodes B I D. Among nodes C G E H F, E comes last in postorder traversal. Therefore, E is the root of the subtree with nodes C G E H F.



4. Among nodes I D, D comes last in postorder traversal. Therefore, D is the root of the subtree with nodes I D. Among nodes C G, C comes last in postorder traversal. Therefore, C is the root of the subtree with nodes C G. Among nodes H F, F comes last in postorder traversal. Therefore F is the root of the subtree with nodes H F.



4.9.3 Examples on Tree Creation from Traversal Sequence

Example 4.9.1 : The postorder and inorder traversals of a binary tree are given below. Is it possible to obtain the binary tree ? If yes, obtain a binary tree from the following sequences :

Postorder : C B E H G I F D A

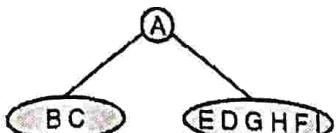
Inorder : B C A E D G H F I

Solution :

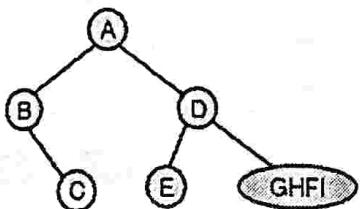
1. In postorder traversal, root is the last node of the sequence. Hence A is the root of the binary tree.



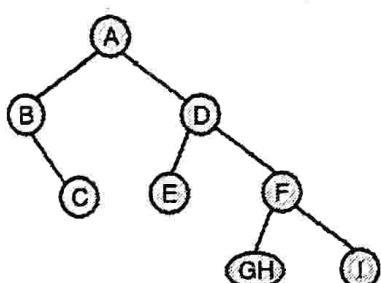
2. From inorder sequence, we can find left and right descendants of the root.



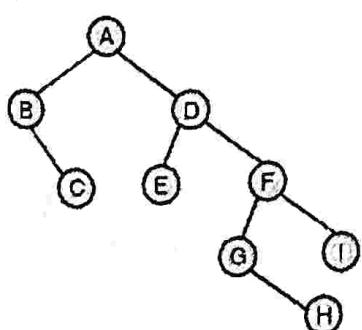
3. Among nodes BC, B comes last in postorder sequence. Therefore, B is the root of the subtree with nodes BC. Among nodes EDGHFI, D comes last in postorder sequence. Therefore, D is the root of the subtree with nodes EDGHFI.



4. Among nodes GHFI, F comes last in postorder sequence. Therefore, F is the root of the subtree with nodes GHFI.



5. Among nodes GH, G comes last in postorder sequence. Therefore, G is the root of the subtree with nodes GH.



Example 4.9.2 : Suppose the following sequences, list the nodes of a binary tree T in pre-order and in-order respectively.

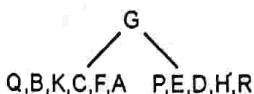
Preorder : G, B, Q, A, C, K, F, P, D, E, R, H

Inorder : Q, B, K, C, F, A, G, P, E, D, H, R

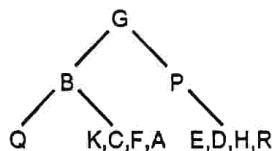
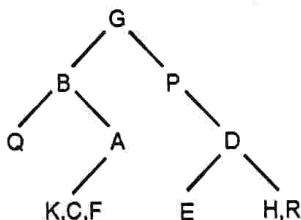
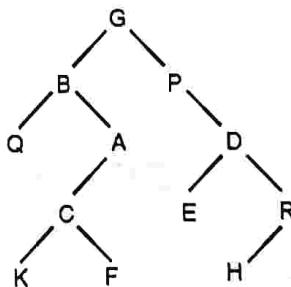
Construct a binary tree.

Solution :

- Step 1 :** In preorder traversal root is the first node. Hence G is the root. From inorder traversal, one can find left and right descendants of the root.



- Step 2 :** Among nodes (Q, B, K, C, F, A), B comes first in preorder traversal. Among nodes (P, E, D, H, R), P comes first in preorder traversal.

**Step 3 :****Step 4 :**

Example 4.9.3 : From the given traversals construct the binary tree.

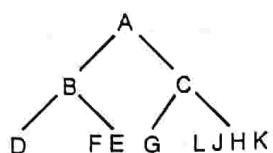
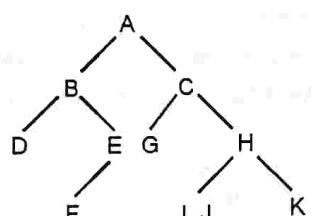
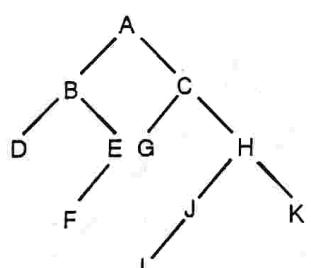
Inorder : DBFEAGCLJHK

Postorder : DFEBGLJKHCA

MU - Dec. 19, 8 Marks

**Solution :**

Step 1 : In postorder traversal, root is the last node of the sequence. Hence A is the root. From inorder sequence, we can find left and right descendants of the root.

**Step 2 :****Step 3 :****Step 4 :**

Example 4.9.4 : Construct binary tree for the pre order and inorder traversal sequences :

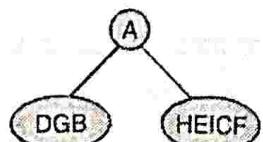
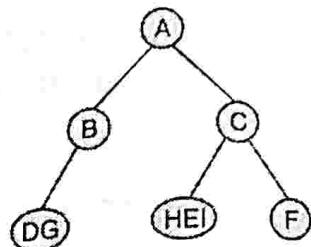
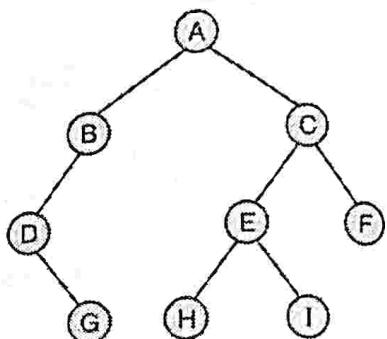
| | | | | | | | | | |
|------------|---|---|---|---|---|---|---|---|---|
| Preorder : | A | B | D | G | C | E | H | I | F |
| Inorder : | D | G | B | A | H | E | I | C | F |

MU - May 14, Dec. 16, 8 Marks

Solution :

Preorder sequence : A B D G C E H I F

Inorder sequence : D G B A H E I C F

Step 1 :**Step 2 :****Step 3 :**

Program 4.9.1 : Write a program using object oriented programming using C++ to create a binary tree if inorder & preorder

MU - Dec. 16, 5 Marks

```

#include <iostream.h>
#include <string.h>
struct node
{
    node *left, *right;
    char data;
};
class tree
{
    node *root;
public:
    void inorderl(node *);
    void preorderl(node *);
    node *createpreorder(char *, char *);
    //create a tree from preorder+inorder
    void dividepre(char *pre, char *in, char *prel,
    char *pre2, char *inl, char *in2);
};

void createpre(char *pre, char *in)
{
    root = createpreorder(pre, in);
}

void preorder() { preorderl(root); }

void inorder() { inorderl(root); }
}
  
```

```

int belongs(char x, char a[ ]);  

//find whether the character x is in a[ ]  

void tree::preorder1(node *p)  

{  

    if(p!=NULL)  

    {  

        cout<<p->data;  

        preorder1(p->left);  

        preorder1(p->right);  

    }  

}  

void tree::inorder1(node *p)  

{  

    if(p!=NULL)  

    {  

        inorder1(p->left);  

        cout<<p->data;  

        inorder1(p->right);  

    }  

}  

node* tree::createpreorder(char *pre, char *ino)  

{  

    /* Preorder and inorder are preorder and inorder  

    sequences respectively */  

    char pre1[20], pre2[20], in1[20], in2[20];  

    node *p = NULL;  

    //Preorder sequence gives the root  

    if(strlen(pre) == 0)  

        return(NULL);  

    p = new node;  

    p->data = pre[0];  

    /*Divide the preorder and inorder sequence for left  

    and right subtrees*/  

    dividepre(pre, ino, pre1, pre2, in1, in2);  

    p->left = createpreorder(pre1, in1);  

    p->right = createpreorder(pre2, in2);  

    return(p);  

}

```

```

void tree::dividepre(char *pre, char *in, char *pre1,  

                     char *pre2, char *in1, char *in2)  

{  

    int i, j, k;  

    for(i = 0; in[i]! = pre[0]; i++)  

        // Left subtree, inorder  

        in1[i] = in[i];  

    in1[i] = '\0';  

    i++;  

    for(j = 0; in[i+j]! = '\0'; j++)  

        // Right subtree, inorder  

        in2[j] = in[i+j];  

    // divide the preorder sequence  

    in2[j] = '\0';  

    i = j = 0;  

    for(k = 1; pre[k]! = '\0'; k++)  

        if(belongs(pre[k], in1))  

            // Belongs to left subtree  

            pre1[i++] = pre[k];  

        else  

            pre2[j++] = pre[k];  

    // Belongs to right subtree  

    pre1[i] = pre2[j] = '\0';  

}  

int belongs(char x, char a[ ])  

{  

    int i;  

    for(i = 0; a[i]! = '\0'; i++)  

        if(a[i] == x)  

            return(1);  

    return(0);  

}  

void main()  

{  

    char pre[20], in[20], post[20];  

    tree T;  

    cout<<"\n\n Create a tree from preorder and  

    inoder sequence:\n ";
}

```



```

cout << "\n Enter Preorder Sequence : ";
cin >> pre;
cout << "\n Enter inorder sequence : ";
cin >> in;
T.createpre(pre, in);
cout << "\n \n Preorder on the tree : ";
T.preorder();
cout << "\n Inorder on the tree : ";
T.inorder();
cout << "\n Enter inorder sequence : ";
cin >> in;
T.createpost(post, in);
cout << "\n Preorder on the tree : ";
T.preorder();
cout << "\n Inorder on the tree : ";
T.inorder();
}

```

Output

Create a tree from preorder and inorder sequence:
Enter Preorder Sequence : -+a*bcd
Enter inorder sequence : a+b*c-d
Preorder on the tree : -+a*bcd
Inorder on the tree : a+b*c-d
Postorder on the tree : abc*+d-

4.10 Binary Search Tree (BST)

MU - May 15

University Question

Q. What is an binary search tree ? Explain with an example.
(May 15, 5 Marks)

4.10.1 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.

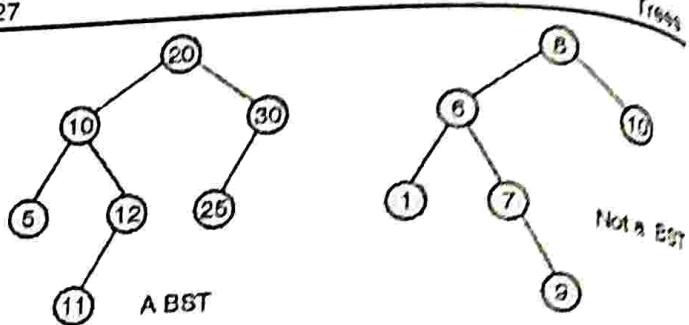


Fig. 4.10.1 : Left tree is a BST, right tree is not a binary search tree

- (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. 4.10.1, 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.

4.10.2 Operations on a Binary Search Tree

- (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;

```

4.10.2(A) Initialize Operation

Initially the tree is empty and hence the referencing pointer root should be set to NULL.

'C' function for Initialize

```

BSTnode* initialize()
{
    return(NULL);
}

```

4.10.2(B) Find Operation

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

```

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);
}

```

4.10.2(C) Make Empty Operation

This function deletes every node of the tree. It also releases the memory acquired by the node.

'C' Function to Release Memory :

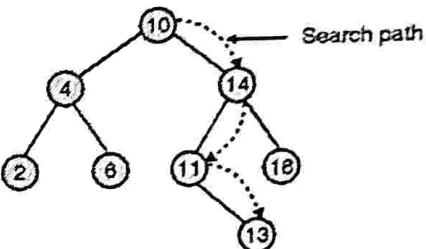
```

BSTnode *makeempty(BSTnode *root)
{
    if(root != NULL)
    {
        makeempty(root → left);
        makeempty(root → right);
        free(root);
    }
    return(NULL);
}

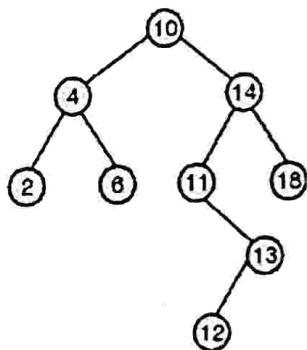
```

4.10.2(D) Insert Operation

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. 4.10.2 : 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. 4.10.2 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);
}
```

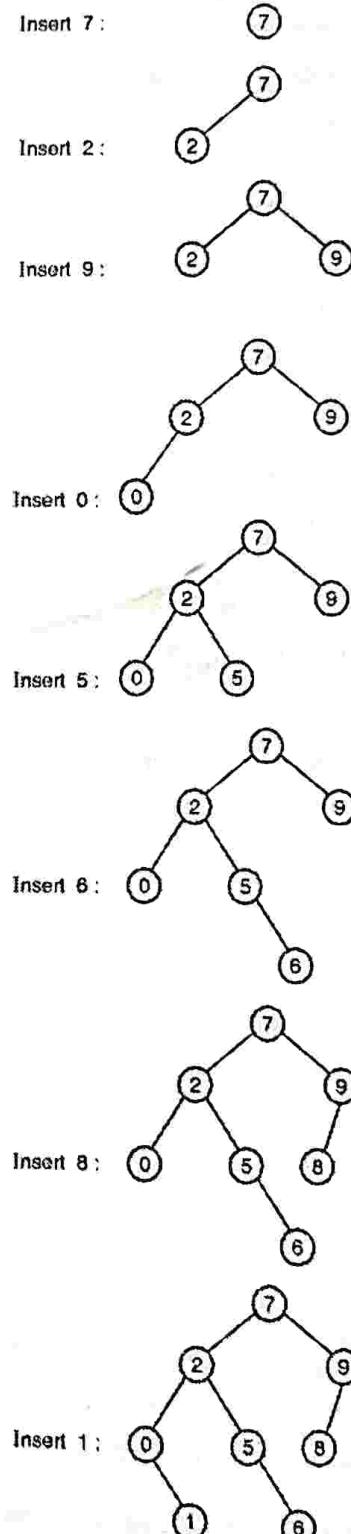
'C' Function for insert() - Non Recursive

```
BSTnode *insert(BST *T, int x)
{
    BSTnode *p, *q, *r;
    // acquire memory for the new node
    r = (BST*)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);
}
```

4.10.2(E) Example on Creation of a BST

Example 4.10.1 : Insert the integers 7, 2, 9, 0, 5, 6, 8, 1 into a binary search tree by repeated application of insert operation.

Solution :



4.10.2(F) Delete Operation

MU - May 19

University Question

Q. Explain different cases for deletion of a node in binary search tree. Write function for each case. (May 19, 10 Marks)

In order to delete a node, we must find the node to be deleted. The node to be deleted may be :

- A leaf node
- A node with one child
- 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. 4.10.3. 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.

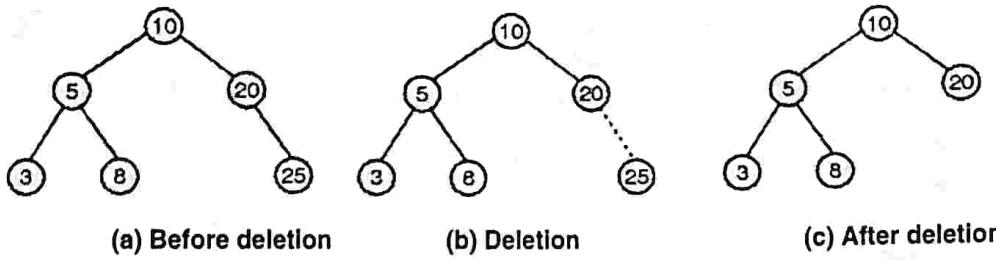


Fig. 4.10.3 : Deletion of node (25), a leaf node

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. 4.10.4. 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. 4.10.5 shows deletion of node (15). It is left child of its parent node (20).

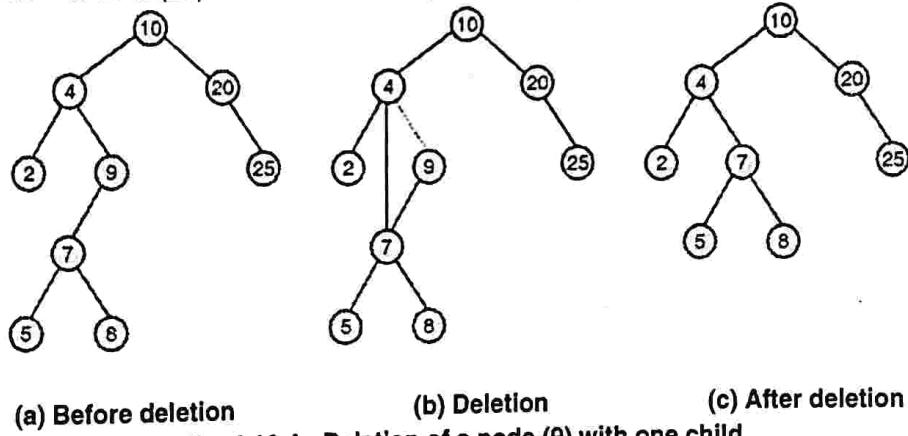


Fig. 4.10.4 : Deletion of a node (9) with one child

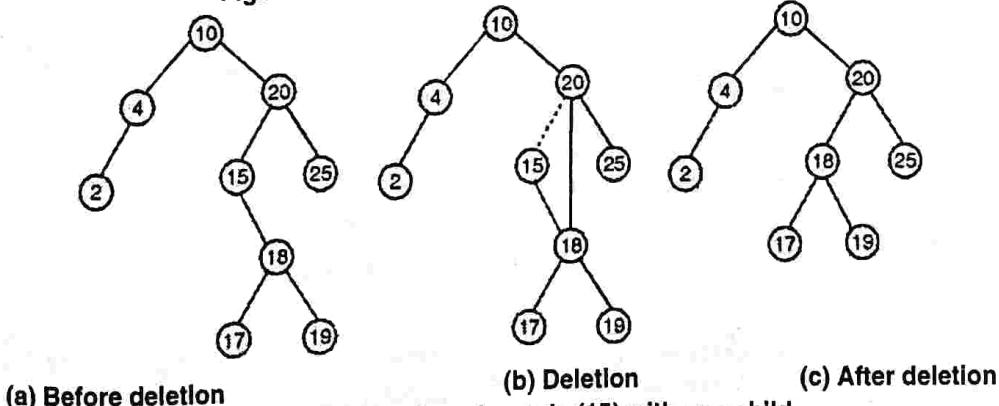


Fig. 4.10.5 : Deletion of a node (15) with one child



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. 4.10.6. 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. 4.10.6(b). Now, the node q is deleted.

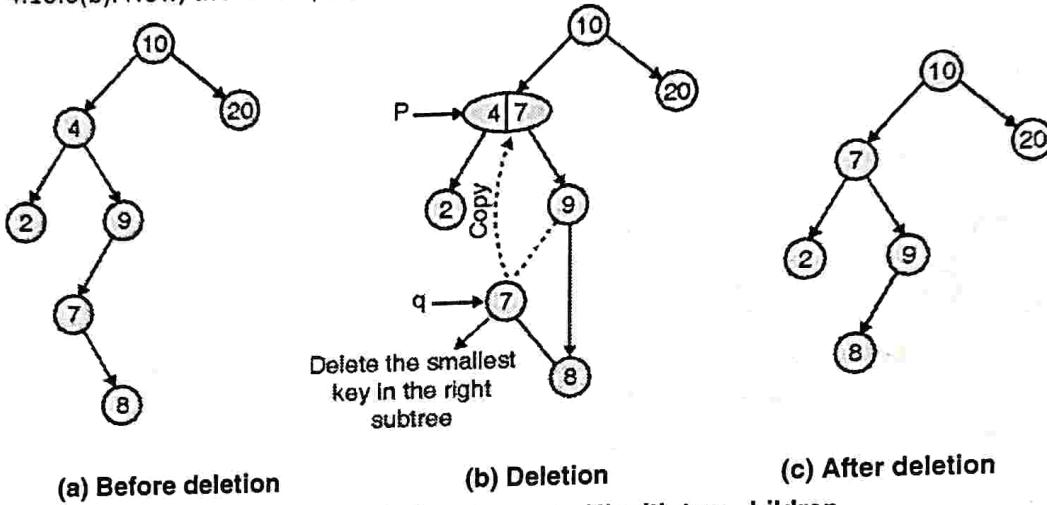


Fig. 4.10.6 : Deletion of a node (4) with two children

'C' Function for Deletion

```
BSTnode *delete(BSTnode *T, int x)
{
    if(T == NULL)
    {
        printf("\n Element 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, temp ->data);
return(T);
}

```

4.10.2(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("\n Enter no. of nodes :");
    scanf("%d", &n);
    printf("\n Enter tree values :");
    for(i = 0; i < n; i++)
    {
        scanf("%d", &x);
        root = insert(root, x);
    }
    return(root);
}

```

4.10.2(H) Find Min

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);
}

```

4.10.2(I) Find Max

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);
}

```

4.10.3 Program for Various Operations on BST

Program 4.10.1 : Program showing various operations on a binary search tree.

OR Write a program in 'C' for deletion of a node from a Binary Search Tree. The program should consider all the cases.

MU - Dec. 13, May 16, 10 Marks

OR Write a program to implement Binary Search Tree (BST), show BST for the following input: 10, 5, 4, 12, 15, 11, 3

MU - Dec. 17, 10 Marks

```

#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("\n inorder 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("\n largest key in the tree = %d",
p->data);
printf("\n **** delete operation ****");
printf("\n Enter 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 *delet(BSTnode *T, int x)
{
    BSTnode *temp;
    if(T == NULL)
    {
        printf("\n Element 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
temp = find_min(T->right);
T->data = temp->data;
T->right = delet(T->right, x);
return(T);
}

BSTnode *create()
{

```

```

int n, x, i;
BSTnode *root;
root = NULL;
printf("\n Enter no. of nodes :");
scanf("%d", &n);
printf("\n Enter 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

```

Explanation :

BST for 10, 5, 4, 12, 15, 11, 3 are shown in Fig. P. 4.10.1.

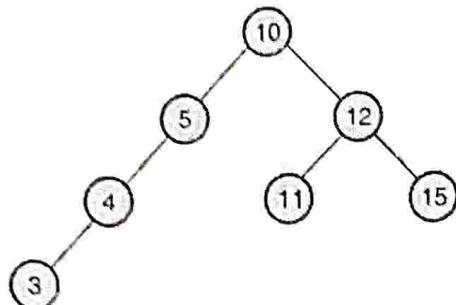


Fig. P. 4.10.1

Program 4.10.2 : Write a program in 'C' to implement Binary search on sorted set of integers. **MU - May 14, 10 Marks**

```

//*****To implement binary search :*****//

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

//int stepcount = 0, swapcount = 0, compcount = 0;
int binsearch(int a[], int i, int j, int key); //Recursive

void main()
{
    int a[30], key, n, i, result;
    clrscr();
    printf("\nEnter No. of elements : ");
    scanf("%d", &n);
    printf("\nEnter a sorted list of %d elements : ", n);
    for(i = 0; i < n; i++)
        scanf("%d", &a[i]);
    printf("\nEnter 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);
    /* printf("\n No. of steps = %d", stepcount);
    printf("\n No. of swaps = %d", swapcount);
    printf("\n No. of comparisons = %d", compcount); */
    getch();
}

int binsearch(int a[], int i, int j, int key)
  
```

```

    {
        int c;
        if(i > j)
        {
            //compcount++;
            //stepcount += 2;
            return(-1);
        }
        c = (i+j)/2;
        if(key == a[c])
        {
            compcount++;
            stepcount += 2;
            return(c);
        }
        if(key > a[c])
        {
            // stepcount += 1;
            return(binsearch(a, c+1, j, key));
        }
        // stepcount += 1;
        return(binsearch(a, i, c-1, key));
    }
  
```

Output

Enter No. of elements : 5

Enter a sorted list of 5 elements : 2

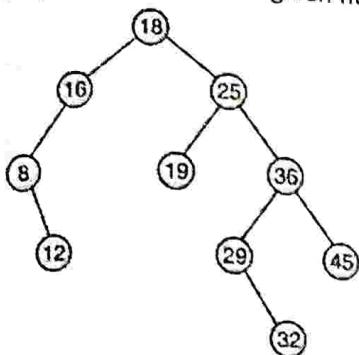
4
1
9
7

Enter the element to be searched : 9
Found at location = 4

Example 4.10.3 : Consider the following list of numbers:
18, 25, 16, 36, 08, 29, 45, 12, 32, 19
Create a binary search tree using these numbers and display them in a non-decreasing order. Write a 'C' program for the same.

MU - Dec. 14, 10 Marks

Solution : Binary search tree of the given numbers



List in non-decreasing order

8 12 16 18 19 25 29 32 36 45

Program

```

#include <stdio.h>
#include <stdlib.h>
typedef struct BSTnode
{
    int data;
    struct BSTnode * left, * right;
} BSTnode;
BSTnode *insert(BSTnode *, int);
BSTnode * create( );
void inorder(BSTnode * T) ;
void main()
{
    BSTnode * root;
    root = create();
    inorder(root);
}
void inorder(BSTnode * T)
{
    if(T!=NULL)
    {
        inorder(T->left);
        printf("%5d", T->data);
        inorder(T->right);
    }
}
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 *create()
{
    int n, x, i;
    BSTnode *root;
    root = NULL;
    printf("\n Enter no. of nodes :");
    scanf("%d", &n);
    printf("\n Enter tree values :");
    for(i = 0; i<n; i++)
    {
        scanf("%d", &x);
        root = insert(root, x);
    }
    return(root);
}
  
```

4.11 AVL Trees

MU - Dec. 13, May 15, May 17, Dec. 17

University Questions

- | | |
|----------------------------------|---------------------|
| Q. Discuss AVL trees. | (Dec. 13, 5 Marks) |
| Q. What is an AVL tree. | (May 15, 5 Marks) |
| Q. Explain AVL trees. | (May 17, 3 Marks) |
| Q. Write short note on AVL tree. | (Dec. 17, 10 Marks) |

- An AVL (Adelson-Velskii and Landis) tree is a height balance tree. These trees are binary search trees in which the heights of two siblings are not permitted to differ by more than one.



- i.e. $| \text{Height of the left subtree} - \text{height of the right subtree} | \leq 1$
- Searching time in a binary search tree is $O(h)$, where h is the height of the tree. For efficient searching, it is necessary that height should be kept to minimum.
- A full binary search tree with n nodes will have a height of $O(\log_2 n)$. In practice, it is very difficult to control the height of a BST. It lies between $O(n)$ to $O(\log_2 n)$. An AVL tree is a close approximation of full binary search tree.

4.11.1 Height Balanced Tree

- An empty tree is height balanced.
- A binary tree with h_l and h_r as height of left and right subtree respectively is height balanced if $| h_l - h_r | \leq 1$
- A binary tree is height balanced if every subtree of the given tree is height balanced.
- An AVL tree is a height balanced binary search tree.

4.11.2 Balance Factor

- The balance factor, $BF(T)$ of a node T in a binary tree is defined as $h_l - h_r$ where h_l and h_r are the heights of the left and the right subtrees of T . A binary search tree with balance factors is shown in the Fig. 4.11.1.

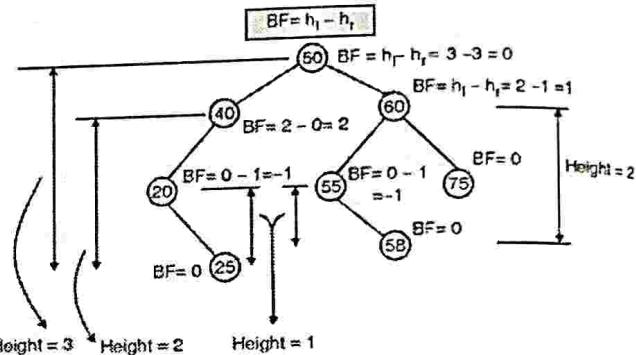


Fig. 4.11.1 : A sample BST with balance factors

- Tree of Fig. 4.11.1 is not an AVL tree. The balance factor of the node with data 40 is +2. Trees of Fig. 4.11.2 are non-AVL trees. Trees of Fig. 4.11.3 are AVL-trees.

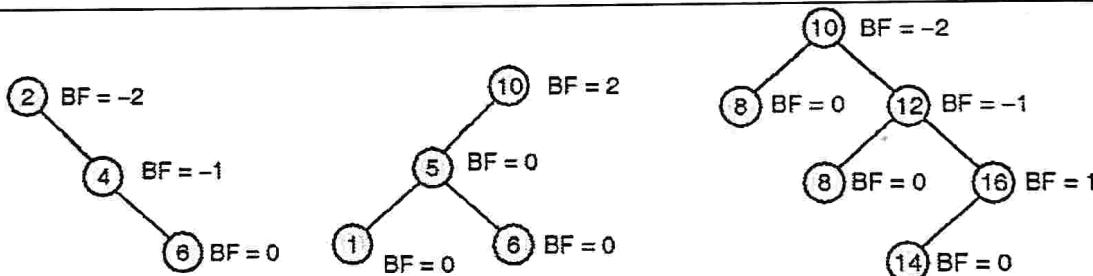


Fig. 4.11.2 : Non-AVL trees

- The balance factor of a node in an AVL tree could be $-1, 0$ or 1 .
- If the balance factor of a node is 0 then the heights of the left and right subtrees are equal.

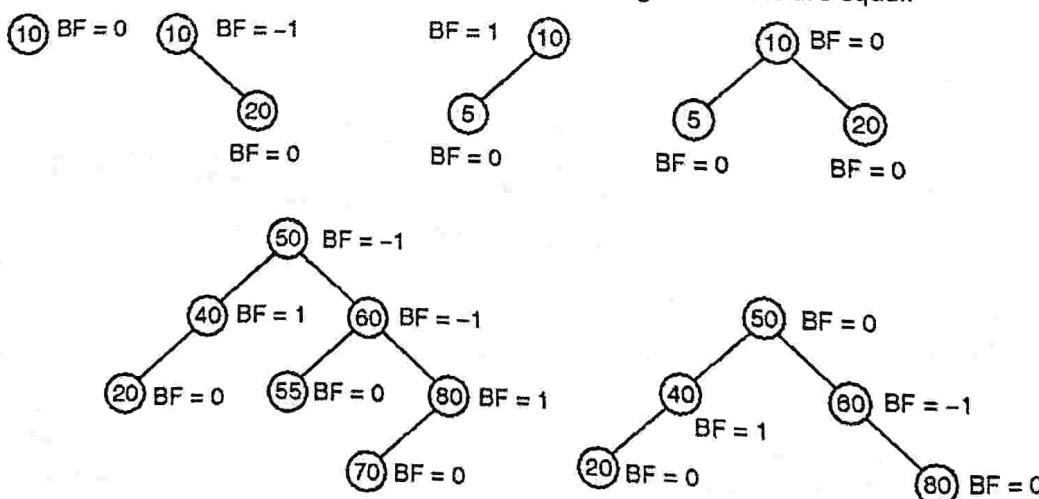


Fig. 4.11.3 : AVL trees

- If the balance factor of a node is +1 then the height of the left subtree is one more than the height of the right subtree.
- If the balance factor of a node is -1 then the height of the left subtree is one less than the height of the right subtree.

4.11.3 Structure of a Node in AVL Tree

Operations on AVL tree requires calculation of balance factors of nodes. To represent a node in AVL tree, a new field is introduced. This new field stores the height of the node. A node in AVL tree can be defined in the following way :

```
typedef struct node
{
    int data ;
    struct node *left, *right;
    int height;
} node;
```

4.11.4 'C' Function for Finding the Balance Factor of a Node

```
int BF(node *T)
{
    int lh, rh;
    if(T->left == NULL)
        lh = 0;
    else
        lh = 1+T->left->height;
    if(T->right == NULL)
        rh = 0;
    else
        rh = 1+ T->right->height;
    return(lh-rh);
}
```

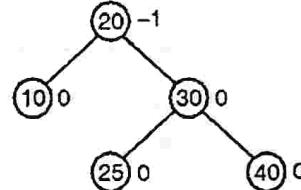
/ The function height(), for finding the height of node is given below */*

```
int height(node *T)
{
    int lh, rh;
    if(T->left == NULL)
        lh = 0;
    else
```

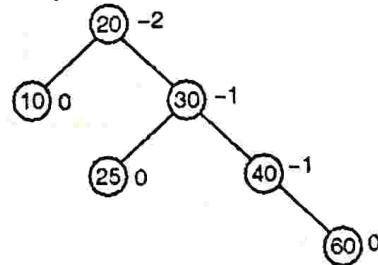
```
lh = 1 + T->left->height
if(T->right == NULL)
    rh = 0;
else
    rh = 1 + T->right->height
if(lh>rh)
    return(lh);
return(rh);
}
```

4.11.5 Insertion of a Node into an AVL Tree

- Insertion of a new data, say k into an AVL tree is carried out in two steps :
 - (1) Insert the new element, treating the AVL tree as a binary search tree.
 - (2) Update the balance factors (information, height) working upward from the point of insertion to the root. It should be clear that after insertion, the nodes on the path from point of insertion to the root may have their height altered.
- The steps to insert a new element with value k into an AVL tree begins with comparing k with the value stored in the root.



(a) A sample AVL tree with balance factors



(b) Tree of Fig. 4.11.4(a) after Insertion of 60. Tree is not longer an AVL tree

Fig. 4.11.4

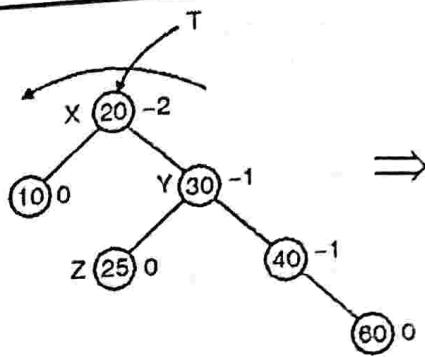
- If k is found to be larger than the value stored in T, then insertion is carried out into the right subtree else insertion is carried out into the left subtree.



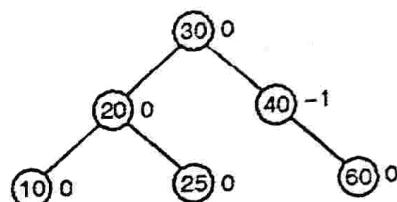
- The recursion terminates when the left or right subtree to which insertion is to be made happens to be empty. Consider the AVL tree of Fig. 4.11.4. Balance factor of each node is shown against the node.
- A new element with value 60 is inserted in a way we insert an element in a binary search tree. After insertion of 60, the balance factor of root has become -2.
- Hence, it is no longer an AVL tree. The balance factor of the root has become -2, because the height of its right subtree rooted at (30) has increased by 1.
- Rebalancing of the tree is carried out through rotation of the deepest node with $BF = 2$ or $BF = -2$.

Rotation :

- Fig. 4.11.5 shows the rotation of tree of Fig. 4.11.4. After rotation, the tree becomes an AVL tree.
- Balance factor of node X in Fig. 4.11.5(a) is -2.
- Balance factor of -2, tells that the right subtree of X is heavier.
- In order to rebalance the tree, the tree rooted at X (node with $BF = -2$) is rotated left.



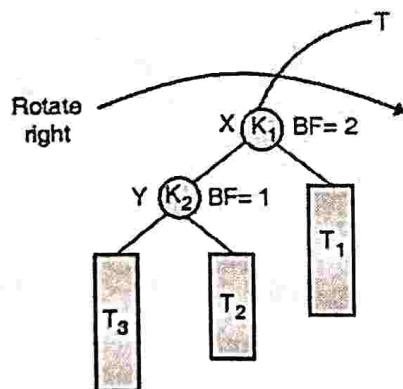
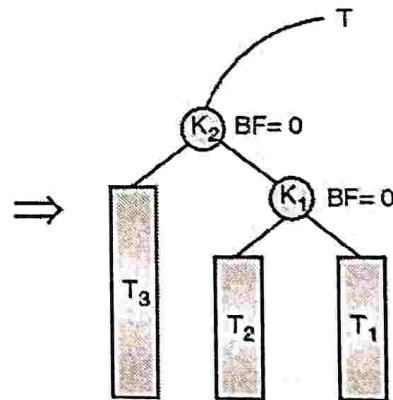
(a) Before rotation



(b) After rotation

Fig. 4.11.5 : AVL property is destroyed by insertion of 60.
AVL property is fixed by left rotation of tree rooted at X

4.11.5(A) Rotate Left

(a) Tree with $BF = -2$ at node X

(b) Tree after rotation

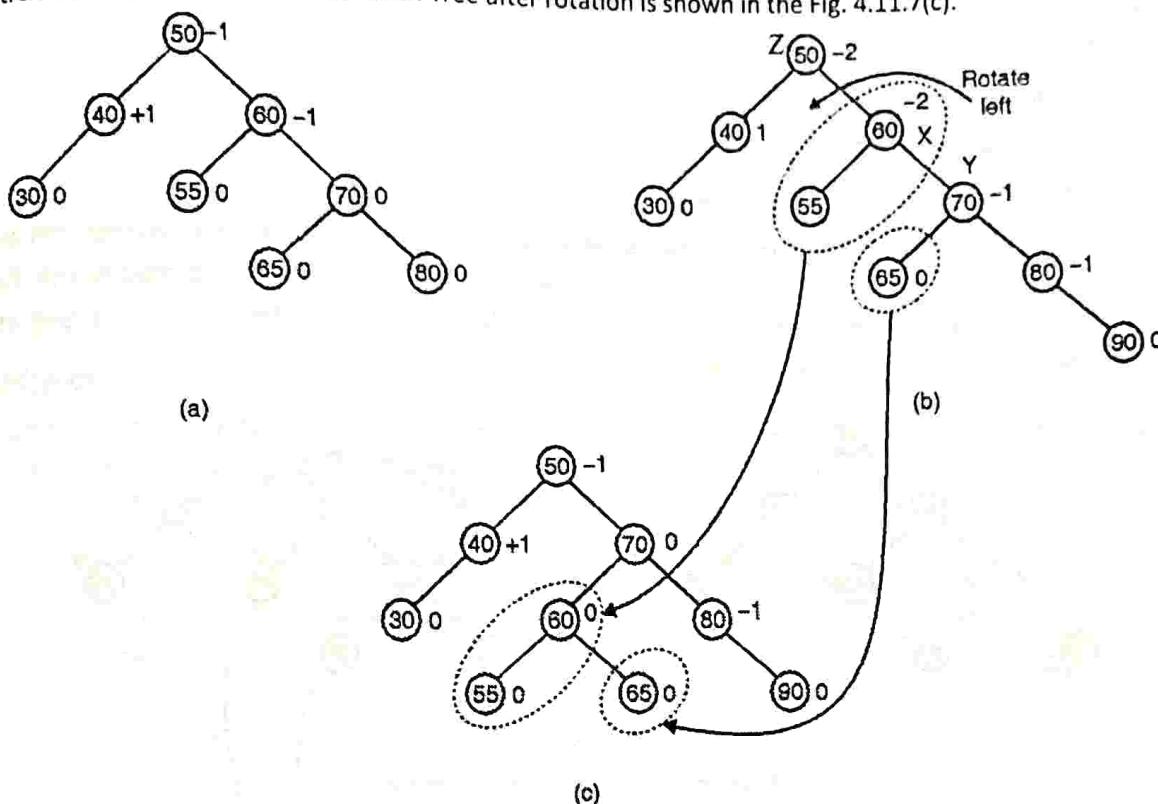
Fig. 4.11.6

- When a tree rooted at X is rotated left, the right child of X i.e. Y will become the root.
- The node X will become the left child of Y.
- Tree T_2 , which was the left child of Y will become the right child of X.
- A program segment to rotate left a tree T , rooted at node X (as shown in Fig. 4.11.6).


