SRM VALLIAMMAI ENGINEERINGCOLLEGE

(An Autonomous Institution)

SRM Nagar, Kattankulathur-603203.

DEPARTMENT OF COMPUTER APPLICATIONS

Regulation 2024 - Lab Manual *for* I Semester

MC4165 DATA STRUCTURES LABORATORY

Academic Year 2024-2025 (ODD Semester)

Prepared by

Dr. D. Sridevi
Associate Professor
Department of Computer Applications

MC4165 DATA STRUCTURES LABORATORY L T P C

0 0 2 1

COURSE OBJECTIVES:

- To understand the usage of advanced tree structures.
- To familiarize the usage of heap structures.
- To learn the usage of graph data structures and spanning trees.
- To learn how to analyze the complexities of algorithms.
- To explore the various design strategies of algorithms.

EXPERIMENTS:

1. Implementation of Stack ADT and Queue ADT.

2. Implementation of Binary Search tree and its operations.

3. Implementation of AVL tree and its operations.

4. Implementation of Hashing techniques such as separate chaining.

5. Implementation of representation of graphs and topological sort.

6. Implementation of a spanning tree for a given graph using Prim's algorithm.

7. Implementation of shortest path algorithms such as Dijkstra's algorithm.

8. Implementation of iterative and recursive algorithms with its complexity analysis.

9. Implementation of Merge sort algorithm analysis using divide and conquer approach.

10. Implementation of matrix chain multiplication using dynamic programming approach.

11. Implementation of Huffman coding using greedy approach

TOTAL: 60 PERIODS

COURSE OUTCOMES:

On completion of the course, the students will be able to:

1. Implement basic and advanced data structures extensively.

2. Choose and apply suitable hierarchical data structures for real time problems.

3. Apply suitable heap data structures based on the problem requirements.

4. Design and apply algorithms using graph structures.

5. Design and implement iterative and recursive algorithms with minimum complexity.

6. Design and develop efficient algorithms by adopting suitable algorithm design strategies

SOFTWARE REQUIREMENTS

Operating Systems: Linux / Windows 7 or higher

Software: Turbo C / C / C++ / Equivalent Software.

CO – PO – PSO Mapping

CO	POs					
	PO1	**PO2**	**PO3**	**PO4**	**PO5**	**PO6**
1	2	1	3	3	2	2
2	2	1	3	3	2	2
3	1	1	3	2	2	2
4	2	1	3	2	2	2
5	2	1	3	3	2	3
Avg	**1.8**	**1**	**3**	**2.6**	**2**	**2.2**

PROGRAM EDUCATIONAL OBJECTIVES (PEOs)

1.To prepare students with breadth of knowledge to comprehend, analyze, design and create computing solutions to real-life problems and to excel in industry/ technical profession.

2.To provide students with solid foundation in mathematical and computing fundamentals and techniques required to solve technology related problems and to pursue higher studies and research.

3.To inculcate a professional and ethical attitude in students, to enable them to work towards a broad social context.

4.To empower students with skills required to work as member and leader in multidisciplinary teams and with continuous learning ability on technology and trends needed for a successful career.

PROGRAM OUTCOMES (POs)

After going through the two years of study, our master's in computer applications Graduates will exhibit ability to:

PO#	Programme Outcomes
1.	An ability to independently carry out research/investigation and development work to solve practical problems.
2.	An ability to write and present a substantial technical report/document.
3.	An ability to demonstrate a degree of mastery over design and development of computer applications.
4.	An ability to create, select, adapt and apply appropriate innovative techniques, resources, and modern computing tools to complex computing activities with an understanding of the limitations.
5.	An ability to recognize the need and to engage in independent learning for continual development as a computing professional.
6.	An ability to function effectively as an individual and as a member/leader of a team in various technical environments.

COURSE OUTCOMES:

Course Name: MC4165 DATA STRUCTURES LABORATORY

Year of study: 2024 –2025

MC4165.1	Implement basic and advanced data structures extensively.
MC4165.2	Choose and apply suitable hierarchical data structures for real time problems.
MC4165.3	Apply suitable heap data structures based on the problem requirements.
MC4165.4	Design and apply algorithms using graph structures.
MC4165.5	Design and implement iterative and recursive algorithms with minimum complexity.
MC4165.6	Design and develop efficient algorithms by adopting suitable algorithm design strategies

CO-PO Matrix:

1	2	1	3	3	2	2
2	2	1	3	3	2	2
3	1	1	3	2	2	2
4	2	1	3	2	2	2
5	2	1	3	3	2	3
Avg	1.8	1	3	2.6	2	2.2

Justification:

Course Outcome	Program Outcome	Value	Justification
MC4165.1	PO1	3	Students will be able to write, debug, and execute programs in a chosen programming language
	PO5	2	Students will develop a strong understanding of core data structures such as arrays, linked lists, stacks, queues, trees, graphs, and hash tables.
MC4165.2	PO1	2	Students will be able to implement and analyze fundamental algorithms for searching, sorting, and traversal.
	PO2	2	Students will enhance their problem-solving skills by applying appropriate data structures and algorithms to solve computational problems.
MC4165.3	PO1	2	Students will gain practical experience by working on projects that require the application of programming and data structure concepts.
	PO2	2	Students will develop collaboration skills by working in teams to complete lab assignments and projects.
MC4165.4	PO1	3	Students will learn effective debugging techniques and how to write test cases to ensure their code is robust and error-free.

MC4165.5	PO1	3	Students will be able to apply theoretical concepts learned in lectures to practical scenarios in the laboratory.
	PO2	3	Students will learn how to optimize code for performance, including memory management and efficient algorithm design.
	PO3	3	Develop the ability to analyze problems, design algorithms, and implement solutions using appropriate data structures and programming techniques.
	PO5	2	Apply data structures to solve real-world problems and understand their practical applications.

CO-PO Average:

CO	PO1	PO2	PO3	PO4	PO5	PO6
MC4165	1.8	1	3	2.6	2	2.2

ASSESSMENT METHOD

i) EVALUATION PROCEDURE FOR EACH EXPERIMENTS

S.No	Description	Mark
1.	Aim & Pre-Lab discussion	20
2.	Observation	20
3.	Conduction and Execution	30
4.	Output & Result	10
5.	Viva	20
	Total	**100**

ii) INTERNAL ASSESSMENT FOR LABORATORY

S.No	Description	Mark
1.	Conduction & Execution of Experiment	25
2.	Record	10
3.	Model Test	15
	Total	**50**

TABLE OF CONTENTS

***Topic beyond the syllabus**

Ex.No:1.a **IMPLEMENTATION OF SINGLY LINKED LIST**

AIM

To write a C program to implement singly linked list.

PRE LAB-DISCUSSION

Linked list is a linear data structure. It is a collection of data elements, called nodes pointing to the next node by means of a pointer. In linked list, each node consists of its own data and the address of the next node and forms a chain.

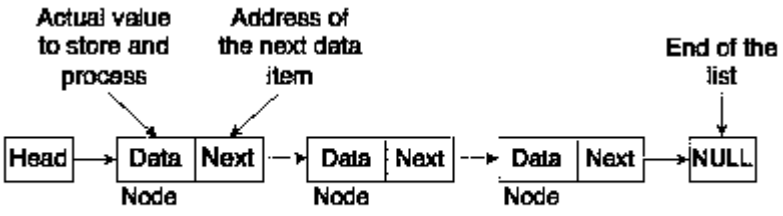
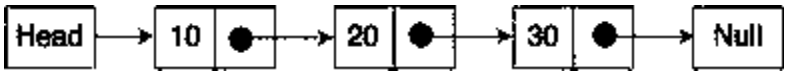


Fig. Linked List

Linked list contains a link element called first and each link carries a data item. Entry point intothe linked list is called the head of the list. Link field is called next and each link is linked with its next link. Last link carries a link to null to mark the end of thelist.

Linked list is a dynamic data structure. While accessing a particular item, start at the head and follow the references until you get that data item.

Linked list is used while dealing with an unknown number of objects:



Linked list contains two fields - First field contains value and second field contains a link to the next node. The last node signifies the end of the list that means NULL. The real-life example of Linked List is that of Railway Carriage. It starts from engine and then the coaches follow. Coaches can traverse from one coach to other, if they connected to each other.

ALGORITHM

1: Start

2: Creation: Get the number of elements, and create the nodes having structures DATA,LINK and store the element in Data field, link them together to form a linked list.

3: Insertion: Get the number to be inserted and create a new node store the value in DATA field and insert the node in the required position.

4: Deletion: Get the number to be deleted. Search the list from the beginning and locate the node then delete the node.

5: Display: Display all the nodes in the list.

6: Stop.

PROGRAM

```
// Linked list implementation of list ADT's
// Include necessary libraries and define constants
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>

typedef struct list
{
   int no; // number
   struct list *next; // pointer to next node
} LIST;

LIST *p, *t, *h, *y, *ptr, *pt;

void create(void);
void insert(void);
void delet(void);
void display(void);

int j, pos, k = 1, count;

void main()
{
   // Code to initialize the list and perform operations on it
   int n, i = 1, opt;
   p = NULL;

  printf("LINKED LIST IMPLEMENTATION OF LIST ADTs\n\n");
   printf("Enter the number of nodes: ");
   scanf("%d", &n);
   printf("\n");
   count = n;
   while (i <= n)
   {
      create(); // create nodes
      i++;
   }
   printf("\n");
   do
   {
      printf("**************************\n");
      printf("*       MAIN MENU       *\n");
      printf("**************************\n");
      printf("1. Insert\n");
      printf("2. Delete\n");
```

```
        printf("3. Display\n");

        printf("4. Exit\n");
        printf("Enter your option: ");
        scanf("%d", &opt);
        printf("\n");
        switch (opt)
        {
            case 1:
                insert(); // insert node
                count++;
                break;
            case 2:
                delet(); // delete node
                count--;
                if (count == 0)
                {
                    printf("\nList is empty\n");
                }
                break;
            case 3:
                printf("List elements are:\n");
                display(); // display nodes
                break;
            case 4:
                printf("Thank you for using the program!\n");
                exit(0);
                break;
        }
    } while (opt != 4);
    getch();
}
void create()
{
    // Code to create a new node and add it to the list
    if (p == NULL)
    {
        p = (LIST *)malloc(sizeof(LIST));
        printf("Enter the element: ");
        scanf("%d", &p->no);
        p->next = NULL;
        h = p;
    }
    else
    {
        t = (LIST *)malloc(sizeof(LIST));
```

```
        printf("Enter the element: ");
        scanf("%d", &t->no);
        t->next = NULL;
        p->next = t;
        p = t;
    }
}
void insert()
{
    // Code to insert an element into the list
    t = h;
    p = (LIST *)malloc(sizeof(LIST));
    printf("Enter the element to be inserted: ");
    scanf("%d", &p->no);
    printf("Enter the position to insert: ");
    scanf("%d", &pos);
    if (pos == 1)
    {
        h = p;
        h->next = t;
    }
    else
    {
        for (j = 1; j < (pos - 1); j++)
            t = t->next;
        p->next = t->next;
        t->next = p;
        t = p;
    }
    printf("\n");
}
void delet()
{
    // Code to delete an element from the list
    printf("Enter the position to delete: ");
    scanf("%d", &pos);
    if (pos == 1)
    {
        h = h->next;
    }
    else
    {
        t = h;
        for (j = 1; j < (pos - 1); j++)
            t = t->next;
        pt = t->next->next;
        free(t->next);
```

```
        t->next = pt;
    }
    printf("\n");
}

void display()
{
    // Code to display the elements of the list
    t = h;
    while (t->next != NULL)
    {
        printf("%d\t", t->no);
        t = t->next;
    }
    printf("%d", t->no);
    printf("\n\n");
}
```

OUTPUT

```
LINKED LIST IMPLEMENTATION OF LIST ADTs

Enter the number of nodes: 5

Enter the element: 12
Enter the element: 23
Enter the element: 34
Enter the element: 45
Enter the element: 56
*************************
*       MAIN MENU       *
*************************
1. Insert
2. Delete
3. Display
4. Exit
Enter your option: 1
Enter the element to be inserted: 27
Enter the position to insert: 3
*************************
*       MAIN MENU       *
*************************
1. Insert
2. Delete
3. Display
4. Exit
Enter your option: 2
Enter the position to delete: 4
```

```
*************************
*      MAIN MENU      *
*************************
1. Insert
2. Delete
3. Display
4. Exit
Enter your option: 3
List elements are:
12      23      27      45      56

*************************
*      MAIN MENU      *
*************************
1. Insert
2. Delete
3. Display
4. Exit
Enter your option: 4
Thank you for using the program!
```

EXPLANATION

Node Definition: The LIST structure defines a node containing an integer (no) and a pointer to the next node (next).

Pointer Variables: p, t, h, y, ptr, and pt are used as pointers to LIST nodes, indicating operations like insertion, deletion, and traversal within the linked list.

Functions:

create(): Creates a new node and adds it to the end of the linked list.

insert(): Inserts a new node at a specified position in the linked list.

delet(): Deletes a node at a specified position from the linked list.

display(): Displays all elements of the linked list.

Main Function:

Initializes the linked list with user-defined number of nodes. Provides a menu-driven interface (opt) to perform insert, delete, display, and exit operations on the linked list.

Dynamic Memory Allocation: Uses malloc() to allocate memory for new nodes dynamically, which is a characteristic feature of linked lists.

Traversal: Traverses through the linked list using pointers (t) to access and manipulate nodes.

VIVA QUESTIONS

1. How would you define a node in a singly linked list?
2. What is the role of the head pointer in a singly linked list?
3. Can you provide an example of a problem where using a singly linked list would be advantageous?
4. How would you handle edge cases such as inserting or deleting from an empty list?
5. Compare and contrast singly linked lists with doubly linked lists and circular linked lists.

RESULT

Thus, the C program to implement Singly Linked List was completed successfully.

Ex.No. 1.b LINKED LIST IMPLEMENTATION OF STACK ADT

AIM

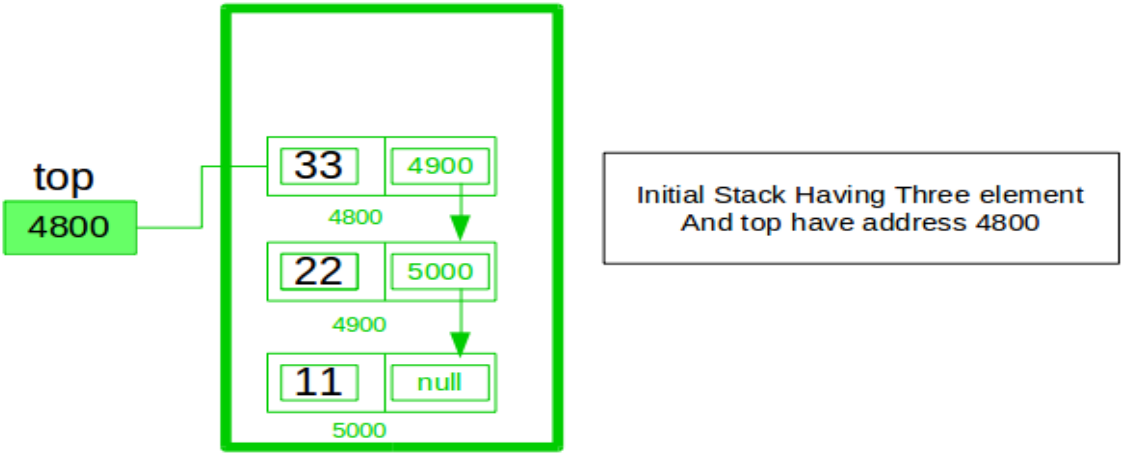
To write a C program to implement Stack operations such as push, pop and display using linked list.

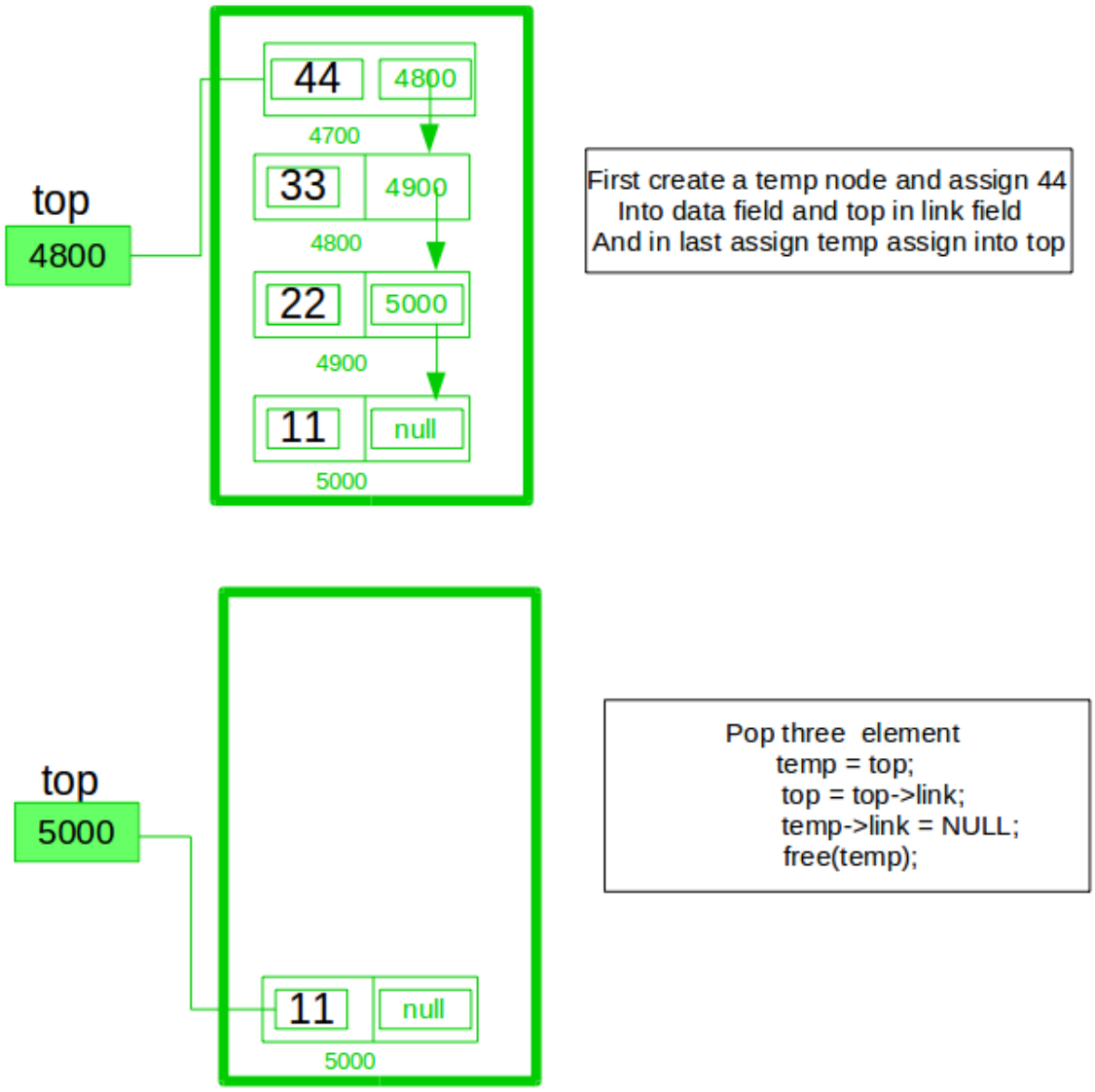
PRE LAB-DISCUSSION

The major problem with the stack implemented using array is, it works only for fixed number of data values. That means the amount of data must be specified at the beginning of the implementation itself. Stack implemented using array is not suitable, when we don't know the size of data which we are going to use. A stack data structure can be implemented by using linked list data structure. The stack implemented using linked list can work for unlimited number of values. That means, stack implemented using linked list works for variable size of data. So, there is no need to fix the size at the beginning of the implementation. The Stack implemented using linked list can organize as many data values as we want.

