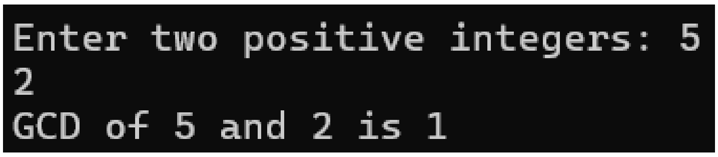
PRACTICAL-1

QUES: *Write a program to find the GCD of two Number using recursion.*

CODE:

```
#include <stdio.h>

int gcd(int a, int b){
                        if (b == 0)
                        return a;
                        return gcd(b, a % b);
                }

int main(){
        int num1, num2;
        printf("Enter two positive integers: ");
        scanf("%d %d", &num1, &num2);
        printf("GCD of %d and %d is %d\n", num1, num2, gcd(num1, num2));
        return 0;
                }
```

OUTPUT:



```
Enter two positive integers: 5
2
GCD of 5 and 2 is 1
```

PRACTICAL-2

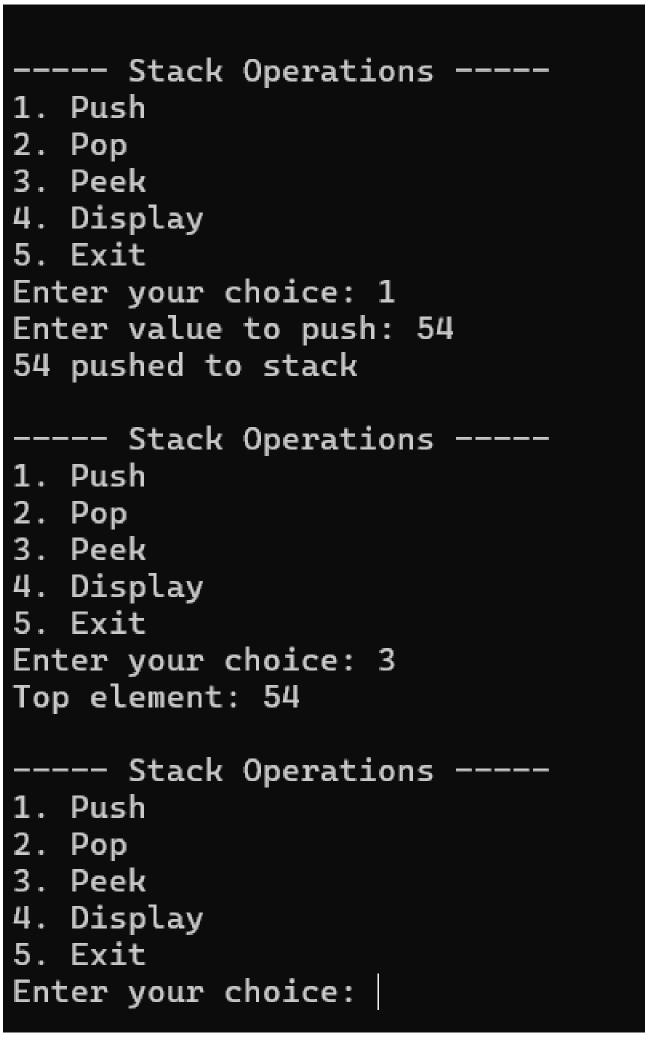
QUES: *Write a program to implement stacks using array.*

CODE:

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

#define MAX_SIZE 100

struct Stack{
    int arr[MAX_SIZE];
    int top;
        };

void initialize(struct Stack *stack){
    stack->top = -1;
                }

bool isEmpty(struct Stack *stack){
    return (stack->top == -1);
                    }

bool isFull(struct Stack *stack){
    return (stack->top == MAX_SIZE - 1);
                        }

void push(struct Stack *stack, int value{
    if (isFull(stack)){
        printf("Stack Overflow! Cannot push %d\n", value);
            return;
                }

    stack->arr[++(stack->top)] = value;
    printf("%d pushed to stack\n", value);
                            }

int pop(struct Stack *stack){
    if (isEmpty(stack)){
        printf("Stack Underflow! Cannot pop from empty stack\n");
        return -1;
                }

    return stack->arr[(stack->top)--];
                        }
```

```
int peek(struct Stack *stack){
    if (isEmpty(stack)){
        printf("Stack is empty\n");
        return -1;
              }

return stack->arr[stack->top];
                 }

void display(struct Stack *stack){
    if (isEmpty(stack)){
        printf("Stack is empty\n");
        return;
            }

printf("Stack elements: ");
for (int i = stack->top; i >= 0; i--){
    printf("%d ", stack->arr[i]);
                   }
printf("\n");
       }

int main(){
    struct Stack stack;
    initialize(&stack);
    int choice, value;

    do{
      printf("\n----- Stack Operations -----\n");
      printf("1. Push\n");
      printf("2. Pop\n");
      printf("3. Peek\n");
      printf("4. Display\n");
      printf("5. Exit\n");
      printf("Enter your choice: ");
      scanf("%d", &choice);

    switch (choice){
        case 1:
            printf("Enter value to push: ");
            scanf("%d", &value);
            push(&stack, value);
            break;

        case 2:
            value = pop(&stack);
            if (value != -1){
                printf("Popped value: %d\n", value);
                                    }
        break;
        case 3:
            value = peek(&stack);
        if (value != -1){
            printf("Top element: %d\n", value);
                            }
        break;

        case 4:
            display(&stack);
        break;

        case 5:
            printf("Exiting program\n");
        break;

        default:
            printf("Invalid choice. Please try again.\n"); } }
        while (choice != 5);
        return 0; }
```

OUTPUT:



```
----- Stack Operations -----
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 1
Enter value to push: 54
54 pushed to stack

----- Stack Operations -----
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 3
Top element: 54

----- Stack Operations -----
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 
```

PRACTICAL-2

QUES: *Write a program to implement queues using arrays.*

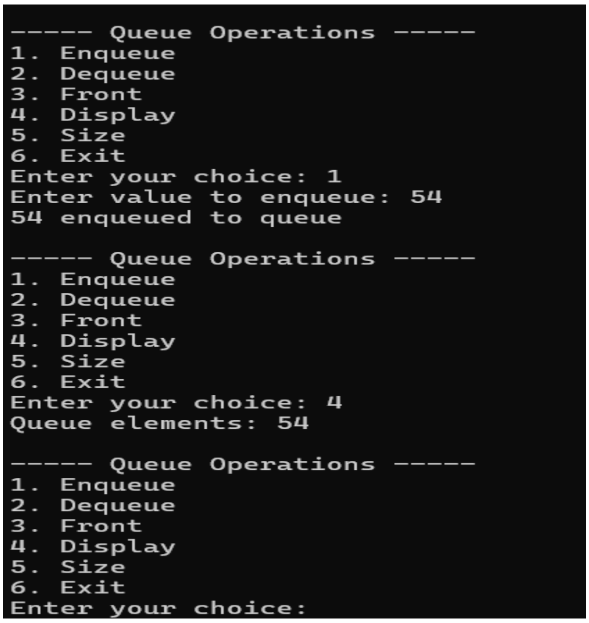
CODE:

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

#define MAX_SIZE 5

struct Queue{
        int items[MAX_SIZE];
        int front;
        int rear;
        int size; };

void initializeQueue(struct Queue *q){
        q->front = -1;
        q->rear = -1;
        q->size = 0;}

bool isFull(struct Queue *q){
        return q->size == MAX_SIZE;}

bool isEmpty(struct Queue *q){
        return q->size == 0;}

void enqueue(struct Queue *q, int value){
        if (isFull(q)){
                printf("Queue Overflow! Cannot enqueue %d\n", value);
                return;}

if (isEmpty(q)){
        q->front = 0;}

q->rear = (q->rear + 1) % MAX_SIZE;
q->items[q->rear] = value;
q->size++;

printf("%d enqueued to queue\n", value);}

int dequeue(struct Queue *q){
        if (isEmpty(q)){
                printf("Queue Underflow! Cannot dequeue from empty queue\n");
                return -1;}

int value = q->items[q->front];
```

```
if (q->front == q->rear){
initializeQueue(q);}
else{
    q->front = (q->front + 1) % MAX_SIZE;
    q->size--;}

return value;}

int front(struct Queue *q){
    if (isEmpty(q)){
        printf("Queue is empty\n");
        return -1;}

return q->items[q->front];}

void display(struct Queue *q){
    if (isEmpty(q)){
        printf("Queue is empty\n");
        return;}

printf("Queue elements: ");

int count = 0;
int i = q->front;

while (count < q->size){
    printf("%d ", q->items[i]);
    i = (i + 1) % MAX_SIZE;
    count++;}

printf("\n");}

int getSize(struct Queue *q){
return q->size;}

int main(){
    struct Queue queue;
    initializeQueue(&queue);

    int choice, value;

do{
    printf("\n----- Queue Operations -----\n");
    printf("1. Enqueue\n");
    printf("2. Dequeue\n");
    printf("3. Front\n");
    printf("4. Display\n");
    printf("5. Size\n");
    printf("6. Exit\n");
    printf("Enter your choice: ");
```

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

    switch (choice){
        case 1:
            printf("Enter value to enqueue: ");
            scanf("%d", &value);
            enqueue(&queue, value);
            break;

        case 2:
            value = dequeue(&queue);
            if (value != -1){
                printf("Dequeued value: %d\n", value);}
            break;

        case 3:
            value = front(&queue);
            if (value != -1){
                printf("Front element: %d\n", value);}
            break;

        case 4:
            display(&queue);
            break;

        case 5:
            printf("Queue size: %d\n", getSize(&queue));
            break;

        case 6:
            printf("Exiting program\n");
            break;

        default:
            printf("Invalid choice. Please try again.\n");
                                                        }}
        while (choice != 6);

        return 0;}
```

OUTPUT:



```
----- Queue Operations -----
1. Enqueue
2. Dequeue
3. Front
4. Display
5. Size
6. Exit
Enter your choice: 1
Enter value to enqueue: 54
54 enqueued to queue

----- Queue Operations -----
1. Enqueue
2. Dequeue
3. Front
4. Display
5. Size
6. Exit
Enter your choice: 4
Queue elements: 54

----- Queue Operations -----
1. Enqueue
2. Dequeue
3. Front
4. Display
5. Size
6. Exit
Enter your choice:
```

PRACTICAL-3

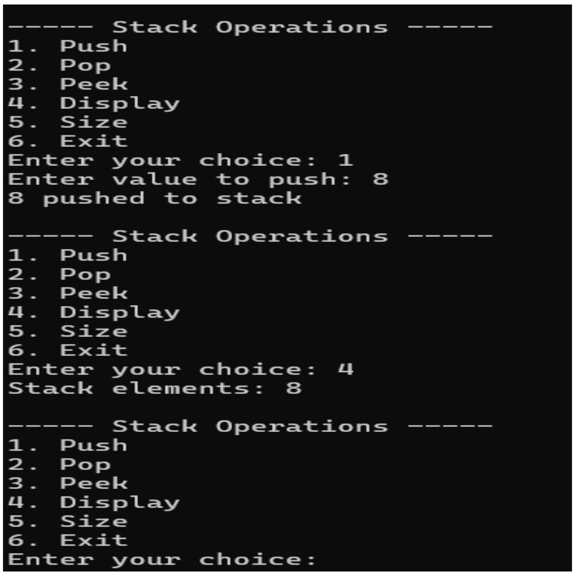
QUES: *Write a program to implement stack using Linked-List.*

CODE:

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

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

struct Stack{
    struct Node *top;
    int size;};

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

struct Stack *initializeStack(){
struct Stack *stack = (struct Stack *)malloc(sizeof(struct Stack));
if (stack == NULL){
    printf("Memory allocation failed\n");
    exit(1);}
    stack->top = NULL;
    stack->size = 0;
    return stack;}

bool isEmpty(struct Stack *stack){
return (stack->top == NULL);}

void push(struct Stack *stack, int data){
    struct Node *newNode = createNode(data);
    newNode->next = stack->top;
    stack->top = newNode;
    stack->size++;

    printf("%d pushed to stack\n", data);}

int pop(struct Stack *stack){
    if (isEmpty(stack)){
        printf("Stack Underflow! Cannot pop from empty stack\n");
        return -1;}

struct Node *temp = stack->top;
int data = temp->data;

stack->top = stack->top->next;

free(temp);
stack->size--;

return data;}

int peek(struct Stack *stack){
    if (isEmpty(stack)){
        printf("Stack is empty\n");
        return -1;}
    return stack->top->data;}

void display(struct Stack *stack){
    if (isEmpty(stack)){
        printf("Stack is empty\n");
        return;}

struct Node *temp = stack->top;
printf("Stack elements: ");

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

int getSize(struct Stack *stack){
    return stack->size;}

void freeStack(struct Stack *stack){
    struct Node *current = stack->top;
    struct Node *next;

    while (current != NULL){
        next = current->next;
        free(current);
        current = next;
                }

    free(stack);}

int main(){
    struct Stack *stack = initializeStack();
    int choice, data;
    do{
      printf("\n----- Stack Operations -----\n");
```

```c
  printf("1. Push\n");
printf("2. Pop\n");
printf("3. Peek\n");
printf("4. Display\n");
printf("5. Size\n");
printf("6. Exit\n");
printf("Enter your choice: ");
scanf("%d", &choice);
switch (choice){
    case 1:
        printf("Enter value to push: ");
        scanf("%d", &data);
        push(stack, data);
        break;

    case 2:
        data = pop(stack);
        if (data != -1){
            printf("Popped value: %d\n", data);}
        break;

    case 3:
        data = peek(stack);
        if (data != -1){
            printf("Top element: %d\n", data);}
        break;

    case 4:
        display(stack);
        break;

    case 5:
        printf("Stack size: %d\n", getSize(stack));
        break;
    case 6:
        printf("Exiting program\n");
        break;
    default:
        printf("Invalid choice. Please try again.\n");}}
while (choice != 6);
freeStack(stack);
return 0;}
```

