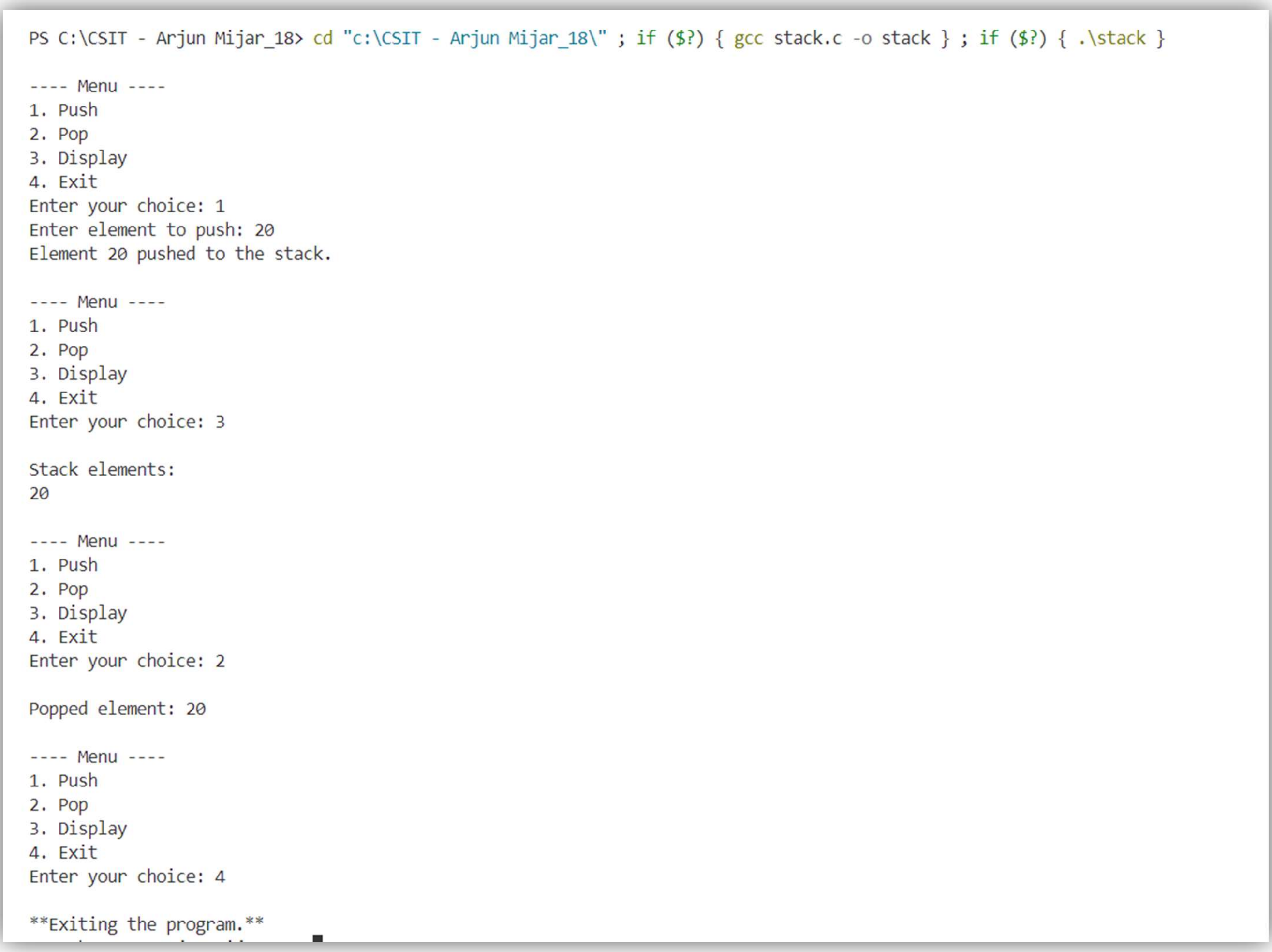
1. Menu program implementation: Stack operation using Array

```
#include <stdio.h>

#define SIZE 100

int stack[SIZE], top = -1;

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

int main() {
    int choice, value;

    do {
        printf("\n---- Menu ----\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 element to push: ");
                scanf("%d", &value);
                push(value);
                break;

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

            case 3:
                display();
                break;

            case 4:
                printf("**Exiting the program.**\n");
                break;

            default:
                printf("Invalid choice. Please enter a valid option.\n");
        }
    } while (choice != 4);

    return 0;
}

void push(int value) {
    if (top == SIZE - 1) {
        printf("Stack Overflow. Cannot push element.\n");
    } else {
        top++;
        stack[top] = value;
        printf("Element %d pushed to the stack.\n", value);
    }
}
```

```
int pop() {
    int value = -1; // Default value for an empty stack

    if (top == -1) {
        printf("Stack Underflow. Cannot pop element.\n");
    } else {
        value = stack[top];
        top--;
    }

    return value;
}

void display() {
    if (top == -1) {
        printf("Stack is empty.\n");
    } else {
        printf("Stack elements:\n");
        for (int i = top; i >= 0; i--) {
            printf("%d\n", stack[i]);
        }
    }
}
```