```
node *temp;
temp = Y → left; save the address of left child of Y.
T = Y; Node Y becomes the root node.
Y → left = X; X becomes the left child of Y.
X → right = temp; left child of Y becomes the right child of X.
```

The tree of Fig. 4.11.7(a) is an AVL tree. After insertion of the element 90, the tree no longer remains an AVL tree. The balance factor of nodes X and Z has becomes -2 shown in Fig. 4.11.7(b).

In order to restore back the balance nature, the node X (the deepest node with BF equal to ± 2) is rotated left. After rotation, tree once again becomes an AVL tree. Tree after rotation is shown in the Fig. 4.11.7(c).



(a) An AVL tree (b) Tree after insertion of 90 (c) Tree after rotation

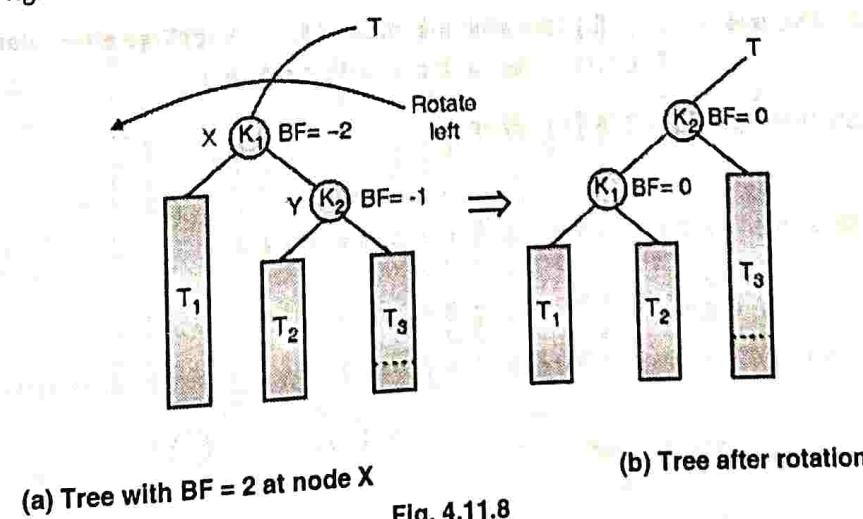
Fig. 4.11.7 : An example of left rotation

4.11.5(B) Rotate Right

When a tree rooted at X is rotated right, the left child of X i.e. Y will become the root.

The node X will become the right child of Y.

Tree T_2 , which was the right child of Y will become the left child of X.



(a) Tree with $BF = 2$ at node X

(b) Tree after rotation

Fig. 4.11.8



Balance factor of +2, tells that the left subtree of X is heavier. In order to rebalance the tree, the tree rooted at X (node with BF = 2) is rotated right;

A program segment to rotate right a tree T, rooted at node X (as shown in Fig. 4.11.8).

node *temp;

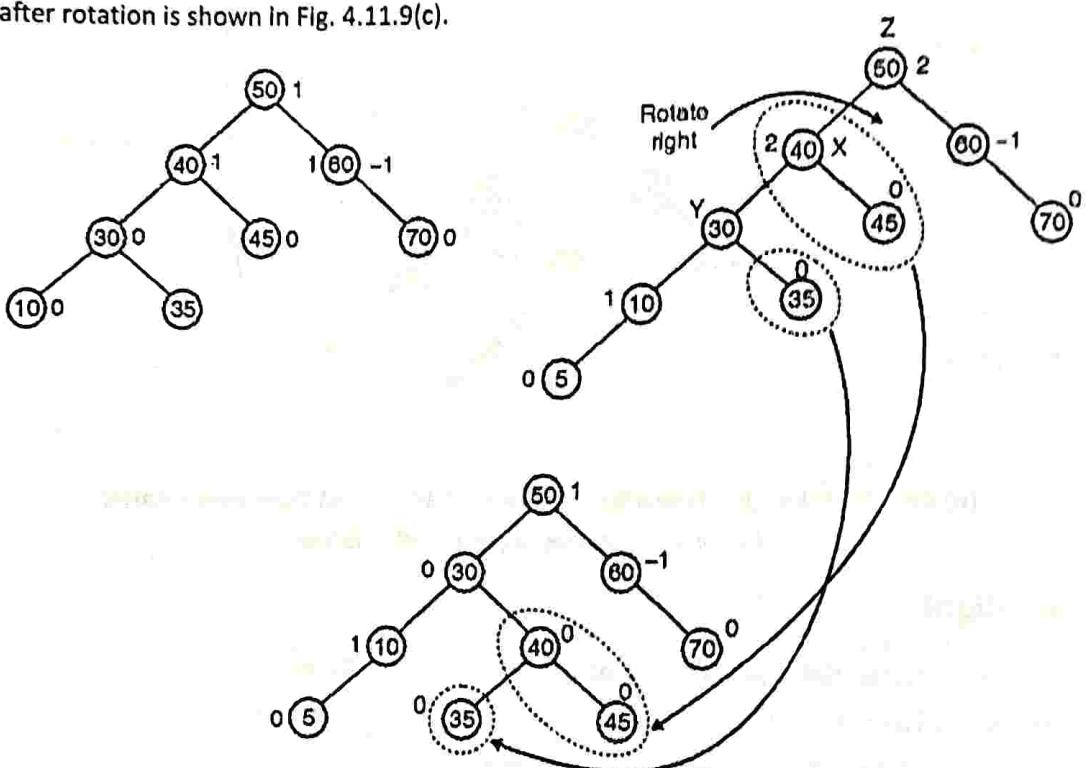
temp = Y → right; save the address of right child of Y.

T = Y; Node Y becomes the root node.

Y → right = X ; X becomes the right child of Y.

X → left = temp; right child of Y becomes the left child of X.

The tree of Fig. 4.11.9(a) is an AVL tree. After insertion of the element with value 5, the tree no longer remains an AVL tree. The balance factors of nodes X and Z has become 2. Shown in Fig. 4.11.9(b). In order to restore back the balanced nature, the node X (the deepest node with BF equal ± 2) is rotated right. After rotation, the tree has once again become an AVL tree. Tree after rotation is shown in Fig. 4.11.9(c).



(a) An AVL tree

(b) Tree after insertion of 5

(c) Tree after rotation

Fig. 4.11.9 : An example of right rotation

4.11.5(C) Single Rotation and Double Rotation

Single rotation :

LL : Let X be the node with BF equal to +2 after insertion of the new node A.

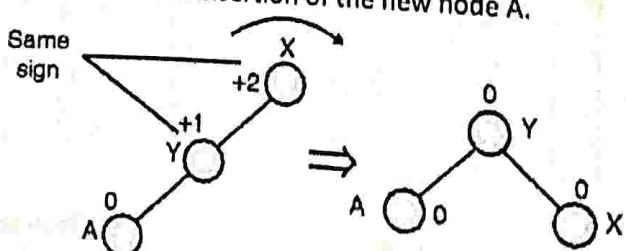


Fig. 4.11.10(a) : Situation known as LL (left of left)

New node A is inserted in the left subtree of the left subtree of X. Balance nature of the tree can be restored through single right rotation of the node X. It is shown in the Fig. 4.11.10(a).

RR : Let X be the node with BF equal to -2, after insertion of the new node A.

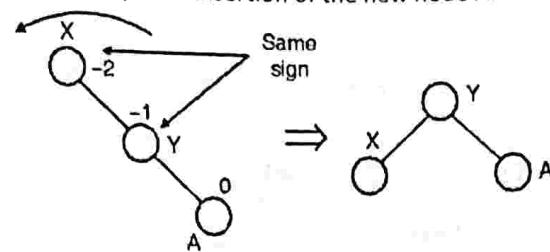


Fig. 4.11.10(b) : Situation known as RR (right of right)

New node A is inserted in the right subtree of the right subtree of X. Balance nature of the tree can be restored through single left rotation of the node X is shown in the Fig. 4.12.10(b).

Double rotation :

LR : Let X be the node with BF equal to +2, after insertion of the new node A. Balance factor of the node Y, the left child of the node X becomes -1 after insertion of A.

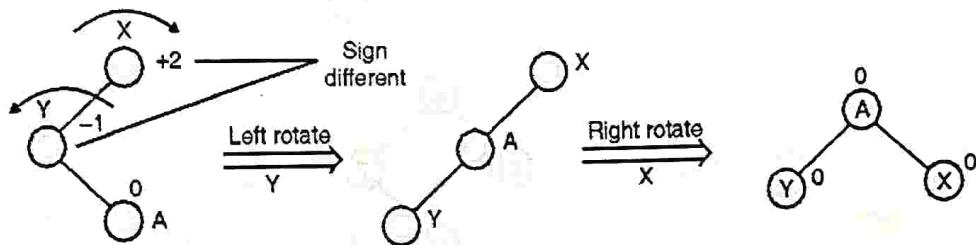


Fig. 4.11.10(c) : Situation known as LR (right of left)

New node A is inserted in the right subtree of the left subtree of X. Balance nature of the tree can be restored through double rotation

- (a) node Y is rotated left (b) node X is rotated right.

It is shown in the Fig. 4.11.10(c).

RL : Let X be the node with $BF = -2$, after insertion of the new node A. Balance factor of the node Y, the right child of the node X becomes +1 after insertion of A.

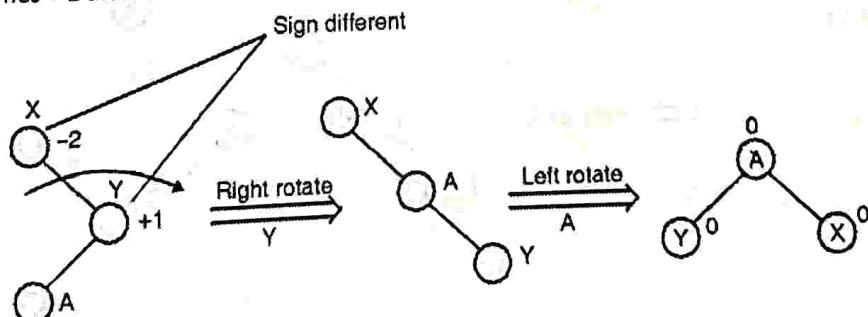


Fig. 4.11.10(d) : Situation known as RL (left of right)

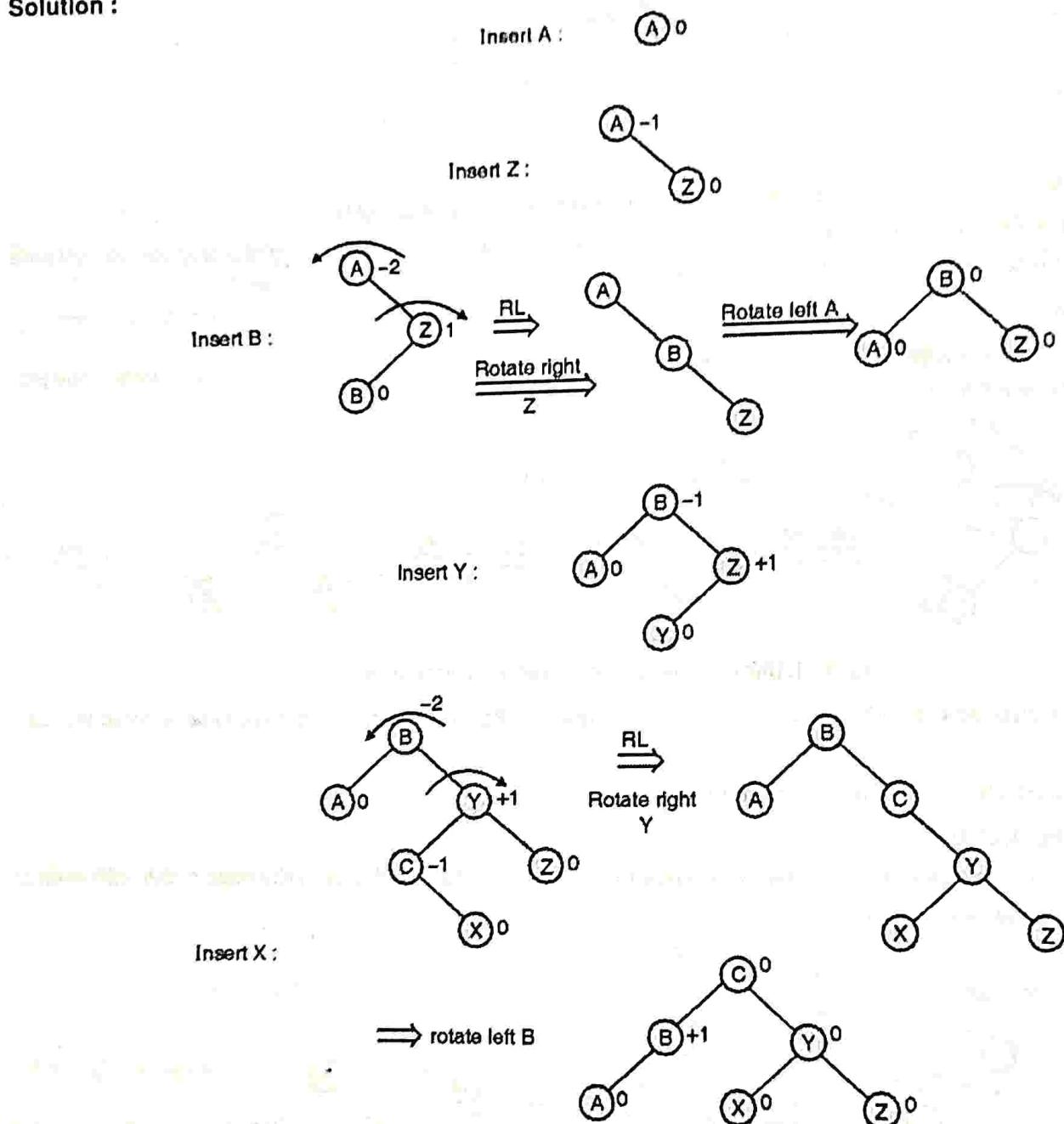
New node A is inserted in the left subtree of the right subtree of X. Balance nature of the tree can be restored through double rotation.

- (a) node Y is rotated right. (b) node X is rotated left.

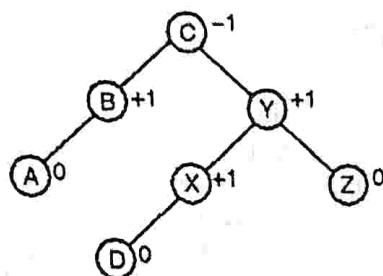
Situation is explained through Fig. 4.11.10(d).

Example 4.11.1 : Draw diagram to show different stages during the building of AVL tree for the following sequence of keys; A, Z, B, Y, C, X, D, U, E. In each case show the balanced factor of all the nodes and name the type of rotation used for balancing.

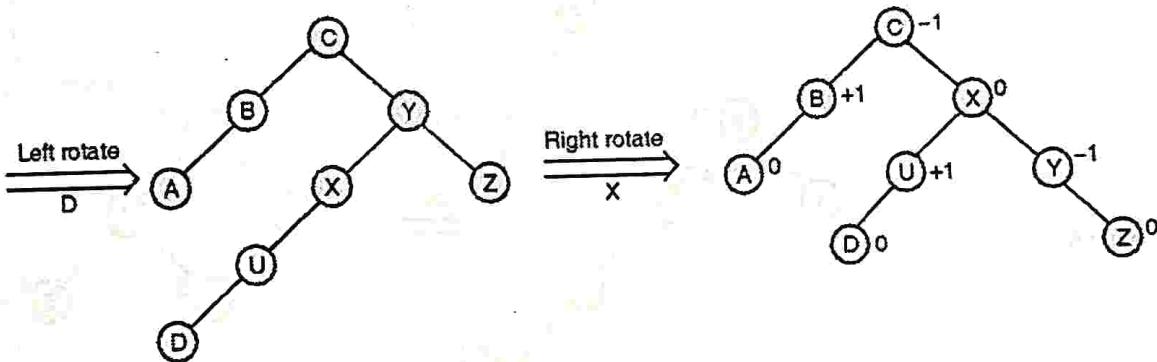
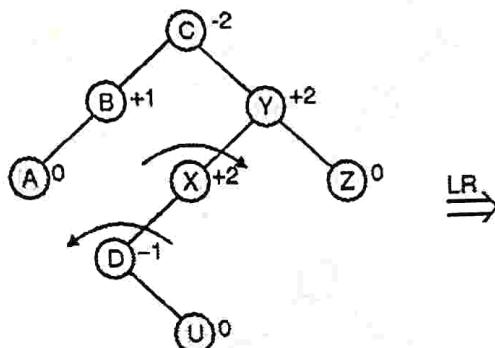
Solution :



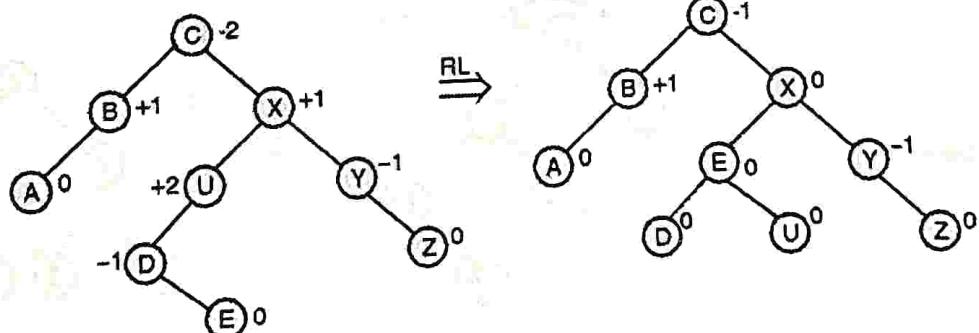
Insert D :



Insert U :



Insert E :



Example 4.11.2 : Insert the following sequence of keys into an AVL tree. Find out the type of rotation required in each case

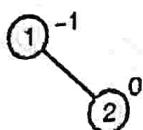
1 2 3 4 8 7 6 5 11 10 12

**Solution :**

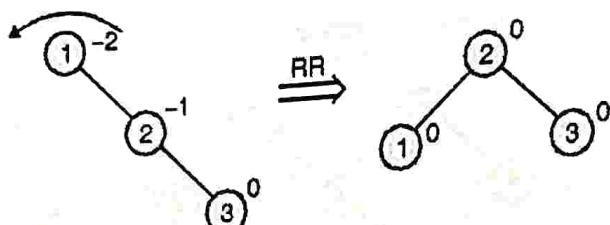
Insert 1 :



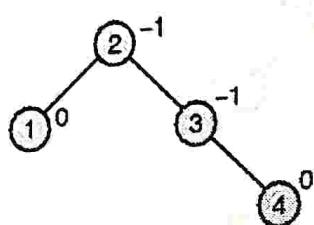
Insert 2 :



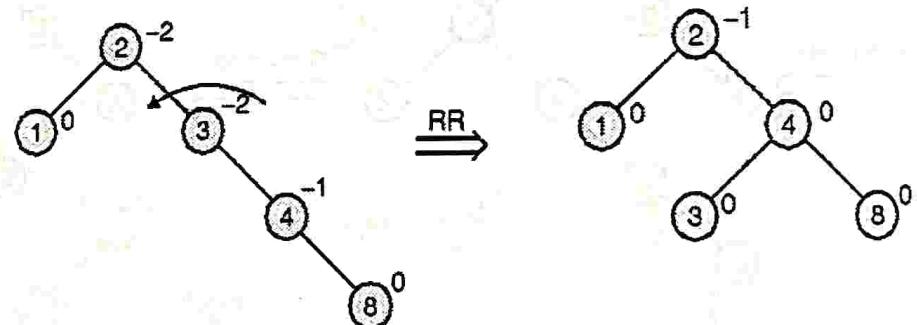
Insert 3 :



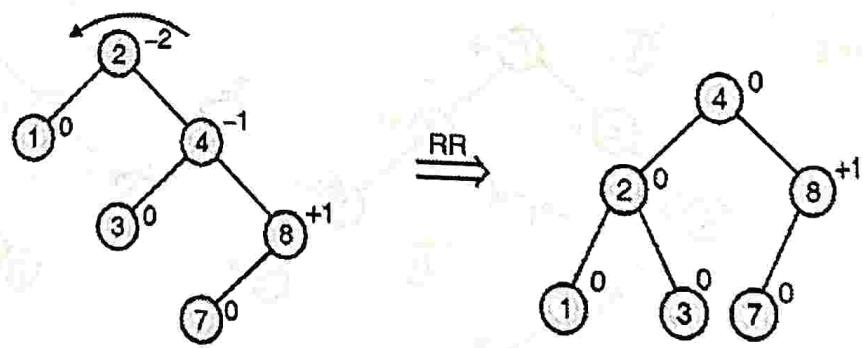
Insert 4 :



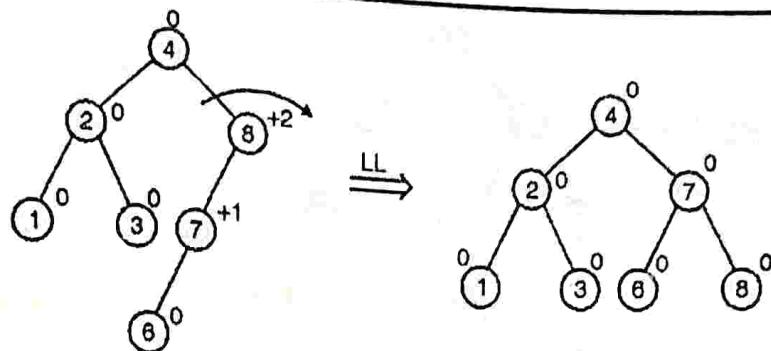
Insert 8 :



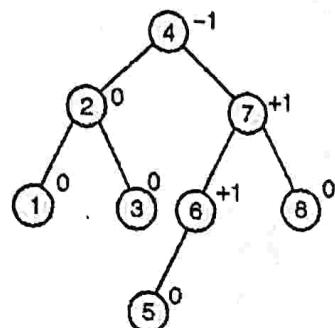
Insert 7 :



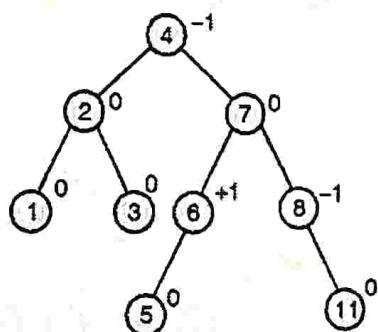
Insert 6 :



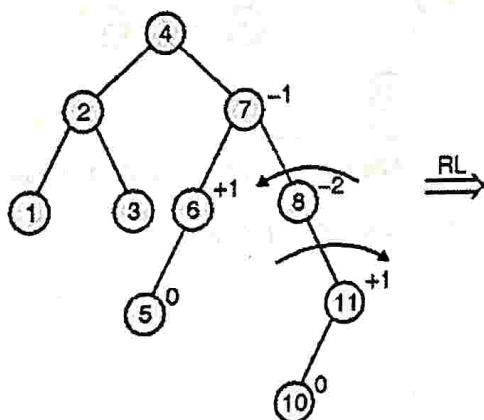
Insert 5 :

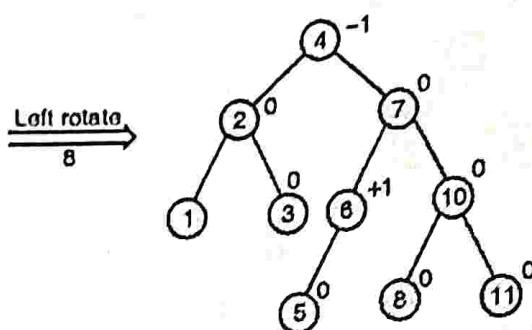
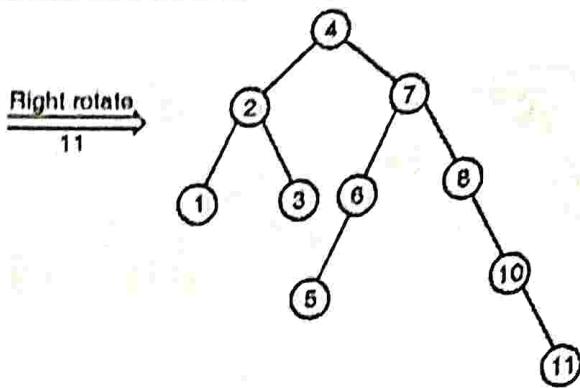


Insert 11 :

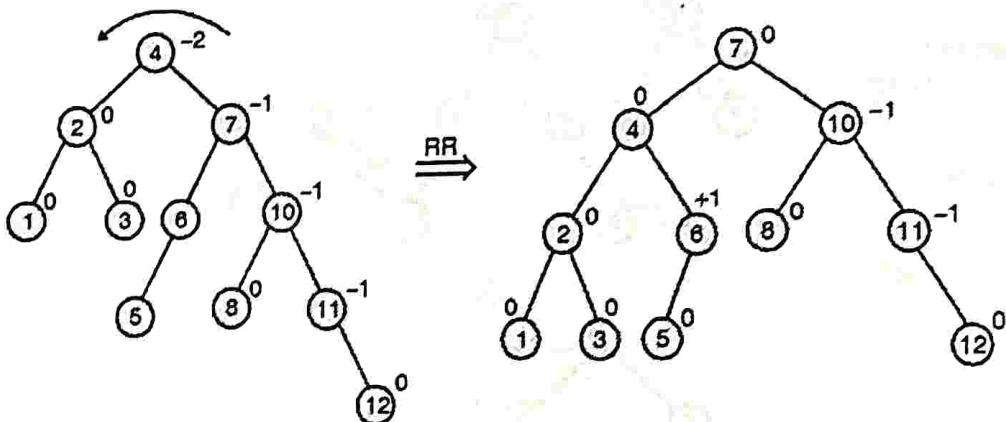


Insert 10 :





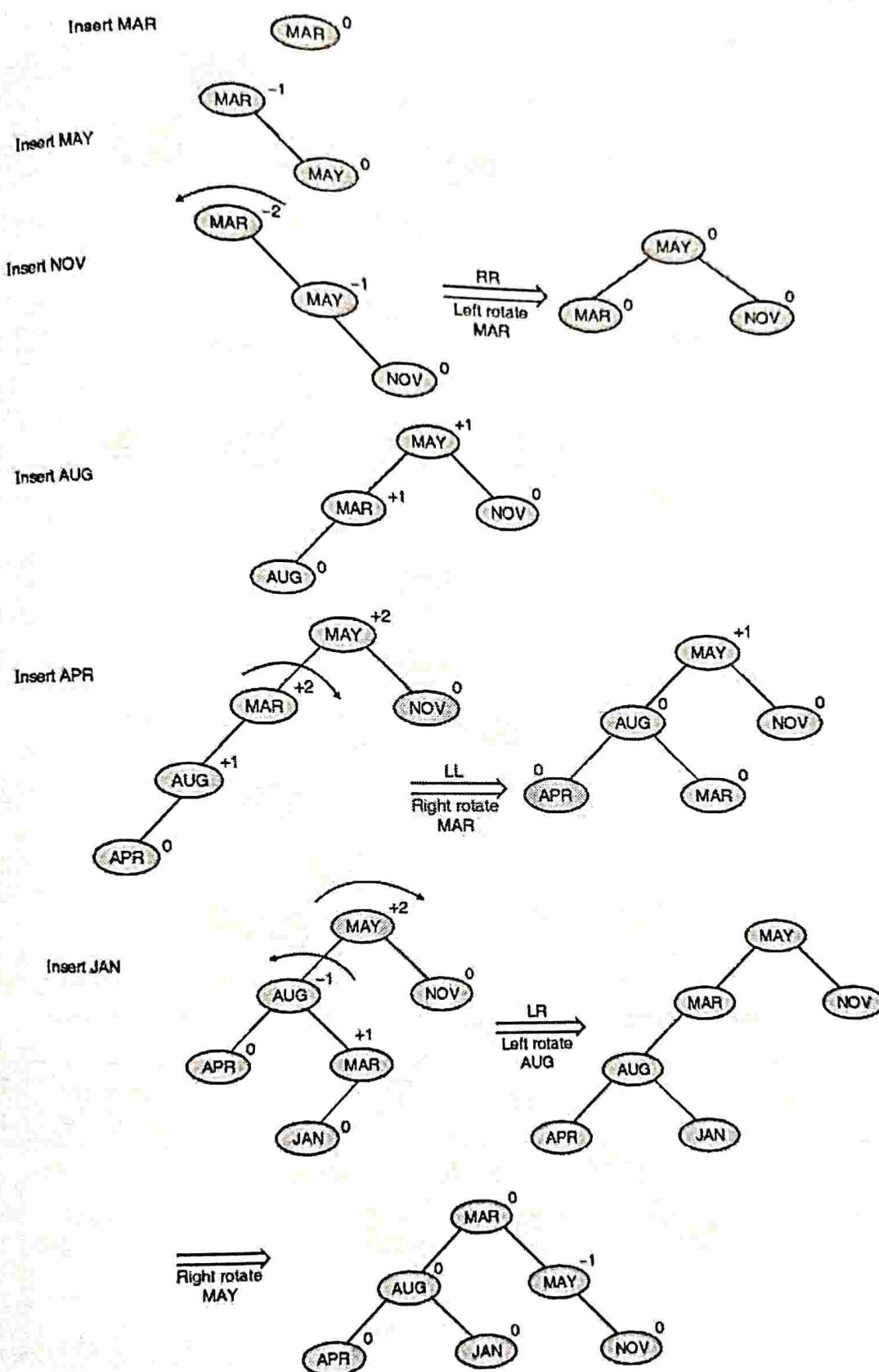
Insert 12 :



Example 4.11.3 : Insert the following sequence of keys into an AVL tree. Find out the type of rotation required in each case.

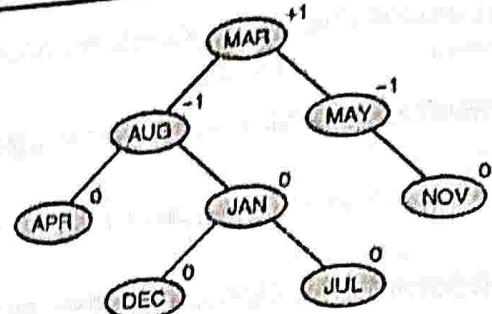
MAR, MAY, NOV, AUG, APR, JAN, DEC, JUL, FEB, JUN,

Solution:

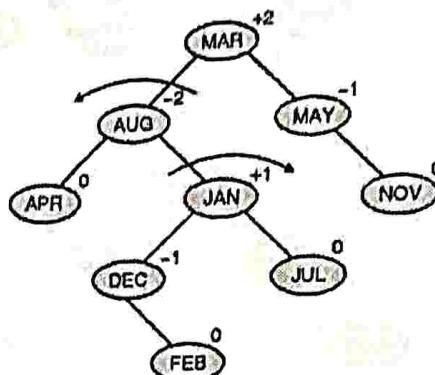




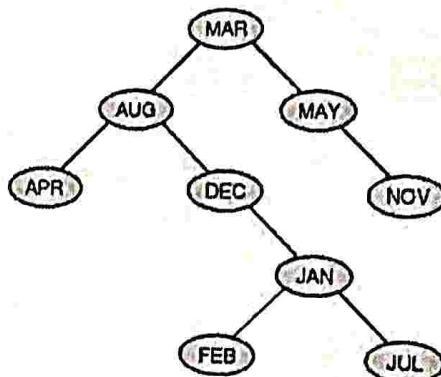
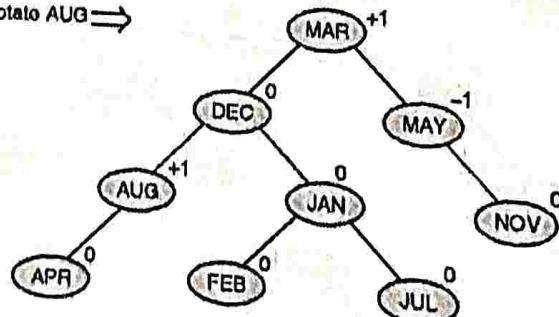
Insert JUL.



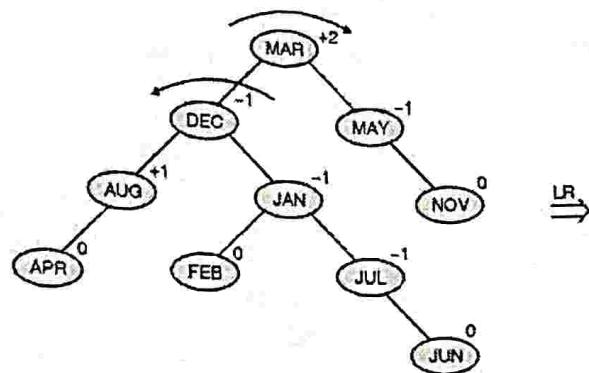
Insert FEB



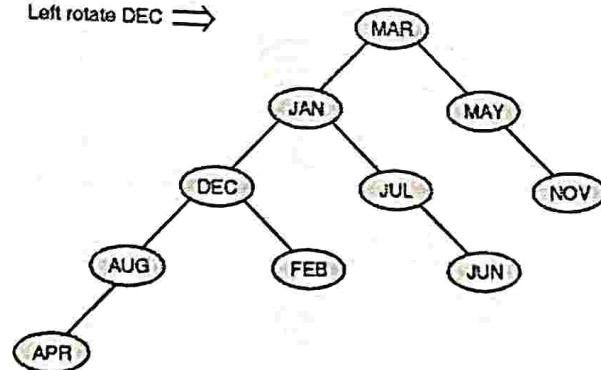
RL

Right rotate JAN \Rightarrow Left rotate AUG \Rightarrow 

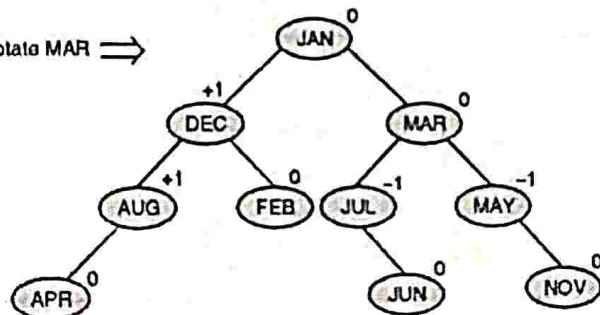
Insert JUN



Left rotate DEC \Rightarrow



Right rotate MAR \Rightarrow



Example 4.11.4 : Create AVL tree for the following data. Show all steps with rotations :

CAR, BAG, ANT, BAT, CAT, ART, APT.

Solution :

| Sr. No. | Data to be inserted | Tree after insertion | Tree after rotation |
|---------|---------------------|----------------------|---------------------|
| 1. | CAR | | |
| 2. | BAG | | |



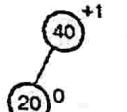
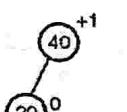
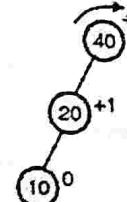
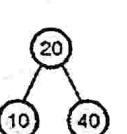
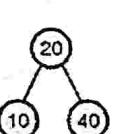
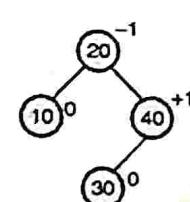
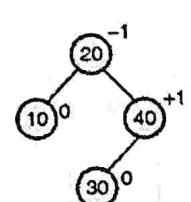
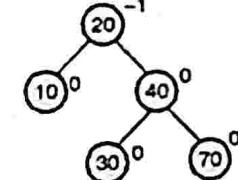
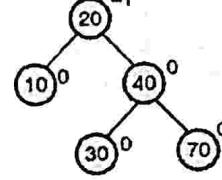
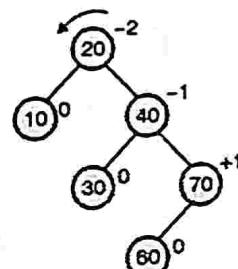
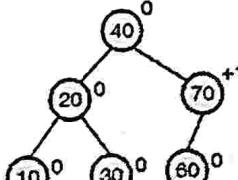
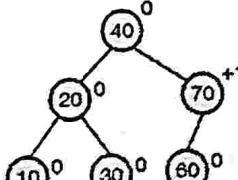
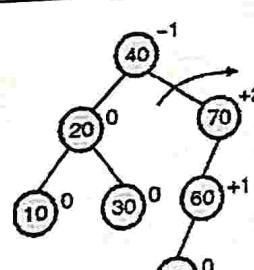
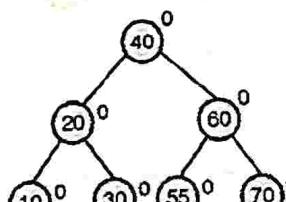
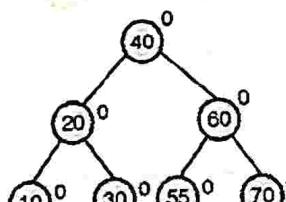
| Sr. No. | Data to be inserted | Tree after insertion | Tree after rotation |
|---------|---------------------|--|---|
| 3. | ANT | AVL tree after inserting ANT. Root: CAR (+2). Left child: BAG (+1). Right child: ANT (0). | AVL tree after LL rotation. Root: BAO (0). Left child: ANT (0). Right child: CAR (0). |
| 4. | BAT | AVL tree after inserting BAT. Root: BAG (-1). Left child: ANT (0). Right child: CAR (+1). BAT is at depth 2. | |
| 5. | CAT | AVL tree after inserting CAT. Root: BAG (-1). Left child: ANT (0). Middle child: CAR (0). Right child: BAT (0). CAT is at depth 2. | |
| 6. | ART | AVL tree after inserting ART. Root: BAG (0). Left child: ANT (-1). Middle child: CAR (0). Right child: BAT (0). ART is at depth 2. | |
| 7. | APT | AVL tree after inserting APT. Root: BAG (+1). Left child: ANT (-2). Middle child: CAR (0). Right child: BAT (0). APT is at depth 2. An RL rotation is shown to balance it into a tree where APT is at depth 1. | AVL tree after RL rotation. Root: BAG (0). Left child: APT (0). Middle child: CAR (0). Right child: BAT (0). APT is at depth 1. |

Example 4.11.5 : Create an AVL tree for the following data :

40 20 10 30 70 60 55

Solution :

| Sr. No. | Data to be inserted | Tree of insertion | Tree after balancing |
|---------|---------------------|--|--|
| 1. | 40 | AVL tree after inserting 40. Root: 40 (0). | AVL tree after inserting 40. Root: 40 (0). |

| Sr. No. | Data to be inserted | Tree of insertion | Tree after balancing |
|---------|---------------------|--|---|
| 2. | 20 |  |  |
| 3. | 10 |  \Rightarrow  |  |
| 4. | 30 |  |  |
| 5. | 70 |  |  |
| 6. | 60 |  \Rightarrow  |  |
| 7. | 55 |  \Rightarrow  |  |



Example 4.11.6 : Insert the following elements in a AVL search tree : 27, 25, 23, 29, 35, 33, 34

Solution :

| Sr. No. | Data | Tree after Insertion | Tree after rotation |
|---------|------|----------------------|---------------------|
| 1. | 27 | | |
| 2. | 25 | | |
| 3. | 23 | | |
| 4. | 29 | | |
| 5. | 35 | | |
| 6. | 33 | | |
| 7. | 34 | | |

Example 4.11.7 : Insert the following elements in AVL tree : 44, 17, 32, 78, 50, 88, 48, 62, 54.
Explain the different rotations that will be used.

MU - Dec. 14, 10 Marks

Solution :

| Sr. No. | Data to be inserted | Tree after insertion | Tree after rotation |
|---------|---------------------|----------------------|---------------------|
| 1. | 44 | | |
| 2. | 17, 32 | LR | |
| 3. | 78 | | |
| 4. | 50 | RL | |
| 5. | 88 | = | = |
| 6. | 48 | | |



| Sr. No. | Data to be inserted | Tree after insertion | Tree after rotation |
|---------|---------------------|--|---------------------|
| 7. | 62 | <pre> graph TD 50((50)) -- "+1" --> 32((32)) 50 -- "-1" --> 78((78)) 32 -- "0" --> 17((17)) 32 -- "-1" --> 44((44)) 78 -- "0" --> 62((62)) 62 -- "0" --> 88((88)) </pre> | |
| 8. | 54 | <pre> graph TD 50((50)) -- "0" --> 32((32)) 50 -- "+1" --> 78((78)) 32 -- "0" --> 17((17)) 32 -- "-1" --> 44((44)) 78 -- "+1" --> 62((62)) 62 -- "0" --> 54((54)) </pre> | |

Example 4.11.8 : Construct AVL tree for the following data 50, 25, 10, 5, 7, 3, 30, 20, 8, 15

MU - May 15, 5 Marks

Solution :

AVL Trees

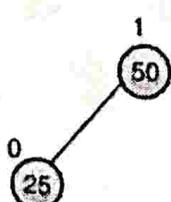
- An AVL (Adelson-Velskii and Landis) tree is a height balance tree. These trees are binary search trees in which the heights of two siblings are not permitted to differ by more than one.
i.e. $| \text{Height of the left subtree} - \text{height of the right subtree} | \leq 1$
- Searching time in a binary search tree is $O(h)$, where h is the height of the tree. For efficient searching, it is necessary that height should be kept to minimum. A full binary search tree with n nodes will have a height of $O(\log_2 n)$. In practice, it is very difficult to control the height of a BST. It lies between $O(n)$ to $O(\log_2 n)$. An AVL tree is a close approximation of full binary search tree.

AVL Tree

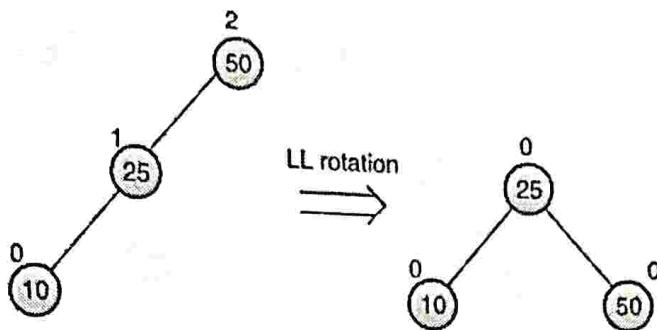
Insert 50



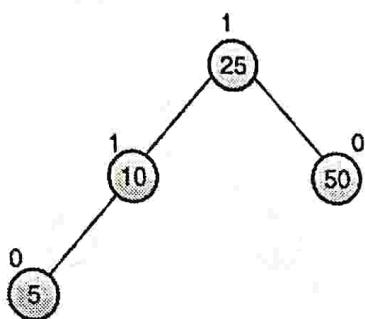
Insert 25



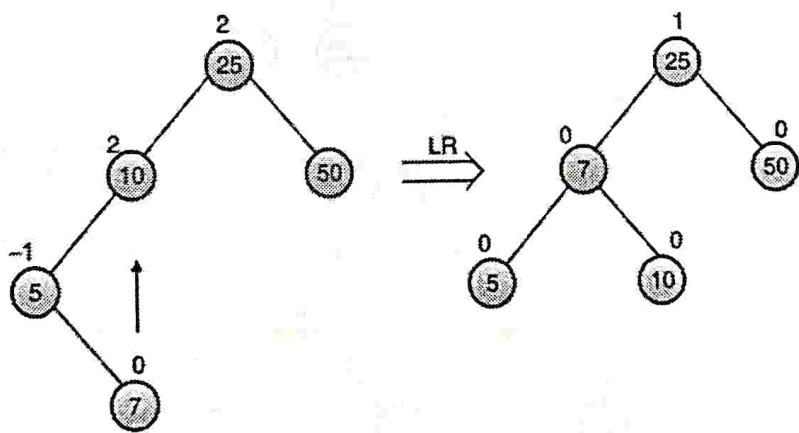
Insert 10



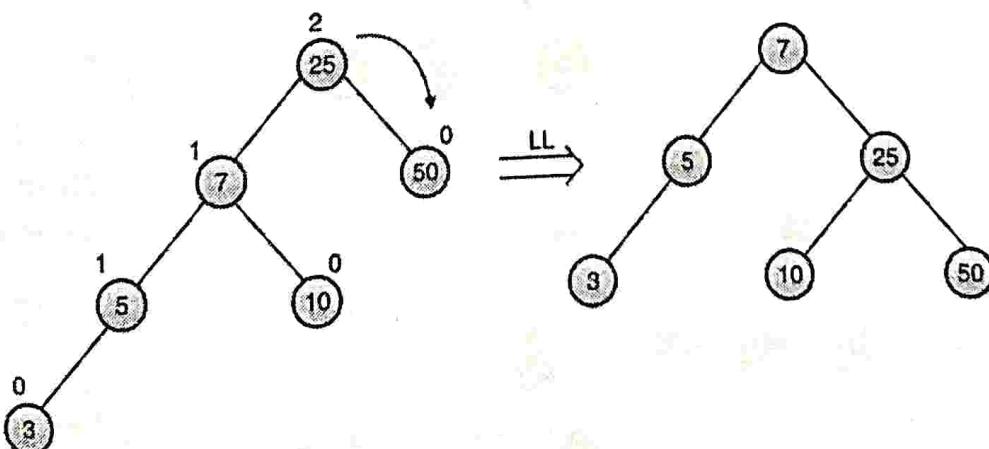
Insert 5



Insert 7

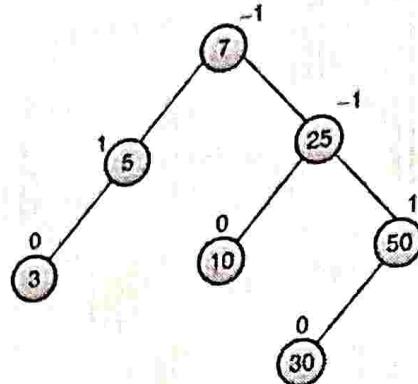


Insert 3

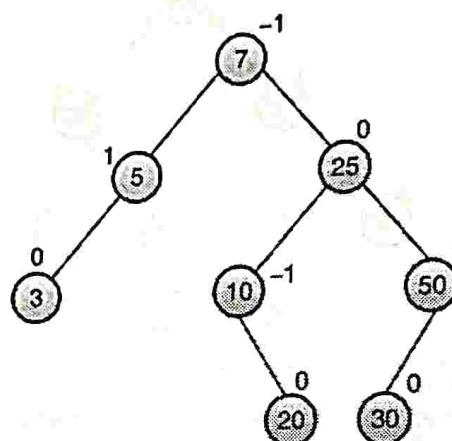




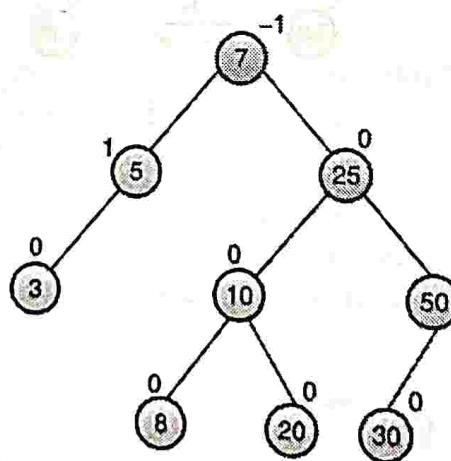
Insert 30



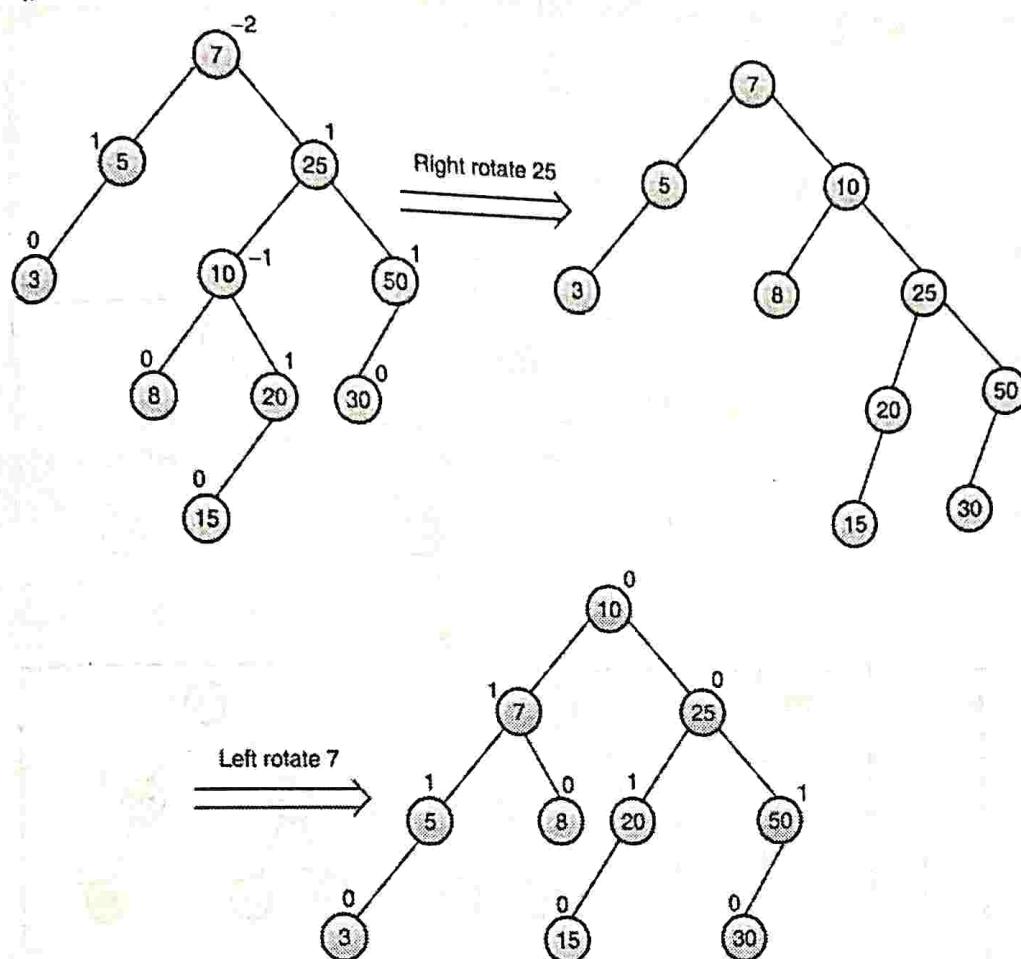
Insert 20



Insert 8



Insert 15



Example 4.11.9 Insert the following elements in a AVL search tree :

40, 23, 32, 84, 55, 88, 46, 71, 57

Explain different rotations used in AVL trees

MU - Dec. 16, 10 Marks

Solution

| Sr. No. | Data to be inserted | Tree after insertion | Tree after rotation |
|---------|---------------------|----------------------|---------------------|
| 1. | 40 | | |
| 2. | 23 | | |
| 3. | 32 | LR | |



| Sr. No. | Data to be Inserted | Tree after Insertion | Tree after rotation |
|---------|---------------------|----------------------|---------------------|
| 4. | 84 | | |
| 5. | 55 | | |
| 6. | 88 | | |
| 7. | 46 | | |
| 8. | 71 | | |

| Sr. No. | Data to be inserted | Tree after insertion | Tree after rotation |
|---------|---------------------|----------------------|---------------------|
| 9. | 57 | | |

Example 4.11.10 : Insert the following elements in a AVL search tree :
 63, 52, 49, 83, 92, 29, 23, 54, 13, 99

MU - May 17, 7 Marks

Solution :

| Sr. No. | Data to be inserted | Tree after insertion | Tree after rotation |
|---------|---------------------|----------------------|---------------------|
| 1. | 63 | | |
| 2. | 52 | | |
| 3. | 49 | | |
| 4. | 83 | | |



| Sr. No. | Data to be inserted | Tree after insertion | Tree after rotation |
|---------|---------------------|--|--|
| 5. | 92 | AVL tree after inserting 92. Root 52 has left child 49 (balance factor -2) and right child 63 (balance factor -2). 63 has left child 83 (balance factor -1) and right child 92 (balance factor 0). A curved arrow labeled "RR" indicates a right rotation around 49, resulting in a balanced tree where 49 is the root, 52 is its right child (balance factor 0), and 83 is its left child (balance factor 0). 83 has left child 63 (balance factor 0) and right child 92 (balance factor 0). | Balanced AVL tree after RR rotation. Root 52 has left child 49 (balance factor 0) and right child 83 (balance factor 0). 49 has left child 63 (balance factor 0) and right child 92 (balance factor 0). |
| 6. | 29 | AVL tree after inserting 29. Root 52 has left child 49 (balance factor 0) and right child 83 (balance factor 0). 49 has left child 29 (balance factor +1) and right child 63 (balance factor 0). 83 has left child 63 (balance factor 0) and right child 92 (balance factor 0). | |
| 7. | 23 | AVL tree after inserting 23. Root 52 has left child 49 (balance factor +1) and right child 83 (balance factor 0). 49 has left child 29 (balance factor +1) and right child 23 (balance factor 0). 83 has left child 63 (balance factor 0) and right child 92 (balance factor 0). A curved arrow labeled "LL" indicates a left rotation around 29, resulting in a balanced tree where 29 is the root, 52 is its left child (balance factor 0), and 83 is its right child (balance factor 0). 83 has left child 63 (balance factor 0) and right child 92 (balance factor 0). | Balanced AVL tree after LL rotation. Root 52 has left child 29 (balance factor 0) and right child 83 (balance factor 0). 29 has left child 23 (balance factor 0) and right child 49 (balance factor 0). 83 has left child 63 (balance factor 0) and right child 92 (balance factor 0). |
| 8. | 54 | AVL tree after inserting 54. Root 52 has left child 29 (balance factor 0) and right child 83 (balance factor +1). 29 has left child 23 (balance factor 0) and right child 49 (balance factor 0). 83 has left child 63 (balance factor +1) and right child 92 (balance factor 0). 63 has left child 54 (balance factor 0). | |
| 9. | 13 | AVL tree after inserting 13. Root 52 has left child 29 (balance factor +1) and right child 83 (balance factor +1). 29 has left child 23 (balance factor 0) and right child 49 (balance factor 0). 83 has left child 63 (balance factor +1) and right child 92 (balance factor 0). 63 has left child 54 (balance factor 0). | |

| Sr. No. | Data to be inserted | Tree after insertion | Tree after rotation |
|---------|---------------------|----------------------|---------------------|
| 10. | 99 | | |

Example 4.11.11 : Insert the following elements in AVL tree: 44, 17, 32, 78, 50, 88, 48, 62, 54. Explain different rotations that can be used.

MU - May 18, Dec. 19, 10 Marks

Solution :

| Sr. No. | Data to be inserted | Tree after insertion | Tree after rotation |
|---------|---------------------|----------------------|---------------------|
| 1. | 44 | | |
| 2. | 17 | | |
| 3. | 32 | | |
| 4. | 78 | | |
| 5. | 50 | | |



| Sr. No. | Data to be inserted | Tree after insertion | Tree after rotation |
|---------|---------------------|----------------------|---------------------|
| 6. | 88 | RR | |
| 7. | 48 | | |
| 8. | 62 | | |
| 9. | 54 | | |

4.11.5(D) 'C' Function for Insertion of an Element into an AVL Tree

```

typedef struct node
{
    int data;
    struct node *left, *right;
    int ht;
}node;
node *insert(node *T, int x)
{
    if(T == NULL)
    {
        T = (node*)malloc(sizeof(node));
        T->data = x;
        T->left = NULL;
        T->right = NULL;
    }
    else
        if(x > T->data) // insert in right subtree
    {
        T->right = insert(T->right, x);
        if((BF(T) == -2)
            if(x > T->right->data)
                T = RR(T);
            else
                T = RL(T);
        }
        else
            if(x < T->data)
            {
                T->left = insert(T->left, x);
                if(BF(T) == 2)
                    if(x < T->left->data)
                        T = LL(T);
                    else
                        T = LR(T);
            }
        T->ht = height(T);
        return(T);
    }
}

```

4.11.5(E) 'C' Function to Find Height of AVL Tree

```

int height(node *T)
{
    int lh, rh;
    if(T->left == NULL)
        lh = 0;
    else
        lh = 1 + T->left->height;
    if(T->right == NULL)
        rh = 0;
    else
        rh = 1 + T->right->height;
    if(lh>rh)
        return(lh);
    return(rh);
}

```

4.11.5(F) 'C' Function to Rotate Right

```

node *rotateright(node *x)
{
    node *y;
    y = x->left;
    x->left = y->right;
    y->right = x;
    x->ht = height(x);
    y->ht = height(y);
    return(y);
}

```

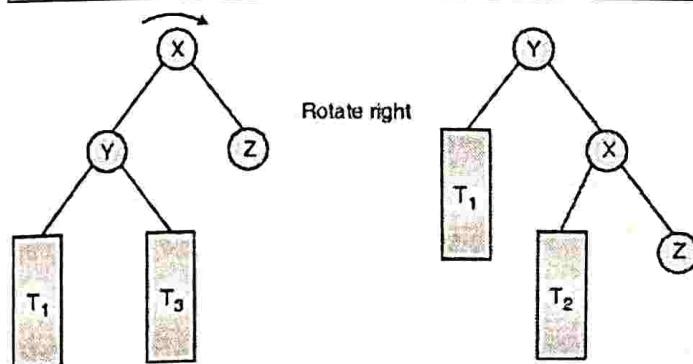


Fig. 4.11.11

4.11.5(G) 'C' Function to Rotate Left

```
node *rotateleft(node *x)
{
    node *y;
    y = x->right;
    x->right = y->left;
    y->left = x;
    x->ht = height(x);
    y->ht = height(y);
    return(y);
}
```

4.11.5(H) 'C' Function for RR

```
node *RR(node *T)
{
    T = rotateleft(T);
    return(T);
}
```

4.11.5(I) 'C' Function for LL

```
node *LL(node *T)
{
    T = rotateright(T);
    return(T);
}
```

4.11.5(J) 'C' Function for LR

```
node *LR(node *T)
{
    T->left = rotateleft(T->left);
    T = rotateright(T);
    return(T);
}
```

4.11.5(K) 'C' Function for RL

```
node *RL(node *T)
{
    T->right = rotateright(T->right);
    T = rotateleft(T);
}
```

```
return(T);
}
```

4.12 Application of Trees

MU - May 14, May 16, Dec. 16, May 17

University Questions

- Q. Discuss practical application of trees.
 (May 14, Dec. 16, May 17, 5 Marks)
- Q. Describe expression Tree with an example.
 (May 16, 5 Marks)

4.12.1 Expression Trees

MU - Dec. 17, Dec. 18, May 19, Dec. 19

University Questions

- Q. Write short note on Expression trees.
 (Dec. 17, May 19, 10 Marks)
- Q. What are expression trees ? What are its advantages ?
 (Dec. 18, Dec. 19, 3 Marks)

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. 4.12.1 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.

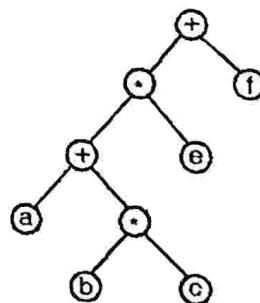


Fig. 4.12.1

Prefix (expression tree of Fig. 4.12.1) = + * + a * b c e f
 Postfix (expression tree of Fig. 4.12.1) = 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 pointers 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.

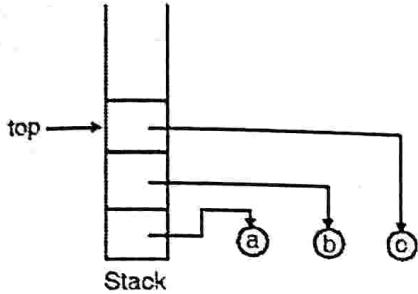


Fig. 4.12.2

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. 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.

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.

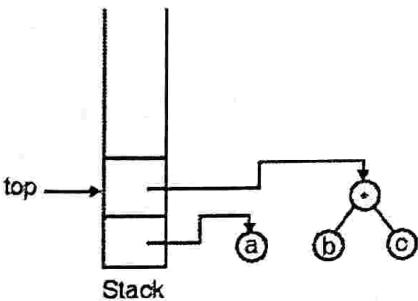


Fig. 4.12.3

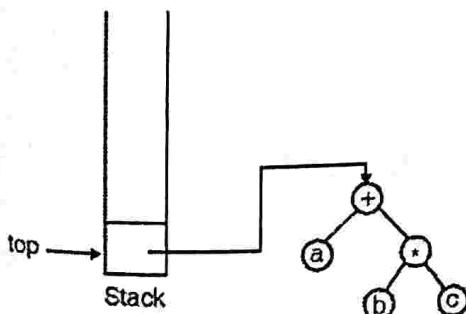


Fig. 4.12.4

Next, an '=' is read, one node tree is created and a pointer to it is pushed onto the stack.

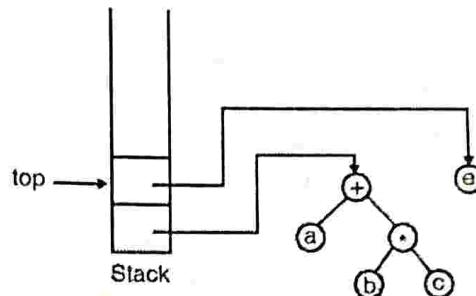


Fig. 4.12.5

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.

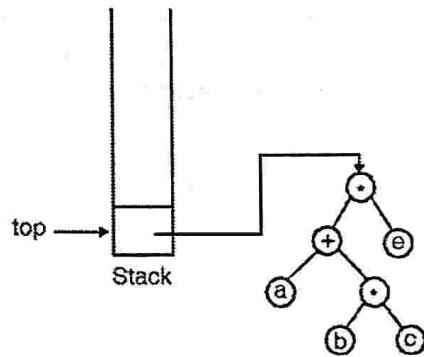


Fig. 4.12.6

Continuing, a '+' is read, a one node tree is created and a pointer to it is pushed onto the stack.

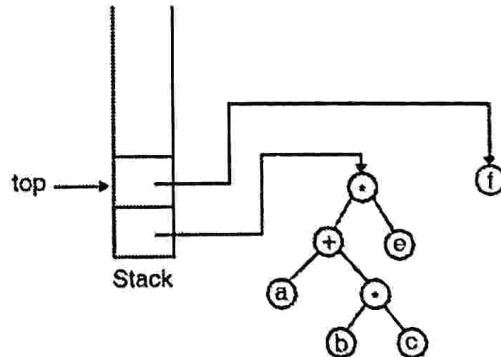


Fig. 4.12.7

Finally, a '+' is read, two trees are merged, and a pointer to the final tree is pushed on the stack.

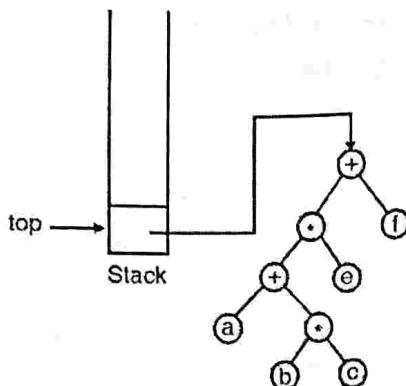


Fig. 4.12.8

4.12.2 Program on Expression Tree from Postfix Expression

Program 4.12.1 : Program for creating an expression tree from a postfix expression and printing its preorder and inorder traversal sequence

```
#include<conio.h>
#include<stdio.h>
typedef struct node
{
    char data;
    struct node *left, *right;
}node;
typedef struct stack
{
    node *data[30];
    int top;
}stack;
void push(stack *s, node *x)
{
    s->top = s->top + 1;
    s->data[s->top] = x;
}
node *pop(stack *s)
{
    node *x;
    x = s->data[s->top];
    s->top = s->top - 1;
    return(x);
}
void preorder(node *T)
{
    if(T!=NULL)
```

```
    printf("%c", T->data);
    preorder(T->left);
    preorder(T->right);
}

void inorder(node *T)
{
    if(T!=NULL)
        inorder(T->left);
    printf("%c", T->data);
    inorder(T->right);
}

void main()
{
    char c;
    stack s;
    node *top, *t1, *t2;
    top = NULL;
    s.top = -1;
    while((c = getchar())!= '\n ')
    {
        if(isalpha(c))
        {
            top = (node*)malloc(sizeof(node));
            top->left = NULL;
            top->right = NULL;
            top->data = c;
            push(&s, top);
        }
        else
        {
            t2 = pop(&s);
            t1 = pop(&s);
            top = (node*)malloc(sizeof(node));
            top->data = c;
            top->left = t1;
            top->right = t2;
            push(&s, top);
        }
    }
}
```

```

printf("\n preorder/prefix :");
preorder(top);
printf("\n inorder/infix :");
inorder(top);
}

```

Output

AB*C+
preorder/prefix : + * ABC
inorder/infix : A * B + C

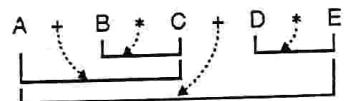
4.12.3 Conversion of an Expression into Binary Tree

Example 4.12.1 : Construct the binary tree for the following expression.

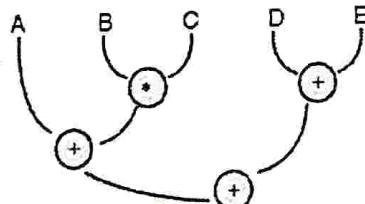
$$A + B * C + D * E$$

Solution :

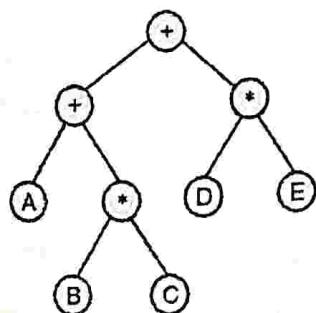
Step 1: Group elements as per the sequence of evaluation. [This step is similar to fully parenthesizing an expression].



Step 2 : Move the operator at the center of the group.



Step 3 : Invert the structure.

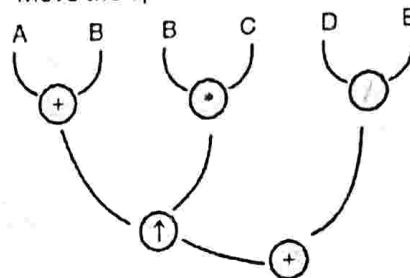


Example 4.12.2 : Construct the binary tree for the following expression : $(A + B) \uparrow (B * C) + D / E$

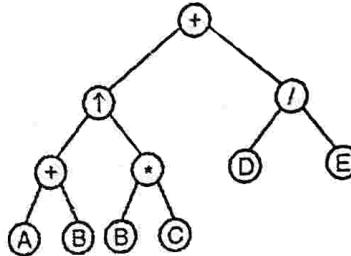
Solution :

Step 1 : Group elements as per the sequence of evaluation. [This step is similar to fully parenthesizing an expression].

Step 2 : Move the operator at the center of the group.



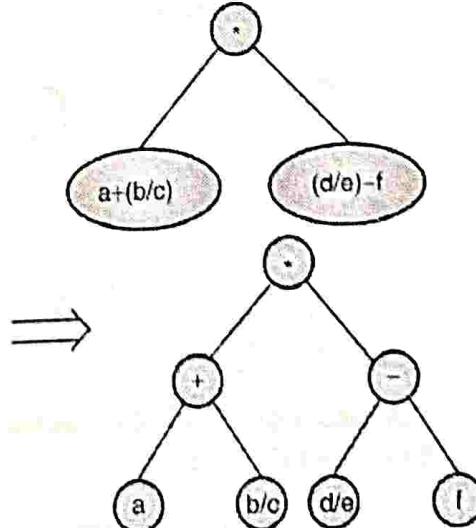
Step 3 : Inverting the structure, we get

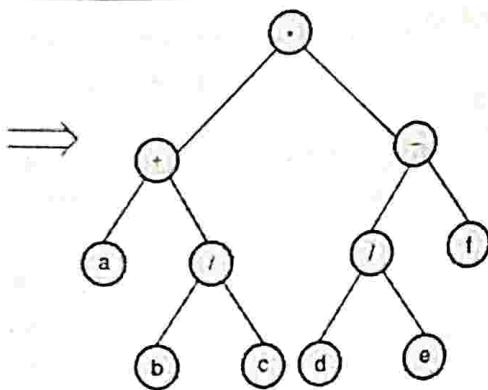


Example 4.12.3 : Derive the expression tree for the following algebraic expression : $(a + (b/c))^* (cd/e) - f$ **MU - Dec. 18, 5 Marks**

Solution :

Expression tree for $(a + (b/c))^* (cd/e) - f$ is shown in following



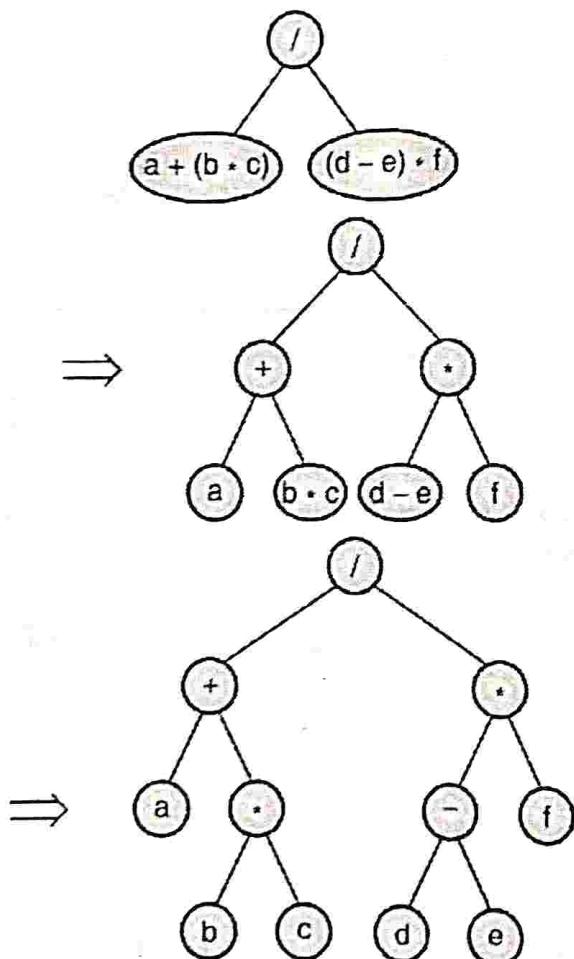


Example 4.12.4 : Derive an expression tree for :

$$(a + (b * c)) / ((d - e) * f)$$

MU - Dec. 19, 4 Marks

Solution : Expression tree for $(a + (b * c)) / ((d - e) * f)$ is shown in following



4.12.4 Construction of an Expression Tree from Infix Expression

Algorithm : Structure of a tree node is given below :

```
typedef struct node
{ char data;
```

```
struct node *left, *right;
} node;
```

A stack of nodes is being used in the algorithm. Stack is being defined by the following structure.

