

# RAJALAKSHMI ENGINEERING COLLEGE

An Autonomous Institution, Affiliated to Anna University

Rajalakshmi Nagar, Thandalam – 602 105



## DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

**CS23231 – DATA STRUCTURES**  
(Regulation 2023)

**LAB MANUAL**

Name : ..... Nithish Balaji .....

Register No. : ..... 231501115 .....

Year / Branch / Section : ..... First, AIML - B .....

Semester : ..... II .....

Academic Year : ..... 2023-2024 .....

## LESSON PLAN

| Course Code | Course Title<br>(Laboratory Integrated Theory Course) | L | T | P | C |
|-------------|---|---|---|---|---|
| CS23231     | Data Structures                                       | 1 | 0 | 6 | 4 |

| LIST OF EXPERIMENTS |  |
|---------------------|--|
| Sl. No              | Name of the experiment   |
| Week 1              | Implementation of Single Linked List (Insertion, Deletion and Display)   |
| Week 2              | Implementation of Doubly Linked List (Insertion, Deletion and Display)   |
| Week 3              | Applications of Singly Linked List (Polynomial Manipulation)             |
| Week 4              | Implementation of Stack using Array and Linked List implementation       |
| Week 5              | Applications of Stack (Infix to Postfix)                                 |
| Week 6              | Applications of Stack (Evaluating Arithmetic Expression)                 |
| Week 7              | Implementation of Queue using Array and Linked List implementation       |
| Week 8              | Implementation of Binary Search Tree                                     |
| Week 9              | Performing Tree Traversal Techniques                                     |
| Week 10             | Implementation of AVL Tree   |
| Week 11             | Performing Topological Sorting   |
| Week 12             | Implementation of BFS, DFS   |
| Week 13             | Implementation of Prim's Algorithm                                       |
| Week 14             | Implementation of Dijkstra's Algorithm                                   |
| Week 15             | Program to perform Sorting   |
| Week 16             | Implementation of Open Addressing (Linear Probing and Quadratic Probing) |
| Week 17             | Implementation of Rehashing  |

## INDEX

| S. No. | Name of the Experiment   | Expt. Date | Faculty Sign |
|--------|--|------------|--------------|
| 1      | Implementation of Single Linked List (Insertion, Deletion and Display) | 2/2/24     |              |
| 2      | Implementation of Doubly Linked List (Insertion, Deletion and Display) | 9/3/24     |              |
| 3      | Applications of Singly Linked List (Polynomial Manipulation)           | 16/3/24    |              |
| 4      | Implementation of Stack using Array and Linked List implementation     | 23/3/24    |              |
| 5      | Applications of Stack (Infix to Postfix)                               | 30/3/24    |              |
| 6      | Applications of Stack (Evaluating Arithmetic Expression)               | 6/4/24     |              |
| 7      | Implementation of Queue using Array and Linked List implementation     | 13/4/24    |              |
| 8      | Performing Tree Traversal Techniques                                   | 20/4/24    |              |
| 9      | Implementation of Binary Search Tree                                   | 27/4/24    |              |
| 10     | Implementation of AVL Tree   | 4/5/24     |              |
| 11     | Implementation of BFS, DFS   | 11/5/24    |              |
| 12     | Performing Topological Sorting   | 11/5/24    |              |
| 13     | Implementation of Prim's Algorithm                                     | 18/5/24    |              |
| 14     | Implementation of Dijkstra's Algorithm                                 | 18/5/24    |              |
| 15     | Program to perform Sorting   | 25/5/24    |              |
| 16     | Implementation of Collision Resolution Techniques                      | 1/6/24     |              |

|              |   |
|--------------|---|
| EXPT NO.: 1  | <b>Implementation of Single Linked List</b><br><b>(Insertion, Deletion and Display)</b> |
| DATE: 2/2/24 |   |

### AIM:

To write a program to implement singly linked list.

### PROGRAM:

```
// Linked List Node
struct node {
    int info;
    struct node* link;
};
struct node* start = NULL;

// Function to create list with n nodes initially
void createList()
{
    if (start == NULL) {
        int n;
        printf("\nEnter the number of nodes: ");
        scanf("%d", &n);
        if (n != 0) {
            int data;
            struct node* newnode;
            struct node* temp;
            newnode = malloc(sizeof(struct node));
            start = newnode;
            temp = start;
            printf("\nEnter number to"
                " be inserted : ");
            scanf("%d", &data);
            start->info = data;

            for (int i = 2; i <= n; i++) {
                newnode = malloc(sizeof(struct node));
                temp->link = newnode;
                printf("\nEnter number to"
                    " be inserted : ");
                scanf("%d", &data);
                newnode->
```

```

        temp = temp->link;
    }
    }
    printf("\nThe list is created\n");
}
else
    printf("\nThe list is already created\n");
}

```

```

// Function to traverse the linked list
void traverse()
{
    struct node* temp;

    // List is empty
    if (start == NULL)
        printf("\nList is empty\n");

    // Else print the LL
    else {
        temp = start;
        while (temp != NULL) {
            printf("Data = %d\n", temp->info);
            temp = temp->link;
        }
    }
}

```

```

// Function to insert at the front
// of the linked list
void insertAtFront()
{
    int data;
    struct node* temp;
    temp = malloc(sizeof(struct node));
    printf("\nEnter number to"
           " be inserted : ");
    scanf("%d", &data);
    temp->info = data;

    // Pointer of temp will be
    // assigned to start
    temp->link = start;
    start = temp;
}

```

```

// Function to insert at the end of
// the linked list
void insertAtEnd()
{
    int data;
    struct node *temp, *head;
    temp = malloc(sizeof(struct node));

    // Enter the number

```

```

printf("\nEnter number to"
      " be inserted : ");
scanf("%d", &data);

// Changes links
temp->link = 0;
temp->info = data;
head = start;
while (head->link != NULL) {
    head = head->link;
}
head->link = temp;
}

// Function to insert at any specified
// position in the linked list
void insertAtPosition()
{
    struct node *temp, *newnode;
    int pos, data, i = 1;
    newnode = malloc(sizeof(struct node));

    // Enter the position and data
    printf("\nEnter position and data :");
    scanf("%d %d", &pos, &data);

    // Change Links
    temp = start;
    newnode->info = data;
    newnode->link = 0;
    while (i < pos - 1) {
        temp = temp->link;
        i++;
    }
    newnode->link = temp->link;
    temp->link = newnode;
}

// Function to delete from the front
// of the linked list
void deleteFirst()
{
    struct node* temp;
    if (start == NULL)
        printf("\nList is empty\n");
    else {
        temp = start;
        start = start->link;
        free(temp);
    }
}

// Function to delete from the end
// of the linked list
void deleteEnd()

```

```

{
    struct node *temp, *prevnode;
    if (start == NULL)
        printf("\nList is Empty\n");
    else {
        temp = start;
        while (temp->link != 0) {
            prevnode = temp;
            temp = temp->link;
        }
        free(temp);
        prevnode->link = 0;
    }
}

// Function to delete from any specified
// position from the linked list
void deletePosition()
{
    struct node *temp, *position;
    int i = 1, pos;

    // If LL is empty
    if (start == NULL)
        printf("\nList is empty\n");

    // Otherwise
    else {
        printf("\nEnter index : ");

        // Position to be deleted
        scanf("%d", &pos);
        position = malloc(sizeof(struct node));
        temp = start;

        // Traverse till position
        while (i < pos - 1) {
            temp = temp->link;
            i++;
        }

        // Change Links
        position = temp->link;
        temp->link = position->link;

        // Free memory
        free(position);
    }
}

// Function to find the maximum element
// in the linked list
void maximum()
{
    int a[10];

```

```

int i;
struct node* temp;

// If LL is empty
if (start == NULL)
    printf("\nList is empty\n");

// Otherwise
else {
    temp = start;
    int max = temp->info;

    // Traverse LL and update the
    // maximum element
    while (temp != NULL) {

        // Update the maximum
        // element
        if (max < temp->info)
            max = temp->info;
        temp = temp->link;
    }
    printf("\nMaximum number "
        "is : %d ",
        max);
}

// Function to find the mean of the
// elements in the linked list
void mean()
{
    int a[10];
    int i;
    struct node* temp;

    // If LL is empty
    if (start == NULL)
        printf("\nList is empty\n");

    // Otherwise
    else {
        temp = start;

        // Stores the sum and count of
        // element in the LL
        int sum = 0, count = 0;
        float m;

        // Traverse the LL
        while (temp != NULL) {

            // Update the sum
            sum = sum + temp->info;