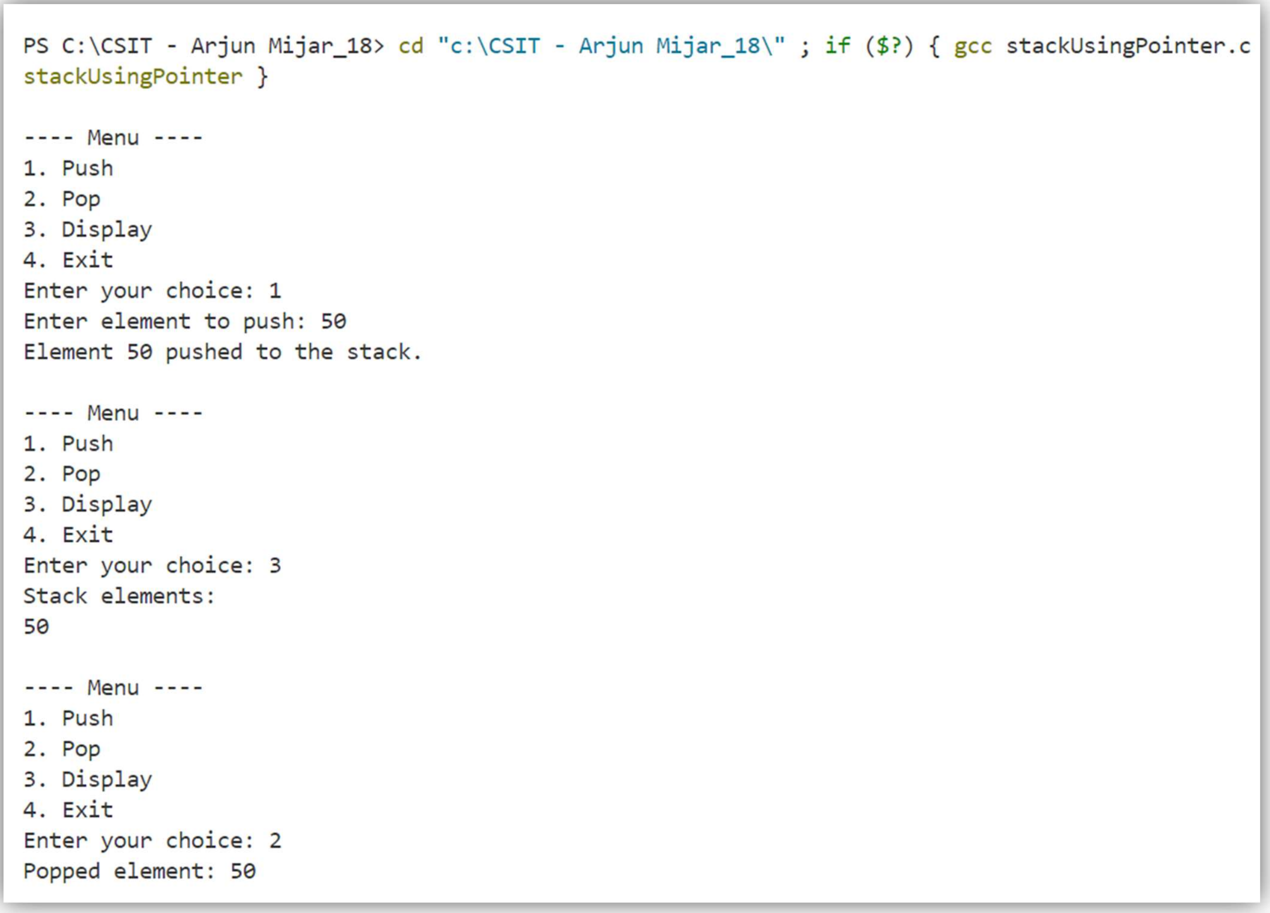
```
PS C:\CSIT - Arjun Mijar_18> cd "c:\CSIT - Arjun Mijar_18\" ; if ($?) { gcc stack.c -o stack } ; if ($?) { .\stack }

---- Menu ----
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 1
Enter element to push: 20
Element 20 pushed to the stack.

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

Stack elements:
20

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

Popped element: 20

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

**Exiting the program.**
```