In linked list implementation of a stack, every new element is inserted as 'top' element. That means every newly inserted element is pointed by 'top'. Whenever we want to remove an element from the stack, simply remove the node which is pointed by 'top' by moving 'top' to its next node in the list. The next field of the first element must be always NULL.

Example:





There are two basic operations performed in a Stack:
1. Push():is used to add or insert new elements into thestack.
2. Pop():is used to delete or remove an element from thestack.

ALGORITHM

1: Start.

2: push operation inserts an element at the front.

3: pop operation deletes an element at the front of the list;

4: display operation displays all the elements in the list.

5: Stop.

PROGRAM

```
// Linked list implementation of stack
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
```

```
void pop();
void push(int value);
void display();

struct node
{
   int data;
   struct node *link;
};

struct node *top = NULL, *temp;
void main()
{
   int choice, data;
   printf("LINKED LIST IMPLEMENTATION OF STACK\n\n");

   while (1) // infinite loop is used to insert/delete infinite number of elements in stack
   {
      printf("**************************\n");
      printf("*       MAIN MENU       *\n");
      printf("**************************\n");
      printf("1. Push\n");
      printf("2. Pop\n");
      printf("3. Display\n");
      printf("4. Exit\n");
      printf("Enter your choice: ");
      scanf("%d", &choice);
      printf("\n");
      switch (choice)
      {
         case 1: // To push a new element into stack
         {
            printf("Enter a new element: ");
            scanf("%d", &data);
            push(data);
            break;
         }
         case 2: // pop the element from stack
         {
            pop();
            break;
         }
         case 3: // Display the stack elements
         {
            display();
            break;
         }
```

```
            case 4: // To exit
            {
               printf("Thank you for using the program. Exiting...\n");
               exit(0);
            }
         }
      }
      getch();
      // return 0;
}
void display()
{
   temp = top;
   if (temp == NULL)
   {
      printf("\nStack is empty\n");
   }
   printf("\nThe Contents of the Stack are...\n");
   while (temp != NULL)
   {
      printf(" %d->", temp->data);
      temp = temp->link;
   }
   printf("\n\n");
}

void push(int data)
{
   temp = (struct node *)malloc(sizeof(struct node)); // creating a space for the new element.
   temp->data = data;
   temp->link = top;
   top = temp;
   display();
}

void pop()
{
   if (top != NULL)
   {
      printf("The popped element is %d", top->data);
      printf("\n");
      top = top->link;
   }
   else
   {
      printf("\nStack Underflow");
   }
```

```
    display();
}
```

OUTPUT

```
LINKED LIST IMPLEMENTATION OF STACK

*************************
*       MAIN MENU       *
*************************
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 1

Enter a new element: 12

The Contents of the Stack are...
 12->

*************************
*       MAIN MENU       *
*************************
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 1

Enter a new element: 23

The Contents of the Stack are...
 23-> 12->

*************************
*       MAIN MENU       *
*************************
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 2

The popped element is 23

The Contents of the Stack are...
 12->
```

```
*************************
*      MAIN MENU      *
*************************
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 1

Enter a new element: 34
The Contents of the Stack are...
 34-> 12->

*************************
*      MAIN MENU      *
*************************

1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 3


The Contents of the Stack are...
 34-> 12->

*************************
*      MAIN MENU      *
*************************
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 4

Thank you for using the program. Exiting...
```

EXPLANATION

Initial Display: The program starts by displaying the main menu and prompts the user to enter a choice.
Push Operation (Choice 1): User chooses to push a new element (12) onto the stack. After pushing, the stack contents (12) are displayed.
Push Operation Again (Choice 1): User pushes another element (23) onto the stack. Updated stack contents (23-> 12->) are displayed.
Pop Operation (Choice 2): User chooses to pop an element from the stack. The element 23 is removed (popped), and the updated stack contents (12->) are displayed.

Push Operation Again (Choice 1): User pushes another element (34) onto the stack. Updated stack contents (34-> 12->) are displayed.
Display Operation (Choice 3): User chooses to display the stack contents. The current stack contents (34-> 12->) are displayed.
Exit Operation (Choice 4): User chooses to exit the program.

VIVA QUESTIONS

1. What challenges might arise when implementing stack operations using a linked list?
2. How would you handle stack overflow or underflow conditions in a linked list-based stack?
3. What are some practical applications of stack data structures in computer science?
4. What is the top of the stack, and why is it important?
5. How do you check if a stack implemented using a linked list is empty?
6. How do you retrieve the top element of a stack without removing it?

RESULT

Thus, the C program to implement Stack using linked list was completed successfully.

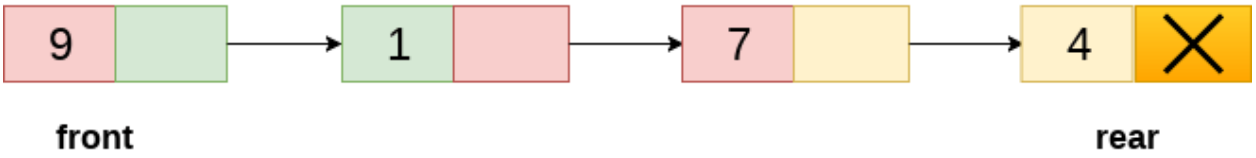
Ex.No.1.c LINKED LIST IMPLEMENTATION OF LINEAR QUEUE ADT

AIM

To write a C program to implement Queue operations such as enqueue, dequeue and display using linked list.

PRE LAB-DISCUSSION

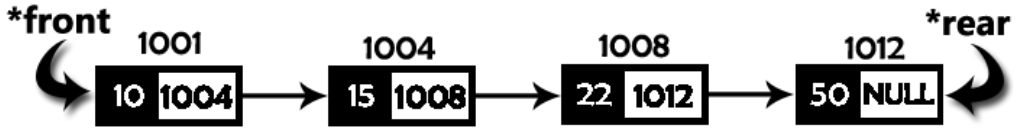
The major problem with the queue implemented using array is, It will work for only fixed number of data. That means the amount of data must be specified in the beginning itself. Queue using array is not suitable when we don't know the size of data which we are going to use. A queue data structure can be implemented using linked list data structure. The queue which is implemented using linked list can work for unlimited number of values. That means, queue using linked list can work for variable size of data (No need to fix the size at beginning of the implementation).



Linked Queue

The Queue implemented using linked list can organize as many data values as we want. In linked list implementation of a queue, the last inserted node is always pointed by '**rear**' and the first node is always pointed by '**front**'.

Example



In above example, the last inserted node is 50 and it is pointed by '**rear**' and the first inserted node is 10 and it is pointed by '**front**'. The order of elements inserted is 10, 15, 22 and 50.

There are two basic operations performed on a Queue.

Enqueue(): This function defines the operation for adding an element into queue.

Dequeue(): This function defines the operation for removing an element from queue.

Key Characteristics of a Linear Queue ADT:

FIFO Order: Elements are inserted at the rear (end) and removed from the front (beginning).

Single Directional: The queue grows in one direction, from front to rear.

Operations: Basic operations include insertion (enqueue), deletion (dequeue), and display.

ALGORITHM

1: Start.

2: Enqueue operation inserts an element at the rear of the list.

3: Dequeue operation deletes an element at the front of the list.

4: Display operation display all the element in the list.

5: Stop.

PROGRAM

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

struct Node
{
   int data;
   struct Node *next;
};

struct Node *front = NULL, *rear = NULL;

void insert(int);
void delet();
void display();

void main()
{
   int choice, value;
   printf("LINKED LIST IMPLEMENTATION OF QUEUES\n\n");

   while (1) {
          printf("**************************\n");
          printf("***     MAIN MENU     ***\n");
          printf("**************************\n");
          printf("1. Insert\n");
          printf("2. Delete\n");
          printf("3. Display\n");
          printf("4. Exit\n");
          printf("Enter your choice: ");
          scanf("%d", &choice);
          printf("\n");

          switch (choice) {
             case 1:
                printf("Enter the value to be inserted: ");
                scanf("%d", &value);
                insert(value);
                break;
```

```
                case 2:
                    delet();
                    break;
                case 3:
                    display();
                    break;
                case 4:
                    printf("\nThank you for using the program. Exiting...\n");
                    exit(0);
                default:
                    printf("\nInvalid option. Try again.\n");
            }
    }
}

void insert(int value)
{
    struct Node *newNode;
    newNode = (struct Node *)malloc(sizeof(struct Node));
    newNode -> data = value;
    newNode -> next = NULL;
    if (front == NULL)
            front = rear = newNode;
    else {
            rear -> next = newNode;
            rear = newNode;
    }
    printf("\nINSERTION SUCCESSFULL\n\n");
}

void delet()
{
    if (front == NULL)
        printf("EMPTY QUEUE\n\n");
    else {
        struct Node *temp = front;
        front = front->next;
        printf("Deleted element: %d\n\n", temp -> data);
        free(temp);
    }
}
void display()
{
    if (front == NULL)
        printf("EMPTY QUEUE\n\n");
    else {
```

```
        struct Node *temp = front;

 while (temp -> next != NULL) {
            printf("%d -> ", temp -> data);
            temp = temp->next;
        }
        printf("%d -> NULL\n\n", temp -> data);
    }
}
```

OUTPUT

LINKED LIST IMPLEMENTATION OF QUEUES

```
*************************
***     MAIN MENU     ***
*************************
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 1

Enter the value to be inserted: 12

INSERTION SUCCESSFULL

*************************
***     MAIN MENU     ***
*************************
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 1

Enter the value to be inserted: 23

INSERTION SUCCESSFULL

*************************
***     MAIN MENU     ***
*************************
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 2
```

Deleted element: 12

****
*** MAIN MENU ***
****
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 1

Enter the value to be inserted: 34

INSERTION SUCCESSFULL

****
*** MAIN MENU ***
****
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 3

23 -> 34 -> NULL

****
*** MAIN MENU ***
****
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 4

Thank you for using the program. Exiting...

EXPLANATION

Node Structure: Each node contains data and a pointer to the next node.

```
struct Node {
   int data;
   struct Node *next;
};
```

Global Pointers: front and rear pointers are used to track the front and rear of the queue.

```
struct Node *front = NULL, *rear = NULL;
```

Insertion (Enqueue):
New elements are added at the rear of the queue. If the queue is empty, both front and rear are set to the new node. Otherwise, the new node is added after the rear, and rear is updated.

Deletion (Dequeue):
Elements are removed from the front of the queue. If the queue is empty, a message is displayed. Otherwise, the front element is removed, and front is updated to the next node.

Display:
The queue elements are displayed from the front to the rear. If the queue is empty, a message is displayed. The primary operations (insert, delete, display) maintain the FIFO order, and the use of linked nodes allows dynamic memory management, accommodating the dynamic nature of queues.

VIVA QUESTIONS

1. Compare and contrast a queue implemented using a linked list versus an array.
2. How does a linked list-based queue handle memory allocation compare to an array-based queue?
3. What are the limitations or drawbacks of using a linked list for implementing a queue?
4. How would you handle queue underflow or overflow conditions in a linked list-based queue?
5. What modifications would you make to the linked list-based queue implementation to support priority queues?
6. How do you check if a queue implemented using a linked list is empty?
7. How do you define a node in the linked list for queue implementation?

RESULT

Thus, the C program to implement linear queue using linked list was completed successfully

Ex.No:2 IMPLEMENTATION OF BINARY SEARCH TREES

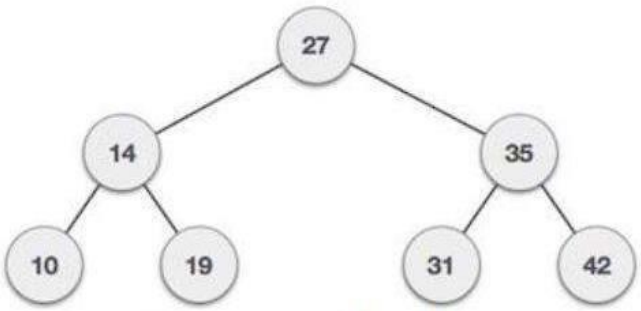
AIM

To write a C program to implement the binary search trees.

PRE LAB-DISCUSSION

A binary search tree (BST) is a tree in which all nodes follows the below mentioned properties

- The left sub-tree of a node has key less than or equal to its parent node'skey.
- The right sub-tree of a node has key greater than or equal to its parent node's key.
- Thus, a binary search tree (BST) divides all its sub-trees into two segments;
- left sub- tree and right sub-tree and can be definedas
  left_subtree (keys) ≤ node (key) ≤ right_subtree (keys)



Following are basic primary operations of a tree which are following.

- Search − search an element in atree.
- Insert − insert an element in atree.
- Delete – removes an existing node from thetree
- Preorder Traversal − traverse a tree in a preordermanner.
- Inorder Traversal − traverse a tree in an inordermanner.
- Postorder Traversal − traverse a tree in a postordermanner.

ALGORITHM

1: Start the process.

2: Initialize and declare variables.

3: Construct the Tree

4: Data values are given which we call a key and a binary search tree

5: To search for the key in the given binary search tree, start with the root node and compare the key with the data value of the root node. If they match, return the root pointer.

6: If the key is less than the data value of the root node, repeat the process by using the left subtree.

7: Otherwise, repeat the same process with the right subtree until either a match is found or the subtree under consideration becomes an empty tree.

8: Terminate

PROGRAM

```
// Implementation of binary search trees

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

struct tree
{
   int data;
   struct tree *lchild;
   struct tree *rchild;
} *t, *temp;

int element;

void inorder(struct tree *);
void preorder(struct tree *);
void postorder(struct tree *);
struct tree *create(struct tree *, int);
struct tree *find(struct tree *, int);
struct tree *insert(struct tree *, int);
struct tree *del(struct tree *, int);
struct tree *findmin(struct tree *);
struct tree *findmax(struct tree *);
int main(void)
{
   int ch;
   printf("BINARY SEARCH TREE\n\n");
   do
   {
      printf("**************************\n");
      printf("***      MAIN MENU      ***\n");
      printf("**************************\n");
      printf("1. Create\n");
      printf("2. Insert\n");
      printf("3. Delete\n");
      printf("4. Find\n");
      printf("5. FindMin\n");
      printf("6. FindMax\n");
      printf("7. Inorder\n");
      printf("8. Preorder\n");
      printf("9. Postorder\n");
      printf("10. Exit\n");
      printf("Enter your choice: ");
```

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

printf("\n");
switch (ch)
{
   case 1:
      printf("Enter the data: ");
      scanf("%d", &element);
      printf("\n");
      t = create(t, element);
      inorder(t);
      printf("\n\n");
      break;
   case 2:
      printf("Enter the data: ");
      scanf("%d", &element);
      printf("\n");
      t = insert(t, element);
      inorder(t);
      printf("\n\n");
      break;
   case 3:
      printf("Enter the data: ");
      scanf("%d", &element);
      printf("\n");
      t = del(t, element);
      inorder(t);
      printf("\n\n");
      break;
   case 4:
      printf("Enter the data: ");
      scanf("%d", &element);
      temp = find(t, element);
      if (temp->data == element)
         printf("\nElement %d is at %d", element, temp);
      else
         printf("\nElement is not found");
      printf("\n\n");
      break;
   case 5:
      temp = findmin(t);
      printf("Min element = %d", temp->data);
      printf("\n\n");
      break;
   case 6:
      temp = findmax(t);
      printf("Max element = %d", temp->data);
```

```
            printf("\n\n");
            break;

        case 7:
            inorder(t);
            printf("\n\n");
            break;
        case 8:
            preorder(t);
            printf("\n\n");
            break;
        case 9:
            postorder(t);
            printf("\n\n");
            break;
        case 10:
            printf("Thank you for using the binary search tree program!\n");
            exit(0);
      }
   } while (ch <= 10);
}

struct tree *create(struct tree *t, int element)
{
   t = (struct tree *)malloc(sizeof(struct tree));
   t->data = element;
   t->lchild = NULL;
   t->rchild = NULL;
   return t;
}
struct tree *find(struct tree *t, int element)
{
   if (t == NULL)
      return NULL;
   if (element < t->data)
      return (find(t->lchild, element));
   else if (element > t->data)
      return (find(t->rchild, element));
   else
      return t;
}
struct tree *findmin(struct tree *t)
{
   if (t == NULL)
      return NULL;
   else if (t->lchild == NULL)
      return t;
```

```
    else
        return (findmin(t->lchild));
}

struct tree *findmax(struct tree *t)
{
    if (t != NULL)
    {
        while (t->rchild != NULL)
            t = t->rchild;
    }
    return t;
}

struct tree *insert(struct tree *t, int element)
{
    if (t == NULL)
    {
        t = (struct tree *)malloc(sizeof(struct tree));
        t->data = element;
        t->lchild = NULL;
        t->rchild = NULL;
        return t;
    }
    else if (element < t->data)
    {
        t->lchild = insert(t->lchild, element);
    }
    else if (element > t->data)
    {
        t->rchild = insert(t->rchild, element);
    }
    else if (element == t->data)
    {
        printf("Element already present\n");
    }
    return t;
}

struct tree *del(struct tree *t, int element)
{
    if (t == NULL)
        printf("Element not found\n");
    else if (element < t->data)
        t->lchild = del(t->lchild, element);
    else if (element > t->data)
        t->rchild = del(t->rchild, element);
```

```
else if (t->lchild && t->rchild)
   {
      temp = findmin(t->rchild);
      t->data = temp->data;
      t->rchild = del(t->rchild, t->data);
      temp = t;
      if (t->lchild == NULL)
         t = t->rchild;
      return t;
   }
   else if (t->rchild == NULL)
      t = t->lchild;
   free(temp);
   return t;
}
void inorder(struct tree *t)
{
   if (t == NULL)
      return;
   else
   {
      inorder(t->lchild);
      printf("\t%d", t->data);
      inorder(t->rchild);
   }
}
void preorder(struct tree *t)
{
   if (t == NULL)
      return;
   else
   {
      printf("\t%d", t->data);
      preorder(t->lchild);
      preorder(t->rchild);
   }
}
void postorder(struct tree *t)
{
   if (t == NULL)
      return;
   else
```

```
    {
        postorder(t->lchild);
        postorder(t->rchild);
        printf("\t%d", t->data);
    }

}
```

