DATA STRUCTURES

PRACTICAL FILE

B.Tech CSE-DS

Submitted By:
KUSHANK KHUNDARA
ENR. No. 02716451924

Submitted to:
Ms. KAHISKA WADHWA

INDEX

11	**Program to perform insertion, deletion and searching of a key in Binary Search Tree.**
12	**Program to implement Heap Sort.**
13	**Write a program to perform insertion, deletion and traversal on an AVL Tree.**
14	**Write a program to implement shell sort.**
15	**Write a program to implement Merge Sort.**
16	**Write a program to implement Quick Sort.**
17	**Write a program to implement BFS and DFS.**
18	**Write a program to perform Hashing using different resolution techniques.**
19	**Write a program to implement an algorithm for minimum spanning tree.**
20	**Write a program to implement the shortest path algorithm.**

Q1. WAP TO CALCULATE GCD OF TWO NUMBERS USING RECURSION

PROGRAM:

```
#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 numbers: ");
    scanf("%d %d", &num1, &num2);
    printf("The GCD of %d and %d is: %d\n", num1, num2, gcd(num1, num2));


    return 0;
}
```

OUTPUT:

Output   Clear

```
Enter two numbers: 8 9
The GCD of 8 and 9 is: 1


=== Code Execution Successful ===
```

Q2. WAP TO IMPLEMENT -
i) STACK USING ARRAYS

<u>PROGRAM:</u>

```c
#include <stdio.h>
#include <stdlib.h>
#define MAX 5

struct Stack {
    int arr[MAX];
    int top;
};
void initStack(struct Stack *stack) {
    stack->top = -1;
}
int isFull(struct Stack *stack) {
    return stack->top == MAX - 1;
}
int isEmpty(struct Stack *stack) {
    return stack->top == -1;
}
void push(struct Stack *stack, int value) {
    if (isFull(stack)) {
        printf("Stack is full! Cannot push %d\n", value);
    } else {
        stack->arr[++stack->top] = value;
        printf("%d pushed onto stack\n", value);
    }
}
int pop(struct Stack *stack) {
    if (isEmpty(stack)) {
        printf("Stack is empty! Cannot pop\n");
        return -1;
    } else {
        int poppedValue = stack->arr[stack->top--];
        return poppedValue;
    }
}
int peek(struct Stack *stack) {
    if (isEmpty(stack)) {
        printf("Stack is empty! Nothing to peek\n");
```

```
        return -1;
    } else {
        return stack->arr[stack->top];
    }
}
void display(struct Stack *stack) {
    if (isEmpty(stack)) {
        printf("Stack is empty\n");
    } else {
        printf("Stack contents: ");
        for (int i = 0; i <= stack->top; i++) {
            printf("%d ", stack->arr[i]);
        }
        printf("\n");
    }
}
int main() {
    struct Stack stack;
    initStack(&stack);
    push(&stack, 10);
    push(&stack, 20);
    push(&stack, 30);
    push(&stack, 40);
    push(&stack, 50);
    display(&stack);
    push(&stack, 60);
    printf("Popped element: %d\n", pop(&stack));
    display(&stack);
    printf("Top element is: %d\n", peek(&stack));
    return 0;
}
```

OUTPUT:

Output Clear

```
10 pushed onto stack
20 pushed onto stack
30 pushed onto stack
40 pushed onto stack
50 pushed onto stack
Stack contents: 10 20 30 40 50
Stack is full! Cannot push 60
Popped element: 50
Stack contents: 10 20 30 40
Top element is: 40


=== Code Execution Successful ===
```

ii) QUEUE USING ARRAYS

PROGRAM:

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

#define MAX 5

int queue[MAX];
int front = -1, rear = -1;

int isFull() {
    if (rear == MAX - 1) {
        return 1;
    }
    return 0;
}

int isEmpty() {
    if (front == -1 || front > rear) {
        return 1;
    }
    return 0;
}

void enqueue(int value) {
    if (isFull()) {
        printf("Queue is full, cannot enqueue %d\n", value);
        return;
    }
    if (front == -1) {
        front = 0;
    }
    rear++;
    queue[rear] = value;
    printf("Enqueued: %d\n", value);
}

int dequeue() {
    if (isEmpty()) {
        printf("Queue is empty, cannot dequeue\n");
        return -1;
    }
```

```
    int value = queue[front];
    front++;
    return value;
}

void display() {
    if (isEmpty()) {
        printf("Queue is empty\n");
        return;
    }
    printf("Queue elements: ");
    for (int i = front; i <= rear; i++) {
        printf("%d ", queue[i]);
    }
    printf("\n");
}

int main() {
    int choice, value;

    while (1) {
        printf("\nQueue Operations:\n");
        printf("1. Enqueue\n");
        printf("2. Dequeue\n");
        printf("3. Display\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter value to enqueue: ");
                scanf("%d", &value);
                enqueue(value);
                break;
            case 2:
                value = dequeue();
                if (value != -1) {
                    printf("Dequeued: %d\n", value);
                }
                break;
            case 3:
                display();
                break;
```

```
            case 4:
                exit(0);
            default:
                printf("Invalid choice! Please try again.\n");
        }
    }

    return 0;
}
```

OUTPUT:

Output Clear

```
Queue Operations:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 1
Enter value to enqueue: 5
Enqueued: 5

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

Queue Operations:
1. Enqueue
2. Dequeue
3. Display
4. Exit
```

Q3. WAP TO IMPLEMENT
i) STACK USING LINKED LIST

PROGRAM:

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

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

struct Node* top = NULL;

int isEmpty() {
    return top == NULL;
}

void push(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = top;
    top = newNode;
}

int pop() {
    if (isEmpty()) {
        printf("Stack is empty\n");
        return -1;
    }
    struct Node* temp = top;
    int value = top->data;
    top = top->next;
    free(temp);
    return value;
}

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

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

int main() {
    int choice, value;

    while (1) {
        printf("\nStack Operations:\n");
        printf("1. Push\n");
        printf("2. Pop\n");
        printf("3. Display\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter value to push: ");
                scanf("%d", &value);
                push(value);
                break;
            case 2:
                value = pop();
                if (value != -1) {
                    printf("Popped: %d\n", value);
                }
                break;
            case 3:
                display();
                break;
            case 4:
                exit(0);
            default:
                printf("Invalid choice! Please try again.\n");
        }
    }

    return 0;
```

