RAJALAKSHMI ENGINEERING COLLEGE

An AUTONOMOUS Institution

Affiliated to ANNA UNIVERSITY, Chennai

25 years OF ACADEMIC EXCELLENCE

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Laboratory Manual

REGULATION 2023

CS23231 – DATA STRUCTURES

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	: M JANANY
Register No.	: 2116231901012
Year / Branch / Section	: I / CSE(CS)
Semester	: II
Academic Year	: 2023-24

LESSON PLAN

Course Code	Course Title (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

Note: Students have to write the Algorithms at left side of each problem statements.

Ex. No.: 1	Implementation of Single Linked List	Date:

Write a C program to implement the following operations on Singly Linked List.

(i) **Insert a node in the beginning of a list.**

(ii) **Insert a node after P**

(iii) **Insert a node at the end of a list**

(iv) **Find an element in a list**

(v) **FindNext**

(vi) **FindPrevious**

(vii) **isLast**

(viii) **isEmpty**

(ix) **Delete a node in the beginning of a list.**

(x) **Delete a node after P**

(xi) **Delete a node at the end of a list**

(xii) **Delete the List**

Algorithm:

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

struct node
{
    int data;
    struct node *link;
}*first=NULL;

void insert_beg(int);
void insert_end(int);
void insert_mid(int,int);
void del_first();
void del_last();
void del_anypos(int);
void display();
void del_all();
void isLast(int);
void isEmpty();
void findnext(int);
void findprev(int);
```

```c
int count();
void search(int);

void insert_beg(int roll)
{
    struct node *newnode;
    newnode=(struct node*)malloc(sizeof(struct node));
    newnode->data=roll;
    if(first==NULL){
        newnode->link=NULL;
        first=newnode;
    }
    else
    {
        newnode->link=first;
        first=newnode;
    }
    printf("Data inserted\n");
}


void insert_end(int roll)
{
    struct node *newnode,*temp;
    temp=first;
    newnode=(struct node*)malloc(sizeof(struct node));
    newnode->data=roll;
    if(first==NULL)
    {
        newnode->link=NULL;
        first=newnode;
    }
    else
    {
        while(temp->link!=0)
        {
            temp=temp->link;
        }
            newnode->link=NULL;
            temp->link=newnode;
            temp=NULL;
    }
    printf("Data Inserted\n");
}

void display()
{
    struct node *temp=NULL;
    temp=first;
    if(temp!=NULL){
        while(temp!=NULL)
    {
```

```
        printf("%d ",temp->data);
        temp=temp->link;
    }
}
else{
    printf("\nNo data inside");
}
}

void insert_mid(int loc,int roll)
{
    struct node *newnode,*temp=NULL;
    temp=first;
    int i=1;
    newnode=(struct node*)malloc(sizeof(struct node));
    newnode->data=roll;
    int t=count();
    if(loc==0)
    insert_beg(roll);
     else if(loc<t)
        {
                while(i<loc)
                {
                        temp=temp->link;
                        i++;

                }
        newnode->link=temp->link;
        temp->link=newnode;
         printf("Data Inserted\n");
        }
        else if(loc==t){
            insert_end(roll);
        }
        else if(loc>t+1){
            printf("Out of bounds");
        }
}

int count(){
    struct node *temp=first;
   int count=0;
   while(temp!=NULL){
    temp=temp->link;
    count++;
    }
    return count;
}

void del_first()
{
    struct node *temp=NULL;
```

```
    temp=first;
    if(first==NULL){
    printf("INVALID OPERATION");
    }
    else{
        first=temp->link;
        free(temp);
        temp=NULL;
    }
    printf("Data deleted\n");
}


void del_last()
{
    struct node *temp=NULL,*temp1=NULL;
    temp=first;
    while(temp->link!=0){
        temp1=temp;
        temp=temp->link;
    }
    free(temp);
    temp=NULL;
    temp1->link=NULL;
    printf("Data Deleted\n");
}


void del_anypos(int pos)
{
    struct node *temp=NULL,*temp1=NULL;
    temp=first;
    if(pos==0)
    {
        del_first();
    }
    else{
        for(int i=1;i<=pos;i++)
        {
            if(temp==NULL)
            {printf("INVALID");
            break;}
            else{
                temp1=temp;
                temp=temp->link;
            }
        }
        if(temp->link!=NULL){
        temp1->link=(temp->link)->link;}
        else{temp1->link=(temp->link);}
        free(temp);
        temp=NULL;
```

```
        temp1=NULL;
       }
       printf("Data Deleted\n");
}

void del_all()
{
    struct node *temp=first,*temp1=NULL;
    while(temp!=NULL){
        temp1=temp;
        temp=temp->link;
        free(temp1);
        first=NULL;
    }
    temp=NULL;temp1=NULL;
    printf("\nAll data deleted successfully");
}

void isEmpty()
{
    if(first==NULL){
        printf("\nThe list is empty\n");
    }
    else{
        printf("\nThe list is not empty\n");
    }
}

void isLast(int pos)
{
    struct node *temp=first;
    int i=1;
    while(i<pos)
    {
        temp=temp->link;
        i++;
    }
if(temp->link == NULL)
  printf("\nIt is the last node");
else
 printf("\nIt is not the last node");
}

void search(int data)
{
    int c=1;
    struct node *temp=first;
    if(first==NULL){
        printf("\nThe list is empty\n");
    }
    else{
    while(temp!=NULL && temp->data!=data){
```

```c
        temp=temp->link;
        c++;
        if(c>count())
        printf("No data in list\n");
    else
    continue;
    }
    printf("\n%d is the position of data\n",c);
}
}

void findnext(int data)
{
    int c=1;
    struct node *temp=first;
    if(first==NULL){
        printf("\nThe list is empty\n");
    }
    else{
    while(temp!=NULL && temp->data!=data){
        temp=temp->link;
        c++;
        if(c>count())
        printf("No data in list\n");
    else
    continue;
    }
    printf("\n%d is the position of data\n",c+1);
}
}

void findprev(int data)
{
       int c=1;
    struct node *temp=first;
    if(first==NULL){
        printf("\nThe list is empty\n");
    }
    else{
    while(temp!=NULL && temp->data!=data){
        temp=temp->link;
        c++;
        if(c>count())
        printf("No data in list\n");
    else
    continue;
    }
    printf("\n%d is the position of data\n",c-1);
}
}

int main()
```

```
{
    int n,ch,pos,t;
    printf("MENU DRIVEN PROGRAM:\n");
    printf("0. Exit\n");
    printf("1. Insert a node at the beginning\n");
    printf("2. Insert a node at the end\n");
    printf("3. Insert a node after P\n");
    printf("4. Search an element\n");
    printf("5. Find next\n");
    printf("6. Find previous\n");
    printf("7. isLast\n");
    printf("8. isEmpty\n");
    printf("9. Delete at beg\n");
    printf("10. Delete after P\n");
    printf("11. Delete at end\n");
    printf("12. Delete list\n");
    printf("13. Display\n");
    while(1){
    printf("\nEnter your choice :  ");
    scanf("%d",&ch);
    switch (ch)
    {
    case 1:
    printf("\nEnter roll to insert at beginning : ");
    scanf("%d",&n);
    insert_beg(n);
    break;

    case 2:
    printf("\nEnter roll to insert at end : ");
    scanf("%d",&n);
    insert_end(n);
    break;

    case 3:
    printf("Enter P : ");
    scanf("%d",&pos);
    printf("\nEnter roll to insert after P : ");
    scanf("%d",&n);
    insert_mid(pos,n);
    break;

    case 4:
    printf("\nEnter data to search : ");
    scanf("%d",&n);
    search(n);
    break;

    case 5:
    printf("\nEnter data to findnext : ");
    scanf("%d",&n);
    findnext(n);
```