```



```

        temp = temp->link;
        count++;
    }

    // Find the mean
    m = sum / count;

    // Print the mean value
    printf("\nMean is %f ", m);
}

// Function to sort the linked list
// in ascending order
void sort()
{
    struct node* current = start;
    struct node* index = NULL;
    int temp;

    // If LL is empty
    if (start == NULL) {
        return;
    }

    // Else
    else {

        // Traverse the LL
        while (current != NULL) {
            index = current->link;

            // Traverse the LL nestedly
            // and find the minimum
            // element
            while (index != NULL) {

                // Swap with it the value
                // at current
                if (current->info > index->info) {
                    temp = current->info;
                    current->info = index->info;
                    index->info = temp;
                }
                index = index->link;
            }

            // Update the current
            current = current->link;
        }
    }
}

// Function to reverse the linked list

```

```

void reverseLL()
{
    struct node *t1, *t2, *temp;
    t1 = t2 = NULL;

    // If LL is empty
    if (start == NULL)
        printf("List is empty\n");

    // Else
    else {

        // Traverse the LL
        while (start != NULL) {

            // reversing of points
            t2 = start->link;
            start->link = t1;
            t1 = start;
            start = t2;
        }
        start = t1;

        // New head Node
        temp = start;

        printf("Reversed linked "
               "list is : ");

        // Print the LL
        while (temp != NULL) {
            printf("%d ", temp->info);
            temp = temp->link;
        }
    }
}

// Function to search an element in linked list
void search()
{
    int found = -1;
    // creating node to traverse
    struct node* tr = start;

    // first checking if the list is empty or not
    if (start == NULL) {
        printf("Linked list is empty\n");
    }
    else {
        printf("\nEnter the element you want to search: ");
        int key;
        scanf("%d", &key);

        // checking by traversing
    }
}

```

```

while (tr != NULL) {
    // checking for key
    if (tr->info == key) {
        found = 1;
        break;
    }
    // moving forward if not at this position
    else {
        tr = tr->link;
    }
}

// printing found or not
if (found == 1) {
    printf(
        "Yes, %d is present in the linked list.\n",
        key);
}
else {
    printf("No, %d is not present in the linked "
        "list.\n",
        key);
}
}

// Driver Code
int main()
{
    createList();
    int choice;
    while (1) {

        printf("\n\t1 To see list\n");
        printf("\t2 For insertion at "
            " starting\n");
        printf("\t3 For insertion at "
            " end\n");
        printf("\t4 For insertion at "
            "any position\n");
        printf("\t5 For deletion of "
            "first element\n");
        printf("\t6 For deletion of "
            "last element\n");
        printf("\t7 For deletion of "
            "element at any position\n");
        printf("\t8 To find maximum among"
            " the elements\n");
        printf("\t9 To find mean of "
            "the elements\n");
        printf("\t10 To sort element\n");
        printf("\t11 To reverse the "
            "linked list\n");
        printf("\t12 Search an element in linked list\n");
        printf("\t13 To exit\n");
        printf("\nEnter Choice :\n");
    }
}

```

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

switch (choice) {
case 1:
    traverse();
    break;
case 2:
    insertAtFront();
    break;
case 3:
    insertAtEnd();
    break;
case 4:
    insertAtPosition();
    break;
case 5:
    deleteFirst();
    break;
case 6:
    deleteEnd();
    break;
case 7:
    deletePosition();
    break;
case 8:
    maximum();
    break;
case 9:
    mean();
    break;
case 10:
    sort();
    break;
case 11:
    reverseLL();
    break;
case 12:
    search();
    break;
case 13:
    exit(1);
    break;
default:
    printf("Incorrect Choice\n");
}
}
return 0;
}
```

## OUTPUT:

```
1  To see list
2  For insertion at starting
3  For insertion at end
4  For insertion at any position
5  For deletion of first element
6  For deletion of last element
7  For deletion of element at any position
8  To find maximum among the elements
9  To find mean of the elements
10 To sort element
11 To reverse the linked list
12 Search an element in linked list
13 To exit

Enter Choice :
```

## RESULT:

Hence the program has been executed successfully.

|              |   |
|--------------|---|
| EXPT NO.: 2  | <b>Implementation of Doubly Linked List<br/>(Insertion, Deletion and Display)</b> |
| DATE: 9/3/24 |   |

### AIM:

To write a program to implement doubly linked list.

### PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>
int i = 0;

// Node for Doubly Linked List
typedef struct node {
    int key;
    struct node* prev;
    struct node* next;
} node;

// Head, Tail, first & temp Node
node* head = NULL;
node* first = NULL;
node* temp = NULL;
node* tail = NULL;

// Function to add a node in the
// Doubly Linked List
void addnode(int k)
{
```

```

// Allocating memory
// to the Node ptr
node* ptr
    = (node*)malloc(sizeof(node));

// Assign Key to value k
ptr->key = k;

// Next and prev pointer to NULL
ptr->next = NULL;
ptr->prev = NULL;

// If Linked List is empty
if (head == NULL) {
    head = ptr;
    first = head;
    tail = head;
}

// Else insert at the end of the
// Linked List
else {
    temp = ptr;
    first->next = temp;
    temp->prev = first;
    first = temp;
    tail = temp;
}

// Increment for number of Nodes
// in the Doubly Linked List
i++;
}

// Function to traverse the Doubly
// Linked List
void traverse()
{
    // Nodes points towards head node
    node* ptr = head;

    // While pointer is not NULL,
    // traverse and print the node
    while (ptr != NULL) {

        // Print key of the node
        printf("%d ", ptr->key);
        ptr = ptr->next;
    }

    printf("\n");
}

```

```

// Function to insert a node at the
// beginning of the linked list
void insertatbegin(int k)
{

    // Allocating memory
    // to the Node ptr
    node* ptr
        = (node*)malloc(sizeof(node));

    // Assign Key to value k
    ptr->key = k;

    // Next and prev pointer to NULL
    ptr->next = NULL;
    ptr->prev = NULL;

    // If head is NULL
    if (head == NULL) {
        first = ptr;
        first = head;
        tail = head;
    }

    // Else insert at beginning and
    // change the head to current node
    else {
        temp = ptr;
        temp->next = head;
        head->prev = temp;
        head = temp;
    }
    i++;
}

```

```

// Function to insert Node at end
void insertatend(int k)
{

    // Allocating memory
    // to the Node ptr
    node* ptr
        = (node*)malloc(sizeof(node));

    // Assign Key to value k
    ptr->key = k;

    // Next and prev pointer to NULL
    ptr->next = NULL;
    ptr->prev = NULL;

    // If head is NULL
    if (head == NULL) {
        first = ptr;

```



```

        first = head;
        tail = head;
    }

    // Else insert at the end
    else {
        temp = ptr;
        temp->prev = tail;
        tail->next = temp;
        tail = temp;
    }
    i++;
}

// Function to insert Node at any
// position pos
void insertatpos(int k, int pos)
{
    // For Invalid Position
    if (pos < 1 || pos > i + 1) {
        printf("Please enter a"
               " valid position\n");
    }

    // If position is at the front,
    // then call insertatbegin()
    else if (pos == 1) {
        insertatbegin(k);
    }

    // Position is at length of Linked
    // list + 1, then insert at the end
    else if (pos == i + 1) {
        insertatend(k);
    }

    // Else traverse till position pos
    // and insert the Node
    else {
        node* src = head;

        // Move head pointer to pos
        while (pos-- > 0) {
            src = src->next;
        }

        // Allocate memory to new Node
        node **da, **ba;
        node* ptr
            = (node*)malloc(
                sizeof(node));
        ptr->next = NULL;
        ptr->prev = NULL;
        ptr->key = k;
    }
}