OUTPUT:



```
----- Stack Operations -----
1. Push
2. Pop
3. Peek
4. Display
5. Size
6. Exit
Enter your choice: 1
Enter value to push: 8
8 pushed to stack

----- Stack Operations -----
1. Push
2. Pop
3. Peek
4. Display
5. Size
6. Exit
Enter your choice: 4
Stack elements: 8

----- Stack Operations -----
1. Push
2. Pop
3. Peek
4. Display
5. Size
6. Exit
Enter your choice:
```

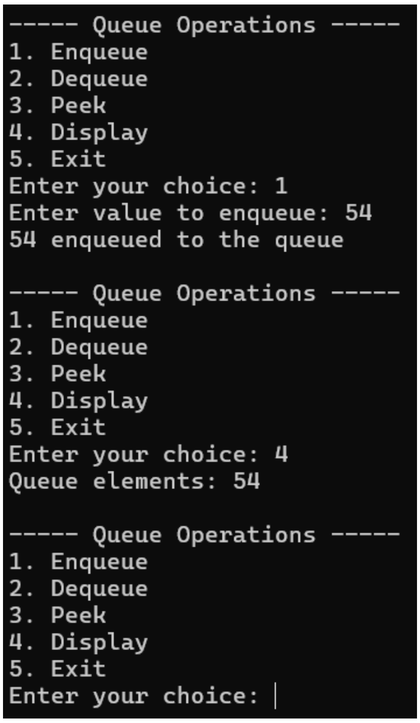
PRACTICAL-3

QUES: *Write a program to implement Queue using Linked-List.*

CODE:

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

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

struct Queue{
    struct Node *front, *rear;};

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

    struct Queue *initializeQueue(){
        struct Queue *queue = (struct Queue *)malloc(sizeof(struct Queue));
        if (queue == NULL){
            printf("Memory allocation failed\n");
            exit(1);}
    queue->front = queue->rear = NULL;
    return queue;}

bool isEmpty(struct Queue *queue){
    return (queue->front == NULL);}

void enqueue(struct Queue *queue, int data){
    struct Node *newNode = createNode(data);

    if (isEmpty(queue)){
        queue->front = queue->rear = newNode;
        printf("%d enqueued to the queue\n", data);
        return;}

queue->rear->next = newNode;
queue->rear = newNode;
printf("%d enqueued to the queue\n", data);}

int dequeue(struct Queue *queue){
    if (isEmpty(queue)){
        printf("Queue is empty. Cannot dequeue.\n");
        return -1;}

struct Node *temp = queue->front;
int data = temp->data;

queue->front = queue->front->next;

if (queue->front == NULL){
    queue->rear = NULL;}

free(temp);
return data;}

int peek(struct Queue *queue){
    if (isEmpty(queue)){
        printf("Queue is empty. Cannot peek.\n");
        return -1;}
    return queue->front->data;}

void display(struct Queue *queue){
    if (isEmpty(queue)){
        printf("Queue is empty.\n");
        return;}

struct Node *temp = queue->front;
printf("Queue elements: ");
while (temp != NULL){
    printf("%d ", temp->data);
    temp = temp->next;}
printf("\n");}

void freeQueue(struct Queue *queue){
    struct Node *current = queue->front;
    struct Node *next;

    while (current != NULL){
        next = current->next;
        free(current);
        current = next;}

free(queue);}

int main(){
    struct Queue *queue = initializeQueue();
    int choice, data;

    do{
        printf("\n----- Queue Operations -----\n");
        printf("1. Enqueue\n");
        printf("2. Dequeue\n");
        printf("3. Peek\n");
        printf("4. Display\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice){
            case 1:
                printf("Enter value to enqueue: ");
                scanf("%d", &data);
                enqueue(queue, data);
                break;

            case 2:
                data = dequeue(queue);
                if (data != -1){
                    printf("Dequeued value: %d\n", data);}
                break;

            case 3:
                data = peek(queue);
                if (data != -1){
                    printf("Front element: %d\n", data);}
                break;

            case 4:
                display(queue);
                break;

            case 5:
                printf("Exiting program\n");
                break;

            default:
                printf("Invalid choice. Please try again.\n");}

while (choice != 5){

    freeQueue(queue);

    return 0;}
```

OUTPUT:



```
----- Queue Operations -----
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Enter your choice: 1
Enter value to enqueue: 54
54 enqueued to the queue

----- Queue Operations -----
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Enter your choice: 4
Queue elements: 54

----- Queue Operations -----
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Enter your choice: |
```

PRACTICAL-4

QUES: *Write a program to evaluate infix,postfix,prefix expression.*

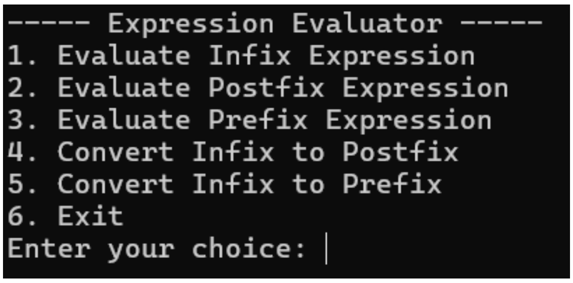
CODE:

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

#define MAX_SIZE 100

typedef struct {
int top;
int items[MAX_SIZE];}
IntStack;

typedef struct {
        int top;
        char items[MAX_SIZE];}
 CharStack;

void initIntStack(IntStack* s) {
s->top = -1;}

void initCharStack(CharStack* s) {
        s->top = -1;}

bool isIntStackEmpty(IntStack* s) {
        return s->top == -1;}

bool isCharStackEmpty(CharStack* s) {
        return s->top == -1;}

bool isIntStackFull(IntStack* s) {
        return s->top == MAX_SIZE - 1;}

bool isCharStackFull(CharStack* s) {
        return s->top == MAX_SIZE - 1;}

void intPush(IntStack* s, int value) {
        if (isIntStackFull(s)) {
                printf("Stack Overflow\n");
                return;}
        s->items[++(s->top)] = value;}

void charPush(CharStack* s, char value) {
        if (isCharStackFull(s)) {
                printf("Stack Overflow\n");
                return;}
```

```
    s→items[++(s->top)] = value;}

int intPop(IntStack* s) {
    if (isIntStackEmpty(s)) {
        printf("Stack Underflow\n");
        return -1;}
    return s->items[(s->top)--];}

char charPop(CharStack* s) {
    if (isCharStackEmpty(s)) {
        printf("Stack Underflow\n");
        return '\0';}
    return s->items[(s->top)--];}

int intPeek(IntStack* s) {
    if (isIntStackEmpty(s)) {
        printf("Stack is empty\n");
        return -1;}
    return s->items[s->top];}

char charPeek(CharStack* s) {
    if (isCharStackEmpty(s)) {
        printf("Stack is empty\n");
        return '\0';}
    return s->items[s->top];}

int precedence(char op) {
    switch (op) {
        case '+':
        case '-':
        return 1;
        case '*':
        case '/':
        return 2;
        case '^':
        return 3;}
    return -1}

bool isOperator(char ch) {
    return (ch == '+' || ch == '-' || ch == '*' || ch == '/' || ch == '^');}

int performOperation(int a, int b, char op) {
    switch (op) {
        case '+': return a + b;
        case '-': return a - b;
        case '*': return a * b;
        case '/':
        if (b == 0) {
            printf("Error: Division by zero\n");
            exit(1);}
    return a / b;
    case '^': {
        int result = 1;
        for (int i = 0; i < b; i++)
        result *= a;
        return result;}}
    return 0;}

int evaluatePostfix(char* expression) {
    IntStack stack;
    initIntStack(&stack);
    for (int i = 0; expression[i]; i++) {
        if (expression[i] == ' ' || expression[i] == '\t')
            continue;
        if (isdigit(expression[i])) {
            int num = 0;
            while (isdigit(expression[i])) {
                num = num * 10 + (expression[i] - '0');
                i++;}
            i--;
            intPush(&stack, num);}
        else if (isOperator(expression[i])) {
            int val2 = intPop(&stack);
            int val1 = intPop(&stack);
            intPush(&stack, performOperation(val1, val2, expression[i]));}}
return intPop(&stack);}

int evaluatePrefix(char* expression) {
    IntStack stack;
    initIntStack(&stack);
    int len = strlen(expression);
    for (int i = len - 1; i >= 0; i--) {
        if (expression[i] == ' ' || expression[i] == '\t')
            continue;
        if (isdigit(expression[i])) {
            int num = 0;
            int power = 1;
            while (i >= 0 && isdigit(expression[i])) {
                num = num + (expression[i] - '0') * power;
                power *= 10;
                i--;}
            i++;
            intPush(&stack, num);}
        else if (isOperator(expression[i])) {
        int val1 = intPop(&stack);
        int val2 = intPop(&stack);
        intPush(&stack, performOperation(val1, val2, expression[i]));}}
        return intPop(&stack);}

void infixToPostfix(char* infix, char* postfix) {
    CharStack stack;
    initCharStack(&stack);
    int j = 0;
    for (int i = 0; infix[i]; i++) {
        char c = infix[i];
        if (isdigit(c)) {
            while (isdigit(infix[i]))
                postfix[j++] = infix[i++];
                i--;
                postfix[j++] = ' ';}
            else if (c == '(') {
                charPush(&stack, c);}
            else if (c == ')') {
                while (!isCharStackEmpty(&stack) && charPeek(&stack) != '(')
                    postfix[j++] = charPop(&stack);
            if (!isCharStackEmpty(&stack) && charPeek(&stack) != '(')
                printf("Invalid expression\n");
            else
                charPop(&stack);}
        else if (isOperator(c)) {
while (!isCharStackEmpty(&stack) && precedence(c) <= precedence(charPeek(&stack)))
    postfix[j++] = charPop(&stack);
charPush(&stack, c);}}
while (!isCharStackEmpty(&stack))
postfix[j++] = charPop(&stack);
postfix[j] = '\0';}

void reverseString(char* str) {
    int len = strlen(str);
    for (int i = 0; i < len / 2; i++) {
        char temp = str[i];
        str[i] = str[len - i - 1];
        str[len - i - 1] = temp;}}

void infixToPrefix(char* infix, char* prefix) {
    char reversedInfix[MAX_SIZE];
    strcpy(reversedInfix, infix);
    reverseString(reversedInfix);
    for (int i = 0; reversedInfix[i]; i++) {
        if (reversedInfix[i] == '(')
            reversedInfix[i] = ')';
        else if (reversedInfix[i] == ')')
        reversedInfix[i] = '(';}
    char reversedPostfix[MAX_SIZE];
infixToPostfix(reversedInfix, reversedPostfix);
strcpy(prefix, reversedPostfix);
reverseString(prefix);}

int evaluateInfix(char* expression) {
    char postfix[MAX_SIZE];
infixToPostfix(expression, postfix);
return evaluatePostfix(postfix);}
```

```
int main() {
        char expression[MAX_SIZE];
        int choice;
do {
        printf("\n----- Expression Evaluator -----\n");
        printf("1. Evaluate Infix Expression\n");
        printf("2. Evaluate Postfix Expression\n");
        printf("3. Evaluate Prefix Expression\n");
        printf("4. Convert Infix to Postfix\n");
        printf("5. Convert Infix to Prefix\n");
        printf("6. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        getchar();
        if (choice >= 1 && choice <= 5) {
                printf("Enter the expression: ");
                fgets(expression, MAX_SIZE, stdin);
                expression[strcspn(expression, "\n")] = 0;}
switch (choice) {
        case 1:
                printf("Result: %d\n", evaluateInfix(expression));
                break;
        case 2:
                printf("Result: %d\n", evaluatePostfix(expression));
                break;
        case 3:
                printf("Result: %d\n", evaluatePrefix(expression));
                break;
        case 4: {
                char postfix[MAX_SIZE];
                infixToPostfix(expression, postfix);
                printf("Postfix expression: %s\n", postfix);
                break;}
        case 5: {
                char prefix[MAX_SIZE];
                infixToPrefix(expression, prefix);
                printf("Prefix expression: %s\n", prefix);
                break;}
        case 6:
                printf("Exiting program\n");
                break;
        default:
                printf("Invalid choice. Please try again.\n");}
        } while (choice != 6);
                return 0;}
```