```
typedef struct stack
```

```
{ node *data[20];
```

```
int top;
```

```
} stack;
```

```
stack S1, S2;
```

```
while(! end of input)
```

```
{
```

```
    x = next input;
```

```
    P ← a new node; [P is a pointer]
```

```
    P → data = x;
```

set the left and right pointer of node to NULL i.e. P

```
→ left = P → right = NULL;
```

```
if(x = 'C')
```

```
    push P into S1
```

```
else
```

```
if(x is an alphanumeric)
```

```
push P into S2
```

```
else
```

```
if(x is an operator)
```

```
{
```

```
    while(precedence(x) ≤ precedence(top(S1)))
```

```
    { P = POP(S1)
```

```
        P → right = POP(S2)
```

```
        P → left = POP(S2)
```

```
        push(S2, P)
```

two trees have been popped from S2 and merged into a single tree with the root popped from S1

```
}
```

```
push P into S1
```

```
}
```

```
while(! empty(S1))
```

```
{
```

```
    P = POP(S1);
```

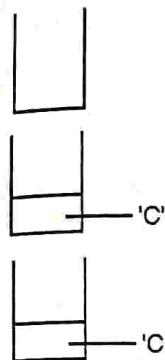
```
    P → right = POP(S2);
```

```
    P → left = POP(S2);
```

```
    push(S2, P);
```

Simulation of algorithm for $(a + b) * c / d \wedge e$

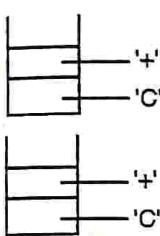
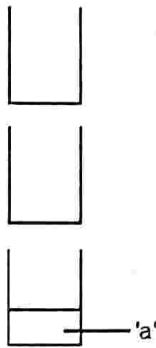
Operator's stack (say S1)



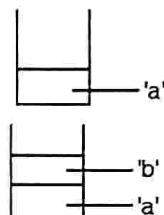
Input expression

- Initially
 $(a + b) * c / d \wedge e$
 push 'c' into S1
 $a + b) * c / d \wedge e$
 push a into S2
 $+ b) * c / d \wedge e$
 push '+' into S1

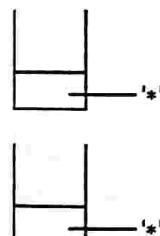
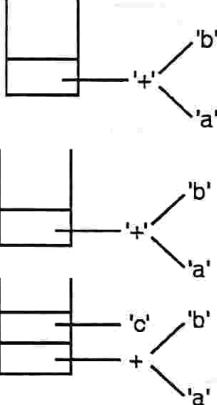
Output stack (say S2)



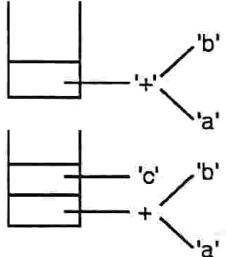
$b) * c / d \wedge e$
 push 'b' into S2



POP '+' and 'c' from the S1. Merge '+' and 'a' into a single tree in S2.
 $*c / d \wedge e$



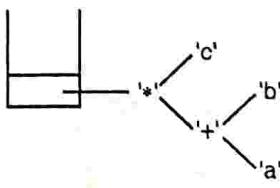
Push '*' into S1
 $c / d \wedge e$
 push 'c' into S2
 $/ d \wedge e$



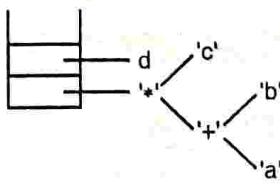
POP '*' from S1. POP two trees from S2 and merge, then into a single tree with '*' as the root. Push '/' into S1.



$d \wedge e$ push d into S2



$\wedge e$





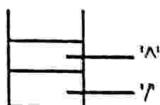
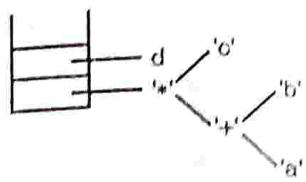
Operator's stack (say S1)



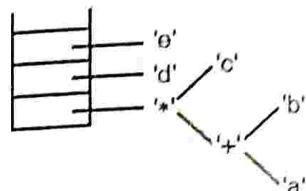
Input expression

push '^' into S1
e

Output stack (say S2)



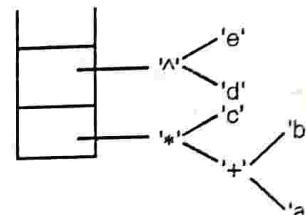
push 'e' into S2
end of input



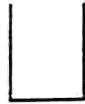
POP '^' from S1. POP two trees from S2 and
merge them into a single tree with '^' as
the root.



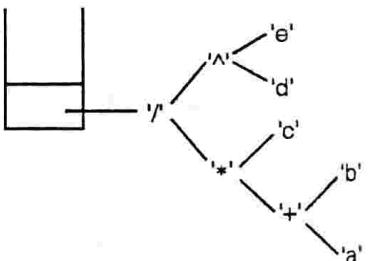
End of input



POP '/' from S1. POP two trees from S2 and
merge them into a single tree with '/' as
the root.



End of input



4.13 Huffman Algorithm

4.13.1 Huffman Codes

MU - Dec. 13, Dec. 15, May 15, Dec. 16, May 17,
Dec. 17, May 18, May 19

University Questions

- Q. What is Huffman coding ?
(Dec. 13, May 17, 5 Marks)
- Q. Write a function to implement an Huffman coding given a symbol and its frequency.
(Dec. 15, 10 Marks)

- Q. Explain Huffman Algorithm with an example.
(May 15, 5 Marks)
- Q. Give practical application of tree.
(Dec. 16, May 17, 5 Marks)
- Q. Write short note on : Huffman coding
(Dec. 16, 4 Marks)
- Q. Explain Huffman Encoding with suitable example.
(Dec. 17, May 18, May 19, 10 Marks)

- A text message can be converted into a sequence of 0's and 1's by replacing each character of the message with its code.

Table 4.13.1 : Binary code

| Input symbol | Code |
|--------------|------|
| a | 000 |
| b | 001 |
| c | 010 |
| d | 011 |
| e | 100 |

- An input message bbca can be encoded as 001001010000 using the codes given in the Table 4.13.1.
- Huffman coding technique is based on variable length code size. All character in a file, do not appear with the same frequency. There could be a large disparity

between the most frequent and least frequent characters. A large data file may have large amount of digits, blanks and etc. but very few alphabets. A simple strategy of providing **short codes** to most frequently occurring characters can considerably reduce the size of the encoded message.

| Symbol | Frequency | Code1 | Code2 |
|--------|-----------|-------|-------|
| a | .12 | 000 | 000 |
| b | .40 | 001 | 11 |
| c | .15 | 010 | 01 |
| d | .08 | 011 | 001 |
| e | .25 | 100 | 10 |

Fig. 4.13.1 : Two binary codes

Message → b c b e b c

Encoded using code 1 → 001010001100001010 (length = 18)

Encoded using code 2 → 110111101101 (length = 12)

- Encoding of the message using code1 (as given in Fig. 4.13.1) requires 18 bits of storage space, whereas if code 2 is used for encoding, only 12 bits will be required. Thus, there is a saving of 33% of storage space, if code 2 is used for encoding of the message "bcbebc".
- Huffman code has prefix property. Prefix property allows one to decode a string of 0's and 1's by repeatedly deleting prefixes of the string that are codes for characters.

Example : Decoding 1101 11 101101 using code 2.

(a) Prefix 11 encoded as b and deleted from the message

| Input | Output |
|---------------|--------------------------|
| 01 111 011 01 | b |
| 11101101 | bc[01(c) is deleted] |
| 101101 | bcb[11(b) is deleted] |
| 1101 | bcbe[10(e) is deleted] |
| 01 | bcbeb[11(b) is deleted] |
| - | bcbebc[01(c) is deleted] |

Fig. 4.13.2 : Decoding using code 2[Prefix property]

4.13.2 Representation of Binary Codes as a Binary Tree

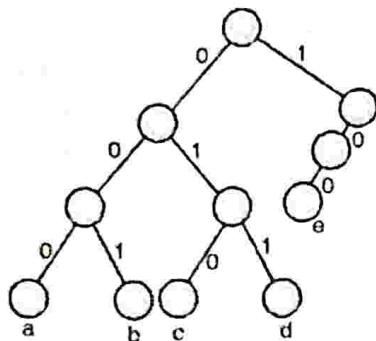
MU - Dec. 15.

University Question

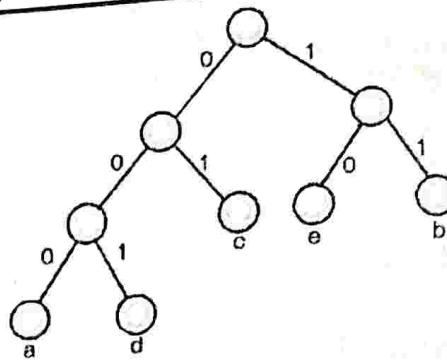
Q. Write a function to implement an Huffman coding given a symbol and its frequency.

(Dec. 15, 10 Marks)

- Tree in Fig. 4.13.3 has data only at leaves. Code of each character can be found by starting from the root of the tree and recording the path. 0 indicates a left branch and 1 indicates a right branch. For instance 'd' using code 2 is reached by going left, then left and finally right. This is encodes as 001.
- If the characters are placed only at the leaves, any sequence of bits can be decoded unambiguously (Prefix property).



Binary tree for code 1



Binary tree for code 2

Fig. 4.13.3 : Binary tree representation of codes with prefix property

4.13.3 Huffman's Algorithm

MU - May 15, Dec. 15

University Questions

- Q. Write a function to implement of HUFFMAN coding given a symbol and it's frequency. (Dec. 15, 10 Marks)
- Q. Explain Huffman's algorithm with an example. (May 15, 5 Marks)

- It is an algorithm for finding the best possible prefix code for given frequency of occurrences of input alphabets in any message.
- Maintain a forest of trees. Each character is converted into a single node tree. Each node is labelled by its frequency (Probability).
- The weight of a tree is equal to the sum of the frequencies of its leaves.
- Select the two trees in the forest that have smallest weights. Combine these two trees into one weight of the combined tree = sum of the weights of the two trees.

This process continues until one tree remains.

Example 4.13.1 : Suppose characters a, b, c, d, e, f have probabilities .07, .09, .12, .22, .23, .27 respectively. Find an optional Huffman code and draw the Huffman tree. What is the average code length.

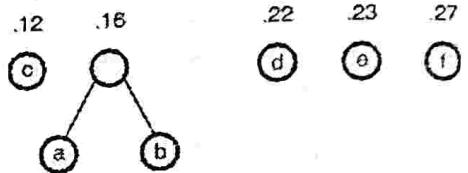
MU - Dec. 19, 8 Marks

Solution :

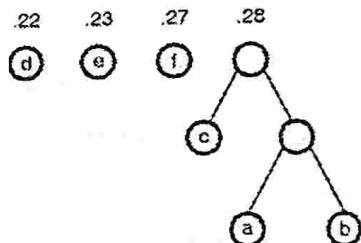
Step 1 : Each node is represented as a tree

| | | | | | |
|-----|-----|-----|-----|-----|-----|
| .07 | .09 | .12 | .22 | .23 | .27 |
| (a) | (b) | (c) | (d) | (e) | (f) |

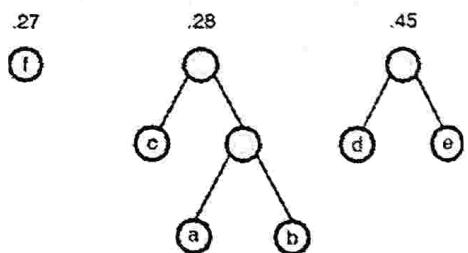
Step 2 : Merge a, b [Trees of minimum weights]



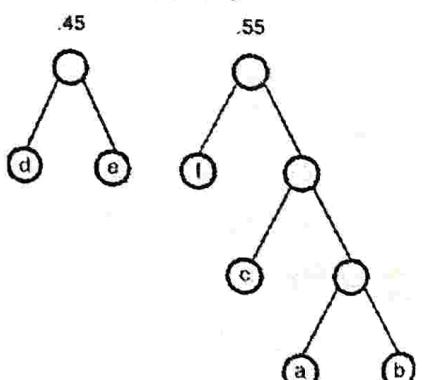
Step 3 : Merge c and (a, b)



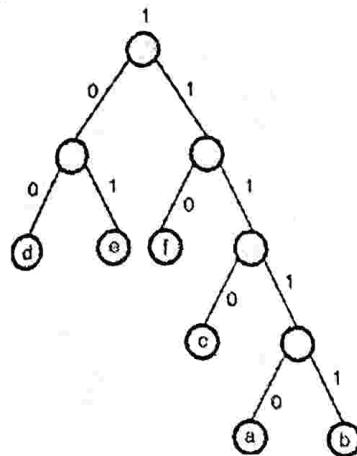
Step 4 : Merge d and e



Step 5 : Merge f and (a, b, c)



Step 6 : Merge (d e) and (f c a b)



| Symbols | Probability | Huffman code |
|---------|-------------|--------------|
| a | 0.07 | 1110 |
| b | 0.09 | 1111 |
| c | 0.12 | 110 |
| d | 0.22 | 00 |
| e | 0.23 | 01 |
| f | 0.27 | 10 |

Average code length

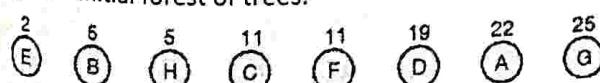
$$\begin{aligned}
 &= 0.07 \times 4 + 0.09 \times 4 + 0.12 \times 3 + 0.22 \times \\
 &\quad 2 + 0.23 \times 2 + 0.27 \times 2 \\
 &= 0.64 + 0.36 + 1.44 = 2.44 \text{ bits.}
 \end{aligned}$$

Example 4.13.2 : For the given data, build the Huffman's tree and explain each step separately.

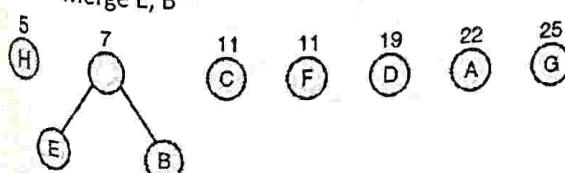
| Data Item | Weight |
|-----------|--------|
| A | 22 |
| B | 5 |
| C | 11 |
| D | 19 |
| E | 2 |
| F | 11 |
| G | 25 |
| H | 5 |

Solution :

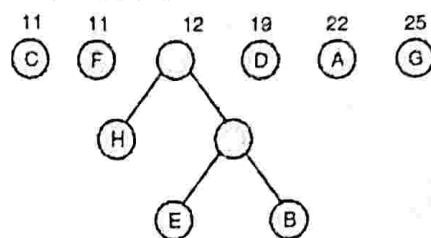
Step 1 : Initial forest of trees.



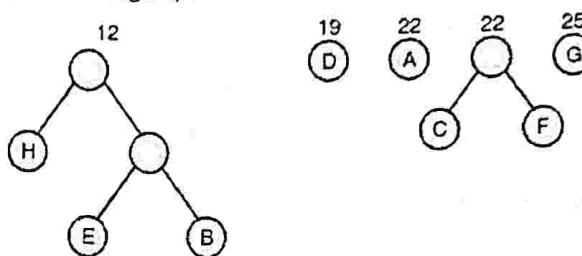
Step 2 : Merge E, B



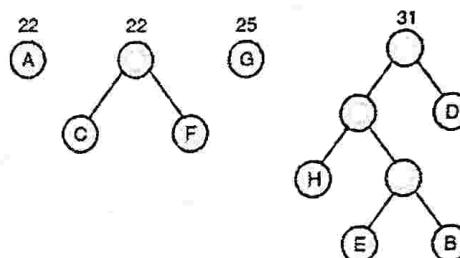
Step 3 : Merge H, (E, B)



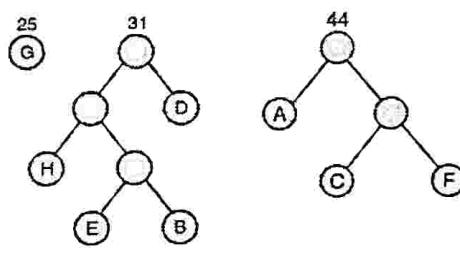
Step 4 : Merge C, F



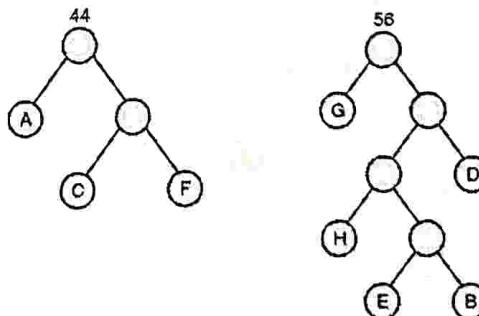
Step 5 : Merge (H, E, B) and D



Step 6 : Merge A, (C, F)

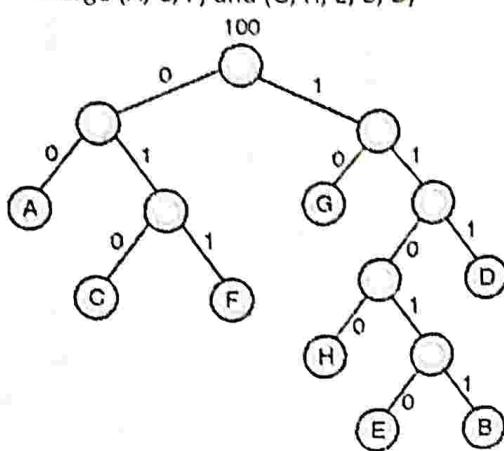


Step 7 : Merge G, (H, E, B, D)





Step 8 : Merge (A, C, F) and (G, H, E, B, D)



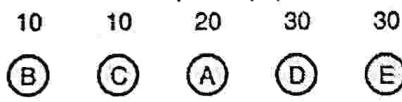
Example 4.13.3 : Construct the Huffman Tree and determine the code for the following characters whose frequencies are as given :

| Characters | A | B | C | D | E |
|------------|----|----|----|----|----|
| Frequency | 20 | 10 | 10 | 30 | 30 |

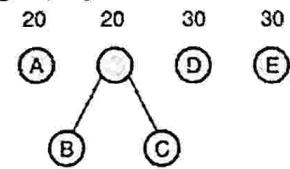
MU - Dec. 13, 5 Marks.

Solution : Construction of Huffman tree :

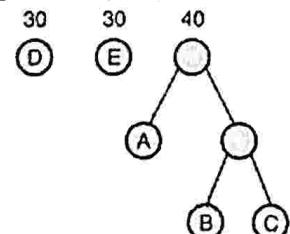
Step 1 : Each node is represented as a tree. These trees are stored in a priority queue.



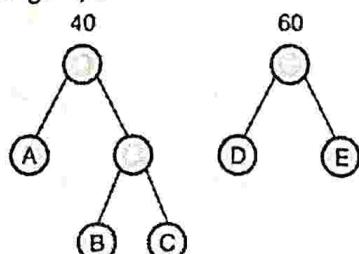
Step 2 : Merge B, C [Trees of minimum weights]



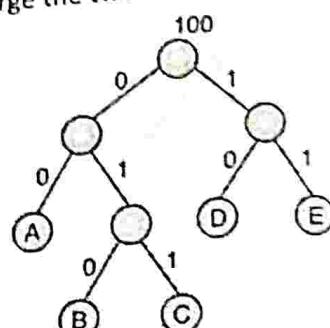
Step 3 : Merge A and (B, C)



Step 4 : Merge D, E



Step 5 : Merge the two trees



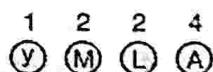
| Data Item | Code |
|-----------|------|
| A | 00 |
| B | 010 |
| C | 011 |
| D | 10 |
| E | 11 |

Example 4.13.4 : Apply Huffman Coding for the word 'MALAYALAM'. Give the Huffman code for each symbol. MU - May 16, 8 Marks

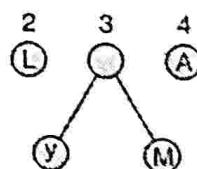
solution :

| Data | Weight |
|------|--------|
| M | 2 |
| A | 4 |
| L | 2 |
| Y | 1 |

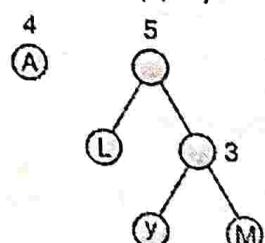
Step 1 : Initial forest of trees



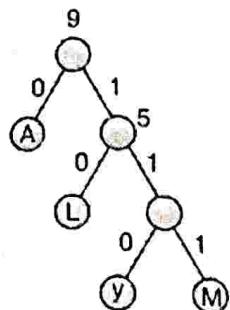
Step 2 : Merging Y and M



Step 3 : Merging L and (Y, M)



Step 4 : Merging A and (L, Y, M)



Step 5 :

| Data | Huffman Code |
|------|--------------|
| A | 0 |
| L | 10 |
| Y | 110 |
| M | 111 |

Ex. 4.13.5 : Construct the Huffman Tree and determine the code for each symbol in the sentence "ENGINEERING" MU - May 17, 8 Marks

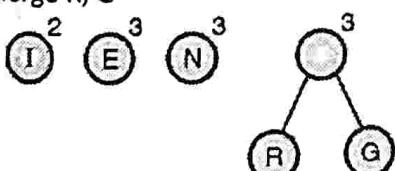
Solution : Frequency of each letter in "ENGINEERING"

| Symbol | Frequency |
|--------|-----------|
| E | 3 |
| N | 3 |
| G | 2 |
| I | 2 |
| R | 1 |

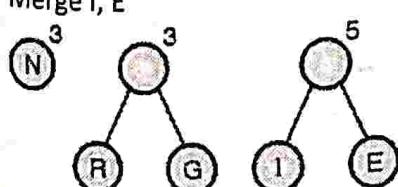
Step 1 : Each node is represented as a tree. These trees are stored in a priority queue.



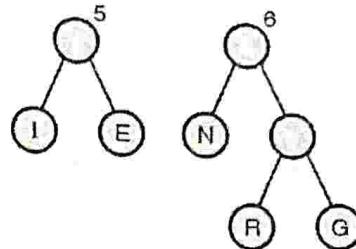
Step 2 : Merge R, G



Step 3 : Merge I, E



Step 4 : Merge N, (R, G)



Step 5 : Merge the two trees

Code for each symbol

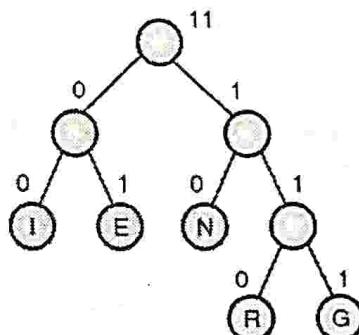
I - 00

E - 01

N - 10

R - 110

G - 111



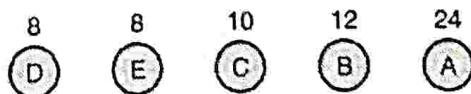
Example 4.13.6 : Given the frequency for the following symbols, compute the Huffman code for each symbol.

| Symbol | A | B | C | D | E |
|-----------|----|----|----|---|---|
| Frequency | 24 | 12 | 10 | 8 | 8 |

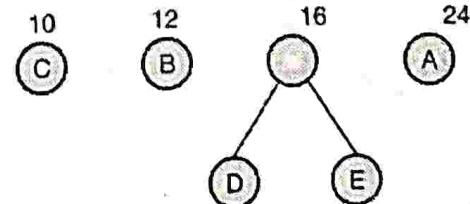
MU – Dec. 18, 10 Marks

Solution :

Step 1 : Each node is represented as a tree. These trees are stored in a priority queue.

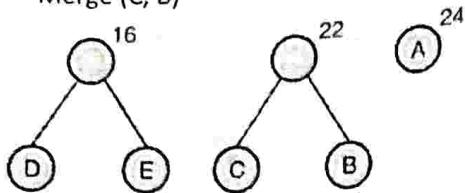


Step 2 : Merge (D, E)

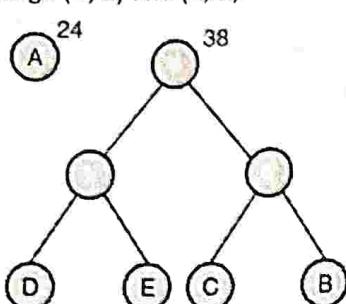




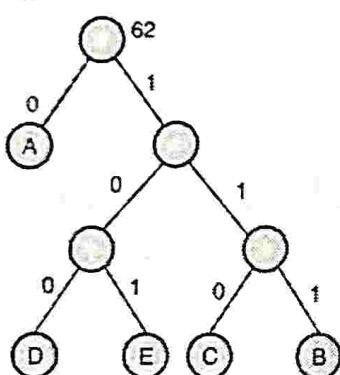
Step 3 : Merge (C, B)



Step 4 : Merge (D, E) and (C, B)



Step 5 : Merge the last two trees



Huffman Code :

A-0
B-111
C-110
D-100
E-101

4.13.4 Program for Huffman Tree

Program 4.13.1 : Program for creation of Huffman tree

- (1) Trees are maintained in priority linked list, ordered by weight.
- (2) Function insert() is used for inserting a tree in priority linked list.

```
#include<conio.h>
#include<stdlib.h>
#include<stdio.h>
// Structure of tree node
typedef struct treenode
```

```
{
    float freq;
    char data;
    struct treenode *left, *right;
}treenode;
/* Structure of node of linked list */
typedef struct node
{
    treenode *data; //Address of tree
    struct node *next;
}node;
node *insert(node *, treenode *);
void main()
{
    treenode *p, *t1, *t2;
    node *head;
    int n, i;
    char x;
    float probability;
    head = NULL; //Empty linked list
    printf("\n Enter No. of alphabets :");
    scanf("%d", &n);
    for(i = 0; i < n; i++)
    {
        flushall();
        printf("\n Enter alphabet :");
        scanf("%c", &x);
        flushall();
        printf("\n Enter frequency :");
        scanf("%f", &probability);
        /* create a new tree and insert it in
         the priority linked list */
        p = (treenode*)malloc(sizeof(treenode));
        p->left = p->right = NULL;
        p->data = x;
        p->freq = probability;
        head = insert(head, p);
    }
    /* create the final tree by merging of two trees of
     smaller weights (n-1) merges will be required*/
    for(i = 1; i < n; i++)
    {
        t1 = head->data; //first tree
        t2 = head->next->data; //second tree
        head = head->next->next;
    }
}
```

```

/*Remove first 2 trees from linked list*/
/*Merge t1 and t2 with new tree in P */
p = (treenode *)malloc(sizeof(treenode));
p->left = t1;
p->right = t2;
p->freq = t1->freq + t2->freq;
head = insert(head, p);
}

node *insert(node *head, treenode *t)
{
    node *p, *q;
    p = (node *)malloc(sizeof(node));
    p->data = t;
    p->next = NULL;
    if(head == NULL) //Empty linked list
        return(p);
    if(t->freq < head->data->freq)
    {
        p->next = head;
        return(p);
    }
    // Locate the point of insertion
    q = head;
    while(q->next != NULL &&
        t->freq > q->next->data->freq)
    {
        q = q->next;
        p->next = q->next;
        q->next = p;
    }
    return(head);
}

```

4.14 B-Trees

MU - Dec. 14, Dec. 17, Dec. 19

University Questions

Q. What is Multi-way search Tree ? Explain with an example. (Dec. 14, 5 Marks)

Q. Explain B-Tree. (Dec. 17, Dec. 19, 5 Marks)

B-tree is another very popular search tree. The node in a binary tree like AVL tree contains only one record. AVL tree is commonly stored in primary memory. In database application, where huge volume of data is handled, the search tree cannot be accommodated in primary memory. B-trees are primarily meant for secondary storage.

- A B-tree is a M-way tree. An M-way tree can have maximum of M children.

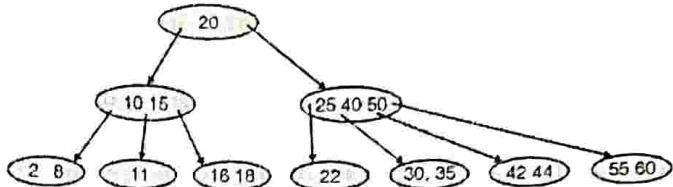


Fig. 4.14.1 : An example of 4-way tree

- An M-way tree contains multiple keys in a node. This leads to reduction in overall height of the tree. If a node of M-way tree holds K number of keys then it will have $K + 1$ children.

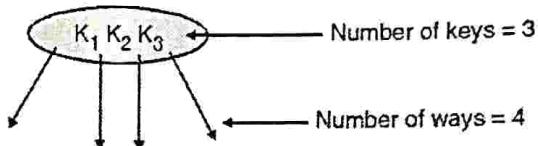
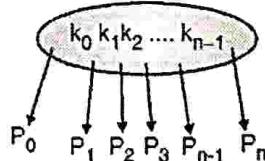


Fig. 4.14.2 : An M-way tree with 3 keys and 4 children

Definition :

A B-tree of order M is a M-way search tree with the following properties :

1. The root can have 1 to $M - 1$ keys.
2. All nodes (except the root) have between $[(M - 1)/2]$ and $M - 1$ keys.
3. All leaves are at the same depth.
4. If a node has t number of children then it must have $(t - 1)$ number of keys.
5. Keys of a node are stored in ascending order.



6. $K_0, K_1, K_2 \dots K_{n-1}$ are the keys stored in the node. Subtrees are pointed by $P_0, P_1 \dots P_n$.

then $K_0 \geq$ all keys of the subtree P_0

$K_1 \geq$ all keys of the subtree P_1

:

:

$K_{n-1} \geq$ all keys of the subtree P_{n-1}

$K_{n-1} <$ all keys of the subtree P_n .



- An example of B-tree of order 4 is shown in Fig. 4.14.3.

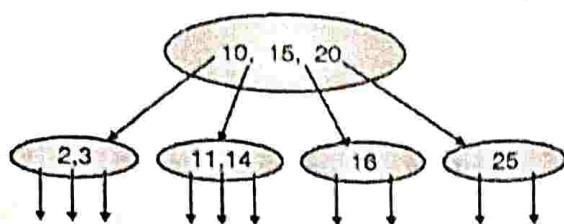
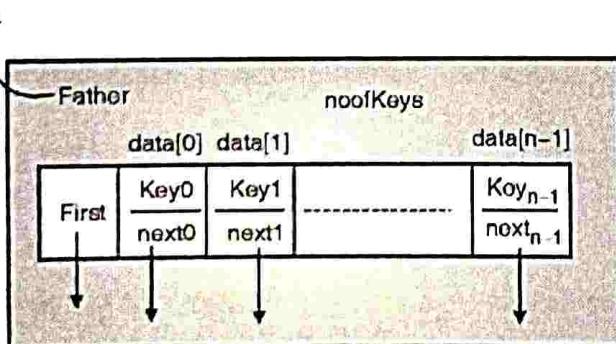


Fig. 4.14.3

Representation of a node of B-tree

```

#define MAX 5
class node;
struct pair
{
    node *next;
    int key;
};
class node
{
public :
    node * first;
    node * father;
    pair data [MAX];
    int noofkeys;
};
  
```



- Structure pair is being used to combine a key and the associated tree pointer.
- Class node can store a maximum of MAX pairs of (key, next). A node with MAX number of keys will give rise to MAX + 1 ways. The additional tree pointer is designated as 'first'.
- 'noofkeys' gives the actual number of keys stored in a node.
- The pointer 'father' points to the father of a node. 'father' pointer will be NULL for the root node.

4.14.1 Insertion of a Key into a B-tree

A key is always inserted in a leaf node. To insert a value X into a B-tree, there are following steps.

Step (a) : If the root is NULL then

- Acquire memory for a new node
root = new node;
- Insert a pair consisting of (X, NULL) in the root node.
root → insertinanode(mypair) [mypair consists of (X, NULL)]

Step (b) : If the root is not NULL then

Step 1 : Find the correct leaf node to which X should be added.

Step 2 : If the leaf node has space to accomodate X, it is inserted. The sorted nature of keys inside the node should be preserved after insertion.

if(p → noofkeys < mkeys)
P → insertinanode(mypair);

The variable 'mkeys' gives the maximum number of keys to be stored in a node. The pair 'mypair' consists of (X, NULL). The function 'insertinanode()' inserts the pair 'mypair' at the appropriate place.

```

void insertinanode(pair mypair)
{
    int i;
    for( i = noofkeys - 1 ; i > 0
        && data [i].Key > mypair.key; i --)
    {
        data[i + 1 ] = data[i];
    }
    data[i + 1 ] = mypair;
    noofkeys ++ ;
}
  
```

The for loop shifts the higher value keys to the right and thus creating space for the new key to be inserted.

- Start traversing from the root with the help of pointer P.
P = root;
- go down to leaf node by selecting the appropriate tree pointer data[i].next such that X > data[i].key.
while(1 (P → leafnode()))
P = P → nextindex(X);

The function leafnode(), checks whether the current node is a leaf node.

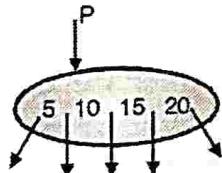
The function nextindex(), gives the appropriate tree pointer for the given X to traverse down to the leaf node.

The function nextindex() is explained as follows :

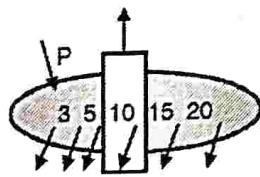
```
node *nextindex(int X)
{
    int i;
    if(X < data [0] . key )
        return first;
    for(i = 0; i < noofkeys; i++)
    {
        if(X < = data [i] . key )
            return data[i - 1].next;
    }
    return data[i - 1 ].Next ;
}
```

Step 3: If the leaf node does not have space to accommodate X. We split the node into two parts. Process of splitting is explained for a 5-way (maximum of 4 keys) tree.

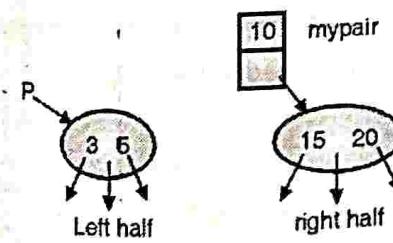
Node for insertion :



Data to be inserted (X) = 3



After 3 is inserted, 10 will become the centre key. Node is split into two parts with the centre key moving up.



The centre key is stored in a pair, 'mypair' with its next field pointing to right half. 'mypair' is moved to the parent node along with the right half. If the parent node is full it is also split. Splitting can propagate all the way upto the root node, creating a new level if the root is split.

C++ function for insert operation

```
void btree::insert(int x)
{
    int index;
    pair mypair;
    node *p, *q;
    mypair.key = x;
    mypair.next = NULL;
    if(root == NULL)
    {
        root = new node;
        root -> insertinanode(mypair);
    }
    else
    {
        p = root;
        while(!(p-> leafnode( )))
            p = p-> nextindex(x);
        if(p->noofkeys<mkeys)
            p-> insertinanode(mypair);
        else
        {
            mypair = p-> splitanode(mypair);
            while(1)
            {
                if(p == root)
                {
                    q = new node;
                    q -> data[0] = mypair;
                    q-> first = root ;
                    q-> father = NULL;
                    q-> noofkeys = 1;
                    root = q;
                    q-> first-> father = q;
                    q-> data[0].next-> father = q;
                    return;
                }
                else
                {
                    p = p-> father;
                    if(p-> noofkeys <mkeys)
                    {
                        p-> insertinanode(mypair);
                        return;
                    }
                }
            }
        }
    }
}
```

```
        else  
            mypair = p->splitanode(mypair);  
    }  
}  
}  
}  
}
```

Example on Insertion In a 5-way B-tree

Fig. 4.14.4, split caused due to Insertion of a new key with value 3. Final B-tree is shown in Fig. 4.14.4(c).

Fig. 4.14.4 is an example of 5-way B-tree. First four keys, 1, 5, 9 and 11 can be stored in the root node. An additional insertion of key will cause an overflow. On insertion of a new key with value 3, the centre key moves up after splitting the node with left node as 1, 3 and right node 9, 11.

Since, the current node (causing overflow) is a root node, the centre key is stored in a new node. B-tree after the split is shown in the Fig. 4.14.4(c).

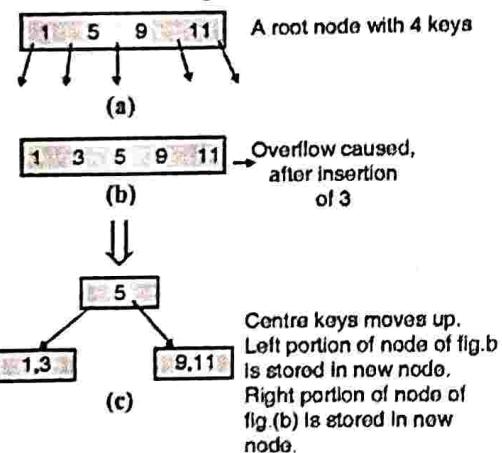
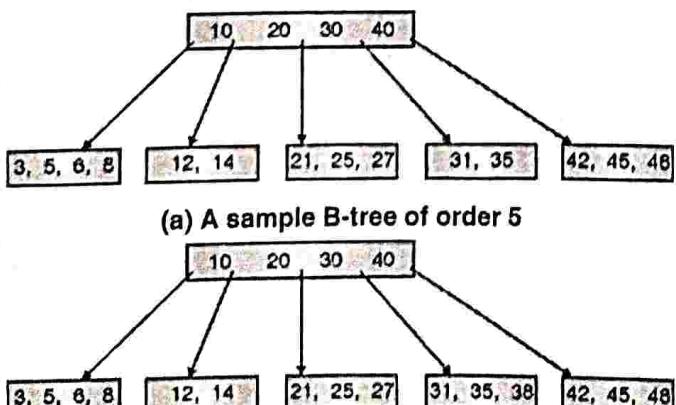


Fig. 4.14.4 : 5-way B-tree

Another Example :



(b) The B-tree after a key with value 38 is added

Fig. 4.14.5

Consider the B-tree of example 4.14.4(a). A new key 38 is to be inserted. Since 38 lies between 30 and 40, subtree shown as 'Y' is selected for insertion. As, there is space in the node, 38 is inserted and insertion process completes. Fig. 4.14.4(b) is the final B-tree after insertion of key with value 38.

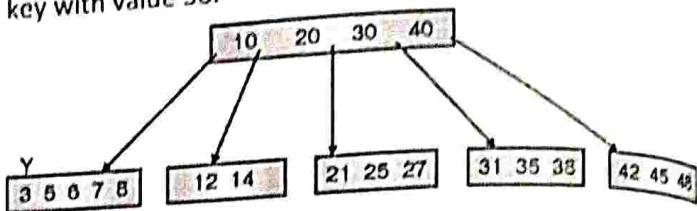
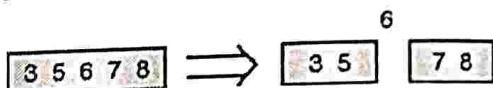


Fig. 4.14.5(c) : B-tree after a key with value 8 is added to the B-tree of Fig. 4.14.5(b)

After insertion of the new key with value 7, an overflow in node Y has been caused. Number of keys in B-tree of order 5 cannot be more than 4. The leaf node is split into two as shown in the following :



Now, the centre key (6, here) moves up.

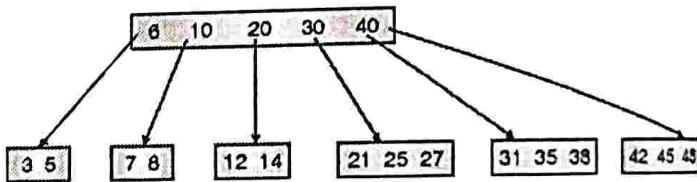


Fig. 4.14.5(d) : B-tree after the split and movement of the centre key in the root node

In Fig. 4.14.5(d), the root node contains 5 keys and hence an overflow has occurred in the root node. Now, the root node is split into two as shown following.



Centre key with value 20 is stored in a new node.

Fig. 4.14.5(e) gives the final B-tree after insertion of a key with value 3 in the B-tree of Fig. 4.14.5(b).

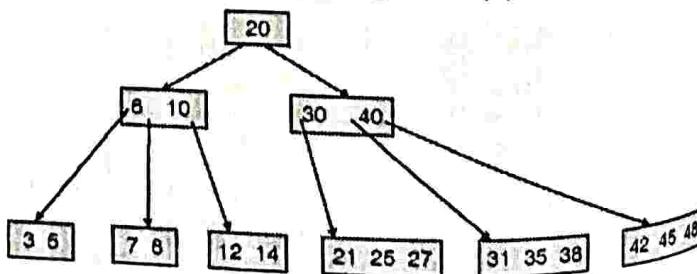
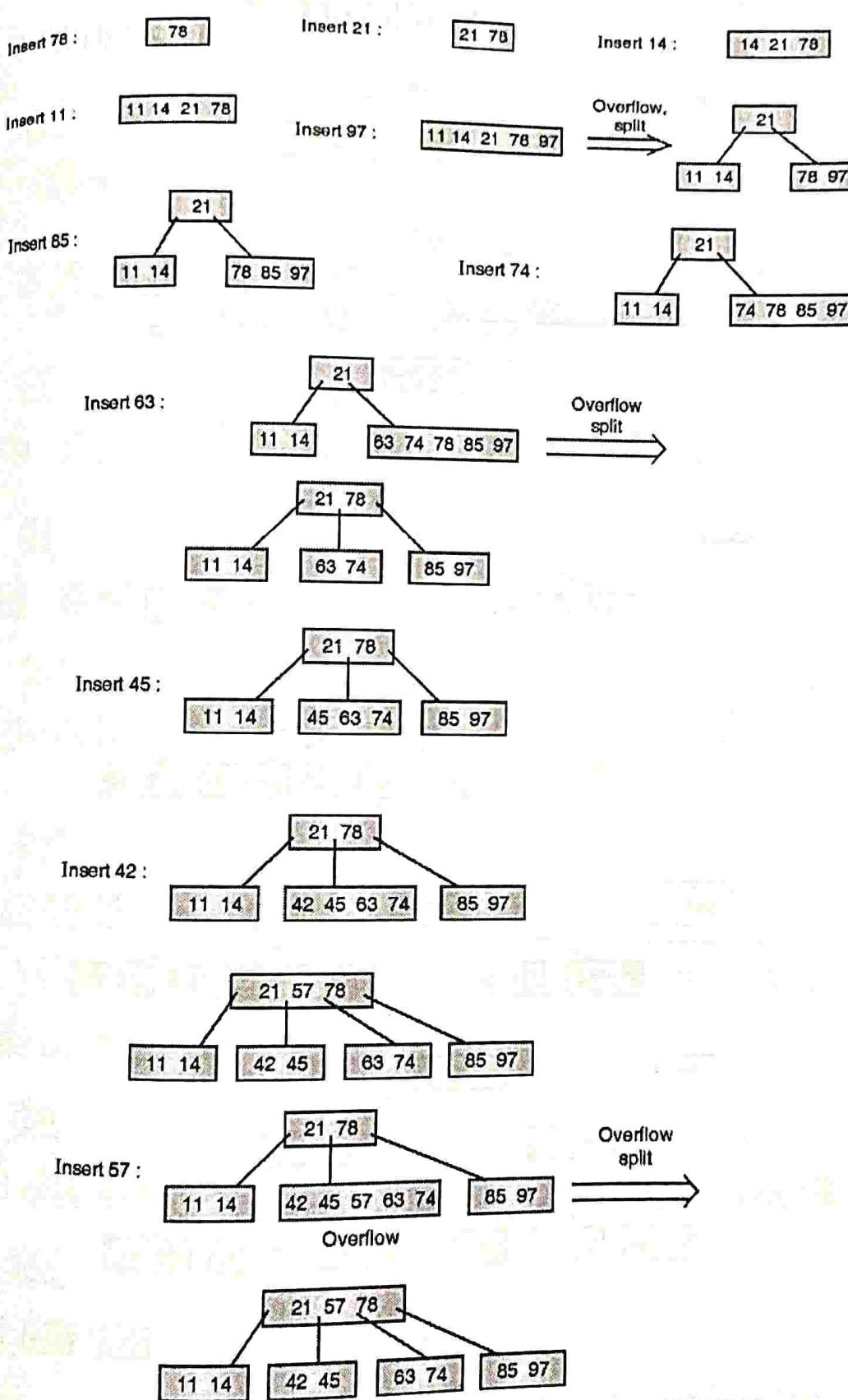
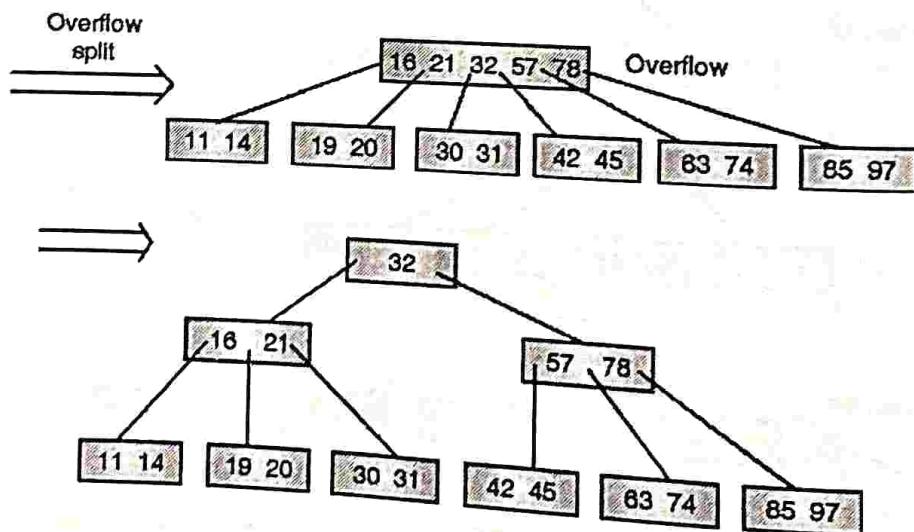
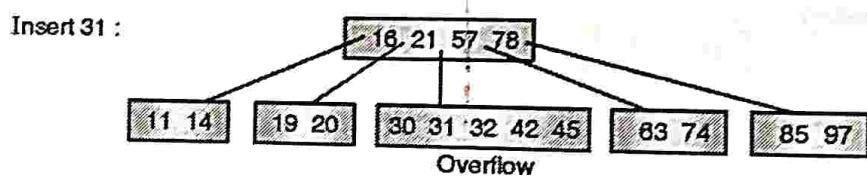
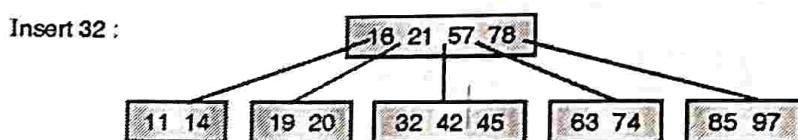
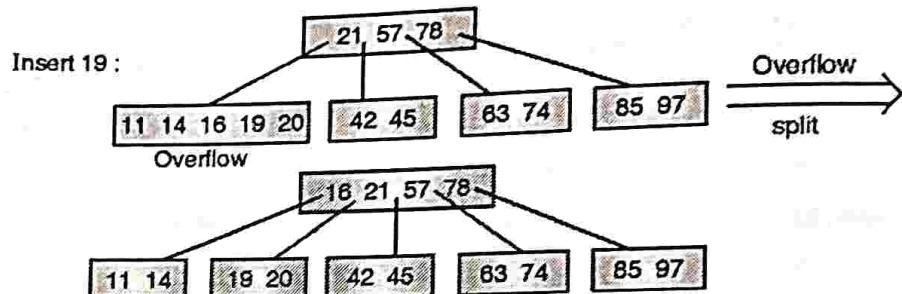
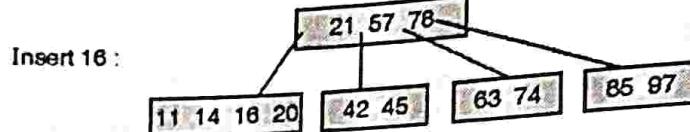
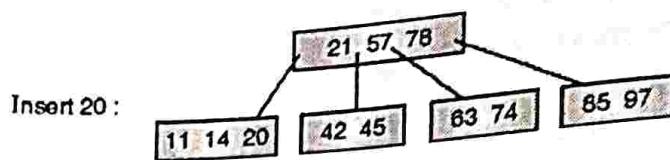


Fig. 4.14.5(e) : The B-tree after a key value 7 is added to the B-tree of Fig. 4.14.5(b)

Example 4.14.1: Explain the steps to build a B-tree of order 5 for the following data :
78, 21, 14, 11, 97, 85, 74, 63, 45, 42, 57, 20, 16, 19, 32, 30, 31.

Solution :

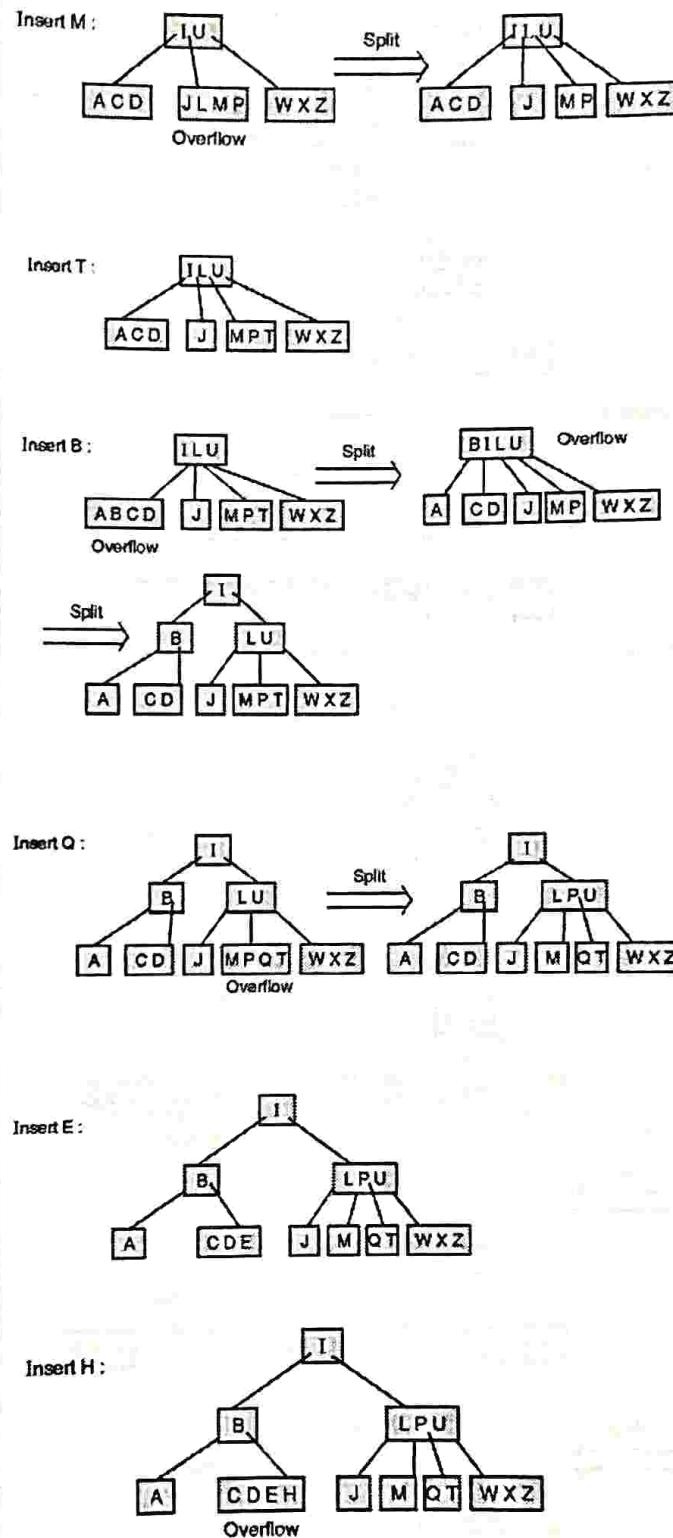
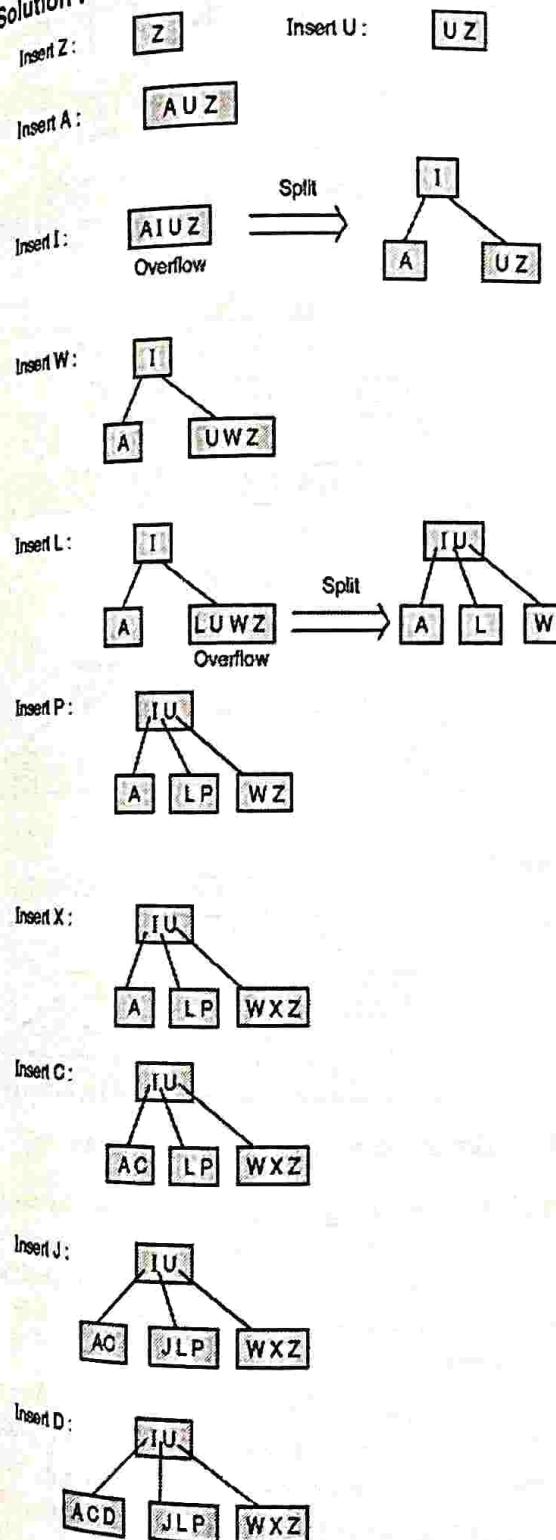


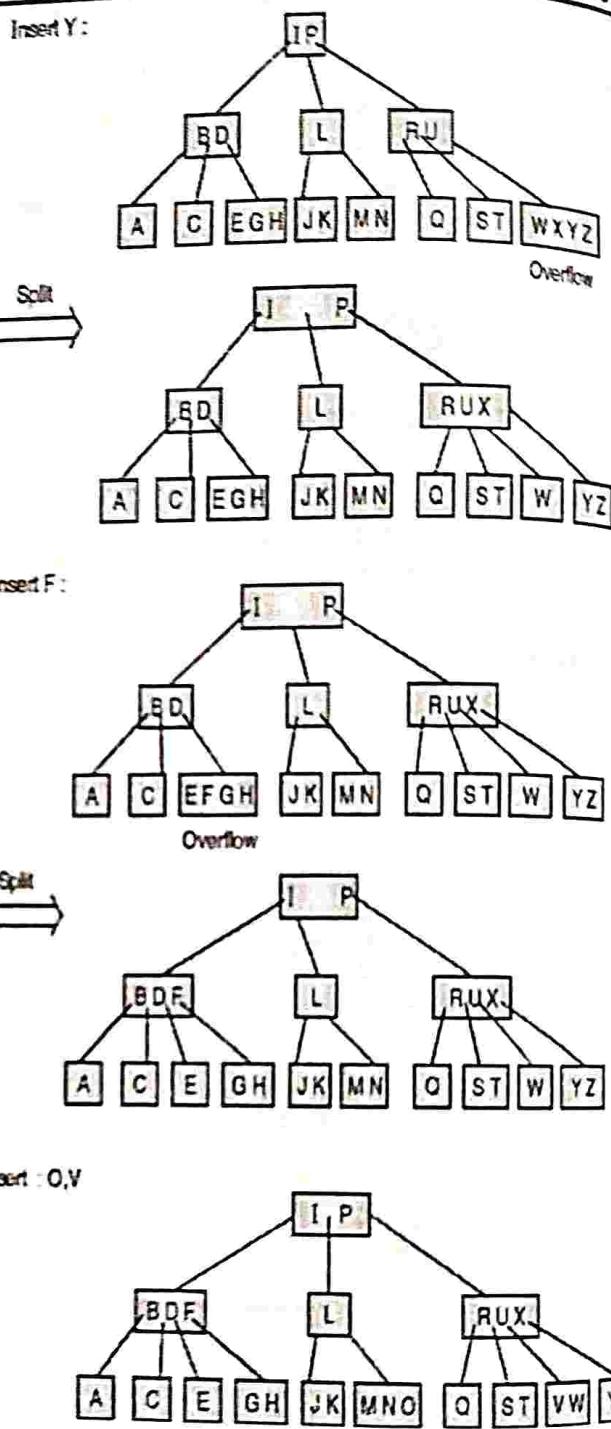
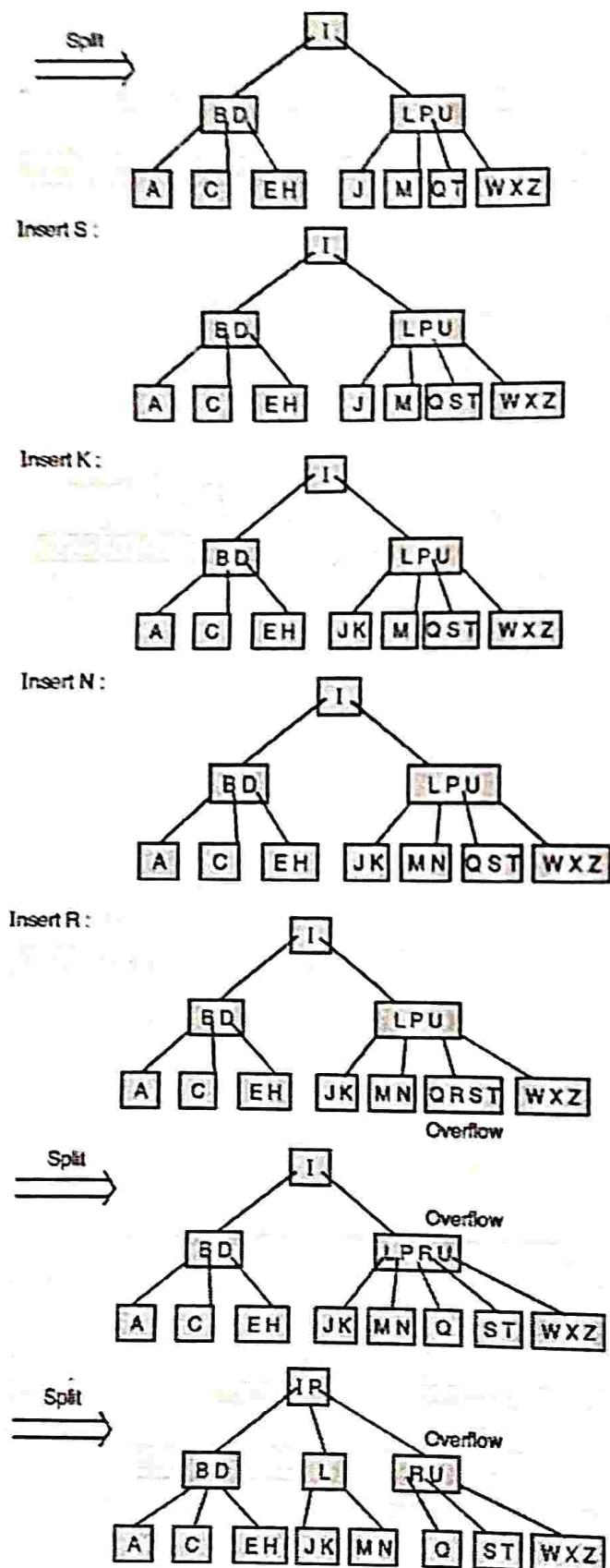


Example 4.14.2: Consider the following sequence of nodes and show the growth of the B-tree of order – 4

Z, U, A, I, W, L, P, X, C, J, D, M, T, B, Q, E, H, S, K, N, R, G, Y, F, O, Y.

Solution :





4.14.2 Deleting a Value from a B-tree

Recall our deletion algorithm for binary search trees; if the value to be deleted is in a node of degree 2, we would replace the value with the smallest value (inorder successor) in its right subtree and then delete the node in the right subtree containing the smallest value.

We can use a similar strategy to delete a value from a B-tree. If the value to be deleted does not occur in a leaf, we replace it with the smallest value (successor) in its right subtree and then proceed to delete the value from the leaf node.

For example, if we wish to delete 67 from the B-tree of Fig. 4.14.6(a), we would find the smallest in 67's right subtree, replace 67 with 68 and then delete the occurrence of 68 in the right subtree.

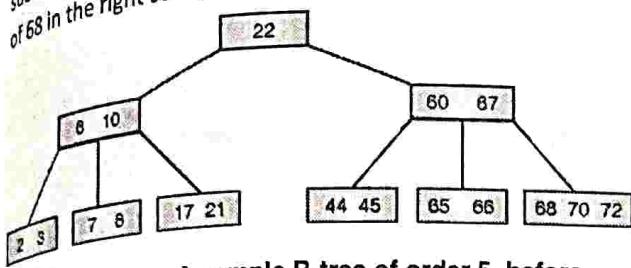


Fig. 4.14.6(a) : A sample B-tree of order 5, before deletion of the key with value 67.

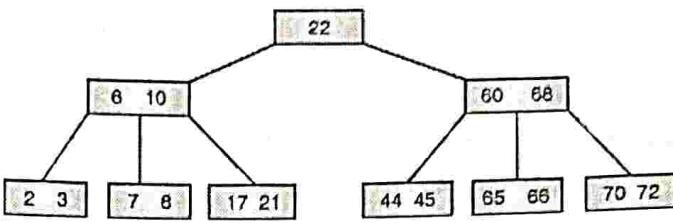


Fig. 4.14.6(b) : B-tree of Fig. 4.14.6(a) after deletion of the key with value 67.

- If deletion of a key causes the node to have less than $\frac{M-1}{2}$ (for a M-way tree) keys, we say an underflow has occurred.
- If underflow does not occur, the deletion algorithm finishes. If it does occur, it must be fixed.

Strategy for fixing underflow :

1. If the left neighbour contains more than $\frac{M-1}{2}$ keys, it can borrow a key from the left neighbour. The concept is shown in the Fig. 4.14.6(c).

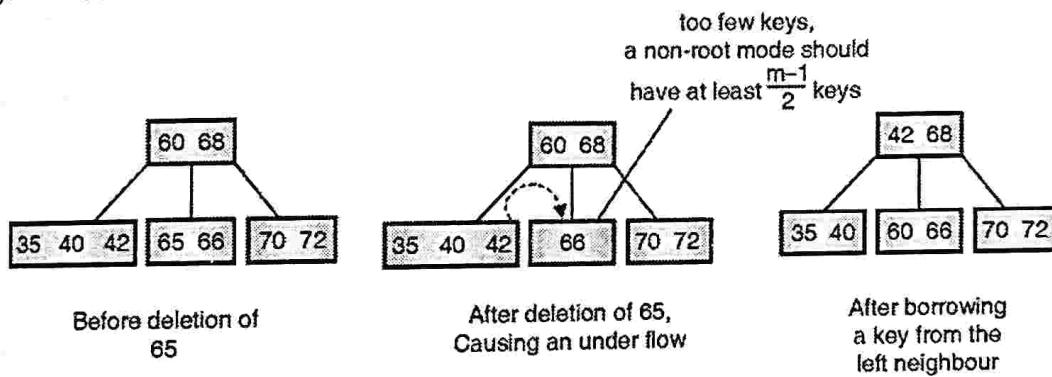


Fig. 4.14.6(c) : Borrowing a key from the left neighbour to fix an underflow.

2. If the left neighbour contains $\frac{M-1}{2}$ keys and the right neighbor contains more than $\frac{M-1}{2}$ keys, it can borrow a key from the right neighbour.
3. If both left and right neighbours contain $\frac{M-1}{2}$ keys, we join together the current node and its neighbour, either left or right, to form a combined node and we must also include in the combined node the value in the parent node that is in-between these two nodes.

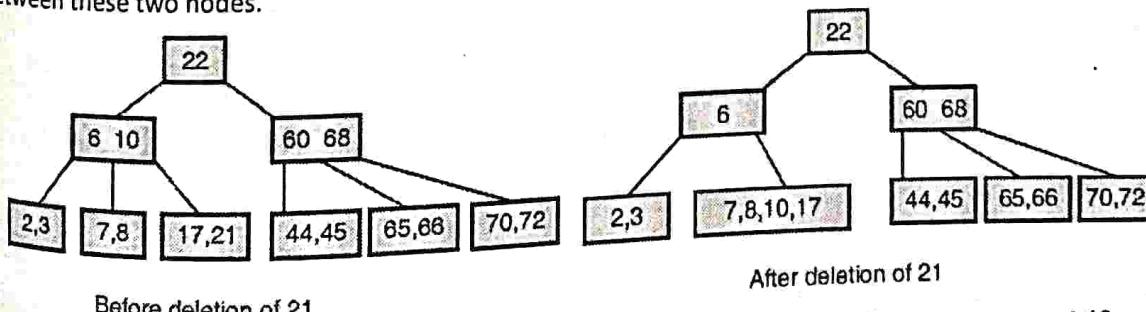


Fig. 4.14.6(d) : Node [7, 8] and the node [17] are merged along with their common parent 10.



Suppose 21 is to be deleted from the B-tree of Fig. 4.14.6(d). Since it is a B-tree of order 5, every node except the root node must have at least $\frac{5-1}{2} = 2$ keys. If 21 is deleted from the leaf node [17, 21] then this node will not have minimum number of keys after deletion. Fig. 4.14.6(d) shows the situation of the tree before deletion of 21 and subsequent merging of this node with its left sibling.

After merging the node [6, 10] becomes [6] and does not have the required minimum number of keys, as shown in the right tree of Fig. 4.14.6(d). To correct this problem, the right sibling of [6] and its parent are merged with [6] to form a new node [6, 22, 60, 68].

This node, now becomes the new root. The final tree after deletion of 21 is shown in Fig. 4.14.6(e).

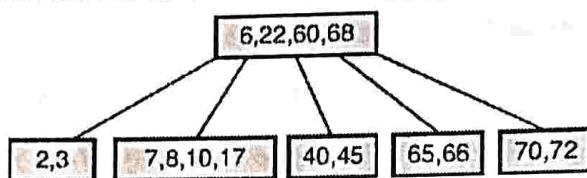


Fig. 4.14.6(e) : The B-tree after deletion of key 21 from B-tree of Fig. 4.14.7(d).

C++ function for delete operation