}

OUTPUT:

Output Clear

```
2. Pop
3. Display
4. Exit
Enter your choice: 1
Enter value to push: 5

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

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

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

ii) QUEUE USING LINKED LISTS

PROGRAM:

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

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

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

int isEmpty() {
    return front == NULL;
}

void enqueue(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = NULL;
    if (rear == NULL) {
        front = rear = newNode;
    } else {
        rear->next = newNode;
        rear = newNode;
    }
}

int dequeue() {
    if (isEmpty()) {
        printf("Queue is empty\n");
        return -1;
    }
    struct Node* temp = front;
    int value = front->data;
    front = front->next;
    if (front == NULL) {
        rear = NULL;
    }
    free(temp);
    return value;
```

```
}

void display() {
    if (isEmpty()) {
        printf("Queue is empty\n");
        return;
    }
    struct Node* temp = front;
    printf("Queue elements: ");
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

int main() {
    int choice, value;

    while (1) {
        printf("\nQueue Operations:\n");
        printf("1. Enqueue\n");
        printf("2. Dequeue\n");
        printf("3. Display\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter value to enqueue: ");
                scanf("%d", &value);
                enqueue(value);
                break;
            case 2:
                value = dequeue();
                if (value != -1) {
                    printf("Dequeued: %d\n", value);
                }
                break;
            case 3:
                display();
                break;
            case 4:
```

```
            exit(0);
        default:
            printf("Invalid choice! Please try again.\n");
        }
    }

    return 0;
}
```

OUTPUT:

Output Clear

```
2. Pop
3. Display
4. Exit
Enter your choice: 1
Enter value to push: 5

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

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

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

Q4. PROGRAM TO EVALUATE INFIX, POSTFIX, PREFIX EXPRESSIONS.

PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <math.h>
#define MAX 100

struct Stack {
   int arr[MAX];
   int top;
};
void initStack(struct Stack* stack) {
   stack->top = -1;
}
int isEmpty(struct Stack* stack) {
   return stack->top == -1;
}
int isFull(struct Stack* stack) {
   return stack->top == MAX - 1;
}
void push(struct Stack* stack, int value) {
   if (isFull(stack)) {
      printf("Stack Overflow\n");
      return;
   }
   stack->arr[++(stack->top)] = value;
}

int pop(struct Stack* stack) {
   if (isEmpty(stack)) {
      printf("Stack Underflow\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];
}
int precedence(char operator) {
    if (operator == '+' || operator == '-') {
        return 1;
    } else if (operator == '*' || operator == '/') {
        return 2;
    } else if (operator == '^') {
        return 3;
    }
    return 0;
}
int isOperator(char ch) {
    return (ch == '+' || ch == '-' || ch == '*' || ch == '/' || ch == '^');
}
int applyOperator(int operand1, int operand2, char operator) {
    switch (operator) {
        case '+': return operand1 + operand2;
        case '-': return operand1 - operand2;
        case '*': return operand1 * operand2;
        case '/': return operand1 / operand2;
        case '^': return (int)pow(operand1, operand2);
    }
    return 0;
}
int evaluateInfix(char* expression) {
    struct Stack values, operators;
    initStack(&values);
    initStack(&operators);
    int i = 0;

    while (expression[i] != '\0') {
        char ch = expression[i];

        if (isdigit(ch)) {
            push(&values, ch - '0');
        } else if (ch == '(') {
            push(&operators, ch);
        } else if (ch == ')') {
            while (!isEmpty(&operators) && peek(&operators) != '(') {
                int val2 = pop(&values);
                int val1 = pop(&values);
                char operator = pop(&operators);
                push(&values, applyOperator(val1, val2, operator));
```

```
        }
        pop(&operators);
    } else if (isOperator(ch)) {
        while (!isEmpty(&operators) && precedence(peek(&operators)) >= precedence(ch)) {
            int val2 = pop(&values);
            int val1 = pop(&values);
            char operator = pop(&operators);
            push(&values, applyOperator(val1, val2, operator));
        }
        push(&operators, ch);
    }
    i++;
  }
  while (!isEmpty(&operators)) {
    int val2 = pop(&values);
    int val1 = pop(&values);
    char operator = pop(&operators);
    push(&values, applyOperator(val1, val2, operator));
  }

  return pop(&values);
}
int evaluatePostfix(char* expression) {
  struct Stack stack;
  initStack(&stack);
  int i = 0;

  while (expression[i] != '\0') {
    char ch = expression[i];

    if (isdigit(ch)) {
      push(&stack, ch - '0');
    } else if (isOperator(ch)) {
      int val2 = pop(&stack);
      int val1 = pop(&stack);
      push(&stack, applyOperator(val1, val2, ch));
    }
    i++;
  }
  return pop(&stack);
}
int evaluatePrefix(char* expression) {
  struct Stack stack;
  initStack(&stack);
```

```
    int i = strlen(expression) - 1;
    while (i >= 0) {
        char ch = expression[i];

        if (isdigit(ch)) {
            push(&stack, ch - '0');
        } else if (isOperator(ch)) {
            int val1 = pop(&stack);
            int val2 = pop(&stack);
            push(&stack, applyOperator(val1, val2, ch));
        }
        i--;
    }
    return pop(&stack);
}

int main() {
    char infix[MAX], postfix[MAX], prefix[MAX];
    int choice;
    while (1) {
        printf("\nChoose an option:\n");
        printf("1. Evaluate Infix Expression\n");
        printf("2. Evaluate Postfix Expression\n");
        printf("3. Evaluate Prefix Expression\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter infix expression: ");
                scanf("%s", infix);
                printf("Result: %d\n", evaluateInfix(infix));
                break;
            case 2:
                printf("Enter postfix expression: ");
                scanf("%s", postfix);
                printf("Result: %d\n", evaluatePostfix(postfix));
                break;
            case 3:
                printf("Enter prefix expression: ");
                scanf("%s", prefix);
                printf("Result: %d\n", evaluatePrefix(prefix));
                break;
```

```
        case 4:
          exit(0);
        default:
          printf("Invalid choice\n");
      }
    }
    return 0;
}
```