OUTPUT:



```
----- Expression Evaluator -----
1. Evaluate Infix Expression
2. Evaluate Postfix Expression
3. Evaluate Prefix Expression
4. Convert Infix to Postfix
5. Convert Infix to Prefix
6. Exit
Enter your choice: |
```

PRACTICAL-5

QUES: *Write a program to convert infix expression to postfix expression.*

CODE:

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

#define MAX_SIZE 100

typedef struct{
	int top;
	char items[MAX_SIZE];}
 Stack;

void initStack(Stack *s){
	s->top = -1;}

bool isEmpty(Stack *s){
	return s->top == -1;}

bool isFull(Stack *s){
	return s->top == MAX_SIZE - 1;}

void push(Stack *s, char value){
	if (isFull(s)){
		printf("Stack Overflow\n");
		return;}
	s->items[++(s->top)] = value;}

char pop(Stack *s){
	if (isEmpty(s)){
		printf("Stack Underflow\n");
		return '\0';}
	return s->items[(s->top)--];}

char peek(Stack *s){
	if (isEmpty(s)){
		return '\0';}
return s->items[s->top];}

int precedence(char op){
	switch (op){
		case '+':
		case '-':
		return 1;
		case '*':
		case '/':
		return 2;
```

```
            case '^':
            return 3;}
            return -1;}

    bool isOperator(char ch){
        return (ch == '+' || ch == '-' || ch == '*' || ch == '/' || ch == '^');}

    void infixToPostfix(char *infix, char *postfix){
        Stack stack;
        initStack(&stack);
        int i, j = 0;

        for (i = 0; infix[i]; i++){
            char c = infix[i];

            if (isalnum(c)){
                postfix[j++] = c;}
            else if (c == '('){
                push(&stack, c);}
            else if (c == ')'){
                while (!isEmpty(&stack) && peek(&stack) != '('){
                    postfix[j++] = pop(&stack);}

            if (!isEmpty(&stack) && peek(&stack) == '('){
                pop(&stack);}
            else{
                printf("Invalid expression: Mismatched parentheses\n");
                exit(1);}}
            else if (isOperator(c))={
while (!isEmpty(&stack) && peek(&stack) != '(' && precedence(c) <= precedence(peek(&stack)))
{
    postfix[j++] = pop(&stack);}
    push(&stack, c);}
        else if (c == ' ' || c == '\t'){
        continue;}
        else{
            printf("Invalid character in expression: %c\n", c);
            exit(1);}}

        while (!isEmpty(&stack)){
            if (peek(&stack) == '('){
                printf("Invalid expression: Mismatched parentheses\n");
                exit(1);}
            postfix[j++] = pop(&stack);}

        postfix[j] = '\0';}

    bool validateInfix(char *infix){
        int parenCount = 0;
        int i;
    bool lastWasOperator = true;
```

```
        for (i = 0; infix[i]; i++){
            char c = infix[i];

            if (c == ' ' || c == '\t'){
                continue;}

            if (c == '('){
                parenCount++;
                lastWasOperator = true;}
            else if (c == ')'){
                parenCount--;
            if (parenCount < 0){
                printf("Error: Mismatched parentheses\n");
                return false;}
            lastWasOperator = false;}
            else if (isOperator(c)){
                if (lastWasOperator){
    printf("Error: Consecutive operators or operator after opening parenthesis\n");
                return false;}
            lastWasOperator = true;}
            else if (isalnum(c)){
            if (!lastWasOperator && i > 0 && (isalnum(infix[i - 1]) || infix[i - 1] == ')'))
{
    printf("Error: Missing operator between operands\n");
    return false;}
            lastWasOperator = false;}
        else{
            printf("Error: Invalid character '%c'\n", c);
            return false;}}

        if (parenCount != 0){
            printf("Error: Mismatched parentheses\n");
            return false;}

        if (lastWasOperator){
            printf("Error: Expression ends with an operator\n");
            return false;}

        return true;}

        int main(){
            char infix[MAX_SIZE], postfix[MAX_SIZE];

            printf("Enter infix expression: ");
            fgets(infix, MAX_SIZE, stdin);

            infix[strcspn(infix, "\n")] = 0;

            if (!validateInfix(infix)){
                return 1;}
```

The picture can't be displayed.

```
infixToPostfix(infix, postfix);

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

return 0;}
```

OUTPUT:

PRACTICAL-6

QUES: *Write a program to implement circular Linked-List*

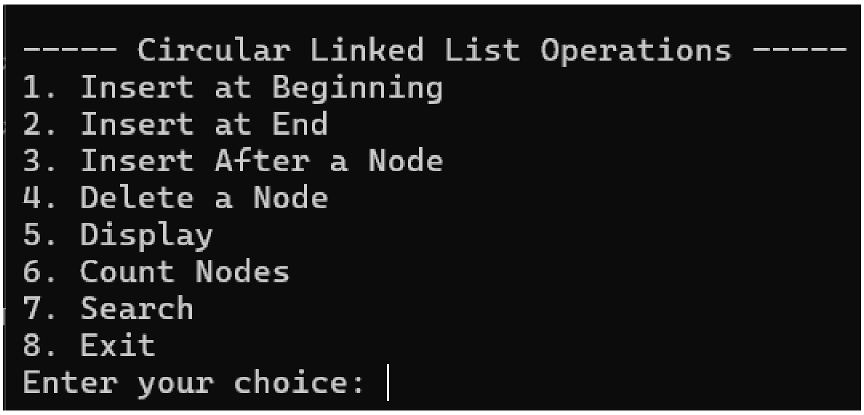
CODE:

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

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

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

bool isEmpty(struct Node *head){
        return head == NULL;}

struct Node *insertAtBeginning(struct Node *head, int data){
        struct Node *newNode = createNode(data);

        if (isEmpty(head)){
                return newNode;}

        struct Node *temp = head;
        while (temp->next != head){
                temp = temp->next;}

        newNode->next = head;
        temp->next = newNode;

        return newNode;}

struct Node *insertAtEnd(struct Node *head, int data){
        struct Node *newNode = createNode(data);

        if (isEmpty(head)){
                return newNode;}

        struct Node *temp = head;
        while (temp->next != head){
                temp = temp->next;}

temp->next = newNode;
newNode->next = head;
```

```
return head;}

void insertAfter(struct Node *head, int key, int data){
        if (isEmpty(head)){
                printf("List is empty. Cannot insert after a specific node.\n");
                return;}

struct Node *temp = head;

do{
        if (temp->data == key){
                struct Node *newNode = createNode(data);
                newNode->next = temp->next;
                temp->next = newNode;
                printf("Node inserted after %d\n", key);
                return;}
        temp = temp→next;}
         while (temp != head);

printf("Node with value %d not found\n", key);}

struct Node *deleteNode(struct Node *head, int key){
        if (isEmpty(head)){
                printf("List is empty. Nothing to delete.\n");
                return NULL;}

        if (head->data == key){
                if (head->next == head){
                        free(head);
                        return NULL;}

        struct Node *temp = head;
        while (temp->next != head){
                temp = temp->next;}

        struct Node *newHead = head->next;
        temp->next = newHead;
        free(head);
        return newHead;}

        struct Node *curr = head;
        struct Node *prev = NULL;

do{
        prev = curr;
        curr = curr->next;

        if (curr->data == key){
                prev->next = curr->next;
                free(curr);
```

```
            printf("Node with value %d deleted\n", key);
            return head;}}
 while (curr != head);

        printf("Node with value %d not found\n", key);
        return head;}

void display(struct Node *head){
        if (isEmpty(head)){
            printf("List is empty\n");
            return;}

struct Node *temp = head;
printf("Circular Linked List: ");

do{
        printf("%d -> ", temp->data);
        temp = temp→next;}
 while (temp != head);

printf("(back to %d)\n", head->data);}

int countNodes(struct Node *head){
if (isEmpty(head)){
        return 0;}

int count = 0;
struct Node *temp = head;

do{
        count++;
        temp = temp→next;}
        while (temp != head);
        return count;}

bool search(struct Node *head, int key){
        if (isEmpty(head)){
            return false;}

struct Node *temp = head;

do{
        if (temp->data == key){
        return true;}
        temp = temp→next;}
 while (temp != head);

return false;}

void freeList(struct Node *head){
        if (isEmpty(head)){
            return;}

struct Node *current = head;
struct Node *next;

do{
        next = current->next;
        free(current);
        current = next;}
         while (current != head);}

int main(){
        struct Node *head = NULL;
        int choice, data, key;

        do{
            printf("\n----- Circular Linked List Operations -----\n");
            printf("1. Insert at Beginning\n");
            printf("2. Insert at End\n");
            printf("3. Insert After a Node\n");
            printf("4. Delete a Node\n");
            printf("5. Display\n");
            printf("6. Count Nodes\n");
            printf("7. Search\n");
            printf("8. Exit\n");
            printf("Enter your choice: ");
            scanf("%d", &choice);

        switch (choice){
            case 1:
                    printf("Enter value to insert: ");
                    scanf("%d", &data);
                    head = insertAtBeginning(head, data);
                    break;

            case 2:
                    printf("Enter value to insert: ");
                    scanf("%d", &data);
                    head = insertAtEnd(head, data);
                    break;

            case 3:
                    printf("Enter the node value after which to insert: ");
                    scanf("%d", &key);
                    printf("Enter value to insert: ");
                    scanf("%d", &data);
                    insertAfter(head, key, data);
                    break;

            case 4:
                    printf("Enter value to delete: ");
                    scanf("%d", &data);
                    head = deleteNode(head, data);
                    break;

            case 5:
                    display(head);
                    break;

            case 6:
                    printf("Number of nodes: %d\n", countNodes(head));
                    break;

            case 7:
                    printf("Enter value to search: ");
                    scanf("%d", &data);
                    if (search(head, data)){
                        printf("%d found in the list\n", data);}
                    else{
                        printf("%d not found in the list\n", data);}
                    break;

            case 8:
                    printf("Exiting program\n");
                    break;

            default:
                    printf("Invalid choice. Please try again.\n");}}
while (choice != 8);

freeList(head);

return 0;}
```

OUTPUT:



```
----- Circular Linked List Operations -----
1. Insert at Beginning
2. Insert at End
3. Insert After a Node
4. Delete a Node
5. Display
6. Count Nodes
7. Search
8. Exit
Enter your choice:
```

PRACTICAL-7

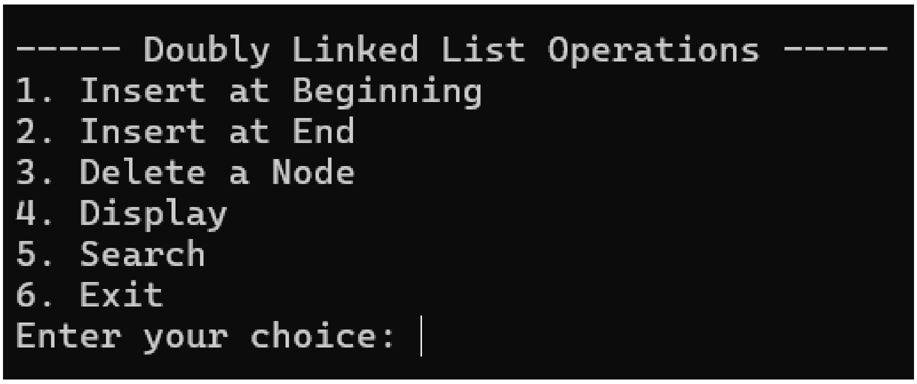
QUES: *Write a program to implement Doubly Linked-List.*