```

void btree::Delete(int x)
{
    node *left, *right;
    pair *centre;
    node *p, *q;
    int i, j, centreindex;
    p = search(x);
    for(i = 0; p->data[i].key != x; i++)
        /* if p is not a leaf node then locate its successor in
           a leaf node, replace x with its
           successor/predecessor and delete it.*/
    if(!p->leafnode())
    {
        q = p->data[i].next;
        while(!q->leafnode())
            q = q->first;
        p->data[i].key = q->data[0].key;
        p = q;          //p points to leaf node
        x = q->data[0].key;
        i = 0;
    }
    // if(p->leafnode()), p will always be a leaf node
    for(i = i+1; i < p->noofkeys; i++)
        p->data[i-1] = p->data[i];
}
  
```

```

p->noofkeys--;
while(1)
{
    if(p->noofkeys >= mkeys/2)
        return;
    if(p == root)
        if(p->noofkeys>0)
            return;
        else
            {
                root = p->first;
                return;
            }
    // otherwise
    q = p->father;
    if(q->first == p || q->data[0].next == p)
    {
        left = q->first;
        right = q->data[0].next;
        centre = &(q->data[0]);
        centreindex = 0;
    }
    else
    {
        for(i = 1; i < q->noofkeys; i++)
    }
}
  
```

```

if(q->data[i].next == p)
break;
left = q->data[i-1].next;
right = q->data[i].next;
centre = &(q->data[i]);
centreindex = i;

}

/* case 1 : left has one extra key, move a
key from left */
if(left->noofkeys > mkeys/2)

{
    for(i = right->noofkeys-1; i>= 0; i--)
        right->data[i+1] = right->data[i];
    right->noofkeys++;
    right->data[0].key = centre->key;
    centre->key = left->data[left->
noofkeys - 1].key;
    left->noofkeys--;
    return;
}

/* case 2 : right has one extra key, move a
key from right */
else
if(right->noofkeys > mkeys/2)

{
    left->data[left->noofkeys].key =
centre->key;
    left->noofkeys++;
    centre->key = right->data[0].key;
    for(i = 1; i<right->noofkeys; i++)
        right->data[i-1] = right->data[i];
    right->noofkeys--;
    return;
}

else
{
    //merge left and right
    left->data[left->noofkeys].key =
centre->key;
    left->noofkeys++;
    for(j = 0; j<right->noofkeys; j++)
}

```

```

left->data[left->noofkeys+j] =
right->data[j];
left->noofkeys+ = right->noofkeys;
/* delete the pair from the parent
cout<<"\n centre index, noofkeys ";
cout<<centreindex<< " " << q->
noofkeys; */
for(i = centreindex+1; i<q->noofkeys
; i++)
    q->data[i-1] = q->data[i];
    q->noofkeys--;
    p = q; //for next iteration
}
}
}
}

```

Example 4.14.3 : Explain the steps to delete the following data from a B-tree of order 5.

Data : 74, 32

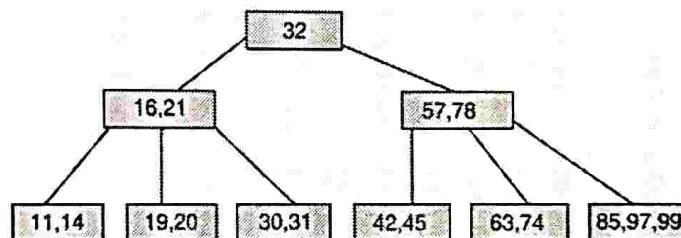
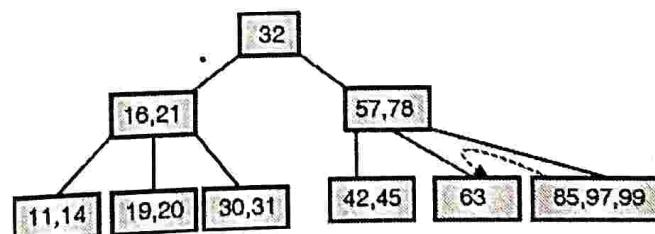


Fig. Ex. 4.14.3

Solution : Every node except the root node should have at least $\frac{5-1}{1} = 2$ keys.

Deletion of 74 :

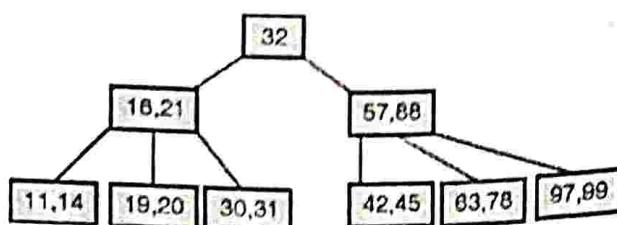
Step 1 : locate 74, since it is in a leaf node it can be deleted in a straight way.



Step 2 : After deletion of 74, the node is left with only one key [63]. It can borrow a key from its right neighbour as the right neighbour has one extra key.

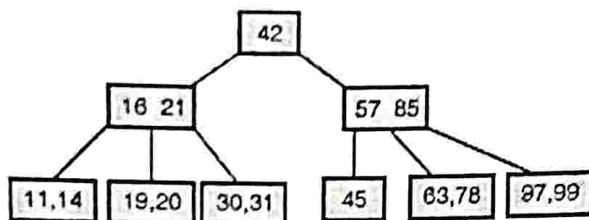


The final tree is shown as follows :

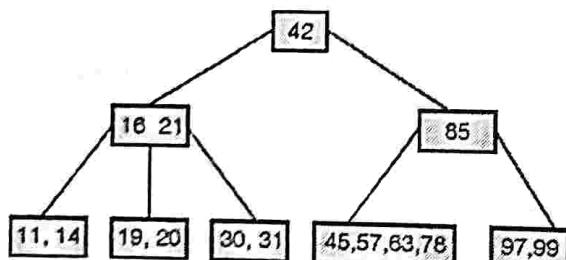


Deletion of 32

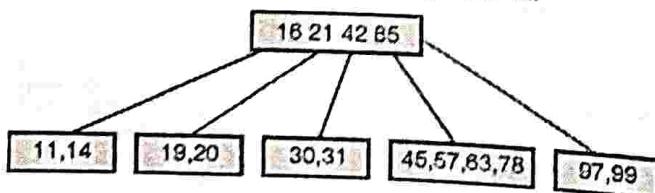
Step 1 : Since, 32 is not in a leaf node it is replaced with its successor and then its successor is deleted. Successor of 32 is 42.



Step 2 : node 45 does not satisfy the minimum key requirement condition. It does not have a left sibling. Its right sibling does not have an extra key. Node 45 is merged with 63, 78 along with their common parent 57.



Step 3 : node 85 does not satisfy the minimum key requirement condition. It does not have a right sibling. Its left sibling 16, 21 does not have an extra key. Node 85 is merged with 16, 21 along with their common parent 42.



4.14.3 B-tree as an ADT

ADT for B-tree is being given using object oriented complete. A program is also included to use the ADT.

Program 4.14.1 : B-tree as an ADT.

```

#define MAX 5
#include <iostream.h>
#include <conio.h>
class node;
struct pair
{
    node *next;
    int key;
};
class node
{
public:
    int noofkeys;
    pair data[MAX];
    node *father;
    node *first;
    node();
    int leafnode();
    void insertinanode(pair x);
    pair splitanode(pair x);
    /* splitting of a leaf node due to overflow, address
       of the second half is returned */
    node *nextindex(int x);
    //index of the subtree containing x
    void display();
};

void node::display()
{
    int i;
    cout<<"(";
    for(i = 0; i<noofkeys; i++)
        cout<<data[i].key<<" ";
    cout<<")";
}
  
```

```

}

node *node::nextindex(int x)
{
    int i;
    if(x<data[0].key)
        return(first);
    for(i = 0; i<noofkeys ; i++)
        if(x <= data[i].key)
            return data[i-1].next;
    return data[i-1].next;
}

int node::leafnode()
{
    if(data[0].next == NULL)
        return 1;
    return 0;
}

void node::insertinanode(pair x)
{
    int i;
    for(i=noofkeys-1;i>=0&&data[i].key>x.key;i--)
        data[i+1] = data[i];
    data[i+1] = x;
    noofkeys++;
}

pair node::splitanode(pair x)
{
    node *T;
    pair mypair;
    int i, j, centre;
    centre = (noofkeys-1)/2;
    //cout<<"\n centre = "<<centre;
    T = new node;
    if(x.key>data[centre].key)

```

```

//Divide the node in two parts(original and T)
{
    for(i=centre+1, j=0;i<= noofkeys; i++,j++)
        T->data[j] = data[i];
    T->noofkeys = noofkeys-centre-1;
    noofkeys = noofkeys-T->noofkeys;
    T->insertinanode(x);
    T->first = T->data[0].next;
    T->father = father;
    mypair.key = T->data[0].key;
    mypair.next = T;
    //Delete the first key from node T
    for(i = 1; i<T->noofkeys; i++)
        T->data[i-1] = T->data[i];
    T->noofkeys--;
}
else
{
    for(i = centre, j = 0; i<noofkeys; i++, j++)
        T->data[j] = data[i];
    T->noofkeys = noofkeys-centre;
    noofkeys = noofkeys-T->noofkeys;
    insertinanode(x);
    T->father = father;
    mypair.key = T->data[0].key;
    mypair.next = T;
    //Delete the first key from node T
    for(i = 1; i<T->noofkeys; i++)
        T->data[i-1] = T->data[i];
    T->noofkeys--;
}
return(mypair);
}

node::node()

```



```
{
    for(int i = 0; i <= MAX; i++)
        data[i].next = NULL;
    noofkeys = 0;
    father = NULL;
}
```

```
class Q
```

```
{    node *data[60];
```

```
    int R, F;
```

```
    public:
```

```
    Q()
    {
```

```
        R = F = 0;
    }
```

```
    int empty()
    {
```

```
        if(R == F)
            return 1;
    }
```

```
    else
        return 0;
    }
```

```
    node *dequeue()
```

```
{
```

```
    return data[F++];
}
```

```
}
```

```
    void enqueue(node *x)
```

```
{
```

```
    data[R++] = x;
}
```

```
    void makeempty()
```

```
{
```

```
    R = F = 0;
}
```

```
};
```

```
class btree
```

```
{
```

```
    int mkeys; //maximum no. of keys in a node
```

```
    node *root;
```

```
    public:
```

```
    btree(int n)
```

```
{
```

```
        mkeys = n;
    }
```

```
    root = NULL;
}
```

```
    void insert(int x);

```

```
    void displaytree();

```

```
    node *search(int x);

```

```
    void Delete(int x);
}
```

```
node * btree::search(int x)
```

```
{
```

```
    node *p;

```

```
    int i;

```

```
    p = root;

```

```
    while(p != NULL)
    {
```

```
        for(i = 0; i < p->noofkeys; i++)
    
```

```
        if(x == p->data[i].key)
    
```

```
            return(p);
    }
```

```
    p = p->nextindex(x);
}
```

```
    return NULL;
}
```

```
void btree::displaytree()
```

```
{
```

```
    Q q1, q2;

```

```
    node *p;

```

```
    q1.enqueue(root);

```

```

while(!q1.empty())
{
    q2.makeempty();
    cout<<"\n";
    while(!q1.empty())
    {
        p = q1.dequeue();
        p->display(); cout<<" ";
        if(!p->leafnode())
        {
            q2.enqueue(p->first);
            for(int i=0; i<p->noofkeys; i++)
                q2.enqueue(p->data[i].next);
        }
    }
    q1 = q2;
}
}

void btree::insert(int x)
{
    int index;
    pair mypair;
    node *p, *q;
    mypair.key = x;
    mypair.next = NULL;
    if(root == NULL)
    {
        root = new node;
        root->insertinanode(mypair);
    }
    else
    {
        p = root;
        while(!(p->leafnode()))
            p = p->nextindex(x);
    }
}

```

```

if(p->noofkeys < mkeys)
    p->insertinanode(mypair);
else
{
    mypair = p->splitanode(mypair);
    while(1)
    {
        if(p == root)
        {
            q = new node;
            q->data[0] = mypair;
            q->first = root;
            q->father = NULL;
            q->noofkeys = 1;
            root = q;
            q->first->father = q;
            q->data[0].next->father = q;
            return;
        }
        else
        {
            p = p->father;
            if(p->noofkeys < mkeys)
            {
                p->insertinanode(mypair);
                return;
            }
            else
                mypair=p->splitanode(mypair);
        }
    }
}

```



```
void btree::Delete(int x)
{
    node *left, *right;
    pair *centre;
    node *p, *q;
    int i, j, centreindex;
    p = search(x);
    for(i = 0; p->data[i].key != x; i++)
    ;
    /* if p is not a leaf node then locate its successor
       in a leaf node, replace x with its
       successor/predecessor and delete it.*/
    if(!p->leafnode())
    {
        q = p->data[i].next;
        while(!q->leafnode())
            q = q->first;
        p->data[i].key = q->data[0].key;
        p = q;           //p points to leaf node
        x = q->data[0].key;
        i = 0;
    }
    //if(p->leafnode()), p will always be a leaf node
    for(i = i+1; i < p->noofkeys; i++)
        p->data[i-1] = p->data[i];
    p->noofkeys--;
    while(1)
    {
        if(p->noofkeys >= mkeys/2)
            return;
        if(p == root)
            if(p->noofkeys>0)
                return;
        else
    }
```

```
{
    root = p->first;
    return;
}
// otherwise
q = p->father;
if(q->first == p || q->data[0].next == p)
{
    left = q->first;
    right = q->data[0].next;
    centre = &(q->data[0]);
    centreindex = 0;
}
else
{
    for(i = 1; i < q->noofkeys; i++)
        if(q->data[i].next == p)
            break;
    left = q->data[i-1].next;
    right = q->data[i].next;
    centre = &(q->data[i]);
    centreindex = i;
}
/*case 1 : left has one extra key, move a key from
left */
if(left->noofkeys > mkeys/2)
{
    for(i = right->noofkeys-1; i >= 0; i--)
        right->data[i+1] = right->data[i];
    right->noofkeys++;
    right->data[0].key = centre->key;
    centre->key = left->data[left->
noofkeys - 1].key;
    left->noofkeys--;
    return;
}
```

```

/* case 2 : right has one extra key, move a key
from right*/
else
if(right->noofkeys >mkeys/2)
{
    left->data[left->noofkeys].key
    = centre->key;
    left->noofkeys++;
    centre->key = right->data[0].key;
    for(i = 1; i<right->noofkeys; i++)
        right->data[i-1] = right->data[i];
    right->noofkeys--;
    return;
}
else
{ //merge left and right
    left->data[left->noofkeys].key
    = centre->key;
    left->noofkeys++;
    for(j = 0; j<right->noofkeys; j++)
        left->data[left->noofkeys+j]
        = right->data[j];
    left->noofkeys+ = right->noofkeys;
    /* delete the pair from the parent
    //cout<<"\n centre index, noofkeys ";
    //cout<<centreindex<<" "<<q->noofkeys;
    */
    for(i = centreindex+1; i<q->noofkeys ; i++)
        q->data[i-1] = q->data[i];
    q->noofkeys--;
    p = q; //for next iteration
}
}
}

```

```

void main()
{
    int n, i, x, op;
    node *p;
    cout<<"\n maximum no. of keys in a node ? :";
    cin>>n;
    btree b(n);
    do
    {
        cout<<"\n\n 1)Insert\n2)Search\n 3)Delete\n
4)Print\n 5)Quit";
        cout<<"\n Enter your choice : ";
        cin>>op;
        switch(op)
        {
            case 1: cout<<"\n Enter a data : ";
                cin >> x;
                b.insert(x);
                cout<<"\n Tree after insertion : ";
                b.displaytree();
                break;
            case 2: cout<<"\n Enter a data : ";
                cin>>x;
                p = b.search(x);
                if(p!=NULL)
                {
                    cout<<"\n Found in the node : ";
                    p->display();
                }
                else
                    cout<<"\n Not found";
                break;
            case 3: cout<<"\n Enter a data : ";
                cin>>x;

```



```

p = b.search(x);
if(p! = NULL)
{
    b.Delete(x);
    b.displaytree();
}
else
    cout<<"\n Not found";
break;
case 4: b.displaytree();
break;
}
}while(op! = 5);
}

```

4.15 B+ Trees

MU - Dec. 17

University Question

Q. Explain B+ Tree. (Dec. 17, 2 Marks)

- B⁺ tree is a variation of B-tree data structure. In a B⁺ tree , data pointers are stored only at the leaf nodes of the tree. In a B⁺ tree structure of a leaf node differs from the structure of internal nodes.
- The leaf nodes have an entry for every value of the search field, along with a data pointer to the record (or to the block that contains this record).
- The leaf nodes of the B⁺ tree are linked together to provide ordered access on the search field to the records.
- Internal nodes of a B⁺ tree are used to guide the search.
- Some search field values from the leaf nodes are repeated in the internal nodes of the B⁺ tree.

Structure of internal node

The structure of the internal node is shown in Fig. 4.15.1.

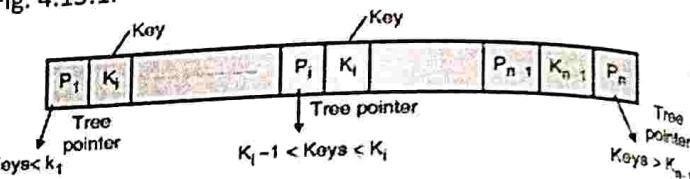


Fig. 4.15.1 : Structure of an internal node of B⁺ tree

- Each internal node is of the form $\langle P_1, K_1, P_2, K_2 \dots P_{n-1}, K_{n-1}, P_n \rangle$
- K_i is the key and P_i is a tree pointer
- Within each internal node, $K_1 < K_2, \dots < K_{n-1}$
- For all search field value x in the subtree pointed at by P_i , we have $K_{i-1} < x \leq K_i$.
- Each internal node has at most p tree pointers.
- Each internal node, except the root, has at least $\lceil (P/2) \rceil$ tree pointers.

Structure of a leaf node : The structure of a leaf node of a B+tree is shown in Fig. 4.15.2.

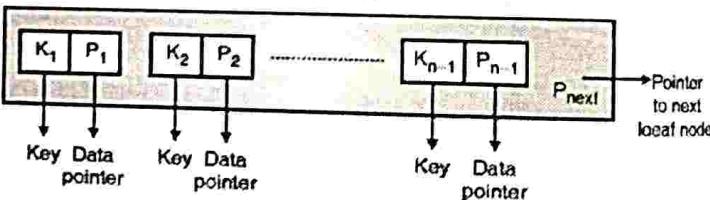
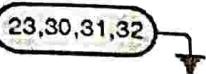
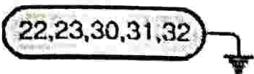
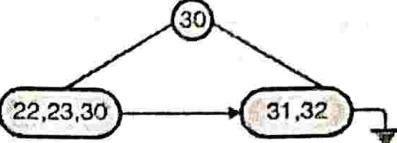
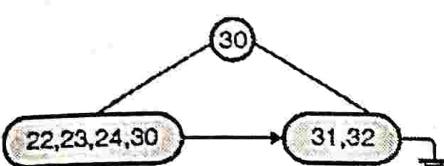
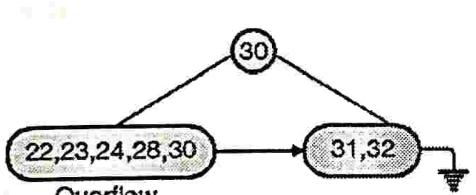
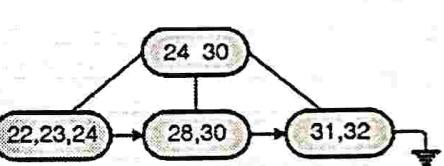
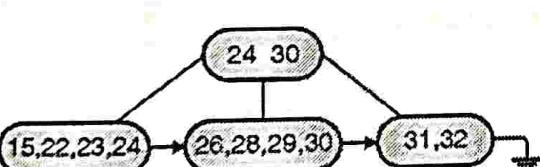
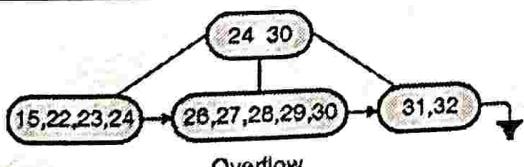
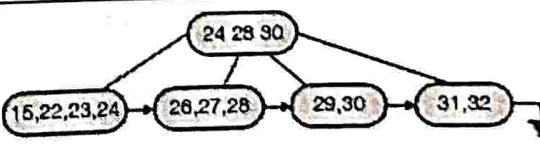
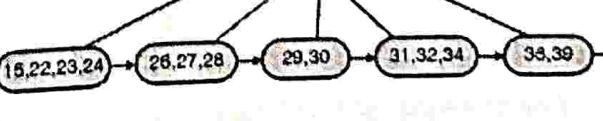


Fig. 4.15.2 : Structure of a leaf node of B⁺ tree

- Each leaf node is of the form $\langle K_1, P_1 \rangle, \langle K_2, P_2 \rangle \dots \langle K_{n-1}, P_{n-1} \rangle, P_{next}$
- Within each leaf node, $K_1 < K_2 \dots < K_{n-1}$.
- P_i is a data pointer that points to the record whose search field value is k_i .
- Each leaf node has at least $\lceil (P/2) \rceil$ values.
- All leaf nodes are at the same level.

Example 4.15.1 : Construct a B⁺ tree of order 5 for the following data : 30, 31, 23, 32, 22, 28, 24, 29, 15, 26, 27, 34, 39, 36

Solution :

| Insert 30, 31, 23, 32 | Tree after insertion  | Tree after splitting (if required) |
|-----------------------------|---|--|
| 22 |  Overflow |  |
| 24 |  | |
| 28 |  Overflow |  |
| 29, 15, 26 |  | |
| 27 |  Overflow |  |
| 34, 39, 36 |  Overflow |  |

4.16 Splay Tree

MU - May 18, Dec. 19

University Question

Q. Explain Splay Tree. (May. 18, Dec. 19, 5 Marks)

- Splay tree is a binary search tree.
- In a splay tree, M consecutive operations can be performed in $O(M \log N)$ time.
- A single operation may require $O(N)$ time but average time to perform M operations will need $O(M \log N)$ time.

- When a node is accessed, it is moved to the top through a set of operations known as splaying. Splaying technique is similar to rotation in an AVL tree. This will make the future access of the node cheaper.
- Unlike AVL tree, splay trees do not have the requirement of storing Balance Factor of every node. This saves the space and simplifies algorithm to a great extent.
- There are two standard techniques of splaying.
 - (1) Bottom up splaying
 - (2) Top down splaying

4.16.1 Bottom up Splaying

- Idea behind bottom up splaying is explained below :
- Rotation is performed bottom up along the access path.
- Let X be a (non root) node on the access path at which we are rotating.
- i) If the parent of X is the root of the tree, we rotate X and the parent of X. This will be the last rotation required.
- ii) If X has both a parent (P) and Grand parent (G) then like an AVL tree there could be four cases. These four cases are :
 - 1) X is a left child and P is a left child
 - 2) X is a left child and P is right child
 - 3) X is a right child and P is a left child
 - 4) X is a right child and P is a right child.

We will consider two cases and other two are symmetrical.

Case I :

X is left child of its parent P and P is a left child of its parent G (Zig-Zig).

A Zig-Zig rotation is shown in the Fig. 4.16.1.

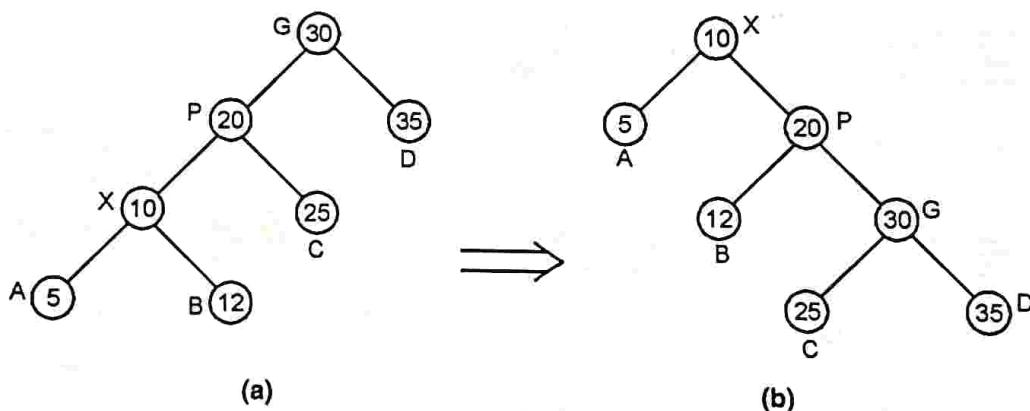


Fig. 4.16.1 : A Zig-Zig rotation

- X becomes the right root node. P becomes the right child of X and G becomes the right child of P.
- Right sub tree of node P in Fig. 4.16.1 (a) becomes the left sub tree of G in Fig. 4.16.1 (b).
- Right sub tree of X in Fig. 4.16.1(a) becomes the left sub tree of P in Fig. 4.16.1 (b).

Case II :

X is the right child of its parent and P is the left child of its parent G (Zig-Zag)

A Zig-Zag rotation is shown in the Fig. 4.16.2.

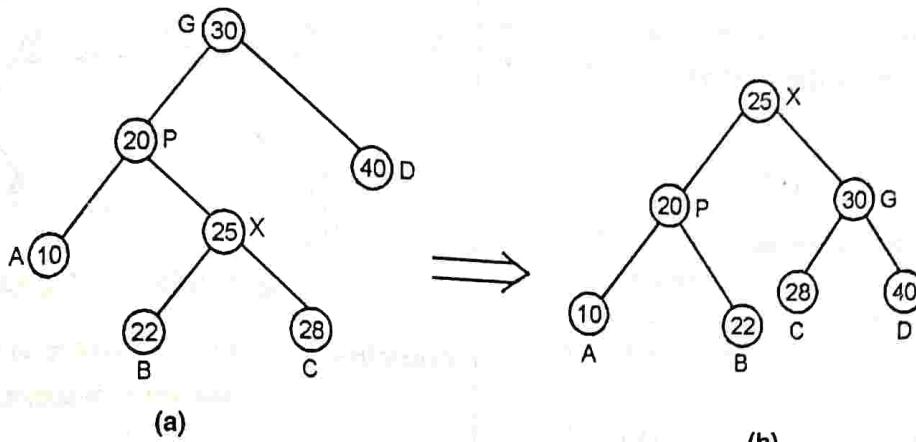


Fig. 4.16.2 : A Zig-Zag rotation

- X becomes the root node.
- G becomes the right child of X.
- Left child of X, becomes the right child of P
- Right child of X becomes the left child of G.

Case III :

X is a right child of its parent P and P is a right child of its parent G (Zag-Zag). A Zag-Zag rotation is shown in the Fig. 4.16.3.

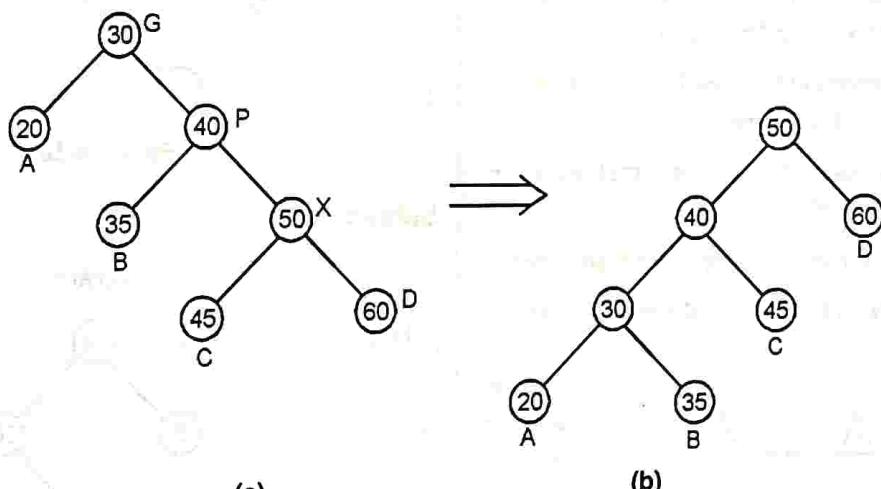


Fig. 4.16.3 : A Zag-Zag rotation

Case IV :

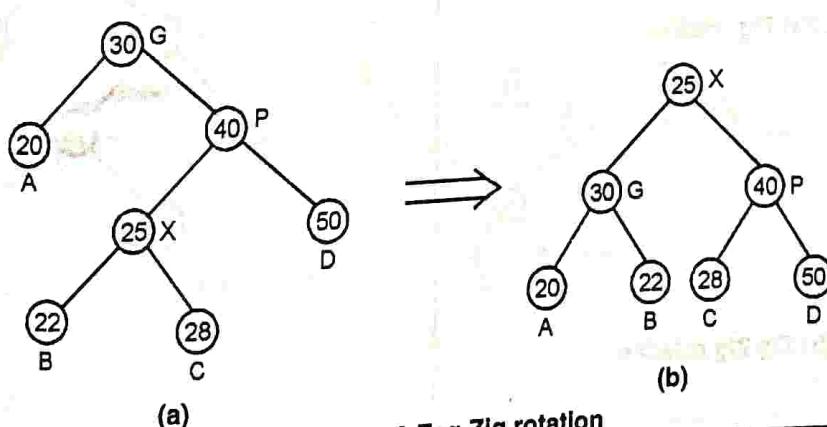


Fig. 4.16.4 : A Zag-Zig rotation



X is left child of its parent P and P is the right child of its parent G (Zag-Zig). A Zag-Zig rotation is shown in the Fig. 4.16.4

4.16.2 Top Down Splaying

When an item X is inserted as a leaf, a series of tree rotations brings X at the root. These rotations are known as splaying. A splay is also performed during searches, and if an item is not found, a splay is performed on the last node on the access path.

- A top down traversal is performed to locate the leaf node.
- Splaying is done using bottom up traversal.
- This can be done by storing the access path, during top down traversal on a stack.

Top down splaying is based on splaying on the initial traversal path. A stack is not required to save the traversal path.

At any point in the access, we have :

- i) A current node X that is the root of its sub tree and represented as the "middle" tree.
- ii) Tree L stores nodes in the tree T that are less than X, but not in the X's sub tree.
- iii) Tree R stores nodes in the tree T that are larger than X, but not in X's sub tree.
- iv) Initially, X is the root of T, and L and R are empty.

Three rotations Zig, Zig-Zig and Zig-Zag are shown in Fig. 4.16.5.

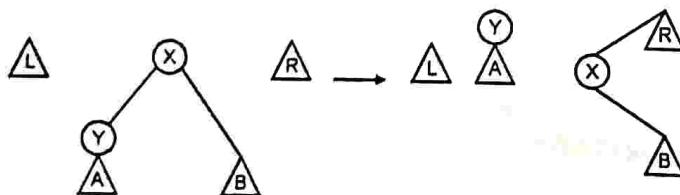


Fig. 4.16.5(a) Zig rotation

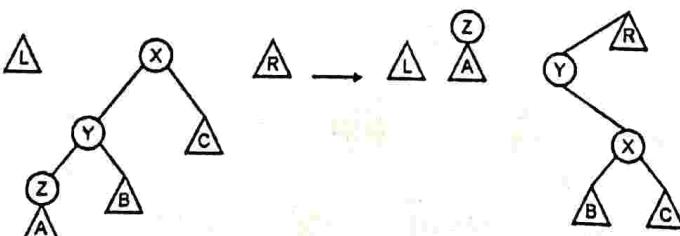


Fig. 4.16.5(b) Zig-Zig rotation

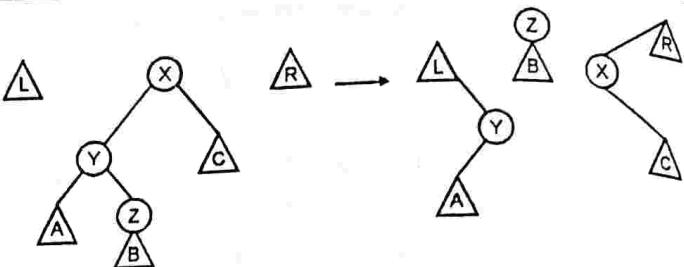


Fig. 4.16.5(c) Zig-Zag rotation

Example 4.16.1 : Perform splaying on the tree given below element 19 is accessed.

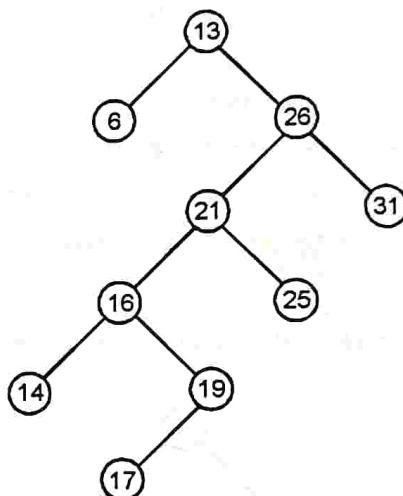
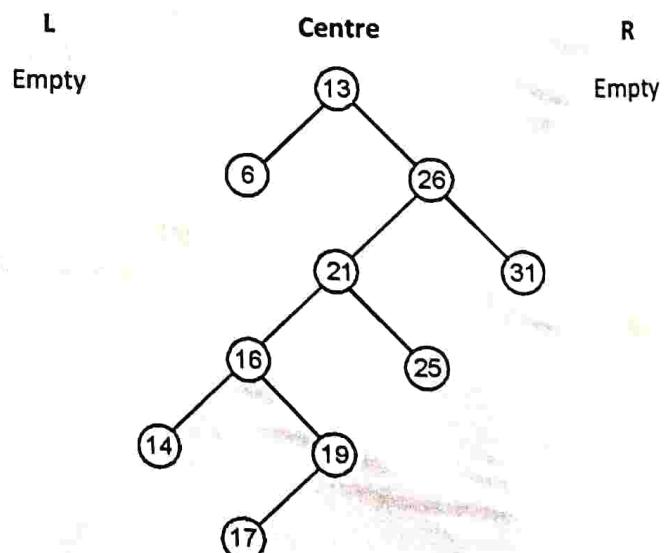


Fig. Ex. 4.16.1

Solution :



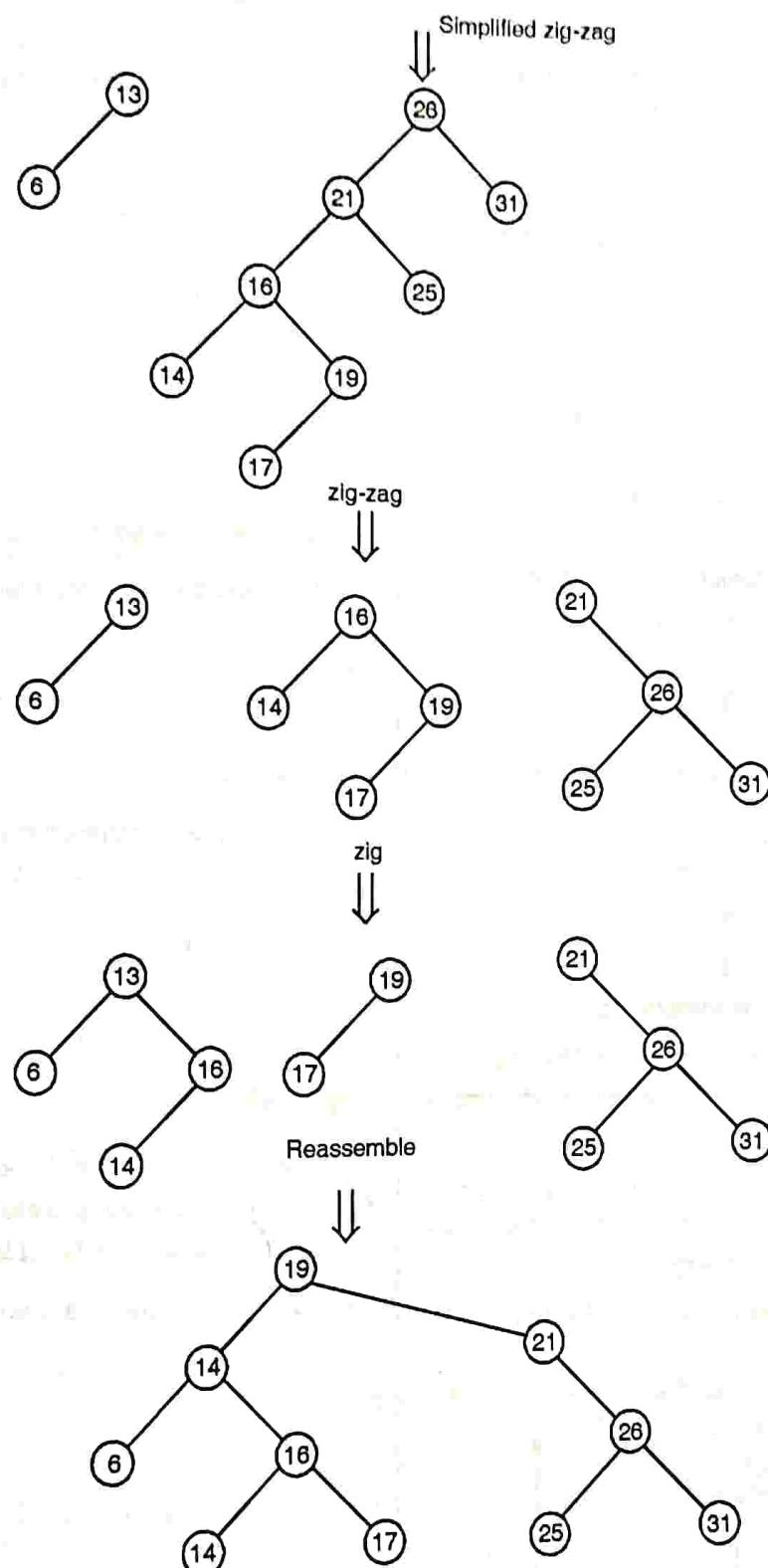


Fig. Ex. 4.16.1(a) : Steps in top-down splay



4.17 Trie Indexing

MU - May 18, Dec. 18

University Questions

- Q. Explain Trie. (May 18, 5 Marks)
 Q. Describe Tries with an Example. (Dec. 18, 5 Marks)

- Trie indexing is useful when key values are of varying size.
- A trie is a tree of degree $P \geq 2$.
- Tries are useful for storing words as a string of characters.
- In a trie, each path from the root to a leaf corresponds to one word.
- Node of the trie correspond to the prefixed of words.
- A sample trie of words {THEN, THIS, SIN} is shown in the Fig. 4.17.1.

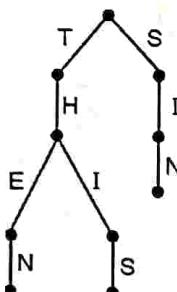


Fig. 4.17.1 : A sample trie

To avoid confusion between words like THE and THEN, a special end marker symbol '\0' is added at the end of each word.

In Fig. 4.17.2, there is a trie representing the set of words {THE, THEN, TIN, SIN, THIN, SING}. Endmarker symbol '\0' is added to end each word.

Each node of a trie has at most 27 children one for each letter and for '\0'.

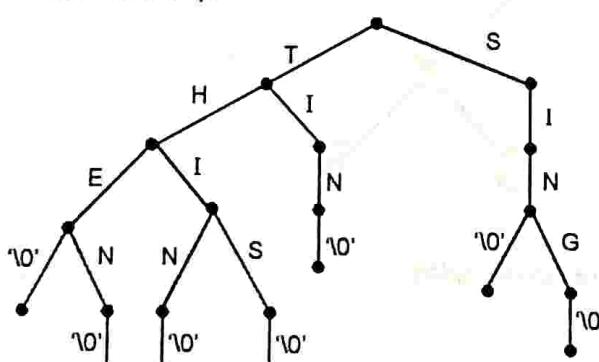


Fig. 4.17.2 : A trie

- Most nodes will have fewer than 27 children.
- A leaf node reached by an edge labelled '\0' cannot have any children and it need not be there.

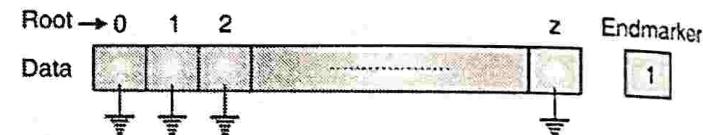
Structure of a trie node

```

Struct trie
{   trie *data [26] ;
    int endmarker ;
    //endmarker = 1 indicates end of a word.
};

data[0] is for character A
:
data[25] is for character t
    
```

An empty trie is represented by a tree of single node. Data pointers are set to NULL.



C++ function to create an empty trie

```

root = new trie ;
root → endmarker = 1 ;
for(i = 0; i < 26; i++)
    root → data [i] = NULL ;
    
```

Insertion :

A character word can be added recursively to a trie. We can trace the existing prefix of a word in the trie and remaining characters can be added to the trie.

C++ function to insert a word in a trie.

```

insert(trie *head, char *x)
{
    int i ;
    if (*x == '\0')
        head → endmarker = 1 ; // end of word
    else
    {
        head → endmarker = 0;
        if(head → data [*x - 65] == NULL)
        {
            head → data [*x - 65] = new trie ;
            head = head → data [*x - 65];
        }
    }
}
    
```

```

/* 65 has been subtracted from x to map
it to an integer location */
for(i = 0; i < 26; i++)
head → data [i] = NULL ;
insert(head, x+1);
}
}
}

```

Deletion

- Deletion of a word from trie can be implemented with the help of a stack.
- As we traverse the trie to locate the word, address of each node is pushed on top of the stack.
- Now, the chain leading to node containing the current word is deleted.