OUTPUT

```
BINARY SEARCH TREE

*************************
***      MAIN MENU      ***
*************************
1. Create
2. Insert
3. Delete
4. Find
5. FindMin
6. FindMax
7. Inorder
8. Preorder
9. Postorder
10. Exit
Enter your choice: 1

Enter the data: 12

       12

*************************
***      MAIN MENU      ***
*************************
1. Create
2. Insert
3. Delete
4. Find
5. FindMin
6. FindMax
7. Inorder
8. Preorder
9. Postorder
10. Exit
Enter your choice: 2

Enter the data: 11
```

```
      11      12
```

```
**************************
***      MAIN MENU      ***
**************************
1. Create
2. Insert
3. Delete
4. Find
5. FindMin
6. FindMax
7. Inorder
8. Preorder
9. Postorder
10. Exit
Enter your choice: 2

Enter the data: 23

      11      12      23

**************************
***      MAIN MENU      ***
**************************
1. Create
2. Insert
3. Delete
4. Find
5. FindMin
6. FindMax
7. Inorder
8. Preorder
9. Postorder
10. Exit
Enter your choice: 3

Enter the data: 23

      11      12

**************************
***      MAIN MENU      ***
```

```
**************************
1. Create
2. Insert
3. Delete
4. Find

5. FindMin
6. FindMax
7. Inorder
8. Preorder
9. Postorder
10. Exit
Enter your choice: 2

Enter the data: 24

      11      12      24

**************************
***      MAIN MENU      ***
**************************
1. Create
2. Insert
3. Delete
4. Find
5. FindMin
6. FindMax
7. Inorder
8. Preorder
9. Postorder
10. Exit
Enter your choice: 4

Enter the data: 11

Element 11 is at 1395266672

**************************
***      MAIN MENU      ***
**************************
1. Create
2. Insert
3. Delete
4. Find
5. FindMin
6. FindMax
7. Inorder
```

```
8. Preorder
9. Postorder
10. Exit
Enter your choice: 5

Min element = 11

*************************
***      MAIN MENU      ***
*************************
1. Create
2. Insert
3. Delete
4. Find
5. FindMin
6. FindMax
7. Inorder
8. Preorder
9. Postorder
10. Exit
Enter your choice: 6

Max element = 24

*************************
***      MAIN MENU      ***
*************************
1. Create
2. Insert
3. Delete
4. Find
5. FindMin
6. FindMax
7. Inorder
8. Preorder
9. Postorder
10. Exit
Enter your choice: 7

      11      12      24

*************************
***      MAIN MENU      ***
*************************
1. Create
2. Insert
3. Delete
```

4. Find
5. FindMin
6. FindMax
7. Inorder
8. Preorder
9. Postorder
10. Exit
Enter your choice: 8

12 11 24

****
*** MAIN MENU ***
****
1. Create
2. Insert
3. Delete
4. Find
5. FindMin
6. FindMax
7. Inorder
8. Preorder
9. Postorder
10. Exit
Enter your choice: 9

11 24 12

****
*** MAIN MENU ***
****
1. Create
2. Insert
3. Delete
4. Find
5. FindMin
6. FindMax
7. Inorder
8. Preorder
9. Postorder
10. Exit
Enter your choice: 10

Thank you for using the binary search tree program!

EXPLANATION
Operations Breakdown

1. **Create**: Initializes a BST with the given element.
2. **Insert**: Inserts an element into the BST.
3. **Delete**: Deletes an element from the BST.
4. **Find**: Searches for an element in the BST.
5. **FindMin**: Finds the minimum element in the BST.
6. **FindMax**: Finds the maximum element in the BST.
7. **Inorder**: Displays the BST elements in inorder traversal.
8. **Preorder**: Displays the BST elements in preorder traversal.
9. **Postorder**: Displays the BST elements in postorder traversal.
10. **Exit**: Exits the program.

VIVA QUESTIONS

1. Why is it said that in-order traversal of a BST produces a sorted sequence?
2. Can a BST have duplicate values? Why or why not?
3. Explain how the BST property ensures efficient search operations.
4. What is the worst-case time complexity for BST operations, and under what conditions does it occur?
5. What are some common applications of BSTs?

RESULT

Thus, the C program to implement the binary search trees was completed successfully.

Ex. No.3 **IMPLEMENTATION OF AVL TREES**

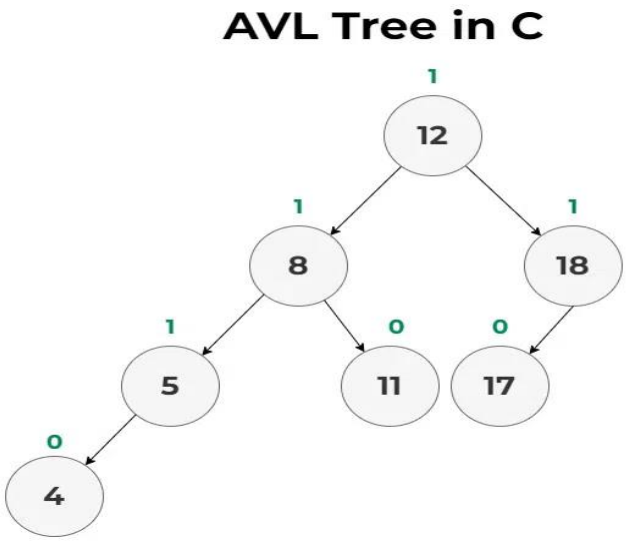
AIM:

To implement AVL Trees.

PRE LAB-DISCUSSION:

AVL Tree in C

An AVL tree is a self-balancing binary search tree that was created by Adelson-Velsky and Landis, hence the name AVL. It is a height balanced tree that keeps the difference between the height of the left and right subtrees in the range [-1, 0, 1]. This difference is called balanced factor and tree is said to be unbalanced when this balance factor is outside the specified range. Unbalanced tree is balanced through specific rotation operations during insertions and deletions.



Representation of AVL Tree in C

In C, an AVL tree node is typically defined using a struct. Each node contains the data, pointers to its left and right children, and an integer to store its height.

```
struct AVLNode {
   int key;
   struct AVLNode* left;
   struct AVLNode* right;
   int height;
};
```

AVL Tree Rotations in C

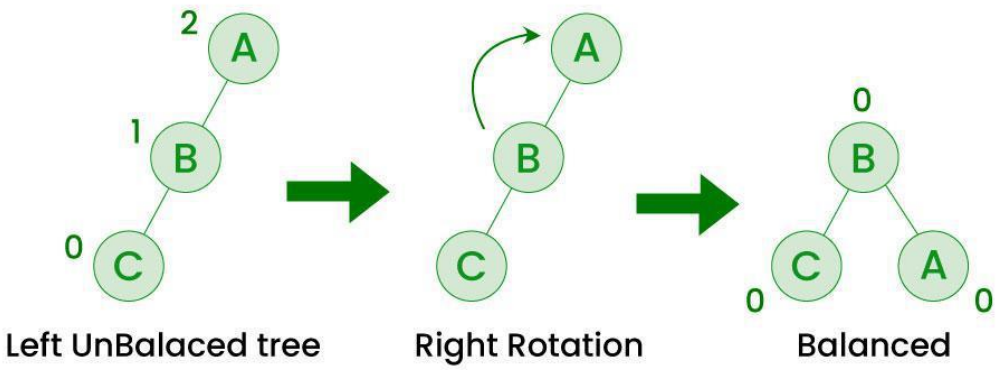
Rotations are the most important part of the working of the AVL tree. They are responsible for maintaining the balance in the AVL tree. There are 4 types of rotations based on the 4 possible cases:

1. Right Rotation (RR)

2. Left Rotation (LL)
3. Left-Right Rotation (LR)
4. Right-Left Rotation (RL)

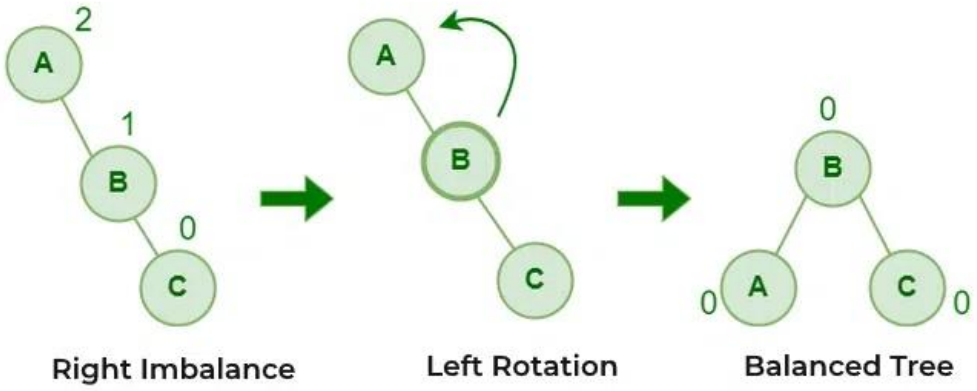
Right Rotation (RR)

The RR rotation is applied when a node becomes unbalanced due to an insertion into the right subtree of its right child. This type of imbalance is called **Left Imbalance.** The solution to it is to take the unbalanced node, and rotate the top edge (that is connected with parent) 90° to the right (clockwise).



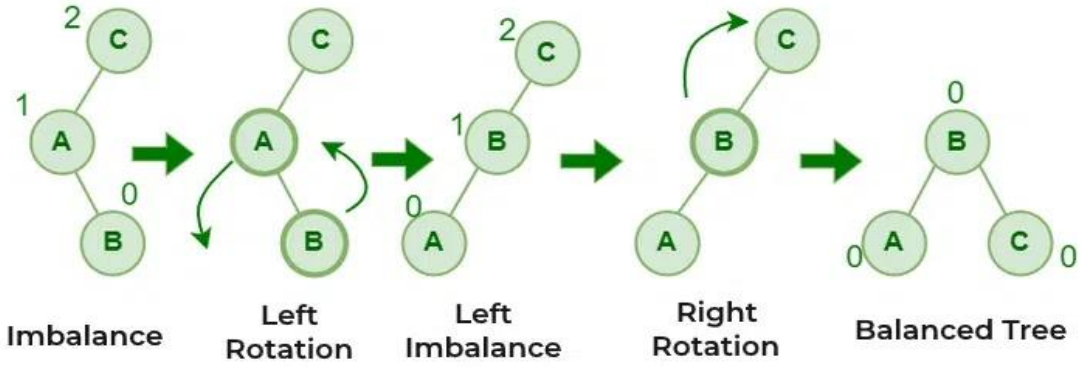
Left Rotation (LL)

The LL rotation is used in an AVL tree to balance a node that becomes unbalanced due to an insertion into the left subtree of its left child. It is called **Left Imbalance.** The solution to it is to take the unbalanced node, and rotate the top edge (that is connected with parent) 90° to the left(anti-clockwise).



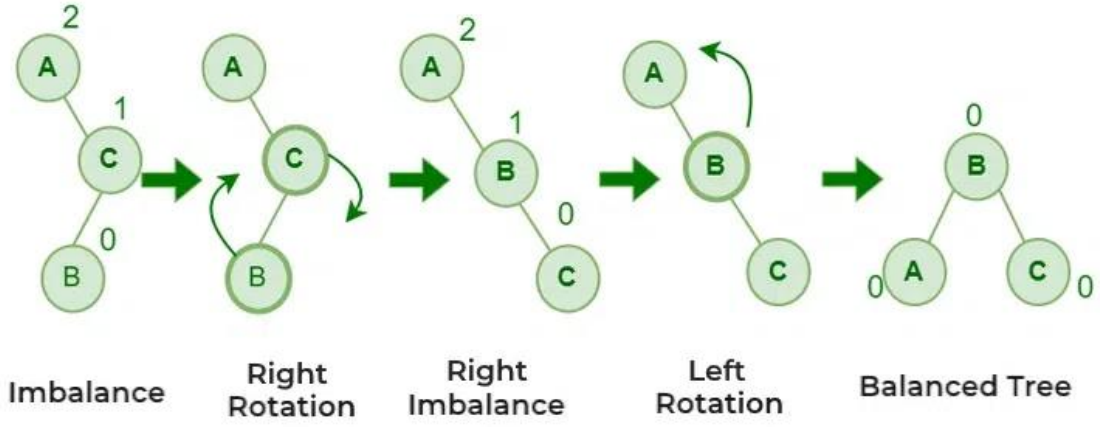
Left-Right Rotation (LR)

A right-left rotation is used when a left child of a node is right-heavy. It helps in balancing the tree after a double imbalance, which is another specific insertion case. We first perform a left rotation on the left child and follow it up with a right rotation on the original node.



Right-Left Rotation (RL)

The right left rotation is performed when a right child of a node is left-heavy. We perform a right rotation on the right child and follow it up with a left rotation on the original node.



Implementation of AVL Tree in C

We will implement the AVL tree along with its basic operations. Following are the basic operations of AVL tree:

Operation	Description	Time Complexity	Space Complexity
Search	Finds a specific value in the tree.	O(log n)	O(log n)
Insert	Adds a new value to the tree while maintaining balance.	O(log n)	O(1)

Operation	Description	Time Complexity	Space Complexity
Delete	Removes a specific value from the tree and maintains balance.	O(log n)	O(1)

Search Operation Implementation

We implement the search operation on AVL tree using the following algorithm:

1. If root is null or key is present at root, return root.
2. If key is smaller than root's key, search in left subtree.
3. If key is larger than root's key, search in right subtree.

Insert Operation Implementation

Insert operation may create an imbalance in the tree which should be handled by the program according to the case.

- If the current node is NULL, create a new node with the given key.
- If the key is less than the current node's key, recursively insert the key in the left subtree.
- If the key is greater than the current node's key, recursively insert the key in the right subtree.
- After insertion, update the height of the current node.
- Calculate the balance factor of the current node.
- If the balance factor is greater than 1 (left-heavy), check for Left-Left (LL) or Left-Right (LR) case:
    - **LL Case**: If the left child's key is greater than the inserted key, perform a right rotation.
    - **LR Case**: If the left child's key is less than the inserted key, perform a left rotation on the left child, then a right rotation on the current node.
- If the balance factor is less than -1 (right-heavy), check for Right-Right (RR) or Right-Left (RL) case:
    - **RR Case**: If the right child's key is less than the inserted key, perform a left rotation.
    - **RL Case**: If the right child's key is greater than the inserted key, perform a right rotation on the right child, then a left rotation on the current node.

Delete Operation Implementation

- If the node to be deleted is not found, return NULL.
- If the key to be deleted is less than the current node's key, recursively delete the key in the left subtree.
- If the key to be deleted is greater than the current node's key, recursively delete the key in the right subtree.
- If the key to be deleted is equal to the current node's key:
    - If the node has only one child or no child, replace the node with its child.
    - If the node has two children, find the in order successor (smallest in the right subtree), copy its key to the node, and delete the inorder successor.
- After deletion, update the height of the current node.
- Calculate the balance factor of the current node.
- Check and fix imbalance just like in insertion.
- Return the node after performing necessary rotations to balance the tree.