CODE:

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

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

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

struct Node *insertAtBeginning(struct Node *head, int data){
struct Node *newNode = createNode(data);
if (head == NULL){
return newNode;}
newNode->next = head;
head->prev = newNode;
return newNode;}

struct Node *insertAtEnd(struct Node *head, int data){
struct Node *newNode = createNode(data);
if (head == NULL){
return newNode;}
struct Node *temp = head;
while (temp->next != NULL){
temp = temp->next;}
temp->next = newNode;
newNode->prev = temp;
return head;}

struct Node *deleteNode(struct Node *head, int key){
if (head == NULL){
printf("List is empty. Nothing to delete.\n");
return NULL;}

struct Node *temp = head;
if (temp->data == key){
head = temp->next;
if (head != NULL){
head->prev = NULL;}
free(temp);
return head;}

while (temp != NULL && temp->data != key){
temp = temp->next;}

if (temp == NULL){
printf("Node with value %d not found\n", key);
return head;}

if (temp->next != NULL){
temp->next->prev = temp->prev;}
if (temp->prev != NULL){
temp->prev->next = temp->next;}

free(temp);
return head;}

void display(struct Node *head){
if (head == NULL){
printf("List is empty\n");
return;}

struct Node *temp = head;
printf("Doubly Linked List: ");
while (temp != NULL){
printf("%d <-> ", temp->data);
temp = temp->next;}
printf("NULL\n");}

bool search(struct Node *head, int key){
struct Node *temp = head;
while (temp != NULL){
if (temp->data == key){
return true;}
temp = temp->next;}
return false;}

void freeList(struct Node *head){
struct Node *current = head;
struct Node *next;

while (current != NULL){
next = current->next;
free(current);
current = next;}}
```

```
int main(){
struct Node *head = NULL;
int choice, data;

do{
    printf("\n----- Doubly Linked List Operations -----\n");
    printf("1. Insert at Beginning\n");
    printf("2. Insert at End\n");
    printf("3. Delete a Node\n");
    printf("4. Display\n");
    printf("5. Search\n");
    printf("6. Exit\n");
    printf("Enter your choice: ");
    scanf("%d", &choice);

switch (choice){
    case 1:
        printf("Enter value to insert: ");
        scanf("%d", &data);
        head = insertAtBeginning(head, data);
        break;

    case 2:
        printf("Enter value to insert: ");
        scanf("%d", &data);
        head = insertAtEnd(head, data);
        break;

    case 3:
        printf("Enter value to delete: ");
        scanf("%d", &data);
        head = deleteNode(head, data);
        break;

    case 4:
        display(head);
        break;

    case 5:
        printf("Enter value to search: ");
        scanf("%d", &data);
        if (search(head, data)){
            printf("%d found in the list\n", data);}
        else{
            printf("%d not found in the list\n", data);}
            break;

    case 6:
        printf("Exiting program\n");
        break;
default:
    printf("Invalid choice. Please try again.\n");}}
     while (choice != 6);
    freeList(head);
    return 0;}
```

OUTPUT:



```
----- Doubly Linked List Operations -----
1. Insert at Beginning
2. Insert at End
3. Delete a Node
4. Display
5. Search
6. Exit
Enter your choice: 
```

PRACTICAL-8

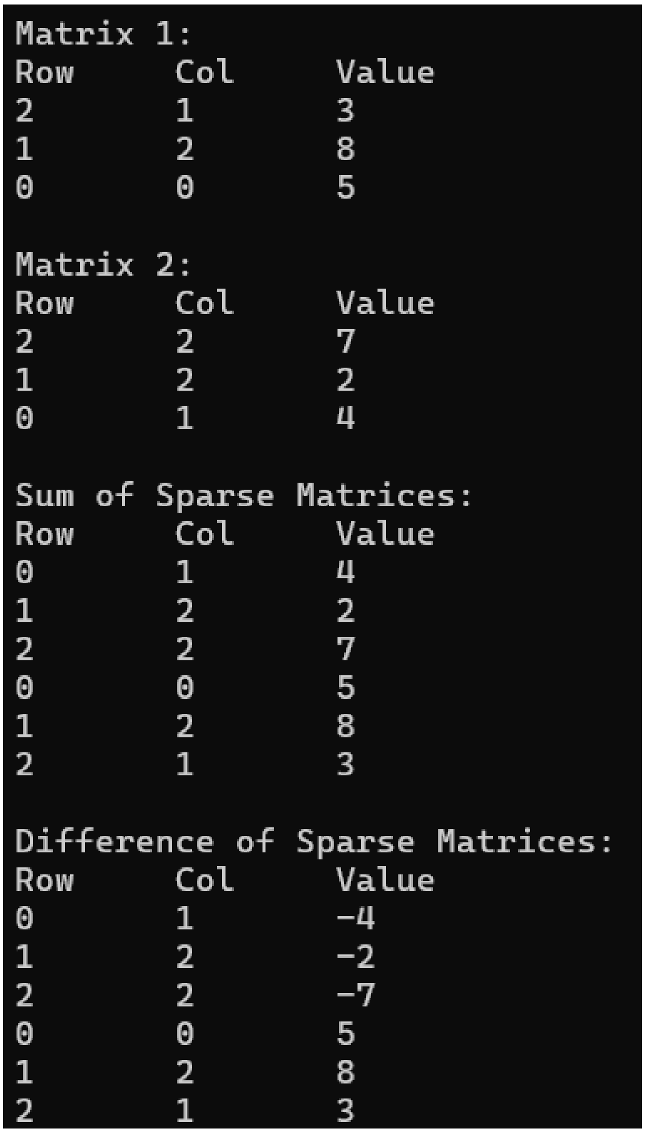
QUES: *Write a program to perform addition and subtraction on two sparse matrices.*

CODE:

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

struct Node{
int row;
int col;
int value;
struct Node *next;};

struct Node *createNode(int row, int col, int value){
struct Node *newNode = (struct Node *)malloc(sizeof(struct Node));
newNode->row = row;
newNode->col = col;
newNode->value = value;
newNode->next = NULL;
return newNode;}

struct Node *insert(struct Node *head, int row, int col, int value){
struct Node *newNode = createNode(row, col, value);
newNode->next = head;
return newNode;}

void display(struct Node *head){
struct Node *temp = head;
printf("Row\tCol\tValue\n");
while (temp != NULL){
printf("%d\t%d\t%d\n", temp->row, temp->col, temp->value);
temp = temp->next;}}

struct Node *addSparseMatrices(struct Node *mat1, struct Node *mat2){
struct Node *result = NULL;
struct Node *p1 = mat1;
struct Node *p2 = mat2;

while (p1 != NULL && p2 != NULL){
        if (p1->row < p2->row || (p1->row == p2->row && p1->col < p2->col)){
                result = insert(result, p1->row, p1->col, p1->value);
                p1 = p1->next;}
        else if (p1->row > p2->row || (p1->row == p2->row && p1->col > p2->col)){
                result = insert(result, p2->row, p2->col, p2->value);
                p2 = p2->next;}
        else{
                result = insert(result, p1->row, p1->col, p1->value + p2->value);
                p1 = p1->next;
                p2 = p2->next;}}

while (p1 != NULL){
        result = insert(result, p1->row, p1->col, p1->value);
        p1 = p1->next;}

        while (p2 != NULL){
                result = insert(result, p2->row, p2->col, p2->value);
                p2 = p2->next;}
                return result;}

        struct Node *subtractSparseMatrices(struct Node *mat1, struct Node *mat2){
                struct Node *result = NULL;
                struct Node *p1 = mat1;
                struct Node *p2 = mat2;

                while (p1 != NULL && p2 != NULL){
                    if (p1->row < p2->row || (p1->row == p2->row && p1->col < p2->col)){
                        result = insert(result, p1->row, p1->col, p1->value);
                        p1 = p1->next;}
                else if (p1->row > p2->row || (p1->row == p2->row && p1->col > p2->col)){
                    result = insert(result, p2->row, p2->col, -p2->value);
                    p2 = p2->next;}
                else{
                    result = insert(result, p1->row, p1->col, p1->value - p2->value);
                    p1 = p1->next;
                    p2 = p2->next;}}

        while (p1 != NULL){
                result = insert(result, p1->row, p1->col, p1->value);
                p1 = p1->next;}

        while (p2 != NULL){
                result = insert(result, p2->row, p2->col, -p2->value);
                p2 = p2->next;}
                return result;}

        int main(){
                struct Node *mat1 = NULL;
                struct Node *mat2 = NULL;

                mat1 = insert(mat1, 0, 0, 5);
                mat1 = insert(mat1, 1, 2, 8);
                mat1 = insert(mat1, 2, 1, 3);

                mat2 = insert(mat2, 0, 1, 4);
                mat2 = insert(mat2, 1, 2, 2);
                mat2 = insert(mat2, 2, 2, 7);

                printf("Matrix 1:\n");
```

```
            display(mat1);

            printf("\nMatrix 2:\n");
            display(mat2);

    struct Node *sum = addSparseMatrices(mat1, mat2);
    printf("\nSum of Sparse Matrices:\n");
    display(sum);

    struct Node *difference = subtractSparseMatrices(mat1, mat2);
            printf("\nDifference of Sparse Matrices:\n");
            display(difference);

    return 0;}
```

OUTPUT:



```
Matrix 1:
Row     Col     Value
2       1       3
1       2       8
0       0       5

Matrix 2:
Row     Col     Value
2       2       7
1       2       2
0       1       4

Sum of Sparse Matrices:
Row     Col     Value
0       1       4
1       2       2
2       2       7
0       0       5
1       2       8
2       1       3

Difference of Sparse Matrices:
Row     Col     Value
0       1       -4
1       2       -2
2       2       -7
0       0       5
1       2       8
2       1       3
```

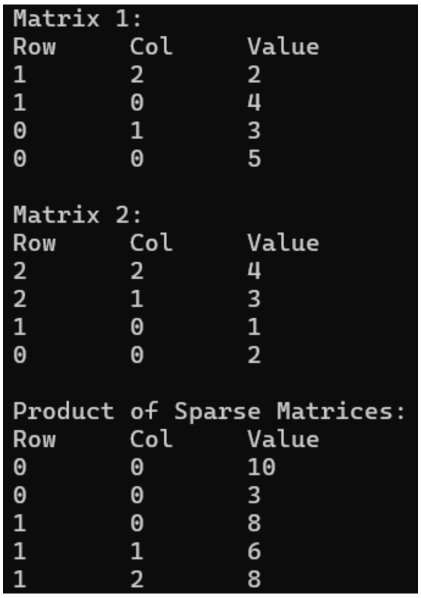
PRACTICAL-9

QUES: *Write a program to perform multiplication of two sparse matrices.*

CODE:

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

struct Node{
        int row;
        int col;
        int value;
        struct Node *next;};

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

struct Node *insert(struct Node *head, int row, int col, int value){
        struct Node *newNode = createNode(row, col, value);
        newNode->next = head;
        return newNode;}

void display(struct Node *head){
        struct Node *temp = head;
        printf("Row\tCol\tValue\n");
        while (temp != NULL){
                printf("%d\t%d\t%d\n", temp->row, temp->col, temp->value);
                temp = temp->next;}}

struct Node *multiplySparseMatrices(struct Node *mat1, struct Node *mat2, int rows1, int cols1, int
cols2){
        struct Node *result = NULL;

        struct Node *p1 = mat1;
        while (p1 != NULL){
                struct Node *p2 = mat2;
                while (p2 != NULL){
                        if (p1->col == p2->row){
                                int row = p1->row;
                                int col = p2->col;
                                int value = p1->value * p2->value;
                                result = insert(result, row, col, value);}
                        p2 = p2->next;}
                p1 = p1->next;}

        return result;}

void freeSparseMatrix(struct Node *head){
        struct Node *current = head;
        struct Node *next;
        while (current != NULL){
                next = current->next;
                free(current);
                current = next;}}

int main(){
        struct Node *mat1 = NULL;
        struct Node *mat2 = NULL;

        mat1 = insert(mat1, 0, 0, 5);
        mat1 = insert(mat1, 0, 1, 3);
        mat1 = insert(mat1, 1, 0, 4);
        mat1 = insert(mat1, 1, 2, 2);

        mat2 = insert(mat2, 0, 0, 2);
        mat2 = insert(mat2, 1, 0, 1);
        mat2 = insert(mat2, 2, 1, 3);
        mat2 = insert(mat2, 2, 2, 4);

        printf("Matrix 1:\n");
        display(mat1);

        printf("\nMatrix 2:\n");
        display(mat2);

        struct Node *product = multiplySparseMatrices(mat1, mat2, 2, 3, 3);
        printf("\nProduct of Sparse Matrices:\n");
        display(product);

        freeSparseMatrix(mat1);
        freeSparseMatrix(mat2);
        freeSparseMatrix(product);

        return 0;}
```

OUTPUT:



```
Matrix 1:
Row     Col     Value
1       2       2
1       0       4
0       1       3
0       0       5

Matrix 2:
Row     Col     Value
2       2       4
2       1       3
1       0       1
0       0       2

Product of Sparse Matrices:
Row     Col     Value
0       0       10
0       0       3
1       0       8
1       1       6
1       2       8
```

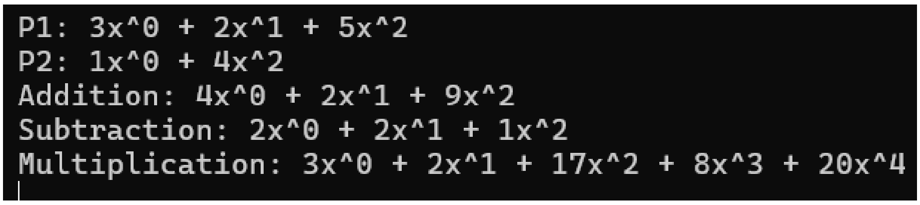
PRACTICAL-10

QUES: *Write a program to perform polynomial arithmetic.*

CODE:

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

typedef struct{
    int degree;
    int *coefficients;}
Polynomial;

Polynomial createPolynomial(int degree){
    Polynomial p;
    p.degree = degree;
    p.coefficients = (int *)malloc((degree + 1) * sizeof(int));
    for (int i = 0; i <= degree; i++){
    p.coefficients[i] = 0;}
    return p;}

Polynomial addPolynomials(Polynomial p1, Polynomial p2){
    int maxDegree = (p1.degree > p2.degree) ? p1.degree : p2.degree;
    Polynomial result = createPolynomial(maxDegree);

for (int i = 0; i <= maxDegree; i++){
    int coeff1 = (i <= p1.degree) ? p1.coefficients[i] : 0;
    int coeff2 = (i <= p2.degree) ? p2.coefficients[i] : 0;
    result.coefficients[i] = coeff1 + coeff2;}
    return result;}

Polynomial subtractPolynomials(Polynomial p1, Polynomial p2){
    int maxDegree = (p1.degree > p2.degree) ? p1.degree : p2.degree;
    Polynomial result = createPolynomial(maxDegree);

    for (int i = 0; i <= maxDegree; i++){
        int coeff1 = (i <= p1.degree) ? p1.coefficients[i] : 0;
        int coeff2 = (i <= p2.degree) ? p2.coefficients[i] : 0;
        result.coefficients[i] = coeff1 - coeff2;}
    return result;}

Polynomial multiplyPolynomials(Polynomial p1, Polynomial p2){
    Polynomial result = createPolynomial(p1.degree + p2.degree);

    for (int i = 0; i <= p1.degree; i++){
        for (int j = 0; j <= p2.degree; j++){
            result.coefficients[i + j] += p1.coefficients[i] * p2.coefficients[j];}}
            return result;}

    void printPolynomial(Polynomial p){
        for (int i = 0; i <= p.degree; i++){
                if (p.coefficients[i] != 0){
                        printf("%dx^%d ", p.coefficients[i], i);
                        if (i < p.degree){
                                printf("+ ");}}}
printf("\n");}

int main(){
        Polynomial p1 = createPolynomial(2);
        p1.coefficients[0] = 3;
        p1.coefficients[1] = 2;
        p1.coefficients[2] = 5;

        Polynomial p2 = createPolynomial(2);
        p2.coefficients[0] = 1;
        p2.coefficients[2] = 4;

        printf("P1: ");
        printPolynomial(p1);
        printf("P2: ");
        printPolynomial(p2);

        Polynomial sum = addPolynomials(p1, p2);
        printf("Addition: ");
        printPolynomial(sum);

        Polynomial difference = subtractPolynomials(p1, p2);
        printf("Subtraction: ");
        printPolynomial(difference);

        Polynomial product = multiplyPolynomials(p1, p2);
        printf("Multiplication: ");
        printPolynomial(product);

        free(p1.coefficients);
        free(p2.coefficients);
        free(sum.coefficients);
        free(difference.coefficients);
        free(product.coefficients);

        return 0;}
```

OUTPUT:



```
P1: 3x^0 + 2x^1 + 5x^2
P2: 1x^0 + 4x^2
Addition: 4x^0 + 2x^1 + 9x^2
Subtraction: 2x^0 + 2x^1 + 1x^2
Multiplication: 3x^0 + 2x^1 + 17x^2 + 8x^3 + 20x^4
```

PRACTICAL-11

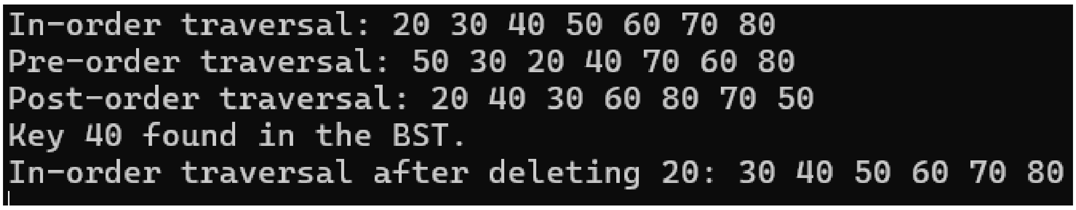
QUES: *Write a program to perform insertion,deletion,searching of a key and traversal in a Binary Search Tree.*

CODE:

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

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

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

Node *insert(Node *root, int key){
    if (root == NULL){
        return createNode(key);}
    if (key < root->key){
        root->left = insert(root->left, key);}
    else if (key > root->key){
        root->right = insert(root->right, key);}
    return root;}

Node *search(Node *root, int key){
    if (root == NULL || root->key == key){
        return root;}
    if (key < root->key){
        return search(root->left, key);}
    return search(root->right, key);}

    Node *findMin(Node *root){
        while (root->left != NULL){
            root = root->left;}
        return root;}

    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){
            Node *temp = root->right;
            free(root);
            return temp;}
    else if (root->right == NULL){
        Node *temp = root->left;
        free(root);
        return temp;}
    Node *temp = findMin(root->right);
    root->key = temp->key;
    root->right = deleteNode(root->right, temp->key);}
    return root;}

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

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

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

    int main(){
        Node *root = NULL;

        root = insert(root, 50);
        root = insert(root, 30);
        root = insert(root, 20);
        root = insert(root, 40);
        root = insert(root, 70);
        root = insert(root, 60);
        root = insert(root, 80);

        printf("In-order traversal: ");
        inOrder(root);
        printf("\n");
        printf("Pre-order traversal: ");
        preOrder(root);
        printf("\n");

        printf("Post-order traversal: ");
        postOrder(root);
        printf("\n");
```

```
int key = 40;
Node *foundNode = search(root, key);
if (foundNode){
        printf("Key %d found in the BST.\n", key);}
else{
        printf("Key %d not found in the BST.\n", key);}

root = deleteNode(root, 20);
printf("In-order traversal after deleting 20: ");
inOrder(root);
printf("\n");

return 0;}
```

OUTPUT:



```
In-order traversal: 20 30 40 50 60 70 80
Pre-order traversal: 50 30 20 40 70 60 80
Post-order traversal: 20 40 30 60 80 70 50
Key 40 found in the BST.
In-order traversal after deleting 20: 30 40 50 60 70 80
```

PRACTICAL-12

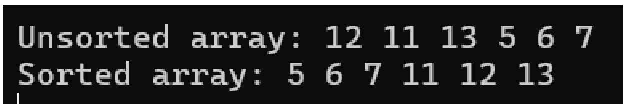
QUES: *Write a program to implement Heap Sort.*

CODE:

```
#include <stdio.h>

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

void heapify(int arr[], int n, int i){
        int largest = i;
        int left = 2 * i + 1;
        int right = 2 * i + 2;

        if (left < n && arr[left] > arr[largest]){
                largest = left;}

        if (right < n && arr[right] > arr[largest]){
                largest = right;}

        if (largest != i){
                swap(&arr[i], &arr[largest]);
                heapify(arr, n, largest);}}

void heapSort(int arr[], int n){
        for (int i = n / 2 - 1; i >= 0; i--){
                heapify(arr, n, i);}

        for (int i = n - 1; i > 0; i--){
                swap(&arr[0], &arr[i]);
                heapify(arr, i, 0);}}

void printArray(int arr[], int n){
        for (int i = 0; i < n; i++){
                printf("%d ", arr[i]);}
        printf("\n");}

int main(){
        int arr[] = {12, 11, 13, 5, 6, 7};
        int n = sizeof(arr) / sizeof(arr[0]);

        printf("Unsorted array: ");
        printArray(arr, n);

        heapSort(arr, n);

        printf("Sorted array: ");
```

```
    printArray(arr, n);

    return 0;}
```

OUTPUT:



```
Unsorted array: 12 11 13 5 6 7
Sorted array: 5 6 7 11 12 13
```

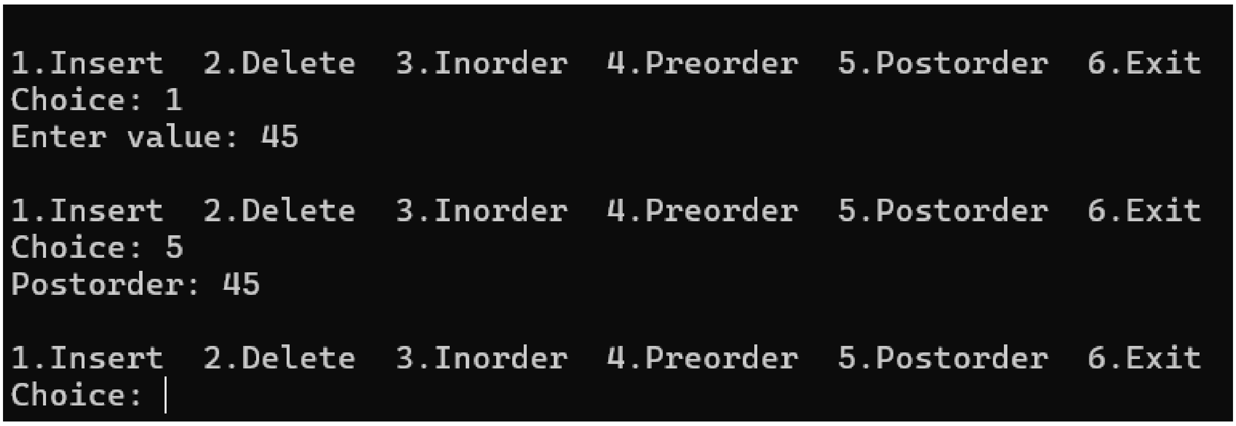
PRACTICAL -13

QUES: *Write a program to perform insertion, deletion and traversal on an AVL Tree.*

CODE:

```
#include <stdio.h>
#include <stdlib.h>
typedef struct Node {
    int data;
    struct Node *left, *right;
    int height;} Node;
int height(Node *node) {
    return node ? node->height : 0;}
int max(int a, int b) {
    return (a > b) ? a : b;}
Node* newNode(int data) {
    Node* node = (Node*)malloc(sizeof(Node));
    node->data = data;
    node->left = node->right = NULL;
    node->height = 1;
    return node;}
int getBalance(Node *node) {
    return node ? height(node->left) - height(node->right) : 0;}
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;}
Node* insert(Node* node, int data) {
    if (node == NULL)
        return newNode(data);
    if (data < node->data)
        node->left = insert(node->left, data);
    else if (data > node->data)
        node->right = insert(node->right, data);
    else
        return node;
    node->height = 1 + max(height(node->left), height(node->right));
    int balance = getBalance(node);
    if (balance > 1 && data < node->left->data)
        return rightRotate(node);
    if (balance < -1 && data > node->right->data)
        return leftRotate(node);
    if (balance > 1 && data > node->left->data) {
        node->left = leftRotate(node->left);
        return rightRotate(node);}
    if (balance < -1 && data < node->right->data) {
        node->right = rightRotate(node->right);
        return leftRotate(node);}
    return node;}
Node* minValueNode(Node* node) {
    Node* current = node;
    while (current->left != NULL)
        current = current->left;
    return current;}
Node* deleteNode(Node* root, int key) {
    if (root == NULL)
        return root;
    if (key < root->data)
        root->left = deleteNode(root->left, key);
    else if (key > root->data)
        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 = minValueNode(root->right);
            root->data = temp->data;
            root->right = deleteNode(root->right, temp->data);}}
    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 inOrder(Node* root) {
    if (root) {
        inOrder(root->left);
        printf("%d ", root->data);
        inOrder(root->right);}}
void preOrder(Node* root) {
    if (root) {
        printf("%d ", root->data);
        preOrder(root->left);
        preOrder(root->right);}}
void postOrder(Node* root) {
    if (root) {
        postOrder(root->left);
        postOrder(root->right);
        printf("%d ", root->data);}}
void freeTree(Node* root) {
    if (root) {
        freeTree(root->left);
        freeTree(root->right);
        free(root);}}
int main() {
    Node* root = NULL;
    int choice, data;
    while (1) {
        printf("\n1.Insert  2.Delete  3.Inorder  4.Preorder  5.Postorder  6.Exit\n");
        printf("Choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Enter value: ");
                scanf("%d", &data);
                root = insert(root, data);
                break;
            case 2:
                printf("Enter value: ");
                scanf("%d", &data);
                root = deleteNode(root, data);
                break;
            case 3:
                printf("Inorder: ");
                inOrder(root);
                printf("\n");
                break;
            case 4:
                printf("Preorder: ");
                preOrder(root);
                printf("\n");
                break;
            case 5:
                printf("Postorder: ");
                postOrder(root);
                printf("\n");
                break;
            case 6:
                freeTree(root);
                printf("Exiting.\n");
                exit(0);
            default:
                printf("Invalid choice\n");}}
                getch();
    return 0;}
```

OUTPUT:



```
1.Insert  2.Delete  3.Inorder  4.Preorder  5.Postorder  6.Exit
Choice: 1
Enter value: 45

1.Insert  2.Delete  3.Inorder  4.Preorder  5.Postorder  6.Exit
Choice: 5
Postorder: 45

1.Insert  2.Delete  3.Inorder  4.Preorder  5.Postorder  6.Exit
Choice: 
```

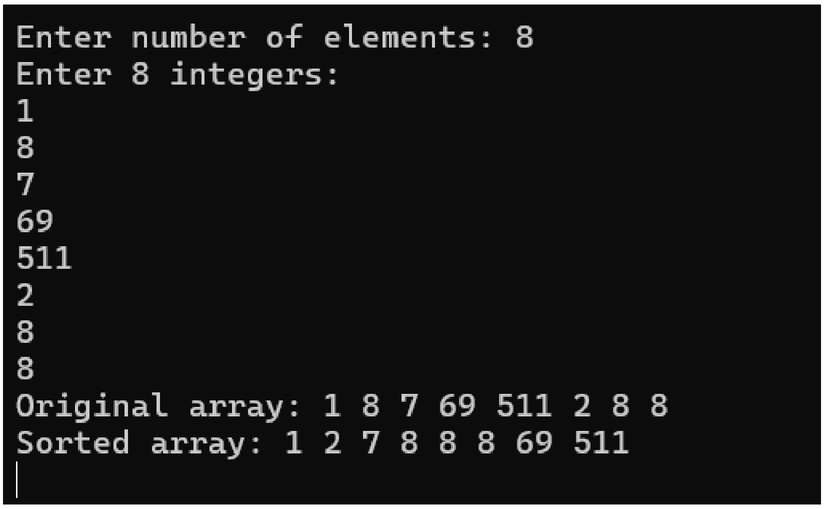
PRACTICAL-14

QUES: *Write a program to implement shell sort..*

CODE:

```
#include <stdio.h>
void shellSort(int arr[], int n) {
    for (int gap = n/2; gap > 0; gap /= 2) {
        for (int i = gap; i < n; i++) {
            int temp = arr[i];
            int j;
            for (j = i; j >= gap && arr[j - gap] > temp; j -= gap) {
                arr[j] = arr[j - gap];}
            arr[j] = temp;}}}
void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);}
    printf("\n");}
int main() {
    int arr[100], n;
    printf("Enter number of elements: ");
    scanf("%d", &n);
    printf("Enter %d integers:\n", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);}
    printf("Original array: ");
    printArray(arr, n);
    shellSort(arr, n);
    printf("Sorted array: ");
    printArray(arr, n);
    getch();
    return 0;}
```

OUTPUT:



```
Enter number of elements: 8
Enter 8 integers:
1
8
7
69
511
2
8
8
Original array: 1 8 7 69 511 2 8 8
Sorted array: 1 2 7 8 8 8 69 511
```

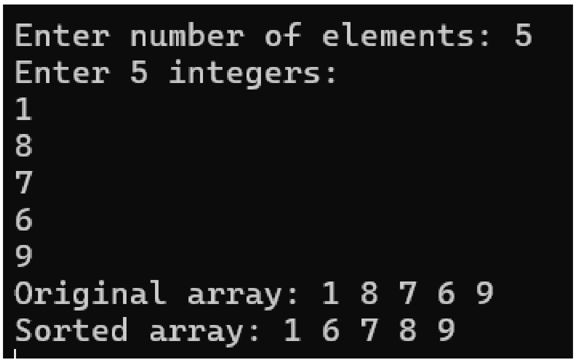
PRACTICAL-15

QUES: *Write a program to implement Merge Sort.*

CODE:

```
#include <stdio.h>
#include <stdlib.h>
void merge(int arr[], int l, int m, int r) {
    int i, j, k;
    int n1 = m - l + 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 = l;
    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);}}
void printArray(int arr[], int size) {
    for(int i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");}
int main() {
    int arr[100], n;
    printf("Enter number of elements: ");
    scanf("%d", &n);
    printf("Enter %d integers:\n", n);
    for(int i = 0; i < n; i++)
        scanf("%d", &arr[i]);
    printf("Original array: ");
    printArray(arr, n);
    mergeSort(arr, 0, n - 1);
    printf("Sorted array: ");
    printArray(arr, n);
    return 0;}
```

OUTPUT:



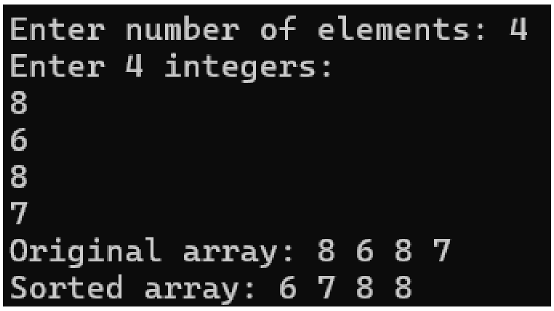
```
Enter number of elements: 5
Enter 5 integers:
1
8
7
6
9
Original array: 1 8 7 6 9
Sorted array: 1 6 7 8 9
```

PRACTICAL-16

QUES: *Write a program to implement Quick Sort.*
CODE:

```
#include <stdio.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 printArray(int arr[], int size) {
    for(int i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");}
int main() {
    int arr[100], n;
    printf("Enter number of elements: ");
    scanf("%d", &n);
    printf("Enter %d integers:\n", n);
    for(int i = 0; i < n; i++)
        scanf("%d", &arr[i]);
    printf("Original array: ");
    printArray(arr, n);
    quickSort(arr, 0, n - 1);
    printf("Sorted array: ");
    printArray(arr, n);
    return 0;}
```

OUTPUT:



```
Enter number of elements: 4
Enter 4 integers:
8
6
8
7
Original array: 8 6 8 7
Sorted array: 6 7 8 8
```

PRACTICAL-17

QUES: *Write a program to implement BFS and DFS.*

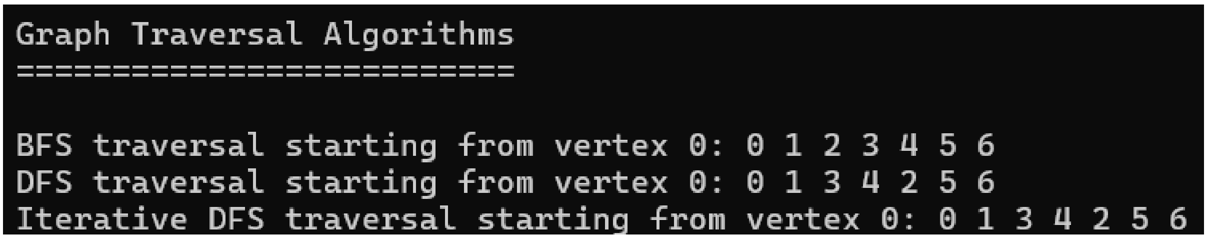
CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define MAX_VERTICES 100
typedef struct {
    int items[MAX_VERTICES];
    int front;
    int rear;} Queue;
typedef struct {
    int items[MAX_VERTICES];
    int top;} Stack;
typedef struct {
    int vertices;
    bool adjacencyMatrix[MAX_VERTICES][MAX_VERTICES];} Graph;
Queue* createQueue() {
    Queue* q = (Queue*)malloc(sizeof(Queue));
    q->front = -1;
    q->rear = -1;
    return q;}
bool isEmpty(Queue* q) {
    return q->rear == -1;}
void enqueue(Queue* q, int value) {
    if (q->rear == MAX_VERTICES - 1)
        printf("Queue is full\n");
    else {
        if (q->front == -1)
            q->front = 0;
        q->rear++;
        q->items[q->rear] = value;}}
int dequeue(Queue* q) {
    int item;
    if (isEmpty(q)) {
        printf("Queue is empty\n");
        return -1;
    } else {
        item = q->items[q->front];
        q->front++;
        if (q->front > q->rear) {
            q->front = -1;
            q->rear = -1;}
        return item;}}
Stack* createStack() {
    Stack* s = (Stack*)malloc(sizeof(Stack));
    s->top = -1;
    return s;}
bool isStackEmpty(Stack* s) {
    return s->top == -1;}
```

```
void push(Stack* s, int value) {
    if (s->top == MAX_VERTICES - 1)
        printf("Stack is full\n");
    else {
        s->top++;
        s->items[s->top] = value;}}
int pop(Stack* s) {
    if (isStackEmpty(s)) {
        printf("Stack is empty\n");
        return -1;
    } else {
        int item = s->items[s->top];
        s->top--;
        return item;}}
Graph* createGraph(int vertices) {
    Graph* graph = (Graph*)malloc(sizeof(Graph));
    graph->vertices = vertices;
    for (int i = 0; i < vertices; i++) {
        for (int j = 0; j < vertices; j++) {
            graph->adjacencyMatrix[i][j] = false;}}
    return graph;}
void addEdge(Graph* graph, int src, int dest) {
    graph->adjacencyMatrix[src][dest] = true;
    graph->adjacencyMatrix[dest][src] = true;}
void BFS(Graph* graph, int startVertex) {
    bool visited[MAX_VERTICES] = {false};
    Queue* q = createQueue();
    visited[startVertex] = true;
    printf("BFS traversal starting from vertex %d: ", startVertex);
    printf("%d ", startVertex);
    enqueue(q, startVertex);
    while (!isEmpty(q)) {
        int currentVertex = dequeue(q);
        for (int i = 0; i < graph->vertices; i++) {
            if (graph->adjacencyMatrix[currentVertex][i] && !visited[i]) {
                printf("%d ", i);
                visited[i] = true;
                enqueue(q, i);}}}
    printf("\n");
    free(q);}
void DFSRecursive(Graph* graph, int vertex, bool visited[]) {
    visited[vertex] = true;
    printf("%d ", vertex);
    for (int i = 0; i < graph->vertices; i++) {
        if (graph->adjacencyMatrix[vertex][i] && !visited[i])
            DFSRecursive(graph, i, visited);}}
void DFS(Graph* graph, int startVertex) {
    bool visited[MAX_VERTICES] = {false};
    printf("DFS traversal starting from vertex %d: ", startVertex);
    DFSRecursive(graph, startVertex, visited);
    printf("\n");}
```

```
void DFSIterative(Graph* graph, int startVertex) {
    bool visited[MAX_VERTICES] = {false};
    Stack* s = createStack();
    printf("Iterative DFS traversal starting from vertex %d: ", startVertex);
    push(s, startVertex);
    while (!isStackEmpty(s)) {
        int currentVertex = pop(s);
        if (!visited[currentVertex]) {
            printf("%d ", currentVertex);
            visited[currentVertex] = true;}
        for (int i = graph->vertices - 1; i >= 0; i--) {
            if (graph->adjacencyMatrix[currentVertex][i] && !visited[i]) {
                push(s, i);}}}
    printf("\n");
    free(s);}
int main() {
    Graph* graph = createGraph(7);
    addEdge(graph, 0, 1);
    addEdge(graph, 0, 2);
    addEdge(graph, 1, 3);
    addEdge(graph, 1, 4);
    addEdge(graph, 2, 5);
    addEdge(graph, 2, 6);
    printf("Graph Traversal Algorithms\n");
    printf("==========================\n\n");
    BFS(graph, 0);
    DFS(graph, 0);
    DFSIterative(graph, 0);
    free(graph);
    getch();
    return 0;}
```