OUTPUT:

Output   Clear

```
Choose an option:
1. Evaluate Infix Expression
2. Evaluate Postfix Expression
3. Evaluate Prefix Expression
4. Exit
Enter your choice: 1
Enter infix expression (single-digit, no spaces): 2+3/4
Result: 2

Choose an option:
1. Evaluate Infix Expression
2. Evaluate Postfix Expression
3. Evaluate Prefix Expression
4. Exit
Enter your choice: 2
Enter postfix expression (single-digit, no spaces): 23+
Result: 5

Choose an option:
1. Evaluate Infix Expression
2. Evaluate Postfix Expression
3. Evaluate Prefix Expression
4. Exit
Enter your choice: 3
Enter prefix expression (single-digit, no spaces): +54
Result: 9
```

Q5. Program to convert Infix Expression to Postfix Expression.

<u>Program:</u>

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

#define MAX 100

int precedence(char op) {
    if (op == '+' || op == '-') return 1;
    if (op == '*' || op == '/') return 2;
    if (op == '^') return 3;
    return 0;
}
int isOperator(char ch) {
    return (ch == '+' || ch == '-' || ch == '*' || ch == '/' || ch == '^');
}
void infixToPostfix(char *infix, char *postfix) {
    char stack[MAX];
    int top = -1, k = 0;

    for (int i = 0; infix[i] != '\0'; i++) {
        char ch = infix[i];

        if (isalnum(ch)) {
            postfix[k++] = ch;
        } else if (ch == '(') {
            stack[++top] = ch;
        } else if (ch == ')') {
            while (top != -1 && stack[top] != '(') {
                postfix[k++] = stack[top--];
            }
            top--;
        } else if (isOperator(ch)) {
            while (top != -1 && precedence(stack[top]) >= precedence(ch)) {
                postfix[k++] = stack[top--];
            }
            stack[++top] = ch;
        }
```

```
    }

    while (top != -1) {
        postfix[k++] = stack[top--];
    }
    postfix[k] = '\0';
}
int main() {
    char infix[MAX], postfix[MAX];

    printf("Enter an infix expression: ");
    fgets(infix, sizeof(infix), stdin);
    infix[strcspn(infix, "\n")] = '\0';

    infixToPostfix(infix, postfix);

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

    return 0;
}
```

OUTPUT:

Output Clear

```
Enter an infix expression: 5+9/8+3
Postfix Expression: 598/+3+


=== Code Execution Successful ===
```

Q6. Program to implement circular linked list.

<u>PROGRAM:</u>

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

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

struct Node* head = NULL;

void insert(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    struct Node* temp = head;
    newNode->data = data;
    newNode->next = head;

    if (head == NULL) {
        head = newNode;
        newNode->next = head;
    } else {
        while (temp->next != head) {
            temp = temp->next;
        }
        temp->next = newNode;
    }
}

void delete(int key) {
    if (head == NULL) return;

    struct Node *temp = head, *prev = NULL;

    if (temp->data == key) {
        if (temp->next == head) {
            free(temp);
            head = NULL;
        } else {
            while (temp->next != head) {
                temp = temp->next;
            }
```

```c
            temp->next = head->next;
            struct Node* toDelete = head;
            head = head->next;
            free(toDelete);
        }
        return;
    }

    prev = head;
    temp = head->next;

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

    if (temp->data == key) {
        prev->next = temp->next;
        free(temp);
    } else {
        printf("Element not found\n");
    }
}

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

    struct Node* temp = head;
    do {
        printf("%d -> ", temp->data);
        temp = temp->next;
    } while (temp != head);
    printf("(head)\n");
}

int main() {
    int choice, data;

    while (1) {
        printf("\n1. Insert\n2. Delete\n3. Display\n4. Exit\nEnter your choice: ");
        scanf("%d", &choice);
```

```
        switch (choice) {
            case 1:
                printf("Enter data to insert: ");
                scanf("%d", &data);
                insert(data);
                break;
            case 2:
                printf("Enter data to delete: ");
                scanf("%d", &data);
                delete(data);
                break;
            case 3:
                display();
                break;
            case 4:
                exit(0);
            default:
                printf("Invalid choice\n");
        }
    }

    return 0;
}
```

OUTPUT:

Output Clear

```
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 1
Enter data to insert: 3

1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 1
Enter data to insert: 2

1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 3
3 -> 2 -> (head)

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

Q7. Program to implement Doubly Linked List.

Program:

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

struct Node {
    int data;
    struct Node* next;
    struct Node* prev;
};
struct Node* head = NULL;
void insertAtEnd(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    struct Node* temp = head;
    newNode->data = data;
    newNode->next = NULL;
    newNode->prev = NULL;

    if (head == NULL) {
        head = newNode;
        return;
    }

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

    temp->next = newNode;
    newNode->prev = temp;
}
void deleteNode(int key) {
    if (head == NULL) return;

    struct Node* temp = head;

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

```
    }
    while (temp != NULL && temp->data != key) {
        temp = temp->next;
    }
    if (temp == NULL) {
        printf("Element not found\n");
        return;
    }
    if (temp->next != NULL) {
        temp->next->prev = temp->prev;
    }
    if (temp->prev != NULL) {
        temp->prev->next = temp->next;
    }
    free(temp);
}

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

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

    if (head == NULL) return;

    temp = head;
    while (temp->next != NULL) {
        temp = temp->next;
    }

    printf("Backward Traversal: ");
    while (temp != NULL) {
        printf("%d <-> ", temp->data);
        temp = temp->prev;
    }
    printf("NULL\n");
```

```
}
int main() {
    int choice, data;
    while (1) {
        printf("\n1. Insert at End\n2. Delete\n3. Display\n4. Exit\nEnter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Enter data to insert: ");
                scanf("%d", &data);
                insertAtEnd(data);
                break;
            case 2:
                printf("Enter data to delete: ");
                scanf("%d", &data);
                deleteNode(data);
                break;
            case 3:
                display();
                break;
            case 4:
                exit(0);
            default:
                printf("Invalid choice\n");
        }
    }
    return 0;
}
```

OUTPUT:

Output Clear