ALGORITHM

A binary search tree x is called an AVL tree,
if:
1. b(x.key) ∈ {−1, 0, 1}, and

2. x.leftChild and x.rightChild are both AVL trees. = the height balance of every node must be -1, 0, or 1 insert/delete via standard algorithms
• after insert/delete: load balance b(node) might be changed to +2 or −2 for certain nodes
• re-balance load after each step
Insert operation may cause balance factor to become 2 or –2 for some node
› only nodes on the path from insertion point to root node have possibly changed in height
› So after the Insert, go back up to the root node by node, updating heights
› If a new balance factor (the difference hleft – hright) is 2 or –2, adjust tree by rotation around the node
Let the node that needs rebalancing be α.
There are 4 cases:
Outside Cases (require single rotation) :
1. Insertion into left subtree of left child of α.
2. Insertion into right subtree of right child of α.
Inside Cases (require double rotation) :
3. Insertion into right subtree of left child of α.
4. Insertion into left subtree of right child of α.
The rebalancing is performed through four separate rotation algorithms.
You can either keep the height or just the difference in height, i.e. the balance factor; this has to be modified on the path of insertion even if you don‟t perform rotations
Once you have performed a rotation (single or double) you won‟t need to go back up the tree

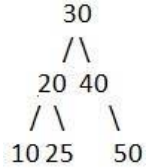
```
#include <stdio.h>
#include <stdlib.h>

// An AVL tree node
struct Node {
    int key;
    struct Node *left, *right;
    int height;
};

// A utility function to get maximum of two integers
int max(int a, int b) {
    return (a > b)? a : b;
}

// A utility function to get the height of the tree
int height(struct Node *N) {
    if (N == NULL)
        return 0;
    return N->height;
}

// Helper function that allocates a new node with the given key and NULL left and right pointers.
struct Node* newNode(int key) {
    struct Node* node = (struct Node*) malloc(sizeof(struct Node));
    node->key = key;
    node->left = NULL;
    node->right = NULL;
    node->height = 1; // new node is initially added at leaf
    return(node);
}

// A utility function to right rotate subtree rooted with y
struct Node *rightRotate(struct Node *y) {
    struct Node *x = y->left;
    struct Node *T2 = x->right;

    // Perform rotation
    x->right = y;
    y->left = T2;

    // Update heights
    y->height = max(height(y->left), height(y->right)) + 1;
    x->height = max(height(x->left), height(x->right)) + 1;

    // Return new root
    return x;
}

// A utility function to left rotate subtree rooted with x
struct Node *leftRotate(struct Node *x) {
```

```
    struct Node *y = x->right;
    struct Node *T2 = y->left;


    // Perform rotation
    y->left = x;
    x->right = T2;

    // Update heights
    x->height = max(height(x->left), height(x->right)) + 1;
    y->height = max(height(y->left), height(y->right)) + 1;

    // Return new root
    return y;
}

// Get Balance factor of node N
int getBalance(struct Node *N) {
    if (N == NULL)
        return 0;
    return height(N->left) - height(N->right);
}

// Recursive function to insert a key in the subtree rooted with node and returns the new root of the
subtree.
struct Node* insert(struct Node* node, int key) {
    // 1. Perform the normal BST insertion
    if (node == NULL)
        return(newNode(key));

    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);
    else // Equal keys are not allowed in AVL tree
        return node;

    // 2. Update height of this ancestor node
    node->height = 1 + max(height(node->left), height(node->right));

    // 3. Get the balance factor of this ancestor node to check whether this node became unbalanced
    int balance = getBalance(node);

    // If this node becomes unbalanced, then there are 4 cases

    // Left Left Case
    if (balance > 1 && key < node->left->key)
        return rightRotate(node);

    // Right Right Case
    if (balance < -1 && key > node->right->key)
        return leftRotate(node);

    // Left Right Case
```

```
    if (balance > 1 && key > node->left->key) {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }

    // Right Left Case
    if (balance < -1 && key < node->right->key) {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }

    // Return the (unchanged) node pointer
    return node;
}

// A utility function to print preorder traversal of the tree. The function also prints height of every node
void preOrder(struct Node *root) {
    if (root != NULL) {
        printf("%d ", root->key);
        preOrder(root->left);
        preOrder(root->right);
    }
}

// Driver program to test above functions
int main() {
    struct Node *root = NULL;

    /* Constructing tree given in the above figure */
    root = insert(root, 10);
    root = insert(root, 20);
    root = insert(root, 30);
    root = insert(root, 40);
    root = insert(root, 50);
    root = insert(root, 25);

    printf("Preorder traversal of the constructed AVL tree is \n");
    preOrder(root);

    return 0;
}
```

OUTPUT

The constructed AVL Tree would be



Explanation:

- The AVL tree is constructed by inserting nodes with the values 10, 20, 30, 40, 50, and 25.

- The preOrder function prints the tree nodes in pre-order traversal (root, left, right).

Viva Questions:

1. Why is balancing necessary in AVL Trees?
2. How do you calculate the height of a node in an AVL Tree?
3. What is the balance factor in an AVL Tree?
4. Explain the four cases of imbalance in AVL Trees and their corresponding rotations.
5. How does the preOrder function work in the AVL Tree implementation?
6. What is the significance of updating the height of a node in the AVL Tree?
7. What is the purpose of the getBalance function in the AVL Tree implementation?

Result: Thus, the implementation of AVL tree was completed successfully.

Ex.No .4 IMPLEMENTATION OF HASHING TECHNIQUES SUCH AS SEPARATE CHAINING

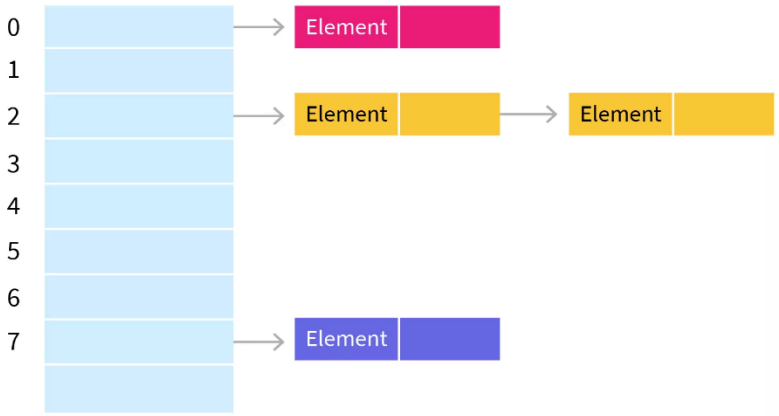
AIM

To write a C program to implement the concept of hashing using separate chaining.

PRE LAB-DISCUSSION

In hashing there is a hash function that maps keys to some values. But these hashing function may lead to collision that is two or more keys are mapped to same value. Chain hashing avoids collision. The idea is to make each cell of hash table point to a linked list of records that have same hash function value.

Let's create a hash function, such that our hash table has 'N' number of buckets. To insert a node into the hash table, we need to find the hash index for the given key. And it could be calculated using the hash function**. h(key) = key % table size**



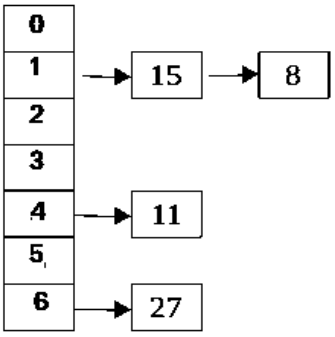
Example: hashIndex = key % no of Buckets

Insert: Move to the bucket corresponds to the above calculated hash index and insert the new node at the end of the list.

Here hash index = keys (15,11,27,8) % 7. Each key gets modulo with 7 to get the respected index value. Ex: 15%7=1,11%7=4,27%7=6 & 8%7= 1. % means, it takes the reminder value.

Let's say hash table with 7 buckets (0, 1, 2, 3, 4, 5, 6)

Keys arrive in the Order (15, 11 , 27 , 8)



ALGORITHM

1: Start

2: Create Table size

3: Create hash function

4: To insert a node into the hash table, we need to find the hash index for the given key. And it could be calculated using the hash function.

5: Display hash entry.

6: Stop

PROGRAM

```
#include <stdio.h>
#include <stdlib.h>
#define TABLE_SIZE 3
struct node
{
   int data;
   struct node *next;
};
struct node *head[TABLE_SIZE] = {NULL}, *c, *p;
void insert(int i, int val)
{
   struct node *newnode = (struct node *)malloc(sizeof(struct node));
   newnode->data = val;
   newnode->next = NULL;
   if (head[i] == NULL)
      head[i] = newnode;
   else
   {
      c = head[i];
      while (c->next != NULL)
         c = c->next;
      c->next = newnode;
   }
}
void display(int i)
{
   if (head[i] == NULL)
   {
      if (i == 0)
         printf("No Hash Entry");
      return;
   }
   else
   {
      printf("%d ->", head[i]->data);
      for (c = head[i]->next; c != NULL; c = c->next)
         printf("%d ->", c->data);
   }
}
void main()
{
   int opt, val, i;
```

```
    printf("HASHING WITH SEPARATE CHAINING\n\n");
    while (1)
    {
        printf("**********************\n");
        printf("*     MAIN MENU     *\n");
        printf("**********************\n");
        printf("1. Insert\n");
        printf("2. Display\n");
        printf("3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &opt);
        switch (opt)
        {
            case 1:
                printf("\nEnter a value to insert into the hash table:\n");
                scanf("%d", &val);
                i = val % TABLE_SIZE;
                insert(i, val);
                printf("\n");
                break;
            case 2:
                for (i = 0; i < TABLE_SIZE; i++)
                {
                    printf("\nHash entries at index %d\n", i);
                    display(i);
                }
                printf("\n\n");
                break;
            case 3:
                printf("\nThank you for using the program. Exiting...\n");
                exit(0);
        }
    }
}
```

OUTPUT

```
HASHING WITH SEPARATE CHAINING
**********************
*     MAIN MENU     *
**********************
1. Insert
2. Display
3. Exit
Enter your choice: 1

Enter a value to insert into the hash table:
5
```

****
* MAIN MENU *
****
1. Insert
2. Display
3. Exit
Enter your choice: 1
Enter a value to insert into the hash table:
9
****
* MAIN MENU *
****
1. Insert
2. Display
3. Exit
Enter your choice: 1

Enter a value to insert into the hash table:
4
****
* MAIN MENU *
****
1. Insert
2. Display
3. Exit
Enter your choice: 1

Enter a value to insert into the hash table:
6

****
* MAIN MENU *
****
1. Insert
2. Display
3. Exit
Enter your choice: 1

Enter a value to insert into the hash table:
7

****
* MAIN MENU *
****

1. Insert
2. Display
3. Exit

Enter your choice: 2

Hash entries at index 0
9 ->6 ->

Hash entries at index 1
4 ->7 ->
Hash entries at index 2
5 ->

****
* MAIN MENU *
****
1. Insert
2. Display
3. Exit
Enter your choice: 3

Thank you for using the program. Exiting...

EXPLANATION

Insert a value into the hash table:

- User chooses to insert a value.
- Enter a value, for example, 5.
- The index for insertion is calculated as 5 % 3 = 2.
- 5 is inserted at index 2.

Insert another value into the hash table:

- Enter a value, for example, 9.
- The index for insertion is calculated as 9 % 3 = 0.
- 9 is inserted at index 0.

Insert another value into the hash table:

- Enter a value, for example, 4.
- The index for insertion is calculated as 4 % 3 = 1.
- 4 is inserted at index 1.

Insert another value into the hash table:

- Enter a value, for example, 6.
- The index for insertion is calculated as 6 % 3 = 0.
- 6 is inserted at index 0, and since 9 is already present at index 0, 6 is appended after 9.

Insert another value into the hash table:

- Enter a value, for example, 7.
- The index for insertion is calculated as 7 % 3 = 1.
- 7 is inserted at index 1, and since 4 is already present at index 1, 7 is appended after 4.

Display the hash table:

- The program prints the contents of each index in the hash table.

Index 0: Contains values 9 and 6.

Index 1: Contains values 4 and 7.

Index 2: Contains value 5.

VIVA QUESTIONS

1. If several elements are competing for the same bucket in the hash table, what is itcalled?
2. How to insert a node in hashtable?
3. What is sizeof()function?
4. How to delete a node from hashtable.
5. What is Index value?

RESULT

Thus, the C program to implement hashing using separate chaining was completed successfully.

Ex. No. 5a **IMPLEMENTATION OF REPRESENTATION OF GRAPH**

AIM

To represent graph using adjacency list.

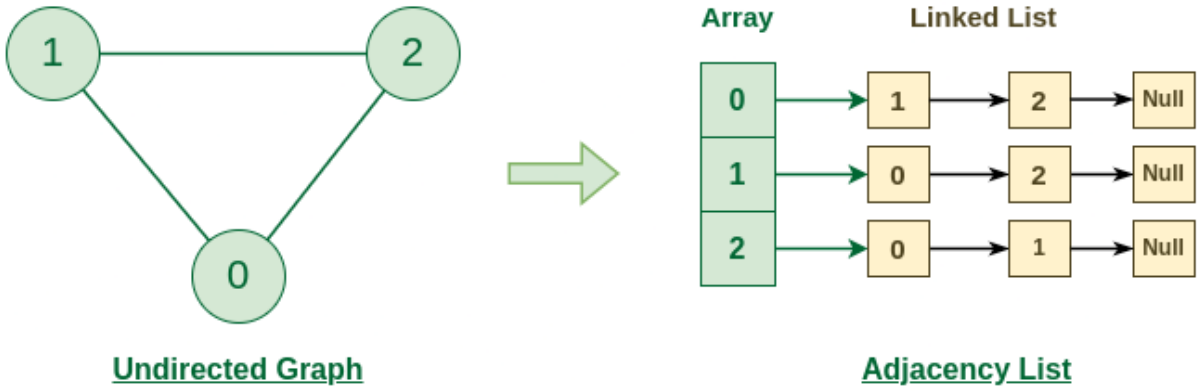
Pre- Lab Discussion:

Representing a graph through the adjacency list saves a lot of space and is more efficient than the other methods that are used to represent a graph in C. To represent a graph using an adjacency list an array of linked lists is created where the index of the array represents the source vertex and all the other adjacent vertices to this node are stored in the form of a linked list in the array.

For example, if there are n vertices in the graph So, we will create an array of list of size n: adjList[n]. Here, adjList[0] will have all the nodes which are connected to vertex 0, called as neighbour vertex 0. Similarly, adjList[1] will have all the nodes which are connected to vertex 1 , and so on.

Representation of Undirected Graph Adjacency List

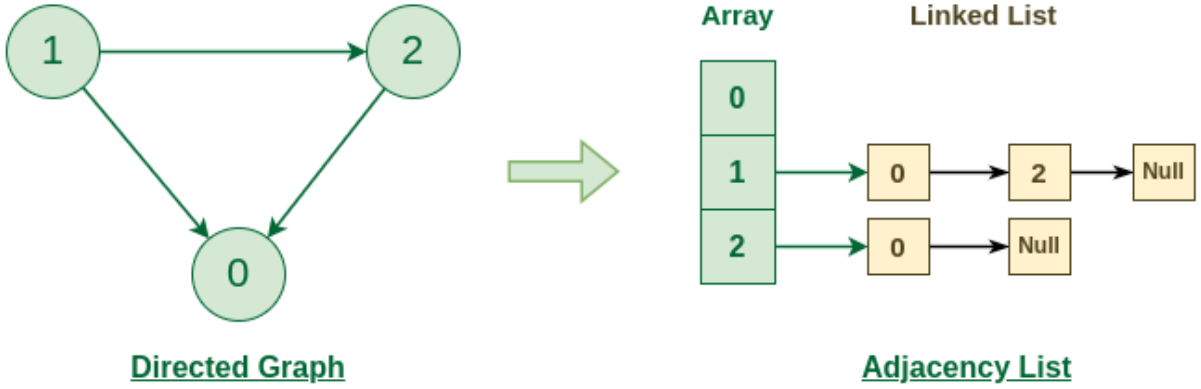
Let us consider the below undirected graph having 3 nodes labeled 0, 1, 2.



Graph Representation of Undirected graph to Adjacency List

First, create an array of lists of size 3 as there are 3 vertices in the graph. Here, we can see that node 0 is connected to both nodes 1 and 2 so, **Vertex 0** has 1 and 2 as its neighbors so, inserting vertex 1 and 2 at index 0 of the array.
Vertex 1 is having 0 and 2 as neighbors so, inserting vertex 0 and 2 at index 1 of the array.
Similarly, **Vertex 2** is having 0 as a neighbor so, inserting vertex 0 at index 2 of the array.

Representation of Directed Graph to Adjacency List
Let us consider the below directed graph having 3 nodes labeled 0, 1, 2.



Graph Representation of Directed graph to Adjacency List

First, create an array of lists of size 3 as there are 3 vertices in the graph. Now, **Vertex 0** has no neighbors so, leaving index 0 of the array empty.
Vertex 1 is having 0 and 2 as neighbors so, inserting vertex 0 and 2 at index 1 of the array.
Similarly, **Vertex 2** is having 0 as a neighbor so, inserting vertex 0 at index 2 of the array.

Algorithm

1. For a graph with |V| vertices, an **adjacency matrix** is a |V|×|V| matrix of 0s and 1s, where the entry in row i and column j is 1 if and only if the edge (i,j) in the graph.
2. Consider the graph to be represented
3. Start from an edge
4. If the edge connects the vertices i,j the mark the i^{th} row and j^{th} column of the adjacency matrix as 1 otherwise store 0
5. Finally print the adjacency matrix

Program

```
#include<stdio.h>
#define V 5

//init matrix to 0
void init(int arr[][V])

{

   int i,j;

   for(i = 0; i < V; i++)
      for(j = 0; j < V; j++)
         arr[i][j] = 0;
}

//Add edge. set arr[src][dest] = 1
void addEdge(int arr[][V],int src, int dest)
```

```
{
    arr[src][dest] = 1;
}
void printAdjMatrix(int arr[][V])
{
    int i, j;

    for(i = 0; i < V; i++)

    {
        for(j = 0; j < V; j++)
        {
            printf("%d ", arr[i][j]);
        }
        printf("\n");
    }
}
//print the adjMatrix
int main()
{
   int adjMatrix[V][V];

   init(adjMatrix);
   addEdge(adjMatrix,0,1);
   addEdge(adjMatrix,0,2);
   addEdge(adjMatrix,0,3);
   addEdge(adjMatrix,1,3);
   addEdge(adjMatrix,1,4);
   addEdge(adjMatrix,2,3);
   addEdge(adjMatrix,3,4);

   printAdjMatrix(adjMatrix);

   return 0;
}
```

OUTPUT

0 1 1 1 0

0 0 0 1 1

0 0 0 1 0

0 0 0 0 1

0 0 0 0 0

Explanation:
The provided C program initializes an adjacency matrix for a directed graph, adds some edges, and prints the adjacency matrix. Here's what the program does step-by-step:

1. **Initialization**:
   - The init function initializes all entries of the adjacency matrix to 0.
2. **Adding Edges**:
   - The addEdge function sets the entry arr[src][dest] to 1 to indicate an edge from vertex src to vertex dest.
3. **Printing the Adjacency Matrix**:
   - The printAdjMatrix function prints the adjacency matrix.

Adjacency Matrix Creation
For the graph described by the edges added in the main function:

- addEdge(adjMatrix, 0, 1); adds an edge from vertex 0 to vertex 1.
- addEdge(adjMatrix, 0, 2); adds an edge from vertex 0 to vertex 2.
- addEdge(adjMatrix, 0, 3); adds an edge from vertex 0 to vertex 3.
- addEdge(adjMatrix, 1, 3); adds an edge from vertex 1 to vertex 3.
- addEdge(adjMatrix, 1, 4); adds an edge from vertex 1 to vertex 4.
- addEdge(adjMatrix, 2, 3); adds an edge from vertex 2 to vertex 3.
- addEdge(adjMatrix, 3, 4); adds an edge from vertex 3 to vertex 4.

1. The row and column indices correspond to the vertices of the graph.
2. A 1 at position arr[i][j] indicates an edge from vertex i to vertex j.
3. A 0 indicates no edge between the vertices.