```
    break;

    case 6:
    printf("\nEnter data to findprev : ");
    scanf("%d",&n);
    findprev(n);
    break;

    case 7:
    printf("\nEnter position to check last : ");
    scanf("%d",&pos);
    isLast(pos);
    break;

    case 8:
    isEmpty();
    break;

    case 9:
    del_first();
    break;

    case 10:
    printf("\nEnter pos to del after P : ");
    scanf("%d",&pos);
    del_anypos(pos);
    break;

    case 11:
    del_last();
    break;

    case 12:
    del_all();
    break;

    case 13:
    display();
    break;

    default:
        printf("\nMENU EXITED");
        break;
    }
    if(ch==0){
        break;
    }
    else
    continue;
    }
}
```

Ex. No.: 2	Implementation of Doubly Linked List	Date:

Write a C program to implement the following operations on Doubly Linked List.

(i) Insertion

(ii) Deletion

(iii) Search

(iv) Display

Algorithm:

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

void insert_beg(int);
void insert_end(int);
void insert_mid(int,int);
void display();
void del_beg();
void del_end();
void del_mid(int);
void search(int);
int count();

struct node
{
    int data;
    struct node *prev,*next;
}*first=NULL,*last=NULL;

void insert_beg(int roll)
{
    struct node *newnode;
    newnode=(struct node *)malloc(sizeof(struct node));
    newnode->data=roll;
    if(first!=NULL){
        newnode->prev=NULL;
        newnode->next=first;
        first->prev=newnode;
        first=newnode;
    }
    else{
        newnode->prev=NULL;
```

```
        newnode->next=NULL;
        first=newnode;
        last=newnode;
       }
}

void insert_end(int roll)
{
    struct node *newnode;
    newnode=(struct node *)malloc(sizeof(struct node));
    newnode->data=roll;
    if(first==NULL)
    {
        newnode->prev=NULL;
        newnode->next=NULL;
        first=newnode;
        last=newnode;
    }
    else
    {
        newnode->next=NULL;
        newnode->prev=last;
        last->next=newnode;
        last=newnode;
    }
}

void insert_mid(int pos,int roll)
{
    struct node *newnode,*temp=first;
    int c=count();
    newnode=(struct node *)malloc(sizeof(struct node));
    newnode->data=roll;
    if(pos==1)
    {
        insert_beg(roll);
    }
    else if(pos>(c+1)){
        printf("\nOut of bounds\n");
    }
    else if(pos==c+1){
        insert_end(roll);
    }
    else
    {
    for(int i=1;i<pos-1;i++)
    {
        temp=temp->next;
    }
    newnode->next=temp->next;
    newnode->prev=temp;
    if(temp->next!=NULL){
```

```
    (temp->next)->prev=newnode;
    }
    temp->next=newnode;
    }
}

void display()
{
    struct node *temp=NULL;
    temp=first;
    if(temp!=NULL){
        while(temp!=NULL)
    {
        printf("%d ",temp->data);
        temp=temp->next;
    }
}
else{
    printf("\nNo data inside");
}
}

void del_beg()
{
    struct node *temp=first;
    first=temp->next;
    free(temp);
    first->prev=NULL;
    printf("\nDisplay after deleting first node\n");
    display();
}

void del_end()
{
    struct node *temp=first,*temp1=NULL;
    while(temp->next!=NULL){
        temp1=temp;
        temp=temp->next;
    }
    temp1->next=NULL;
    free(temp);
    printf("\nDisplaying after deleting last node\n");
    display();
}

int count()
{
    int count=0;
    struct node *temp=first;
    while(temp!=NULL)
    {
        temp=temp->next;
```

```
        count++;
        }
        return count;
}

void del_mid(int pos)
{
    if(pos==1){
        del_beg();
    }
    struct node *temp=first,*temp1=NULL;
    for(int i=1;i<pos;i++){
        temp1=temp;
        temp=temp->next;
    }
    temp1->next=temp->next;
    (temp->next)->prev=temp1;
    free(temp);
    temp=NULL;
    printf("\nDisplay after deletion : ");
    display();
}

void search(int data)
{
    int c=1;
    struct node *temp=first;
    if(first==NULL){
        printf("\nThe list is empty\n");
    }
    else{
    while(temp!=NULL && temp->data!=data){
        temp=temp->next;
        c++;
    }
    if(c>count()){
    printf("\nNo data in list");
}
else
    printf("\n%d is the position of data\n",c);
}
}

void del_all()
{
    struct node *temp=first,*temp1=NULL;
    while(temp!=NULL){
        temp1=temp;
        temp=temp->next;
        free(temp1);
        first=NULL;
    }
```

```
    temp=NULL;temp1=NULL;
    printf("\nAll data deleted successfully");
}


int main()
{
    int n,ch,pos,t;
    printf("MENU DRIVEN PROGRAM:\n");
    printf("0. Exit\n");
    printf("1. Insert a node at the beginning\n");
    printf("2. Insert a node at the end\n");
    printf("3. Insert a node at any position\n");
    printf("4. Search an element\n");
    printf("5. Delete at beginning \n");
    printf("6. Delete at any position\n");
    printf("7. Delete at end\n");
    printf("8. Delete list\n");
    printf("9. Display\n");
    while(1){
    printf("\nEnter your choice :  ");
    scanf("%d",&ch);
    switch (ch)
    {
    case 1:
    printf("\nEnter roll to insert at beginning : ");
    scanf("%d",&n);
    insert_beg(n);
    break;

    case 2:
    printf("\nEnter roll to insert at end : ");
    scanf("%d",&n);
    insert_end(n);
    break;

    case 3:
    printf("Enter pos to insert : ");
    scanf("%d",&pos);
    printf("\nEnter data to insert after pos : ");
    scanf("%d",&n);
    insert_mid(pos,n);
    break;

    case 4:
    printf("\nEnter data to search : ");
    scanf("%d",&n);
    search(n);
    break;

    case 5:
```

```
    del_beg();
    break;

    case 6:
    printf("\nEnter pos to del : ");
    scanf("%d",&pos);
    del_mid(pos);
    break;

    case 7:
    del_end();
    break;

    case 8:
    del_all();
    break;

    case 9:
    display();
    break;

    default:
        printf("\nMENU EXITED");
        break;
    }
    if(ch==0){
        break;
    }
    else
    continue;
    }
}
```

Ex. No.: 3	Polynomial Manipulation	Date:

Write a C program to implement the following operations on Singly Linked List.