```

```

        // Change the previous and next
        // pointer of the nodes inserted
        // with previous and next node
        ba = &src;
        da = &(src->prev);
        ptr->next = (*ba);
        ptr->prev = (*da);
        (*da)->next = ptr;
        (*ba)->prev = ptr;
        i++;
    }
}

```

```

// Function to delete node at the
// beginning of the list
void delatbegin()
{
    // Move head to next and
    // decrease length by 1
    head = head->next;
    i--;
}

```

```

// Function to delete at the end
// of the list
void delatend()
{
    // Move tail to the prev and
    // decrease length by 1
    tail = tail->prev;
    tail->next = NULL;
    i--;
}

```

```

// Function to delete the node at
// a given position pos
void delatpos(int pos)
{
    // If invalid position
    if (pos < 1 || pos > i + 1) {
        printf("Please enter a"
               " valid position\n");
    }

    // If position is 1, then
    // call delatbegin()
    else if (pos == 1) {
        delatbegin();
    }

    // If position is at the end, then
    // call delatend()
    else if (pos == i) {

```

```

        delatend();
    }

    // Else traverse till pos, and
    // delete the node at pos
    else {
        // Src node to find which
        // node to be deleted
        node* src = head;
        pos--;

        // Traverse node till pos
        while (pos--) {
            src = src->next;
        }

        // previous and after node
        // of the src node
        node **pre, **aft;
        pre = &(src->prev);
        aft = &(src->next);

        // Change the next and prev
        // pointer of pre and aft node
        (*pre)->next = (*aft);
        (*aft)->prev = (*pre);

        // Decrease the length of the
        // Linked List
        i--;
    }
}

// Driver Code
int main()
{
    // Adding node to the linked List
    addnode(2);
    addnode(4);
    addnode(9);
    addnode(1);
    addnode(21);
    addnode(22);

    // To print the linked List
    printf("Linked List: ");
    traverse();

    printf("\n");

    // To insert node at the beginning
    insertatbegin(1);
    printf("Linked List after"
           " inserting 1 "
           "at beginning: ");
}

```

```

        traverse();

        // To insert at the end
        insertatend(0);
        printf("Linked List after "
               "deleting node "
               "at position 5: ");
        delatpos(5);
        traverse();

        return 0;
}

```

## OUTPUT:

Linked List: 2 4 9 1 21 22

Linked List after inserting 1 at beginning: 1 2 4 9 1 21 22

Linked List after inserting 0 at end: 1 2 4 9 1 21 22 0

Linked List after inserting 44 after 3rd Node: 1 2 4 44 9 1 21 22 0

Linked List after deleting node at beginning: 2 4 44 9 1 21 22 0

Linked List after deleting node at end: 2 4 44 9 1 21 22

Linked List after deleting node at position 5: 2 4 44 9 21 22

## RESULT:

Hence the program has been executed successfully.

|             |   |
|-------------|---|
| EXPT NO.: 3 | <b>Applications of Singly Linked List<br/>(Polynomial Manipulation)</b> |
| DATE:       |   |

### AIM:

To write a program to implement polynomial manipulation.

### PROGRAM:

```
#include <bits/stdc++.h>
using namespace std;

// Node structure containing powerer
// and coefficient of variable
struct Node {
    int coeff, power;
    Node* next;
};

// Function add a new node at the end of list
Node* addnode(Node* start, int coeff, int power)
{
    // Create a new node
    Node* newnode = new Node;
    newnode->coeff = coeff;
    newnode->power = power;
    newnode->next = NULL;

    // If linked list is empty
    if (start == NULL)
        return newnode;

    // If linked list has nodes
    Node* ptr = start;
```

```

    while (ptr->next != NULL)
        ptr = ptr->next;
    ptr->next = newnode;

    return start;
}

// Function To Display The Linked list
void printList(struct Node* ptr)
{
    while (ptr->next != NULL) {
        cout << ptr->coeff << "x^" << ptr->power ;
        if( ptr->next!=NULL && ptr->next->coeff >=0)
            cout << "+";

        ptr = ptr->next;
    }
    cout << ptr->coeff << "\n";
}

// Function to add coefficients of
// two elements having same power
void removeDuplicates(Node* start)
{
    Node *ptr1, *ptr2, *dup;
    ptr1 = start;

    /* Pick elements one by one */
    while (ptr1 != NULL && ptr1->next != NULL) {
        ptr2 = ptr1;

        // Compare the picked element
        // with rest of the elements
        while (ptr2->next != NULL) {

            // If power of two elements are same
            if (ptr1->power == ptr2->next->power) {

                // Add their coefficients and put it in 1st element
                ptr1->coeff = ptr1->coeff + ptr2->next->coeff;
                dup = ptr2->next;
                ptr2->next = ptr2->next->next;

                // remove the 2nd element
                delete (dup);
            }
            else
                ptr2 = ptr2->next;
        }
        ptr1 = ptr1->next;
    }
}

```

```

    }
}

// Function two Multiply two polynomial Numbers
Node* multiply(Node* poly1, Node* poly2,
              Node* poly3)
{
    // Create two pointer and store the
    // address of 1st and 2nd polynomials
    Node *ptr1, *ptr2;
    ptr1 = poly1;
    ptr2 = poly2;
    while (ptr1 != NULL) {
        while (ptr2 != NULL) {
            int coeff, power;

            // Multiply the coefficient of both
            // polynomials and store it in coeff
            coeff = ptr1->coeff * ptr2->coeff;

            // Add the powerer of both polynomials
            // and store it in power
            power = ptr1->power + ptr2->power;

            // Invoke addnode function to create
            // a newnode by passing three parameters
            poly3 = addnode(poly3, coeff, power);

            // move the pointer of 2nd polynomial
            // two get its next term
            ptr2 = ptr2->next;
        }

        // Move the 2nd pointer to the
        // starting point of 2nd polynomial
        ptr2 = poly2;

        // move the pointer of 1st polynomial
        ptr1 = ptr1->next;
    }

    // this function will be invoke to add
    // the coefficient of the elements
    // having same powerer from the resultant linked list
    removeDuplicates(poly3);
    return poly3;
}

// Driver Code

```

```

int main()
{
    Node *poly1 = NULL, *poly2 = NULL, *poly3 = NULL;

    // Creation of 1st Polynomial:  $3x^2 + 5x^1 + 6$ 
    poly1 = addnode(poly1, 3, 3);
    poly1 = addnode(poly1, 6, 1);
    poly1 = addnode(poly1, -9, 0);

    // Creation of 2nd polynomial:  $6x^1 + 8$ 
    poly2 = addnode(poly2, 9, 3);
    poly2 = addnode(poly2, -8, 2);
    poly2 = addnode(poly2, 7, 1);
    poly2 = addnode(poly2, 2, 0);

    // Displaying 1st polynomial
    cout << "1st Polynomial:- ";
    printList(poly1);

    // Displaying 2nd polynomial
    cout << "2nd Polynomial:- ";
    printList(poly2);

    // calling multiply function
    poly3 = multiply(poly1, poly2, poly3);

    // Displaying Resultant Polynomial
    cout << "Resultant Polynomial:- ";
    printList(poly3);

    return 0;
}

```

### OUTPUT:

```

1st Polynomial:-  $3x^3+6x^1-9$ 
2nd Polynomial:-  $9x^3-8x^2+7x^1+2$ 
Resultant Polynomial:-  $27x^6-24x^5+75x^4-123x^3+114x^2-51x^1-18$ 

```

### RESULT:

Hence the program has been executed successfully.



|               |   |
|---------------|---|
| EXPT NO.: 4   | <b>Implementation of Stack using Array and<br/>Linked List implementation</b> |
| DATE: 23/3/24 |   |

### AIM:

To write a program to implement Stack using Array and Linked List.