```

void Delete(trie *head, char *x)
{ stack S;
while(*x != '\0')
{ S.push(head → data [*x - 65])
  head = head → data [*x - 65];
  x++;
}
}

```

```

while(! S.empty( )) && chain(head = S.pop)
delete head;
}

int chain(trie * head)
{ n = 0;
for(i = 0; i < 26; i++)
if(head → data [i] != NULL )
n++;
if(n == 0 || n == 1)
return 1; // part of chain
else
return 0;
}

```

Function chain(), checks whether the node is part of the chain leading to terminal node of a trie.

4.17.1 Compact Trie

In this trie, chains which lead to leaves are trimmed.

In such tries, these are two types of nodes :

1. Terminal node – Stores a word
2. Internal node – To locate a word

Example 4.17.1 Create a compact trie with the following words : August, aveuger, Heinkel, Hell, driver, Machhi, masadyer, Spitfire, sykhot.

Solution :

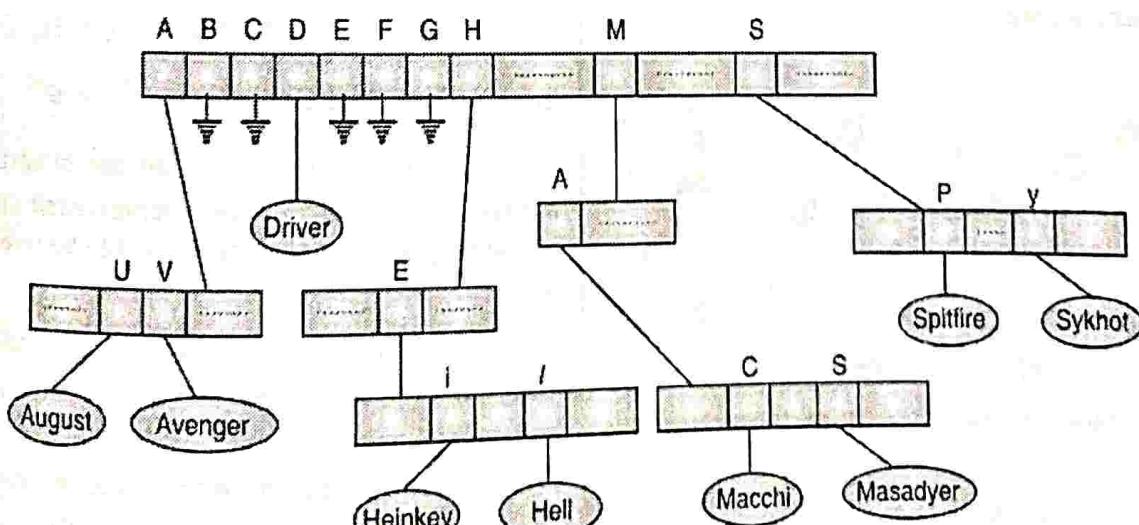


Fig. 4.17.1



Module 5

Graphs

Syllabus

Introduction, Graph Terminologies, Representation of graph, Graph Traversals – Depth First Search (DFS) and Breadth First Search (BFS), Graph Application – Topological Sorting.

5.1 Terminology and Representation

5.1.1 Definition

MU - May 15, May 18

University Question

Q. What is a graph ? (May 15, May 18, 5 Marks)

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. 5.1.1.

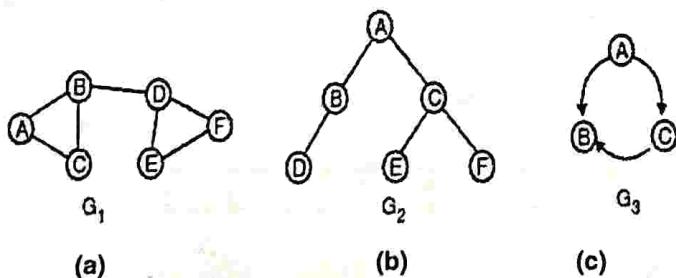


Fig. 5.1.1 : Graphs

The set representation for each of these graphs is given by

$$V(G_1) = \{A, B, C, D, E, F\}$$

$$V(G_2) = \{D, E, F\}$$

$$V(G_3) = \{A, B, C\}$$

$$E(G_1) = \{(A, B), (A, C), (B, C), (B, D), (D, E), (E, F), (F, A)\}$$

$$E(G_2) = \{(D, E), (D, F), (E, F)\}$$

$$E(G_3) = \{(A, B), (A, C), (B, C)\}$$

5.1.2 Undirected Graph

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.

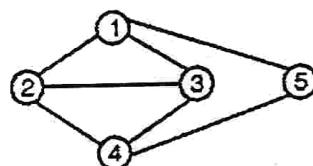


Fig. 5.1.2 : Example of an undirected graph

The set of vertices $V = \{1, 2, 3, 4, 5\}$.

The set of edges $E = \{(1, 2), (1, 3), (1, 5), (2, 3), (2, 4), (3, 4), (4, 5)\}$.

5.1.3 Directed Graph

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 .

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. 5.1.3.

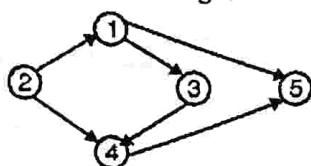


Fig. 5.1.3 : Example of a directed graph

The set of vertices $V = \{1, 2, 3, 4, 5, 6\}$.
 The set of edges $E = \{(1,3), (1,5), (2,1), (2,4), (3,4), (4,5)\}$

5.1.4 A 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. 5.1.4.

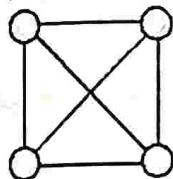


Fig. 5.1.4 : A complete graph

5.1.5 Weighted Graph

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. 5.1.5 is an example of a weighted graph.

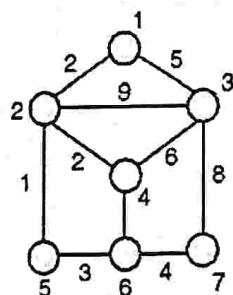


Fig. 5.1.5 : A weighted graph

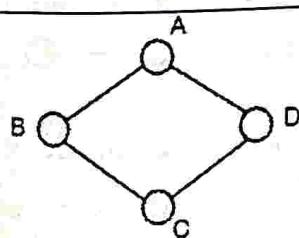


Fig. 5.1.8 : A disconnected graph with 3 components

The graph in Fig. 5.1.7 is a connected graph, but the graph in Fig. 5.1.8 is not connected as there is no path between D and E or between G and H. The graph of Fig. 5.1.8 consists of 3 connected components.

5.1.6 Adjacent Nodes

Two vertices V_1 and V_2 are said to adjacent if there is an edge between V_1 and V_2 .

5.1.7 Path

A path from vertex V_0 to V_n is a sequence of vertices $V_0, V_1, V_2 \dots V_{n-1}, V_n$. 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.

5.1.8 Cycle

A cycle is a simple path that begins and ends at the same vertex. Fig. 5.1.6 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

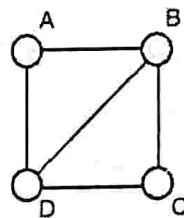


Fig. 5.1.6 : A graph with cycles

5.1.9 Connected Graph

A graph is said to be connected if there exists a path between every pair of vertices V_i and V_j .

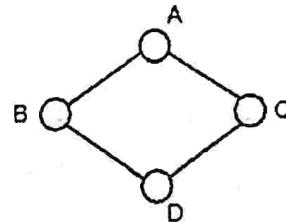


Fig. 5.1.7 : A connected graph



5.1.10 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 $E(G)$.

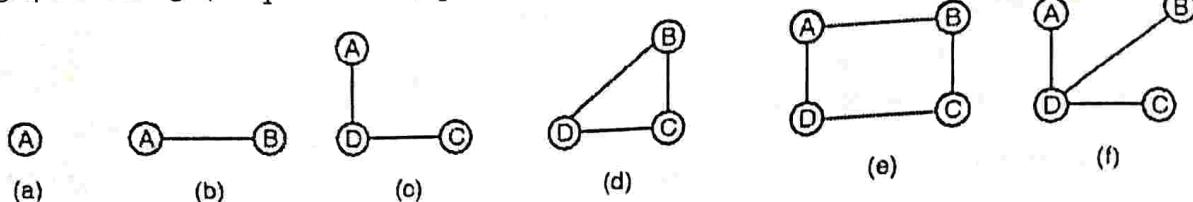


Fig. 5.1.9 : Some subgraphs of the graph of Fig. 5.1.6

5.1.11 Component

A component H of an undirected graph is a maximal connected subgraph. The graph of Fig. 5.1.10 has three components H_1 , H_2 and H_3 .

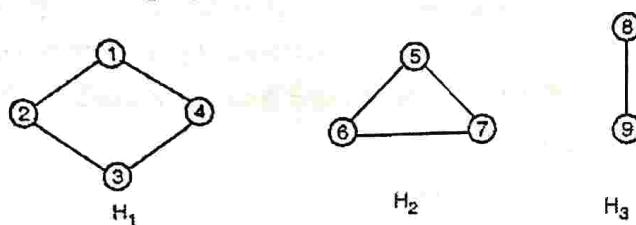


Fig. 5.1.10 : A graph with three components

5.1.12 Degree of a Vertex

- 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**.

5.1.13 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. 5.1.11.

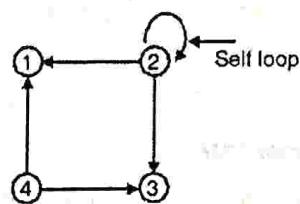


Fig. 5.1.11 : A graph with self loop

5.1.14 Multigraph

A graph with multiple occurrences of the same edge is known as a multigraph. An example of multigraph is shown in the Fig. 5.1.12.

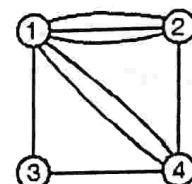


Fig. 5.1.12 : A multigraph

5.1.15 Tree

A tree is a connected graph without any cycle. The graphs of Fig. 5.1.13 are not trees as they contain cycles. The graphs of Fig. 5.1.14 are example of trees.

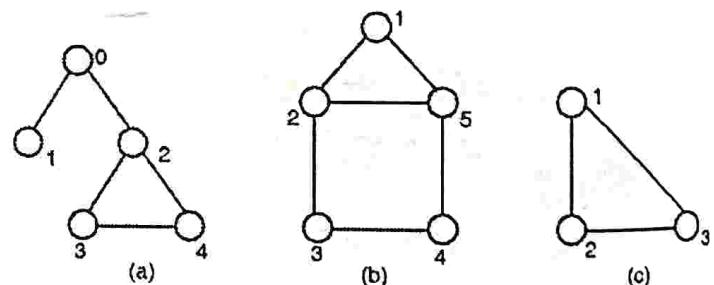


Fig. 5.1.13 : Graphs are not trees

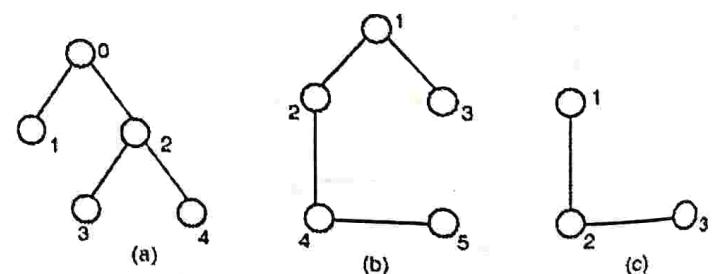


Fig. 5.1.14 : Example of trees

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.

5.1.16 Spanning Trees

A spanning tree of a graph $G = (V, E)$ is a 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. 5.1.16 gives some of the spanning trees of the graph shown in Fig. 5.1.15.

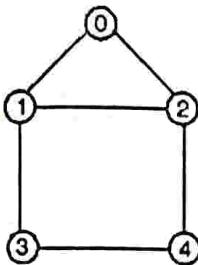


Fig. 5.1.15 : A sample connected graph

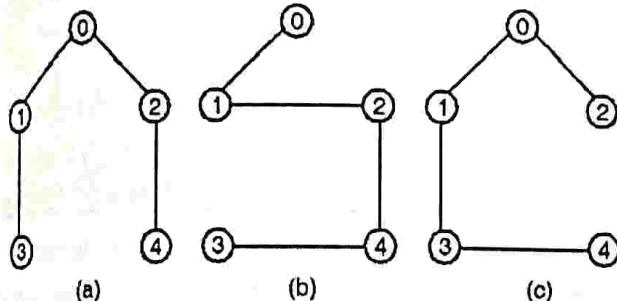


Fig. 5.1.16 : Spanning trees of the graph of Fig. 5.1.15

5.1.17 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.

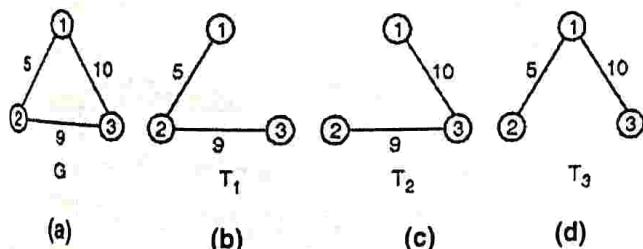


Fig. 5.1.17 : An example of minimal spanning tree

$G \rightarrow$ A sample weighted graph

$T_1 \rightarrow$ A spanning tree of G with cost $5 + 9 = 14$

$T_2 \rightarrow$ A spanning tree of G with cost $10 + 9 = 19$

$T_3 \rightarrow$ 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 .

5.2 Representation of Graphs

5.2.1 Adjacency Matrix

MU - Dec. 13, Dec. 14, May 15, Dec. 16, May 17,

May 18, Dec. 18, Dec. 19

University Questions

- Q. Explain methods to represent a graph. (May 15, May 18, 5 Marks)
- Q. Explain various techniques of graph representation. (Dec. 13, Dec. 16, 5 Marks)
- Q. Explain with examples different techniques to represent the graph data structure on a computer. Give 'C' language representations for the same. (Dec. 14, 5 Marks)
- Q. What are the various techniques/ways to represent Graph in memory ? (May 17, Dec. 19, 5 Marks)
- Q. What are different ways of representing a Graph data structure on a computer. (Dec. 18, 5 Marks)

- A two dimensional matrix can be used to store a graph.

A graph $G = (V, E)$ where $V = \{0, 1, 2, \dots n - 1\}$ can be represented using a two dimensional integer array of size $n \times n$.

`int adj[20][20];` can be used to store a graph with 20 vertices.

$\text{adj}[i][j] = 1$, 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
 $\text{adj}[i][j] = 1$, indicates a directed edge from vertex i to vertex j .

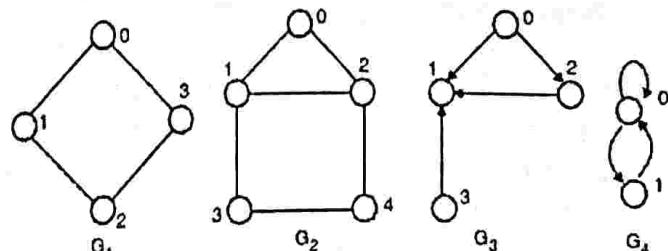


Fig. 5.2.1 : Graphs G_1 , G_2 , G_3 and G_4



| | 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)

| | 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)

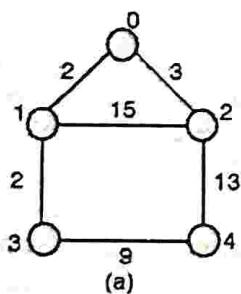
Fig. 5.2.2 : Adjacency matrix representation of graphs G_1 , G_2 , G_3 and G_4 of Fig. 5.2.1

Adjacency matrix representation of a weighted graph

For weighted graph, the matrix $\text{adj}[][]$ is represented as :

If there is an edge between vertices i and j then $\text{Adj}[i][j] = \text{weight of the edge}(i, j)$ otherwise, $\text{Adj}[i][j] = 0$

Fig. 5.2.3 is an example of representation of a weighted graph.



| | 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. 5.2.3 : 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($\text{adj}[i][j] == 1$)

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.

5.2.2 Adjacency List

MU - Dec. 13, Dec. 14, Dec. 16, May 17

University Questions

- Q. Explain various techniques of graph representation. (Dec. 13, Dec. 16, 5 Marks)
- Q. Explain with examples different techniques to represent the graph data structure on a computer. Give 'C' language representations for the same. (Dec. 14, 5 Marks)
- Q. What are the various techniques to represent Graph in memory ? (May 17, 5 Marks)

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 $G = (V, E)$

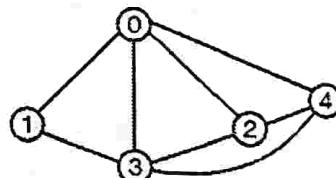
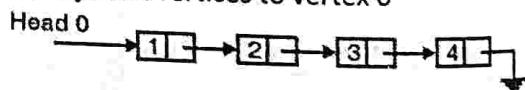
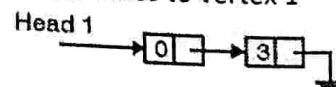


Fig. 5.2.4 : A graph

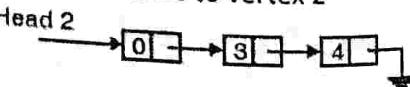
List of adjacent vertices to vertex 0



List of adjacent vertices to vertex 1



List of adjacent vertices to vertex 2



List of adjacent vertices to vertex 3

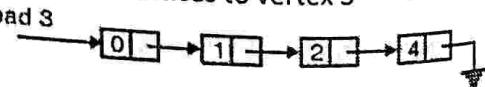


Fig. 5.2.5(Continued...)

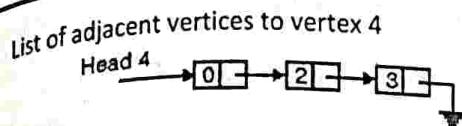


Fig. 5.2.5 : Adjacency list for each vertex of graph of Fig. 5.2.4

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. 5.2.6 shows the adjacency list representation of graph of Fig. 5.2.4.

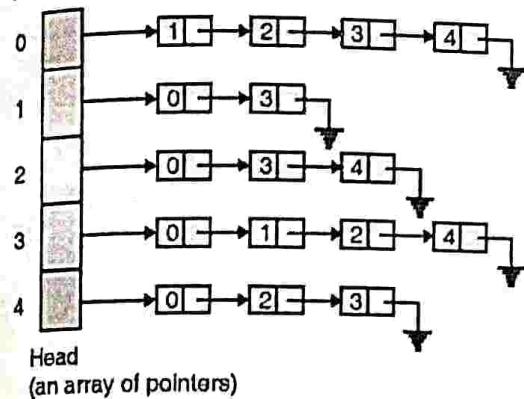


Fig. 5.2.6 : Adjacency list representation of the graph of Fig. 5.2.4

- 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 in-degree 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 follows :

```
typedef struct node
{
    struct node * next;
    int vertex;
    int weight;
}node;
```

Example 5.2.1 : For the diagraph given below, obtain

- (1) Indegree and outdegree of each vertex.
- (2) Its adjacency matrix.
- (3) Its adjacency list representation.

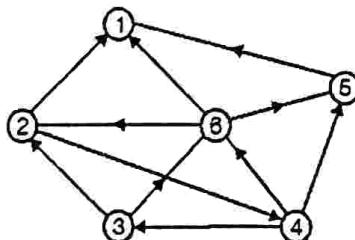


Fig. Ex. 5.2.1

Solution :

| Vertex No. | Indegree | Outdegree |
|------------|----------|-----------|
| 1 | 3 | 0 |
| 2 | 1 | 2 |
| 3 | 1 | 2 |
| 4 | 1 | 3 |
| 5 | 2 | 1 |
| 6 | 2 | 2 |

**Adjacency matrix :**

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 1 | 0 | 0 | 1 | 0 | 0 |
| 3 | 0 | 1 | 0 | 0 | 0 | 1 |
| 4 | 0 | 0 | 1 | 0 | 1 | 1 |
| 5 | 1 | 0 | 0 | 0 | 0 | 0 |
| 6 | 1 | 0 | 0 | 0 | 1 | 0 |

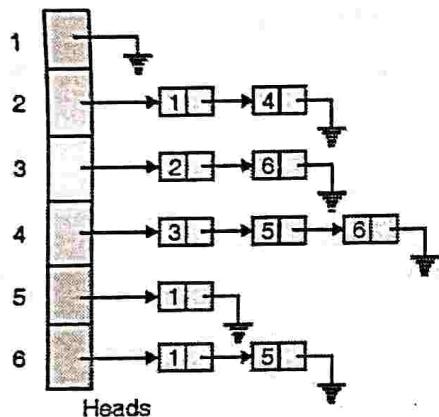
Adjacency list representation :

Fig. Ex. 5.2.1(a)

Example 5.2.2 : Construct adjacency matrix and adjacency list for the following graph.

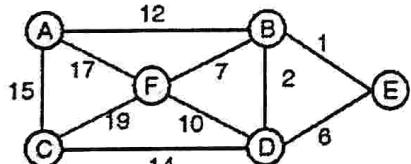


Fig. Ex. 5.2.2

Solution :**Adjacency matrix representation :**

| | A | B | C | D | E | F |
|---|----|----|----|----|---|----|
| A | 0 | 12 | 15 | 0 | 0 | 17 |
| B | 12 | 0 | 0 | 2 | 1 | 7 |
| C | 15 | 0 | 0 | 14 | 0 | 19 |
| D | 0 | 2 | 14 | 0 | 6 | 10 |
| E | 0 | 1 | 0 | 6 | 0 | 0 |
| F | 17 | 7 | 19 | 10 | 0 | 0 |

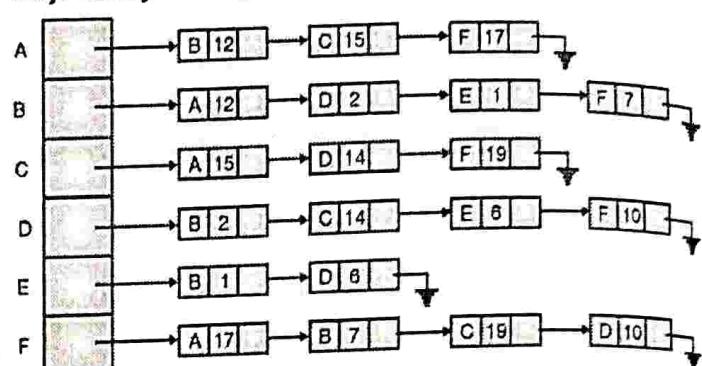
Adjacency list representation :

Fig. Ex. 5.2.2(a)

Example 5.2.3 : Give the adjacency matrix and adjacency list of the following graph :

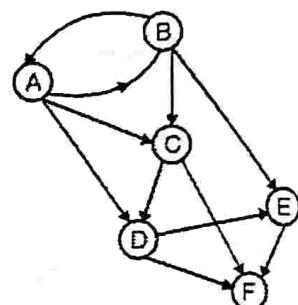


Fig. Ex. 5.2.3

Solution :**Adjacency matrix :**

| | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| A | 0 | 1 | 1 | 1 | 0 | 0 |
| B | 1 | 0 | 1 | 0 | 1 | 0 |
| C | 0 | 0 | 0 | 1 | 0 | 1 |
| D | 0 | 0 | 0 | 0 | 1 | 1 |
| E | 0 | 0 | 0 | 0 | 0 | 1 |
| F | 0 | 0 | 0 | 0 | 0 | 0 |

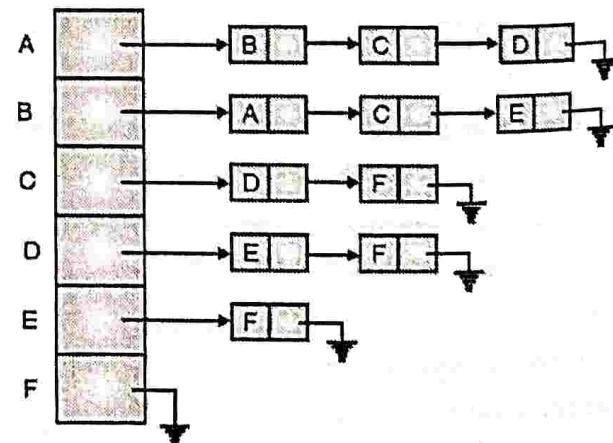
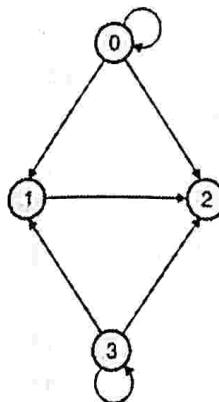
Adjacency list representation :

Fig. Ex. 5.2.3(a)

Example 5.2.4 : Represent following graph using adjacency matrix and adjacency list.



Solution : Adjacency matrix :

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

Adjacency list representation:

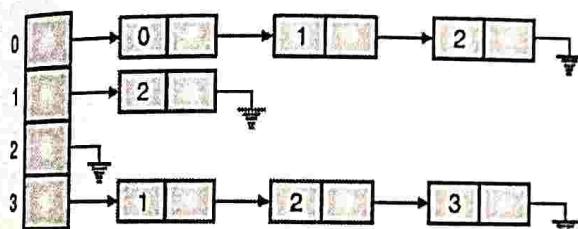


Fig. Ex. 5.2.4

Example 5.2.5 : Write adjacency list for the graph given

Fig. Ex. 5.2.5.

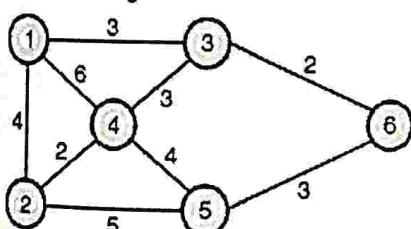
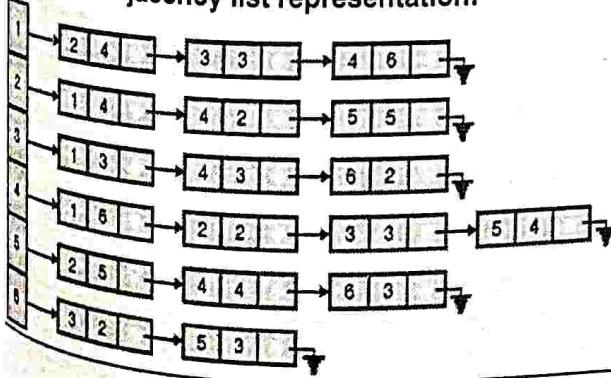


Fig. Ex. 5.2.5

Solution : Adjacency list representation:



Program 5.2.1 : Program to compute the indegree and outdegree 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 i'th column
    // outdegree if node i = no. of 1's in i'th 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 \t indegree = %d \t outdegree = %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 5.2.2 : 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("\n Node No = %d \t indegree = %d \t
               outdegree = %d", i, i_degree, o_degree);
    }
}
```

```
void read_graph()
{
    int i, vi, vj, no_of_edges;
    printf("\n Enter 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("\n Enter no. of edges : ");
        scanf("%d", &no_of_edges);
        for(i = 0; i < no_of_edges; i++)
        {
            printf("\n Enter 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
    {
        // goto 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

```

Program 5.2.3 : Write a function to create GRAPH using adjacency matrix method.

```

int n;                      //no. of vertices
int adj[10][10];           //adjacency matrix
void create()
{
    int i, u, v, nl, j, w;
    printf("\n Enter no. of vertices : ");
    scanf("%d", &n);
    printf("\n Enter no. of edges : ");
    scanf("%d", &nl);
    //initialize the adjacency matrix
    for(i = 0; i < n; i++)
        for(j = 0; j < n; j++)
            adj[i][j] = 0;
    //read edges
    printf("\n Enter edge as(u, v, w): ");
    for(i = 0; i < nl; i++)
    {
        scanf("%d%d%d", &u, &v, &w);
        adj[u][v] = adj[v][u] = w;
    }
}

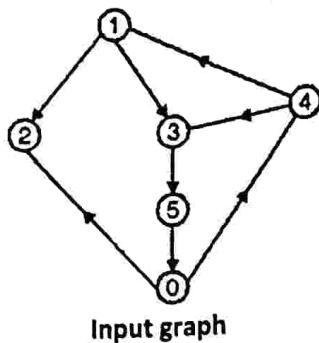
```

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 i^{th} linked list and the "vertex" field of the node is set to V_j .

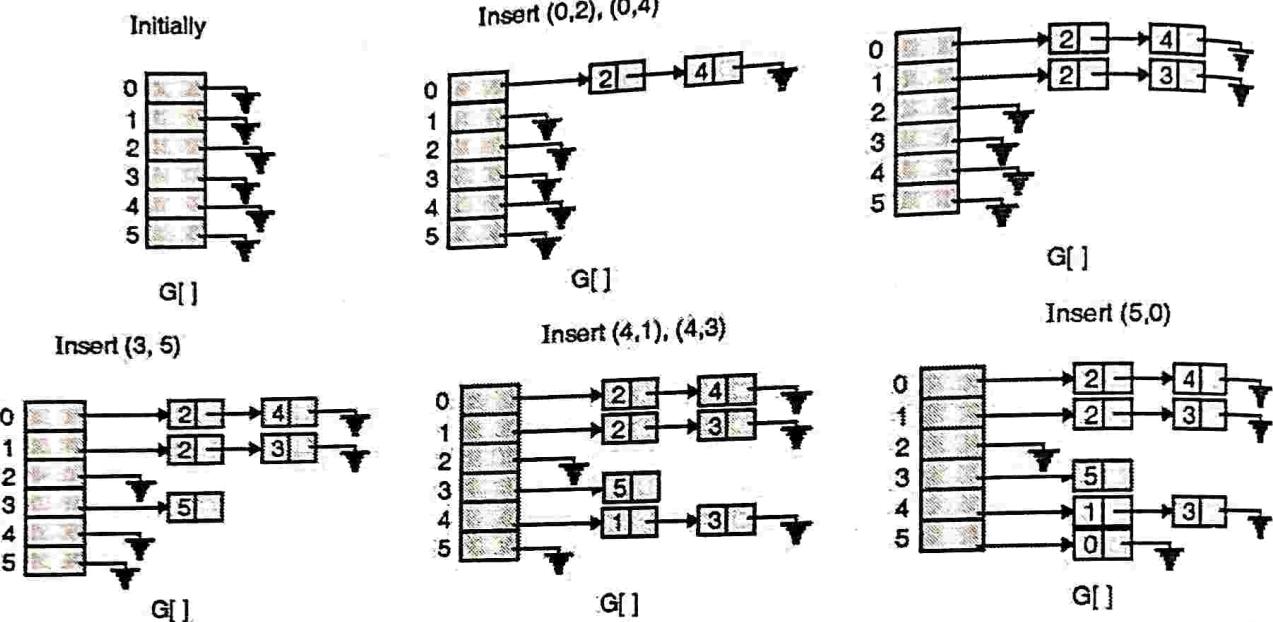
Working of readgraph() is shown as follows :



List of edges :



(0, 4), (0, 2)
(1, 2), (1, 3)
(3, 5)
(4, 1), (4, 3)
(5, 0)



Example 5.2.6 : Write a function to print indegree, outdegree and total degree of given vertex.

5.2.3 Path Matrix

MU - Dec. 13, Dec. 14, Dec. 16, May 17

University Questions

- Q. Explain various techniques of graph representations. (Dec. 13, Dec. 16, 5 Marks)
- Q. Explain with examples different techniques to represent the graph data structure on a computer. Give 'C' language representations for the same. (Dec. 14, 5 Marks)
- Q. What are the various techniques to represent Graph in memory? (May 17, 5 Marks)

A path matrix for a graph $G = (V, E)$ with n vertices is defined given below.

A path matrix is an $n \times n$ matrix whose elements are given by

$$a_{ij} = \begin{cases} 1, & \text{if there is path from } V_i \text{ to } V_j \\ 0, & \text{otherwise} \end{cases}$$

```
void details(int G[][10], int n, int v)
{
    //v is the given vertex
    int indegree = 0, outdegree = 0;
    for(i = 0; i < n; i++)
    {
        if(G[v][i] > 0)
            outdegree++;
        if(G[i][v] > 0)
            indegree++;
    }
    printf("\n Indegree = %d\n outdegree = %d",
          indegree, outdegree);
}
```

Example : A path matrix for graph given in Fig. 5.2.7 is given in Fig. 5.2.8.

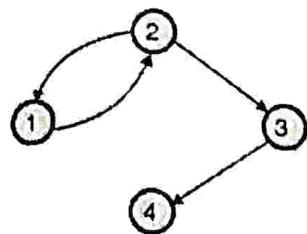


Fig. 5.2.7 : A sample graph

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 |
| 2 | 1 | 1 | 1 | 1 |
| 3 | 0 | 0 | 0 | 1 |
| 4 | 0 | 0 | 0 | 0 |

Fig. 5.2.8 : Path matrix for graph appearing in Fig. 5.2.7

- Path matrix can be found by calculating A^2, A^3, \dots, A^n and then adding them together. The matrix A is the adjacency matrix.

$$\text{If } B = A + A^2 + A^3 + \dots + A^n$$

then the path matrix P can be obtained from B by replacing each non-zero element by 1.

Example 5.2.7 : Is each of the following graphs strongly connected.

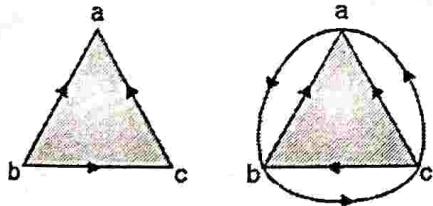


Fig. Ex. 5.2.7

Solution :

- First graph is not strongly connected. There is no path from the vertex 'a' to vertex 'c'.
- Second graph is strongly connected.

5.3 Traversal of Graphs

MU - Dec. 17

University Question

- Q. Write short note on Graph Traversal Techniques
(Dec. 17, 10 Marks)

Most of graph problems involve traversal of a graph. Traversal of a graph means visiting each node and visiting exactly once. Two commonly used techniques are :

1. Depth First Search (DFS)
2. Breadth First Search (BFS)

5.3.1 Depth First Search (DFS)

MU - Dec. 13, Dec. 14, Dec. 15, May 18, May 19

University Questions

- Explain various graph traversal techniques with examples. (Dec. 13, May 18, 8 Marks)
- Write a function for DFS traversal of graph. Explain its working with an example. (Dec. 14, 5 Marks)
- What are different methods for traversing the graph. Explain DFS in detail with an example. Write a function for DFS. (Dec. 15, 10 Marks)
- Explain Depth First Search(DFS) Traversal with an example. (May 19, 5 Marks)

It is like preorder traversal of tree. Traversal can start from any vertex, say V_i . V_i is visited and then all vertices adjacent to V_i are traversed recursively using DFS.

DFS($G, 1$) is given by

- Visit(1)
 - DFS($G, 2$)
 - DFS($G, 3$)
 - DFS($G, 4$)
 - DFS($G, 5$)
- } All nodes adjacent to 1

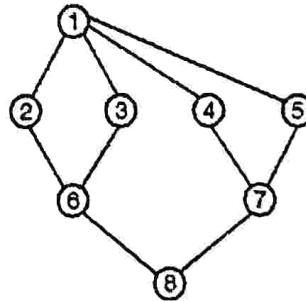


Fig. 5.3.1 : Graph G

Since, a graph can have cycles. We must avoid re-visiting a node. To do this, when we visit a vertex V , we mark it visited. A node that has already been marked as visited, should not be selected for traversal. Marking of visited vertices can be done with the help of a global array $\text{visited}[]$. Array $\text{visited}[]$ is initialized to false(0).



5.3.1(A) Algorithm for Depth First Search (Recursive)

```

n ← number of nodes
(i) Initialize visited[ ] to false(0)
    for(i = 0; i < n; i++)
        visited[i] = 0;
(ii) void DFS(vertex i) [DFS starting from i]
    {
        visited[i] = 1;
        for each w adjacent to i
            if(!visited[w])
                DFS(w); //Recursive call to DFS from an adjacent node.
    }
}

```

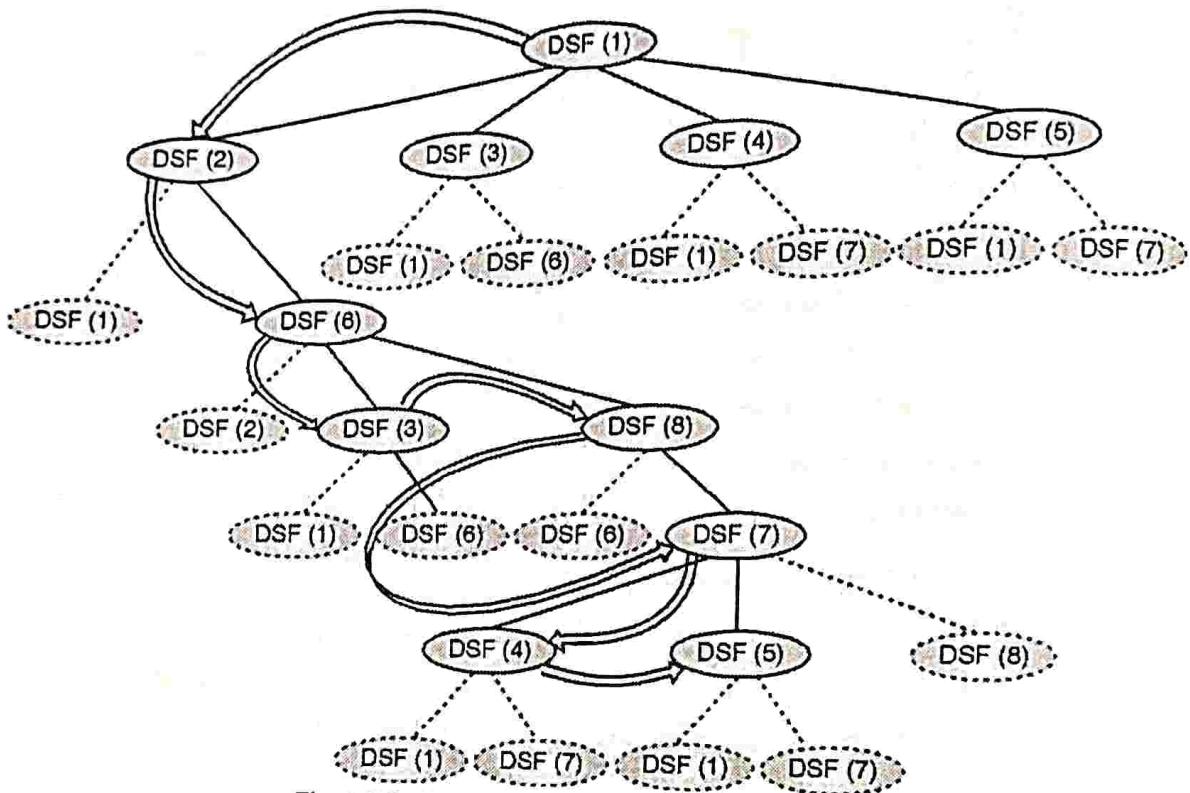


Fig. 5.3.2 : DFS traversal on graph of Fig. 5.3.1

DFS traversal on graph of Fig. 5.3.1.

DSF(i) Node i can be used for recursive traversal using DFS().

DSF(i) Node i is already visited

- Traversals start from vertex 1. Vertices 2, 3, 4 and 5 are adjacent to vertex 1. Vertex 1 is marked as visited.

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|-----------|---|---|---|---|---|---|---|
| Visited → | 1 | 0 | 0 | 0 | 0 | 0 | 0 |

- Out of the adjacent vertices 2, 3, 4 and 5, vertex number 2 is selected for further traversal. Vertices 1 and 6 are adjacent to vertex 2. Vertex 2 is marked as visited.

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|-----------|---|---|---|---|---|---|---|
| Visited → | 1 | 1 | 0 | 0 | 0 | 0 | 0 |

- Out of the adjacent vertices 1 and 6, vertex 1 has already been visited. Vertex number 6 is selected for further traversal. Vertices 2, 3 and 8 are adjacent to vertex 6. Vertex 6 is marked as visited.

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|-----------|---|---|---|---|---|---|---|
| Visited → | 1 | 1 | 0 | 0 | 0 | 1 | 0 |

- Out of the adjacent vertices 2, 3 and 8, vertex 2 is already visited. Vertex number 3 is used for further expansion. Vertices 1 and 6 are adjacent to vertex 3. Vertex 3 is marked as visited.

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|-----------|---|---|---|---|---|---|---|
| Visited → | 1 | 1 | 1 | 0 | 0 | 1 | 0 |

- Vertices 1 and 6 are already visited, therefore it goes back to vertex 6. (Vertex 6 is predecessor of vertex 3 in DFS() sequence).

- Out of the adjacent vertices 2, 3 and 8, 2 and 3 are visited. It selects vertex 8 for further expansion. Vertices 6 and 7 are adjacent to vertex 8. Vertex 8 is marked as visited.

| | | | | | | | |
|-----------|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| Visited → | 1 | 1 | 1 | 0 | 0 | 1 | 0 |

- Out of the adjacent vertices 6 and 7, vertex 6 is already visited. Vertex number 7 is used for further expansion. Vertices 4, 5 and 8 are adjacent to vertex number 7. Vertex 7 is marked as visited.

| | | | | | | | |
|-----------|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| Visited → | 1 | 1 | 1 | 0 | 0 | 1 | 1 |

- Out of the adjacent vertices 4, 5 and 8, vertex 4 is selected for further expansion. Vertices 1 and 7 are adjacent to vertex 4. Vertex 4 is marked as visited.

| | | | | | | | |
|-----------|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| Visited → | 1 | 1 | 1 | 1 | 0 | 1 | 1 |

- Adjacent vertices 1 and 7 are already visited. It goes back to vertex 7 and selects the next unvisited node 5 for further expansion. Vertex 5 is marked as visited.

| | | | | | | | |
|-----------|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| Visited → | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

- DFS traversal sequence → 1, 2, 6, 3, 8, 7, 4, 5

Program 5.3.1 : Program to implement DFS traversal on a graph represented using an adjacency matrix.

OR

Write the recursive function for DFS

MU - May 19, 5 Marks

```
#include<conio.h>
#include<stdio.h>
void DFS(int);
int G[10][10], visited[10], n;
// n->no. of vertices, graph is stored in array G[10][10]
void main()
{
    int i, j;
    printf("\n Enter no. of vertices: ");
    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]);
    // visited is initialise to zero
    for(i = 0; i < n; i++)
        visited[i] = 0;
}
```

```
visited[i] = 0;
DFS(0);
}
void DFS(int i)
{
    int j;
    printf("\n %d", i);
    visited[i] = 1;
    for(j = 0; j < n; j++)
        if(!visited[j] && G[i][j] == 1)
            DFS(j);
}
```

Output :

Enter no. of vertices : 8

Enter adjacency matrix of the graph : 0 1 1 1 1 0 0 0

1 0 0 0 0 1 0 0

1 0 0 0 0 1 0 0

1 0 0 0 0 0 1 0

1 0 0 0 0 0 1 0

0 1 1 0 0 0 0 1

0 0 1 1 0 0 1

0 0 0 0 1 1 0 0

0

1

5

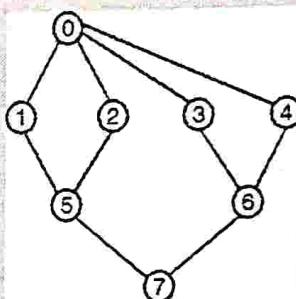
2

7

6

3

4



Graph used for input

Program 5.3.2 : Program to implement DFS traversal on a graph represented using an adjacency list.

```
#include<conio.h>
#include<stdio.h>
typedef struct node
{
    struct node *next;
    int vertex;
}node;
node *G[20];           // heads of linked list
int visited[20], n;
void read_graph();     //create adjacency list
void insert(int, int);
/* insert an edge(vi, vj) in the adjacency list */
void DFS(int);
void main()
{
    int i;
    read_graph(); // initialised visited to 0
    for(i = 0; i < n; i++)
        for(j = 0; j < n; j++)
            if(G[i] == NULL || G[i] -> vertex != j)
                insert(i, j);
}
```



```

visited[i] = 0;
DFS(0);
}

void DFS(int i)
{
    node *p;
    printf("\n %d", i);
    p = G[i];
    visited[i] = 1;
    while(p != NULL)
    {
        i = p->vertex;
        if(!visited[i])
            DFS(i);
        p = p->next;
    }
}

void read_graph()
{
    int i, vi, vj, no_of_edges;
    printf("\n Enter 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("\n Enter no of edges : ");
        scanf("%d", &no_of_edges);
        for(i = 0; i < no_of_edges; i++)
        {
            printf("\n Enter 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 vj in the linked list for vi
    if(G[vi] == NULL)
        G[vi] = q;
    else
    {
        // goto end of linked list
        p = G[vi];
        while(p->next != NULL)
            p = p->next;
        p->next = q;
    }
}

```

```

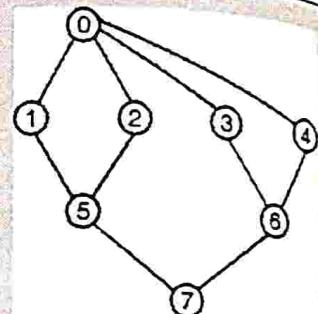
    }
}
```

Output :

```

Enter no. of vertices : 8
Enter no. of edges : 10
Enter an edge(u, v) : 0 1
Enter an edge(u, v) : 0 2
Enter an edge(u, v) : 0 3
Enter an edge(u, v) : 0 4
Enter an edge(u, v) : 1 5
Enter an edge(u, v) : 2 5
Enter an edge(u, v) : 3 6
Enter an edge(u, v) : 4 6
Enter an edge(u, v) : 5 7
Enter an edge(u, v) : 6 7

```



Graph used for input

```

0
1
5
2
7
6
3
4
```

5.3.1(B) Non-Recursive DFS Traversal

Non-recursive DFS, uses a stack to remove recursion. All unvisited vertices adjacent to the one being visited are pushed onto a stack. Traversal is continued by popping a vertex from the stack.

Algorithm :

```
DFS_Nonrecursive(vertex i)
```

```

{
    vertex w;
    stack S;
    initialize the stack S;
    push i in the stack S;
    while(stack is not empty)
    {
        i = pop(S);
        if(!visited[i])
        {
            visited[i] = 1;
            for each w adjacent to i
                if(!visited[w])
                    push w in the stack S;
        }
    }
}
```

Example 5.3.1 : Show the working of non-recursive DFS algorithm on the following graph.

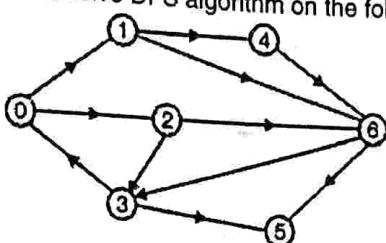


Fig. Ex. 5.3.1

Solution :

| Stack contents | Visited[] | Vertex visited | Action |
|------------------|--------------------------------|---------------------|--|
| NULL | 0 1 2 3 4 5 6 0 0 0 0 0 0 0 | - | initial |
| ↓ top 0 | 0 1 2 3 4 5 6 0 0 0 0 0 0 0 | - | push the initial vertex |
| ↓ top 1 2 | 0 1 2 3 4 5 6 1 0 0 0 0 0 0 | 0 | pop(), visit(), push adjacent vertices |
| ↓ top 1 3 6 | 0 1 2 3 4 5 6 1 0 1 0 0 0 0 | 0, 2 | pop(), visit(), push adjacent vertices |
| ↓ top 1 3 3 5 | 0 1 2 3 4 5 6 1 0 1 0 0 0 1 | 0, 2, 6 | pop(), visit(), push adjacent vertices |
| ↓ top 1 3 3 | 0 1 2 3 4 5 6 1 0 1 0 0 1 1 | 0, 2, 6, 5 | pop(), visit() |
| ↓ top 1 3 | 0 1 2 3 4 5 6 1 0 1 1 0 1 1 | 0, 2, 6, 5, 3 | pop() |
| ↓ top 1 | 0 1 2 3 4 5 6 1 0 1 1 0 1 1 | 0, 2, 6, 5, 3 | pop(), visit(), push adjacent vertices |
| ↓ 4 | 0 1 2 3 4 5 6 1 1 1 1 0 1 1 | 0, 2, 6, 5, 3, 1 | visit |
| NULL | 0 1 2 3 4 5 6 1 1 1 1 1 1 1 | 0, 2, 6, 5, 3, 1, 4 | complete |



5.3.2 Breadth First Search(BFS)

MU - Dec. 13, May 18

University Question

Q. Explain various graph traversal techniques with examples.
(Dec. 13, May 18, 8 Marks)

It is another popular approach used for visiting the vertices of a graph. This method starts from a given vertex V_0 . V_0 is marked as visited. All vertices adjacent to V_0 are visited next. Let the vertices adjacent to V_0 are $V_{10}, V_{11}, V_{12} \dots V_{1n}$, $V_{11}, V_{12} \dots V_{1n}$ and they are marked as visited. All unvisited vertices adjacent to $V_{11}, V_{12} \dots V_{1n}$ are visited next. The method continues until all vertices are visited. The algorithm for BFS has to maintain a list of vertices which have been visited but not explored for adjacent vertices. The vertices which have been visited but not explored for adjacent vertices can be stored in queue.

- Initially the queue contains the starting vertex.
- In every iteration, a vertex is removed from the queue and its adjacent vertices which are not visited as yet are added to the queue.
- The algorithm terminates when the queue becomes empty.

Fig. 5.3.3 gives the BFS sequence on various graphs.

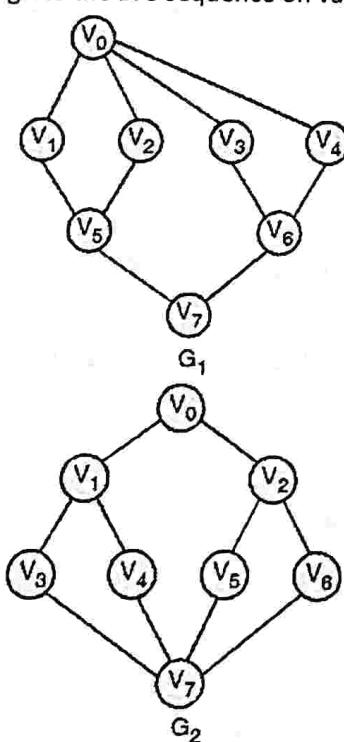
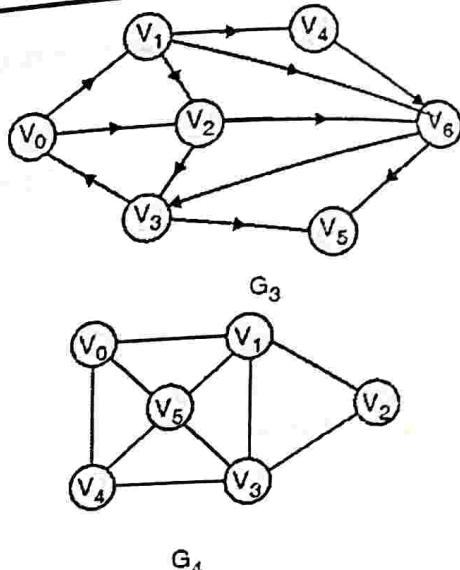


Fig. 5.3.3(Contd...)



BFS sequence :

$G_1 \rightarrow V_0 | V_1 V_2 V_3 V_4 | V_5 V_6 | V_7$

$G_2 \rightarrow V_0 | V_1 V_2 | V_3 V_4 V_5 V_6 | V_7$

$G_3 \rightarrow V_0 | V_1 V_2 | V_4 V_6 V_3 | V_5 V_6$

$G_4 \rightarrow V_0 | V_1 V_4 V_5 | V_2 V_3$

Fig. 5.3.3 : BFS traversal on G_1, G_2, G_3 and G_4

5.3.2(A) Algorithm for BFS

MU - May 14

University Question

Q. Explain BFS algorithm with examples.

(May 14, 5 Marks)

```
/* Array visited[ ] is initialize to 0 . BFS traversal on the
graph G is carried out beginning at vertex V */
void BFS(int V)
{
    q : a queue type variable;
    initialize q;
    visited[v] = 1; /* Mark v as visited */
    add the vertex V to queue q;
    while(q is not empty)
    {
        v ← delete an element from the queue;
        for all vertices w adjacent from V
        {
            if(!visited[w])
            {
                visited[w] = 1;
                add the vertex w to queue q;
            }
        }
    }
}
```

Example 5.3.2 : Show the working of BFS algorithm on the following graph.

MU - May 14, May 16, Dec. 18, 5 Marks

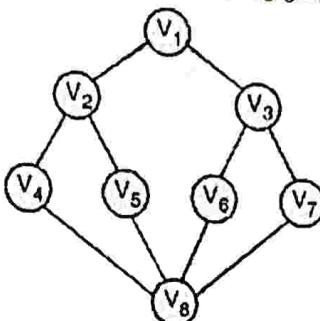


Fig. Ex. 5.3.2 : A sample graph

Solution :

| Queue | Visited[] | Vertex visited | Action |
|---|------------------------------------|---|--|
| NULL | 1 2 3 4 5 6 7 8 0 0 0 0 0 0 0 0 | - | - |
| V ₁ | 1 2 3 4 5 6 7 8 1 0 0 0 0 0 0 0 | V ₁ | add(q, V ₁) visit(V ₁) |
| V ₂ V ₃ | 1 2 3 4 5 6 7 8 1 1 1 0 0 0 0 0 | V ₁ V ₂ V ₃ | delete(q), add and visit adjacent vertices |
| V ₃ V ₄ V ₅ | 1 2 3 4 5 6 7 8 1 1 1 1 1 0 0 0 | V ₁ V ₂ V ₃ V ₄ V ₅ | delete(q), add and visit adjacent vertices |
| V ₄ V ₅ V ₆ V ₇ | 1 2 3 4 5 6 7 8 1 1 1 1 1 1 1 0 | V ₁ V ₂ V ₃ V ₄ V ₅ V ₆ V ₇ | delete(q), add and visit adjacent vertices |
| V ₅ V ₆ V ₇ V ₈ | 1 2 3 4 5 6 7 8 1 1 1 1 1 1 1 1 | V ₁ V ₂ V ₃ V ₄ V ₅ V ₆ V ₇ V ₈ | delete(q), add and visit adjacent vertices |
| V ₆ V ₇ V ₈ | 1 2 3 4 5 6 7 8 1 1 1 1 1 1 1 1 | V ₁ V ₂ V ₃ V ₄ V ₅ V ₆ V ₇ V ₈ | delete(q) |
| V ₇ V ₈ | 1 2 3 4 5 6 7 8 1 1 1 1 1 1 1 1 | V ₁ V ₂ V ₃ V ₄ V ₅ V ₆ V ₇ V ₈ | delete(q), add and visit adjacent vertices |
| V ₈ | 1 2 3 4 5 6 7 8 1 1 1 1 1 1 1 1 | V ₁ V ₂ V ₃ V ₄ V ₅ V ₆ V ₇ V ₈ | delete(q) |
| NULL | 1 2 3 4 5 6 7 8 1 1 1 1 1 1 1 1 | V ₁ V ₂ V ₃ V ₄ V ₅ V ₆ V ₇ V ₈ | algorithm terminates as the queue is empty |



Program 5.3.3 : Program to Implement BFS traversal on a graph represented using adjacency matrix.

MU - May 16, Dec. 16, Dec. 18, 10 Marks

OR
Write a program In C to implement the BFS traversal of a graph.

MU - May 16, Dec. 18, 5 Marks

OR
Write a function for BFS traversal of graph. **MU - Dec. 16, 10 Marks**

```
#include<conio.h>
#include<stdio.h>
#define MAX 10
typedef struct Q
{
    int R, F;
    int data[MAX];
}Q;
int empty(Q *P);
int full(Q *P);
void enqueue(Q *P, int x);
int dequeue(Q *P);
void BFS(int);
int G[MAX][MAX];
int n;
void main()
{
    int i, j, v;
    printf("\nEnter no. of vertices : ");
    scanf("%d", &n);
    printf("\nEnter the adjacency matrix of graph : ");
    for(i = 0; i < n; i++)
        for(j = 0; j < n; j++)
            scanf("%d", &G[i][j]);
    printf("\nEnter the starting vertex for BFS");
    scanf("%d", &v);
    BFS(v);
    getch();
}
void BFS(int v)
{
    int visited[MAX], i;
    Q q;
    q.R = q.F = -1; // initialize the queue
    for(i = 0; i < n; i++)
        visited[i] = 0;
    enqueue(&q, v);
    printf("\nvisit\t%d", v);
```

```
visited[v] = 1;
while(!empty(&q))
{
    v = dequeue(&q);
    // visit and add adjacent vertices
    for(i = 0; i < n; i++)
        if(visited[i] == 0 && G[v][i] != 0)
    {
        enqueue(&q, i);
        visited[i] = 1;
        printf("\n visit\t%d", i);
    }
}
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;
        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);
}

```

Output :

Enter no. of vertices : 8

Enter the adjacency matrix of graph :

```

0 1 1 1 1 0 0 0
1 0 0 0 0 1 0 0
1 0 0 0 0 1 0 0
1 0 0 0 0 0 1 0
1 0 0 0 0 0 1 0
0 1 1 0 0 0 0 1
0 0 0 1 1 0 0 1
0 0 0 0 1 1 0

```

Enter the starting vertex for BFS 0

```

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

```

Program 5.3.4 : Program to implement BFS traversal on a graph implemented through adjacency list.

```

#include<conio.h>
#include<stdio.h>
#include<stdlib.h>
#define MAX 20
typedef struct Q
{
    int data[MAX];
    int R, F;
}Q;
typedef struct node
{
    struct node *next;
    int vertex;
}node;
void enqueue(Q *, int), BFS(int);

```

```

int dequeue(Q *), empty(Q *), full(Q *);
node *G[20]; //heads of the linked list
int n; // Number of nodes
void readgraph(); //Create an adjacency list
void insert(int vi, int vj);
//insert an edge(vi, vj) in adjacency list
void main()
{
    int i;
    readgraph();
    BFS(0);
}

void BFS(int v)
{
    int i, visited[MAX], w;
    Q q;
    node *p;
    q.R = q.F = -1; //initialise
    for(i = 0; i < n; i++)
        visited[i] = 0;
    enqueue(&q, v);
    printf("\n Visit\t%d", v);
    visited[v] = 1;
    while(!empty(&q))
    {
        v = dequeue(&q);
        /* insert all unvisited, adjacent vertices of v
        into queue */
        for(p = G[v]; p != NULL; p = p->next)
        {
            w = p->vertex;
            if(visited[w] == 0)
            {
                enqueue(&q, w);
                visited[w] = 1;
                printf("\n visit\t%d", w);
            }
        }
    }
    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;
        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 readgraph()
{
    int i, vi, vj, no_of_edges;
    printf("\n Enter no. of vertices :");
    scanf("%d", &n);
    //initialise G[ ] with NULL
    for(i = 0; i < n; i++)
        G[i] = NULL;
    //read edges and insert them in G[ ]
    printf("\n Enter no of edges :");
    scanf("%d", &no_of_edges);
    for(i = 0; i < no_of_edges; i++)
    {
        printf("\n Enter an edge(u, v) :");
    }
}

```

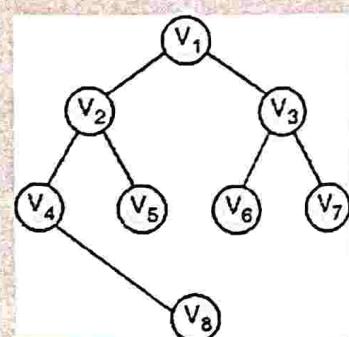
```

scanf("%d%d", &vi, &vj);
insert(vi, vj);
insert(vj, vi);
}
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 for the vertex
       number Vi */
    if(G[vi] == NULL)
        G[vi] = q;
    else
    {
        // go to the end of linked list
        p = G[vi];
        while(p->next != NULL)
            p = p->next;
        p->next = q;
    }
}

```

Output :

Enter no. of vertices : 8
 Enter no. of edges : 10
 Enter an edge(u, v) : 0 1.
 Enter an edge(u, v) : 0 2
 Enter an edge(u, v) : 0 3
 Enter an edge(u, v) : 0 4
 Enter an edge(u, v) : 1 5
 Enter an edge(u, v) : 2 5
 Enter an edge(u, v) : 3 6
 Enter an edge(u, v) : 4 6
 Enter an edge(u, v) : 5 7
 Enter an edge(u, v) : 6 7



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

Example 5.3.3 : Show DFS and BFS for the graph given in Fig. Ex. 5.3.3.

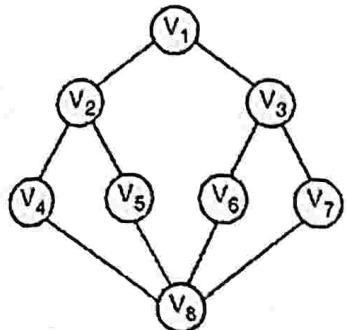


Fig. Ex. 5.3.3

Solution :

DFS traversal sequence : $V_1 V_2 V_4 V_8 V_5 V_6 V_3 V_7$

DFS spanning tree

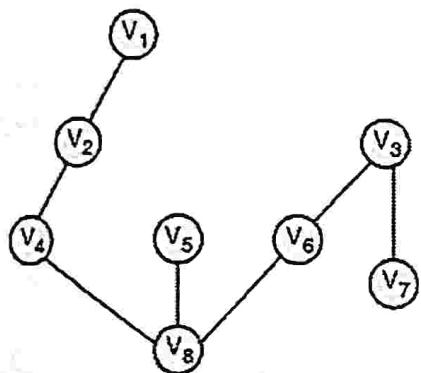


Fig. Ex. 5.3.3(a)

BFS traversal sequence :

V_1
 $V_2 V_3$
 $V_4 V_5 V_6 V_7$
 V_8

BFS spanning tree :

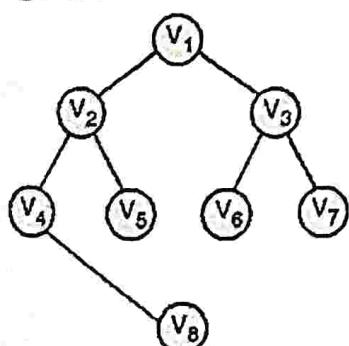


Fig. Ex. 5.3.3(b)

Example 5.3.4 : Write DFS and BFS traversal for graph given in Fig. Ex. 5.3.4.

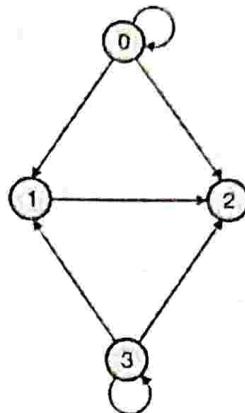
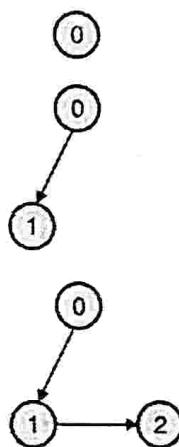


Fig. Ex. 5.3.4

Solution :

DFS traversal :



BFS traversal :

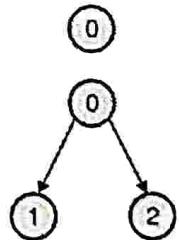


Fig. Ex. 5.3.4(a)

Example 5.3.5 : Write DFS and BFS for the graph given in Fig. Ex. 5.3.5 starting with vertex 1.

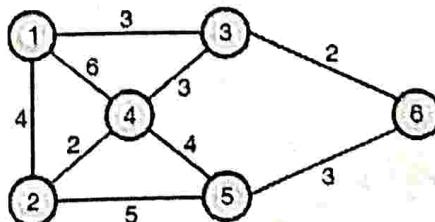


Fig. Ex. 5.3.5

**Solution :**DFS : $1 \rightarrow 2 \rightarrow 4 \rightarrow 3 \rightarrow 6 \rightarrow 5$

BFS :

1

2 3 4

5 6

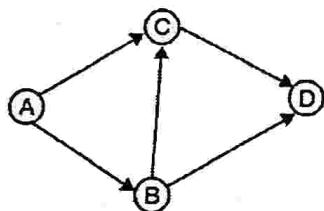
5.4 Topological Sorting**MU - Dec.17, May 19, Dec. 19****University Questions****Q. Explain Topological sorting with example.**

(Dec. 17, Dec. 19, 10 Marks)

Q. Write Short note on : Topological Sorting

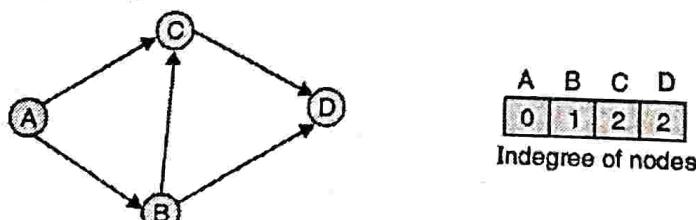
(May 19, 10 Marks)

- A topological sort is an ordering of vertices in a directed acyclic graph, such that if there is a path from V_i to V_j then V_j appears after V_i in the ordering.
- Topological sorting is important in project scheduling.