(i) Polynomial Addition

(ii) Polynomial Subtraction

(iii) Polynomial Multiplication

Algorithm:

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

// Define structure for a term in polynomial
struct Term {
    int coefficient;
    int exponent;
    struct Term *next;
};
typedef struct Term Term;

// Function to create a new term
Term *createTerm(int coeff, int exp) {
    Term *newTerm = (Term *)malloc(sizeof(Term));
    if (newTerm == NULL) {
        printf("Memory allocation failed\n");
```

```
        exit(1);
    }
    newTerm->coefficient = coeff;
    newTerm->exponent = exp;
    newTerm->next = NULL;
    return newTerm;
}

// Function to insert a term into the polynomial
void insertTerm(Term **poly, int coeff, int exp) {
    Term *newTerm = createTerm(coeff, exp);
    if (*poly == NULL) {
        *poly = newTerm;
    } else {
        Term *temp = *poly;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newTerm;
    }
}

// Function to display the polynomial
void displayPolynomial(Term *poly) {
    if (poly == NULL) {
        printf("Polynomial is empty\n");
    } else {
        while (poly != NULL) {
            printf("(%dx^%d) ", poly->coefficient, poly->exponent);
            poly = poly->next;
            if (poly != NULL) {
                printf("+ ");
            }
        }
        printf("\n");
    }
}

// Function to add two polynomials
Term *addPolynomials(Term *poly1, Term *poly2) {
    Term *result = NULL;
    while (poly1 != NULL && poly2 != NULL) {
        if (poly1->exponent > poly2->exponent) {
            insertTerm(&result, poly1->coefficient, poly1->exponent);
            poly1 = poly1->next;
        } else if (poly1->exponent < poly2->exponent) {
            insertTerm(&result, poly2->coefficient, poly2->exponent);
            poly2 = poly2->next;
        } else {
            insertTerm(&result, poly1->coefficient + poly2->coefficient, poly1-
>exponent);
            poly1 = poly1->next;
```

```
            poly2 = poly2->next;
        }
    }
    while (poly1 != NULL) {
        insertTerm(&result, poly1->coefficient, poly1->exponent);
        poly1 = poly1->next;
    }
    while (poly2 != NULL) {
        insertTerm(&result, poly2->coefficient, poly2->exponent);
        poly2 = poly2->next;
    }
    return result;
}

// Function to subtract two polynomials
Term *subtractPolynomials(Term *poly1, Term *poly2) {
    Term *result = NULL;
    while (poly1 != NULL && poly2 != NULL) {
        if (poly1->exponent > poly2->exponent) {
            insertTerm(&result, poly1->coefficient, poly1->exponent);
            poly1 = poly1->next;
        } else if (poly1->exponent < poly2->exponent) {
            insertTerm(&result, -poly2->coefficient, poly2->exponent);
            poly2 = poly2->next;
        } else {
            insertTerm(&result, poly1->coefficient - poly2->coefficient, poly1-
>exponent);
            poly1 = poly1->next;
            poly2 = poly2->next;
        }
    }
    while (poly1 != NULL) {
        insertTerm(&result, poly1->coefficient, poly1->exponent);
        poly1 = poly1->next;
    }
    while (poly2 != NULL) {
        insertTerm(&result, -poly2->coefficient, poly2->exponent);
        poly2 = poly2->next;
    }
    return result;
}

// Function to multiply two polynomials
Term *multiplyPolynomials(Term *poly1, Term *poly2) {
    Term *result = NULL;
    Term *temp1 = poly1;
    while (temp1 != NULL) {
        Term *temp2 = poly2;
        while (temp2 != NULL) {
            insertTerm(&result, temp1->coefficient * temp2->coefficient, temp1-
>exponent + temp2->exponent);
            temp2 = temp2->next;
```

```c
        }
        temp1 = temp1->next;
    }
    return result;
}

// Main function
int main() {
    Term *poly1 = NULL;
    Term *poly2 = NULL;

    // Insert terms for polynomial 1
    insertTerm(&poly1, 5, 2);
    insertTerm(&poly1, -3, 1);
    insertTerm(&poly1, 2, 0);

    // Insert terms for polynomial 2
    insertTerm(&poly2, 4, 3);
    insertTerm(&poly2, 2, 1);

    printf("Polynomial 1: ");
    displayPolynomial(poly1);

    printf("Polynomial 2: ");
    displayPolynomial(poly2);

    Term *sum = addPolynomials(poly1, poly2);
    printf("Sum: ");
    displayPolynomial(sum);

    Term *difference = subtractPolynomials(poly1, poly2);
    printf("Difference: ");
    displayPolynomial(difference);

    Term *product = multiplyPolynomials(poly1, poly2);
    printf("Product: ");
    displayPolynomial(product);