Figure: Output

2. Menu program implementation: Stack operation using Pointer

```
#include <stdio.h>

#define SIZE 100

int stack[SIZE], *top = NULL;

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

int main() {
    int choice, value;

    top = stack; // Initialize top to the beginning of the stack array

    do {
        printf("\n---- Menu ----\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 element to push: ");
                scanf("%d", &value);
                push(value);
                break;

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

            case 3:
                display();
                break;

            case 4:
                printf("Exiting the program.\n");
                break;

            default:
                printf("Invalid choice. Please enter a valid option.\n");
        }
    } while (choice != 4);

    return 0;
}

void push(int value) {
    if (top == stack + SIZE - 1) {
        printf("Stack Overflow. Cannot push element.\n");
    } else {
        *top = value;
        top++;
        printf("Element %d pushed to the stack.\n", value);
    }
}
```

```
int pop() {
    int value = -1; // Default value for an empty stack

    if (top == stack) {
        printf("Stack Underflow. Cannot pop element.\n");
    } else {
        top--;
        value = *top;
    }

    return value;
}

void display() {
    if (top == stack) {
        printf("Stack is empty.\n");
    } else {
        printf("Stack elements:\n");
        for (int *ptr = top - 1; ptr >= stack; ptr--) {
            printf("%d\n", *ptr);
        }
    }
}
```

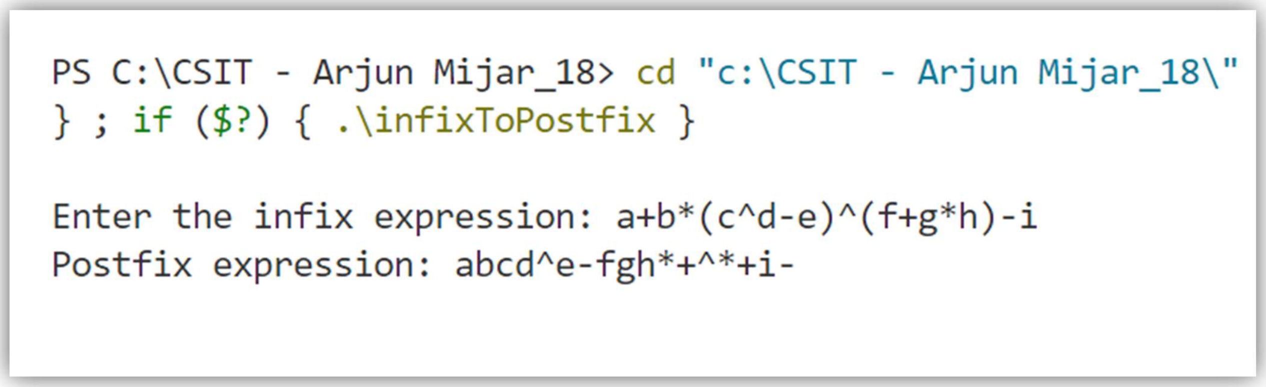


```
PS C:\CSIT - Arjun Mijar_18> cd "c:\CSIT - Arjun Mijar_18\" ; if ($?) { gcc stackUsingPointer.c
stackUsingPointer }

---- Menu ----
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 1
Enter element to push: 50
Element 50 pushed to the stack.

---- Menu ----
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 3
Stack elements:
50

---- Menu ----
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 2
Popped element: 50
```