### PROGRAM:

#### USING ARRAY:

```
#include<stdio.h>
int stack[100],choice,n,top,x,i;
void push(void);
void pop(void);
void display(void);
int main()
{
top=-1;
printf("\n Enter the size of STACK[MAX=100]:");
scanf("%d",&n);
printf("\n\t STACK OPERATIONS USING ARRAY");
printf("\n\t-----");
printf("\n\t 1.PUSH\n\t 2.POP\n\t 3.DISPLAY\n\t 4.EXIT");
do
{
printf("\n Enter the Choice:");
scanf("%d",&choice);
switch(choice)
{
case 1:
{
push();
```

```

break;
}
case 2:
{
pop();
break;
}
case 3:
{
display();
break;
}
case 4:
{
printf("\n\t EXIT POINT ");
break;
}
default:
{
printf ("\n\t Please Enter a Valid Choice(1/2/3/4)");
}
}
}
while(choice!=4);
return 0;
}
void push()
{
if(top>=n-1)
{
printf("\n\tSTACK is over flow");
}
else
{
printf(" Enter a value to be pushed:");
scanf("%d",&x);
top++;
stack[top]=x;
}
}
void pop()
{
if(top<=-1)
{

printf("\n\t Stack is under flow");
}
else
{
printf("\n\t The popped elements is %d",stack[top]);
}
}
}

```

```

top--;
}
}
void display()
{
if(top>=0)
{
printf("\n The elements in STACK \n");
for(i=top; i>=0; i--)
printf("\n%d",stack[i]);
printf("\n Press Next Choice");
}
else
{
printf("\n The STACK is empty");
}
}

```

#### USING LINKED LIST:

```

#include <stdio.h>
#include <stdlib.h>
struct Node
{
int Data;
struct Node *next;
}*top;
void popStack()
{
struct Node *temp, *var=top;
if(var==top)
{
top = top->next;
free(var);
}
else
printf("\nStack Empty");
}
void push(int value)
{
struct Node *temp;
temp=(struct Node *)malloc(sizeof(struct Node));
temp->Data=value;
if (top == NULL)
{
top=temp;
top->next=NULL;
}

else
{

```

```

temp->next=top;
top=temp;
}
}
void display()
{
struct Node *var=top;
if(var!=NULL)
{
printf("\nElements are as:\n");
while(var!=NULL)
{
printf("\t%d\n",var->Data);
var=var->next;
}
printf("\n");
}
else
printf("\nStack is Empty");
}
int main()
{
int i=0;
top=NULL;
clrscr();
printf(" \n1. Push to stack");
printf(" \n2. Pop from Stack");
printf(" \n3. Display data of Stack");
printf(" \n4. Exit\n");
while(1)
{
printf(" \nChoose Option: ");
scanf("%d",&i);
switch(i)
{
case 1:
{
int value;
printf("\nEnter a value to push into Stack: ");
scanf("%d",&value);
push(value);
break;
}
case 2:

{
popStack();
printf("\n The last element is popped");
break;
}
}
}

```

```
case 3:
{
display();
break;
}
case 4:
{
struct Node *temp;
while(top!=NULL)
{
temp = top->next;
free(top);
top=temp;
}
exit(0);
}
default:
{
printf("\nwrong choice for operation");
}}}}
```

### OUTPUT-1:

Enter the size of stack  
STACK OPERATION USING ARRAY  
1.PUSH  
2.POP  
3.DISPLAY  
4.EXIT  
Enter the choice:1  
Enter a value to be pushed:98

### OUTPUT-2:

1.Push to stack

2. Pop from stack
3. Display data of stack
4. Exit

Choose option 1

Enter a value to push into stack

RESULT:

Hence the program has been executed successfully.

|               |   |
|---------------|---|
| EXPT NO.: 5   | <b>Applications of Stack (Infix to Postfix)</b> |
| DATE: 30/3/24 |   |

### AIM:

To write a program to implement infix to postfix program.

### PROGRAM:

```
#include<stdio.h>
#include<conio.h>
#include<alloc.h>
int top=0,st[20];
char inf[40],post[40];
void postfix();
void push(int);
char pop();
void main()
{
clrscr();
printf("Enter the infix expression:");
scanf("%s",inf);
postfix();
getch();
}
void postfix()
{int i,j=0;
for(i=0;inf[i]!=0;i++)
{switch(inf[i])
{
case '+':while(st[top]>=1)

post[j++]=pop();
push(1);
```

```

break;
case '-':while(st[top]>=1)
post[j++]=pop();
push(2);
break;
case '*':while(st[top]>=3)
post[j++]=pop();
push(3);
break;
case '/':while(st[top]>=4)
post[j++]=pop();
push(4);
break;
case '^':
post[j++]=pop();
push(5);
break;
case '(':push(0);
break;
case ')':while(st[top]!=0)
post[j++]=pop();
top--;
break;
default:
post[j++]=inf[i];
}}
while(top>0)
post[j++]=pop();
printf("\nPostfix expression is =>\n\t\t%s",post);
}void push(int ele)
{
top++;
st[top]=ele;
}char pop()
{int el;
char e;
el=st[top];
top--;
switch(el)
{case 1:
e='+';
break;
case 2:
e='-';

break;
case 3:
e='*';
break;
case 4:

```



```
e=' / ' ;  
break;  
case 5:  
e=' ^ ' ;  
break;  
}return(e);
```

### OUTPUT:

Enter the infix expression:((a+b)\*(c+d)\*(e/f)^g)

Postfix expression is =>

ab+cd+\*ef/\*g^

### RESULT:

Hence the program has been executed successfully.

|              |   |
|--------------|---|
| EXPT NO.: 6  | <b>Applications of Stack (Evaluating Arithmetic Expression)</b> |
| DATE: 6/4/24 |   |

### AIM:

To write a program to Evaluate Arithmetic Expression.

### PROGRAM:

```
#include <stdio.h>
#include <string.h>
int top = -1;
int stack[100];
void push (int data) {
    stack[++top] = data;
}
int pop () {
    int data;
    if (top == -1)
        return -1;
    data = stack[top];
    stack[top] = 0;
    top--;
    return (data);
}
int main()
{
    char str[100];
    int i, data = -1, operand1, operand2, result;
    printf("Enter ur postfix expression:");
    fgets(str, 100, stdin);
    for (i = 0; i < strlen(str); i++)

    {
        if (isdigit(str[i]))
        {
            data = (data == -1) ? 0 : data;
```

```
data = (data * 10) + (str[i] - 48);
continue;
}
if (data != -1)
{
push(data);
}
if (str[i] == '+' || str[i] == '-' || str[i] == '*' || str[i] == '/')
{
operand2 = pop();
operand1 = pop();
if (operand1 == -1 || operand2 == -1)
break;
switch (str[i])
{
case '+':
result = operand1 + operand2;
push(result);
break;
case '-':
result = operand1 - operand2;
push(result);
break;
case '*':
result = operand1 * operand2;
push(result);
break;
case '/':
result = operand1 / operand2;
push(result);
break;
}
}
data = -1;
}
if (top == 0)
printf("The answer is:%d\n", stack[top]);
else
printf("u have given wrong postfix expression\n");
return 0;
}
```

### OUTPUT:

Enter you postfix expression: 10 20 \* 30 40 10 /-+

The answer is: 226

### RESULT:

Hence the program has been executed successfully.

|               |   |
|---------------|---|
| EXPT NO.: 7   | <b>Implementation of Queue using Array and<br/>Linked List implementation</b> |
| DATE: 13/4/24 |   |

### AIM:

To write a program to implement Queue using Array and Linked List.