    return 0;
}
```

Ex. No.: 4	Implementation of Stack using Array and Linked List Implementation	Date:

Write a C program to implement a stack using Array and linked List implementation and execute the following operation on stack.

(i) **Push an element into a stack**

(ii) **Pop an element from a stack**

(iii) **Return the Top most element from a stack**

(iv) **Display the elements in a stack**

Algorithm:

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

// Structure for node in linked list implementation
struct Node {
    int data;
    struct Node* next;
};

// Structure for stack using linked list implementation
struct StackLL {
    struct Node* top;
};

// Structure for stack using array implementation
struct StackArray {
    int* array;
    int top;
    int capacity;
};

// Function to initialize stack using linked list implementation
struct StackLL* createStackLL() {
    struct StackLL* stack = (struct StackLL*)malloc(sizeof(struct StackLL));
    stack->top = NULL;
    return stack;
}

// Function to initialize stack using array implementation
```

```
struct StackArray* createStackArray(int capacity) {
    struct StackArray* stack = (struct StackArray*)malloc(sizeof(struct
StackArray));
    stack->capacity = capacity;
    stack->top = -1;
    stack->array = (int*)malloc(stack->capacity * sizeof(int));
    return stack;
}

// Function to check if the stack is empty (linked list implementation)
int isEmptyLL(struct StackLL* stack) {
    return stack->top == NULL;
}

// Function to check if the stack is empty (array implementation)
int isEmptyArray(struct StackArray* stack) {
    return stack->top == -1;
}

// Function to push element into stack using linked list implementation
void pushLL(struct StackLL* stack, int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = stack->top;
    stack->top = newNode;
}

// Function to push element into stack using array implementation
void pushArray(struct StackArray* stack, int data) {
    if (stack->top == stack->capacity - 1) {
        printf("Stack Overflow\n");
        return;
    }
    stack->array[++stack->top] = data;
}

// Function to pop element from stack using linked list implementation
int popLL(struct StackLL* stack) {
    if (isEmptyLL(stack)) {
        printf("Stack Underflow\n");
        return -1;
    }
    struct Node* temp = stack->top;
    int data = temp->data;
    stack->top = stack->top->next;
    free(temp);
    return data;
}

// Function to pop element from stack using array implementation
int popArray(struct StackArray* stack) {
```

```
    if (isEmptyArray(stack)) {
        printf("Stack Underflow\n");
        return -1;
    }
    return stack->array[stack->top--];
}

// Function to return top element from stack using linked list implementation
int peekLL(struct StackLL* stack) {
    if (isEmptyLL(stack)) {
        printf("Stack is empty\n");
        return -1;
    }
    return stack->top->data;
}

// Function to return top element from stack using array implementation
int peekArray(struct StackArray* stack) {
    if (isEmptyArray(stack)) {
        printf("Stack is empty\n");
        return -1;
    }
    return stack->array[stack->top];
}

// Function to display elements in stack using linked list implementation
void displayLL(struct StackLL* stack) {
    if (isEmptyLL(stack)) {
        printf("Stack is empty\n");
        return;
    }
    struct Node* temp = stack->top;
    printf("Elements in stack: ");
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

// Function to display elements in stack using array implementation
void displayArray(struct StackArray* stack) {
    if (isEmptyArray(stack)) {
        printf("Stack is empty\n");
        return;
    }
    printf("Elements in stack: ");
    for (int i = stack->top; i >= 0; i--) {
        printf("%d ", stack->array[i]);
    }
    printf("\n");
}
```

```c
int main() {
    // Test linked list implementation
    struct StackLL* stackLL = createStackLL();
    pushLL(stackLL, 1);
    pushLL(stackLL, 2);
    pushLL(stackLL, 3);
    displayLL(stackLL);
    printf("Top element: %d\n", peekLL(stackLL));
    printf("Popped element: %d\n", popLL(stackLL));
    displayLL(stackLL);

    // Test array implementation
    struct StackArray* stackArray = createStackArray(5);
    pushArray(stackArray, 4);
    pushArray(stackArray, 5);
    pushArray(stackArray, 6);
    displayArray(stackArray);
    printf("Top element: %d\n", peekArray(stackArray));
    printf("Popped element: %d\n", popArray(stackArray));
    displayArray(stackArray);

    return 0;
}
```

Ex. No.: 5	Infix to Postfix Conversion	Date:

Write a C program to perform infix to postfix conversion using stack.

Algorithm:

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

#define SIZE 20

int s[SIZE];
int top = -1;
char input[20];
char post[20] = " ";

void push(char ch);
char pop();
int in_precedence(char ch);
int stack_precedence(char ch);
void string_concate(char ch);

int main() {
    int i;

    // Input the string
    printf("Enter the string: ");
    scanf("%s", input);
    printf("string = %s\n", input);

    // Initializing the stack with '#' symbol
    top = 0;
    s[top] = '#';

    // Process the input string character by character
    for (i = 0; input[i] != '\0'; i++) {
        if (input[i] >= 'a' && input[i] <= 'z') {
            // If the character of the input string is operand, then append it to
the postfix string
            string_concate(input[i]);
        } else {
            // If the character of the input string is operator/brackets, then
check the precedence
            // while the precedence of the input operator < precedence of stack
operator, pop and append to postfix string
```

```
            while ((in_precedence(input[i])) < (stack_precedence(s[top]))) {
                char Temp = pop();
                string_concate(Temp);
            }
            // Push the opening bracket and operator onto the stack if the
precedence of input operator is greater than the stack operator
            if (in_precedence(input[i]) != stack_precedence(s[top])) {
                push(input[i]);
            } else {
                pop(); // To pop the opening bracket
            }
        }
    }

    // When we reach the end of the string and if there is anything left in the
stack, pop it and append to the postfix string
    while (s[top] != '#') {
        string_concate(pop());
    }

    printf("Postfix: %s\n", post);

    return 0;
}

int in_precedence(char ch) {
    switch (ch) {
        case '(': return 7;
        case '^': return 6;
        case '*':
        case '/': return 3;
        case '+':
        case '-': return 1;
        case ')': return 0;
        default: return -1;
    }
}

int stack_precedence(char ch) {
    switch (ch) {
        case '(': return 0;
        case '^': return 5;
        case '*':
        case '/': return 4;
        case '+':
        case '-': return 2;
        case '#': return -1;
        default: return -1;
    }
}
```

```c
void push(char ch) {
    if (top == SIZE - 1) {
        printf("Overflow\n");
    } else {
        top = top + 1;
        s[top] = ch;
    }
}

char pop() {
    char c;
    if (top == -1) {
        printf("Underflow\n");
        return -1;
    } else {
        c = s[top];
        top = top - 1;
    }
    return c;
}

void string_concate(char ch) {
    int len = strlen(post);
    post[len] = ch;
    post[len + 1] = '\0'; // Null-terminate the string
    printf("\npost = %s\n", post);
}
```

Ex. No.: 6	Evaluating Arithmetic Expression	Date:

Write a C program to evaluate Arithmetic expression using stack.

Algorithm:

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

#define MAX_SIZE 100

int stack[MAX_SIZE];
int top = -1;

void push(int item) {
    if (top >= MAX_SIZE - 1) {
        printf("Stack Overflow\n");
    } else {
        top++;
        stack[top] = item;
    }
}

int pop() {
    if (top < 0) {
        printf("Stack Underflow\n");
        return -1;
    } else {
        return stack[top--];
    }
}

int evaluateExpression(char* exp) {
    int i, operand1, operand2, result;

    for (i = 0; exp[i] != '\0'; i++) {
        if (isdigit(exp[i])) {
            push(exp[i] - '0');
        } else {
            operand2 = pop();
            operand1 = pop();

            switch (exp[i]) {
                case '+':
                    push(operand1 + operand2);
```

```
                    break;
                case '-':
                    push(operand1 - operand2);
                    break;
                case '*':
                    push(operand1 * operand2);
                    break;
                case '/':
                    if (operand2 == 0) {
                        printf("Error: Division by zero\n");
                        return -1;
                    }
                    push(operand1 / operand2);
                    break;
                default:
                    printf("Error: Unsupported operator %c\n", exp[i]);
                    return -1;
            }
        }
    }

    result = pop();

    if (top >= 0) {
        printf("Error: Invalid postfix expression\n");
        return -1;
    }

    return result;
}

int main() {
    char exp[MAX_SIZE];

    printf("Enter the arithmetic expression: ");
    scanf("%s", exp);

    int result = evaluateExpression(exp);

    if (result != -1) {
        printf("Result: %d\n", result);
    }

    return 0;
}
```

Ex. No.: 7	Implementation of Queue using Array and Linked List Implementation	Date:

Write a C program to implement a Queue using Array and linked List implementation and execute the following operation on stack.