Figure: Output

3. Write a program to convert Infix Expression into Postfix Expression

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

// Function to return precedence of operators
int prec(char c) {
        if (c == '^')
                return 3;
        else if (c == '/' || c == '*')
                return 2;
        else if (c == '+' || c == '-')
                return 1;
        else
                return -1;
}

// Function to return associativity of operators
char associativity(char c) {
        if (c == '^')
                return 'R';
        return 'L'; // Default to left-associative
}

// The main function to convert infix expression to postfix expression
void infixToPostfix(char s[]) {
        char result[1000];
        int resultIndex = 0;
        int len = strlen(s);
        char stack[1000];
        int stackIndex = -1;

        for (int i = 0; i < len; i++) {
                char c = s[i];

                // If the scanned character is an operand, add it to the output string.
                if ((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z') || (c >= '0' && c <= '9')) {
                        result[resultIndex++] = c;
                }
                // If the scanned character is an '(', push it to the stack.
                else if (c == '(') {
                        stack[++stackIndex] = c;
                }
                // If the scanned character is an ')', pop and add to the output string from the stack
                // until an '(' is encountered.
                else if (c == ')') {
                        while (stackIndex >= 0 && stack[stackIndex] != '(') {
                                result[resultIndex++] = stack[stackIndex--];
                        }
                        stackIndex--; // Pop '('
                }
                // If an operator is scanned
                else {
                        while (stackIndex >= 0 && (prec(s[i]) < prec(stack[stackIndex]) ||
                                                                        prec(s[i]) == prec(stack[stackIndex])
&&
                                                                                associativity(s[i]) == 'L')) {
                                result[resultIndex++] = stack[stackIndex--];
                        }
                        stack[++stackIndex] = c;
                }
        }
        // Pop all the remaining elements from the stack
        while (stackIndex >= 0) {
                result[resultIndex++] = stack[stackIndex--];
        }

        result[resultIndex] = '\0';
        printf("%s\n", result);
}

// Driver code
int main() {
        char exp[1000];
        printf("Enter the infix expression: ");
        fgets(exp, sizeof(exp), stdin);
        // Remove the newline character if present
        exp[strcspn(exp, "\n")] = '\0';
        // Function call
        infixToPostfix(exp);

        return 0;
}
```



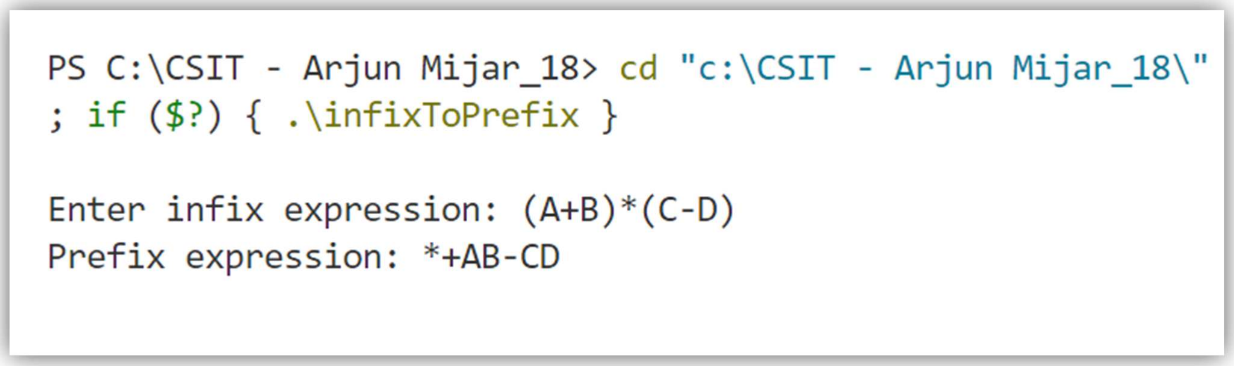
```
PS C:\CSIT - Arjun Mijar_18> cd "c:\CSIT - Arjun Mijar_18\"
} ; if ($?) { .\infixToPostfix }