### PROGRAM:

#### USING ARRAY:

```
#include<stdio.h >
#include<conio.h >
#include<alloc.h >
struct queue
{
int data;
struct queue *next;
};
struct queue *addq(struct queue *front);
struct queue *delq(struct queue *front);
void main()
{
struct queue *front;
int reply,option,data;
clrscr();
front=NULL;
do
{
printf("\n1.addq");
printf("\n2.delq");
printf("\n3.exit");
printf("\nSelect the option");
scanf("%d",&option);
switch(option)
{
case 1 : //addq
front=addq(front);
printf("\n The element is added into the queue");
```

```

break;
case 2 : //delq
front=delq(front);
break;
case 3 : exit(0);
}
}while(1);
}
struct queue *addq(struct queue *front)
{
struct queue *c,*r;
//create new node
c=(struct queue*)malloc(sizeof(struct queue));
if(c==NULL)
{
printf("Insufficient memory");
return(front);
}
//read an insert value from console
printf("\nEnter data");
scanf("%d",&c->data);
c->next=NULL;
if(front==NULL)
{
front=c;
}
else
{
//insert new node after last node
r=front;
while(r->next!=NULL)
{
r=r->next;
}}
return(front);
}
struct queue *delq(struct queue *front)
{
struct queue *c;
if(front==NULL)
{
printf("Queue is empty");
return(front);
}
//print the content of first node
printf("Deleted data:%d",front->data);

//delete first node
c=front;
front=front->next;

```

```
free(c);
return(front);
}
```

#### USING LINKED LIST:

```
#include<stdio.h>
#include<stdlib.h>
#define maxsize 5
void insert();
void delete();
void display();
int front = -1, rear = -1;
int queue[maxsize];
void main ()
{
    int choice;
    while(choice != 4)
    {
        printf("\n***Main Menu***\n");
        printf("\n===== \n");
        printf("\n1.insert an element\n2.Delete an element\n3.Display the
        queue\n4.Exit\n");
        printf("\nEnter your choice ?");
        scanf("%d",&choice);
        switch(choice)
        {
            case 1:
                enqueue();
                break;
            case 2:
                dequeue();
                break;
            case 3:
                display();
                break;
            case 4:
                exit(0);
                break;
            default:
                printf("\nEnter valid choice??\n");
        }
    }
}

void enqueue()
{
    int item;
    printf("\nEnter the element\n");
    scanf("\n%d",&item);
```

```

if(rear == maxsize-1)
{
printf("\nOVERFLOW\n");

return;
}
if(front == -1 && rear == -1)
{
front = 0;
rear = 0;
}
else
{
rear = rear+1;
}
queue[rear] = item;
printf("\nValue inserted ");
}
void dequeue()
{
int item;
if (front == -1 || front > rear)
{
printf("\nUNDERFLOW\n");
return;
}
else
{
item = queue[front];
if(front == rear)
{
front = -1;
rear = -1 ;
}
else
{
front = front + 1;
}
printf("\nvalue deleted ");
}
}
void display()
{
int i;
if(rear == -1)

{
printf("\nEmpty queue\n");
}
else

```



```
{ printf("\nprinting values ..... \n");  
for(i=front;i<=rear;i++)  
{  
printf("\n%d\n",queue[i]);  
}  
}  
}
```

### OUTPUT:

```
1.addq  
2.delq  
3.exit  
Select the option 1  
Enter data 8  
1.addq  
2.delq  
3.exit  
Select the option 1  
Enter data 5
```

\*\*\*Main Menu\*\*\*

```
=====
```

```
1.insert an element  
2.Delete an element  
3.Display the queue  
4.Exit  
Enter your choice ?1  
Enter the element  
123  
Value inserted
```

### RESULT:

Hence the program has been executed successfully.

|               |                       |
|---------------|-----------------------|
| EXPT NO.: 8   | <b>TREE TRAVERSAL</b> |
| DATE: 20/4/24 |                       |

### AIM:

To write a program to implement Tree Traversal.

### PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>
struct node {
int element;
struct node* left;
struct node* right;
};
/*To create a new node*/
struct node* createNode(int val)
{
struct node* Node = (struct node*)malloc(sizeof(struct node));
Node->element = val;
Node->left = NULL;
Node->right = NULL;

return (Node);
}
/*function to traverse the nodes of binary tree in preorder*/
void traversePreorder(struct node* root)
{
if (root == NULL)
return;
printf(" %d ", root->element);
traversePreorder(root->left);
traversePreorder(root->right);
}
```

TREE TRAVERSAL

```

/*function to traverse the nodes of binary tree in Inorder*/
void traverseInorder(struct node* root)
{
    if (root == NULL)
        return;
    traverseInorder(root->left);
    printf(" %d ", root->element);
    traverseInorder(root->right);
}
/*function to traverse the nodes of binary tree in postorder*/
void traversePostorder(struct node* root)
{
    if (root == NULL)
        return;
    traversePostorder(root->left);
    traversePostorder(root->right);
    printf(" %d ", root->element);
}

int main()
{
    struct node* root = createNode(36);
    root->left = createNode(26);
    root->right = createNode(46);
    root->left->left = createNode(21);

    TREE TRAVERSAL
    root->left->right = createNode(31);
    root->left->left->left = createNode(11);
    root->left->left->right = createNode(24);
    root->right->left = createNode(41);
    root->right->right = createNode(56);
    root->right->right->left = createNode(51);
    root->right->right->right = createNode(66);

    printf("\n The Preorder traversal of given binary tree is -\n");
    traversePreorder(root);

    printf("\n The Inorder traversal of given binary tree is -\n");
    traverseInorder(root);

    printf("\n The Postorder traversal of given binary tree is -\n");
    traversePostorder(root);

    return 0;
}

```

**OUTPUT:**

The Preorder traversal of given binary tree is -

36 26 21 11 24 31 46 41 56 51 66

The Inorder traversal of given binary tree is -

11 21 24 26 31 36 41 46 51 56 66

The Postorder traversal of given binary tree is -

11 24 21 31 26 41 51 66 56 46 36

### RESULT:

Hence the program has been executed successfully.

|               |   |
|---------------|---|
| EXPT NO.: 9   | <b>Implementation of Binary Search Tree</b> |
| DATE: 27/4/24 |   |

### AIM:

To write a program to implement BST.

### PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>
struct BinaryTreeNode {
    int key;
    struct BinaryTreeNode *left, *right;
};
struct BinaryTreeNode* newNodeCreate(int value)
{
    struct BinaryTreeNode* temp= (struct
BinaryTreeNode*)malloc(sizeof(struct BinaryTreeNode));
    temp->key = value;
    temp->left = temp->right = NULL;
    return temp;
}
struct BinaryTreeNode*searchNode(struct BinaryTreeNode* root, int target)
{
    if (root == NULL || root->key == target) {
        return root;
    }
    if (root->key < target) {
        return searchNode(root->right, target);
    }
    return searchNode(root->left, target);
}

struct BinaryTreeNode*insertNode(struct BinaryTreeNode* node, int value)
{
    if (node == NULL) {
        return newNodeCreate(value);
    }
    if (value < node->key) {
        node->left = insertNode(node->left, value);
    }
}
```

```

        else if (value > node->key) {
            node->right = insertNode(node->right, value);
        }
        return node;
    }
}

void postOrder(struct BinaryTreeNode* root)
{
    if (root != NULL) {
        postOrder(root->left);
        postOrder(root->right);
        printf(" %d ", root->key);
    }
}

void inOrder(struct BinaryTreeNode* root)
{
    if (root != NULL) {
        inOrder(root->left);
        printf(" %d ", root->key);
        inOrder(root->right);
    }
}

void preOrder(struct BinaryTreeNode* root)
{
    if (root != NULL) {
        printf(" %d ", root->key);
        preOrder(root->left);
        preOrder(root->right);
    }
}

struct BinaryTreeNode* findMin(struct BinaryTreeNode* root)
{
    if (root == NULL) {
        return NULL;
    }
    else if (root->left != NULL) {
        return findMin(root->left);
    }
    return root;
}

struct BinaryTreeNode* delete (struct BinaryTreeNode* root,int x)
{
    if (root == NULL)
        return NULL;
    if (x > root->key) {
        root->right = delete (root->right, x);
    }
    else if (x < root->key) {
        root->left = delete (root->left, x);
    }
    else {
        if (root->left == NULL && root->right == NULL) {
            free(root);
            return NULL;
        }
        else if (root->left == NULL || root->right == NULL) {
            struct BinaryTreeNode* temp;
            if (root->left == NULL) {

```

```

        temp = root->right;
    }
    else {
        temp = root->left;
    }
    free(root);
    return temp;
}
else {
    struct BinaryTreeNode* temp= findMin(root->right);
    root->key = temp->key;
    root->right = delete (root->right, temp->key);
}
}
return root;
}
int main()
{
    struct BinaryTreeNode* root = NULL;
    root = insertNode(root, 50);
    insertNode(root, 30);
    insertNode(root, 20);
    insertNode(root, 40);
    insertNode(root, 70);
    insertNode(root, 60);
    insertNode(root, 80);
    if (searchNode(root, 60) != NULL) {
        printf("60 found");
    }
    else {
        printf("60 not found");
    }
    printf("\n");
    postOrder(root);
    printf("\n");
    preOrder(root);
    printf("\n");
    inOrder(root);
    printf("\n");
    struct BinaryTreeNode* temp = delete (root, 70);
    printf("After Delete: \n");
    inOrder(root);
    return 0;
}

```

**OUTPUT:**

60 found

20 40 30 60 80 70 50

50 30 20 40 70 60 80

20 30 40 50 60 70 80

After Delete:

20 30 40 50 60 80

### RESULT:

Hence the program has been executed successfully.



|              |                                   |
|--------------|-----------------------------------|
| EXPT NO.: 10 | <b>Implementation of AVL Tree</b> |
| DATE: 4/5/24 |                                   |

### AIM:

To write a program to implement AVL tree.