Viva Questions:

- What is an adjacency list, and how does it differ from an adjacency matrix?
- In what situations would an adjacency matrix be preferable to an adjacency list?
- What are the steps involved in adding an edge to the adjacency list representation?
- How can you implement graph traversal algorithms like BFS or DFS using an adjacency list?
- How would you implement the functionality to remove an edge from the adjacency list representation?

Result : Thus the implementation of graph representation was completed successfully.

Ex.No. 5b. IMPLEMENTATION OF TOPOLOGICAL SORT

AIM:

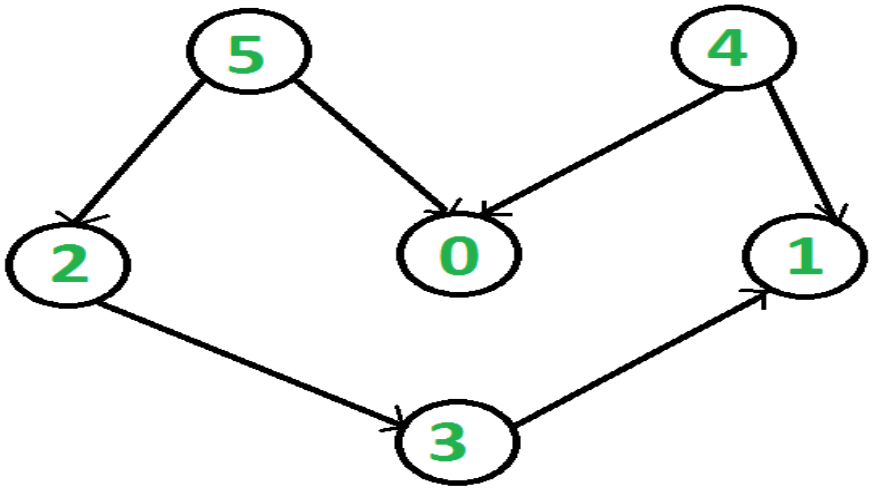
To write a C program to perform topological sorting (Application of a graph).

Pre-Lab Discussion:

Topological Sorting

Topological Sorting is the process in which the main goal is to find an ordering of vertices in a directed acyclic graph (DAG) that places vertex u before vertex v for any directed edge (u, v). "Topological order" is the name given to this linear arrangement. If a DAG contains cycles, it might not have any topological ordering at all.

For example, a topological sorting of the following graph is "5 4 2 3 1 0". There can be more than one topological sorting for a graph. Another topological sorting of the following graph is "4 5 2 3 1 0". The first vertex in topological sorting is always a vertex with an in-degree of 0 (a vertex with no incoming edges).



ALGORITHM:

1. Start the program.

2. Read the number of vertices and adjacency matrix of a graph.

3. Find a vertex with no incoming edges.

4. Delete it along with all the edges outgoing from it.

5. If there are more than one such vertices then break the tie randomly.

6. Store the vertices that are deleted.

7. Display these vertices that give topologically sorted list.

8. Stop the program.

PROGRAM:

```
#include <stdio.h>
int main( )
{
int i,j,k,n,a[10][10],indeg[10],flag[10],count=0;
printf("Enter the no of vertices:\n");
scanf("%d",&n);
printf("Enter the adjacency matrix:\n");
for(i=0;i<n;i++){
printf("Enter row %d\n",i+1);
for(j=0;j<n;j++)
scanf("%d",&a[i][j]);
}
for(i=0;i<n;i++)
{
indeg[i]=0;
flag[i]=0;
}
for(i=0;i<n;i++)
for(j=0;j<n;j++)
indeg[i]=indeg[i]+a[j][i];
printf("\nThe topological order is:");
while(count<n)
{
for(k=0;k<n;k++)
{
if((indeg[k]==0) && (flag[k]==0))
{
```

```
printf("%d ",(k+1));
flag [k]=1;
}
for(i=0;i<n;i++)
{
if(a[i][k]==1)
indeg[k]--;  }  }
count++;
}
return 0;
}
```

OUTPUT :

```
Enter the no of vertices:
4
Enter the adjacency matrix:
Enter row 1
0 1 1 0
Enter row 2
0 0 0 1
Enter row 3
0 0 0 1
Enter row 4
0 0 0 0
The topological order is:1 2 3 4
```

Explanation of the Output:

- The vertices are processed in the order where each vertex is followed by the next vertex it points to.
- For the provided adjacency matrix, the vertices 1, 2, 3, and 4 are processed in this order, reflecting a valid topological sort.

Viva Questions:

- What is topological sorting, and for which type of graphs is it used?
- Why is topological sorting not possible for graphs with cycles?
- How does the algorithm determine which vertex to process next?
- What happens if the graph is not a DAG, and how does the code handle such cases?
- How would you handle graphs with self-loops or multiple edges between the same pair of vertices?

RESULT:

Thus a C program to perform topological sorting is written and executed successfully.

Ex. No.6 IMPLEMENTATION OF A SPANNING TREE FOR A GIVEN GRAPH USING PRIM'S ALGORITHM.

AIM:

To implement a spanning tree for a given graph using Prim's algorithm.

Pre-Lab Discussion:

panning tree - A spanning tree is the subgraph of an undirected connected graph.

Minimum Spanning tree - Minimum spanning tree can be defined as the spanning tree in which the sum of the weights of the edge is minimum. The weight of the spanning tree is the sum of the weights given to the edges of the spanning tree.

Prim's Algorithm is a greedy algorithm that is used to find the minimum spanning tree from a graph. Prim's algorithm finds the subset of edges that includes every vertex of the graph such that the sum of the weights of the edges can be minimized.

Prim's algorithm starts with the single node and explores all the adjacent nodes with all the connecting edges at every step. The edges with the minimal weights causing no cycles in the graph got selected.

How does the prim's algorithm work?

Prim's algorithm is a greedy algorithm that starts from one vertex and continue to add the edges with the smallest weight until the goal is reached. The steps to implement the prim's algorithm are given as follows -

- First, we have to initialize an MST with the randomly chosen vertex.
- Now, we have to find all the edges that connect the tree in the above step with the new vertices. From the edges found, select the minimum edge and add it to the tree.
- Repeat step 2 until the minimum spanning tree is formed.

The applications of prim's algorithm are -

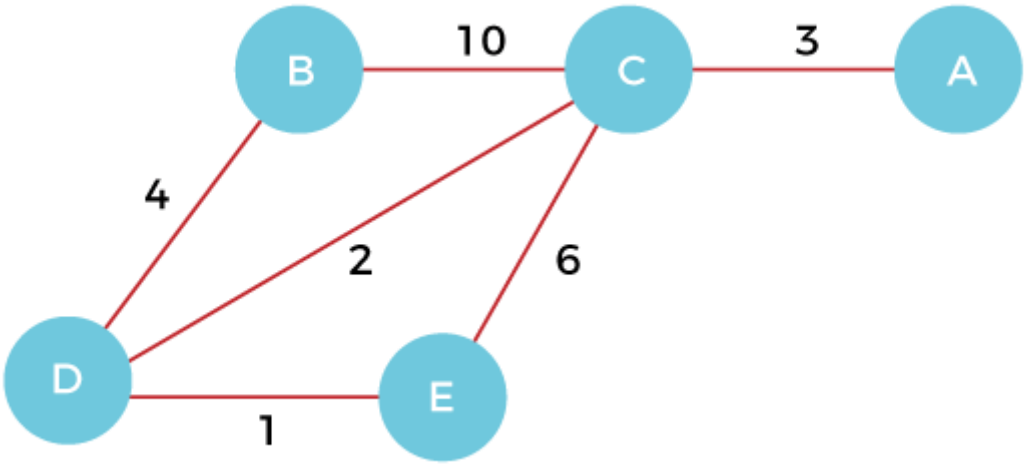
- Prim's algorithm can be used in network designing.
- It can be used to make network cycles.
- It can also be used to lay down electrical wiring cables.

Example of prim's algorithm

Now, let's see the working of prim's algorithm using an example. It will be easier to understand the prim's algorithm using an example.

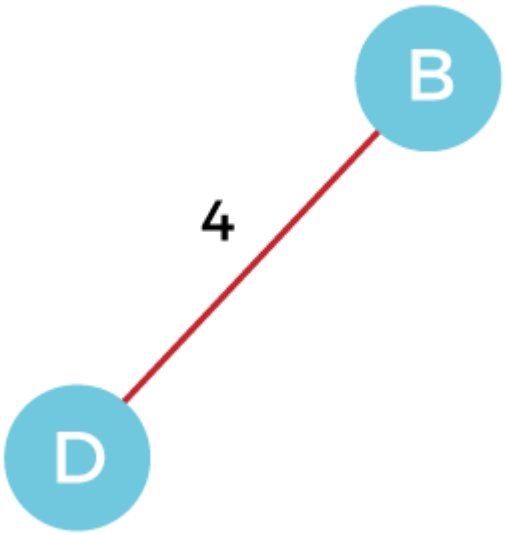
Now, let's see the working of prim's algorithm using an example. It will be easier to understand the prim's algorithm using an example.

Suppose, a weighted graph is -

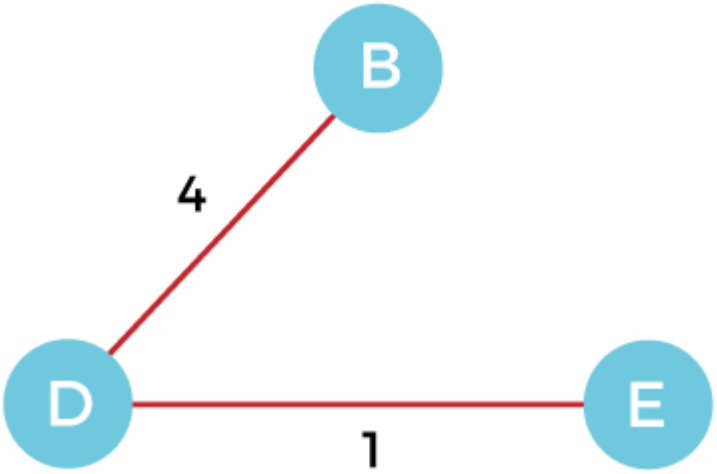


Step 1 - First, we have to choose a vertex from the above graph. Let's choose B.

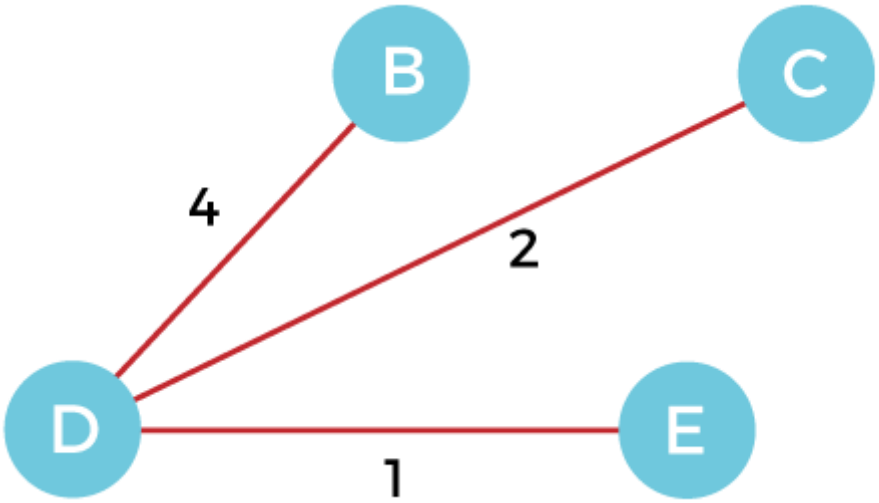
Step 2 - Now, we have to choose and add the shortest edge from vertex B. There are two edges from vertex B that are B to C with weight 10 and edge B to D with weight 4. Among the edges, the edge BD has the minimum weight. So, add it to the MST.



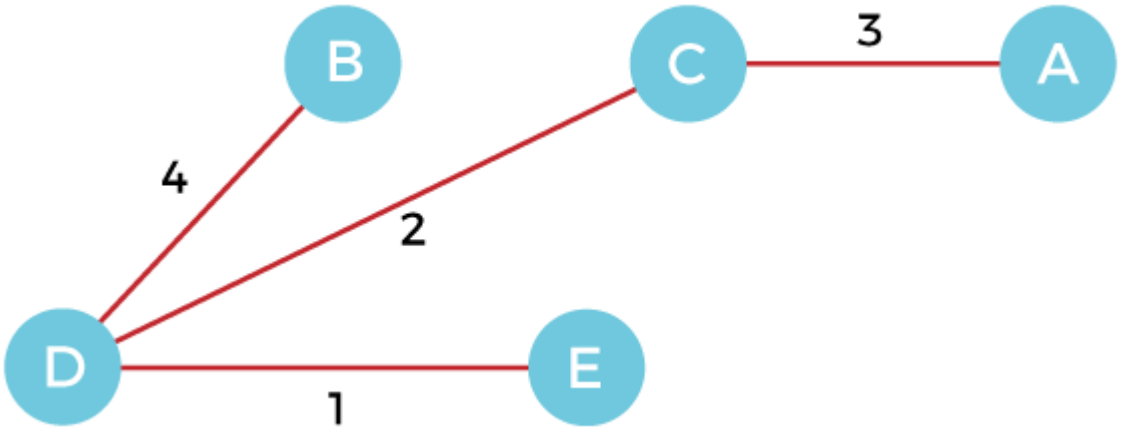
Step 3 - Now, again, choose the edge with the minimum weight among all the other edges. In this case, the edges DE and CD are such edges. Add them to MST and explore the adjacent of C, i.e., E and A. So, select the edge DE and add it to the MST.



Step 4 - Now, select the edge CD, and add it to the MST.



Step 5 - Now, choose the edge CA. Here, we cannot select the edge CE as it would create a cycle to the graph. So, choose the edge CA and add it to the MST.



So, the graph produced in step 5 is the minimum spanning tree of the given graph. The cost of the MST is given below -

Cost of MST = 4 + 2 + 1 + 3 = 10 units.

Algorithm:

The steps for implementing Prim's algorithm are as follows:

1. Initialize the minimum spanning tree with a vertex chosen at random.
2. Find all the edges that connect the tree to new vertices, find the minimum and add it to the tree
3. Keep repeating step 2 until we get a minimum spanning tree

Program:

Prim's Algorithm in C

```
#include<stdio.h>
#include<stdbool.h>
#define INF 9999999
// number of vertices in graph
#define V 5
// create a 2d array of size 5x5
//for adjacency matrix to represent graph
int G[V][V] = {
  {0, 9, 75, 0, 0},
  {9, 0, 95, 19, 42},
  {75, 95, 0, 51, 66},
  {0, 19, 51, 0, 31},
  {0, 42, 66, 31, 0}};
int main() {
  int no_edge;  // number of edge
  // create a array to track selected vertex
  // selected will become true otherwise false
  int selected[V];
```

```
// set selected false initially
memset(selected, false, sizeof(selected));
  // set number of edge to 0
no_edge = 0;
// the number of egde in minimum spanning tree will be
// always less than (V -1), where V is number of vertices in
//graph
// choose 0th vertex and make it true
selected[0] = true;
int x;  //  row number
int y;  //  col number

// print for edge and weight
printf("Edge : Weight\n");

while (no_edge < V - 1) {
  //For every vertex in the set S, find the all adjacent vertices
  // , calculate the distance from the vertex selected at step 1.
  // if the vertex is already in the set S, discard it otherwise
  //choose another vertex nearest to selected vertex  at step 1.

  int min = INF;
  x = 0;
  y = 0;

  for (int i = 0; i < V; i++) {
    if (selected[i]) {
      for (int j = 0; j < V; j++) {
        if (!selected[j] && G[i][j]) {  // not in selected and there is an edge
```

```
          if (min > G[i][j]) {
            min = G[i][j];
            x = i;
            y = j;
          }
        }
      }
    }
    printf("%d - %d : %d\n", x, y, G[x][y]);
    selected[y] = true;
    no_edge++;
  }

  return 0;
}
```

Output

Edge : Weight

0 - 1 : 9

1 - 3 : 19

3 - 4 : 31

3 - 2 : 51

Explanation:

Graph Representation (Adjacency Matrix)

The adjacency matrix G[V][V] represents a graph with 5 vertices and the following edges and weights:

	0	1	2	3	4
0	0	9	75	0	0
1	9	0	95	19	42
2	75	95	0	51	66
3	0	19	51	0	31
4	0	42	66	31	0

1. The first edge chosen is 0 - 1 with weight 9.
2. From vertex 1, the next edge chosen is 1 - 3 with weight 19.
3. From vertex 3, the next edge chosen is 3 - 4 with weight 31.
4. Finally, the edge 3 - 2 with weight 51 is added to complete the MST.

This gives the final MST with a total weight of 9+19+31+51=110

Note:

- The adjacency matrix has some 0 entries, indicating no direct edge between certain vertices.
- The program uses Prim's algorithm to pick the smallest weight edges, connecting all vertices while avoiding cycles.

Viva Questions:

1. What is Prim’s Algorithm and what is it used for?
2. How does Prim's Algorithm differ from Kruskal's Algorithm?
3. How does the algorithm determine the next vertex to add to the MST?
4. What are the time and space complexities of Prim’s Algorithm when implemented with an adjacency matrix?
5. How would you modify the program to use an adjacency list instead of an adjacency matrix?

RESULT:

Thus, a C program to implement a spanning tree for a given graph using Prim’s algorithm is written and executed successfully

Ex.No:7 IMPLEMENTATION OF DIJKSTRA'S ALGORITHM

AIM

To write a C program to implement the shortest path using Dijkstra's algorithm.

PRE LAB-DISCUSSION

Dijkstra's algorithm has many variants but the most common one is to find the shortest paths from the source vertex to all other vertices in the graph.

The following are the basic concepts of Dijkstra's Algorithm:

1. Dijkstra's Algorithm begins at the node we select (the source node), and it examines the graph to find the shortest path between that node and all the other nodes in the graph.
2. The Algorithm keeps records of the presently acknowledged shortest distance from each node to the source node, and it updates these values if it finds any shorter path.
3. Once the Algorithm has retrieved the shortest path between the source and another node, that node is marked as 'visited' and included in the path.
4. The procedure continues until all the nodes in the graph have been included in the path. In this manner, we have a path connecting the source node to all other nodes, following the shortest possible path to reach each node.

Understanding the Working of Dijkstra's Algorithm:

A graph and source vertex are requirements for Dijkstra's Algorithm. This Algorithm is established on Greedy Approach and thus finds the locally optimal choice (local minima in this case) at each step of the Algorithm.