```
1. Insert at End
2. Delete
3. Display
4. Exit
Enter your choice: 1
Enter data to insert: 5

1. Insert at End
2. Delete
3. Display
4. Exit
Enter your choice: 1
Enter data to insert: 2

1. Insert at End
2. Delete
3. Display
4. Exit
Enter your choice: 3
Forward Traversal: 5 <-> 2 <-> NULL
Backward Traversal: 2 <-> 5 <-> NULL

1. Insert at End
2. Delete
3. Display
4. Exit
Enter your choice:
```

Q8. Program to perform addition and subtraction of two sparse matrices.

PROGRAM

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

struct SparseMatrix {
    int row;
    int col;
    int value;
};

void inputMatrix(struct SparseMatrix mat[], int *nonZeroCount) {
    int rows, cols, count = 0;
    printf("Enter number of rows and columns: ");
    scanf("%d %d", &rows, &cols);

    printf("Enter non-zero elements (row, col, value):\n");
    while (1) {
        int r, c, v;
        printf("Enter (row, col, value) or -1 to stop: ");
        scanf("%d", &r);
        if (r == -1) break;
        scanf("%d %d", &c, &v);

        mat[count].row = r;
        mat[count].col = c;
        mat[count].value = v;
        count++;
    }
    *nonZeroCount = count;
}

void addMatrices(struct SparseMatrix mat1[], int count1, struct SparseMatrix mat2[], int count2,
struct SparseMatrix result[], int *countResult) {
    int i = 0, j = 0, k = 0;

    while (i < count1 && j < count2) {
        if (mat1[i].row == mat2[j].row && mat1[i].col == mat2[j].col) {
            result[k].row = mat1[i].row;
            result[k].col = mat1[i].col;
```

```
            result[k].value = mat1[i].value + mat2[j].value;
            i++;
            j++;
        } else if (mat1[i].row < mat2[j].row || (mat1[i].row == mat2[j].row && mat1[i].col <
mat2[j].col)) {
            result[k] = mat1[i];
            i++;
        } else {
            result[k] = mat2[j];
            j++;
        }
        if (result[k].value != 0) {
            k++;
        }
    }

    while (i < count1) {
        result[k++] = mat1[i++];
    }
    while (j < count2) {
        result[k++] = mat2[j++];
    }

    *countResult = k;
}

void subtractMatrices(struct SparseMatrix mat1[], int count1, struct SparseMatrix mat2[], int
count2, struct SparseMatrix result[], int *countResult) {
    int i = 0, j = 0, k = 0;

    while (i < count1 && j < count2) {
        if (mat1[i].row == mat2[j].row && mat1[i].col == mat2[j].col) {
            result[k].row = mat1[i].row;
            result[k].col = mat1[i].col;
            result[k].value = mat1[i].value - mat2[j].value;
            i++;
            j++;
        } else if (mat1[i].row < mat2[j].row || (mat1[i].row == mat2[j].row && mat1[i].col <
mat2[j].col)) {
            result[k++] = mat1[i++];
        } else {
            result[k] = mat2[j];
            result[k].value = -mat2[j].value;
            j++;
```

```
        }
        if (result[k].value != 0) {
            k++;
        }
    }

    while (i < count1) {
        result[k++] = mat1[i++];
    }
    while (j < count2) {
        result[k] = mat2[j];
        result[k].value = -mat2[j].value;
        j++;
        k++;
    }

    *countResult = k;
}

void displayMatrix(struct SparseMatrix mat[], int count) {
    printf("Resulting Sparse Matrix:\n");
    for (int i = 0; i < count; i++) {
        printf("(%d, %d) = %d\n", mat[i].row, mat[i].col, mat[i].value);
    }
}

int main() {
    struct SparseMatrix mat1[100], mat2[100], result[200];
    int count1 = 0, count2 = 0, countResult = 0;

    printf("Input first sparse matrix:\n");
    inputMatrix(mat1, &count1);

    printf("\nInput second sparse matrix:\n");
    inputMatrix(mat2, &count2);

    printf("\nAddition of matrices:\n");
    addMatrices(mat1, count1, mat2, count2, result, &countResult);
    displayMatrix(result, countResult);

    printf("\nSubtraction of matrices:\n");
    subtractMatrices(mat1, count1, mat2, count2, result, &countResult);
    displayMatrix(result, countResult);
```

```
    return 0;
}
```

OUTPUT:

Output Clear

```
Enter (row, col, value) or -1 to stop: 0 0 8
Enter (row, col, value) or -1 to stop: -1

Input second sparse matrix:
Enter number of rows and columns: 3 3
Enter non-zero elements (row, col, value):
Enter (row, col, value) or -1 to stop: 0 2 0
Enter (row, col, value) or -1 to stop: 4 0 3
Enter (row, col, value) or -1 to stop: 0 0 5
Enter (row, col, value) or -1 to stop: -1

Addition of matrices:
Resulting Sparse Matrix:
(0, 0) = 6
(1, 0) = 5
(0, 0) = 8
(4, 0) = 3
(0, 0) = 5

Subtraction of matrices:
Resulting Sparse Matrix:
(0, 0) = 6
(1, 0) = 5
(1, 0) = 5
(4, 0) = 3
(0, 0) = 8
(4, 0) = -3
(0, 0) = -5
```

<u>Q9.</u> Program to perform multiplication of two sparse matrices.