### PROGRAM:

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

// Structure of the tree node
struct node {
    int data;
    struct node* left;
    struct node* right;
    int ht;
};

// Global initialization of root node
struct node* root = NULL;

// Function prototyping
struct node* create(int);
struct node* insert(struct node*, int);
struct node* delete(struct node*, int);
struct node* search(struct node*, int);
struct node* rotate_left(struct node*);
struct node* rotate_right(struct node*);
int balance_factor(struct node*);
int height(struct node*);
void inorder(struct node*);
void preorder(struct node*);
void postorder(struct node*);

int main() {
    int user_choice, data;
    char user_continue = 'y';
    struct node* result = NULL;

    while (user_continue == 'y' || user_continue == 'Y') {
        printf("\n\n----- AVL TREE ----- \n");
```

```

printf("\n1. Insert");
printf("\n2. Delete");
printf("\n3. Search");
printf("\n4. Inorder");
printf("\n5. Preorder");
printf("\n6. Postorder");
printf("\n7. EXIT");

printf("\n\nEnter Your Choice: ");
scanf("%d", &user_choice);

switch(user_choice) {
    case 1:
        printf("\nEnter data: ");
        scanf("%d", &data);
        root = insert(root, data);
        break;

    case 2:
        printf("\nEnter data: ");
        scanf("%d", &data);
        root = delete(root, data);
        break;

    case 3:
        printf("\nEnter data: ");
        scanf("%d", &data);
        result = search(root, data);
        if (result == NULL) {
            printf("\nNode not found!");
        } else {
            printf("\n Node found");
        }
        break;

    case 4:
        inorder(root);
        break;

    case 5:
        preorder(root);
        break;

    case 6:
        postorder(root);
        break;

    case 7:
        printf("\n\tProgram Terminated\n");
        return 1;

    default:
        printf("\n\tInvalid Choice\n");
}

printf("\n\nDo you want to continue? ");
scanf(" %c", &user_continue);

```

```

    }

    return 0;
}

// Creates a new tree node
struct node* create(int data) {
    struct node* new_node = (struct node*) malloc(sizeof(struct node));
    if (new_node == NULL) {
        printf("\nMemory can't be allocated\n");
        return NULL;
    }
    new_node->data = data;
    new_node->left = NULL;
    new_node->right = NULL;
    return new_node;
}

// Rotates to the left
struct node* rotate_left(struct node* root) {
    struct node* right_child = root->right;
    root->right = right_child->left;
    right_child->left = root;

    // Update the heights of the nodes
    root->ht = height(root);
    right_child->ht = height(right_child);

    // Return the new node after rotation
    return right_child;
}

// Rotates to the right
struct node* rotate_right(struct node* root) {
    struct node* left_child = root->left;
    root->left = left_child->right;
    left_child->right = root;

    // Update the heights of the nodes
    root->ht = height(root);
    left_child->ht = height(left_child);

    // Return the new node after rotation
    return left_child;
}

// Calculates the balance factor of a node
int balance_factor(struct node* root) {
    int lh, rh;
    if (root == NULL)
        return 0;
    if (root->left == NULL)
        lh = 0;
    else
        lh = 1 + root->left->ht;
    if (root->right == NULL)
        rh = 0;

```

```

        else
            rh = 1 + root->right->ht;
        return lh - rh;
    }

// Calculate the height of the node
int height(struct node* root) {
    int lh, rh;
    if (root == NULL) {
        return 0;
    }
    if (root->left == NULL) {
        lh = 0;
    } else {
        lh = 1 + root->left->ht;
    }
    if (root->right == NULL) {
        rh = 0;
    } else {
        rh = 1 + root->right->ht;
    }

    if (lh > rh) {
        return lh;
    }
    return rh;
}

// Inserts a new node in the AVL tree
struct node* insert(struct node* root, int data) {
    if (root == NULL) {
        struct node* new_node = create(data);
        if (new_node == NULL) {
            return NULL;
        }
        root = new_node;
    } else if (data > root->data) {
        // Insert the new node to the right
        root->right = insert(root->right, data);

        // Tree is unbalanced, then rotate it
        if (balance_factor(root) == -2) {
            if (data > root->right->data) {
                root = rotate_left(root);
            } else {
                root->right = rotate_right(root->right);
                root = rotate_left(root);
            }
        }
    } else {
        // Insert the new node to the left
        root->left = insert(root->left, data);

        // Tree is unbalanced, then rotate it
        if (balance_factor(root) == 2) {
            if (data < root->left->data) {
                root = rotate_right(root);
            }
        }
    }
}

```

```

        } else {
            root->left = rotate_left(root->left);
            root = rotate_right(root);
        }
    }
}

// Update the heights of the nodes
root->ht = height(root);
return root;
}

// Deletes a node from the AVL tree
struct node* delete(struct node* root, int x) {
    struct node* temp = NULL;

    if (root == NULL) {
        return NULL;
    }

    if (x > root->data) {
        root->right = delete(root->right, x);
        if (balance_factor(root) == 2) {
            if (balance_factor(root->left) >= 0) {
                root = rotate_right(root);
            } else {
                root->left = rotate_left(root->left);
                root = rotate_right(root);
            }
        }
    } else if (x < root->data) {
        root->left = delete(root->left, x);
        if (balance_factor(root) == -2) {
            if (balance_factor(root->right) <= 0) {
                root = rotate_left(root);
            } else {
                root->right = rotate_right(root->right);
                root = rotate_left(root);
            }
        }
    } else {
        if (root->right != NULL) {
            temp = root->right;
            while (temp->left != NULL)
                temp = temp->left;

            root->data = temp->data;
            root->right = delete(root->right, temp->data);
            if (balance_factor(root) == 2) {
                if (balance_factor(root->left) >= 0) {
                    root = rotate_right(root);
                } else {
                    root->left = rotate_left(root->left);
                    root = rotate_right(root);
                }
            }
        }
    } else {

```

```

        return (root->left);
    }
}
root->ht = height(root);
return (root);
}

// Search a node in the AVL tree
struct node* search(struct node* root, int key) {
    if (root == NULL) {
        return NULL;
    }

    if (root->data == key) {
        return root;
    }

    if (key > root->data) {
        search(root->right, key);
    } else {
        search(root->left, key);
    }
}

// Inorder traversal of the tree
void inorder(struct node* root) {
    if (root == NULL) {
        return;
    }

    inorder(root->left);
    printf("%d ", root->data);
    inorder(root->right);
}

// Preorder traversal of the tree
void preorder(struct node* root) {
    if (root == NULL) {
        return;
    }

    printf("%d ", root->data);
    preorder(root->left);
    preorder(root->right);
}