Enter the infix expression: a+b*(c^d-e)^(f+g*h)-i
Postfix expression: abcd^e-fgh*+^*+i-
```

Figure: Output

4. Write a program to convert Infix Expression into Prefix Expression.

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

#define MAX_SIZE 100

// Structure to represent stack
typedef struct
{
  char items[MAX_SIZE];
  int top;
} Stack;

// Function to initialize the stack
void initialize(Stack *s)
{
  s->top = -1;
}

// Function to push an item onto the stack
void push(Stack *s, char c)
{
  if (s->top == MAX_SIZE - 1)
  {
    printf("Stack Overflow\n");
    exit(EXIT_FAILURE);
  }
  s->items[++(s->top)] = c;
}

// Function to pop an item from the stack
char pop(Stack *s)
{
  if (s->top == -1)
  {
    printf("Stack Underflow\n");
    exit(EXIT_FAILURE);
  }
  return s->items[(s->top)--];
}

// Function to check if a character is an operator
int isOperator(char c)
{
  return (c == '+' || c == '-' || c == '*' || c == '/');
}

// Function to return the precedence of an operator
int precedence(char c)
{
  if (c == '+' || c == '-')
    return 1;
  else if (c == '*' || c == '/')
    return 2;
  return 0;
}

// Function to convert infix expression to prefix expression
void infixToPrefix(char infix[], char prefix[])
```

```
{
    int i, j;
    Stack s;
    initialize(&s);

    // Reverse the infix expression
    int len = strlen(infix);
    for (i = len - 1, j = 0; i >= 0; i--)
    {
        if (infix[i] == '(')
            infix[i] = ')';
        else if (infix[i] == ')')
            infix[i] = '(';

        prefix[j++] = infix[i];
    }
    prefix[j] = '\0';

    // Apply algorithm for postfix
    for (i = 0, j = 0; prefix[i] != '\0'; i++)
    {
        if (isalnum(prefix[i]))
        {
            prefix[j++] = prefix[i];
        }
        else if (prefix[i] == '(')
        {
            push(&s, '(');
        }
        else if (prefix[i] == ')')
        {
            while (s.top != -1 && s.items[s.top] != '(')
            {
                prefix[j++] = pop(&s);
            }
            if (s.top == -1)
            {
                printf("Invalid expression\n");
                exit(EXIT_FAILURE);
            }
            pop(&s); // Discard '('
        }
        else
        {
            while (s.top != -1 && precedence(prefix[i]) <= precedence(s.items[s.top]))
            {
                prefix[j++] = pop(&s);
            }
            push(&s, prefix[i]);
        }
    }

    // Pop any remaining operators from the stack
    while (s.top != -1)
    {
        if (s.items[s.top] == '(')
        {
            printf("Invalid expression\n");
            exit(EXIT_FAILURE);
        }
        prefix[j++] = pop(&s);
    }
    // Null-terminate the prefix expression
    prefix[j] = '\0';

    // Reverse the prefix expression to get the final result
    for (i = 0, j = strlen(prefix) - 1; i < j; i++, j--)
    {
        char temp = prefix[i];
        prefix[i] = prefix[j];
        prefix[j] = temp;
    }
}

int main()
{
    char infix[MAX_SIZE];
    char prefix[MAX_SIZE];

    printf("Enter infix expression: ");
    scanf("%s", infix);

    infixToPrefix(infix, prefix);
    printf("Prefix expression: %s\n", prefix);

    return 0;
}
```

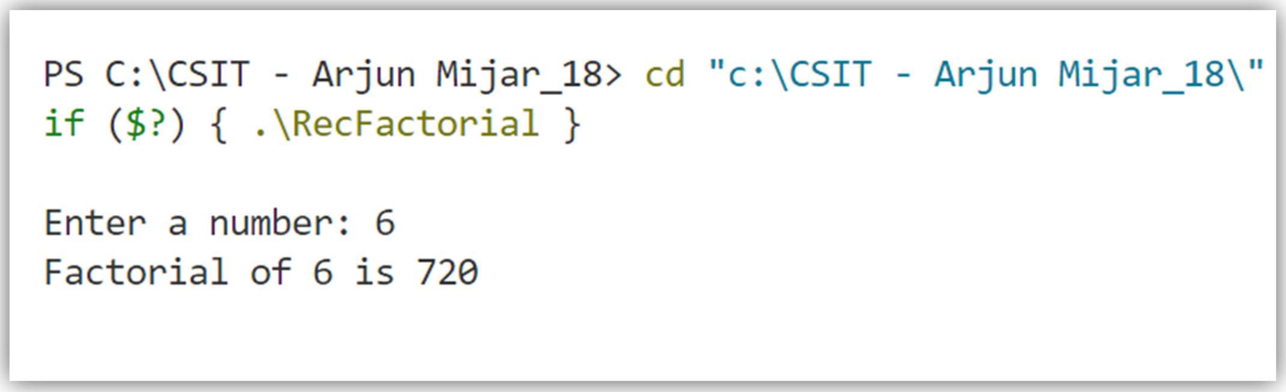


```
PS C:\CSIT - Arjun Mijar_18> cd "c:\CSIT - Arjun Mijar_18\"
; if ($?) { .\infixToPrefix }

Enter infix expression: (A+B)*(C-D)
Prefix expression: *+AB-CD
```