(i) Enqueue

(ii) Dequeue

(iii) Display the elements in a Queue

Algorithm:

```
LINKED LIST IMPLEMENTATION-QUEUE
#include <stdio.h>
#include<stdlib.h>

struct node
{
    int data;
    struct node *link;
}*F=NULL,*R=NULL;

int IsEmpty();
void Enqueue(int);
void Dqueue();
void Display();
int IsEmpty()
{
    if(F==NULL&&R==NULL)
    {
        return 1;
    }
    else
    return 0;
}
void Enqueue(int val)
{
    struct node*newnode;
    newnode=(struct node*)malloc(sizeof(struct node));
    newnode->data=val;
    if(IsEmpty()){
        F=R=newnode;
    }
    else
    {
        R->link=newnode;
```

```
        R=newnode;
    }
    newnode->link=NULL;
}
void Dqueue()
{
    struct node*temp=F;
    if(IsEmpty())
    {
        printf("list is empty");
    }
    else
    {
        printf("\nDeleted element is: %d",temp->data);
        if (F==R)
        F=R=NULL;
        else
        F=F->link;
        free(temp);

    }
}
void Display()
{
    struct node*temp=F;
    if(IsEmpty())
    {
        printf("underflow");
    }
    else
    {
        while(temp!=NULL)
        {
            printf("\n%d",temp->data);
            temp=temp->link;
        }
    }
}
int main()
{
    int choice,t=1,val;
    while (t==1)
    {
        printf("\n\n\nMENU\n");
        printf("1.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:
            printf("Enter the value to be inserted:");
```

```
            scanf("%d",&val);
            Enqueue(val);
            break;
            case 2:
            Dqueue();
            break;
            case 3:
            Display();
            break;
            case 4:
            t=0;

        }
    }
}

ARRAY IMPLEMENTATION-QUEUE
#include<stdio.h>
#include<stdlib.h>
#define size 5
int que[size];
void Enqueue(int);
void Dqueue();
void Display();
int IsFull();
int IsEmpty();
int F=-1,R=-1;
int IsFull()
{
    if (size-1==R)
    {
        return 1;
    }
    else
    return 0;
}
int IsEmpty()
{
    if(F==-1)
    return 1;
    else
    return 0;
}
void Enqueue(int data)
{
    if(IsFull())
    {
        printf("overflow");
    }
    else if(F==-1)
    {
        F=0;
```

```
    }
    R=R+1;
    que[R]=data;
}

void Dqueue()
{
    if(IsEmpty())
    {
        printf("underflow");
    }
    else
    {
       printf("Deleted Element is:%d",que[F]);
       if (R==F)
          R=F=-1;
       else
          F=F+1;
    }
}
void Display()
{
    if(IsEmpty())
    {
        printf("No elements in queue");
    }
    else
    {
        for(int i=F;i<=R;i++)
        {
            printf("%d\n",que[i]);
        }
    }
}
int main()
{
    int choice,t=1,val;
    while (t==1)
    {
        printf("\n\n\nMENU\n");
        printf("1.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:
            printf("Enter the value to be inserted:");
            scanf("%d",&val);
            Enqueue(val);
            break;
            case 2:
```

```
            Dqueue();
            break;
            case 3:
            Display();
            break;
            case 4:
            t=0;

        }
    }
}
```

Write a C program to implement a Binary tree and perform the following tree traversal operation.

(i) Inorder Traversal

(ii) Preorder Traversal

(iii) Postorder Traversal

Algorithm:

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

// Definition of the binary tree node structure
struct node {
    struct node *left;
    int element;
    struct node *right;
};
typedef struct node Node;

// Function declarations
Node *Insert(Node *Tree, int e);
void Inorder(Node *Tree);
void Preorder(Node *Tree);
void Postorder(Node *Tree);

int main() {
    Node *Tree = NULL;
    int n, i, e, ch;

    // Input the number of nodes in the tree
    printf("Enter number of nodes in the tree: ");
    scanf("%d", &n);

    // Input the elements of the tree
    printf("Enter the elements:\n");
    for (i = 1; i <= n; i++) {
        scanf("%d", &e);
        Tree = Insert(Tree, e);
    }

    // Menu for traversal options
    do {
        printf("1. Inorder\n2. Preorder\n3. Postorder\n4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &ch);

        switch (ch) {
            case 1:
```

```
                Inorder(Tree);
                printf("\n");
                break;
            case 2:
                Preorder(Tree);
                printf("\n");
                break;
            case 3:
                Postorder(Tree);
                printf("\n");
                break;
            case 4:
                printf("Exiting...\n");
                break;
            default:
                printf("Invalid choice. Please try again.\n");
        }
    } while (ch != 4);

    return 0;
}

// Function to insert an element into the binary tree
Node *Insert(Node *Tree, int e) {
    Node *NewNode = malloc(sizeof(Node));
    if (Tree == NULL) {
        NewNode->element = e;
        NewNode->left = NULL;
        NewNode->right = NULL;
        Tree = NewNode;
    } else if (e < Tree->element) {
        Tree->left = Insert(Tree->left, e);
    } else if (e > Tree->element) {
        Tree->right = Insert(Tree->right, e);
    }
    return Tree;
}

// Function for inorder traversal
void Inorder(Node *Tree) {
    if (Tree != NULL) {
        Inorder(Tree->left);
        printf("%d\t", Tree->element);
        Inorder(Tree->right);
    }
}

// Function for preorder traversal
void Preorder(Node *Tree) {
    if (Tree != NULL) {
        printf("%d\t", Tree->element);
        Preorder(Tree->left);
```