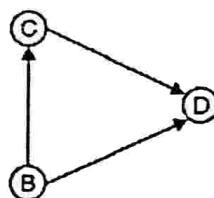
**Fig. 5.4.1**

- Four events A, B, C and D are interconnected as shown in the Fig. 5.4.1.
- Event B can be started after completion of event A.
- Event C can be started after completion of events A and B. Event D can be started after completion events C and B. Therefore, events must be scheduled as per the given sequence : A B C D.

A simple algorithm to find a topological ordering is first to find any vertex, indegree = 0. We can print this vertex, and remove it, along with edges, from the graph. Then we apply this same strategy to the rest of the graph shown below, is step wise implementation of topological sort on graph of Fig. 5.4.1.

**Fig. 5.4.2 : Initial condition**

Step 1 : Delete node A along with edges. Reduce indegree of nodes adjacent to A by 1.

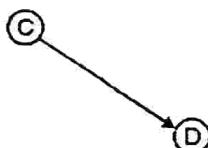


| | | |
|---|---|---|
| B | C | D |
| 0 | 1 | 2 |

Indegree

Graph after deletion of node A.

Step 2 : Delete node B along with edges. Reduce indegree of nodes adjacent to B by 1.



| | |
|---|---|
| C | D |
| 0 | 1 |

Indegree

Graph after deletion of node B.

Step 3 : Delete node C along with edges. Reduce indegree of nodes adjacent to C by 1.



| |
|---|
| 0 |
|---|

Indegree

Step 4 : Finally, Delete the only node D to complete the algorithm.

5.4.1 Program for Topological Sorting**Program 5.4.1 : Topological sorting**

```

#define INFINITY 9999
#include<iostream.h>
#include<conio.h>
#define MAX 10
class graph
{
    int G[MAX][MAX];
    int n;
public:
    graph()
    { n = 0; }
    void readgraph();
    void printgraph();
    void topological();
};
void graph::readgraph()
{
    int i, j;
  
```

```

cout << "\n Enter No. of vertices : ";
cin >> n;
cout << "\n Enter the adjacency matrix : ";
for(i = 0; i < n; i++)
    for(j = 0; j < n; j++)
        cin >> G[i][j];
}

void graph::printgraph()
{
    int i, j;
    for(i = 0; i < n; i++)
    {
        cout << "\n ";
        for(j = 0; j < n; j++)
            cout << " " << G[i][j];
    }
}

void graph::topological()
{
    int visited[MAX], indegree[MAX];
    int i, j;
    // initialize
    for(i = 0; i < n; i++)
    {
        visited[i] = 0;
        indegree[i] = 0;
        for(j = 0; j < n; j++)
            if(G[j][i] != 0)
                indegree[i]++;
    }

    cout << "\n Topological Ordering Sequence : \n ";
    for(i = 0; i < n; i++)
}

```

```

    { // locate a node with indegree == 0
        j = 0;
        while(j < n)
        {
            if(visited[j] == 0 && indegree[j] == 0)
            {
                cout << " " << j;
                // decrement the indegree of nodes adjacent to j
                visited[j] = 1;
                for(int k = 0; k < n; k++)
                    if(G[j][k] != 0)
                        indegree[k]--;
                break;
            }
            j++;
        }
        if(j == n)
        {
            cout << "\n Graph has a cycle : ";
            break;
        }
    }
}

void main()
{
    graph g;
    g.readgraph();
    g.topological();
    getch();
}

```

Example 5.4.1 : Find a topological ordering for the graph.

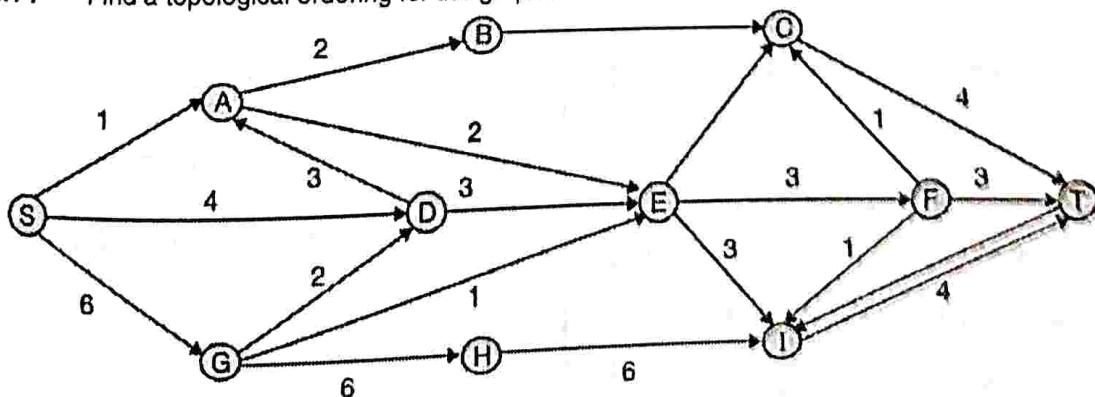


Fig. Ex. 5.4.1



Solution :

| Sr. No. | Nodes Deleted Initial condition | Indegree A B C D E F G H I S T 2 1 3 2 3 1 1 1 3 0 3 |
|---------|------------------------------------|--|
| 1. | Delete node S along with edges. | 1 1 3 1 3 1 0 1 3 - 3 |
| 2. | Delete node G along with edges. | 1 1 3 0 2 1 - 0 3 - 3 |
| 3. | Delete node D along with edges. | 0 1 3 - 1 1 - 0 3 - 3 |
| 4. | Delete node A along with edges. | - 0 3 - 0 1 - 0 3 - 3 |
| 5. | Delete node B along with edges. | - - 2 - 0 1 - 0 3 - 3 |
| 6. | Delete node E along with edges. | - - 1 - - 0 - 0 2 - 3 |
| 7. | Delete node F along with edges. | - - 0 - - - 0 1 - 2 |
| 8. | Delete node C along with edges. | - - - - - 0 1 - 1 |
| 9. | Delete node H along with edges. | - - - - - 0 - 1 |
| 10. | Delete node I along with edges. | - - - - - - 0 |
| 11. | Delete node T along with edges. | - - - - - - - |

∴ Topological ordering = S G D A B E F C H I T

Example 5.4.2 : Find a topological ordering of given graph.

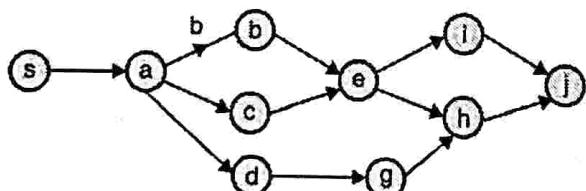


Fig. Ex. 5.4.2

Solution :

| Sr. No. | Nodes Deleted Initial condition | Indegree s a b c d e g h i j 0 1 1 1 1 2 1 2 1 2 |
|---------|------------------------------------|--|
| 1. | Delete s along with edges | - 0 1 1 1 2 1 2 1 2 |
| 2. | Delete a along with edges | - - 0 0 0 2 1 2 1 2 |
| 3. | Delete b along with edges | - - - 0 0 1 1 2 1 2 |

| Sr. No. | Nodes Deleted Initial condition | Indegree s a b c d e g h i j 0 1 1 1 1 2 1 2 1 2 |
|---------|------------------------------------|--|
| 4. | Delete c along with edges | - - - - 0 0 1 2 1 2 |
| 5. | Delete d along with edges | - - - - - 0 0 2 1 2 |
| 6. | Delete e along with edges | - - - - - - 0 1 0 2 |
| 7. | Delete g along with edges | - - - - - - - 0 0 2 |
| 8. | Delete h along with edges | - - - - - - - - 0 1 |
| 9. | Delete i along with edges | - - - - - - - - - 0 |
| 10. | Delete j along with edges | - - - - - - - - - - |

∴ Topological ordering = s a b c d e g h i j

Example 5.4.3 : Sort the digraph for topological sort.

Refer Fig. Ex. 5.4.3 below

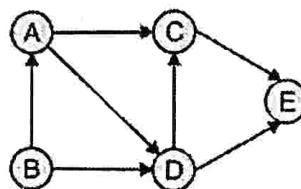
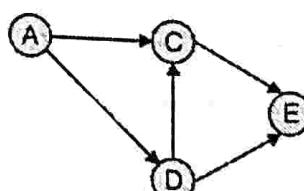


Fig. Ex. 5.4.3

Solution :

| A | B | C | D | E |
|----------|---|---|---|---|
| Indegree | 1 | 0 | 2 | 2 |

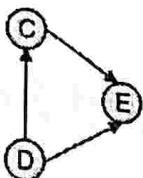
1. Delete node B along with edges B is a node of in degree 0.



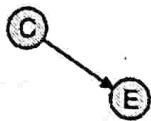
| A | C | D | E |
|----------|---|---|---|
| Indegree | 0 | 2 | 1 |

2. Delete node A along with edges. A is a node of indegree 0.

| | | | |
|----------|---|---|---|
| Indegree | C | D | E |
| | 1 | 0 | 2 |



3. Delete node D along with edges. D is a node of indegree 0.



4.

Indegree

| | |
|---|---|
| C | E |
| 0 | 1 |

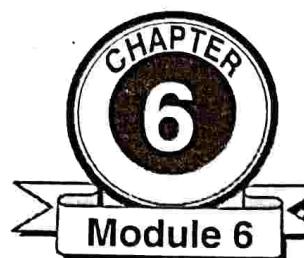
5. Delete node C.



6. Finally delete node E.

Topological sorting list = B, A, D, C, E





Sorting and Searching

Syllabus

Linear Search, Binary Search, Hashing-Concept, Hash Functions, Collision resolution Techniques.

6.1 Searching

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

- (a) Array (b) Linked List
- (c) Binary tree (d) 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.

6.2 Sequential Search

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 i^{th} element.

\therefore Expected number of comparisons for a successful search.

$$C = 1.P_1 + 2.P_2 + \dots + nP_n$$

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

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

$$\therefore C = \frac{1}{n} + \frac{2}{n} + \dots + \frac{n}{n} = \frac{1}{n} (1 + 2 + \dots + n) = \frac{n(n+1)}{2n} = \frac{n+1}{2}$$

Time complexity of sequential search = $O(n)$.

Program 6.2.1 : A sample program for sequential search.

```
#include<stdio.h>
#include<conio.h>
int sequential_search(int a[ ], int key, int n);
void main()
{
    int a[50], n, key, i;
    printf("\n No. of elements :");
    scanf("%d", &n);
    printf("\n Enter %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("\n Found 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

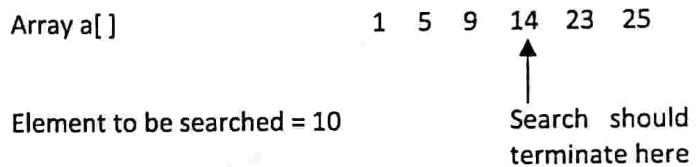
Enter 4 numbers : 66 77 88 99

element to be searched : 88

Found at location = 3

6.2.1 Sequential Search on a Sorted Array

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



- 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) ;
}
```



6.3 Binary Search

MU - Dec. 18

University Question

(Dec. 18, 5 Marks)

Q. Write a function in C to implement binary search.

- 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.

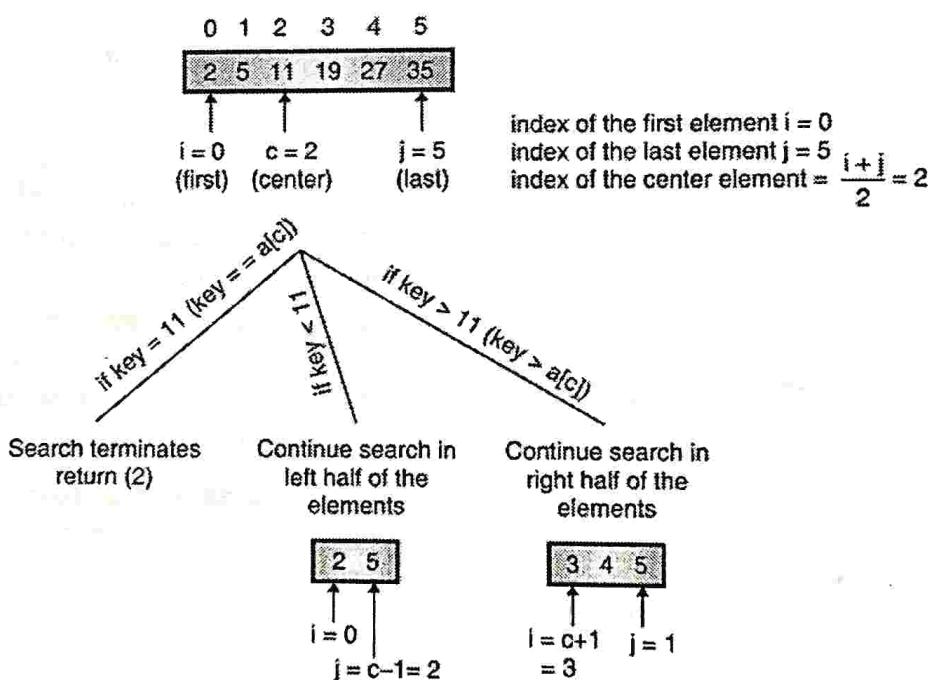


Fig. 6.3.1

- 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 6.3.1 : [Searching the element 29 in the given array]

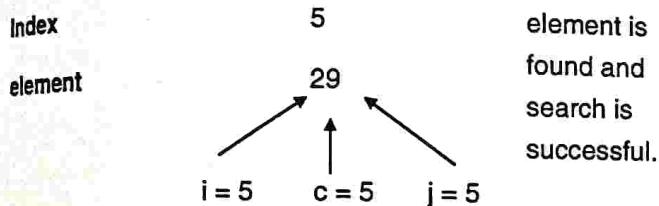
| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|----------|-------|---|----|-------|----|----|----|----|----|
| Elements | 5 | 9 | 11 | 15 | 25 | 29 | 30 | 35 | 40 |
| | i = 0 | c = $\frac{i+j}{2} = \frac{0+8}{2} = 4$ | | j = 8 | | | | | |

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.



Example 6.3.2 : [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-recursive)

```
int bin_search(int a[], int i, int j, int key)
{
    int c;
    c = (i+j)/2;
```

Sorting and Searching

```

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]

$$\therefore \frac{n}{2^h} = 1 \text{ or } 2^h = n \text{ or } h = \log_2 n = O(\log_2 n)$$

Thus, the worst case behaviors of binary search algorithm has a timing complexity = $O(\log_2 n)$

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.

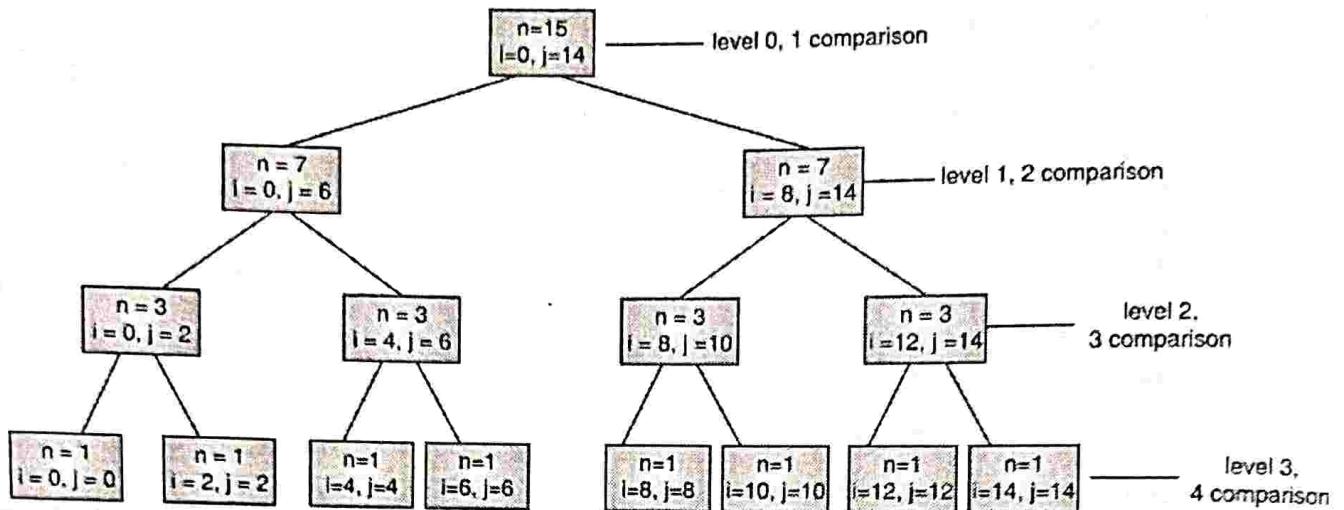


Fig. Ex. 6.3.2

- Relation between last level ($h = 3$) and number of elements ($n = 15$) is given by $2^{3+1} = 15+1$

$$\text{Or } 2^{h+1} = n+1$$

$$\text{Or } h+1 = \log_2(n+1)$$

$$\text{Or } h = [\log_2(n+1)] - 1$$

\therefore Total number of comparison to search every element of array is given by :

$$C = 1.2^0 + 2.2^1 + \dots + (h+1)2^h \quad \dots(1)$$

$$2C = 1.2^1 + 2.2^2 + \dots + h2^h + (h+1)2^{h+1} \quad \dots(2)$$

Subtracting (2) from (1)

$$-C = 1.2^0 + 2^1 + 2^2 + \dots + 2^h - (h+1)2^{h+1}$$

$$\text{or } C = (h+1)2^{h+1} - [2^0 + 2^1 + 2^2 + \dots + 2^h]$$

$$= (h+1)2^{h+1} - (-1 + 2^{h+1})$$

$$= h \cdot 2^{h+1} + 1$$

Therefore, average number of comparisons

$$= \frac{h \cdot 2^{h+1} + 1}{n}$$

$$= \frac{h \cdot 2^{h+1} + 1}{2^{h+1} - 1}$$

$$\approx h \text{ for large } n$$

$$= O \log_2(n)$$

Thus, expected number of comparisons for successful as well as unsuccessful search $O(\log_2 n)$

Analysis of binary search (Best case)

Algorithm will need just 1 comparison if the element is found at the centre.

Timing complexity = $O(1)$

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 6.3.1 :

Program to Implement binary search algorithm.

OR

Write a program to Implement binary search on sorted set of integers.

MU - Dec. 17, 10 Marks

```
#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("\n Enter number of elements :");
scanf("%d", &n);
printf("\n Enter the array elements :");
for(i = 0; i<n; i++)
    scanf("%d", &a[i]);
printf("\n Enter the element to be searched :");
scanf("%d", &key);
bubble_sort(a, n);
result = bin_search(a, key, n);
if(result == -1)
    printf("\n Element not found :");
else
    printf("\n Element 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 6.3.3 : 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 :

| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|----------|---|----|----|----|----|----|----|----|----|----|-----|
| Element | 9 | 17 | 23 | 38 | 45 | 50 | 57 | 76 | 79 | 90 | 100 |
| Position | i | | | | | k | | | | j | |

| i | j | k |
|---|----|---|
| 0 | 10 | 5 |



Step 1 : Since $10 < A[5]$, $j = k - 1 = 4$

| Index | 0 | 1 | 2 | 3 | 4 |
|----------|---|----|----|----|----|
| Element | 9 | 17 | 23 | 38 | 45 |
| Position | i | | k | | j |

| | | |
|---|---|---|
| i | j | k |
| 0 | 4 | 2 |

Step 2 : Since $10 < A[2]$, $j = k - 1 = 1$

| Index | 0 | 1 |
|----------|---|----|
| Element | 9 | 17 |
| Position | i | k |

| | | |
|---|---|---|
| i | j | k |
| 0 | 1 | 0 |

Step 3 : Since $10 > A[0]$, $j = k + 1 = 1$

| Index | 1 |
|----------|----|
| Element | 17 |
| Position | i |

| | | |
|---|---|---|
| i | j | k |
| 1 | 1 | 1 |

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

| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|----------|---|----|----|----|----|----|----|----|----|----|-----|
| Element | 9 | 17 | 23 | 38 | 45 | 50 | 57 | 76 | 79 | 90 | 100 |
| Position | i | | | | | k | | | | | j |

| | | |
|---|----|---|
| i | j | k |
| 0 | 10 | 5 |

Step 1 : Since $100 > A[5]$, $i = k + 1 = 6$

| Index | 6 | 7 | 8 | 9 | 10 |
|----------|----|----|----|----|-----|
| Element | 57 | 76 | 79 | 90 | 100 |
| Position | i | | k | | j |

| | | |
|---|----|---|
| i | j | k |
| 6 | 10 | 8 |

Step 2 : Since $100 > A[8]$, $i = k + 1 = 9$

| Index | 9 | 10 |
|----------|----|-----|
| Element | 90 | 100 |
| Position | i | k |

| | | |
|---|----|---|
| i | j | k |
| 9 | 10 | 9 |

Step 3 : Since $100 > A[9]$, $i = k + 1 = 10$

| Index | 10 |
|----------|-----|
| Element | 100 |
| Position | i |

| | | |
|----|----|----|
| i | j | k |
| 10 | 10 | 10 |

Since, the element to be searched is found at A[10], search terminates with a success.

6.4 Sorting

- Sorting is a process of ordering a list of elements in either ascending or descending order. Sorting can be divided into two categories.
 - (a) Internal sorting
 - (b) External 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.
- 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.

6.4.1 Sort Stability

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 |

(c) Sorted list (not stable)

Fig. 6.4.1

- Fig. 6.4.1(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. 6.4.1(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. 6.4.1(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

6.4.2 Sort Efficiency

- 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.

6.4.3 Passes

- 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 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.

6.5 Insertion Sort

MU - Dec. 19

Q. Write a C function to implement Insertion sort.
(Dec. 19, 5 Marks)

- 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. It 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)

| | | | | | | | | | | | | | | | | |
|---|---|----------|---|---|--------|--|----------|--|---|---|---|---|----------|--|--|--|
| <table border="1"> <tr><td>0</td><td>1</td><td>5</td><td>9</td></tr> <tr><td colspan="2">Sorted</td><td colspan="2">Unsorted</td></tr> </table> | 0 | 1 | 5 | 9 | Sorted | | Unsorted | | <table border="1"> <tr><td>2</td><td>6</td><td>4</td></tr> <tr><td colspan="2">Unsorted</td><td colspan="2"></td></tr> </table> | 2 | 6 | 4 | Unsorted | | | |
| 0 | 1 | 5 | 9 | | | | | | | | | | | | | |
| Sorted | | Unsorted | | | | | | | | | | | | | | |
| 2 | 6 | 4 | | | | | | | | | | | | | | |
| Unsorted | | | | | | | | | | | | | | | | |

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

| | | | | | | | | | | | | | | | |
|---|---|---|---|----------|---|--------|--|--|--|----------|--|---|---|----------|--|
| <table border="1"> <tr><td>0</td><td>1</td><td>2</td><td>5</td><td>9</td></tr> <tr><td colspan="4">sorted</td><td>Unsorted</td></tr> </table> | 0 | 1 | 2 | 5 | 9 | sorted | | | | Unsorted | <table border="1"> <tr><td>6</td><td>4</td></tr> <tr><td colspan="2">Unsorted</td></tr> </table> | 6 | 4 | Unsorted | |
| 0 | 1 | 2 | 5 | 9 | | | | | | | | | | | |
| sorted | | | | Unsorted | | | | | | | | | | | |
| 6 | 4 | | | | | | | | | | | | | | |
| Unsorted | | | | | | | | | | | | | | | |

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

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|----------|---|--------|--|--|--|--|----------|--|---|----------|--|
| <table border="1"> <tr><td>0</td><td>1</td><td>2</td><td>5</td><td>6</td><td>9</td></tr> <tr><td colspan="5">Sorted</td><td>Unsorted</td></tr> </table> | 0 | 1 | 2 | 5 | 6 | 9 | Sorted | | | | | Unsorted | <table border="1"> <tr><td>4</td></tr> <tr><td colspan="2">Unsorted</td></tr> </table> | 4 | Unsorted | |
| 0 | 1 | 2 | 5 | 6 | 9 | | | | | | | | | | | |
| Sorted | | | | | Unsorted | | | | | | | | | | | |
| 4 | | | | | | | | | | | | | | | | |
| Unsorted | | | | | | | | | | | | | | | | |

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

0 1 2 4 5 6 9

Fig. 6.5.1 : Sorting of elements using insertion sort

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

| | | |
|---------|----------|---------|
| Index → | 0 1 2 3 | 4 5 6 |
| 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 the 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 && temp < a[j]; j --)
            a[j + 1] = a[j];
        a[j + 1] = temp;
    }
}
```

```
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

| Elements | | | | | | | 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 = $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = O(n^2)$.

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 = $n - 1 = O(n)$ 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 6.5.1 : A sample program for insertion sort (for an array of integers)**OR** Write a program in C to implement Insertion sort.**MU - May 16, 7 Marks**

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



```

void insertion_sort(int[ ], int);
void main()
{
    int a[50], n, i;
    printf("\n Enter no. of elements :");
    scanf("%d", &n);
    printf("\n Enter array elements :");
    for(i = 0; i<n; i++)
        scanf("%d", &a[i]);
    insertion_sort(a, n);
    printf("\n Sorted 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 : 5
Enter array elements : 57 89 64 56 77 333
Sorted array is      : 56 57 64 77 89 333

```

6.5.1 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.

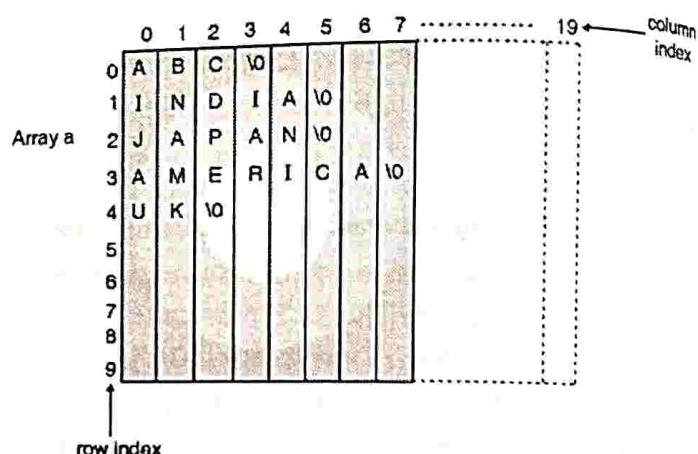
```

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 "AMERICA"

```

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.

**Fig. 6.5.2**

Program 6.5.2 : 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("\n Enter 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 : 5
Enter strings : xyz abc ccc aaa
Sorted strings : aaa abc ccc xyz

```

6.5.2 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 6.5.3 : 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;
    }
}

```

```

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

```

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 | 3 > 1, inner loop terminates |
| | | 1 3 7 2 0 | insert 3 |
| i = 3 | j = 2 | 1 3 7 7 0 | 7 > 2, move 7 right |
| | j = 1 | 1 3 3 7 0 | 3 > 2, move 2 right |
| | j = 0 | 1 3 3 7 0 | 2 > 1, inner loop terminates |
| | | 1 2 3 7 0 | insert 2 |
| i = 4 | j = 3 | 1 2 3 7 7 | 7 > 0 |
| | j = 2 | 1 2 3 3 7 | 3 > 0 |
| | j = 1 | 1 2 2 3 7 | 2 > 0 |
| | j = 0 | 1 1 2 3 7 | 1 > 0 |
| | j = -1 | 1 1 2 3 7 | j = -1, inner loop terminates |
| | | 0 1 2 3 7 | 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 i^{th} element.

$$= \frac{1}{i} [1 + 2 + 3 + \dots + i] = \frac{1}{i} \times \frac{i(i+1)}{2} = \frac{i+1}{2}$$

Since, in the outer loop i varies from 1 to $n-1$.

\therefore Total number of comparisons

$$\begin{aligned} &= \sum_{i=1}^{n-1} \frac{i+1}{2} = \frac{1}{2} \sum_{i=1}^{n-1} (i+1) \\ &= \frac{1}{2} \left[(n-1) + \frac{n(n-1)}{2} \right] \\ &= \frac{1}{2} \left[\frac{2(n-1) + n(n-1)}{2} \right] \\ &= \frac{1}{4} [(n+2)(n-1)] \\ &= \frac{1}{4} [n^2 + n - 2] \\ &= O(n^2) \end{aligned}$$

6.6 Bubble Sort

- 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.

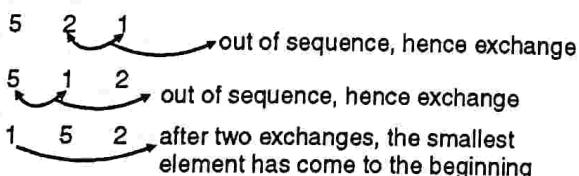


Fig. 6.6.1 : First pass of the bubble sort

- 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 $a[n-1]$.

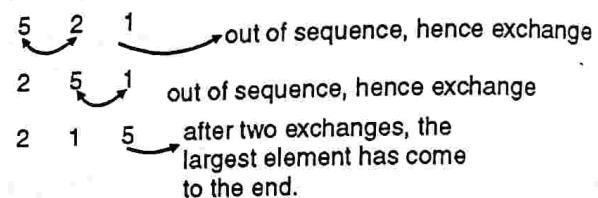


Fig. 6.6.2 : First pass of the bubble sort with comparison in the forward direction

- 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 - i; 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 $n-3$)

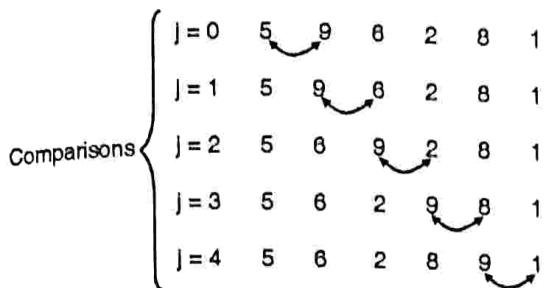
:

:

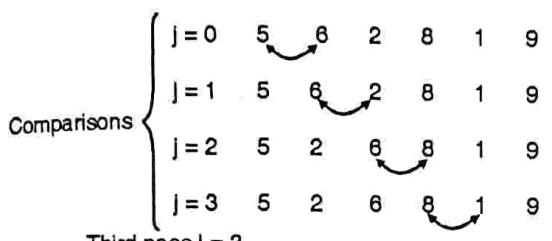
In pass $n-1$ ($i = n-1$),

there will be 1 comparison ($j = 0$ to 0)

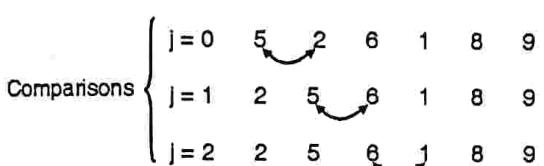
Original array with $n = 6$, Elements : 5 9 6 2 8 1
First pass $i = 1$



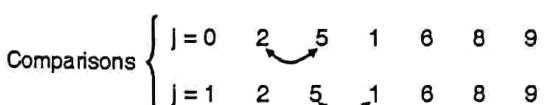
Second pass $i = 2$



Third pass $i = 3$



Fourth pass $i = 4$



Fifth pass $i = 5$



Sorted array $a[] \rightarrow 1 \quad 2 \quad 5 \quad 6 \quad 8 \quad 9$

Fig. 6.6.3 : Illustration of bubble sort

Program 6.6.1 : Program for sorting an Integer array using bubble sort.

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

```
scanf("%d", &a[i]);
bubble_sort(a, n);
printf("\n Sorted 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 : 5
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])
            {
                temp = a[j];
                a[j] = a[j+1];
                a[j+1] = temp;
                flag = 1;
            }
    }
}
```



```

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 $\text{for}(i = 1; i < n; i + 1)$

Line 2 $\text{for}(j = 0; j < n - i; j++)$

Line 3 $\text{if}(a[j] > a[j + 1])$

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

$$= \sum_{i=1}^{n-1} (n-i) = (n-1) + (n-2) + \dots + 3 + 2 + 1$$

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

6.7 Selection Sort

- Selection sort is a very simple sorting method. In the i^{th} pass, we select the element with lowest value among $a[i], a[i + 1] \dots, a[n - 1]$ 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

$\text{for}(i = 0; i < n - 1; i++)$
select the smallest element among
 $a[i], \dots, a[n - 1]$ and swap it with $a[i]$;

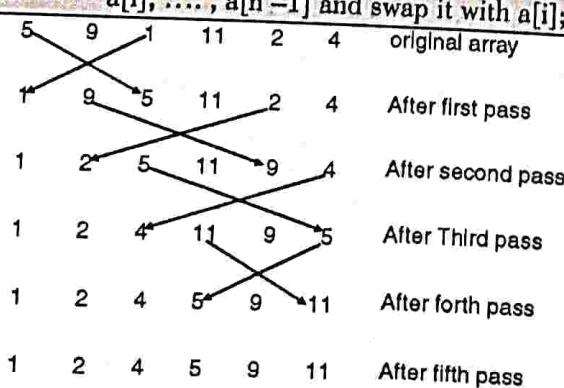


Illustration of selection sort

Fig. 6.7.1

'C' function for selection sort.

```

void selectionsort(int a[ ], int n)
{
    int i, j, k, temp;
    /* i → outer loop, j → inner loop
       k → index of the smallest element
       temp → for swapping */
    for(i = 0; i < n - 1; i++)
    {
        k = i;
        /* ith element is assumed to be the smallest */
        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;
        }
    }
}

```

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,

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

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

Example 6.7.1 : Compare the three sorting algorithms

- Bubble sort
- Selection sort
- 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.

6.8 Quick Sort

MU - Dec. 18

University Question

Q. How does the quick sort technique work?

(Dec. 18, 3 Marks)

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 $n \leq 1$, 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 $x_i \leq V$ left of V .

If the place of the V after re-arrangement is j , all elements with value less than V , appear in $a[0], a[1] \dots a[j-1]$ and all those with value greater than V appear in $a[j+1] \dots a[n-1]$.

3. Apply quick sort recursively to $a[0] \dots a[j-1]$ and to $a[j+1] \dots a[n-1]$

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

- (a) Partitioning the array around V .
- (b) Recursively, sorting the left partition.
- (c) Recursively sorting the right partition.

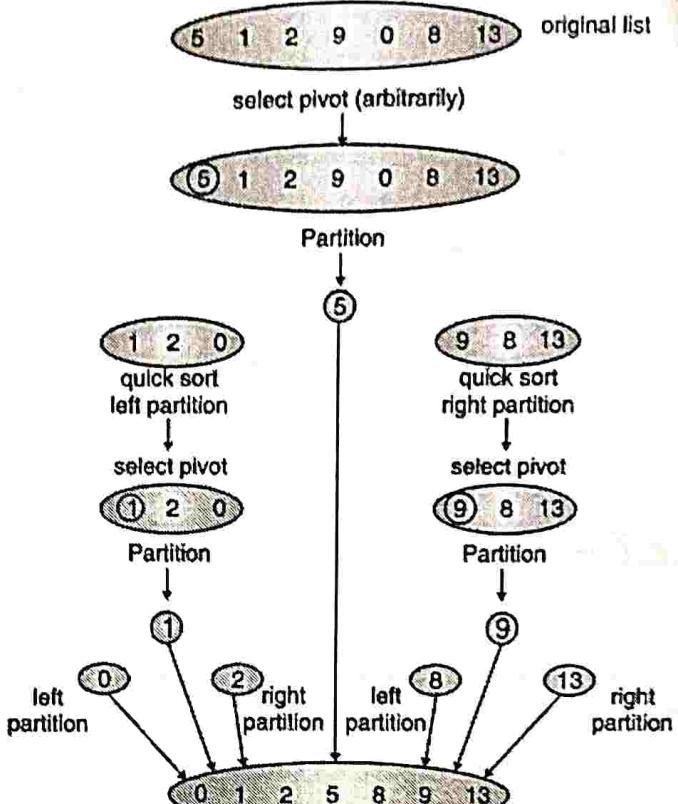


Fig. 6.8.1 : Illustration of quick sort

6.8.1 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

6.8.2 Partitioning

- Let the elements of the array $a[l], a[l+1] \dots a[n-1], a[n]$ 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.
 - $K \geq l$
 - $K \leq n$
- 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.

| | | | | | | | | | | | | |
|----|----|----|----|----|----|---|----|---|----|----|----|----|
| 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 |

$\uparrow \quad \quad \quad \downarrow$

$i=1 \quad \quad \quad j=12$

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 $a[j] > 30$.

| | | | | | | | | | | | | |
|----|----|----|----|----|----|---|----|---|----|----|----|----|
| 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 |

$\uparrow \quad \quad \quad \downarrow$

$i=1 \quad \quad \quad j=11$

interchange

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

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

$\uparrow \quad \quad \quad \downarrow$

$i=5 \quad \quad \quad j=10$

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.

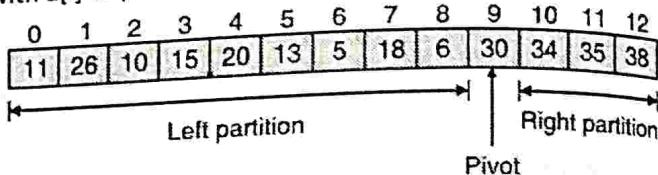
| | | | | | | | | | | | | |
|----|----|----|----|----|----|---|----|---|----|----|----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| 30 | 26 | 10 | 15 | 20 | 13 | 5 | 18 | 6 | 11 | 34 | 35 | 38 |

$\uparrow \quad \quad \quad \downarrow$

$j=9 \quad \quad \quad i=10$

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 elements $a[i]$ with $l \leq i < j$ are less than 30 and all elements $a[j]$ with $u \geq i > j$ 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 6.8.1 : Write a program in 'C' to implement quick sort.

MU - May 14, May 15, Dec. 16, May 17, 10 Marks

OR Write a program in 'C' to perform quick sort.

MU - May 14, Dec. 16, May 17, Dec.18, 10 Marks

OR Write a program to implement Quick sort.

Show the steps to sort the given numbers :

25, 13, 7, 34, 56, 23, 13, 96, 14, 2

MU - Dec.17, 10 Marks

```
#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("\n Enter no. of elements :");
    scanf("%d", &n);
    printf("\n Enter array elements :");
    for(i = 0; i < n; i++)
        scanf("%d", &a[i]);
    quick_sort(a, 0, n-1);
    printf("\n Sorted 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);

}

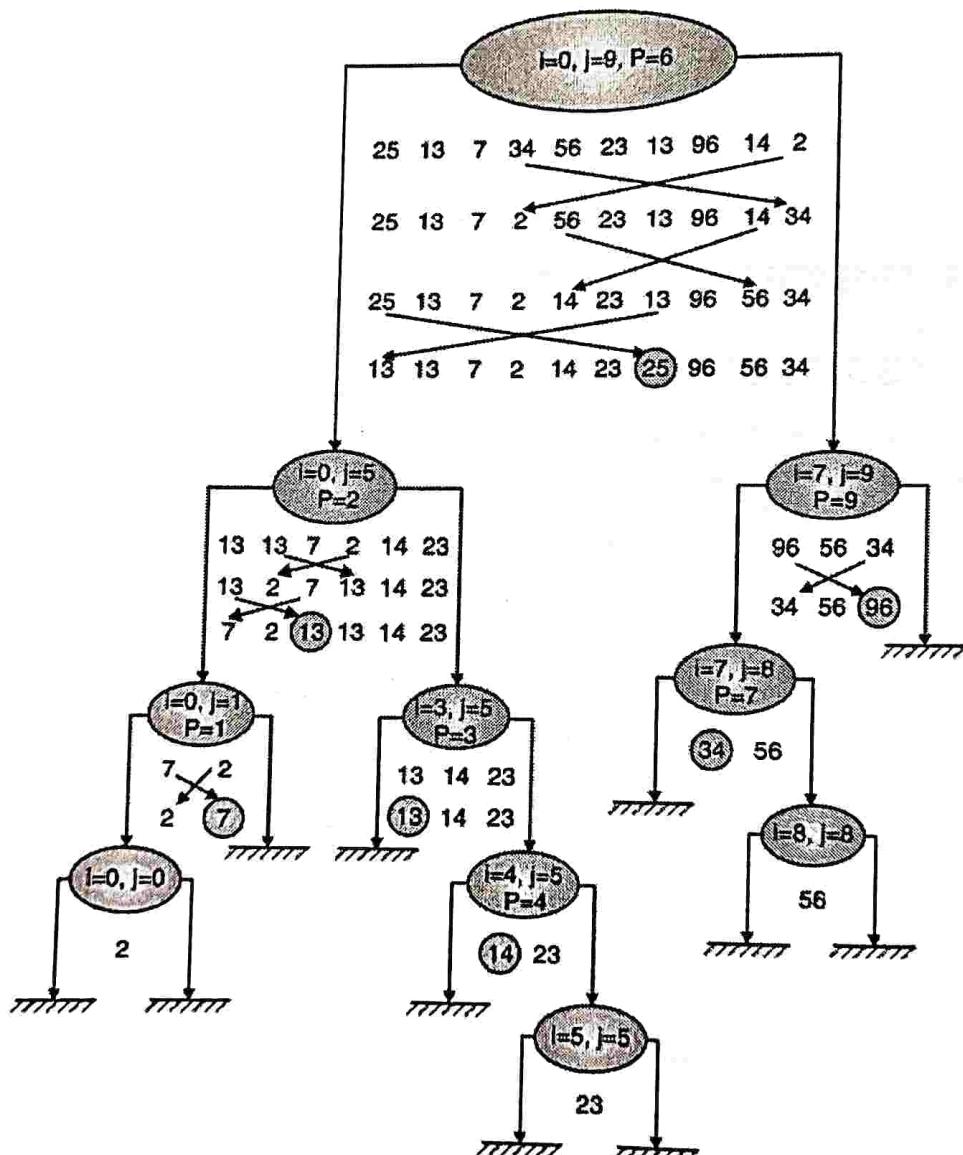
**Output**

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

Steps to sort given numbers

Given number are 25, 13, 7, 34, 56, 23, 13, 96, 14, 2.

Shown in Fig. P. 6.8.1.

**Fig. P. 6.8.1**

Sorted Data = 2 7 13 13 14 23 25 34 56 96

Example 6.8.1 : 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 :

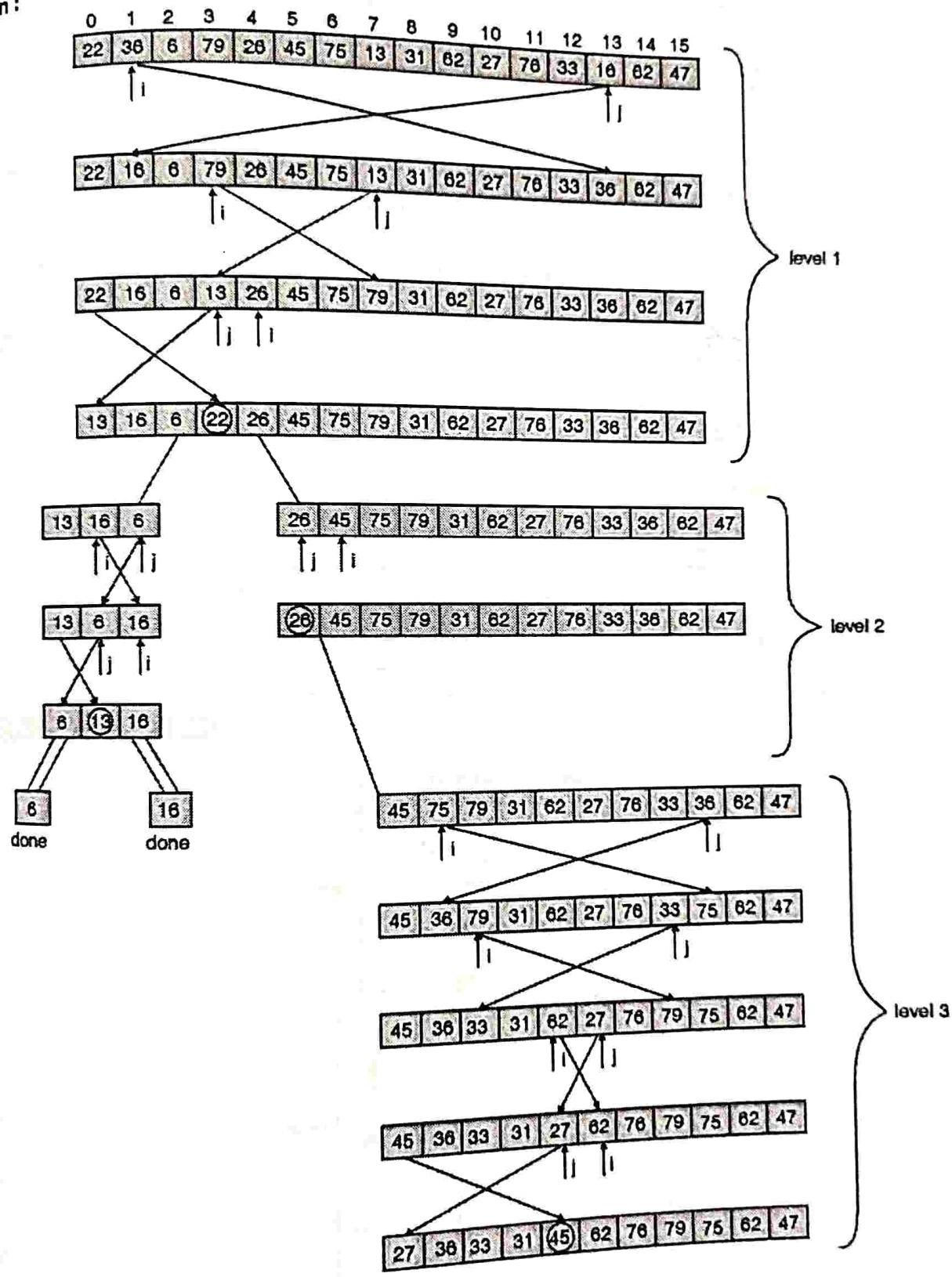


Fig. Ex. 6.8.1(a)

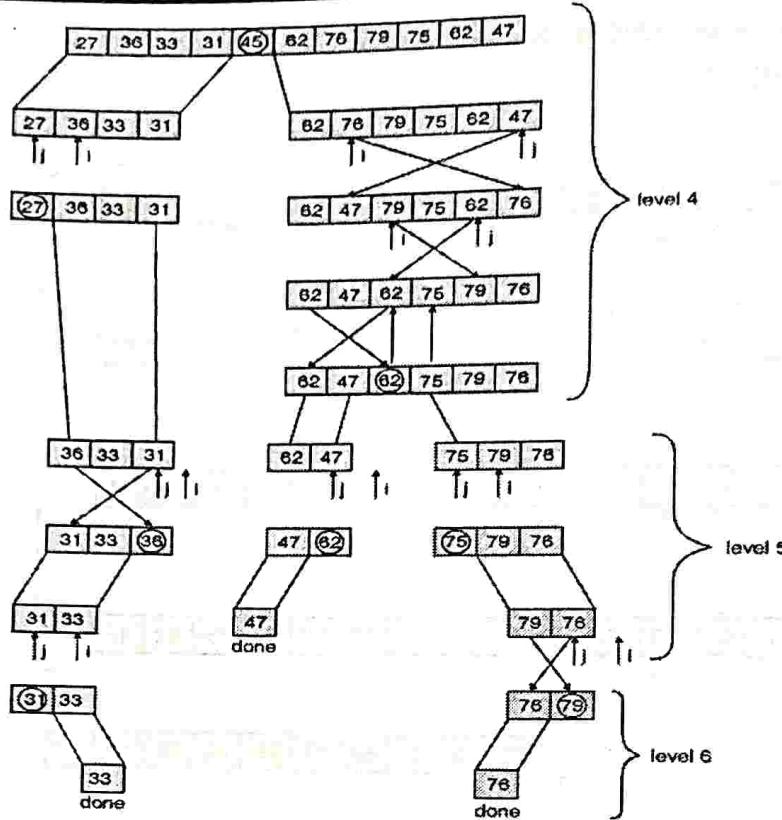


Fig. Ex. 6.8.1(b)

Note : Partition point at each level is encircled.

Example 6.8.2 : Sort the following sequence of elements using quick sort method. 24, 4, 31, 2, 58, 8, 41, 21.

Solution :

MU - May 14, Dec. 16, 5 Marks

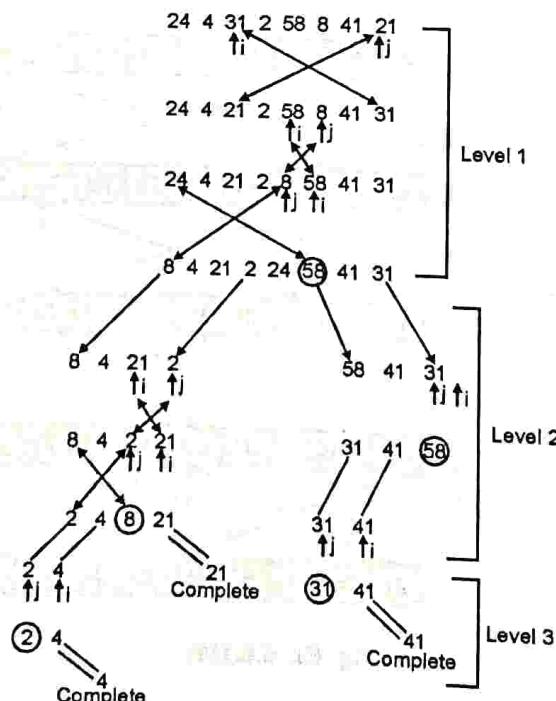


Fig. Ex. 6.8.2

Example 6.8.3 : Sort the following elements using quick sort. 27, 76, 17, 9, 57, 90, 45, 100, 79

Solution :

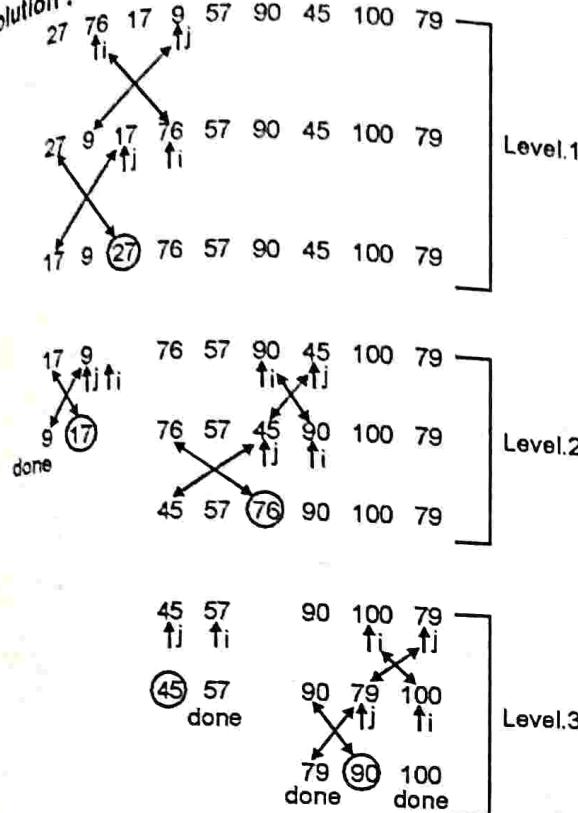


Fig. Ex. 6.8.3

Sorted data = 9 17 27 45 57 76 79 90 100

Example 6.8.4 : Sort the following list of numbers using Quick sort, show the result stepwise.
Solution :

Data after each pass

| Pass | Data | | | | | | | | |
|------|------|------|------|------|------|------|------|------|--|
| 1 | (25) | 29 | 36 | 32 | 38 | 44 | 40 | 54 | |
| 2 | (25) | (29) | 36 | 32 | 38 | 44 | 40 | 54 | |
| 3 | (25) | (29) | (32) | (36) | 38 | 44 | 40 | 54 | |
| 4 | (25) | (29) | (32) | (36) | (38) | 44 | 40 | 54 | |
| 5 | (25) | (29) | (32) | (36) | (38) | (40) | (44) | (54) | |

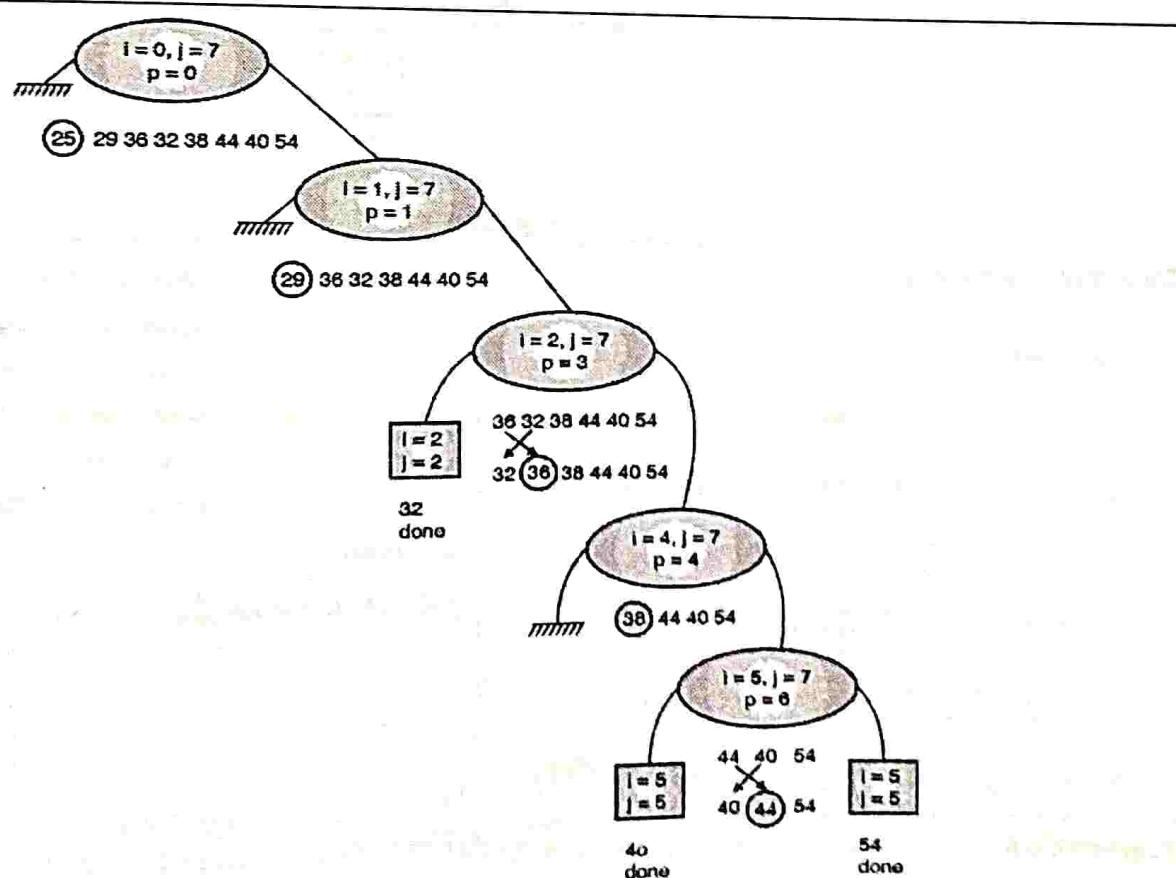


Fig. Ex. 6.8.4



Example 6.8.5 : 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 :

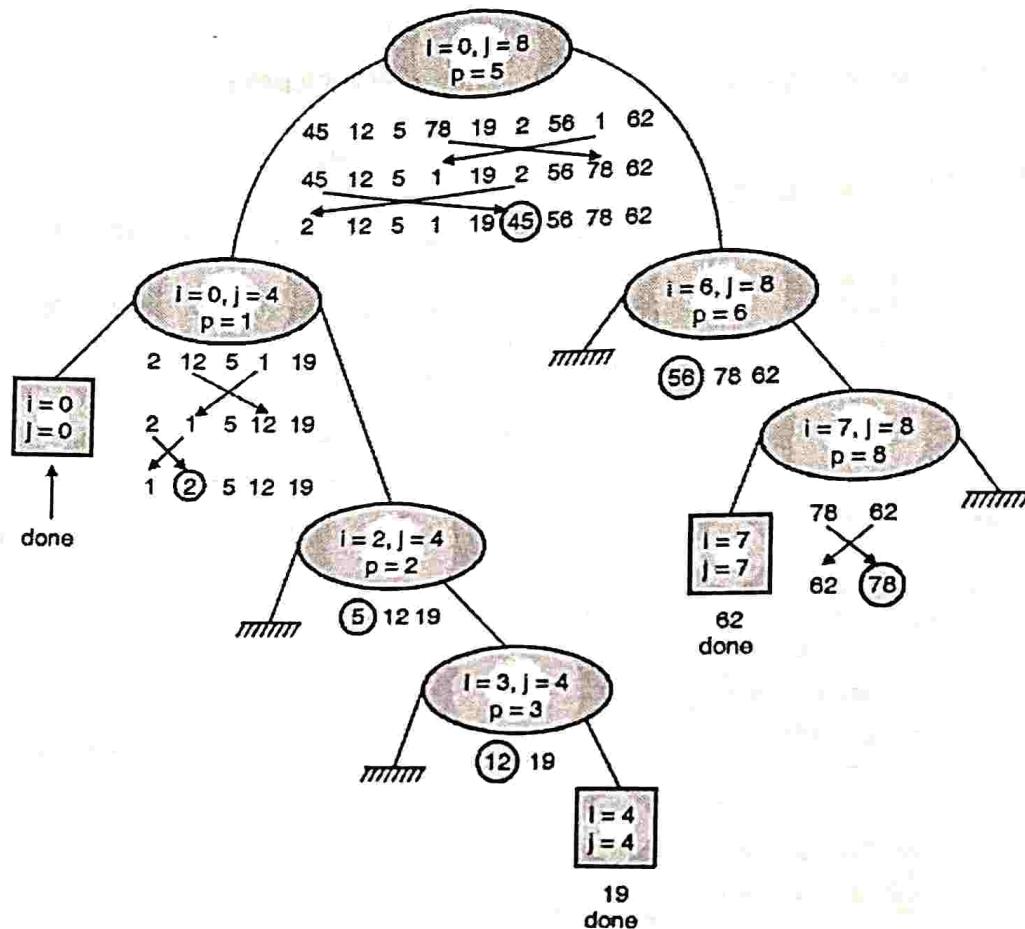


Fig. Ex. 6.8.5

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 6.8.6 : Write pseudo 'c' code to sort 'n' numbers stored in an array in non-increasing order using Quick sort.