<u>PROGRAM:</u>

```
#include <stdio.h>
#include <stdlib.h>
struct SparseMatrix {
    int row;
    int col;
    int value;
};

void inputMatrix(struct SparseMatrix mat[], int *nonZeroCount) {
    int rows, cols, count = 0;
    printf("Enter number of rows and columns: ");
    scanf("%d %d", &rows, &cols);

    printf("Enter non-zero elements (row, col, value):\n");
    while (1) {
        int r, c, v;
        printf("Enter (row, col, value) or -1 to stop: ");
        scanf("%d", &r);
        if (r == -1) break;
        scanf("%d %d", &c, &v);
        mat[count].row = r;
        mat[count].col = c;
        mat[count].value = v;
        count++;
    }
    *nonZeroCount = count;
}

void multiplyMatrices(struct SparseMatrix mat1[], int count1, struct SparseMatrix mat2[], int
count2, struct SparseMatrix result[], int *countResult) {
    int k = 0;

    for (int i = 0; i < count1; i++) {
        for (int j = 0; j < count2; j++) {
            if (mat1[i].col == mat2[j].row) {
                result[k].row = mat1[i].row;
                result[k].col = mat2[j].col;
                result[k].value = mat1[i].value * mat2[j].value;
```

```
                k++;
            }
        }
    }

    *countResult = k;
}

void displayMatrix(struct SparseMatrix mat[], int count) {
    printf("Resulting Sparse Matrix:\n");
    for (int i = 0; i < count; i++) {
        printf("(%d, %d) = %d\n", mat[i].row, mat[i].col, mat[i].value);
    }
}
int main() {
    struct SparseMatrix mat1[100], mat2[100], result[100];
    int count1 = 0, count2 = 0, countResult = 0;

    printf("Input first sparse matrix:\n");
    inputMatrix(mat1, &count1);

    printf("\nInput second sparse matrix:\n");
    inputMatrix(mat2, &count2);

    printf("\nMultiplication of matrices:\n");
    multiplyMatrices(mat1, count1, mat2, count2, result, &countResult);
    displayMatrix(result, countResult);
    return 0;
}
```

OUTPUT:

Output Clear

```
Input first sparse matrix:
Enter number of rows and columns: 3 3
Enter non-zero elements (row, col, value):
Enter (row, col, value) or -1 to stop: 0 0 2
Enter (row, col, value) or -1 to stop: 6 0 1
Enter (row, col, value) or -1 to stop: 8 0 0
Enter (row, col, value) or -1 to stop: -1

Input second sparse matrix:
Enter number of rows and columns: 3 3
Enter non-zero elements (row, col, value):
Enter (row, col, value) or -1 to stop: 5 0 0
Enter (row, col, value) or -1 to stop: 9 0 1
Enter (row, col, value) or -1 to stop: 0 2 0
Enter (row, col, value) or -1 to stop: -1

Multiplication of matrices:
Resulting Sparse Matrix:
(0, 2) = 0
(6, 2) = 0
(8, 2) = 0


=== Code Execution Successful ===
```

<u>Q10.</u> Program to perform polynomial arithmetic(add,sub,mul)

PROGRAM:

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

typedef struct Node {
    int coeff, exp;
    struct Node* next;
} Node;

Node* createNode(int c, int e) {
    Node* n = (Node*)malloc(sizeof(Node));
    n->coeff = c; n->exp = e; n->next = NULL;
    return n;
}

void insertTerm(Node** head, int c, int e) {
    Node* n = createNode(c, e);
    if (!*head || (*head)->exp < e) {
        n->next = *head; *head = n;
    } else {
        Node *t = *head;
        while (t->next && t->next->exp > e) t = t->next;
        if (t->next && t->next->exp == e) {
            t->next->coeff += c; free(n);
            if (!t->next->coeff) { Node* del = t->next; t->next = del->next; free(del); }
        } else { n->next = t->next; t->next = n; }
    }
}

Node* combine(Node* p1, Node* p2, int sign) {
    Node* res = NULL;
    while (p1 || p2) {
        if (p1 && (!p2 || p1->exp > p2->exp)) {
            insertTerm(&res, p1->coeff, p1->exp); p1 = p1->next;
        } else if (p2 && (!p1 || p2->exp > p1->exp)) {
            insertTerm(&res, sign * p2->coeff, p2->exp); p2 = p2->next;
        } else {
            insertTerm(&res, p1->coeff + sign * p2->coeff, p1->exp);
            p1 = p1->next; p2 = p2->next;
        }
    }
```

```c
    return res;
}

Node* multiply(Node* p1, Node* p2) {
    Node* res = NULL;
    for (Node* a = p1; a; a = a->next)
        for (Node* b = p2; b; b = b->next)
            insertTerm(&res, a->coeff * b->coeff, a->exp + b->exp);
    return res;
}

void display(Node* h) {
    if (!h) { printf("0\n"); return; }
    while (h) {
        printf("%dx^%d", h->coeff, h->exp);
        if (h->next) printf(" + ");
        h = h->next;
    }
    printf("\n");
}

void readPoly(Node** poly, const char* msg) {
    int c, e;
    printf("%s\n", msg);
    while (1) {
        printf("Enter coeff and exp (-1 to stop): ");
        scanf("%d", &c);
        if (c == -1) break;
        scanf("%d", &e);
        insertTerm(poly, c, e);
    }
}

int main() {
    Node *p1 = NULL, *p2 = NULL, *res = NULL;
    int ch;
    readPoly(&p1, "First polynomial:");
    readPoly(&p2, "Second polynomial:");

    printf("\n1.Add\n2.Subtract\n3.Multiply\n4.Exit\nChoice: ");
    scanf("%d", &ch);

    if (ch == 1) res = combine(p1, p2, 1);
    else if (ch == 2) res = combine(p1, p2, -1);
```

```
    else if (ch == 3) res = multiply(p1, p2);
    else exit(0);

    printf("Result: ");
    display(res);
    return 0;
}
```

OUTPUT:

Output Clear

```
First polynomial:
Enter coeff and exp (-1 to stop): 3
4
Enter coeff and exp (-1 to stop): 2
2
Enter coeff and exp (-1 to stop): -1
Second polynomial:
Enter coeff and exp (-1 to stop): 5
3
Enter coeff and exp (-1 to stop): 2
2
Enter coeff and exp (-1 to stop): -1