// Postorder traversal of the tree
void postorder(struct node* root) {
    if (root == NULL) {
        return;
    }

    postorder(root->left);
    postorder(root->right);
    printf("%d ", root->data);
}

```

## OUTPUT:

----- AVL TREE -----

1. Insert

2. Delete

3. Search

4. Inorder

5. Preorder

6. Postorder

7. EXIT

Enter Your Choice: 1

Enter data: 5

----- AVL TREE -----

1. Insert

2. Delete

3. Search

4. Inorder

5. Preorder

6. Postorder

7. EXIT

Enter Your Choice: 1

Enter data: 10

## RESULT:

Hence the program has been executed successfully.

|               |                                   |
|---------------|-----------------------------------|
| EXPT NO.: 11  | <b>Implementation of BFS, DFS</b> |
| DATE: 11/5/24 |                                   |

### AIM:

To write a program to implement BFS, DFS.

### PROGRAM:

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

#define MAX 100

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

struct Node* createNode(int v);

struct Graph {
    int numVertices;
    struct Node** adjLists;
    int* visited;
};

struct Graph* createGraph(int vertices);

void addEdge(struct Graph* graph, int src, int dest);

void printGraph(struct Graph* graph);

void BFS(struct Graph* graph, int startVertex);

void DFS(struct Graph* graph, int startVertex);

int main() {
    struct Graph* graph = createGraph(4);
    addEdge(graph, 0, 1);
    addEdge(graph, 0, 2);
    addEdge(graph, 1, 2);
```



```

    addEdge(graph, 2, 0);
    addEdge(graph, 2, 3);
    addEdge(graph, 3, 3);

    printf("Graph:\n");
    printGraph(graph);

    printf("\nBFS Traversal:\n");
    BFS(graph, 2);

    printf("\nDFS Traversal:\n");
    DFS(graph, 2);

    return 0;
}

struct Node* createNode(int v) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->vertex = v;
    newNode->next = NULL;
    return newNode;
}

struct Graph* createGraph(int vertices) {
    struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
    graph->numVertices = vertices;

    graph->adjLists = (struct Node**)malloc(vertices * sizeof(struct
Node*));
    graph->visited = (int*)malloc(vertices * sizeof(int));

    for (int i = 0; i < vertices; i++) {
        graph->adjLists[i] = NULL;
        graph->visited[i] = 0;
    }

    return graph;
}

void addEdge(struct Graph* graph, int src, int dest) {
    struct Node* newNode = createNode(dest);
    newNode->next = graph->adjLists[src];
    graph->adjLists[src] = newNode;

    newNode = createNode(src);
    newNode->next = graph->adjLists[dest];
    graph->adjLists[dest] = newNode;
}

void printGraph(struct Graph* graph) {
    for (int v = 0; v < graph->numVertices; v++) {
        struct Node* temp = graph->adjLists[v];
        printf("Vertex %d: ", v);
        while (temp) {
            printf("%d -> ", temp->vertex);
            temp = temp->next;
        }
    }
}

```

```

        printf("NULL\n");
    }
}

void BFS(struct Graph* graph, int startVertex) {
    struct Node* queue[MAX];
    int front = 0, rear = 0;
    queue[rear] = createNode(startVertex);
    graph->visited[startVertex] = 1;

    printf("Visited %d\n", startVertex);

    while (front <= rear) {
        struct Node* currentNode = queue[front];
        front++;
        while (currentNode) {
            int adjVertex = currentNode->vertex;
            if (!graph->visited[adjVertex]) {
                printf("Visited %d\n", adjVertex);
                queue[++rear] = createNode(adjVertex);
                graph->visited[adjVertex] = 1;
            }
            currentNode = currentNode->next;
        }
    }
}

void DFSUtil(struct Graph* graph, int vertex) {
    struct Node* temp = graph->adjLists[vertex];
    graph->visited[vertex] = 1;
    printf("Visited %d\n", vertex);

    while (temp) {
        int adjVertex = temp->vertex;
        if (!graph->visited[adjVertex]) {
            DFSUtil(graph, adjVertex);
        }
        temp = temp->next;
    }
}

void DFS(struct Graph* graph, int startVertex) {
    graph->visited[startVertex] = 1;
    printf("Visited %d\n", startVertex);

    struct Node* temp = graph->adjLists[startVertex];

    while (temp) {
        int adjVertex = temp->vertex;
        if (!graph->visited[adjVertex]) {
            DFSUtil(graph, adjVertex);
        }
        temp = temp->next;
    }
}

```

## OUTPUT:

Graph:

Vertex 0: 2 -> 1 -> NULL

Vertex 1: 2 -> 0 -> NULL

Vertex 2: 3 -> 0 -> 1 -> NULL

Vertex 3: 3 -> 2 -> NULL

BFS Traversal:

Visited 2

Visited 3

Visited 0

Visited 1

DFS Traversal:

Visited 2

Visited 3

Visited 0

Visited 1

## RESULT:

Hence the program has been executed successfully.

|               |                                       |
|---------------|---------------------------------------|
| EXPT NO.: 12  | <b>PERFORMING TOPOLOGICAL SORTING</b> |
| DATE: 11/5/24 |                                       |

### AIM:

To write a program to implement Topological sorting.

### PROGRAM:

```
#include <stdio.h>

#define MAX_VERTICES 10

int graph[MAX_VERTICES][MAX_VERTICES] = {0};
int visited[MAX_VERTICES] = {0};
int vertices;

void createGraph() {
    int i, j;
    printf("Enter the number of vertices: ");
    scanf("%d", &vertices);
    printf("Enter the adjacency matrix:\n");
    for (i = 0; i < vertices; i++) {
        for (j = 0; j < vertices; j++) {
            scanf("%d", &graph[i][j]);
        }
    }
}

void dfs(int vertex) {
    int i;
    printf("%d ", vertex);
    visited[vertex] = 1;
    for (i = 0; i < vertices; i++) {
        if (graph[vertex][i] && !visited[i]) {
            dfs(i);
        }
    }
}
```

```
int main() {  
    int i;  
    createGraph();  
    printf("Ordering of vertices after DFS traversal:\n");  
    for (i = 0; i < vertices; i++) {  
        if (!visited[i]) {  
            dfs(i);  
        }  
    }  
    return 0;  
}
```

### OUTPUT:

Enter the number of vertices: 4

Enter the adjacency matrix:

0 1 1 0

1 0 0 1

1 0 0 1

0 1 1 0

Ordering of vertices after DFS traversal:

0 1 3 2

### RESULT:

Hence the program has been executed successfully.

|               |   |
|---------------|---|
| EXPT NO.: 13  | <b>Implementation of Prim's Algorithm</b> |
| DATE: 18/5/24 |   |

### AIM:

To write a program to implement Prim's algorithm.

### PROGRAM:

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

#define MAX_VERTICES 10
#define INF 999999

int graph[MAX_VERTICES][MAX_VERTICES];
int vertices;

void createGraph() {
    int i, j;
    printf("Enter the number of vertices: ");
    scanf("%d", &vertices);
    printf("Enter the adjacency matrix:\n");
    for (i = 0; i < vertices; i++) {
        for (j = 0; j < vertices; j++) {
            scanf("%d", &graph[i][j]);
        }
    }
}

int findMinKey(int key[], bool mstSet[]) {
    int min = INF, min_index;
    for (int v = 0; v < vertices; v++) {
        if (mstSet[v] == false && key[v] < min) {
            min = key[v];
            min_index = v;
        }
    }
}
```