OUTPUT:



```
Graph Traversal Algorithms
==========================

BFS traversal starting from vertex 0: 0 1 2 3 4 5 6
DFS traversal starting from vertex 0: 0 1 3 4 2 5 6
Iterative DFS traversal starting from vertex 0: 0 1 3 4 2 5 6
```

PRACTICAL-18

QUES: *Write a program to perform Hashing using different resolution techniques.*

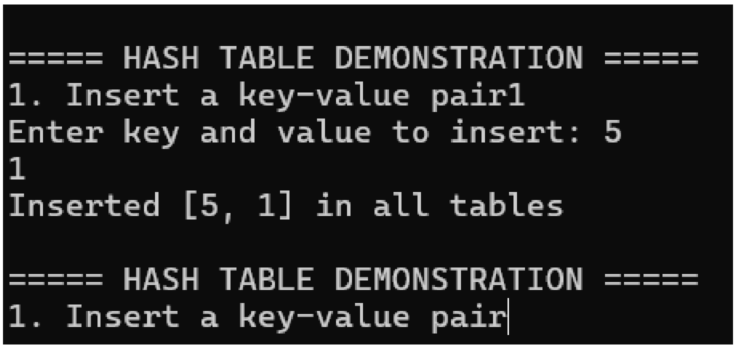
CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define TABLE_SIZE 10
#define DELETED_NODE (struct DataItem*)(0xFFFFFFFFFFFFFFFFL)
struct DataItem {
    int key;
    int data;};
struct LinkedListNode {
    int key;
    int data;
    struct LinkedListNode* next;};
struct DataItem* hashArrayLP[TABLE_SIZE];
struct DataItem* hashArrayQP[TABLE_SIZE];
struct DataItem* hashArrayDH[TABLE_SIZE];
struct LinkedListNode* hashArraySC[TABLE_SIZE];
int hashCode1(int key) {
    return key % TABLE_SIZE;}
int hashCode2(int key) {
    return 7 - (key % 7);}
void initializeHashTables() {
    for (int i = 0; i < TABLE_SIZE; i++) {
        hashArrayLP[i] = NULL;
        hashArrayQP[i] = NULL;
        hashArrayDH[i] = NULL;
        hashArraySC[i] = NULL;}}
void insertLP(int key, int data) {
    struct DataItem* item = (struct DataItem*)malloc(sizeof(struct DataItem));
    item->key = key;
    item->data = data;
    int hashIndex = hashCode1(key);
    while (hashArrayLP[hashIndex] != NULL && hashArrayLP[hashIndex] != DELETED_NODE) {
        if (hashArrayLP[hashIndex]->key == key) {
            hashArrayLP[hashIndex]->data = data;
            free(item);
            return;}
        hashIndex = (hashIndex + 1) % TABLE_SIZE;}
    hashArrayLP[hashIndex] = item;}
void insertQP(int key, int data) {
    struct DataItem* item = (struct DataItem*)malloc(sizeof(struct DataItem));
    item->key = key;
    item->data = data;
    int hashIndex = hashCode1(key);
    int i = 0;
    while (hashArrayQP[(hashIndex + i*i) % TABLE_SIZE] != NULL &&
           hashArrayQP[(hashIndex + i*i) % TABLE_SIZE] != DELETED_NODE) {
```

```
        if (hashArrayQP[(hashIndex + i*i) % TABLE_SIZE]->key == key) {
            hashArrayQP[(hashIndex + i*i) % TABLE_SIZE]->data = data;
            free(item);
            return;}
        i++;}
    hashArrayQP[(hashIndex + i*i) % TABLE_SIZE] = item;}
void insertDH(int key, int data) {
    struct DataItem* item = (struct DataItem*)malloc(sizeof(struct DataItem));
    item->key = key;
    item->data = data;
    int hashIndex = hashCode1(key);
    int stepSize = hashCode2(key);
    while (hashArrayDH[hashIndex] != NULL && hashArrayDH[hashIndex] != DELETED_NODE) {
        if (hashArrayDH[hashIndex]->key == key) {
            hashArrayDH[hashIndex]->data = data;
            free(item);
            return;}
        hashIndex = (hashIndex + stepSize) % TABLE_SIZE;}
    hashArrayDH[hashIndex] = item;}
void insertSC(int key, int data) {
    int hashIndex = hashCode1(key);
    struct LinkedListNode* newNode = (struct LinkedListNode*)malloc(sizeof(struct
LinkedListNode));
    newNode->key = key;
    newNode->data = data;
    newNode->next = NULL;
    if (hashArraySC[hashIndex] == NULL) {
        hashArraySC[hashIndex] = newNode;
    } else {
        struct LinkedListNode* current = hashArraySC[hashIndex];
        struct LinkedListNode* prev = NULL;
        while (current != NULL) {
            if (current->key == key) {
                current->data = data;
                free(newNode);
                return;}
            prev = current;
            current = current->next;}
        prev->next = newNode;}}
struct DataItem* searchLP(int key) {
    int hashIndex = hashCode1(key);
    int originalIndex = hashIndex;
    while (hashArrayLP[hashIndex] != NULL) {
        if (hashArrayLP[hashIndex]->key == key) {
            return hashArrayLP[hashIndex];}
        hashIndex = (hashIndex + 1) % TABLE_SIZE;
        if (hashIndex == originalIndex) {
            break;}}
    return NULL;}
struct DataItem* searchQP(int key) {
    int hashIndex = hashCode1(key);
    int i = 0;
    int indexToCheck;
    do {
        indexToCheck = (hashIndex + i*i) % TABLE_SIZE;
        if (hashArrayQP[indexToCheck] == NULL) {
            return NULL;}
        if (hashArrayQP[indexToCheck]->key == key) {
            return hashArrayQP[indexToCheck];}
        i++;
    } while (i < TABLE_SIZE);
    return NULL;}
struct DataItem* searchDH(int key) {
    int hashIndex = hashCode1(key);
    int stepSize = hashCode2(key);
    int originalIndex = hashIndex;
    while (hashArrayDH[hashIndex] != NULL) {
        if (hashArrayDH[hashIndex]->key == key) {
            return hashArrayDH[hashIndex];}
        hashIndex = (hashIndex + stepSize) % TABLE_SIZE;
        if (hashIndex == originalIndex) {
            break;}}
    return NULL;}
struct LinkedListNode* searchSC(int key) {
    int hashIndex = hashCode1(key);
    struct LinkedListNode* current = hashArraySC[hashIndex];
    while (current != NULL) {
        if (current->key == key) {
            return current;}
        current = current->next;}
    return NULL;}
bool deleteLP(int key) {
    int hashIndex = hashCode1(key);
    int originalIndex = hashIndex;
    while (hashArrayLP[hashIndex] != NULL) {
        if (hashArrayLP[hashIndex]->key == key) {
            free(hashArrayLP[hashIndex]);
            hashArrayLP[hashIndex] = DELETED_NODE;
            return true;}
        hashIndex = (hashIndex + 1) % TABLE_SIZE;
        if (hashIndex == originalIndex) {
            break;}}
    return false;}
bool deleteQP(int key) {
    int hashIndex = hashCode1(key);
    int i = 0;
    int indexToCheck;
    do {
        indexToCheck = (hashIndex + i*i) % TABLE_SIZE;
        if (hashArrayQP[indexToCheck] == NULL) {
            return false;}
        if (hashArrayQP[indexToCheck]->key == key) {
            free(hashArrayQP[indexToCheck]);
            hashArrayQP[indexToCheck] = DELETED_NODE;
            return true;}
        i++;} while (i < TABLE_SIZE);
    return false;}
bool deleteDH(int key) {
    int hashIndex = hashCode1(key);
    int stepSize = hashCode2(key);
    int originalIndex = hashIndex;
    while (hashArrayDH[hashIndex] != NULL) {
        if (hashArrayDH[hashIndex]->key == key) {
            free(hashArrayDH[hashIndex]);
            hashArrayDH[hashIndex] = DELETED_NODE;
            return true;}
        hashIndex = (hashIndex + stepSize) % TABLE_SIZE;
        if (hashIndex == originalIndex) {
            break;}}
    return false;}
bool deleteSC(int key) {
    int hashIndex = hashCode1(key);
    struct LinkedListNode* current = hashArraySC[hashIndex];
    struct LinkedListNode* prev = NULL;
    if (current == NULL) {
        return false;}
    if (current->key == key) {
        hashArraySC[hashIndex] = current->next;
        free(current);
        return true;}
    while (current != NULL && current->key != key) {
        prev = current;
        current = current->next;}
    if (current == NULL) {
        return false;}
    prev->next = current->next;
    free(current);
    return true;}
void displayLP() {
    printf("\nLinear Probing Hash Table:\n");
    printf("Index\tKey\tValue\n");
    printf("----------------------\n");
    for (int i = 0; i < TABLE_SIZE; i++) {
        if (hashArrayLP[i] != NULL && hashArrayLP[i] != DELETED_NODE) {
            printf("%d\t%d\t%d\n", i, hashArrayLP[i]->key, hashArrayLP[i]->data);
        } else if (hashArrayLP[i] == DELETED_NODE) {
            printf("%d\t<deleted>\n", i);
        } else {
            printf("%d\t<empty>\n", i);}}}
void displayQP() {
    printf("\nQuadratic Probing Hash Table:\n");
    printf("Index\tKey\tValue\n");
```