State its time and space complexity. Apply your algorithm to sort the following numbers in non-increasing order. Display the contents of the array in each pass.

23, 9, 100, 17, 45, 57, 79, 45

Solution :

```
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;  
    i = u;  
    j = l - 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[u] = a[j];
```

```
a[j] = v;
```

```
return(j);
```

Time complexity = $O(n \log n)$

Space complexity = $O(\log n)$

1. Space required to store n numbers, which is of the order of n .
2. Space required on top of the stack to handle recursion. This could vary from 'the order of $\log n$ ' to 'the order of n '.
3. Space required to store other local variables, which is of constant order.

Sorting of data in non-increasing order.

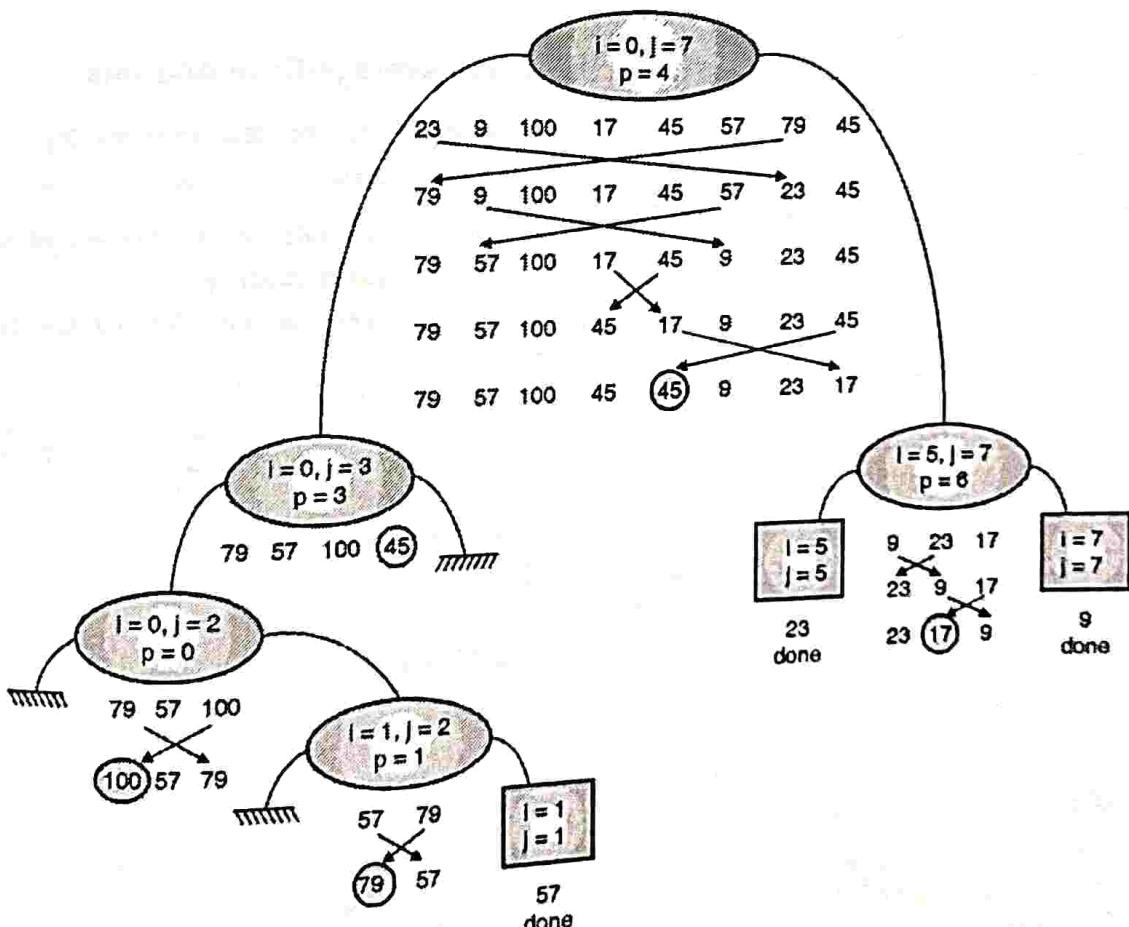


Fig. Ex. 6.8.6



Data after every pass

Pass Data

| | | | | | | | | |
|---|-----|----|-----|----|----|----|----|----|
| 1 | 79 | 57 | 100 | 45 | 45 | 9 | 23 | 17 |
| 2 | 79 | 57 | 100 | 45 | 45 | 9 | 23 | 17 |
| 3 | 100 | 57 | 79 | 45 | 45 | 9 | 23 | 17 |
| 4 | 100 | 79 | 57 | 45 | 45 | 9 | 23 | 17 |
| 5 | 100 | 79 | 57 | 45 | 45 | 23 | 17 | 9 |

6.8.3 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,

$$T(0) = 1, \quad T(1) = 1$$

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 = $T(N - j - 1)$

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

6.8.3(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 $j = 0$

$T(N) = T(0) + T(N - 1) + CN$ for $N > 1$ neglecting $T(0)$ being insignificant

$$\begin{aligned} T(N) &= T(N - 1) + CN \\ &= T(N - 2) + C(N - 1) + CN \\ &= T(N - 3) + C(N - 2) + C(N - 1) + CN \\ &\vdots \\ &\vdots \end{aligned}$$

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

6.8.3(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 $N/2^h = 1$

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

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

6.8.3(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.

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

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

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

With N written as $N - 1$, equation (2) becomes

N-2

$$(N-1)T(N-1) = 2 \sum_{j=0}^{N-1} T(j) + C(N-1)^2 \quad \dots(3)$$

If we subtract (3) from (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)

We get,

$$\begin{aligned} \frac{T(N)}{N+1} &= \frac{T(N-1)}{N} + \frac{2C}{N+1} \\ &= \frac{T(N-2)}{N-1} + \frac{2C}{N} + \frac{2C}{N+1} \\ &= \frac{T(N-3)}{N-2} + \frac{2C}{N-1} + \frac{2C}{N} + \frac{2C}{N+1} \\ &\vdots \\ &\vdots \\ &= \frac{T(1)}{2} + \frac{2C}{3} + \dots + \frac{2C}{N} + \frac{2C}{N+1} \\ &= \frac{T(1)}{2} + 2C \sum_{i=3}^{N+1} \frac{1}{i} \end{aligned}$$

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

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

$$\begin{aligned} \therefore T(N) &= \frac{T(1)}{2} (N+1) + (N+1) \log N \\ &= O(N \log N) \end{aligned}$$

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

Example 6.8.7 : Write a quick sort recursive algorithm.

Solution :

The recursive algorithm for quick sort is given below.

```
void quick_sort(int a[], int l, int u)
{
    int j;
    if(l < u)
    {
        j = partition(a, l, u); /* 1 */
        quick_sort(a, l, j-1); /* 2 */
        quick_sort(a, j+1, u); /* 3 */
    }
}
```

Sorting and Searching

Above algorithm of quick_sort() is similar to preorder traversal of a binary tree.

Line number 1 of quick_sort() can be treated as visit()

Line number 2 of quick_sort() can be treated as traversing of left sub tree.

Line number 3 of quick_sort() can be treated as traversing of right sub tree.

Left child of node (a, l, u) is (a, l, j-1) and the right child of node (a, l, u) is (a, j+1, u).

A stack can be used to remember the current node as we traverse down the tree using the left child. Stack helps in traversing right sub tree of all nodes, visited so far.

'C' function for non-recursive quick sort.

```
void quick_sort(int a[], int l, int u)
{
    stack s;
    int i, j;
    init(&s);
    while(i < u)
    {
        j = partition(a, l, u);
        push(&s, j+1);
        push(&s, u);
        u = j-1; /* goto */
    }
    while(!empty(&s))
    {
        l = pop(&s); /* goto right */
        u = pop(&s);
        while(l < u)
        {
            j = partition(a, l, u);
            push(&s, j+1);
            push(&s, u);
            u = j-1;
        }
    }
}
```

Program 6.8.8 : Program for non-recursive quick sort.

OR Write a program in 'C' to perform quick sort. Show steps with example.

MU - May 14, 10 Marks

OR Write a program in 'C' to implement Quick sort. **MU - May 19, 10 Marks**



```

#include<stdio.h>
#define MAX 20
#include<conio.h>
typedef struct stack
{
    int data[50];
    int top;
}stack;
int empty(stack *s);
void init(stack *s);
void push(stack *s, int);
int pop(stack *s);
int partition(int[], int, int);
void quick_sort(int[], int, int);
void main()
{
    int a[30], n, i;
    printf("\n No. of elements :");
    scanf("%d", &n);
    printf("Enter elements :");
    for(i = 0; i < n; i++)
        scanf("%d", &a[i]);
    quick_sort(a, 0, n-1);
    printf("\n Sorted data ->\n ");
    for(i = 0; i < n; i++)
        printf("%d", a[i]);
    getch();
}
void quick_sort(int a[], int l, int u)
{
    stack s;
    int i, j;
    init(&s);
    while(i < u)
    {
        j = partition(a, l, u);
        push(&s, j+1);
        push(&s, u);
        u = j-1;
    }
    while(!empty(&s))
    {
        l = pop(&s);
        u = pop(&s);
        while(l < u)
        {
    
```

```

            j = partition(a, l, u);
            push(&s, j+1);
            push(&s, u);
            u = j-1;
        }
    }
    void init(stack *s)
    {
        s->top = -1;
    }
    int empty(stack *s)
    {
        if(s->top == -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 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;
            }
        }
    
```

```

    a[i] = a[j];
    a[j] = temp;
}
}while(i < j);
a[i] = a[j];
a[j] = v;
return(j);
}

```

Output

Enter elements : 234 11 555 54
 Sorted data -> 11 54 234 555

6.8.4 Role of Pivot In Efficiency of Quick Sort

- A choice of pivot, always influences the efficiency of quick sort algorithm. The popular choice of pivot as the first element is acceptable when input data is random.
- If the input data is in either ascending or descending order, pivot will provide a poor partition. Quick sort will require quadratic time for sorting.
- Quick sort will give its best behaviour when partition is found near center of the input data. In such a case, quick sort will sort the data in $(n \log n)$ time.
- Median of the array is good choice for partition. Unfortunately, it is hard to calculate and would slow down quick sort.
- A good estimate can be obtained by picking three elements randomly and using the median of these three as pivot.
- As an alternative, input data can be randomised and subsequently, first element can be taken as pivot.

6.9 Two-Way Merge Sort

- Merge sort runs in $O(N \log N)$ running time. It is a very efficient sorting algorithm with near optimal number of comparisons. It is best described using a recursive algorithm.
- Basic operation in merge sort is that of merging of two sorted lists into one sorted list. Merging operation has a linear time complexity.
- Recursive algorithm used for merge sort comes under the category of divide and conquer technique. An array of n elements is split around its centre. Producing two smaller arrays.

- After these two arrays are sorted independently, they can be merged to produce the final sorted array.
- The process of splitting and merging can be carried recursively till there is only one element in the array. An array with 1 element is always sorted. Let us try to understand the method through an example.

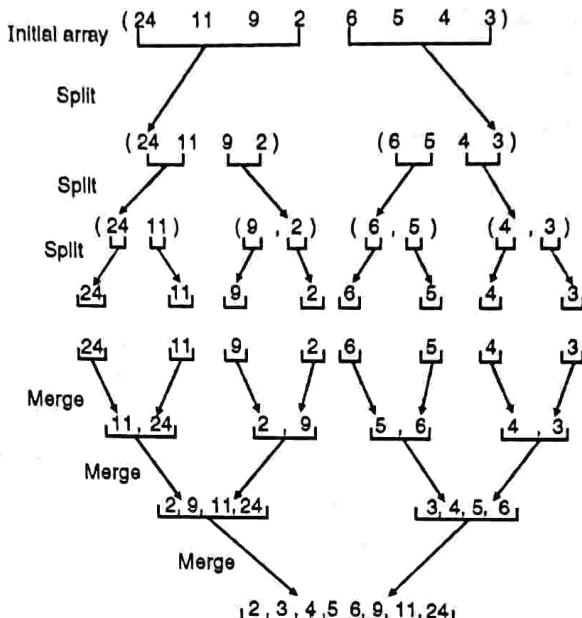


Fig. 6.9.1

- Given array has 8 elements. Index of the first element is $i = 0$, index of the last element $j = 7$. In order to divide the above list around the middle element, the index of the centre element $k = \frac{i+j}{2} = \frac{0+7}{2} = 3$.
- Merge sort is applied recursively to left half of the list from $i = 0$ to $j = 3$.
- After sorting of the left half of the list. Right half of the list is sorted from $i = 4$ to $j = 7$ recursively using merge sort.
- After both the lists are sorted, left list from $i = 0$ to $j = 3$ and right list from $i = 4$ to $j = 7$, these two lists are merged to produce a single sorted array.

'C' function for merge sort.

```

void mergesort(int a[], int i, int j)
{
    int k;
    if(i < j)
    {
        k = (i + j)/2;
        mergesort(a, i, k);
        mergesort(a, k+1, j);
        merge(a, i, k, j);
    }
}

```



Recursive function merge sort can be seen as postorder traversal of the binary tree :

Where traversal of the left subtree is given by - mergesort(a, i, k);

And traversal of the right subtree is given by - mergesort(a, k + 1, j);

Visiting a node is given by merge(a, i, k, j).

- If the root node starts with an array (24, 11, 9, 2, 6, 5, 4, 3) having eight elements. Value of i and j for the root node will be $i = 0$ and $j = 7$ respectively.
- After split, value of i and j for the left child will be $i = 0$ and $j = 3$ and for the right child it will be $i = 4$ and $j = 7$. Recursion will go on expanding until the list can no longer be divided.

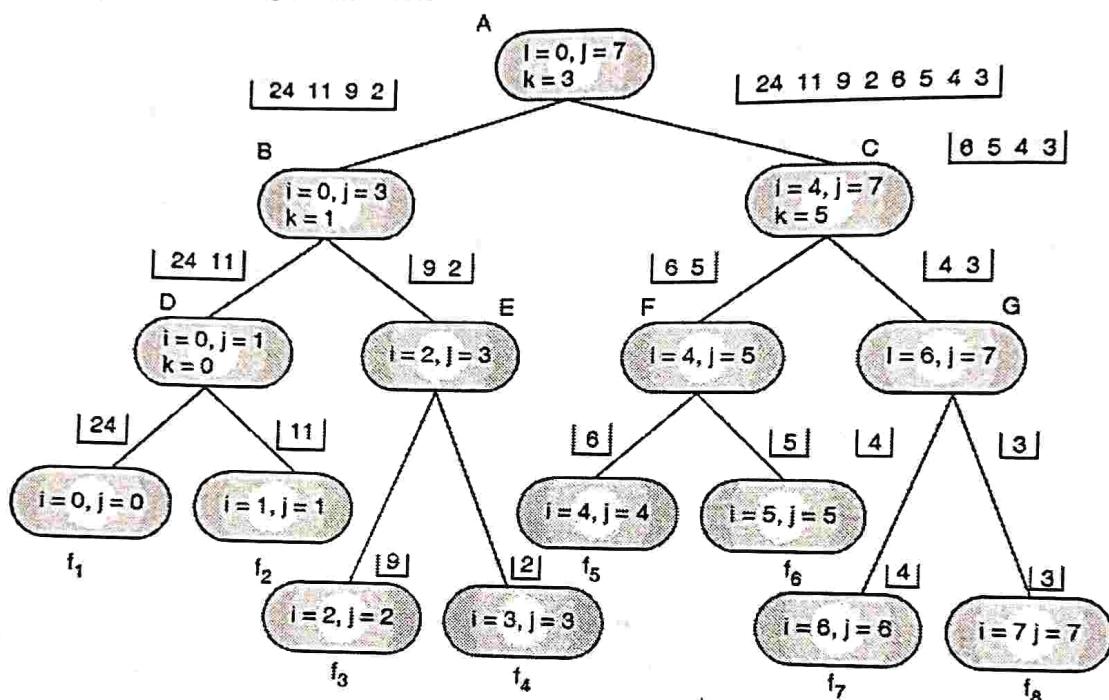


Fig. 6.9.2

Post order traversal on the recursion tree will list the nodes in the following sequence : D E B F G C A

Visit D[merge (24) (11) giving (11, 24)]

Visit E[Merge (9) (2) giving (2, 9)]

Visit B[Merge (11, 24)(2, 9) giving (2, 9, 11, 24)]

Visit F[Merge (6) (5) giving (5, 6)]

Visit G[Merge (4) (3) giving (3, 4)]

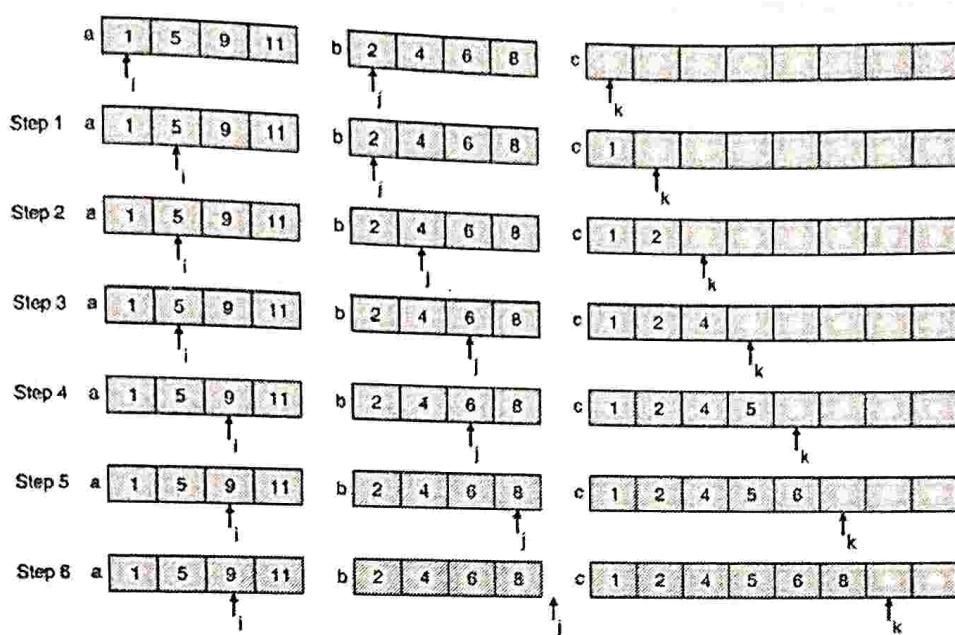
Visit C[Merge (5, 6) (3, 4) giving (3, 4, 5, 6)]

Visit A[Merge (2, 9, 11, 24), (3, 4, 5, 6) giving (2, 3, 4, 5, 6, 9, 11, 24)]

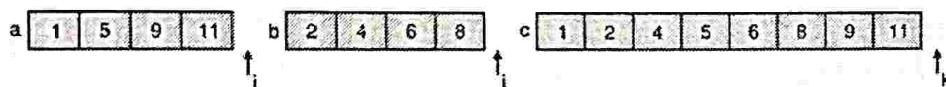
6.9.1 Merging

The fundamental operation in merge sort algorithm is merging two sorted arrays. The merging algorithm takes two sorted arrays $a[]$ and $b[]$ as input and the third array $c[]$ as output. Three variables i , j and k are initially set to the beginning of the three arrays $a[]$, $b[]$ and $c[]$. The smaller of $a[i]$ and $b[j]$ is copied to $c[k]$ and appropriate variables (i , k) or (j , k) are advanced. When either of the input array $a[]$ or $b[]$ is exhausted, the remainder of the other array is copied to $c[]$.

Initial conditions



Array b[] is exhausted, the remainder of the array a[] are added to c[].



Example 6.9.1 : Sort the following list of numbers using merge sort. Show result stepwise :

50, 10, -10, 40, 15, 25, 20, 35, 30

Solution :

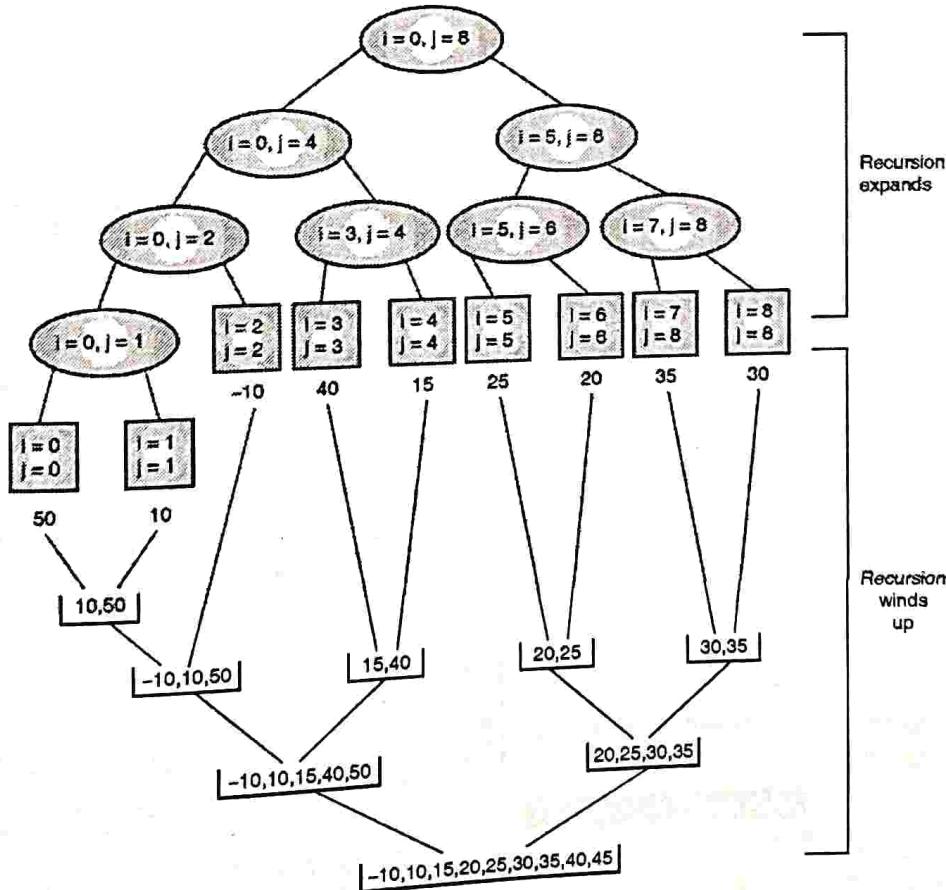


Fig. Ex. 6.9.1



'C' function for merging of two sorted arrays.

```

void merge(int a[ ], int l, int m, int u)
{
    int c[MAX];
    int i, j, k;
    /* First array is supposed to be from location l to m
       of array a[ ] */
    /* Second array is supposed to be from m+1
       to u of array a[ ] */
    // Array c[ ] is used for merging
    // after merging array c[ ] is copied back to a[ ]
    i = l;
    j = m+1;
    k = 0;
    while(i <= m && j <= u)
    {
        if(a[i] < a[j])
        {
            c[k] = a[i];
            k++; i++;
        }
        else
        {
            c[k] = a[j];
            k++; j++;
        }
    }
    while(i <= m)
    {
        c[k] = a[i];
        i++; k++;
    }
    while(j <= u)
    {
        c[k] = a[j];
        k++; j++;
    }
    for(i = l, j = 0; i <= u; i++, j++)
        a[i] = c[j];
}

```

Program 6.9.1 : Program for Implementation of merge sort.

MU - May 18, 10 Marks

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

```

#define MAX 30
void merge_sort(int a[ ], int i, int j);
void merge(int a[ ], int i, int j, int k);
void main()
{
    int a[30], n, i;
    printf("\n No. of elements :");
    scanf("%d", &n);
    printf("Enter array elements :");
    for(i = 0; i < n; i++)
        scanf("%d", &a[i]);
    merge_sort(a, 0, n-1);
    printf("\n Sorted data ->\n ");
    for(i = 0; i < n; i++)
        printf("%d", a[i]);
    getch();
}

void merge_sort(int a[ ], int i, int j)
{
    int k;
    if(i < j)
    {
        k = (i+j)/2;
        merge_sort(a, i, k);
        merge_sort(a, k+1, j);
        merge(a, i, k, j);
    }
}

void merge(int a[ ], int l, int m, int u)
{
    int c[MAX];
    int i, j, k;
    /* First array is supposed to be from location l to m
       of array a[ ] */
    /* Second array is supposed to be from m+1 to u
       of array a[ ] */
    // Array c[ ] is used for merging
    // after merging array c[ ] is copied back to a[ ]
    i = l;
    j = m+1;
    k = 0;
    while(i <= m && j <= u)
    {
        if(a[i] < a[j])
        {

```

```

c[k] = a[i];
k++; i++;
}
else
{
    c[k] = a[j];
    k++; j++;
}
}
while(i <= m)
{
    c[k] = a[i];
    i++; k++;
}
while(j <= u)
{
    c[k] = a[j];
    k++; j++;
}
for(i = l, j = 0; i <= u; i++, j++)
    a[i] = c[j];
}

```

Output

| |
|---|
| No. of elements : 4 |
| Enter array elements: 32 56 22 1 |
| Sorted data -> 1 22 32 56 |

6.9.2 Analysis of Merge Sort

- Since, merge sort is a recursive program, we must write a recurrence relation for the running time.
- Time to merge sort N numbers is equal to the time to merge sort left $N/2$ element, plus time to merge sort right $N/2$ element, plus time to merge two linear lists of $N/2$ elements.
- Merging requires one pass of linear scanning of the two input arrays (each having $N/2$ elements). Hence the time to merge is linear.
- $T(1) = 1$, since recursion terminates when $N = 1$. Time to merge sort an array having 1 element is constant.

$$T(N) = 2T(N/2) + N$$

Or $\frac{T(N)}{N} = \frac{T(N/2)}{N/2} + 1$... (1)

Equation (1) can be written as (expanding $T(N/2)$)

$$= \frac{T(N/4)}{N/4} + 1 + 1 = \frac{T(N/2^2)}{N/2^2} + 2$$

Let us assume, $N = 2^h$ i.e. $N/2^h = 1$

Equation (1) can be written as,

$$\begin{aligned} \frac{T(N)}{N} &= \frac{T(N/2^h)}{N/2^h} + h \\ &= \frac{T(1)}{1} + h = T(1) + h \end{aligned} \quad \dots (2)$$

Since $N = 2^h$

Taking log on both sides, we have

$$\log N = \log_2^{2^h} = h$$

Writing h as $\log N$ in Equation (2)

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

or $T(N) = NT(1) + N \log N = O(N \log N)$

- Although the running time of merge sort is $O(N \log_2 N)$, it is seldom used for internal sorting. Merging of two sorted arrays require additional memory.
- Two sorted arrays are merged through a temporary array and the contents of the temporary array is copied back requiring additional space and time. Merge sort is found to be very effective in external sorting.

6.9.3 Non-Recursive Merge Sort

- Non-recursive merge sort uses bottom up approach.
- When the initial array $a[]$ of N elements is divided into N groups (each group having 1 element), elements inside a group are sorted (a group with 1 element is always sorted). This is also called that Initially, array is organized into runs of length 1.
- The first pass of the algorithm merges n pair of runs of length = 1 at level 0 to create runs of length 2 at level 1.
- The second pass merges a pair of runs of length = 2 at level 1 to create runs of length 4 at level 2. In every successive pass, number of runs is reduced by half and the length of runs is doubled.
- If P passes are required for sorting, then $2^P \geq N$.

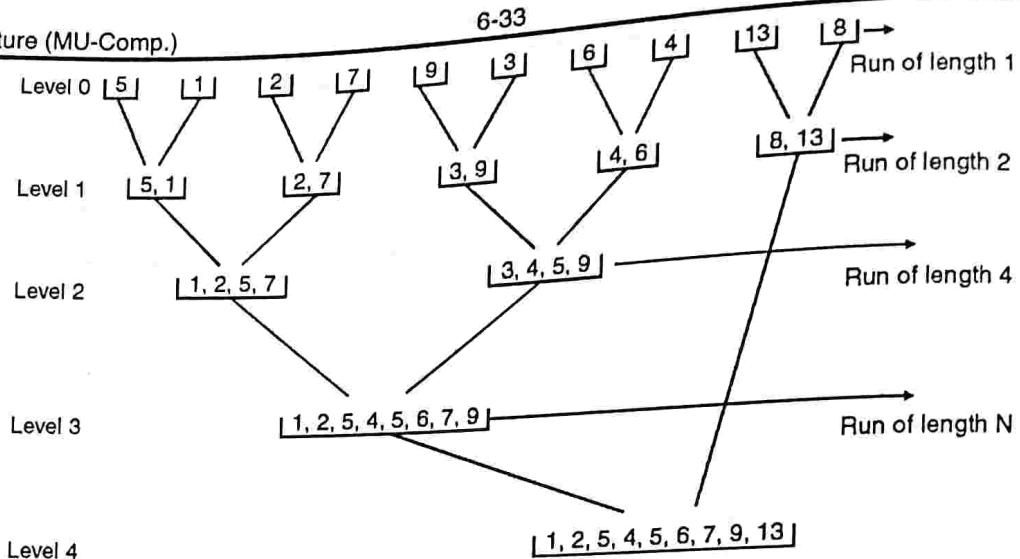


Fig. 6.9.3

'C' function for non-recursive merge sort.

```
void non_recursive_mergesort(int a[], int N)
{
    int run_size, no_of_runs, i, j;
    run_size = 1; //one element at a time at level 0
    while(run_size < N)
    {
        no_of_runs = N/run_size;
        if(no_of_runs*run_size < N)
            no_of_runs++;
        //last run of smaller size
        i = 0; j = run_size;
        /* i = beginning of the first run
        j = beginning of the next run */
        while(no_of_runs > 1)
        {
            if(i+2*run_size > N)
                //last run of smaller size
                merge(a, i, j-1, N-1);
            else
                merge(a, i, j-1, j+run_size-1);
            i = i+run_size;
            j = j+run_size;
            no_of_runs = no_of_runs-2;
        }
        run_size = run_size * 2;
    }
}
```

6.10 Comparison of Sorting Algorithms

MU - Dec. 14, May 17

University Question

- Q. Compare and contrast Quick sort and Radix sort on basis of their advantages and disadvantages.

(Dec. 14, May 17, 5 Marks)

Comparison of sorting algorithm w.r.t.

(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 | $O(n)$ | $n - 1$ | Stable |
| Worst case | $O(n^2)$ | $n - 1$ | |
| Quick sort | | | |
| Best case | $O(n \log n)$ | $\log n$ | Unstable |
| Worst case | $O(n^2)$ | $n - 1$ | |
| Merge sort | $O(n \log n)$ | $\log n$ | Stable |
| Shell sort | | | |
| Best case | $O(n)$ | $\log n$ | Unstable |
| Worst case | $O(n^2)$ | $\log n$ | |
| Radix sort | $O(n)$ | No. of digits in the largest number | Stable |

6.11 Best-Case, Worst-Case and Average-Case Analysis of 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.

Example 6.11.1 : 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 :

| | | | | | | | | | | | |
|---|----|----|---|----|----|---|---|---|---|--|--|
| | | | | 54 | | | | | | | |
| | | | | 64 | | | | | | | |
| | | | | 74 | 36 | | | | | | |
| | | | | 14 | 66 | | | | | | |
| | 41 | | | | | | | | | | |
| | 1 | 22 | | | | | | | | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | | |

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

Example 6.11.2 : Write the contents of list and each bucket, after each pass using radix sort for the following list of numbers :

10, 2, 15, 246, 37, 4, 25, 62, 100, 17.

Solution :

Buckets after pass 1

| | | | | | | | | | | | |
|-----|---|---|----|---|---|---|----|----|-----|----|--|
| 100 | | | 62 | | | | 25 | | | 17 | |
| 10 | | | 2 | | | | 4 | 15 | 246 | 37 | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | | |

Merged list = 10, 100, 2, 62, 4, 15, 25, 246, 37, 17

Buckets after pass 2

| | | | | | | | | | | | |
|-----|----|----|----|-----|---|---|----|---|---|--|--|
| 4 | 17 | | | | | | | | | | |
| 2 | 15 | | | | | | | | | | |
| 100 | 10 | 25 | 37 | 246 | | | 62 | | | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | | |

merged list = 100, 2, 4, 10, 15, 17, 25, 37, 246, 62

Buckets after pass 3

| | | | | | | | | | | | |
|----|-----|-----|---|---|---|---|---|---|---|--|--|
| 62 | | | | | | | | | | | |
| 37 | | | | | | | | | | | |
| 25 | | | | | | | | | | | |
| 17 | | | | | | | | | | | |
| 15 | | | | | | | | | | | |
| 10 | | | | | | | | | | | |
| 4 | | | | | | | | | | | |
| 2 | 100 | 246 | | | | | | | | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | | |

merged list = 2, 4, 10, 15, 17, 25, 37, 62, 100, 246

Buckets after 2nd pass :

| | | | | | | | | | | | |
|---|----|----|----|----|----|----|----|----|----|--|--|
| | | | | | 59 | 66 | | | | | |
| | | | | | 41 | 54 | 64 | 74 | | | |
| | | | | | 36 | 54 | 64 | 74 | | | |
| | | | | | 14 | 41 | 54 | 64 | 74 | | |
| 1 | 14 | 22 | 36 | 41 | 54 | 59 | 64 | 66 | 74 | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | | |

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

6.12 External Vs Internal Sorting

- Internal sorting requires that input data should be stored in main memory.
- If the input data is too large to fit into memory, internal sorting can not be used.
- External sorting algorithms take advantage of random access nature of primary memory.



For example :

- (a) Shell sort algorithm compares $a[j]$ and $a[j - \text{step}]$
- (b) Partition algorithm in quick sort compares $a[i]$ and $a[j]$.
- If the input is stored on the sequential device than comparison of two random elements will be very costly. Hence, shell sort or quick sort should not be used for external sorting.
- Some of the sorting algorithm are based on comparisons of adjacent elements. These algorithms can be used for external sorting. Merge sort is one of such algorithms with $O(n \log n)$ time complexity.

Examples of internal sorting

1. Quick sort
2. Shell sort
3. Radix sort

These algorithms have already been discussed in detail.

Examples of external sorting

1. Merge sort
2. Bubble sort
3. Selection sort
4. Insertion sort

We will discuss merge sort for external sorting :

Suppose, input data is stored on tape T1 and there are additional three tapes which can be used for merging. If memory can hold maximum of 3 records then status of tapes after every run is shown in the example below.

Initial condition

| | | | | | | | | | | | | | |
|----|---|----|---|---|---|----|----|---|----|----|----|----|----|
| T1 | 5 | 11 | 3 | 9 | 6 | 55 | 42 | 7 | 13 | 25 | 21 | 24 | 15 |
| T2 | | | | | | | | | | | | | |
| T3 | | | | | | | | | | | | | |
| T4 | | | | | | | | | | | | | |

Data on tapes after 1st run

| | | | | | | | | | | | | | |
|----|---|---|----|----|----|----|--|--|--|--|--|--|--|
| T1 | | | | | | | | | | | | | |
| T2 | | | | | | | | | | | | | |
| T3 | 3 | 5 | 11 | 7 | 13 | 42 | | | | | | | |
| T4 | 6 | 9 | 55 | 21 | 24 | 25 | | | | | | | |

Data on tapes after 2nd run

| | | | | | | | |
|----|---|----|----|----|----|----|----|
| T1 | 3 | 5 | 6 | 9 | 11 | 55 | 15 |
| T2 | 7 | 13 | 21 | 24 | 25 | 42 | |
| T3 | | | | | | | |
| T4 | | | | | | | |

Data on tapes after 3rd run

| | | | | | | | | | | | | | |
|----|----|---|---|---|---|----|----|----|----|----|----|----|--|
| T1 | | | | | | | | | | | | | |
| T2 | | | | | | | | | | | | | |
| T3 | 3 | 5 | 6 | 7 | 9 | 11 | 13 | 21 | 24 | 25 | 42 | 55 | |
| T4 | 15 | | | | | | | | | | | | |

Data on tapes after 4th run

| | | | | | | | | | | | | | |
|----|---|---|---|---|---|----|----|----|----|----|----|----|----|
| T1 | 3 | 5 | 6 | 7 | 9 | 11 | 13 | 15 | 21 | 24 | 25 | 42 | 55 |
| T2 | | | | | | | | | | | | | |
| T3 | | | | | | | | | | | | | |
| T4 | | | | | | | | | | | | | |

Example 6.12.1 : "External sorting calls for internal sorting as well". Justify.

Solution :

Every sorting algorithm is based on passes. Inside a pass, records are compared on key values and the records can be shuffled based on the outcome of comparison. In case of external sorting, records to be compared are read into memory variables and they are re-written at appropriate places.

Pseudo code for bubble sort for external sorting is given to explain the above concept.

```

n ← no of records in the file ;
rec1, rec2, : record type ;
for(i = 1 ; i < n ; i++)
  for(j = 0 ; j < n - i ; j++)
    {
      rec1 ← read the jth record ;
      rec2 ← read the (j + 1) th record ;
      if(rec1 · key > rec2 · key)
        {
          write rec2 at jth place ;
          write rec1 at (j + 1) th place ;
        }
    }
}

```

6.13 Hash Tables

6.13.1 What is Hashing ?

MU - May 14, Dec. 16, Dec. 18, Dec. 19

University Question

Q. What is hashing ?

(May 14, Dec. 16, Dec. 18, Dec. 19, 2 Marks)

- Sequential search requires, on the average $O(n)$ comparisons to locate an element. So many comparisons are not desirable for a large database of elements.
- Binary search requires much fewer comparisons on the average $O(\log n)$ but there is an additional requirement that the data should be sorted. Even with best sorting algorithm, sorting of elements require $O(n \log n)$ comparisons.
- There is another widely used technique for storing of data called "hashing".
- It does away with the requirement of keeping data sorted (as in binary search) and its best case timing complexity is of constant order ($O(1)$).
- In its worst case, hashing algorithm starts behaving like linear search.
- Best case timing behaviour of searching using hashing = $O(1)$
- Worst case timing Behaviour of searching using hashing = $O(n)$.
- Since, there is a large gap between its best case $O(1)$ and worst case $O(n)$ behaviour. It should be implemented properly to get an average case behaviour close to $O(1)$. In hashing , the record for a key value "key". is directly referred by calculating the address from the key value.
- Address or location of an element or record, x , is obtained by computing some arithmetic function $f.f(key)$ gives the address of x in the table.
- (a) Table used for storing of records is known as hash table.

Function $f(key)$ is known as hash function.

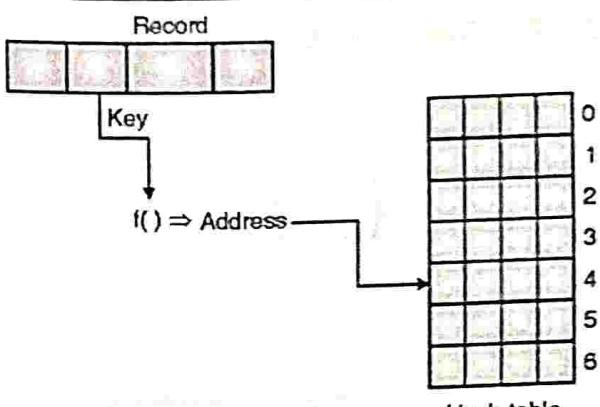


Fig. 6.13.1 : Mapping of record in hash table

Example

Suppose, we wish to implement a hash table for a set of records where the key is a member of set. Set K of strings.

$K = \{\text{"aaa"}, \text{"bbb"}, \text{"ccc"}, \text{"ddd"}, \text{"eee"}, \text{"fff"}, \text{"ggg"}\}$
A function $f : \text{key} \rightarrow \text{Index}$ is given by the following table :

| N | f(x) |
|-------|------|
| "aaa" | 0 |
| "bbb" | 1 |
| "ccc" | 2 |
| "ddd" | 3 |
| "eee" | 4 |
| "fff" | 5 |
| "ggg" | 6 |

Hash table can be implement using an array of records of length $n = 7$.

To store a record with key x , we simply store it at position $f(x)$ in the array. Similarly, to locate the record having key = x , we simply check to see if it is found at position $f(x)$.

6.13.2 Hash Table Data Structure

- There are two different forms of hashing.
 - (a) Open hashing or external hashing
 - (b) Close hashing or internal hashing
- Open or external hashing, allows records to be stored in unlimited space (could be a hard disk). It places no limitation on the size of the tables. Closed or internal hashing, uses a fixed space for storage and thus limits the size of hash table.



6.13.2(A) Open Hashing Data Structure

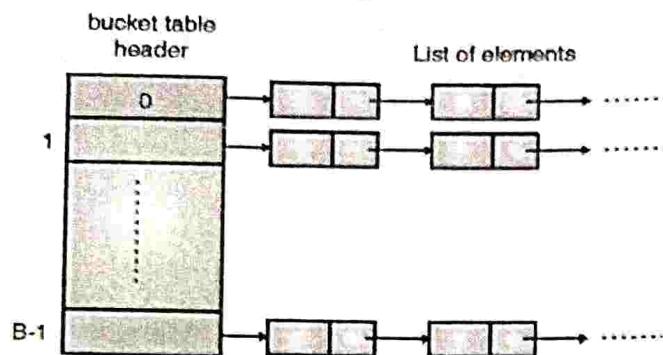


Fig. 6.13.2 : The open hashing data organization

Fig. 6.13.2 gives the basic data structure for open hashing.

The basic idea is that the records [elements] are partitioned into B classes, numbered $0, 1, 2, \dots, B-1$. Hashing function $f(x)$ maps a record with key n to an integer value between 0 and $B-1$. If a record is mapped to location 1 then we say the record is mapped to bucket 1 or the record belongs to class 1 . Each bucket in the bucket table is the head of the linked list of records mapped to that bucket.

6.13.2(B) Closed Hashing Data Structure

A closed hash table keeps the elements in the bucket itself. Only one element can be put in the bucket. If we try to place an element in the bucket $f(n)$ and find it already holds an element, then we say that a collision has occurred. In case of collision, the element should be rehashed to alternate empty location $f_1(x), f_2(x), \dots$ within the bucket table. In closed hashing, collision handling is a very important issue.



Fig. 6.13.3 : Partially filled hash table

6.13.3 Hashing Functions

- We are designing a container which will be used to hold some number of items of a given set K . In this context, we call the elements of the set K keys.
- The general approach is to store the keys in an array. The position of a key in the array is given by a function $h(\cdot)$, called a *hash function*, which determines the position of a given key directly from that key.
- In the general case, we expect the size of the set of keys, $|K|$, to be relatively large or even unbounded. For example, if the keys are 32-bit integers, then $|K| = 2^{32}$.
- Similarly, if the keys are arbitrary character strings of arbitrary length, then $|K|$ is unbounded.
- On the other hand, we also expect that the actual number of items stored in the container to be significantly less than $|K|$. That is, if n is the number of items actually stored in the container, then $n \ll |K|$. Therefore, it seems prudent to use an array of size M , where M is at least as great as the maximum number of items to be stored in the container.
- Consequently, what we need is a function $h : K \rightarrow \{0, 1, \dots, M-1\}$. This function maps the set of values to be stored in the container to subscripts in an array of length M .
- This function is called a *hash function*.
- In general, since, $|K| \geq M$ the mapping defined by hash function will be a *many-to-one mapping*. That is, there will exist many pairs of distinct keys x and y , such that $x \neq y$, for which $h(x)=h(y)$.
- This situation is called a *collision*. Several approaches for dealing with collisions are explored in the following sections.

6.13.3(A) Characteristics of a Good Hash Function

MU - Dec. 18

University Question

Q. What properties should a good hash function demonstrate ?
(Dec. 18, 3 Marks)

- A good hash function avoids collisions.
- A good hash function tends to spread keys evenly in the array.
- A good hash function is easy to compute.

- There are many hashing functions. We will discuss these in the following sections.

6.13.3(B) Division-Method

In this method we use modular arithmetic system to divide the key value by some integer divisor m (may be table size). It gives us the location value, where the element can be placed. We can write,

$$L = (K \bmod m) + 1$$

where L = Location in table/file

K = Key value

m = Table size/number of slots in file

suppose, $K = 23$, $m = 10$ then

$$L = (23 \bmod 10) + 1 = 3 + 1 = 4$$

∴ The key whose value is 23 is placed in 4th location.

6.13.3(C) Midsquare Methods

- In this case, we square the value of a key and take the number of digits required to form an address, from the middle position of squared value.
- Suppose a key value is 16, then its square is 256. Now if we want address of two digits, then you select the address as 56. (i.e. two digits starting from middle of 256).

6.13.3(D) Folding Method

- Most machines have a small number of primitive data types for which there are arithmetic instructions. Frequently key to be used will not fit easily in to one of these data types.
- It is not possible to discard the portion of the key that does not fit into such an arithmetic data type. The solution is to combine the various parts of the key in such a way that all parts of the key affect for final result such an operation is termed **folding** of the key.
- That is the key is actually partitioned into number of parts, each part having the same length as that of the required address. Add the value of each parts, ignoring the final carry to get the required address. This is done in two ways :
 - (a) **Fold-shifting** : Here actual values of each parts of key are added.

- (b) **Fold-boundary** : Here the reversed values of outer parts of key are added.

Suppose, the key is : 12345678, and the required address is of two digits, then break the key into parts : 12, 34, 56, 78.

Add these, we get $12 + 34 + 56 + 78 =$

1 80

Ignore

So we get address as 80. (This is fold-shifting).

For fold boundary, reverse the key parts, we get 21, 34, 56, 87

Add these parts, $21 + 34 + 56 + 87 =$

1 98

Omit

So we get the address as 98.

6.13.3(E) Digit Analysis

- This hashing function is a distribution-dependent. Here we make a statistical analysis of digits of the key, and select those digits (of fixed position) which occur quite frequently.
- Then reverse or shifts the digits to get the address.
- For example, if the key is : 9861234. If the statistical analysis has revealed the fact that the third and fifth position digits occur quite frequently, then we choose the digits in these positions from the key. So we get, 62. Reversing it we get 26 as the address.

6.13.3(F) Length Dependent Method

- In this type of hashing function we use the length of the key along with some portion of the key to produce the address, directly.
- In the indirect method, the length of the key along with some portion of the key is used to obtain intermediate value. Then use any other method to obtain the address values.

6.13.3(G) Algebraic Coding

- Here a n bit key value is represented as a polynomial. The divisor polynomial is then constructed based on the address range required.
- The modular division of key-polynomial by divisor polynomial, to get the address-polynomial.

Let $f(x) = \text{Polynomial of } n \text{ bit key}$



$$= a_1 + a_2 x + \dots + a_n x^{n-1}$$

$$d(x) = \text{Divisor polynomial}$$

$$= x^1 + d_1 + d_2 x + \dots + d_{l-1} x^{l-1}$$

(i.e. if required address is in the range 0 to $k = 2^l - 1$)

Then the required address-polynomial will be,

$$f(x) \bmod d(x) = ab_1 + b_2 x + \dots + b_l x^{l-1}$$

where $(b_1, b_2 \dots b_l)_2$ is the address.

6.13.3(H) Multiplicative Hashing

- This method is based on obtaining an address of a key, based on the multiplication value. If k is the non-negative key, and a constant c , ($0 < c < 1$), compute $kc \bmod 1$, which is a fractional part of kc . Multiply this fractional part by m and take a floor value to get the address.

$$h(k) = \lfloor m(kc \bmod 1) \rfloor, \\ 0 \leq h(k) < m.$$

- So far we talked about how to map a key to address. But, what happens if the two keys yield the same address values? Well it results in collision. Hence one must resolve these collisions.

6.13.4 Collision Resolution Strategies (Synonym Resolution)

MU - May 14, Dec. 16, May 19

University Question

Q. What is meant by collisions?

(May 14, Dec. 16, 4 Marks)

Q. Write Short note on : Collision Handling techniques.

(May 19, 10 Marks)

- Collision resolution is the main problem in hashing. If the element to be inserted is mapped to the same location, where an element is already inserted then we have a collision and it must be resolved.
- There are several strategies for collision resolution. The most commonly used are :
 - Separate chaining** : Used with open hashing.
 - Open addressing** : Used with closed hashing.

6.13.4(A) Separate Chaining

- In this strategy, a separate list of all elements mapped to the same value is maintained. Fig. 6.13.4 shows an implementation of separate chaining.
- Separate chaining is based on collision avoidance.
- If memory space is tight, separate chaining should be avoided.
- Additional memory space for links is wasted in storing address of linked elements.
- Hashing function should ensure even distribution of elements among buckets, otherwise the timing behaviour of most operations on hash table will deteriorate.

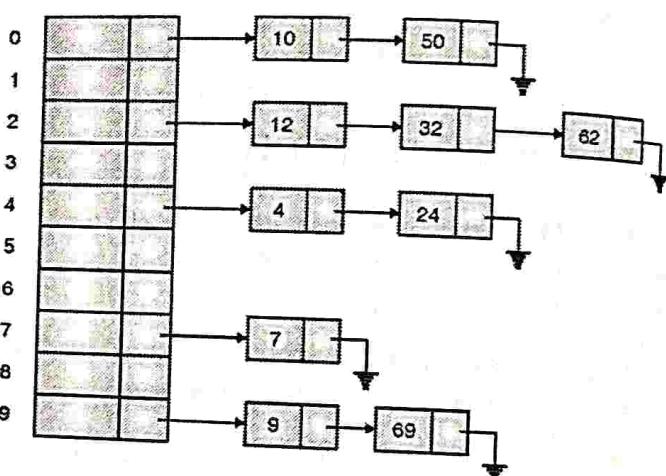


Fig. 6.13.4 : A separate chaining hash table

In the Fig. 6.13.4 :

- A simple hash function $h(x) = x \bmod 10$ is taken. Bucket table has a size of 10.
- To perform a **find**, hash function is used to find the list to be traversed for searching. Then the list is traversed in normal manner.
- To perform an **insert**, the appropriate list is traversed to ensure that the element to be inserted is not present in the list. If the element is not present then it is inserted at the front.

Data structure for separate chaining

```
#define MAX 10
typedef struct node
{
    int data;
    struct node *next;
};

node *hashtable[MAX]; //bucket table header
```

Operations on hash table :(a) **Initialise()** : Header of all lists are set to NULL.

```
void initialise(node *hashtable[ ]) {
    int i;
    for(i = 0; i < MAX; i++)
        hashtable[i] = NULL;
}
```

(b) **Insert ()** : If the item is already present then do nothing; otherwise, place the item at the front of the list.

```
void insert(node *hashtable[ ], int x) {
    int loc;
    node *p, *q;
    loc = x % MAX;
    // Mapped location get memory for new node
    q = (node*)malloc(sizeof(node));
    q->data = x;
    q->next = NULL;
    if(hashtable[loc] == NULL)
        hashtable[loc] = q;
    else
        { // locate the last node of the linked list
            for(p = hashtable[loc]; p->next != NULL;
                p = p->next);
            p->next = q;
        }
}
```

(c) **Find ()** : The function returns a pointer to the node containing the search key.

```
node *find(node *hashtable[ ], int x) {
    int loc;
    node *p;
    loc = x % MAX;
    p = hashtable[loc];
    while(p != NULL && x != p->data)
        p = p->next;
    return(p);
}
```

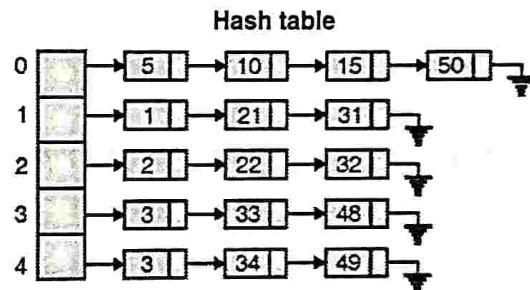
Example 6.13.1 : The integers given below are to be inserted in a hash table with 5 locations using chaining to resolve collisions. Construct hash table and use simplest hash function.

1, 2, 3, 4, 5, 10, 21, 22, 33, 34, 15, 32, 31, 48, 49, 50

Solution :

An element can be mapped to a location in the hash table using the mapping function $\text{key} \% 10$.

| Hash table locations | Mapped elements |
|----------------------|-----------------|
| 0 | 5, 10, 15, 50 |
| 1 | 1, 21, 31 |
| 2 | 2, 22, 32 |
| 3 | 3, 33, 48 |
| 4 | 4, 34, 49 |

**6.13.4(B) Open Addressing**

- Separate chaining requires additional memory space for pointers. Open addressing hashing is an alternate method of handling collision.
- In open addressing, if a collision occurs, alternate cells are tried until an empty cell is found. Because all the data elements are stored inside the table, a larger memory space is needed for open addressing.
- Generally, the load factor should be below 0.5 for open addressing hashing. There are three commonly used collision resolution strategy in open addressing.

1. Linear probing
2. Quadratic probing
3. Double hashing.

1. Linear Probing

In linear probing, whenever there is a collision, cells are searched sequentially (with wraparound) for an



empty cell. Fig. 6.13.5 shows the result of inserting keys {5,18,55,78,35,15} using the hash function ($f(key) = key \% 10$) and linear probing strategy.

| | Empty table | After 5 | After 18 | After 55 | After 78 | After 35 | After 15 |
|---|-------------|---------|----------|----------|----------|----------|----------|
| 0 | | | | | | | 15 |
| 1 | | | | | | | |
| 2 | | | | | | | |
| 3 | | | | | | | |
| 4 | | | | | | | |
| 5 | | 5 | 5 | 5 | 5 | 5 | 5 |
| 6 | | | | 55 | 55 | 55 | 55 |
| 7 | | | | | | 35 | 35 |
| 8 | | | 18 | 18 | 18 | 18 | 18 |
| 9 | | | | | 78 | 78 | 78 |

Fig. 6.13.5 : Open addressing hash table with linear probing after each iteration

Operations on hash table using linear probing

In order to present algorithms, certain assumptions are made.

1. Elements of the array consists only of the key field.
2. Array hashtable[] is assumed to be of size MAX.
3. Another array T is used to indicate whether an element of hashtable[] is empty or not.

'C' function for inserting an element "key" in the hash table.

```
int insert_linear_probe(int hashtable[], int key, int T[])
{
    int i, j;
    j = key % MAX; //Mapped location
    for(i = 0; i < MAX; i++)
    {
        if(T[j] == 0) //an empty cell is found
        {
            hashtable[j] = key;
            T[j] = 1;
            return(j);
        }
        j = (j+1)%MAX;
    }
}
```

```
/*next location in circular way*/
}
//otherwise the table is full
return(-1);
}
```

- The Instruction $j = (j+1)\% MAX$ gives the wraparound effect.
- The function returns (-1) when no space is available for insertion.
- Array T[] should be initialized to 0 at the beginning. The procedure for searching looks for the element "key" in the hashtable[]. The function returns the index of hashtable[], where "key" is found. The function returns (-1) if the "key" is not found.

'C' function for searching an element "Key" in the hash table.

```
int search_linear_probe(int hashtable[], int key, int T[])
{
    int i, j;
    j = key % MAX; //Mapped location
    for(i = 0; i < MAX; i++)
    {
        if(T[j] == 1 && hashtable[j] == key)
            return(j);
        j = (j+1)%MAX;
        /*next location in circular way*/
    }
    //otherwise the table is full
    return(-1);
}
```

- Linear probing is easy to implement but it suffers from "primary clustering". When many keys are mapped to the same location (clustering), linear probing will not distribute these keys evenly in the hashtable.
- These keys will be stored in neighborhood of the location where they are mapped. This will lead to clustering of keys around the point of collision.