Each Vertex in this Algorithm will have two properties defined for it:

1. Visited Property
2. Path Property

Let us understand these properties in brief.

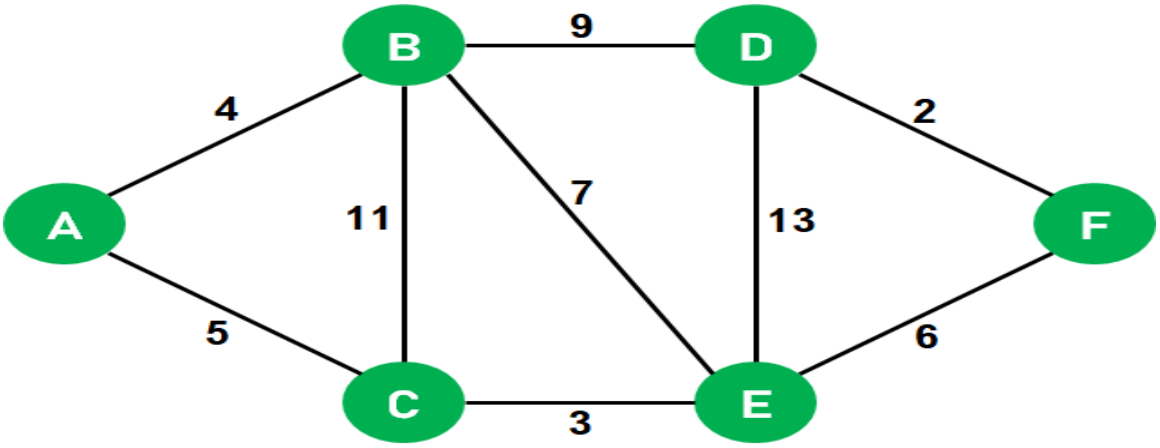
1. **Visited Property:**

- The 'visited' property signifies whether or not the node has been visited.
- We are using this property so that we do not revisit any node.
- A node is marked visited only when the shortest path has been found.

2. **Path Property:**

- The 'path' property stores the value of the current minimum path to the node.
- The current minimum path implies the shortest way we have reached this node till now.
- This property is revised when any neighbor of the node is visited.
- This property is significant because it will store the final answer for each node.

Example:



1. A = 0
2. B = 4 (A -> B)
3. C = 5 (A -> C)
4. D = 4 + 9 = 13 (A -> B -> D)
5. E = 5 + 3 = 8 (A -> C -> E)
6. F = 5 + 3 + 6 = 14 (A -> C -> E -> F)

ALGORITHM

1. Read the number of vertices n and the number of edges edges.

2. Initialize the adjacency matrix graph with zeros.

3. Read each edge (u, v, w) and populate the adjacency matrix with weights.

4. Read the source vertex src.

5. Initialize dist[] array where dist[i] will hold the shortest distance from src to vertex i. Set all distances to infinity (INF) except the distance to the source itself, which is zero.

6. Initialize a visited[] array to keep track of vertices for which the minimum distance from the source is calculated. Set all entries to 0 (false).

7. Find the Vertex with Minimum Distance:

   Create a helper function minDistance() that scans the dist[] array to find the vertex with the minimum distance that hasn't been visited yet.

8. Repeat the following steps n-1 times (for each vertex):

   Select the vertex u with the minimum distance from the dist[] array using minDistance().

   Mark vertex u as visited. Update dist[] for each adjacent vertex v of u. For each vertex v:

   Check if v is not visited. Check if there is an edge from u to v.

   Check if the total weight of the path from the source to v through u is smaller than the current value of dist[v]. If so, update dist[v].

9. Print the Result:

10. After the main loop completes, print the shortest distances from the source to all vertices.

PROGRAM

```
#include <stdio.h>
#include <limits.h>
#define MAX 100
#define INF INT_MAX
int n; // Number of vertices in the graph
int graph[MAX][MAX]; // Adjacency matrix representation of the graph
int minDistance(int dist[], int visited[]) {
   int min = INF, min_index;
   for (int v = 0; v < n; v++) {
      if (!visited[v] && dist[v] <= min) {
         min = dist[v];
         min_index = v;
      }
   }
   return min_index;
}
void dijkstra(int src) {
   int dist[MAX];
   int visited[MAX] = {0};

   for (int i = 0; i < n; i++) {
      dist[i] = INF;
   }
   dist[src] = 0;

   for (int count = 0; count < n - 1; count++) {
      int u = minDistance(dist, visited);
      visited[u] = 1;

      for (int v = 0; v < n; v++) {
         if (!visited[v] && graph[u][v] && dist[u] != INF && dist[u] + graph[u][v] < dist[v]) {
            dist[v] = dist[u] + graph[u][v];
         }
      }
   }

   printf("Vertex \t Distance from Source\n");
   for (int k = 0; k < n; k++) {
      printf("%d \t %d\n", k, dist[k]);
   }
}
int main() {
   int edges, u, v, w, src;
   printf("Enter the number of vertices: ");
   scanf("%d", &n);
```

```
  printf("Enter the number of edges: ");
    scanf("%d", &edges);

    // Initialize the graph with 0s
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            graph[i][j] = 0;
        }
    }

    for (int m = 0; m < edges; m++) {
        printf("Enter edge (u v w): ");
        scanf("%d %d %d", &u, &v, &w);
        graph[u][v] = w;
        graph[v][u] = w; // If the graph is undirected
    }

    printf("Enter the source vertex: ");
    scanf("%d", &src);
    dijkstra(src);

    return 0;
}
```

OUTPUT

```
Enter the number of vertices: 5
Enter the number of edges: 7
Enter edge (u v w): 0 1 10
Enter edge (u v w): 0 4 20
Enter edge (u v w): 1 2 10
Enter edge (u v w): 1 3 50
Enter edge (u v w): 1 4 10
Enter edge (u v w): 2 3 10
Enter edge (u v w): 3 4 30

Enter the source vertex: 0

Vertex   Distance from Source
0        0
1        10
2        20
3        30
4        20
```

EXPLANATION

Initialization:

- Number of vertices: 5
- Number of edges: 7
- The adjacency matrix is initialized with all zeros.
- The edges are then populated into the adjacency matrix as input.

Graph Representation: The adjacency matrix after input:

```
0   10   0   0   20
10   0  10  50  10
0   10   0  10   0
0   50  10   0  30
20  10   0  30   0
```

1. **Dijkstra's Algorithm Execution:**
    - Start from vertex 0.
    - Set distance of the source vertex (0) to 0.
    - Initialize all other distances to infinity.
2. **Step-by-Step Execution:**
    - **Step 1:** Visit vertex 0 (distance 0).
        - Update distances of neighbors: 1 (10), 4 (20).
    - **Step 2:** Visit vertex 1 (distance 10).
        - Update distances of neighbors: 2 (20), 3 (60), 4 (20).
    - **Step 3:** Visit vertex 4 (distance 20).
        - Update distances of neighbors: 3 (50).
    - **Step 4:** Visit vertex 2 (distance 20).
        - Update distances of neighbors: 3 (30).
    - **Step 5:** Visit vertex 3 (distance 30).
        - No further updates as all neighbors have been visited or have shorter paths already.

Final Result:

The shortest distances from the source vertex (0) to all other vertices are:

- Vertex 0: Distance 0
- Vertex 1: Distance 10
- Vertex 2: Distance 20
- Vertex 3: Distance 30
- Vertex 4: Distance 20

The output shows the minimum distance from the source vertex to each of the other vertices using Dijkstra's algorithm.

VIVA QUESTIONS

1. How to find the adjacency matrix?
2. Which algorithm solves the single pair shortest path problem?
3. How to find shortest distance from source vertex to target vertex?
4. What is the maximum possible number of edges in a directed graph with no self-loops having 8vertices?
5. Does Dijkstra’s Algorithm work for both negative and positive weights?

RESULT

Thus, the C program to implement shortest path using Dijkstra’s algorithm was completed successfully.

Ex.No. 8. IMPLEMENTATION OF ITERATIVE AND RECURSIVE ALGORITHMS WITH ITS COMPLEXITY ANALYSIS.

AIM:

To implement iterative and recursive algorithms with its complexity analysis using C Program.

A program is called recursive when an entity calls itself. A program is called iterative when there is a loop (or repetition).

Algorithm: (using Recursion)

1. Start.
2. Check the base case:
3. If n == 0, return 1. (By definition, 0! = 1).
4. Recursive Step: If n > 0, return n * factorial(n - 1).
5. This step reduces the problem size by calculating the factorial of n-1 recursively until it reaches 0.
6. End.

Algorithm: (using Iteration)

1. Start.
2. Initialize a variable res to 1. This will store the result of the factorial calculation.
3. For loop iteration:
   Set up a loop from i = 2 to n (both inclusive).
   For each value of i, multiply res by i and store the result back in res.
4. After the loop ends, res will hold the value of n!.
5. Return the value of res.
6. End.

Program:

```
// C program to find factorial of given number

#include <stdio.h>


// ----- Recursion -----

// method to find factorial of given number

int factorialUsingRecursion(int n)

{

   if (n == 0)

      return 1;


   // recursion call
```

```
    return n * factorialUsingRecursion(n - 1);
}
 // ----- Iteration -----
// Method to find the factorial of a given number
int factorialUsingIteration(int n)
{
    int res = 1, i;
     // using iteration
    for (i = 2; i <= n; i++)
        res *= i;
     return res;
}
 // Driver method
int main()
{
    int num = 5;
    printf("Factorial of %d using Recursion is: %d\n", num,
           factorialUsingRecursion(5));
     printf("Factorial of %d using Iteration is: %d", num,
           factorialUsingIteration(5));
     return 0;
}
```

Output

Factorial of 5 using Recursion is: 120

Factorial of 5 using Iteration is: 120

Complexity Analysis:

1. Recursive Factorial

Time Complexity:

- Each recursive call decreases n by 1, and the function is called n times before reaching the base case (n == 0).
- Therefore, the time complexity is **O(n)**.

Space Complexity:

- Each recursive call adds a new frame to the function call stack.
- Since there are n recursive calls, the space complexity due to the recursion stack is **O(n)**.

2. Iterative Factorial

Time Complexity:

- The for loop iterates from 2 to n, performing n-1 iterations.
- Therefore, the time complexity is **O(n)**.

Space Complexity:

- In the iterative approach, only a few variables are used (res, i), and no extra space is required for recursive calls.
- Therefore, the space complexity is **O(1)**.

Algorithm	Time Complexity	Space Complexity
Recursive Factorial	O(n)	O(n)
Iterative Factorial	O(n)	O(1)

Conclusion:

- Both the recursive and iterative algorithms have the same time complexity of **O(n)**.
- The recursive algorithm has higher space complexity **O(n)** due to the recursion call stack, while the iterative approach is more efficient in terms of space with **O(1)** complexity.

RESULT

Thus the program to implement iterative and recursive algorithms with its complexity analysis is successfully executed and the output is verified.

Ex. No. 9 IMPLEMENTATION OF MERGE SORT ALGORITHM ANALYSIS USING DIVIDE AND CONQUER APPROACH

AIM:

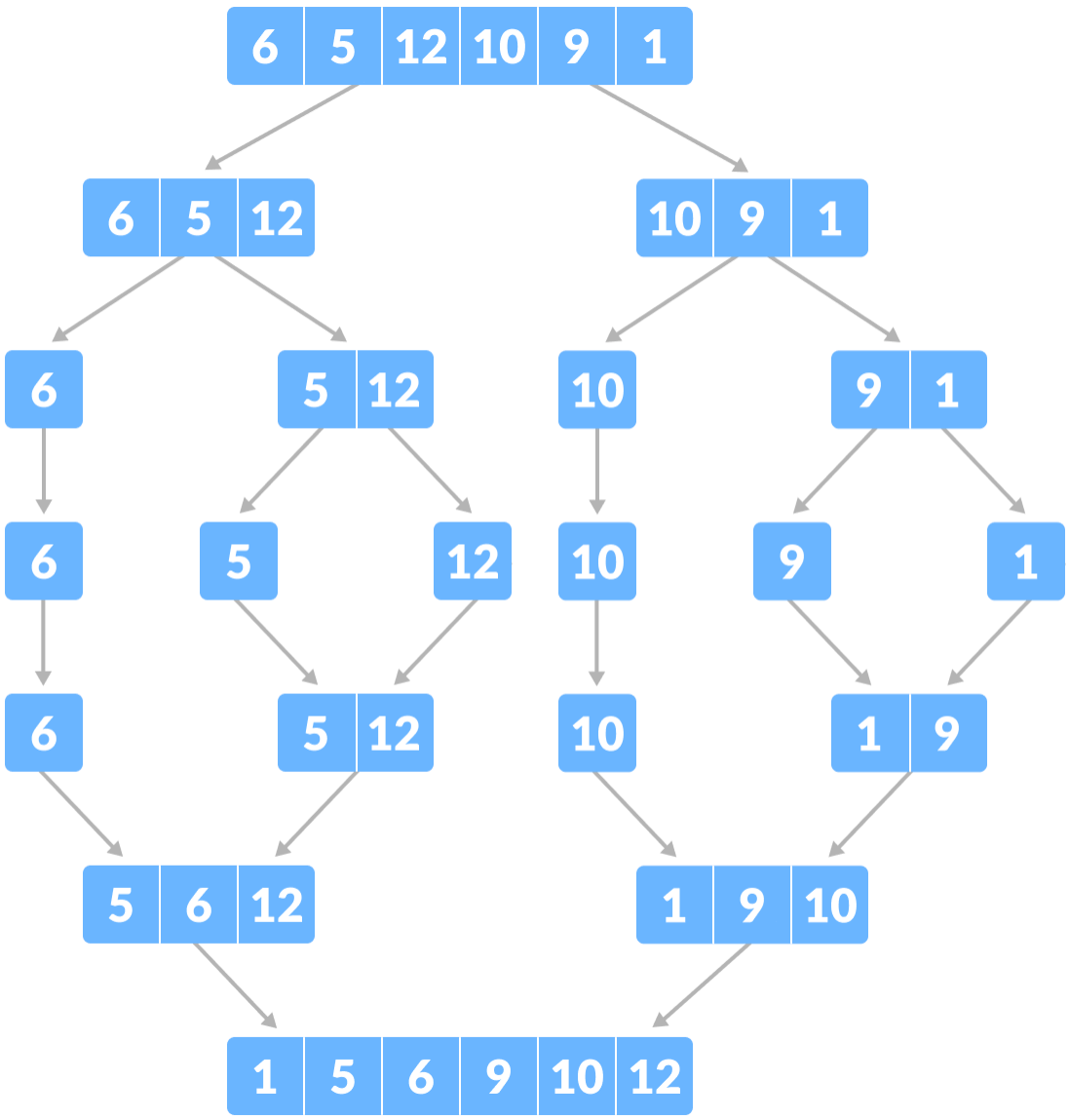
To sort an array of N numbers using Merge sort.

Pre Lab-Discussion:

Merge Sort Algorithm

Merge Sort is one of the most popular sorting algorithms that is based on the principle of Divide and Conquer Algorithm.

Here, a problem is divided into multiple sub-problems. Each sub-problem is solved individually. Finally, sub-problems are combined to form the final solution.



Divide and Conquer Strategy

Using the **Divide and Conquer** technique, we divide a problem into subproblems. When the solution to each subproblem is ready, we 'combine' the results from the subproblems to solve the main problem.

Suppose we had to sort an array A. A subproblem would be to sort a sub-section of this array starting at index p and ending at index r, denoted as A[p..r].

Divide

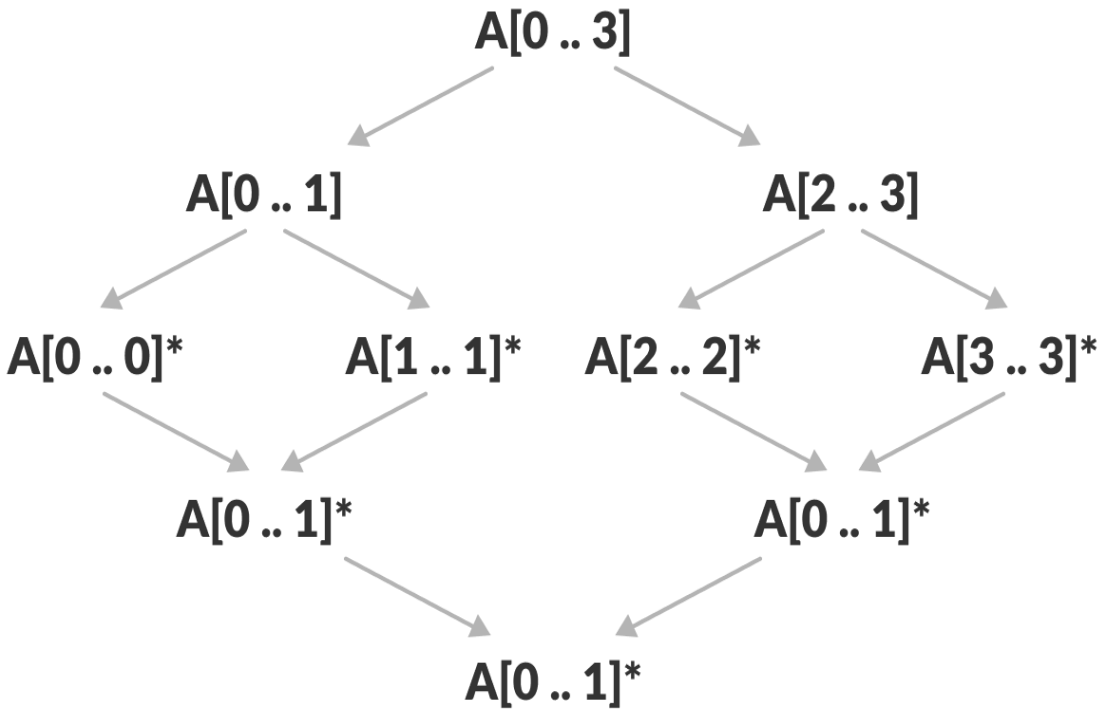
If q is the half-way point between p and r, then we can split the subarray A[p..r] into two arrays A[p..q] and A[q+1, r].