```

        }
    }
    return min_index;
}

void printMST(int parent[]) {
    printf("Edge \tWeight\n");
    for (int i = 1; i < vertices; i++) {
        printf("%d - %d \t%d \n", parent[i], i, graph[i][parent[i]]);
    }
}

void primMST() {
    int parent[vertices];
    int key[vertices];
    bool mstSet[vertices];

    for (int i = 0; i < vertices; i++) {
        key[i] = INF;
        mstSet[i] = false;
    }

    key[0] = 0; // Make key 0 so that this vertex is picked as the first
vertex
    parent[0] = -1; // First node is always root of MST

    for (int count = 0; count < vertices - 1; count++) {
        int u = findMinKey(key, mstSet);
        mstSet[u] = true;
        for (int v = 0; v < vertices; v++) {
            if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v]) {
                parent[v] = u;
                key[v] = graph[u][v];
            }
        }
    }
    printMST(parent);
}

int main() {
    createGraph();
    primMST();
    return 0;
}

```

### OUTPUT:

Enter the number of vertices: 5

Enter the adjacency matrix:

0 2 0 6 0

2 0 3 8 5

0 3 0 0 7

6 8 0 0 9

0 5 7 9 0

| Edge | Weight |
|------|--------|
|------|--------|

|       |   |
|-------|---|
| 0 - 1 | 2 |
|-------|---|

|       |   |
|-------|---|
| 1 - 2 | 3 |
|-------|---|

|       |   |
|-------|---|
| 1 - 4 | 5 |
|-------|---|

|       |   |
|-------|---|
| 0 - 3 | 6 |
|-------|---|

### RESULT:

Hence the program has been executed successfully.



|               |   |
|---------------|---|
| EXPT NO.: 14  | <b>Implementation of Dijkstra's Algorithm</b> |
| DATE: 18/5/24 |   |

### AIM:

To write a program to implement Dijkstra's Algorithm.

### PROGRAM:

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

#define MAX_VERTICES 10
#define INF 999999

int graph[MAX_VERTICES][MAX_VERTICES];
int vertices;

void createGraph() {
    int i, j;
    printf("Enter the number of vertices: ");
    scanf("%d", &vertices);
    printf("Enter the adjacency matrix:\n");
    for (i = 0; i < vertices; i++) {
        for (j = 0; j < vertices; j++) {
            scanf("%d", &graph[i][j]);
        }
    }
}

int minDistance(int dist[], bool sptSet[]) {
    int min = INF, min_index;
    for (int v = 0; v < vertices; v++) {
        if (sptSet[v] == false && dist[v] <= min) {
            min = dist[v];
            min_index = v;
        }
    }
}
```

```

    }
    return min_index;
}

void printSolution(int dist[]) {
    printf("Vertex \t Distance from Source\n");
    for (int i = 0; i < vertices; i++) {
        printf("%d \t %d\n", i, dist[i]);
    }
}

void dijkstra(int src) {
    int dist[vertices];
    bool sptSet[vertices];

    for (int i = 0; i < vertices; i++) {
        dist[i] = INF;
        sptSet[i] = false;
    }

    dist[src] = 0;

    for (int count = 0; count < vertices - 1; count++) {
        int u = minDistance(dist, sptSet);
        sptSet[u] = true;

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

    printSolution(dist);
}

int main() {
    createGraph();
    int source;
    printf("Enter the source vertex: ");
    scanf("%d", &source);
    dijkstra(source);
    return 0;
}

```

## OUTPUT:

Enter the number of vertices: 5

Enter the adjacency matrix:

0 10 0 30 100

10 0 50 0 0

0 50 0 20 10

30 0 20 0 60

100 0 10 60 0

Enter the source vertex: 0

| Vertex | Distance from Source |
|--------|----------------------|
|--------|----------------------|

|   |   |
|---|---|
| 0 | 0 |
|---|---|

|   |    |
|---|----|
| 1 | 10 |
|---|----|

|   |    |
|---|----|
| 2 | 50 |
|---|----|

|   |    |
|---|----|
| 3 | 30 |
|---|----|

|   |    |
|---|----|
| 4 | 60 |
|---|----|

## RESULT:

Hence the program has been executed successfully.

|               |                                   |
|---------------|-----------------------------------|
| EXPT NO.: 15  | <b>Program to perform Sorting</b> |
| DATE: 25/5/24 |                                   |

### AIM:

To write a program to implement Sorting.

### PROGRAM:

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

void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = (low - 1);

    for (int j = low; j <= high - 1; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

void merge(int arr[], int l, int m, int r) {
```

```

int i, j, k;
int n1 = m - 1 + 1;
int n2 = r - m;

int L[n1], R[n2];

for (i = 0; i < n1; i++)
    L[i] = arr[l + i];
for (j = 0; j < n2; j++)
    R[j] = arr[m + 1 + j];

i = 0;
j = 0;
k = 1;
while (i < n1 && j < n2) {
    if (L[i] <= R[j]) {
        arr[k] = L[i];
        i++;
    } else {
        arr[k] = R[j];
        j++;
    }
    k++;
}

while (i < n1) {
    arr[k] = L[i];
    i++;
    k++;
}

while (j < n2) {
    arr[k] = R[j];
    j++;
    k++;
}
}

void mergeSort(int arr[], int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;

        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);

        merge(arr, l, m, r);
    }
}

int main() {
    int n;
    printf("Enter the number of elements: ");
    scanf("%d", &n);

    int arr[n];
    printf("Enter %d elements:\n", n);
    for (int i = 0; i < n; i++) {

```

```
        scanf("%d", &arr[i]);
    }

    printf("\nSorting using Quick Sort:\n");
    quickSort(arr, 0, n - 1);
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }

    printf("\n\nSorting using Merge Sort:\n");
    mergeSort(arr, 0, n - 1);
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }

    return 0;
}
```

## OUTPUT:

Enter the number of elements: 5

Enter 5 elements:

4 2 5 1 3

Sorting using Quick Sort:

1 2 3 4 5

Sorting using Merge Sort:

1 2 3 4 5

## RESULT:

Hence the program has been executed successfully.

|              |                |
|--------------|----------------|
| EXPT NO.: 16 | <b>HASHING</b> |
| DATE: 1/6/24 |                |

### AIM:

To write a program to implement Hashing.

### PROGRAM:

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

#define TABLE_SIZE 10

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

Node* createNode(int data) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    if (newNode == NULL) {
        printf("Memory allocation failed!\n");
        exit(1);
    }
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

int hashFunction(int key) {
    return key % TABLE_SIZE;
}

Node* insertOpenAddressing(Node* table[], int key) {
    int index = hashFunction(key);
    while (table[index] != NULL) {
        index = (index + 1) % TABLE_SIZE;
    }
    table[index] = createNode(key);
}
```

```

        return table[index];
    }

void displayHashTable(Node* table[]) {
    printf("Hash Table:\n");
    for (int i = 0; i < TABLE_SIZE; i++) {
        printf("%d: ", i);
        Node* current = table[i];
        while (current != NULL) {
            printf("%d ", current->data);
            current = current->next;
        }
        printf("\n");
    }
}

Node* insertClosedAddressing(Node* table[], int key) {
    int index = hashFunction(key);
    if (table[index] == NULL) {
        table[index] = createNode(key);
    } else {
        Node* newNode = createNode(key);
        newNode->next = table[index];
        table[index] = newNode;
    }
    return table[index];
}

int rehashFunction(int key, int attempt) {
    // Double Hashing Technique
    return (hashFunction(key) + attempt * (7 - (key % 7))) % TABLE_SIZE;
}

Node* insertRehashing(Node* table[], int key) {
    int index = hashFunction(key);
    int attempt = 0;
    while (table[index] != NULL) {
        attempt++;
        index = rehashFunction(key, attempt);
    }
    table[index] = createNode(key);
    return table[index];
}

int main() {
    Node* openAddressingTable[TABLE_SIZE] = {NULL};
    Node* closedAddressingTable[TABLE_SIZE] = {NULL};
    Node* rehashingTable[TABLE_SIZE] = {NULL};

    // Insert elements into hash tables
    insertOpenAddressing(openAddressingTable, 10);
    insertOpenAddressing(openAddressingTable, 20);
    insertOpenAddressing(openAddressingTable, 5);

    insertClosedAddressing(closedAddressingTable, 10);
    insertClosedAddressing(closedAddressingTable, 20);
    insertClosedAddressing(closedAddressingTable, 5);
}

```



```
insertRehashing(rehashingTable, 10);
insertRehashing(rehashingTable, 20);
insertRehashing(rehashingTable, 5);

// Display hash tables
displayHashTable(openAddressingTable);
displayHashTable(closedAddressingTable);
displayHashTable(rehashingTable);
return 0;
}
```

## OUTPUT:

### Hash Table (Open Addressing):

0: 10  
1: 20  
2:  
3:  
4:  
5: 5  
6:  
7:  
8:  
9:

### Hash Table (Closed Addressing):

0: 20 10  
1:  
2:  
3:  
4:  
5: 5  
6:  
7:  
8:  
9:

### Hash Table (Rehashing):

0: 10  
1: 20  
2:  
3:  
4:  
5: 5  
6:  
7:  
8:  
9:

## RESULT:

Hence the program has been executed successfully.