Program 6.13.1 : Program on Hash Table – Linear Probing

```
/* Hashing */
/* Program Details - Hashing , handle collision using
linear probing */
#include <stdio.h>
#include <conio.h>
#define SIZE 10           /* size of the hash table*/
#define FALSE 0
```

```

#define TRUE 1
#define h(x) x%SIZE           /*hashing function */
void insert( int data[ ], int flag[ ], int x);
int search(int data[ ], int flag[ ], int x);
void print(int data[ ], int flag[ ]);

void main()
{
    int data[SIZE], flag[SIZE], i, j, x, op, loc;
    /* array data[ ] - is a hash table
       array flag[ ] - if flag[i] is 1 then the ith place of
       the hash table is filled */
    for(i = 0; i<SIZE; i++)      /* initialize */
        flag[i] = FALSE;
    clrscr();
    do
    {
        printf("\n\n 1)Insert\n 2)Search\n 3)Print\n
 4)Quit");
        printf("\n Enter Your Choice : ");
        scanf("%d", &op);
        switch(op)
        {
            case 1:
                printf("\n Enter a number to be inserted: ");
                scanf("%d", &x);
                insert(data, flag, x);
                break;
            case 2: printf("\n Enter a number to be searched : ");
                scanf("%d", &x);
                if((loc = search(data, flag, x)) == -1)
                    printf("\n *Element not found*");
                else
                    printf("\n *Found at the location=%d", loc);
                break;
            case 3: print(data, flag);
                break;
        }
    }while(op!= 4);
}

```

```

void insert( int data[ ], int flag[ ], int x)
{
    int i, j;
    j = h(x);           /*hashed location*/
    for(i = 0; i<SIZE; i++)
    {
        if(flag[j] == FALSE)
            //empty location is found
    }
}

```

```

        data[j] = x;
        flag[j] = TRUE;
        return;
    }
    else
        j = (j+1)%SIZE;
}

printf("\n *****hash table is full");
}

int search(int data[ ], int flag[ ], int x)
{
    int i, j;
    j = h(x);           /*hashed location*/

    for(i = 0; i<SIZE; i++)
        if(flag[j] == TRUE && data[j] == x)
            return(j);
        else
            j = (j+1)%SIZE;

    return(-1);
}

void print(int data[ ], int flag[ ])
{
    int i;
    for(i = 0; i<SIZE; i++)
        if(flag[i])
            printf("\n(%d) %d", i, data[i]);
        else
            printf("\n(%d) ---", i);
}

```

Output

1)Insert
2)Search
3)Print
4)Quit
Enter Your Choice : 1

Enter a number to be inserted:25

1)Insert
2)Search
3)Print
4)Quit
Enter Your Choice : 1



Enter a number to be inserted:35

- 1)Insert
- 2)Search
- 3)Print
- 4)Quit

Enter Your Choice : 1

Enter a number to be inserted:45

- 1)Insert
- 2)Search
- 3)Print
- 4)Quit

Enter Your Choice : 3

- (0) ---
- (1) ---
- (2) ---
- (3) ---
- (4) ---
- (5) 25
- (6) 35
- (7) 45
- (8) ---
- (9) ---

- 1)Insert
- 2)Search
- 3)Print
- 4)Quit

Enter Your Choice : 2

Enter a number to be searched :35

*Found at the location = 6

- 1)Insert
- 2)Search
- 3)Print
- 4)Quit

Enter Your Choice : 2

Enter a number to be searched :15

Element not found

- 1)Insert
- 2)Search
- 3)Print
- 4)Quit

Enter Your Choice : 1

Enter a number to be inserted:27

- 1)Insert
- 2)Search
- 3)Print
- 4)Quit

Enter Your Choice : 3

- (0) ---
- (1) ---
- (2) ---
- (3) ---
- (4) ---
- (5) 25
- (6) 35
- (7) 45
- (8) 27
- (9) ---

- 1)Insert
- 2)Search
- 3)Print
- 4)Quit

Enter Your Choice : 4

2. Quadratic Probing

- One way of reducing "primary clustering" is to use quadratic probing to resolve collision. Suppose the "key" is mapped to the location j and the cell j is already occupied.
- In quadratic probing, the location $j, (j+1), (j+4), (j+9), \dots$ are examined to find the first empty cell where the key is to be inserted.

- This table reduces primary clustering . It does not ensure that all cells in the table will be examined to find an empty cell. Thus, it may be possible that key will not be inserted even if there is an empty cell in the table.

'C' function to insert a new "key" using quadratic probing.

```
int insert_quadratic_probe(int hashtable[ ], int key,
                           int T[ ])
{
    int i, j, start;
    start = key % MAX;           //mapped location
    for(i = 0; i < MAX; i++)
    {
        j = (start + i*i)%MAX;
        if(T[j] == 0)           //empty location is found
        {
            hashtable[j] = key;
            T[j] = 1;
            return j;
        }
    }
    return(-1);
}
```

'C' function for searching an element in the hash table using quadratic probing.

```
int search_quadratic_probe(int hashtable[ ], int key, int
                           T[ ])
{
    int i, j, start;
    start = key % MAX; //mapped location
    for(i = 0; i < MAX; i++)
    {
        j = (start + i*i)%MAX;
        if(T[j] == 1 && hashtable[j] == key)
            return j;
    }
}
```

```
return(-1);
}
```

C. Double Hashing

- This method requires two hashing functions $f_1(key)$ and $f_2(key)$. Problem of clustering can easily be handled through double hashing. Function $f_1(key)$ is known as primary hash function.
- In case the address obtained by $f_1(key)$ is already occupied by a key, the function $f_2(key)$ is evaluated. The second function $f_2(key)$ is used to compute the increment to be added to the address obtained by the first hash function $f_1(key)$ in case of collision.
- The search for an empty location is made successively at the addresses.
 $f_1(key)+f_2(key), f_1(key)+2f_2(key), f_1(key) + 3f_2(key), \dots$
- A function to insert a new key using double hashing is shown below. Two hash functions $f_1()$ and $f_2()$ are assumed to be available.

'C' function to insert a new key using double hashing.

```
int insert_double_hash(int hashtable[ ], int key, int T[ ])
{
    int i, j, start, u;
    start = f1(key) % MAX;
    //mapped location
    u = f2(key); // u will be used for increment
    for(i = 0; i < MAX; i++)
    {
        j = (start + i*u)%MAX;
        if(T[j] == 0) //empty location is found
        {
            T[j] = 1;
            hashtable[j] = key;
            return j;
        }
    }
    return(-1);
}
```



Example 6.13.2 : Using linear probing and quadratic probing insert the following values in a hash table of size 10. Show how many collisions occur in each technique : 99, 33, 23, 44, 56, 43, 19
MU - Dec. 13, May 14, Dec. 16, Dec. 19, 10 Marks

Solution :

1. Linear probing

| | Empty table | After 99 | After 33 | After 23 | After 44 | After 56 | After 43 | After 19 |
|---|-------------|----------|----------|----------|----------|----------|----------|----------|
| 0 | | | | | | | | 19* |
| 1 | | | | | | | | |
| 2 | | | | | | | | |
| 3 | | | 33 | 33 | 33 | 33 | 33 | 33 |
| 4 | | | | 23* | 23* | 23* | 23* | 23* |
| 5 | | | | | 44* | 44* | 44* | 44* |
| 6 | | | | | | 56 | 56 | 56 |
| 7 | | | | | | | 43* | 43* |
| 8 | | | | | | | | |
| 9 | | 99 | 99 | 99 | 99 | 99 | 99 | 99 |

* = Collision
Number of collisions = 4

2. Quadratic probing

| | Empty table | After 99 | After 33 | After 23 | After 44 | After 56 | After 43 | After 19 |
|---|-------------|----------|----------|----------|----------|----------|----------|----------|
| 0 | | | | | | | | 19* |
| 1 | | | | | | | | |
| 2 | | | | | | | | |
| 3 | | | 33 | 33 | 33 | 33 | 33 | 33 |
| 4 | | | | 23* | 23* | 23* | 23* | 23* |
| 5 | | | | | 44* | 44* | 44* | 44* |
| 6 | | | | | | 56 | 56 | 56 |
| 7 | | | | | | | 43* | 43* |
| 8 | | | | | | | | |
| 9 | | 99 | 99 | 99 | 99 | 99 | 99 | 99 |

* = Collision
Number of collisions = 4

Example 6.13.3 : Hash the following in table of size 11. Use any two collision resolution techniques
23, 55, 10, 71, 67, 32, 100, 18, 10, 90, 44

MU - May 16, 10 Marks

Solution : Method I : Linear Probing

1. $23 \% 11 = 1$, $55 \% 11 = 0$

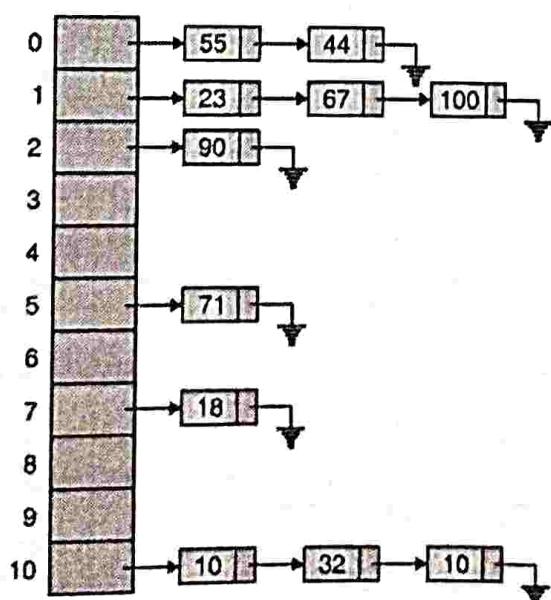
$10 \% 11 = 10,$
 $67 \% 11 = 1$
 $100 \% 11 = 1,$
 $10 \% 11 = 10,$
 $44 \% 11 = 0$

$71 \% 11 = 5$
 $32 \% 11 = 10$
 $18 \% 11 = 7$
 $90 \% 11 = 2$

2.

| | 23(1) | 55(0) | 10(10) | 71(5) | 67(1) | 32(10) | 100(1) | 18(7) | 10(10) | 90(2) | 44(0) |
|----|-------|-------|--------|-------|-------|--------|--------|-------|--------|-------|-------|
| 0 | - | 55 | 55 | 25 | 25 | 25 | 25 | 25 | 25 | 25 | 25 |
| 1 | 23 | 23 | 23 | 23 | 23 | 23 | 23 | 23 | 23 | 23 | 23 |
| 2 | - | - | - | - | 67 | 67 | 67 | 67 | 67 | 67 | 67 |
| 3 | - | - | - | - | - | 32 | 32 | 32 | 32 | 32 | 32 |
| 4 | - | - | - | - | - | - | 100 | 100 | 100 | 100 | 100 |
| 5 | - | - | - | 71 | 71 | 71 | 71 | 71 | 71 | 71 | 71 |
| 6 | - | - | - | - | - | - | - | - | 10 | 10 | 10 |
| 7 | - | - | - | - | - | - | - | 18 | 18 | 18 | 18 |
| 8 | - | - | - | - | - | - | - | - | - | 90 | 90 |
| 9 | - | - | - | - | - | - | - | - | - | - | 44 |
| 10 | - | - | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 |

Method 2 : Separate chaining hash function = data % 11.





Example 6.13.4 : Using Linear probing and Quadratic probing insert the following values in a hash table of size 10. Show how many collisions occur in each iteration : 28, 55, 67, 11, 10, 90, 44
MU - May 17, 10 Marks

Solution :

1. Linear probing

| | Empty table | After 28 | After 55 | After 71 | After 67 | After 1 | After 10 | After 90 | After 44 |
|---|-------------|----------|----------|----------|----------|---------|----------|----------|----------|
| 0 | | | | | | | 10 | 10 | 10 |
| 1 | | | | 71 | 71 | 71 | 71 | 71 | 71 |
| 2 | | | | | | 1 | 1 | 1 | 1 |
| 3 | | | | | | | | 90 | 90 |
| 4 | | | | | | | | | 44 |
| 5 | | | 55 | 55 | 55 | 55 | 55 | 55 | 55 |
| 6 | | | | | | | | | |
| 7 | | | | | 67 | 67 | 67 | 67 | 67 |
| 8 | 28 | 28 | 28 | 28 | 28 | 28 | 28 | 28 | 28 |
| 9 | | | | | | | | | |

Number of collisions = 2

2. Quadratic Probing

| | Empty table | After 28 | After 55 | After 71 | After 67 | After 1 | After 10 | After 90 | After 44 |
|---|-------------|----------|----------|----------|----------|---------|----------|----------|----------|
| 0 | | | | | | | 10 | 10 | 10 |
| 1 | | | | 71 | 71 | 71 | 71 | 71 | 71 |
| 2 | | | | | | 1 | 1 | 1 | 1 |
| 3 | | | | | | | | 90 | * |
| 4 | | | | | | | | | |
| 5 | | | 55 | 55 | 55 | 55 | 55 | 55 | 55 |
| 6 | | | | | | | | | |
| 7 | | | | | 67 | 67 | 67 | 67 | 67 |
| 8 | | 28 | 28 | 28 | 28 | 28 | 28 | 28 | 28 |
| 9 | | | | | | | | | 44 |

Example 6.13.5 : Using Linear probing and Quadratic probing, insert the following values in the hash table of size 10. Show how many collisions occur in each iteration : 28, 55, 71, 67, 11, 10, 90, 44

Linear probing

MU - May 18, 10 Marks

| Data to be Inserted | | | | | | | | | |
|---------------------|----|----|----|----|----|----|----|----|--|
| | 28 | 55 | 71 | 67 | 11 | 10 | 90 | 44 | |
| 0 | | | | | | 10 | 10 | 10 | |
| 1 | | | 71 | 71 | 71 | 71 | 71 | 71 | |
| 2 | | | | | 11 | 11 | 11 | 11 | |
| 3 | | | | | | | 90 | 90 | |
| 4 | | | | | | | | 44 | |
| 5 | | 55 | 55 | 55 | 55 | 55 | 55 | 55 | |
| 6 | | | | | | | | | |
| 7 | | | | 67 | 67 | 67 | 67 | 67 | |
| 8 | 28 | 28 | 28 | 28 | 28 | 28 | 28 | 28 | |
| 9 | | | | | * | | * | * | |
| | | | | | | | | | |

Collision occurred while inserting 11 and 90.

Total number of collisions = 2

Quadratic probing

| | 28 | 55 | 71 | 67 | 11 | 10 | 90 | 44 | Data |
|---|----|----|----|----|----|----|----|----|------|
| 0 | | | | | | 10 | 10 | 10 | |
| 1 | | | 71 | 71 | 71 | 71 | 71 | 71 | |
| 2 | | | | | 11 | 11 | 11 | 11 | |
| 3 | | | | | | | | 44 | |
| 4 | | | | | | | 90 | 90 | |
| 5 | | 55 | 55 | 55 | 55 | 55 | 55 | 55 | |
| 6 | | | | | | | | | |
| 7 | | | | 67 | 67 | 67 | 67 | 67 | |
| 8 | 28 | 28 | 28 | 28 | 28 | 28 | 28 | 28 | |
| 9 | | | | | * | | * | * | |
| | | | | | | | | | |

Number of collisions = 3, it happened while inserting 11, 90 and 44.



6.13.4(C) Primary Clustering

Primary clustering is a term used in the context of linear probing. Primary clustering is the tendency in open-addressing hash tables collision resolution schemes to create long sequences of filled slots.

- Primary clustering basically implies that all keys that collide at address x will extend the cluster that contains x .
- Records tend to cluster in the table under linear probing since the probabilities for which slot to use next are not the same for all slots.

Suggested Experiments

L.1 Implement Stack ADT using array

```
/* Array implementation of stack */  
/* Stack in 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("\nEnter a number :");  
            scanf("%d",&x);  
            push(&s,x);  
        }  
        case 2:  
        if(empty(&s))  
            printf("Stack is empty.....");  
        else  
            printf("Popped value = %d",pop(&s));  
        break;  
        case 3:  
        print(&s);  
        break;  
        case 4:  
        exit(0);  
    }  
}
```

```
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);  
}
```



```

    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

1)Push
2)Pop
3)Print
4)Quit
Enter Your choice: 1
enter a number :10
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: 1

enter a number :21
1)Push
2)Pop
3)Print
4)Quit
Enter Your choice: 3
21 15 10

1)Push
2)Pop
3)Print
4)Quit
Enter Your choice: 2
popped value= 21

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= 10

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.2 Convert an Infix expression to Postfix expression using stack ADT

L.3 Evaluate Postfix Expression using Stack ADT

```
/* Conversion of infix to postfix. Evaluation of postfix expression */
```

```
/* program for conversion of:
```

1. infix into its postfix form
2. infix into its prefix form
3. Evaluation of postfix expression
4. 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;
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 eval_postfix(char postfix[]);
int evaluate(char x,int op1,int op2);

void main()
{ char infix[30],postfix[30],prefix[30];
clrscr();
printf("\n Enter an infix expression : ");
gets(infix);
```

```
infix_to_postfix(infix,postfix);
printf("\n Postfix : %s ",postfix);
printf("\n Postfix evaluation : ");
eval_postfix(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';
}

void eval_postfix(char postfix[])
{
    stack s;
    char x;
    int op1,op2,val,i;
    init(&s);
    for(i=0;postfix[i]!='\0';i++)
    {
        x=postfix[i];
        if(isalpha(x))
        {
            printf("\nEnter the value of %c : ",x);
            scanf("%d",&val);
            push(&s,val);
        }
        else
        {
            //pop two operands and evaluate
            op2=pop(&s);
            op1=pop(&s);
            val=evaluate(x,op1,op2);
            push(&s,val);
        }
    }
    val=pop(&s);
    printf("\n value 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);
}

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 an infix expression : a+b*c
Postfix : abc*+
prefix: +a*b*c

```

Prefix Evaluation :
 Enter the value of c : 12
 Enter the value of b : 24
 Enter the value of a : 10
 value of expression = 298

Postfix evaluation :
 Enter the value of a : 2
 Enter the value of b : 3
 Enter the value of c : 4
 value of expression = 14

L.4 Implement Linear Queue ADT using Array

```
/* A program for queue using an array */
#include<conio.h>
#include<stdio.h>
#define MAX 10

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("\nEnter 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("\nDeleted 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+1)%MAX==P->F)
    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)%MAX;
    return(x);
}
void print(Q *P)
{
    int i;
    if(!empty(P))
    {
        printf("\n");
        for(i=P->F;i!=P->R;i=(i+1)%MAX)
            printf("%d\t",P->data[i]);
    }
    printf("%d\t",P->data[i]);
}

```

Output

- 1)Insert
 - 2)Search
 - 3)Print
 - 4)Quit
- Enter Your Choice : 1

Enter a number to be inserted:25

- 1)Insert
- 2)Search
- 3)Print
- 4)Quit

Enter Your Choice : 1

Enter a number to be inserted:35

- 1)Insert
- 2)Search
- 3)Print
- 4)Quit

Enter Your Choice : 1

Enter a number to be inserted:45

- 1)Insert
- 2)Search
- 3)Print
- 4)Quit

Enter Your Choice : 3

- (0) ---
- (1) ---
- (2) ---
- (3) ---
- (4) ---
- (5) 25
- (6) 35
- (7) 45
- (8) ---
- (9) ---

- 1)Insert
- 2)Search
- 3)Print
- 4)Quit

Enter Your Choice : 2

Enter a number to be searched :35

***Found at the location=6

- 1)Insert
- 2)Search
- 3)Print
- 4)Quit

```
Enter Your Choice : 2
Enter a number to be searched :15
****Element not found****
```

- 1)Insert
- 2)Search
- 3)Print
- 4)Quit

```
Enter Your Choice : 1
```

```
Enter a number to be inserted:27
```

- 1)Insert
- 2)Search
- 3)Print
- 4)Quit

```
Enter Your Choice : 3
```

- (0)---
- (1)---
- (2)---
- (3)---
- (4)---
- (5) 25
- (6) 35
- (7) 45
- (8) 27
- (9)---

- 1)Insert
- 2)Search
- 3)Print
- 4)Quit

```
Enter Your Choice : 4
```

L.5 Implement Circular Queue ADT using Array

```
/* Array implementation of Circular Queue.*/
/* A program for circular queue of strings using an array */
#include<conio.h>
#include<stdio.h>
```

```
#define MAX 10

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;
    int x,y;
    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("\nEnter a number:");
                        scanf("%d",&x);
                        if(!full(&q))
                            enqueue(&q,x);
                        else
                            printf("\nQueue is full !!!");
                        break;
            case 2:   if(!empty(&q))
                        {
                            y=dequeue(&q);
                            printf("\nDeleted Data=%d",y);
                        }
            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+1)%MAX==P->F)
        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)%MAX;
    return(x);
}

void print(Q *P)
{
    int i;
    if(!empty(P))
    {
        printf("\n");
        for(i=P->F;i!=P->R;i=(i+1)%MAX)
            printf("%d\t",P->data[i]);
        printf("%d\t",P->data[i]);
    }
}

```

Output

1)Insert
2)Delete
3)Print
4)Quit
Enter Your Choice:1
Enter a number: 11

1)Insert
2)Delete
3)Print
4)Quit
Enter Your Choice:1
Enter a number:20

1)Insert
2)Delete
3)Print
4)Quit
Enter Your Choice:1
Enter a number:5

1)Insert

2)Delete
3)Print
4)Quit
Enter Your Choice:1
Enter a number:13

1)Insert
2)Delete
3)Print
4)Quit
Enter Your Choice:3
11 20 5 13

1)Insert
2)Delete
3)Print
4)Quit
Enter Your Choice:2
Deleted Data=11

1)Insert
2)Delete
3)Print
4)Quit
Enter Your Choice:2
Deleted Data=20

1)Insert
2)Delete
3)Print
4)Quit
Enter Your Choice:2
Deleted Data=5

1)Insert
2)Delete
3)Print
4)Quit
Enter Your Choice:2
Deleted Data=13

1)Insert

2)Delete
3)Print
4)Quit
Enter Your Choice:2
Queue is empty !!!!

1)Insert
2)Delete
3)Print
4)Quit
Enter Your Choice:4

L.6 Implement Priority Queue ADT using Array

```
/* Array implementation of Priority Queue */
/* A program for static ascending priority queue of
   integers */

#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;
    int 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("\nEnter a number:");
        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)
{
    int i;
    P->R=-1;
    P->F=-1;
}
int empty(Q *P)
{
    if(P->R== -1)
        return(1);
    return(0);
}
int full(Q *P)
{
    int places,i;
    if(P->R==MAX-1)

```

```

    places=P->F;
    if(places > 0) //shift the queue
    {
        for(i=P->F;i<=P->R;i++)
            P->data[i-places]=P->data[i];
        P->F=0;
        P->R=P->R-places;
        for(i=P->R+1;i<MAX;i++)
            P->data[i]=NULL;
    }
}
if(P->R==MAX-1)
    return(1);
return(0);
}

void enqueue(Q *P,int x)
{
    int i;

    if(P->R== -1)
    {
        P->R=P->F=0;
        P->data[P->R]=x;
    }
    else
    {
        for(i=P->R;i>=P->F && x<P->data[i];i--)
            P->data[i+1]=P->data[i];
        P->R=P->R+1;
        P->data[i+1]=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=i+1)
            printf("%d\t",P->data[i]);
    }
}

```

Output

1)Insert
2)Delete
3)Print
4)Quit

Enter Your Choice:1

Enter a number: 11

1)Insert
2)Delete
3)Print
4)Quit

Enter Your Choice:1

Enter a number:20

1)Insert
2)Delete
3)Print
4)Quit

Enter Your Choice:1

Enter a number:5

1)Insert
2)Delete
3)Print
4)Quit

Enter Your Choice:1

Enter a number:13

1)Insert

2)Delete

3)Print

4)Quit

Enter Your Choice:3

5 11 13 20

1)Insert

2)Delete

3)Print

4)Quit

Enter Your Choice:2

Deleted Data=5

1)Insert

2)Delete

3)Print

4)Quit

Enter Your Choice:2

Deleted Data=11

1)Insert

2)Delete

3)Print

4)Quit

Enter Your Choice:2

Deleted Data=13

1)Insert

2)Delete

3)Print

4)Quit

Enter Your Choice:2

Deleted Data=20

1)Insert

2)Delete

3)Print

4)Quit

Enter Your Choice:2

Queue is empty !!!!

1)Insert



- 2)Delete
3)Print
4)Quit
Enter Your Choice:4

L.7 Implement Singly Linked List ADT

Write a C program to implement a singly linked list. The program should be able to perform the following operations.

- (i) Insert a node at the end of the list.
- (ii) Deleting a particular element.
- (iii) Display the linked list

MU - Dec. 18, 8 Marks

```
/* : Implementation of single link 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);
node *copy(node *);
int count(node *);
node *concatenate(node *, node *);
void split(node *);
void main()
{
  int op,op1,x;
  node *head=NULL;
  node *head1=NULL,*head2=NULL,
  *head3=NULL;
```

```
clrscr();
do
{
  printf("\n\n 1)create\n 2)Insert\n 3>Delete\n
  4)Search");
  printf("\n 5)Reverse\n 6)Print\n 7)Count\n 8)Copy\n
  9)Concatenate");
  printf("\n 10)Split\n 11)Quit");
  printf("\n Enter your Choice:");
  scanf("%d",&op);
  switch(op)
  {
    case 1:head=create();break;
    case 2:printf("\n\t 1)Beginning\n\t 2)End\n\t
    3)In between");
      printf("\n Enter your choice : ");
      scanf("%d",&op1);
      printf("\n Enter 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\t 1)Beginning\n\t 2)End\n\t
    3)In between");
      printf("\n Enter 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;
case 7: printf("\nNo.of node = %d",count(head));
    break; //count
case 8: head1=copy(head);//copy
    printf("\nOriginal Linked List :");
    print(head);
    printf("\nCopied Linked List :");
    print(head1);
    break;
case 9:printf("\nEnter the first linked list:");
    head1=create();
    printf("\nEnter the second linked list:");
    head2=create();
    head3=concatenate(head1,head2);
    printf("\nConcatenated Linked List :");
    print(head3);
    break;
//concatenate
case 10:printf("\nEnter a linked list :");
    head1=create();
    split(head1);
    break;
//split
}
}while(op!=11);
}

node *create()
{ node *head,*p;
int i,n;
head=NULL;
printf("\nEnter 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("\n Insert after which number ? :");

```



```

scanf("%d",&y);
//locate the the 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("\n Data 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("\n Underflow....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("\n Underflow....Empty Linked List");
        return(head);
    }
    printf("\n Enter 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("\n Underflow....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("\n Enter the data to be searched: ");
scanf("%d",&data);
p=head;
while(p!=NULL && p->data != data)
{ loc++;
  p=p->next;
}
if(p==NULL)
printf("\n Not found:");
else
printf("\n Found 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);
}

node *copy(node *h)
{
node *head=NULL,*p;
if(h==NULL)
  return head;
//Copy the first node
head=p=(node*)malloc(sizeof(node));

```

```

p->data=h->data;
while(h->next != NULL)
{
  p->next=(node*)malloc(sizeof(node));
  p=p->next;
  h=h->next;
  p->data=h->data;
}
p->next=NULL;
return (head);
}

int count(node *h)
{
int i;
for(i=0; h!=NULL; h=h->next)
  i++;
return(i);
}

node *concatenate( node *h1, node * h2)
{
node *p;
if(h1==NULL)
  return(h2);
if(h2==NULL)
  return(h1);
p=h1;
while(p->next != NULL)
//goto the end of the 1st linked list
p=p->next;
p->next=h2;
return(h1);
}

void split(node *h1)
{
node *p,*q,*h2;
printf("\n Linked list to be split : ");
print(h1);
/*linked list will be broken from the centre using
the pointers p and q*/
if(h1==NULL)
  return;
p=h1;

```



```

q=h1->next;
while(q!=NULL && q->next != NULL)
{
    q=q->next->next;
    p=p->next;
/*When q reaches the last node,p will reach the
centre node*/
}
h2=p->next;
p->next=NULL;
printf("\nFirst half : ");
print(h1);
printf("\nSecond half : ");
print(h2);
}

```

Output

1)create
 2)Insert
 3)Delete
 4)Search
 5)Reverse
 6)Print
 7)Count
 8)Copy
 9)Concatenate
 10)Split
 11)Quit

Enter your Choice:1

Enter no of data:5

Enter the data:10 5 3 6 24

1)create
 2)Insert
 3)Delete
 4)Search
 5)Reverse
 6)Print
 7)Count
 8)Copy
 9)Concatenate
 10)Split

11)Quit
 Enter your Choice:2
 1)Beginning
 2)End
 3)In between

Enter your choice : 3

Enter the data to be inserted : 17

Insert after which number ? : 3

- 1)create
- 2)Insert
- 3)Delete
- 4)Search
- 5)Reverse
- 6)Print
- 7)Count
- 8)Copy
- 9)Concatenate
- 10)Split
- 11)Quit

Enter your choice : 6

10 5 3 17 6 24

- 1)create
- 2)Insert
- 3)Delete
- 4)Search
- 5)Reverse
- 6)Print
- 7)Count
- 8)Copy
- 9)Concatenate
- 10)Split
- 11)Quit

Enter your choice : 3

- 1)Beginning
- 2)End
- 3)In between

Enter your choice : 1

- 1)create

2)Insert

3)Delete

4)Search

5)Reverse

6)Print

7)Count

8)Copy

9)Concatenate

10)Split

11)Quit

Enter your Choice:6

5 3 17 6 24

1)create

2)Insert

3)Delete

4)Search

5)Reverse

6)Print

7)Count

8)Copy

9)Concatenate

10)Split

11)Quit

Enter your Choice:4

Enter the data to be searched: 24

Found at location=5

1)create

2)Insert

3)Delete

4)Search

5)Reverse

6)Print

7)Count

8)Copy

9)Concatenate

10)Split

11)Quit

Enter your Choice:4

Enter the data to be searched: 10

Not found:

1)create

2)Insert

3)Delete

4)Search

5)Reverse

6)Print

7)Count

8)Copy

9)Concatenate

10)Split

11)Quit

Enter your Choice:5

24 6 17 3 5

1)create

2)Insert

3)Delete

4)Search

5)Reverse

6)Print

7)Count

8)Copy

9)Concatenate

10)Split

11)Quit

Enter your Choice:7

No.of node = 5

1)create

2)Insert

3)Delete

4)Search

5)Reverse

6)Print

7)Count

8)Copy

9)Concatenate

10)Split

11)Quit

Enter your Choice:8



Original Linked List :

24 6 17 3 5

Copied Linked List :

24 6 17 3 5

- 1)create
- 2)Insert
- 3)Delete
- 4)Search
- 5)Reverse
- 6)Print
- 7)Count
- 8)Copy
- 9)Concatenate
- 10)Split
- 11)Quit

Enter your Choice:9

Enter the first linked list:

Enter no of data:5

Enter the data:10 21 3 37 85

Enter the second linked list:

Enter no of data:3

Enter the data:41 52 66

Concatenated Linked List :

10 21 3 37 85 41 52 66

- 1)create
- 2)Insert
- 3)Delete
- 4)Search
- 5)Reverse
- 6)Print
- 7)Count
- 8)Copy
- 9)Concatenate
- 10)Split
- 11)Quit

Enter your Choice:10

Enter a linked list :

Enter no of data:3

Enter the data:24 65 89

Linked list to be split :

24 65 89

First half :

24 65

Second half :

89

- 1)create
- 2)Insert
- 3)Delete
- 4)Search
- 5)Reverse
- 6)Print
- 7)Count
- 8)Copy
- 9)Concatenate
- 10)Split
- 11)Quit

Enter your Choice:11

L.8 Implement Circular Linked List ADT

```
/*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("\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;
case2:printf("\n\t1)Beginning\n\t2)End\n\t
3)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;
case3:printf("\n\t1)Beginning\n\t2)End\n\t
3)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("\nEnter 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("\nEnter 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;
            head=p;
        }
    }
    else
    {
        p->next=head->next;
        head->next=p;
        head=p;
    }
    return(head);
}

```

```

    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 Implement Doubly Linked List ADT

```

/* Implementation of Doubly linked list. */
#include<conio.h>
#include<stdlib.h>
#include<stdio.h>
typedef struct dnode
{
    int data;
    struct dnode *next,*prev;
}dnode;
dnode * create();
void print_forward(dnode *);
void print_reverse(dnode *);
void main()
{
    dnode *head;
    head=NULL; // initially the list is empty
    head=create();
    printf("\nElements in forward direction :");
    print_forward(head);
    printf("\nElements in reverse direction :");
    print_reverse(head);
}
dnode *create()
{
    dnode *h,*P,*q;
    int i,n,x;
    h=NULL;
    printf("\nEnter no of elements :");
    scanf("%d",&n);
    for(i=0;i<n;i++)
    {
        printf("\nEnter next data: ");
        scanf("%d",&x);
        q=(dnode*)malloc(sizeof(dnode));
        q->data=x;
        q->prev=q->next=NULL;
        if(h==NULL)
            P=h=q;
        else

```

```

            P->next=q;
            q->prev=P;
            P=q;
    }
    return(h);
}
void print_forward(dnode *h)
{
    while(h!=NULL)
    {
        printf("<- %d ->",h->data);
        h=h->next;
    }
}
void print_reverse(dnode *h)
{
    while(h->next!=NULL)
        h=h->next;
    while(h!=NULL)
    {
        printf("<- %d ->",h->data);
        h=h->prev;
    }
}

```

Output

```

Enter no of elements :4
Enter next data:1
Enter next data:2
Enter next data:3
Enter next data:4
Elements in forward direction :
<-1-> <-2-> <-3-> <-4->
Elements in reverse direction :
<-4-> <-3-> <-2-> <-1->

```

L.10 Implement Stack ADT using Linked List

```

/* Linked implementation of stack.*/
#include<stdio.h>
#include<conio.h>

```



```

typedef struct stack
{
    int data;
    struct stack *next;
} stack;
void init(stack **);
int empty(stack *);
int pop(stack **);
void push(stack **,int);
void print(stack *p);
void main()
{
    stack *TOP;
    int x,op;
    init(&TOP);
    clrscr();
    do {
        printf("\n\n1)Push\n2)Pop\n3)Print\n4)Quit");
        printf("\nEnter Your choice: ");
        scanf("%d",&op);
        switch(op)
        {
            case 1:printf("\nEnter a number :");
                      scanf("%d",&x);
                      push(&TOP,x);
                      break;
            case 2:if(!empty(TOP))
                      {
                          x=pop(&TOP);
                          printf("\npopped value = %d",x);
                      }
            else
                printf("\nStack is empty.....");
                break;
            case 3:print(TOP);break;
        }
    }while(op!=4);
}

void init(stack **T)
{
    *T=NULL;
}

```

```

}
int empty(stack *TOP)
{
    if(TOP==NULL)
        return(1);
    return(0);
}
void push(stack **T,int x)
{
    stack *P;
    P=(stack *)malloc(sizeof(stack));
    P->data=x;
    P->next=*T;
    *T=P;
}
int pop(stack **T)
{
    int x;
    stack * P;
    P=*T;
    *T=P->next;
    x=P->data;
    free(P);
    return(x);
}
void print(stack *p)
{
    printf("\n");
    while(p!=NULL)
    {
        printf(" %d ",p->data);
        p=p->next;
    }
}

```

Output

- 1)Push
 - 2)Pop
 - 3)Print
 - 4)Quit
- Enter Your choice: 1
 enter a number :1
 1)Push

- 2)Pop
3)Print
4)Quit

Enter Your choice: 1
enter a number :2

- 1)Push
2)Pop
3)Print
4)Quit

Enter Your choice: 1
enter a number :3

- 1)Push
2)Pop
3)Print
4)Quit

Enter Your choice: 3
3 2 1

- 1)Push
2)Pop
3)Print
4)Quit

Enter Your choice: 2
popped value= 3

- 1)Push
2)Pop
3)Print
4)Quit

Enter Your choice: 2
popped value= 2

- 1)Push
2)Pop
3)Print
4)Quit

Enter Your choice: 2
popped value= 1

- 1)Push
2)Pop
3)Print
4)Quit

Enter Your choice: 4

L.11 Implement Linear Queue ADT using Linked List

```
#include<conio.h>
#include<stdio.h>
#define MAX 10
typedef struct node
{
    int data;
    struct node *next;
}node;
typedef struct Q
{
    node *R,*F;
}Q;
void initialise(Q *);
int empty(Q *);
int full(Q *);
void enqueue(Q *,int);
int dequeue(Q *);
void print(Q *);

void main()
{
    Q q;
    int x,i,op;
    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("\nEnter a value:");
                      scanf("%d",&x);
                      enqueue(&q,x);
            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 *qP)
{
    qP->R=NULL;
    qP->F=NULL;
}

void enqueue(Q *qP,int x)
{
    node *P;
    P=(node*)malloc(sizeof(node));
    P->data=x;
    P->next=NULL;
    if(empty(qP))
    {
        qP->R=P;
        qP->F=P;
    }
    else
    {
        (qP->R)->next=P;
        qP->R=P;
    }
}

int dequeue(Q *qP)
{
    int x;
    node *P;
    P=qP->F;
    x=P->data;
    if(qP->F==qP->R) //deleting the last element

```

```

initialise(qP);

else
    qP->F=P->next;
free(P);
return(x);

}

void print(Q *qP)
{
    int i;
    node *P;
    P=qP->F;
    while(P!=NULL)
    {
        printf("\n%d",P->data);
        P=P->next;
    }
}

int empty(Q *qp)
{
    if(qp->R==NULL)
        return 1;
    return 0;
}

```

Output

```

1)Insert
2)Delete
3)Print
4)Quit
Enter Your Choice:1
Enter a value:1
1)Insert
2)Delete
3)Print
4)Quit
Enter Your Choice:1
Enter a value:2
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:4
 1)Insert
 2>Delete
 3)Print
 4)Quit
 Enter Your Choice:3
 1
 2
 3
 4
 1)Insert
 2>Delete
 3)Print
 4)Quit
 Enter Your Choice:2
 Deleted Data=1
 1)Insert
 2>Delete
 3)Print
 4)Quit
 Enter Your Choice:2
 Deleted Data=2
 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:2
 Deleted Data=4
 1)Insert

2)Delete
 3)Print
 4)Quit
 Enter Your Choice:2
 Queue is empty !!!!
 1)Insert
 2>Delete
 3)Print
 4)Quit
 Enter Your Choice:4

L.12 Implement Binary Search Tree ADT using Linked List

```
/* Implement 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 *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
temp=find_min(T->right);
T->data=temp->data;
T->right=delet(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 :6
Enter tree values :11 2 5 21 3 6
**** BST created ****
inorder traversal on the tree 2 3 5 6 11 21
smallest key in the tree = 2
largest key in the tree = 21
**** delete operation ****
Enter the key to be deleted :3
inorder traversal after deletion : 2 5 6 11 21
Enter no. of nodes :6
Enter tree values :11 2 5 3 21 6
**** BST created ****
inorder traversal on the tree 2 3 5 6 11 21
smallest key in the tree = 2
largest key in the tree = 21
**** delete operation ****
Enter the key to be deleted :21
inorder traversal after deletion : 2 3 5 6 11

Enter no. of nodes :6
Enter tree values :11 2 5 3 21 6
**** BST created ****
inorder traversal on the tree 2 3 5 6 11 21
smallest key in the tree = 2
largest key in the tree = 21

```



```
**** delete operation ****
```

Enter the key to be deleted :15

Element not found : inorder traversal after deletion :

2 3 5 6 11 21

L.13 Implement Graph Traversal Techniques:

- a) Depth First Search
- b) Breadth First Search

```
/* Implementation of Depth First search and Breadth First Search */
#include<conio.h>
#include<stdio.h>
#include<stdlib.h>
#define MAX 20
typedef struct Q
{
    int data[MAX];
    int R,F;
}Q;
typedef struct node
{
    struct node *next;
    int vertex;
}node;
void enqueue(Q *,int);
int dequeue(Q *);
int empty(Q *);
int full(Q *);
void BFS(int);
void readgraph();           //create an adjacency list
void insert(int vi,int vj); //insert an edge (vi,vj)in adj.list
void DFS(int i);
int visited[MAX];
node *G[20];                //heads of the linked list
int n;                      // no. of nodes

void main()
{
    int i,op;
    clrscr();
}
```

```
do
{
    printf("\n\n1)Create\n2)BFS\n3)DFS\n4)Quit");
    printf("\nEnter Your Choice: ");
    scanf("%d",&op);
    switch(op)
    {
        case 1: readgraph();break;
        case 2: printf("\nStarting Node No. : ");
                  scanf("%d",&i);
                  BFS(i);break;
        case 3: for(i=0;i<n;i++)
                  visited[i]=0;
                  printf("\nStarting Node No. : ");
                  scanf("%d",&i);
                  DFS(i);break;
    }
}while(op!=4);
}

void BFS(int v)
{
    int w,i,visited[MAX];
    Q q;
    node *p;
    q.R=q.F=-1;           //initialise
    for(i=0;i<n;i++)
        visited[i]=0;
    enqueue(&q,v);
    printf("\nVisit\t%d",v);
    visited[v]=1;
    while(!empty(&q))
    {
        v=dequeue(&q);
        /*insert all unvisited,adjacent vertices of v
        into queue*/
        for(p=G[v];p!=NULL;p=p->next)
        {
            w=p->vertex;
            if(visited[w]==0)
            {
                enqueue(&q,w);
            }
        }
    }
}
```

```

        visited[w]=1;
        printf("\nvisit(%d",w);
    }
}

void DFS(int i)
{
    node *p;
    printf("\n%d",i);
    p=G[i];
    visited[i]=1;
    while(p!=NULL)
    {
        i=p->vertex;
        if(!visited[i])
            DFS(i);
        p=p->next;
    }
}
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;
        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 readgraph()
{
    int i,vi,vj,no_of_edges;
    printf("\nEnter no. of vertices :");
    scanf("%d",&n);
    //initialise G[] with 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);
        insert(vj,vi);
    }
}

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 for the vertex no.
     vi */
    if(G[vi]==NULL)

```



```

G[vi]=q;
else
{
    // goto the end of linked list
    p=G[vi];
    while(p->next!=NULL)
        p=p->next;
    p->next=q;
}
}

```

Output

- 1)Create
- 2)BFS
- 3)DFS
- 4)Quit

Enter Your Choice: 1

Enter no. of vertices :4

Enter no of edges :4

Enter an edge (u,v) :2 9

Enter an edge (u,v) :5 7

Enter an edge (u,v) :12 6

Enter an edge (u,v) :8 7

- 1)Create

- 2)BFS

- 3)DFS

- 4)Quit

Enter Your Choice: 2

Starting Node No. : 2

Visit 2

- 1)Create

- 2)BFS

- 3)DFS

- 4)Quit

Enter Your Choice: 3

Starting Node No. : 5

5

7

8

- 1)Create

- 2)BFS

3)DFS

4)Quit

Enter Your Choice:4

L.14 Applications of Binary Search Technique

```

/* Implementation of binary search.*/
#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
```



Note

Appendix A

Solved University Question Papers of Dec. 2017, May 2018 and Dec. 2018

Data Structure

Dec. 2017

- Q. 1(a) Explain ADT. List the Linear and Non-linear data structures with example.
(Sections 1.1.2 and 1.2.1(B)) (5 Marks)
- Q. 1(b) Explain B Tree and B+ Tree.
(Sections 4.14 and 4.15) (5 Marks)
- Q. 1(c) Write a program to implement Binary Search on sorted set of Integers.
(Program 6.3.1) (10 Marks)
- Q. 2(a) Write a program to convert Infix expression into Postfix equation. **(Program 2.4.2)** (10 Marks)
- Q. 2(b) Explain Huffman Encoding with an example.
(Section 4.13.1) (10 Marks)
- Q. 3(a) Write a program to implement Doubly Linked List. Perform the following operations :
(i) Insert a node in the beginning
(ii) Insert a node in the end
(iii) Delete a node from the end
(iv) Display the list
(Program 3.4.3) (10 Marks)
- Q. 3(b) Explain Topological sorting with example.
(Section 5.4) (10 Marks)
- Q. 4(a) Write a program to implement Quick sort. Show the steps to sort the given numbers :
25, 13, 7, 34, 56, 23, 13, 96, 14, 2
(Program 6.8.1) (10 Marks)
- Q. 4(b) Write a program to implement linear queue using array. **(Program 2.13.1)** (10 Marks)
- Q. 5(a) Write a program to implement STACK using linked list. What are the advantages of linked-list over array ? **(Program 3.7.1 and Section 3.1.1)** (10 Marks)
- Q. 5(b) Write a program to implement Binary Search Tree (BST), show BST for the following input: 10, 5, 4, 12, 15, 11, 3 **(Program 4.10.1)** (10 Marks)
- Q. 6(a) Write short note on AVL Tree
(Section 4.11) (10 Marks)

Q. 6(b) Write short note on Graph Traversal Techniques
(Sections 5.3, 5.3.1 and 5.3.2) (10 Marks)

Q. 6(c) Write short note on Expression Trees
(Section 4.12.1) (10 Marks)

Q. 6(d) Write short note on Application of Linked list-Polynomial Addition.
(Sections 3.6.1 and 3.6.2) (10 Marks)

May 2018

- Q. 1(a) Explain different types of data structures with example. **(Section 1.2.1)** (5 Marks)
- Q. 1(b) What is a graph? Explain methods to represent graph. **(Sections 5.1.1 and 5.2)** (5 Marks)
- Q. 1(c) Write a program in 'C' to implement Merge sort.
(Program 6.9.1) (10 Marks)
- Q. 2(a) Write a program in 'C' to implement QUEUE ADT using Linked-List. Perform the following operations :
(i) Insert a node in the Queue.
(ii) Delete a node from the Queue.
(iii) Display Queue elements
(Program 3.8.1) (10 Marks)
- Q. 2(b) Using Linear probing and Quadratic probing, insert the following values in the hash table of size 10. Show how many collisions occur in each iteration : 28, 55, 71, 67, 11, 10, 90, 44
(Example 6.13.5) (10 Marks)
- Q. 3(a) Write a program in 'C' to evaluate postfix expression using STACK ADT.
(Program 2.4.1) (10 Marks)
- Q. 3(b) Explain different types of tree traversals techniques with example. Also write recursive function for each traversal technique.
(Sections 4.7, 4.7.1(A), 4.7.2(B) and 4.7.3(C)) (10 Marks)
- Q. 4(a) State advantages of Linked-List over arrays. Explain different applications of Linked-list.
(Sections 3.1.1, 3.6.1, 3.7 and 3.8) (10 Marks)



- Q. 4(b)** Write a program in 'C' to implement Circular Queue using arrays.
(Program 2.14.1) **(10 Marks)**
- Q. 5(a)** Write a program to implement Singly Linked List. Provide the following operations :
 (i) Insert a node at the specified location
 (ii) Delete a node from end
 (iii) Display the list
(Example 3.2.3) **(10 Marks)**
- Q. 5(b)** Insert the following elements in AVL tree: 44, 17, 32, 78, 50, 88, 48, 62, 54. Explain different rotations that can be used.
(Example 4.11.11) **(10 Marks)**
- Q. 6(a)** Explain the following Splay Tree and Trie.
(Sections 4.16 and 4.17) **(10 Marks)**
- Q. 6(b)** Explain the following Graph Traversal Techniques.
(Sections 5.3, 5.3.1 and 5.3.2) **(10 Marks)**
- Q. 6(c)** Explain the following Huffman Encoding.
(Section 4.13.1) **(10 Marks)**
- Q. 6(d)** Explain the following Double Ended Queue.
(Section 2.16.2) **(10 Marks)**

Dec. 2018

- Q. 1(a)** What are various operations possible on data structures.
(Section 1.3.1) **(5 Marks)**
- Q. 1(b)** What are different ways of representing a Graph data structure on a computer.
(Section 5.2) **(5 Marks)**
- Q. 1(c)** Describe Tries with an example.
(Section 4.17) **(5 Marks)**
- Q. 1(d)** Write a function in C to implement binary search.
(Section 6.3) **(5 Marks)**
- Q. 2(a)** Use stack data structure to check well-formedness of parentheses in an algebraic expression. Write C program for the same.
(Program 2.3.4) **(10 Marks)**
- Q. 2(b)** Given the frequency for the following symbols, compute the Huffman code for each symbol.

| Symbol | A | B | C | D | E |
|-----------|----|----|----|---|---|
| Frequency | 24 | 12 | 10 | 8 | 8 |

(Example 4.13.6) **(10 Marks)**

- Q. 3(a)** Write a C program to implement priority queue using arrays. The program should perform the following operations:
 (i) Inserting in a priority queue.
 (ii) Deletion from a queue.
 (iii) Displaying contents of the queue.
(Program 2.16.1) **(12 Marks)**
- Q. 3(b)** What are expression trees? What are its advantages? Derive the expression tree for the following algebraic expression :
 $(a + (b/c)) * ((d/e) - f)$
(Section 4.12.1 and Example 4.12.3) **(8 Marks)**
- Q. 4(a)** Write a C program to represent and add two polynomials using linked list.
(Program 3.6.2) **(12 Marks)**
- Q. 4(b)** How does the Quicksort technique work ? Give C function for the same.
(Section 6.8 and Program 6.8.1) **(8 Marks)**
- Q. 5(a)** What is a doubly linked list? Give C representation for the same.
(Section 3.4 and Program 3.4.1) **(5 Marks)**
- Q. 5(b)** Given the postorder and inorder traversal of a binary tree, construct the original tree :
 Postorder : D F E B G L J K H C A
 Inorder : D B F E A G C L J H K
(Example 4.9.3) **(10 Marks)**
- Q. 5(c)** What is hashing? What properties should a good hash function demonstrate?
(Sections 6.13.1 and 6.13.3(A)) **(5 Marks)**
- Q. 6(a)** Given an array int a[]={69, 78, 63, 98, 67, 75, 66, 90, 81}. Calculate address of a[5] if base address is 1600.
(Example 1.9.2) **(2 Marks)**
- Q. 6(b)** Give C function for Breadth First search Traversal of a graph. Explain the code with an example.
(Program 5.3.3 and Example 5.3.2) **(10 Marks)**
- Q. 6(c)** Write a C program to implement a singly linked list. The program should be able to perform the following operations.
 (i) Insert a node at the end of the list.
 (ii) Deleting a particular element.
 (iii) Display the linked list
(L.7 - Suggested Experiments) **(8 Marks)**

Appendix B

Solved University Question Paper of May 2019 and Dec. 2019

Data Structure

May 2019

Q. 1(a) Explain Linear and Non-Linear data structures.
(Section 1.2.12) (5 Marks)

Q. 1(b) Explain Priority Queue with example.
(Section 2.16) (5 Marks)

Q. 1(c) Write a program in 'C' to implement Quick sort.
(Program 6.8.8) (10 Marks)

Q. 2(a) Write a program to implement Circular Linked List.
Provide the following operations :
(i) Insert a node .
(ii) Delete a node
(iv) Display the list
(Section 3.3) (10 Marks)

Q. 3(a) Explain Huffman Encoding with suitable example.
(Sections 4.13.1 and 4.13.2) (10 Marks)

Q. 3(b) Write a program in 'C' to check for balanced parenthesis in an expression using stack.
(Program 2.3.4) (10 Marks)

Q. 4(a) Write a program in 'C' to implement Queue using array.
(Program 2.13.1) (10 Marks)

Q. 4(b) Explain different cases for deletion of a node in binary search tree. Write function for each case.
(Section 4.10.2(F)) (10 Marks)

Q. 5(a) Write a program in 'C' to implement Stack using Linked-List. Perform the following operations :
(i) Push
(ii) Pop
(iii) Peek

(iv) Display the stack contents

(Program 3.7.2) (10 Marks)

Q. 5(b) Explain Depth First Search (DFS) Traversal with an example. Write the recursive function for DFS.
(Section 5.3.1 and Program 5.3.1) (10 Marks)

Q. 6(a) Write Short note on : Application of Linked-List – Polynomial addition. **(Section 3.6.2)** (10 Marks)

Q. 6(b) Write Short note on : Collision Handling techniques.
(Section 6.13.4) (10 Marks)

Q. 6(c) Write Short note on : Expression Tree.
(Section 4.12.1) (10 Marks)

Q. 6(d) Write Short note on : Topological Sorting.
(Section 5.4) (10 Marks)

Dec. 2019

Q. 1(a) Define Data Structure. Differentiate linear and non-linear data structures with example.
(Sections 1.2 and 1.2.1.2) (5 Marks)

Q. 1(b) Write a C function to implement Insertion sort.
(Section 6.5) (5 Marks)

Q. 1(c) What are different ways to represent graphs in memory? **(Section 5.2)** (5 Marks)

Q. 1(d) What is expression tree? Derive an expression tree for $(a + (b * c)) / ((d - e) * f)$
(Section 4.12.1 and Example 4.12.4) (5 Marks)

Q. 2(a) What is Hashing ? Hash the following data in a table of size 10 using linear probing and quadratic probing. Also find the number of collisions 63, 82, 94, 77, 53, 87, 23, 55, 10, 44
(Section 6.13.1 and Similar to Example 6.13.2)
(10 Marks)



Q. 2(b) Write a recursive function to perform pre-order traversal of a binary tree.
(Section 4.7.1(A)) (8 Marks)

Q. 2(c) Given an array int a[] = {23, 55, 63, 89, 45, 67, 85, 99}. Calculate address of a[5] if base address is 5100.
(Example 1.9.3) (2 Marks)

Q. 3(a) Write a C program to convert infix expression to postfix expression. **(Program 2.4.2)** (10 Marks)

Q. 3(b) Demonstrate step by step insertion of the following elements in an AVL tree.
63, 9, 19, 18, 108, 99, 81, 45
(Similar to Example 4.11.11) (10 Marks)

Q. 4(a) Write a C program to implement circular linked list that performs following functions

Insert a node in the beginning

Insert a node in the end

Count the number of nodes

Display the list **(Example 3.2.2)** (12 Marks)

Q. 4(b) Given the frequency for the following symbols, compute the Huffman code for each symbol.

| Symbol | A | B | C | D | E | F |
|-----------|---|----|---|----|----|----|
| Frequency | 9 | 12 | 5 | 45 | 16 | 13 |

(Similar to Example 4.13.1) (8 Marks)

Q. 5(a) Explain Double Ended Queue. Write a C program to implement Double Ended Queue.
(Sections 2.16.2 and 2.16.3) (12 Marks)

Q. 5(b) Given the postorder and inorder traversal of a binary tree, construct the original tree :

Postorder : D E F B G L J K H C A

Inorder : D B F E A G C L J H K

(Example 4.9.3) (8 Marks)

Q. 6(I) Explain following with suitable example : B-tree and splay tree **(Sections 4.16 and 4.14)** (10 Marks)

Q. 6(II) Explain following with suitable example : Polynomial representation and addition using linked list
(Section 3.6.2) (10 Marks)

Q. 6(III) Explain following with suitable example : Topological Sorting. **(Section 5.4)** (10 Marks)

Note

The page features a series of horizontal black lines for writing. A single vertical line runs vertically along the left edge of the page, creating a margin. The horizontal lines are evenly spaced and extend across the width of the page.

Your Success is Our Goal.

Semester III - Computer Engineering

ENGINEERING MATHEMATICS - III

Dr. Narendrakumar Dasre, Dr. Sachin Wani, Dr. Pritam Chetan Wani

DISCRETE STRUCTURES AND GRAPH THEORY

Dr. Bhakti Raul-Palkar

DATA STRUCTURE

Dilip Kumar Sultania

DIGITAL LOGIC & COMPUTER ORGANIZATION & ARCHITECTURE

J. S. Katre, Harish G. Narula

COMPUTER GRAPHICS

Dr. Mahesh M. Goyani

eBooks are available on www.techknowledgebooks.com

coming soon.....



es
easy-solutions

Paper Solutions Trusted by lakhs of students from more than 20 years



Head Office :
B/5, First Floor, Maniratna Complex, Taware Colony,
Aranyeshwar Corner, Pune - 411009. Maharashtra, India.
Tel. : 91-20-24221234, 91-20-24225678



ISBN : 978-93-89889-98-7



9 789389 889987

Price ₹ 485/-

M0146C



Distributors

Vidyarthi Sales Agencies

T. : (022) 23867279, ☎ 98197 76110

Ved Book Distributors

80975 71421 / 92208 77214 / ☎ 80973 75002

Bharat Sales Agency

T. : (022) 23819359, ☎ 86572 92797

Our Branches : Pune | Mumbai | Kolhapur | Nagpur | Solapur | Nashik

For Library Orders Contact - Ved Book Distributors M : 80975 71421 / 80973 75002

Email : info@techknowledgebooks.com

Website : www.techknowledgebooks.com

Like us at:



TechknowledgePublications