Conquer

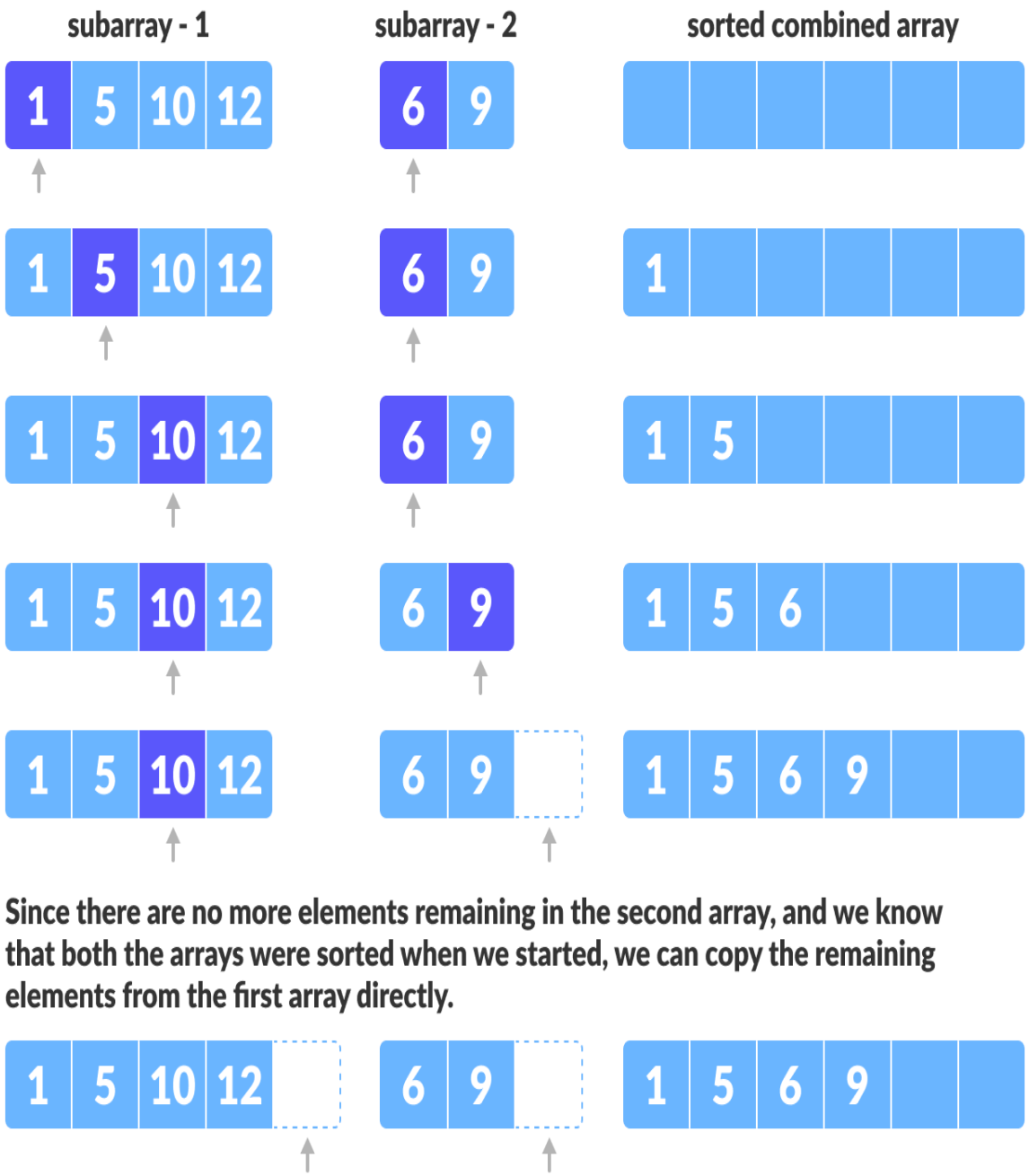
In the conquer step, we try to sort both the subarrays A[p..q] and A[q+1, r]. If we haven't yet reached the base case, we again divide both these subarrays and try to sort them.

Combine

When the conquer step reaches the base step and we get two sorted subarrays A[p..q] and A[q+1, r] for array A[p..r], we combine the results by creating a sorted array A[p..r] from two sorted subarrays A[p..q] and A[q+1, r].



Example:



ALGORITHM

1. Start
2. Read number of array elements n
3. Read array elements Ai
4. Divide the array into sub-arrays with a set of elements
5. Recursively sort the sub-arrays
6. Display both sorted sub-arrays
7. Merge the sorted sub-arrays onto a single sorted array.
8. Display the merge sorted array elements
9. Stop

PROGRAM

```
/* 11d – Merge sort */
#include <stdio.h>
#include <conio.h>
void merge(int [],int ,int ,int );
void part(int [],int ,int );
int size;
main()
{

    int i, arr[30];
    printf("Enter total no. of elements : ");
    scanf("%d", &size);
    printf("Enter array elements : ");
    for(i=0; i<size; i++)
    scanf("%d", &arr[i]);
    part(arr, 0, size-1);
    printf("\n Merge sorted list : ");
    for(i=0; i<size; i++)
    printf("%d ",arr[i]);
    getch();
}


void part(int arr[], int min, int max)
{

    int mid;
    if(min < max)
    {

        mid = (min + max) / 2;
        part(arr, min, mid);
        part(arr, mid+1, max);
        merge(arr, min, mid, max);
    }

    if (max-min == (size/2)-1)


    {

        printf("\n Half sorted list : ");
        for(i=min; i<=max; i++)
        printf("%d ", arr[i]);
    }

}


void merge(int arr[],int min,int mid,int max)
```

```
{

    int  tmp[30];
    int i, j, k, m;
    j = min;
    m = mid + 1;
    for(i=min; j<=mid && m<=max; i++)
    {

        if(arr[j] <= arr[m])
        {

            tmp[i] = arr[j];
            j++;
        }

        else
        {

            tmp[i] = arr[m];
            m++;
        }

    }

    if(j > mid)
    {

        for(k=m; k<=max; k++)
        {

            tmp[i] = arr[k];
            i++;
        }

    }

    else
    {

        for(k=j; k<=mid; k++)
        {

            tmp[i] = arr[k];
            i++;
        }

    }

    for(k=min; k<=max; k++)
    arr[k] = tmp[k];
}
```

OUTPUT

Enter total no. of elements: 6
Enter array elements: 38 27 43 3 9 82

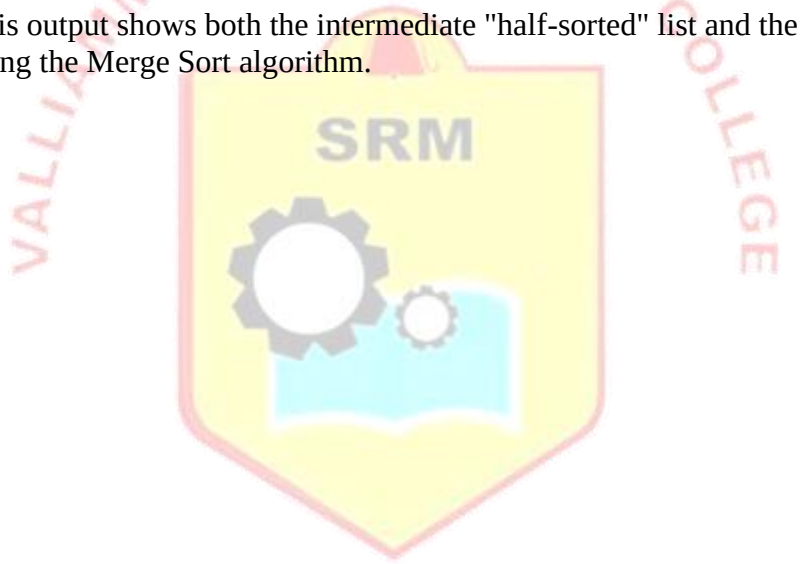
Half sorted list: 3 9 27
Half sorted list: 38 43 82

Merge sorted list: 3 9 27 38 43 82

Explanation:

1. The array is first divided into two parts:
   Left: {38, 27, 43}
   Right: {3, 9, 82}
2. The program prints each half when it is divided:
   First half (left side sorted): {3, 9, 27}
   Second half (right side sorted): {38, 43, 82}
3. Finally, the two halves are merged together to form the fully sorted array:
   Final sorted array: {3, 9, 27, 38, 43, 82}

   This output shows both the intermediate "half-sorted" list and the fully sorted list using the Merge Sort algorithm.



RESULT

Thus array elements was sorted using merge sort's divide and conquer method.

Ex.No. 10 IMPLEMENTATION OF MATRIX CHAIN MULTIPLICATION USING DYNAMIC PROGRAMMING APPROACH

(C Program for Matrix Chain Multiplication using Dynamic Programming (Tabulation))

Aim:

To write a C Program for Matrix Chain Multiplication using Dynamic Programming.

Pre-Lab Discussion:

What is Matrix Chain Multiplication?

Consider a chain of matrices to be multiplied together. For example, we have matrices A, B, C, and D. The goal is to find the most efficient order of multiplication that minimizes the total number of scalar multiplications required. In this case, we can multiply them in different orders: ((AB)C)D, (A(BC))D, or A((BC)D). The Matrix Chain Algorithm helps us determine the optimal order.

The rule for matrix chain multiplication states that the order in which matrices are multiplied together can significantly impact the computational efficiency of the multiplication process. In other words, the way matrices are grouped and multiplied affects the total number of scalar multiplications required.

To illustrate this rule, consider a chain of matrices to be multiplied: A1, A2, A3, …, An. The result of multiplying this chain depends on how the matrices are parenthesized and the order in which the multiplications are performed.

For example, given matrices A, B, and C, we can have two possible ways to multiply them:

1. (AB)C: In this case, we first multiply matrices A and B, and then multiply the resulting matrix with matrix C.
2. A(BC): Here, we multiply matrices B and C first, and then multiply matrix A with the resulting matrix.

The rule is that the order of multiplication can affect the number of scalar multiplications required to obtain the final result. In some cases, different parenthesizations can lead to significantly different numbers of scalar multiplications.

For instance, the number of scalar multiplications required for the product (AB)C can be fewer than the number required for A(BC). The rule emphasizes that the order of multiplication should be chosen in a way that minimizes the total number of scalar multiplications in order to optimize the matrix chain multiplication process.

Matrix Chain Multiplication Example

Let’s consider an example to illustrate the matrix chain multiplication.

Suppose we have a chain of four matrices: A, B, C, and D, with dimensions as follows:

A: 10 x 30

B: 30 x 5

C: 5 x 60

D: 60 x 8

The goal is to find the optimal order of multiplying these matrices that minimizes the total number of scalar multiplications.

To solve this problem, we can apply the Matrix Chain Multiplication algorithm using dynamic programming. Let's go step by step:

Step 1: Create the table We create a 4×4 table to store the minimum number of scalar multiplications needed for each subchain. The table initially looks like this:

0			
	0		
		0	
			0

Step 2: Fill in the table We iterate diagonally through the table, calculating the minimum number of scalar multiplications for each subchain.

First, we consider subchains of length 2: For the subchain AB, the only option is multiplying matrices A and B, resulting in 10 x 5 scalar multiplications. For the subchain BC, the only option is multiplying matrices B and C, resulting in 30 x 5 scalar multiplications. For the subchain CD, the only option is multiplying matrices C and D, resulting in 5 x 8 scalar multiplications.

Updating the table:

0	150		
	0	300	
		0	240
			0

Next, we consider subchains of length 3: For the subchain ABC, we have two options: (AB)C or A(BC). Let's calculate the scalar multiplications for each option:

- (AB)C: We multiply matrices A and B (10 x 30) resulting in a temporary matrix of size 10 x 5. Then, we multiply this temporary matrix with matrix C (10 x 5) resulting in a final matrix of size 10 x 60. The total scalar multiplications for this option are 10 x 30 x 5 + 10 x 5 x 60 = 4500.
- A(BC): We multiply matrices B and C (30 x 5) resulting in a temporary matrix of size 30 x 60. Then, we multiply matrix A (10 x 30) with this temporary matrix, resulting in a final matrix of size 10 x 60. The total scalar multiplications for this option are 10 x 30 x 60 + 10 x 5 x 60 = 9000.

Updating the table:

0	150	4500	
	0	300	240
		0	240
			0

Finally, we consider the entire chain of matrices ABCD: We have two options: ((AB)C)D or (A(BC))D. Calculating the scalar multiplications for each option:

- ((AB)C)D: We multiply matrices (AB) and C, resulting in a temporary matrix of size 10 x 60. Then, we multiply this temporary matrix with matrix D (10 x 60), resulting in a final

matrix of size 10 x 8. The total scalar multiplications for this option are 10 x 5 x 60 + 10 x 60 x 8 = 6000.

- (A(BC))D: We multiply matrices A and (BC), resulting in a temporary matrix of size 10 x 60. Then, we multiply this temporary matrix with matrix D (10 x 60), resulting in a final matrix of size 10 x 8. The total scalar multiplications for this option is 10 x 30 x 5 + 10 x 5 x 8 = 1300.

Updating the table:

0	150	4500	6000
	0	300	240
		0	240
			0

Step 3: Backtrack the optimal solution To determine the optimal order of multiplication, we start at M[1, 4], which gives us the minimum number of scalar multiplications for the entire chain. In this case, M[1, 4] = 6000, indicating that ((AB)C)D is the optimal order.

Step 4: Return the result The minimum number of scalar multiplications required to multiply the entire chain is 6000, and the optimal order is ((AB)C)D.

That's an example of how the Matrix Chain Multiplication algorithm works. By applying dynamic programming, we can find the optimal order and minimize the computational cost of multiplying a chain of matrices.

Algorithm:

1. Iterate from **l = 2 to N-1** which denotes the length of the range:
    a. Iterate from **i = 0 to N-1**:
        i. Find the right end of the range (**j**) having **l** matrices.
        ii. Iterate from **k = i+1 to j** which denotes the point of partition.
            1. Multiply the matrices in range (**i, k**) and (**k, j**).
            2. This will create two matrices with dimensions **arr[i-1]*arr[k]** and **arr[k]*arr[j]**.

3. The number of multiplications to be performed to multiply these two matrices (say **X**) are **arr[i-1]*arr[k]*arr[j]**.
4. The total number of multiplications is **dp[i][k]+ dp[k+1][j] + X**.

2. The value stored at **dp[1][N-1]** is the required answer.

Program:

```
#include <limits.h>

#include <stdio.h>

 // Matrix Ai has dimension p[i-1] x p[i] for i = 1..n

int MatrixChainOrder(int p[], int n)

{

    /* For simplicity of the program,

    one extra row and one

    extra column are allocated in m[][].

    0th row and 0th

    column of m[][] are not used */

    int m[n][n];


    int i, j, k, L, q;


    /* m[i, j] = Minimum number of

    scalar multiplications

    needed to compute the matrix

    A[i]A[i+1]...A[j] =

    A[i..j] where dimension of A[i]

    is p[i-1] x p[i] */
```

```
    // cost is zero when multiplying one matrix.
    for (i = 1; i < n; i++)
        m[i][i] = 0;

    // L is chain length.
    for (L = 2; L < n; L++) {
        for (i = 1; i < n - L + 1; i++)
        {
            j = i + L - 1;
            m[i][j] = INT_MAX;
            for (k = i; k <= j - 1; k++)
            {
                // q = cost/scalar multiplications
                q = m[i][k] + m[k + 1][j]
                    + p[i - 1] * p[k] * p[j];
                if (q < m[i][j])
                    m[i][j] = q;
            }
        }
    }

    return m[1][n - 1];
}

// Driver code
```

```
int main()
{
    int arr[] = { 1, 2, 3, 4 };
    int size = sizeof(arr) / sizeof(arr[0]);

    printf("Minimum number of multiplications is %d ",
        MatrixChainOrder(arr, size));

    getchar();
    return 0;
}
```

Output

Minimum number of multiplications is 18

Explanation:

The provided code implements the **Matrix Chain Multiplication** dynamic programming algorithm to find the minimum number of scalar multiplications required to multiply a sequence of matrices. The dimensions of the matrices are specified by the array arr[].

Input:

The array arr[] = {1, 2, 3, 4} specifies the dimensions of the matrices as follows:

- Matrix A1: 1 × 2
- Matrix A2: 2 × 3
- Matrix A3: 3 × 4

The MatrixChainOrder function will compute the minimum number of scalar multiplications required to multiply these three matrices together. So, the **Output is "**Minimum number of multiplications is **18"**

Viva Questions:

1. **What is the Matrix Chain Multiplication problem?**
2. Why can't we directly multiply matrices in sequence?
3. What is Dynamic Programming, and why is it used here?
4. What is the time complexity of the Matrix Chain Multiplication algorithm using dynamic programming?
5. What happens if the input matrix dimensions are not compatible for multiplication?

RESULT:

Thus, the C Program for Matrix Chain Multiplication using Dynamic Programming is executed successfully and the output is verified.

Ex.No.11 IMPLEMENTATION OF HUFFMAN CODING USING GREEDY APPROACH

AIM:

To write a C program to implement Huffman coding using Greedy approach.

Pre-Lab Discussion:

Huffman coding

Huffman coding was introduced by David Huffman. It is a lossless data compression methodology used before sending the data so that data can be compressed and sent using minimal bits, without redundancy, and without losing any of the details. It generally compresses the most frequently occurring characters.

The characters which have more frequency are assigned with the smallest code and the characters with larger frequency get assigned with the largest code.

The codes are assigned in such a manner that each code will be unique and different for each character.

Prefix rule

Huffman coding is implemented using the prefix rule. The variable-length codes assigned to the characters based on their frequency are the **Prefix code.** Prefix codes are used to remove ambiguity during the decoding of the data, no two characters can have the same prefix code.

Major steps in building a Huffman tree

Huffman coding technique involves two major steps, which are as follows:

- Creating a Huffman tree of the input characters.
- Traversing the tree and assigning codes to the characters.

Working or Creation of Huffman Tree

Huffman Coding step-by-step working or creation of Huffman Tree is as follows:

- Step-1: Calculate the frequency of each string.
- Step-2: Sort all the characters on the basis of their frequency in ascending order.
- Step-3: Mark each unique character as a leaf node.
- Step-4: Create a new internal node.

- Step-5: The frequency of the new node as the sum of the single leaf node
- Step-6: Mark the first node as this left child and another node as the right child of the recently created node.
- Step-7: Repeat all the steps from step-2 to step-6.