```
        Preorder(Tree->right);
    }
}

// Function for postorder traversal
void Postorder(Node *Tree) {
    if (Tree != NULL) {
        Postorder(Tree->left);
        Postorder(Tree->right);
        printf("%d\t", Tree->element);
    }
}
```

Write a C program to implement a Binary Search Tree and perform the following operations.

(i) Insert

(ii) Delete

(iii) Search

(iv) Display

Algorithm:

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

// Definition of the binary tree node structure
struct tree {
    int data;
    struct tree *left;
    struct tree *right;
}*root=NULL;

// Function declarations
void insert();
void deleteNode(struct tree *, int);
struct tree *inorder_succ(struct tree *);
void inorder(struct tree *);
void search();

int main() {
    int ans = 1, key;
    struct tree *ptr = NULL;
    int choice;

    do {
        printf("Enter your choice:\n1. Insert\n2. Delete\n3. Display\n4.
Search\n");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                insert();
                break;
            case 2:
                printf("\nEnter the value to be deleted\n");
                scanf("%d", &key);
                ptr = root;
                deleteNode(ptr, key);
                break;
```

```
            case 3:
                ptr = root;
                inorder(ptr);
                break;
            case 4:
                search();
                break;
        }
        printf("\nWant to continue?\nPress 1.YES \t 0.NO\n");
        scanf("%d", &ans);
    } while (ans == 1);

    return 0;
}

void insert() {
    int Flag = 0, key;
    struct tree *parent = NULL, *ptr = root;

    printf("Enter the value to be inserted\n");
    scanf("%d", &key);

    while (ptr != NULL && Flag == 0) {
        if (key < ptr->data) {
            parent = ptr;
            ptr = ptr->left;
        } else if (key > ptr->data) {
            parent = ptr;
            ptr = ptr->right;
        } else if (key == ptr->data) {
            Flag = 1;
        }
    }

    // Creating new node using malloc and setting the data and links of the new
node
    struct tree *newnode = malloc(sizeof(struct tree));
    newnode->left = newnode->right = NULL;
    newnode->data = key;

    if (parent == NULL) {
        root = newnode;
    } else {
        if (key < parent->data)
            parent->left = newnode;
        else
            parent->right = newnode;
    }
}

void inorder(struct tree *ptr) {
```

```c
    if (ptr != NULL) {
        inorder(ptr->left);
        printf("%d -> ", ptr->data);
        inorder(ptr->right);
    }
}

void search() {
    int Flag = 0, key;
    struct tree *parent = NULL, *ptr = root;

    printf("Enter the key to be searched\n");
    scanf("%d", &key);

    while (ptr != NULL && Flag == 0) {
        if (key < ptr->data) {
            parent = ptr;
            ptr = ptr->left;
        } else if (key > ptr->data) {
            parent = ptr;
            ptr = ptr->right;
        } else if (key == ptr->data) {
            Flag = 1;
            printf("%d found\n", ptr->data);
        }
    }

    if (Flag == 0)
        printf("Required Key not found\n");
}

void deleteNode(struct tree *ptr, int key) {
    struct tree *parent = NULL;
    int Flag = 0;

    while (ptr != NULL && Flag == 0) {
        if (key < ptr->data) {
            parent = ptr;
            ptr = ptr->left;
        } else if (key > ptr->data) {
            parent = ptr;
            ptr = ptr->right;
        } else if (key == ptr->data) {
            Flag = 1;
        }
    }

    if (Flag == 0) {
        printf("Required Key does not exist\n");
        return;
    }
```

```
    // If the node to be deleted is a leaf node
    if (ptr->left == NULL && ptr->right == NULL) {
        if (parent == NULL) {
            root = NULL;
        } else if (key < parent->data) {
            parent->left = NULL;
        } else {
            parent->right = NULL;
        }
        free(ptr);
    }
    // If the node to be deleted has one child
    else if (ptr->left == NULL || ptr->right == NULL) {
        if (parent == NULL) {
            if (ptr->right == NULL)
                root = ptr->left;
            else
                root = ptr->right;
        } else if (key < parent->data) {
            if (ptr->left != NULL)
                parent->left = ptr->left;
            else
                parent->left = ptr->right;
        } else {
            if (ptr->left != NULL)
                parent->right = ptr->left;
            else
                parent->right = ptr->right;
        }
        free(ptr);
    }
    // If the node to be deleted has two children
    else {
        struct tree *new_ptr;
        new_ptr = inorder_succ(ptr->right);
        int save = new_ptr->data;
        deleteNode(ptr, new_ptr->data);
        ptr->data = save;
    }
}

struct tree *inorder_succ(struct tree *pt) {
    while (pt->left != NULL) {
        pt = pt->left;
    }
    return pt;
}
```

Ex. No.: 10	Implementation of AVL Tree	Date:

Write a function in C program to insert a new node with a given value into an AVL tree. Ensure that the tree remains balanced after insertion by performing rotations if necessary. Repeat the above operation to delete a node from AVL tree.

Algorithm:

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

typedef struct Node {
    int key;
    struct Node* left;
    struct Node* right;
    int height;
} Node;

int height(Node* node) {
    if (node == NULL)
        return 0;
    return node->height;
}

int max(int a, int b) {
    return (a > b) ? a : b;
}

Node* newNode(int key) {
    Node* node = (Node*)malloc(sizeof(Node));
    node->key = key;
    node->left = NULL;
    node->right = NULL;
    node->height = 1;
    return node;
}

Node* rightRotate(Node* y) {
    Node* x = y->left;
    Node* T2 = x->right;

    x->right = y;
    y->left = T2;
```

```
    y->height = max(height(y->left), height(y->right)) + 1;
    x->height = max(height(x->left), height(x->right)) + 1;


    return x;
}


Node* leftRotate(Node* x) {
    Node* y = x->right;
    Node* T2 = y->left;


    y->left = x;
    x->right = T2;


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


    return y;
}


int getBalance(Node* N) {
    if (N == NULL)
        return 0;
    return height(N->left) - height(N->right);
}


Node* insert(Node* node, int key) {

    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
        return node;


    node->height = 1 + max(height(node->left), height(node->right));


    int balance = getBalance(node);
```

```
    if (balance > 1 && key < node->left->key)
        return rightRotate(node);


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


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


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


    return node;
}


Node* deleteNode(Node* root, int key) {

    if (root == NULL)
        return root;


    if (key < root->key)
        root->left = deleteNode(root->left, key);


    else if (key > root->key)
        root->right = deleteNode(root->right, key);


    else {

        if ((root->left == NULL) || (root->right == NULL)) {
            Node* temp = root->left ? root->left : root->right;


            if (temp == NULL) {
                temp = root;
                root = NULL;
            } else
                *root = *temp;

            free(temp);
```

```
        } else {

            Node* temp = root->right;
            while (temp->left != NULL)
                temp = temp->left;


            root->key = temp->key;

            root->right = deleteNode(root->right, temp->key);
        }
    }


    if (root == NULL)
        return root;


    root->height = 1 + max(height(root->left), height(root->right));


    int balance = getBalance(root);


    if (balance > 1 && getBalance(root->left) >= 0)
        return rightRotate(root);


    if (balance > 1 && getBalance(root->left) < 0) {
        root->left = leftRotate(root->left);
        return rightRotate(root);
    }


    if (balance < -1 && getBalance(root->right) <= 0)
        return leftRotate(root);


    if (balance < -1 && getBalance(root->right) > 0) {
        root->right = rightRotate(root->right);
        return leftRotate(root);
    }

    return root;
}


void preOrder(Node* root) {
    if (root != NULL) {
        printf("%d ", root->key);
        preOrder(root->left);
```