1.Add
2.Subtract
3.Multiply
4.Exit
Choice: 3
Result: 15x^7 + 6x^6 + 10x^5 + 4x^4


=== Code Execution Successful ===
```

Q11. Program to perform insertion, deletion and searching of a key in Binary Search Tree.

PROGRAM:

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

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

Node* create(int data) {
    Node* n = malloc(sizeof(Node));
    n->data = data; n->left = n->right = NULL;
    return n;
}

Node* insert(Node* r, int d) {
    if (!r) return create(d);
    if (d < r->data) r->left = insert(r->left, d);
    else if (d > r->data) r->right = insert(r->right, d);
    return r;
}

Node* search(Node* r, int key) {
    if (!r || r->data == key) return r;
    return (key < r->data) ? search(r->left, key) : search(r->right, key);
}

Node* minNode(Node* n) {
    while (n && n->left) n = n->left;
    return n;
}

Node* delete(Node* r, int key) {
    if (!r) return r;
    if (key < r->data) r->left = delete(r->left, key);
    else if (key > r->data) r->right = delete(r->right, key);
    else {
        if (!r->left) { Node* t = r->right; free(r); return t; }
        if (!r->right) { Node* t = r->left; free(r); return t; }
```

```
        Node* t = minNode(r->right);
        r->data = t->data;
        r->right = delete(r->right, t->data);
    }
    return r;
}

void inorder(Node* r) {
    if (r) { inorder(r->left); printf("%d ", r->data); inorder(r->right); }
}

int main() {
    Node* root = NULL; int ch, d;
    while (1) {
        printf("\n1.Insert 2.Search 3.Delete 4.Display 5.Exit: ");
        scanf("%d", &ch);
        if (ch == 5) break;
        printf("Value: "); scanf("%d", &d);
        if (ch == 1) root = insert(root, d);
        else if (ch == 2) printf("%d %sfound\n", d, search(root, d) ? "" : "not ");
        else if (ch == 3) root = delete(root, d);
        else if (ch == 4) { inorder(root); printf("\n"); }
        else printf("Invalid choice\n");
    }
}
```

OUTPUT:

Output Clear

```
1.Insert 2.Search 3.Delete 4.Display 5.Exit: 1
Value: 50

1.Insert 2.Search 3.Delete 4.Display 5.Exit: 1
Value: 6

1.Insert 2.Search 3.Delete 4.Display 5.Exit: 1
Value: 8

1.Insert 2.Search 3.Delete 4.Display 5.Exit: 1
Value: 45

1.Insert 2.Search 3.Delete 4.Display 5.Exit: 2
Value: 45
45 found

1.Insert 2.Search 3.Delete 4.Display 5.Exit: 5


=== Code Execution Successful ===
```

Q12. Program to implement Heap Sort.

PROGRAM:

```
#include <stdio.h>

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) {
        int temp = arr[i];
        arr[i] = arr[largest];
        arr[largest] = temp;
        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--) {
        int temp = arr[0];
        arr[0] = arr[i];
        arr[i] = temp;
        heapify(arr, i, 0);
    }
}

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

```
int main() {
    int n;

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

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

    heapSort(arr, n);

    printf("Sorted array:\n");
    displayArray(arr, n);

    return 0;
}
```

OUTPUT:

Output   Clear

```
Enter number of elements: 5
Enter elements:
2 5 4 9 3
Sorted array:
2 3 4 5 9


=== Code Execution Successful ===
```

Q13. Write a program to perform insertion, deletion and traversal on an AVL Tree.

PROGRAM:

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

typedef struct Node {
    int data, height;
    struct Node *left, *right;
} Node;

int height(Node* n) { return n ? n->height : 0; }
int max(int a, int b) { return (a > b) ? a : b; }

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

Node* rotateRight(Node* y) {
    Node *x = y->left, *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* rotateLeft(Node* x) {
    Node *y = x->right, *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) { return n ? height(n->left) - height(n->right) : 0; }

Node* insert(Node* n, int data) {
    if (!n) return newNode(data);
    if (data < n->data) n->left = insert(n->left, data);
    else if (data > n->data) n->right = insert(n->right, data);
    else return n;
```

```
    n->height = 1 + max(height(n->left), height(n->right));
    int b = getBalance(n);

    if (b > 1 && data < n->left->data) return rotateRight(n);
    if (b < -1 && data > n->right->data) return rotateLeft(n);
    if (b > 1 && data > n->left->data) { n->left = rotateLeft(n->left); return rotateRight(n); }
    if (b < -1 && data < n->right->data) { n->right = rotateRight(n->right); return rotateLeft(n); }

    return n;
}

Node* minValueNode(Node* n) {
    while (n && n->left) n = n->left;
    return n;
}

Node* deleteNode(Node* root, int key) {
    if (!root) 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 || !root->right) {
            Node* temp = root->left ? root->left : root->right;
            if (!temp) { free(root); return NULL; }
            *root = *temp; free(temp);
        } else {
            Node* temp = minValueNode(root->right);
            root->data = temp->data;
            root->right = deleteNode(root->right, temp->data);
        }
    }

    if (!root) return root;

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

    if (b > 1 && getBalance(root->left) >= 0) return rotateRight(root);
    if (b > 1 && getBalance(root->left) < 0) { root->left = rotateLeft(root->left); return
rotateRight(root); }
    if (b < -1 && getBalance(root->right) <= 0) return rotateLeft(root);
    if (b < -1 && getBalance(root->right) > 0) { root->right = rotateRight(root->right); return
rotateLeft(root); }
```

```
    return root;
}

void inOrder(Node* r) { if (r) { inOrder(r->left); printf("%d ", r->data); inOrder(r->right); } }
void preOrder(Node* r) { if (r) { printf("%d ", r->data); preOrder(r->left); preOrder(r->right); } }
void postOrder(Node* r) { if (r) { postOrder(r->left); postOrder(r->right); printf("%d ", r->data); } }
void freeTree(Node* r) { if (r) { freeTree(r->left); freeTree(r->right); free(r); } }

int main() {
    Node* root = NULL;
    int ch, val;
    while (1) {
        printf("\n1.Insert 2.Delete 3.InOrder 4.PreOrder 5.PostOrder 6.Exit\nChoice: ");
        scanf("%d", &ch);
        switch (ch) {
            case 1: printf("Enter value: "); scanf("%d", &val); root = insert(root, val); break;
            case 2: printf("Enter value: "); scanf("%d", &val); root = deleteNode(root, val); 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"); return 0;
            default: printf("Invalid choice\n");
        }
    }
}
```