Formulas Used in Huffman Tree

Average code length per character = ∑(frequencyi x code lengthi)/ ∑ frequencyi

Total number of bits in Huffman encoded message

= Total number of characters in the message x Average code length per character

= ∑ (frequency_i x Code length_i)

Example:

Suppose a data file has the following characters and the frequencies. If huffman coding is used, calculate:

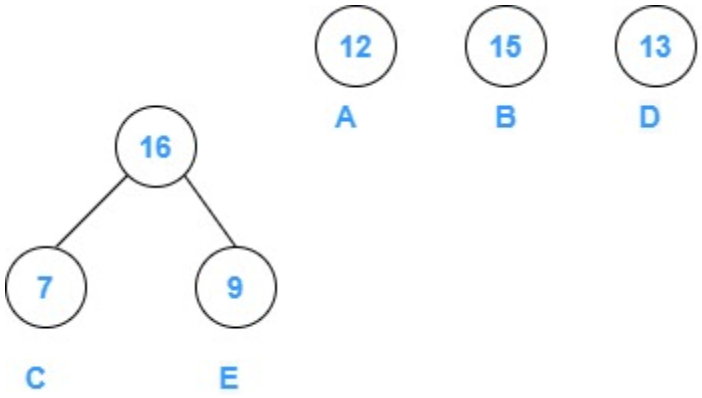
- Huffman Code of each character
- Average code length
- Length of Huffman encoded data

Characters	Frequencies
A	12
B	15
C	7
D	13
E	9

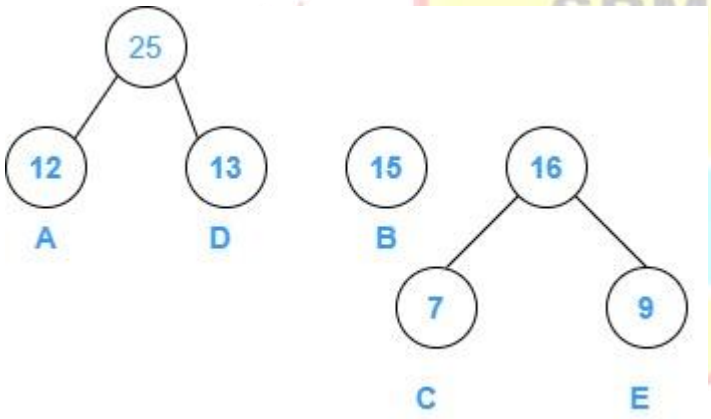
Solution:

Initially, create the Huffman Tree:

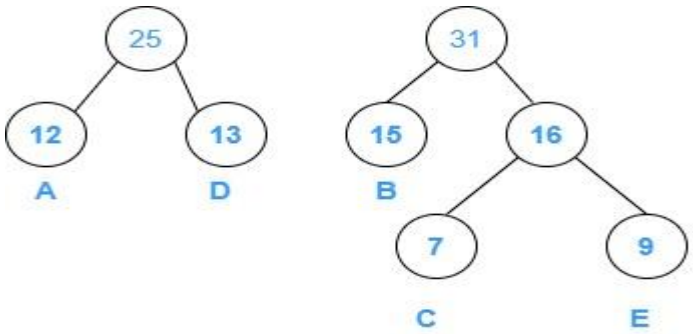
Step-2:



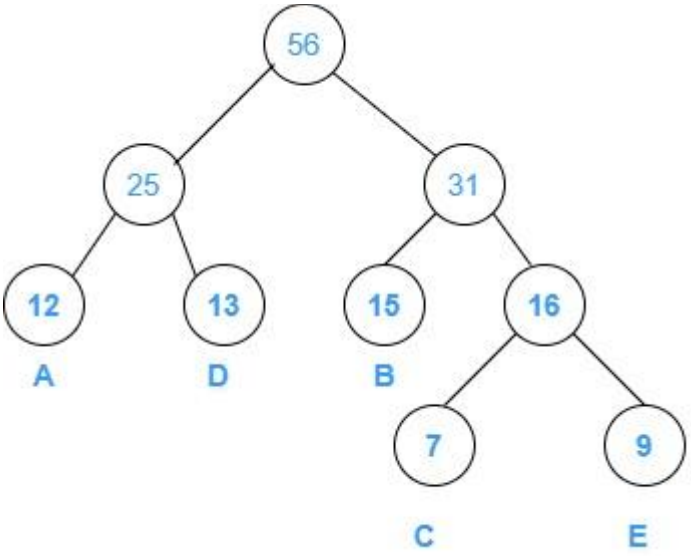
Step-3:



Step-4:



Step-5:

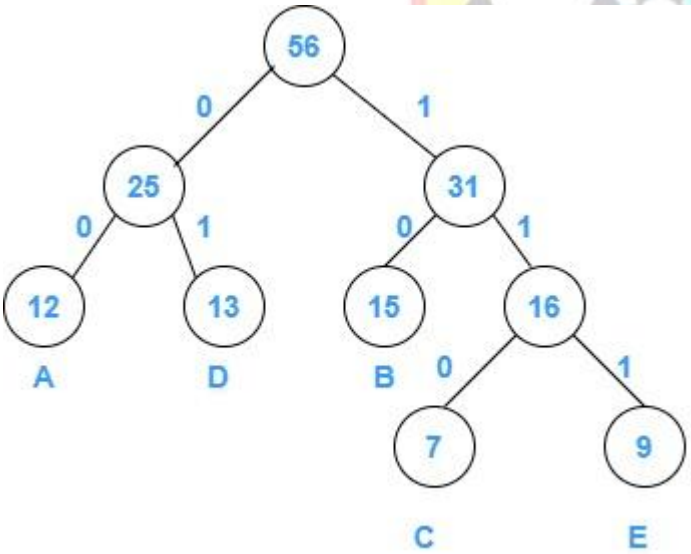


The above tree is a Huffman Tree.

Now, assign weight to all the nodes.

Assign “0” to all left edges and “1” to all right edges.

The tree will become



Huffman Code of each character:

A: 00

B: 10

C: 110

D: 01

E: 111

Average code length per character = ∑(frequencyi x code lengthi)/ ∑ frequencyi

= {(12 x 2) + (13 x 2)+ (15 x 2)+ (7 x 3)+ (9 x 3)} / (12 + 13 + 15 + 7+ 9)

= (24+ 26+ 30 + 21 + 27)/56

= 128/56

= 2.28

Average code length per character = 2.28

Total number of bits in Huffman encoded message

= Total number of characters in the message x Average code length per character

= 56 x 2.28

= 127.68

Algorithm:

Step-1: Calculate the frequency of each string.

Step-2: Sort all the characters on the basis of their frequency in ascending order.

Step-3: Mark each unique character as a leaf node.

Step-4: Create a new internal node.

Step-5: The frequency of the new node as the sum of the single leaf node

Step-6: Mark the first node as this left child and another node as the right child of the recently created node.

Step-7: Repeat all the steps from step-2 to step-6.

Program:

```
// Huffman Coding in C
#include <stdio.h>
#include <stdlib.h>
#define MAX_TREE_HT 50

struct MinHNode {
  char item;
  unsigned freq;
  struct MinHNode *left, *right;
};

struct MinHeap {
  unsigned size;
  unsigned capacity;
  struct MinHNode **array;
};
// Function declaration
void printArray(int arr[], int n);

// Create nodes
struct MinHNode *newNode(char item, unsigned freq) {
  struct MinHNode *temp = (struct MinHNode *)malloc(sizeof(struct MinHNode));

  temp->left = temp->right = NULL;
```

```
  temp->item = item;
  temp->freq = freq;

  return temp;
}

// Create min heap
struct MinHeap *createMinH(unsigned capacity) {
  struct MinHeap *minHeap = (struct MinHeap *)malloc(sizeof(struct MinHeap));

  minHeap->size = 0;

  minHeap->capacity = capacity;

  minHeap->array = (struct MinHNode **)malloc(minHeap->capacity * sizeof(struct MinHNode
*));
  return minHeap;
}

// Function to swap
void swapMinHNode(struct MinHNode **a, struct MinHNode **b) {
  struct MinHNode *t = *a;
  *a = *b;
  *b = t;
}
```

```
// Heapify
void minHeapify(struct MinHeap *minHeap, int idx) {
  int smallest = idx;
  int left = 2 * idx + 1;
  int right = 2 * idx + 2;

  if (left < minHeap->size && minHeap->array[left]->freq < minHeap->array[smallest]->freq)
    smallest = left;

  if (right < minHeap->size && minHeap->array[right]->freq < minHeap->array[smallest]->freq)
    smallest = right;

  if (smallest != idx) {
    swapMinHNode(&minHeap->array[smallest], &minHeap->array[idx]);
    minHeapify(minHeap, smallest);
  }
}

// Check if size if 1
int checkSizeOne(struct MinHeap *minHeap) {
  return (minHeap->size == 1);
}

// Extract min
struct MinHNode *extractMin(struct MinHeap *minHeap) {
```

```
  struct MinHNode *temp = minHeap->array[0];
  minHeap->array[0] = minHeap->array[minHeap->size - 1];

  --minHeap->size;
  minHeapify(minHeap, 0);

  return temp;
}

// Insertion function
void insertMinHeap(struct MinHeap *minHeap, struct MinHNode *minHeapNode) {
  ++minHeap->size;
  int i = minHeap->size - 1;

  while (i && minHeapNode->freq < minHeap->array[(i - 1) / 2]->freq) {
    minHeap->array[i] = minHeap->array[(i - 1) / 2];
    i = (i - 1) / 2;
  }
  minHeap->array[i] = minHeapNode;
}

void buildMinHeap(struct MinHeap *minHeap) {
  int n = minHeap->size - 1;
  int i;
```

```
  for (i = (n - 1) / 2; i >= 0; --i)
    minHeapify(minHeap, i);
}

int isLeaf(struct MinHNode *root) {
  return !(root->left) && !(root->right);
}

struct MinHeap *createAndBuildMinHeap(char item[], int freq[], int size) {
  struct MinHeap *minHeap = createMinH(size);

  for (int i = 0; i < size; ++i)
    minHeap->array[i] = newNode(item[i], freq[i]);

  minHeap->size = size;
  buildMinHeap(minHeap);

  return minHeap;
}

struct MinHNode *buildHuffmanTree(char item[], int freq[], int size) {
  struct MinHNode *left, *right, *top;
  struct MinHeap *minHeap = createAndBuildMinHeap(item, freq, size);

  while (!checkSizeOne(minHeap)) {
```

```
    left = extractMin(minHeap);
    right = extractMin(minHeap);

    top = newNode('$', left->freq + right->freq);

    top->left = left;
    top->right = right;

    insertMinHeap(minHeap, top);
  }
  return extractMin(minHeap);
}
void printHCodes(struct MinHNode *root, int arr[], int top) {
  if (root->left) {
    arr[top] = 0;
    printHCodes(root->left, arr, top + 1);
  }
  if (root->right) {
    arr[top] = 1;
    printHCodes(root->right, arr, top + 1);
  }
  if (isLeaf(root)) {
    printf("  %c   | ", root->item);
    printArray(arr, top);
  }
```

```
}
// Wrapper function
void HuffmanCodes(char item[], int freq[], int size) {
  struct MinHNode *root = buildHuffmanTree(item, freq, size);

  int arr[MAX_TREE_HT], top = 0;

  printHCodes(root, arr, top);
}
// Print the array
void printArray(int arr[], int n) {
  int i;
  for (i = 0; i < n; ++i)
    printf("%d", arr[i]);

  printf("\n");
}

int main() {
  char arr[] = {'A', 'B', 'C', 'D'};
  int freq[] = {5, 1, 6, 3};

  int size = sizeof(arr) / sizeof(arr[0]);

  printf(" Char | Huffman code ");
```

```
  printf("\n--------------------\n");


  HuffmanCodes(arr, freq, size);
}
```

Output:

Char | Huffman code

C | 0

B | 100

D | 101

A | 11

Explanation:

Input:

- Characters: {'A', 'B', 'C', 'D'}
- Frequencies: {5, 1, 6, 3}

For the above input in the program, the output will be

Char | Huffman code

C | 0

B | 100

D | 101

A | 11

- Huffman coding assigns shorter codes to more frequent characters and longer codes to less frequent characters.
- The tree built will assign the following Huffman codes:
    - Character C (frequency 6) is assigned code 0.

- Character A (frequency 5) is assigned code 10.
- Character D (frequency 3) is assigned code 110.
- Character B (frequency 1) is assigned code 111.

The output shows these characters and their respective Huffman codes.

Viva Questions:

1. What is Huffman Coding?
2. What is a Greedy Algorithm?
3. Why is Huffman Coding considered a Greedy Algorithm?
4. What is the significance of the frequency of characters in Huffman Coding?
5. What data structure is used to implement the Huffman Coding algorithm?
6. How do you generate the Huffman codes from the Huffman Tree?
7. How is the Huffman Tree stored or represented in the program?

RESULT:

Thus, the C Program to implement Huffman coding using Greedy approach is executed successfully and the output is verified.

TOPIC BEYOND THE SYLLABUS

IMPLEMENTATION OF TREE TRAVERSAL ALGORITHMS - DEPTH FIRST SEARCH

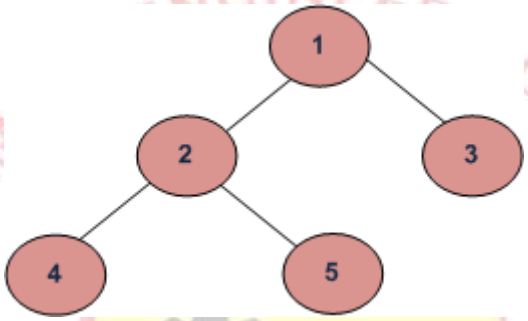
AIM

To write a C program for implementing the Tree traversal algorithm for Depth first traversal.

PRE LAB-DISCUSSION

DFS (Depth-first search) is a technique used for traversing trees or graphs. Here backtracking is used for traversal. In this traversal first, the deepest node is visited and then backtracks to its parent node if no sibling of that node exists.

Example:



Therefore, the Depth First Traversals of this Tree will be:
Inorder: 4 2 5 1 3
Preorder: 1 2 4 5 3
Postorder: 4 5 2 3 1

ALGORITHM

1. Define a structure for a tree node with integer data, and left and right child pointers.
2. Function createNode to allocate memory for a new node, set its data, and initialize its child pointers to NULL.
3. Function insertNode to insert a new node into the binary search tree. If the tree is empty, create a new node. Otherwise, insert the node in the left subtree if the data is less than or equal to the current node's data, otherwise insert it in the right subtree.
4. Read the node values and insert them into the tree.
5. Perform and print preorder, inorder, and postorder traversals.
6. Free the allocated memory for the tree.

PROGRAM

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

// Definition of a tree node
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

// Function to create a new tree node
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}

// Function to insert a node into the tree
struct Node* insertNode(struct Node* root, int data) {
    if (root == NULL) {
        return createNode(data);
    } else {
        if (data <= root->data) {
            root->left = insertNode(root->left, data);
        } else {
            root->right = insertNode(root->right, data);
        }
        return root;
    }
}

// Preorder Traversal
void preorderTraversal(struct Node* node) {
    if (node == NULL)
        return;

    printf("%d ", node->data);
    preorderTraversal(node->left);
```

```
    preorderTraversal(node->right);
}
// Inorder Traversal
void inorderTraversal(struct Node* node) {
    if (node == NULL)
        return;

    inorderTraversal(node->left);
    printf("%d ", node->data);
    inorderTraversal(node->right);
}

// Postorder Traversal
void postorderTraversal(struct Node* node) {
    if (node == NULL)
        return;

    postorderTraversal(node->left);
    postorderTraversal(node->right);
    printf("%d ", node->data);
}

// Function to free memory allocated for the tree nodes
void freeTree(struct Node* root) {
    if (root == NULL)
        return;

    freeTree(root->left);
    freeTree(root->right);
    free(root);
}

int main() {
    struct Node* root = NULL;
    int numNodes, data;

    printf("Enter the number of nodes: ");
    scanf("%d", &numNodes);

    printf("Enter the node values:\n");
```

```
    for (int i = 0; i < numNodes; i++) {
        scanf("%d", &data);
        root = insertNode(root, data);
    }

    printf("\nPreorder traversal: ");
    preorderTraversal(root);
    printf("\n");

    printf("Inorder traversal: ");
    inorderTraversal(root);
    printf("\n");

    printf("Postorder traversal: ");
    postorderTraversal(root);
    printf("\n");

    // Free allocated memory for the tree
    freeTree(root);

    return 0;
}
```

OUTPUT

```
Enter the number of nodes: 5
Enter the node values:
5 3 8 1 4

Preorder traversal: 5 3 1 4 8
Inorder traversal: 1 3 4 5 8
Postorder traversal: 1 4 3 8 5
```

EXPLANATION

Tree Structure:

```
      5
     / \
    3   8
   / \
  1   4
```

Node Insertion:

Insert 5:

Tree is empty. Create root node with data 5.

Insert 3:

3 <= 5, go left.

Left subtree is empty. Create node with data 3.

Insert 8:

8 > 5, go right.

Right subtree is empty. Create node with data 8.

Insert 1:

1 <= 5, go left to node 3.

1 <= 3, go left.

Left subtree of node 3 is empty. Create node with data 1.

Insert 4:

4 <= 5, go left to node 3.

4 > 3, go right.

Right subtree of node 3 is empty. Create node with data 4.

Traversals:

Preorder Traversal:	**Inorder Traversal:**	**Postorder Traversal:**
Visit node 5.	Visit node 1.	Visit node 1.
Visit node 3.	Visit node 3.	Visit node 4.
Visit node 1.	Visit node 4.	Visit node 3.
Visit node 4.	Visit node 5.	Visit node 8.
Visit node 8.	Visit node 8.	Visit node 5.

Memory Freeing: Recursively free memory for nodes 1, 4, 3, 8, and finally 5.

VIVA (PRE & POST LAB) QUESTIONS

1. Explain how DFS can be used to perform topological sorting in a directed acyclic graph (DAG).
2. How does DFS ensure that it goes as deep as possible along each branch before backtracking?
3. Can DFS be used on a binary search tree? How does its behavior differ from using DFS on a general tree?
4. Explain how in-order traversal of a binary search tree (BST) results in a sorted sequence of values.
5. How does DFS differ from Breadth-First Search (BFS)?

RESULT

Thus, the C program for implementing the Tree traversal algorithm for Depth first traversal was implemented successfully.