```
        preOrder(root->right);
    }
}


int main() {
    Node* root = NULL;
    int key;
    int n,value;
    printf("Enter number of nodes to be inserted:");
    scanf("%d",&n);
    for (int i=0;i<n;i++){
        printf("Enter data: ");
        scanf("%d",&value);
        root=insert(root,value);
    }

    printf("Preorder traversal of the AVL tree after insertion: ");
    preOrder(root);
    printf("\n");

    printf("enter key to delete: ");
    scanf("%d",&key);
    root = deleteNode(root,key);

    printf("Preorder traversal of the AVL tree after deletion of node with key %d:
",key);
    preOrder(root);
    printf("\n");

    return 0;
}
```

Ex. No.: 11	Graph Traversal	Date:

Write a C program to create a graph and perform a Breadth First Search and Depth First Search.

Algorithm:

```c
#include<stdio.h>
#include<stdlib.h>
#define size 7
int s[size];
int top=-1;
int pop();
void push(int);
int queue[size];
int front = -1, rear = -1;
void dfs();
void bfs();

int isEmpty() { return front == -1 && rear == -1; }

int isFull() { return rear == size - 1; }

void enqueue(int val) {
    if (!isFull()) {
        if (isEmpty()) {
            front = rear = 0;
        } else {
            rear = (rear + 1) % size;
        }
        queue[rear] = val;
    } else {
        printf("\nQUEUE IS FULL!\n");
    }
}

int dequeue() {
    if (!isEmpty()) {
        int val = queue[front];
        if (front == rear) {
            front = rear = -1;
        } else {
            front = (front + 1) % size;
        }
```

```
        return val;
    } else {
        printf("\nQUEUE IS EMPTY!\n");
        return -1;
    }
}

void dfs(){
    int
g[size][size]={{0,1,1,0,0,0,0},{0,0,0,0,0,0,0},{0,0,0,1,0,1,0},{1,1,0,0,0,0,1},{0,1
,0,0,0,0,0},{0,0,0,0,0,0,1},{0,0,0,0,1,0,0}};
    int visited[size]={0};
    int j,i=0;

   printf("DFS :  ");
    while(i>-1 && i<size)
    {
        if(visited[i]!=1)
        {
        printf("%d->",i);
        visited[i]=1;
        }
        for(i,j=0;j<size;j++)
        {
            if(g[i][j]==1 && visited[j]!=1){
            push(j);
            }
        }
        i=pop();
    }

}

void bfs(){
    int g[size][size] = {
        {0, 1, 1, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 1, 0, 1, 0},
        {1, 1, 0, 0, 0, 0, 1},
        {0, 1, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 1},
        {0, 0, 0, 0, 1, 0, 0}
    };
    int visited[size]={0};

    int i = 0;
    printf("BFS : ");
    visited[i] = 1;
    printf("%d->", i);

    enqueue(i);
```

```
    while (!isEmpty()) {
        int i = dequeue();

        for (int j = 0; j < size; j++) {
            if (g[i][j] && !visited[j]) {
                visited[j] = 1;
                printf("%d->", j);
                enqueue(j);
            }
        }
    }

}


void push(int data)
{
        top=top+1;
        s[top]=data;

}

int pop()
{
        int temp;
        temp=s[top];
        top=top-1;
        return temp;

}

int main()
{
    int ch,ans=1;
    do{
    printf("enter your choice \n1.DFS\n2.BFS\n");
    scanf("%d",&ch);
    switch(ch)
    {
        case 1:
        dfs();
        break;

        case 2:
        bfs();
        break;
    }
    printf("\nWant to continue ?\n1.yes \n0.no\n");
    scanf("%d",&ans);
    }
```

```
    while(ans==1);
}
```

Ex. No.: 12	Topological Sorting	Date:

Write a C program to create a graph and display the ordering of vertices.

Algorithm:

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

int s[100], j, res[100]; /*GLOBAL VARIABLES */

void AdjacencyMatrix(int a[][100], int n) { //To generate adjacency matrix for
given nodes

    int i, j;
    for (i = 0; i < n; i++) {
        for (j = 0; j <= n; j++) {
            a[i][j] = 0;
        }
    }
    for (i = 1; i < n; i++) {
        for (j = 0; j < i; j++) {
            a[i][j] = rand() % 2;
            a[j][i] = 0;
        }
    }
}

void dfs(int u, int n, int a[][100]) { /* DFS */

    int v;
    s[u] = 1;
    for (v = 0; v < n - 1; v++) {
        if (a[u][v] == 1 && s[v] == 0) {
            dfs(v, n, a);
        }
    }
    j += 1;
    res[j] = u;
}

void topological_order(int n, int a[][100]) { /* TO FIND TOPOLOGICAL ORDER*/

    int i, u;
```

```
    for (i = 0; i < n; i++) {
        s[i] = 0;
    }
    j = 0;
    for (u = 0; u < n; u++) {
        if (s[u] == 0) {
            dfs(u, n, a);
        }
    }
    return;
}
int main() {
    int a[100][100], n, i, j;

    printf("Enter number of vertices\n"); /* READ NUMBER OF VERTICES */
    scanf("%d", &n);

    AdjacencyMatrix(a, n); /*GENERATE ADJACENCY MATRIX */

    printf("\t\tAdjacency Matrix of the graph\n"); /* PRINT ADJACENCY MATRIX */
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            printf("\t%d", a[i][j]);
        }
        printf("\n");
    }
    printf("\nTopological order:\n");

    topological_order(n, a);

    for (i = n; i >= 1; i--) {
        printf("-->%d", res[i]);
    }
    return 0;
}
```

Ex. No.: 13	Graph Traversal	Date:

Write a C program to create a graph and find a minimum spanning tree using prims algorithm.

Algorithm:

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

#define MAX_VERTICES 100

// Function to find the vertex with the minimum key value
int minKey(int key[], int mstSet[], int vertices) {
    int min = INT_MAX, minIndex;

    for (int v = 0; v < vertices; v++) {
        if (!mstSet[v] && key[v] < min) {
            min = key[v];
            minIndex = v;
        }
    }

    return minIndex;
}

// Function to print the constructed MST stored in parent[]
void printMST(int parent[], int graph[MAX_VERTICES][MAX_VERTICES], int vertices) {
    printf("Edge \tWeight\n");
    for (int i = 1; i < vertices; i++) {
        printf("%d - %d \t%d\n", parent[i], i, graph[i][parent[i]]);
    }
}

// Function to implement Prim's algorithm for a given graph
void primMST(int graph[MAX_VERTICES][MAX_VERTICES], int vertices) {
    int parent[MAX_VERTICES]; // Array to store the constructed MST
    int key[MAX_VERTICES];    // Key values used to pick the minimum weight edge
    int mstSet[MAX_VERTICES]; // To represent set of vertices included in MST

    // Initialize all keys as INFINITE and mstSet[] as false
    for (int i = 0; i < vertices; i++) {
        key[i] = INT_MAX;
        mstSet[i] = 0;
```

```
    }

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

    // The MST will have vertices-1 edges
    for (int count = 0; count < vertices - 1; count++) {
        // Pick the minimum key vertex from the set of vertices not yet included in
the MST
        int u = minKey(key, mstSet, vertices);