OUTPUT:

Output Clear

```
1.Insert 2.Delete 3.InOrder 4.PreOrder 5.PostOrder 6.Exit
Choice: 1
Enter value: 10

1.Insert 2.Delete 3.InOrder 4.PreOrder 5.PostOrder 6.Exit
Choice: 1
Enter value: 20

1.Insert 2.Delete 3.InOrder 4.PreOrder 5.PostOrder 6.Exit
Choice: 1
Enter value: 29

1.Insert 2.Delete 3.InOrder 4.PreOrder 5.PostOrder 6.Exit
Choice: 4
PreOrder: 20 10 29

1.Insert 2.Delete 3.InOrder 4.PreOrder 5.PostOrder 6.Exit
Choice: 5
PostOrder: 10 29 20

1.Insert 2.Delete 3.InOrder 4.PreOrder 5.PostOrder 6.Exit
Choice: 6
Exiting.


=== Code Execution Successful ===
```

Q14. Write a program to implement shell sort.

PROGRAM:

```
#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);
    return 0;}
```

OUTPUT:

Output Clear

```
Enter number of elements: 5
Enter 5 integers:
6 8 9 7 3
Original array: 6 8 9 7 3
Sorted array: 3 6 7 8 9


=== Code Execution Successful ===
```

Q15. Write a program to implement Merge Sort.

PROGRAM:

```
#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:

Output   Clear

```
Enter number of elements: 6
Enter 6 integers:
5 2 9 8 7 2
Original array: 5 2 9 8 7 2
Sorted array: 2 2 5 7 8 9


=== Code Execution Successful ===
```

Q16. Write a program to implement Quick Sort.

PROGRAM:

```
#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:

Output Clear

```
Enter number of elements: 5
Enter 5 integers:
9 3 2 1 6
Original array: 9 3 2 1 6
Sorted array: 1 2 3 6 9


=== Code Execution Successful ===
```

Q17. Write a program to implement BFS and DFS.

PROGRAM:

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

#define MAX 100

typedef struct { int items[MAX], front, rear; } Queue;
typedef struct { int items[MAX], top; } Stack;
typedef struct { int v; bool adj[MAX][MAX]; } Graph;

Queue* newQ() { Queue* q = malloc(sizeof(Queue)); q->front = q->rear = -1; return q; }
bool qEmpty(Queue* q) { return q->rear == -1; }
void enQ(Queue* q, int val) {
    if (q->rear < MAX - 1) {
        if (q->front == -1) q->front = 0;
        q->items[++q->rear] = val;
    }
}
int deQ(Queue* q) {
    if (qEmpty(q)) return -1;
    int val = q->items[q->front++];
    if (q->front > q->rear) q->front = q->rear = -1;
    return val;
}

Stack* newS() { Stack* s = malloc(sizeof(Stack)); s->top = -1; return s; }
bool sEmpty(Stack* s) { return s->top == -1; }
void push(Stack* s, int val) { if (s->top < MAX - 1) s->items[++s->top] = val; }
int pop(Stack* s) { return sEmpty(s) ? -1 : s->items[s->top--]; }

Graph* newG(int v) {
    Graph* g = malloc(sizeof(Graph)); g->v = v;
    for (int i = 0; i < v; i++) for (int j = 0; j < v; j++) g->adj[i][j] = false;
    return g;
}
void addEdge(Graph* g, int u, int v) { g->adj[u][v] = g->adj[v][u] = true; }

void BFS(Graph* g, int start) {
    bool vis[MAX] = {0}; Queue* q = newQ();
```

```
    vis[start] = true; enQ(q, start); printf("BFS: ");
    while (!qEmpty(q)) {
        int u = deQ(q); printf("%d ", u);
        for (int i = 0; i < g->v; i++) if (g->adj[u][i] && !vis[i]) vis[i] = true, enQ(q, i);
    }
    free(q); puts("");
}

void DFSRec(Graph* g, int u, bool* vis) {
    vis[u] = true; printf("%d ", u);
    for (int i = 0; i < g->v; i++) if (g->adj[u][i] && !vis[i]) DFSRec(g, i, vis);
}

void DFS(Graph* g, int start) {
    bool vis[MAX] = {0}; printf("DFS (recursive): ");
    DFSRec(g, start, vis); puts("");
}

void DFSIt(Graph* g, int start) {
    bool vis[MAX] = {0}; Stack* s = newS(); push(s, start);
    printf("DFS (iterative): ");
    while (!sEmpty(s)) {
        int u = pop(s);
        if (!vis[u]) {
            vis[u] = true; printf("%d ", u);
            for (int i = g->v - 1; i >= 0; i--) if (g->adj[u][i] && !vis[i]) push(s, i);
        }
    }
    free(s); puts("");
}

int main() {
    Graph* g = newG(7);
    addEdge(g, 0, 1); addEdge(g, 0, 2); addEdge(g, 1, 3);
    addEdge(g, 1, 4); addEdge(g, 2, 5); addEdge(g, 2, 6);
    puts("Graph Traversals\n================");
    BFS(g, 0); DFS(g, 0); DFSIt(g, 0);
    free(g); return 0;
}
```

OUTPUT:

Output Clear

```
Graph Traversals
================
BFS: 0 1 2 3 4 5 6
DFS (recursive): 0 1 3 4 2 5 6
DFS (iterative): 0 1 3 4 2 5 6