```c
    printf("---------------------\n");
    for (int i = 0; i < TABLE_SIZE; i++) {
        if (hashArrayQP[i] != NULL && hashArrayQP[i] != DELETED_NODE) {
            printf("%d\t%d\t%d\n", i, hashArrayQP[i]->key, hashArrayQP[i]->data);
        } else if (hashArrayQP[i] == DELETED_NODE) {
            printf("%d\t<deleted>\n", i);
        } else {
            printf("%d\t<empty>\n", i);}}}
void displayDH() {
    printf("\nDouble Hashing Hash Table:\n");
    printf("Index\tKey\tValue\n");
    printf("---------------------\n");
    for (int i = 0; i < TABLE_SIZE; i++) {
        if (hashArrayDH[i] != NULL && hashArrayDH[i] != DELETED_NODE) {
            printf("%d\t%d\t%d\n", i, hashArrayDH[i]->key, hashArrayDH[i]->data);
        } else if (hashArrayDH[i] == DELETED_NODE) {
            printf("%d\t<deleted>\n", i);
        } else {
            printf("%d\t<empty>\n", i);}}}
void displaySC() {
    printf("\nSeparate Chaining Hash Table:\n");
    printf("Index\tEntries\n");
    printf("----------------\n");
    for (int i = 0; i < TABLE_SIZE; i++) {
        printf("%d\t", i);
        if (hashArraySC[i] == NULL) {
            printf("<empty>");
        } else {
            struct LinkedListNode* current = hashArraySC[i];
            while (current != NULL) {
                printf("[%d,%d]", current->key, current->data);
                if (current->next != NULL) {
                    printf(" -> ");}
                current = current->next;}}
        printf("\n");}}
void cleanupHashTables() {
    for (int i = 0; i < TABLE_SIZE; i++) {
        if (hashArrayLP[i] != NULL && hashArrayLP[i] != DELETED_NODE) {
            free(hashArrayLP[i]);}
        if (hashArrayQP[i] != NULL && hashArrayQP[i] != DELETED_NODE) {
            free(hashArrayQP[i]);}
        if (hashArrayDH[i] != NULL && hashArrayDH[i] != DELETED_NODE) {
            free(hashArrayDH[i]);}
        struct LinkedListNode* current = hashArraySC[i];
        while (current != NULL) {
            struct LinkedListNode* temp = current;
            current = current->next;
            free(temp);}}}
void insertInAllTables(int key, int data) {
    insertLP(key, data);
    insertQP(key, data);
    insertDH(key, data);
    insertSC(key, data);}
void displayMenu() {
    printf("\n===== HASH TABLE DEMONSTRATION =====\n");
    printf("1. Insert a key-value pair");}
int main() {
    int choice, key, data;
    struct DataItem* item;
    struct LinkedListNode* node;
    bool deleted;
    initializeHashTables();
    insertInAllTables(1, 20);
    insertInAllTables(11, 30);
    insertInAllTables(21, 40);
    insertInAllTables(31, 50);
    while (1) {
        displayMenu();
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Enter key and value to insert: ");
                scanf("%d %d", &key, &data);
                insertInAllTables(key, data);
                printf("Inserted [%d, %d] in all tables\n", key, data);
                break;
            case 2:
                printf("Enter key to search: ");
                scanf("%d", &key);
                item = searchLP(key);
                if (item != NULL) {
                    printf("Linear Probing: Found [%d, %d]\n", item->key, item-
>data);
                } else {
                    printf("Linear Probing: Key %d not found\n", key);}
                item = searchQP(key);
                if (item != NULL) {
                    printf("Quadratic Probing: Found [%d, %d]\n", item->key, item-
>data);
                } else {
                    printf("Quadratic Probing: Key %d not found\n", key);}
                item = searchDH(key);
                if (item != NULL) {
                    printf("Double Hashing: Found [%d, %d]\n", item->key, item-
>data);
                } else {
                    printf("Double Hashing: Key %d not found\n", key);}
                node = searchSC(key);
                if (node != NULL) {
                    printf("Separate Chaining: Found [%d, %d]\n", node->key, node-
>data);
                } else {
                    printf("Separate Chaining: Key %d not found\n", key);}
                break;
            case 3:
                printf("Enter key to delete: ");
                scanf("%d", &key);
                deleted = deleteLP(key);
                printf("Linear Probing: %s\n", deleted ? "Deleted successfully" :
"Key not found");
                deleted = deleteQP(key);
                printf("Quadratic Probing: %s\n", deleted ? "Deleted successfully"
: "Key not found");
                deleted = deleteDH(key);
                printf("Double Hashing: %s\n", deleted ? "Deleted successfully" :
"Key not found");
                deleted = deleteSC(key);
                printf("Separate Chaining: %s\n", deleted ? "Deleted successfully"
: "Key not found");
                break;
            case 4:
                displayLP();
                displayQP();
                displayDH();
                displaySC();
                break;
            case 5:
                cleanupHashTables();
                printf("Exiting program. Memory cleaned up.\n");
                return 0;
            default:
                printf("Invalid choice. Please try again.\n");}}
                getch();
    return 0;}
```

OUTPUT:



```
===== HASH TABLE DEMONSTRATION =====
1. Insert a key-value pair1
Enter key and value to insert: 5
1
Inserted [5, 1] in all tables

===== HASH TABLE DEMONSTRATION =====
1. Insert a key-value pair
```

PRACTICAL-19

QUES: *Write a program to implement algorithm for Minimum Spanning Tree.*
CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <limits.h>
#define MAX_VERTICES 100
#define INF INT_MAX
typedef struct {
    int src, dest, weight;
} Edge;
typedef struct {
    int V, E;
    Edge* edges;
} Graph;
typedef struct {
    int parent;
    int rank;
} Subset;
Graph* createGraph(int V, int E) {
    Graph* graph = (Graph*)malloc(sizeof(Graph));
    graph->V = V;
    graph->E = E;
    graph->edges = (Edge*)malloc(E * sizeof(Edge));
    return graph;}
int find(Subset subsets[], int i) {
    if (subsets[i].parent != i)
        subsets[i].parent = find(subsets, subsets[i].parent);
    return subsets[i].parent;}
void Union(Subset subsets[], int x, int y) {
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);
    if (subsets[xroot].rank < subsets[yroot].rank)
        subsets[xroot].parent = yroot;
    else if (subsets[xroot].rank > subsets[yroot].rank)
        subsets[yroot].parent = xroot;
    else {
        subsets[yroot].parent = xroot;
        subsets[xroot].rank++;}}
int compareEdges(const void* a, const void* b) {
    return ((Edge*)a)->weight - ((Edge*)b)->weight;}
void kruskalMST(Graph* graph) {
    int V = graph->V;
    Edge result[V-1];
    int e = 0;
    int i = 0;
    qsort(graph->edges, graph->E, sizeof(graph->edges[0]), compareEdges);
    Subset* subsets = (Subset*)malloc(V * sizeof(Subset));
    for (int v = 0; v < V; v++) {
        subsets[v].parent = v;
        subsets[v].rank = 0;}
    while (e < V - 1 && i < graph->E) {
        Edge next_edge = graph->edges[i++];
        int x = find(subsets, next_edge.src);
        int y = find(subsets, next_edge.dest);
        if (x != y) {
            result[e++] = next_edge;
            Union(subsets, x, y);}}
    printf("\nKruskal's MST:\n");
    int totalWeight = 0;
    for (i = 0; i < e; i++) {
        printf("Edge: %d -- %d  Weight: %d\n",
               result[i].src, result[i].dest, result[i].weight);
        totalWeight += result[i].weight;}
    printf("Total Weight of MST: %d\n", totalWeight);
    free(subsets);}
int minKey(int key[], bool mstSet[], int V) {
    int min = INF, min_index;
    for (int v = 0; v < V; v++)
        if (mstSet[v] == false && key[v] < min)
            min = key[v], min_index = v;
    return min_index;}
void primMST(int graph[MAX_VERTICES][MAX_VERTICES], int V) {
    int parent[V];
    int key[V];
    bool mstSet[V];
    for (int i = 0; i < V; i++)
        key[i] = INF, mstSet[i] = false;
    key[0] = 0;
    parent[0] = -1;
    for (int count = 0; count < V - 1; count++) {
        int u = minKey(key, mstSet, V);
        mstSet[u] = true;
        for (int v = 0; v < V; v++)
            if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v])
                parent[v] = u, key[v] = graph[u][v];}
    printf("\nPrim's MST:\n");
    int totalWeight = 0;
    for (int i = 1; i < V; i++) {
        printf("Edge: %d -- %d  Weight: %d\n", parent[i], i, graph[parent[i]][i]);
        totalWeight += graph[parent[i]][i];}
    printf("Total Weight of MST: %d\n", totalWeight);}
int main() {
    int choice, V, E, u, v, w;
    Graph* graph;
    int adjMatrix[MAX_VERTICES][MAX_VERTICES];
    printf("Enter number of vertices: ");
    scanf("%d", &V);
    for (int i = 0; i < V; i++)
        for (int j = 0; j < V; j++)
            adjMatrix[i][j] = 0;
    printf("Enter number of edges: ");
    scanf("%d", &E);
    graph = createGraph(V, E);
    printf("Enter edge information (source destination weight):\n");
    for (int i = 0; i < E; i++) {
        scanf("%d %d %d", &u, &v, &w);
        graph->edges[i].src = u;
        graph->edges[i].dest = v;
        graph->edges[i].weight = w;
        adjMatrix[u][v] = w;
        adjMatrix[v][u] = w;}
    while (1) {
        printf("\n=== MINIMUM SPANNING TREE ALGORITHMS ===\n");
        printf("1. Find MST using Kruskal's Algorithm\n");
        printf("2. Find MST using Prim's Algorithm\n");
        printf("3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                kruskalMST(graph);
                break;
            case 2:
                primMST(adjMatrix, V);
                break;
            case 3:
                free(graph->edges);
                free(graph);
                return 0;
            default:
                printf("Invalid choice! Try again.\n");}}
                getch();}
```

PRACTICAL – 20

QUES: *Write a program to implement shortest path algorithm.*

CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <limits.h>
#define MAX_VERTICES 100
#define INF INT_MAX
void dijkstra(int graph[MAX_VERTICES][MAX_VERTICES], int src, int V) {
    int dist[V];
    bool sptSet[V];
    int parent[V];
    for (int i = 0; i < V; i++) {
        dist[i] = INF;
        sptSet[i] = false;
        parent[i] = -1;}
    dist[src] = 0;
    for (int count = 0; count < V - 1; count++) {
        int min = INF, min_index;
        for (int v = 0; v < V; v++)
            if (sptSet[v] == false && dist[v] <= min)
                min = dist[v], min_index = v;
        int u = min_index;
        sptSet[u] = true;
        for (int v = 0; v < V; v++)
            if (!sptSet[v] && graph[u][v] && dist[u] != INF && dist[u] +
graph[u][v] < dist[v]) {
                parent[v] = u;
                dist[v] = dist[u] + graph[u][v];}}
    printf("\nVertex\tDistance\tPath");
    for (int i = 0; i < V; i++) {
        printf("\n%d -> %d\t%d\t\t", src, i, dist[i]);
        int j = i;
        while (j != src) {
            printf("%d <- ", j);
            j = parent[j];}
        printf("%d", src);}
    printf("\n");}
void bellmanFord(int graph[MAX_VERTICES][MAX_VERTICES], int src, int V) {
    int dist[V];
    int parent[V];
    for (int i = 0; i < V; i++) {
        dist[i] = INF;
        parent[i] = -1;}
    dist[src] = 0;
    for (int i = 0; i < V - 1; i++) {
        for (int u = 0; u < V; u++) {
            for (int v = 0; v < V; v++) {
                if (graph[u][v] && dist[u] != INF && dist[u] + graph[u][v] <
dist[v]) {
                    dist[v] = dist[u] + graph[u][v];
                    parent[v] = u;}}}}
    for (int u = 0; u < V; u++) {
        for (int v = 0; v < V; v++) {
            if (graph[u][v] && dist[u] != INF && dist[u] + graph[u][v] < dist[v]) {
                printf("Graph contains negative weight cycle\n");
                return;}}}
    printf("\nVertex\tDistance\tPath");
    for (int i = 0; i < V; i++) {
        printf("\n%d -> %d\t%d\t\t", src, i, dist[i]);
        int j = i;
        while (j != src && j != -1) {
            printf("%d <- ", j);
            j = parent[j];}
        if (j != -1) printf("%d", src);}
    printf("\n");}
void floydWarshall(int graph[MAX_VERTICES][MAX_VERTICES], int V) {
    int dist[V][V];
    int next[V][V];
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            dist[i][j] = graph[i][j];
            if (graph[i][j] != 0 && graph[i][j] != INF)
                next[i][j] = j;
            else
                next[i][j] = -1;}}
    for (int k = 0; k < V; k++) {
        for (int i = 0; i < V; i++) {
            for (int j = 0; j < V; j++) {
                if (dist[i][k] != INF && dist[k][j] != INF && dist[i][k] +
dist[k][j] < dist[i][j]) {
                    dist[i][j] = dist[i][k] + dist[k][j];
                    next[i][j] = next[i][k];}}}}
    printf("\nAll-Pairs Shortest Paths:\n");
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            if (i != j) {
                printf("\n%d -> %d: Distance = %d, Path: %d", i, j, dist[i][j], i);
                int path = i;
                while (path != j) {
                    path = next[path][j];
                    if (path == -1) {
                        printf(" -> No path exists");
                        break;}
                    printf(" -> %d", path);}}}}
    printf("\n");}
int main() {
    int V, E, choice, src;
    int graph[MAX_VERTICES][MAX_VERTICES];
```

```
printf("Enter number of vertices: ");
scanf("%d", &V);
for (int i = 0; i < V; i++) {
    for (int j = 0; j < V; j++) {
        graph[i][j] = 0;}}
printf("Enter number of edges: ");
scanf("%d", &E);
printf("Enter edge information (source destination weight):\n");
for (int i = 0; i < E; i++) {
    int u, v, w;
    scanf("%d %d %d", &u, &v, &w);
    graph[u][v] = w;}
while (1) {
    printf("\n=== SHORTEST PATH ALGORITHMS ===\n");
    printf("1. Dijkstra's Algorithm\n");
    printf("2. Bellman-Ford Algorithm\n");
    printf("3. Floyd-Warshall Algorithm\n");
    printf("4. Exit\n");
    printf("Enter your choice: ");
    scanf("%d", &choice);
    switch (choice) {
        case 1:
            printf("Enter source vertex: ");
            scanf("%d", &src);
            dijkstra(graph, src, V);
            break;
        case 2:
            printf("Enter source vertex: ");
            scanf("%d", &src);
            bellmanFord(graph, src, V);
            break;
        case 3:
            floydWarshall(graph, V);
            break;
        case 4:
            return 0;
        default:
            printf("Invalid choice! Try again.\n");}}
            getch();}
```