        // Add the picked vertex to the MST Set
        mstSet[u] = 1;

        // Update key value and parent index of the adjacent vertices
        for (int v = 0; v < vertices; v++) {
            // graph[u][v] is non-zero only for adjacent vertices of m
            // mstSet[v] is false for vertices not yet included in MST
            // Update the key only if the graph[u][v] is smaller than the key[v]
            if (graph[u][v] && !mstSet[v] && graph[u][v] < key[v]) {
                parent[v] = u;
                key[v] = graph[u][v];
            }
        }
    }

    // Print the constructed MST
    printMST(parent, graph, vertices);
}

int main() {
    int vertices;

    // Input the number of vertices
    printf("Input the number of vertices: ");
    scanf("%d", &vertices);

    if (vertices <= 0 || vertices > MAX_VERTICES) {
        printf("Invalid number of vertices. Exiting...\n");
        return 1;
    }

    int graph[MAX_VERTICES][MAX_VERTICES];

    // Input the adjacency matrix representing the graph
    printf("Input the adjacency matrix for the graph:\n");
    for (int i = 0; i < vertices; i++) {
        for (int j = 0; j < vertices; j++) {
            scanf("%d", &graph[i][j]);
        }
```

```
    }

    // Perform Prim's algorithm to find the MST
    primMST(graph, vertices);

    return 0;
}
```

Ex. No.: 14	Graph Traversal	Date:

Write a C program to create a graph and find the shortest path using Dijikstra's Algorithm.

Algorithm:

```
#include <stdio.h>
#define size 8
#define INFINITY 10000000;
int g[size][size]={ {0,2,6,0,0,0,0,0},
                    {2,0,0,2,6,0,0,0},
                    {6,0,0,1,0,0,4,0},
                    {0,2,1,0,0,2,0,0},
                    {0,6,0,0,0,3,0,1},
                    {0,0,0,2,3,0,2,0},
                    {0,0,0,2,0,2,0,2},
                    {0,0,0,0,1,0,2,0} };

struct vertex_info
{
    int length;
    int pred;
    char state;
}v[size];

int main()
{
        int i;
    for (i=0;i<size;i++)
        {
                v[i].length=INFINITY;
                v[i].pred=-1;
                v[i].state='N';
        }
        int s=0;
        int d=7;
        v[s].length=0;
        v[s].state='V';

     do
        {
                int i;
        for(i=0;i<size;i++)
                {
                    if (g[s][i]!=0 &&v[i].state=='N')
                    {
```

```
                if(v[i].length>v[s].length+g[s][i])
                        {
                    v[i].length=g[s][i]+v[s].length;
                                       v[i].pred=s;

printf("\nlength[%d]=%d\tpred[%d]=%d",i,v[i].length,i,v[i].pred);
                        }
                    }
            }
        int min=INFINITY;
            s=0;
                for(i=0;i<size;i++)
            {
                    if(v[i].state=='N'&& v[i].length<min)
                    {
                        min=v[i].length;
                        s=i;
                    }
                 }
            v[s].state='V';
            }while(s!=d);
    i=size;
    int path[size];
    printf("\n\nPath=%d->",s);
    do
    {
                path[i--]=s;
          s=v[s].pred;
              printf("%d->",s);
    }while(s>0);

}
```

Write a C program to take n numbers and sort the numbers in ascending order. Try to implement the same using following sorting techniques.

1. **Quick Sort**
2. **Merge Sort**

Algorithm:

```
Quick Sort


#include <stdio.h>

void QuickSort(int a[], int left, int right);

int main() {
    int i, n, a[10];
    printf("Enter the limit: ");
    scanf("%d", &n);
    printf("Enter the elements: ");
    for (i = 0; i < n; i++) {
        scanf("%d", &a[i]);
    }
    QuickSort(a, 0, n - 1);
    printf("The sorted elements are: ");
    for (i = 0; i < n; i++) {
        printf("%d\t", a[i]);
    }
    return 0;
}

void QuickSort(int a[], int left, int right) {
    int i, j, temp, pivot;
    if (left < right) {
        pivot = left;
        i = left + 1;
        j = right;
        while (i <= j) { // Change here to i <= j instead of i < j
            while (i <= right && a[i] < a[pivot]) i++; // Add boundary check
            while (j >= left && a[j] > a[pivot]) j--; // Add boundary check
            if (i < j) {
                temp = a[i];
                a[i] = a[j];
                a[j] = temp;
                i++; // Move pointers after swapping
                j--;
            } else if (i == j) {
                i++;
```

```
            }
        }
        temp = a[pivot];
        a[pivot] = a[j];
        a[j] = temp;
        QuickSort(a, left, j - 1);
        QuickSort(a, j + 1, right);
    }
}
```

MERG SORT

```
#include <stdio.h>

void MergeSort(int arr[], int left, int right);
void Merge(int arr[], int left, int center, int right);

int main() {
    int i, n, arr[20];
    printf("Enter the limit: ");
    scanf("%d", &n);
    printf("Enter the elements: ");
    for (i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }
    MergeSort(arr, 0, n - 1);
    printf("The sorted elements are: ");
    for (i = 0; i < n; i++) {
        printf("%d\t", arr[i]);
    }
    return 0;
}

void MergeSort(int arr[], int left, int right) {
    int center;
    if (left < right) {
        center = (left + right) / 2;
        MergeSort(arr, left, center);
        MergeSort(arr, center + 1, right);
        Merge(arr, left, center, right);
    }
}

void Merge(int arr[], int left, int center, int right) {
    int a[20], b[20], n1, n2, aptr, bptr, cptr, i, j;
```

```
    n1 = center - left + 1;
    n2 = right - center;

    for (i = 0; i < n1; i++) {
        a[i] = arr[left + i];
    }
    for (j = 0; j < n2; j++) {
        b[j] = arr[center + 1 + j];
    }

    aptr = 0;
    bptr = 0;
    cptr = left;

    while (aptr < n1 && bptr < n2) {
        if (a[aptr] <= b[bptr]) {
            arr[cptr] = a[aptr];
            aptr++;
        } else {
            arr[cptr] = b[bptr];
            bptr++;
        }
        cptr++;
    }

    while (aptr < n1) {
        arr[cptr] = a[aptr];
        aptr++;
        cptr++;
    }

    while (bptr < n2) {
        arr[cptr] = b[bptr];
        bptr++;
        cptr++;
    }
}
```

Write a C program to create a hash table and perform collision resolution using the following techniques.

(i) Open addressing

(ii) Closed Addressing

(iii) Rehashing

Algorithm:

```
#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;
}
```

Rajalakshmi™ Institutions

Rajalakshmi Engineering College
Rajalakshmi Nagar Thandalam, Chennai - 602 105.
Phone : +91-44-67181111, 67181112
Website : www.rajalakshmi.org