=== Code Execution Successful ===
```

Q18. Write a program to perform Hashing using different resolution techniques.

PROGRAM:

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

#define TABLE_SIZE 10
#define EMPTY -1

int hashTable[TABLE_SIZE];

void initializeTable() {
    for (int i = 0; i < TABLE_SIZE; i++)
        hashTable[i] = EMPTY;
}

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

void insertLinearProbing(int key) {
    int index = hash(key), i = 0;
    while (hashTable[(index + i) % TABLE_SIZE] != EMPTY && i < TABLE_SIZE)
        i++;
    if (i < TABLE_SIZE)
        hashTable[(index + i) % TABLE_SIZE] = key;
    else
        printf("Hash table is full! Cannot insert key %d\n", key);
}

void insertQuadraticProbing(int key) {
    int index = hash(key), i = 0;
    while (i < TABLE_SIZE) {
        int newIndex = (index + i * i) % TABLE_SIZE;
        if (hashTable[newIndex] == EMPTY) {
            hashTable[newIndex] = key;
            return;
        }
        i++;
    }
```

```
    printf("Hash table is full! Cannot insert key %d\n", key);
}

void displayTable() {
    printf("\nHash Table:\n");
    for (int i = 0; i < TABLE_SIZE; i++) {
        if (hashTable[i] != EMPTY)
            printf("[%d] => %d\n", i, hashTable[i]);
        else
            printf("[%d] => EMPTY\n", i);
    }
}

int main() {
    int n, key, method;
    printf("Hashing with Collision Resolution\n");
    printf("1. Linear Probing\n2. Quadratic Probing\n");
    printf("Choose a method: ");
    scanf("%d", &method);

    if (method != 1 && method != 2) {
        printf("Invalid method selected!\n");
        return 1;
    }

    initializeTable();

    printf("Enter number of elements to insert: ");
    scanf("%d", &n);

    for (int i = 0; i < n; i++) {
        printf("Enter key %d: ", i + 1);
        scanf("%d", &key);
        if (method == 1)
            insertLinearProbing(key);
        else
            insertQuadraticProbing(key);
    }

    displayTable();
    return 0;
}
```

OUTPUT:

Output Clear

```
Hashing with Collision Resolution
1. Linear Probing
2. Quadratic Probing
Choose a method: 1
Enter number of elements to insert: 5
Enter key 1: 5
Enter key 2: 8
Enter key 3: 7
Enter key 4: 6
Enter key 5: 3

Hash Table:
[0] => EMPTY
[1] => EMPTY
[2] => EMPTY
[3] => 3
[4] => EMPTY
[5] => 5
[6] => 6
[7] => 7
[8] => 8
[9] => EMPTY


=== Code Execution Successful ===
```

Q19. Write a program to implement an algorithm for minimum spanning tree.

PROGRAM:

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

#define MAX 100

struct Edge {
    int u, v, weight;
};

struct Graph {
    int V, E;
    struct Edge edges[MAX];
};

int parent[MAX];

// Find with path compression
int find(int i) {
    if (i != parent[i])
        parent[i] = find(parent[i]);
    return parent[i];
}

void unionSets(int u, int v) {
    int rootU = find(u);
    int rootV = find(v);
    parent[rootU] = rootV;
}

int compareEdges(const void* a, const void* b) {
    struct Edge* e1 = (struct Edge*)a;
    struct Edge* e2 = (struct Edge*)b;
    return e1->weight - e2->weight;
}

void kruskal(struct Graph* graph) {
    int V = graph->V;
    struct Edge result[MAX];
```

```
    int e = 0; // Index for result[]
    int i;

    qsort(graph->edges, graph->E, sizeof(graph->edges[0]), compareEdges);

    for (i = 0; i < V; i++)
        parent[i] = i;

    for (i = 0; i < graph->E && e < V - 1; i++) {
        struct Edge next = graph->edges[i];
        int uRoot = find(next.u);
        int vRoot = find(next.v);

        if (uRoot != vRoot) {
            result[e++] = next;
            unionSets(uRoot, vRoot);
        }
    }

    printf("\nMinimum Spanning Tree (Kruskal's Algorithm):\n");
    int totalWeight = 0;
    for (i = 0; i < e; i++) {
        printf("Edge: %d - %d  Weight: %d\n", result[i].u, result[i].v, result[i].weight);
        totalWeight += result[i].weight;
    }
    printf("Total Weight of MST: %d\n", totalWeight);
}

int main() {
    struct Graph graph;

    printf("Enter number of vertices and edges: ");
    scanf("%d %d", &graph.V, &graph.E);

    printf("Enter edges (u v weight):\n");
    for (int i = 0; i < graph.E; i++) {
        scanf("%d %d %d", &graph.edges[i].u, &graph.edges[i].v, &graph.edges[i].weight);
    }

    kruskal(&graph);
    return 0;
}
```

OUTPUT:

Output Clear

```
Enter number of vertices and edges: 5 7
Enter edges (u v weight):
0 1 2
0 3 6
1 2 3
1 3 8
1 4 5
2 4 7
3 4 9

Minimum Spanning Tree (Kruskal's Algorithm):
Edge: 0 - 1  Weight: 2
Edge: 1 - 2  Weight: 3
Edge: 1 - 4  Weight: 5
Edge: 0 - 3  Weight: 6
Total Weight of MST: 16


=== Code Execution Successful ===
```

Q20. Write a program to implement the shortest path algorithm.

PROGRAM:

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

#define V 5

int minDistance(int dist[], int sptSet[]) {
    int min = INT_MAX, min_index = -1;
    for (int v = 0; v < V; v++)
        if (!sptSet[v] && dist[v] <= min) {
            min = dist[v];
            min_index = v;
        }
    return min_index;
}

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

void dijkstra(int graph[V][V], int src) {
    int dist[V];
    int sptSet[V];
    for (int i = 0; i < V; i++)
        dist[i] = INT_MAX, sptSet[i] = 0;

    dist[src] = 0;

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

        for (int v = 0; v < V; v++)
            if (!sptSet[v] && graph[u][v] &&
                dist[u] != INT_MAX &&
                dist[u] + graph[u][v] < dist[v])
                dist[v] = dist[u] + graph[u][v];
```

```
    }

    printSolution(dist);
}

int main() {
    int graph[V][V] = {
        {0, 6, 0, 1, 0},
        {6, 0, 5, 2, 2},
        {0, 5, 0, 0, 5},
        {1, 2, 0, 0, 1},
        {0, 2, 5, 1, 0}
    };

    int source = 0;
    dijkstra(graph, source);

    return 0;
}
```

OUTPUT:

Output

```
Vertex    Distance from Source
0         0
1         3
2         7
3         1
4         2


=== Code Execution Successful ===
```