Figure: Output

5. Write a recursive program to find the factorial value of given number.

```
#include <stdio.h>

// Recursive function to calculate factorial
unsigned long long factorial(int n) {
    // Base case: factorial of 0 is 1
    if (n == 0) {
        return 1;
    }
    // Recursive case: n! = n * (n-1)!
    else {
        return n * factorial(n - 1);
    }
}

int main() {
    int num;
    printf("Enter a number: ");
    scanf("%d", &num);

    // Check for negative input
    if (num < 0) {
        printf("Factorial is not defined for negative numbers.\n");
    } else {
        unsigned long long fact = factorial(num);
        printf("Factorial of %d is %llu\n", num, fact);
    }

    return 0;
}
```

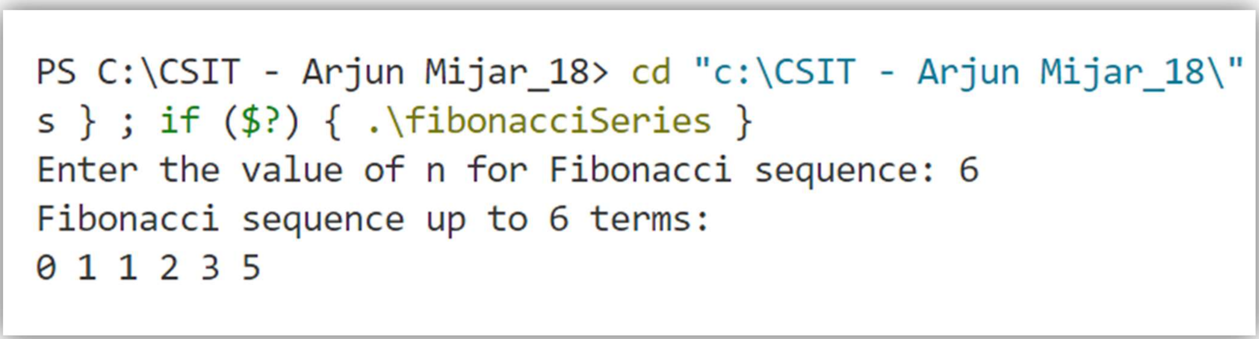


```
PS C:\CSIT - Arjun Mijar_18> cd "c:\CSIT - Arjun Mijar_18\"
if ($?) { .\RecFactorial }

Enter a number: 6
Factorial of 6 is 720
```

Figure: Output

6. Write a recursive program to find a Fibonacci sequence.

```
#include <stdio.h>

// Recursive function to find the nth Fibonacci number
int fibonacci(int n) {
    // Base cases:
    // If n is 0 or 1, return n
    if (n == 0 || n == 1) {
        return n;
    }
    // Recursive case:
    // Fibonacci(n) = Fibonacci(n-1) + Fibonacci(n-2)
    else {
        return fibonacci(n - 1) + fibonacci(n - 2);
    }
}

int main() {
    int n;
    printf("Enter the value of n for Fibonacci sequence: ");
    scanf("%d", &n);

    if (n < 0) {
        printf("Fibonacci sequence is not defined for negative numbers.\n");
    } else {
        printf("Fibonacci sequence up to %d terms:\n", n);
        for (int i = 0; i < n; i++) {
            printf("%d ", fibonacci(i));
        }
        printf("\n");
    }

    return 0;
}
```

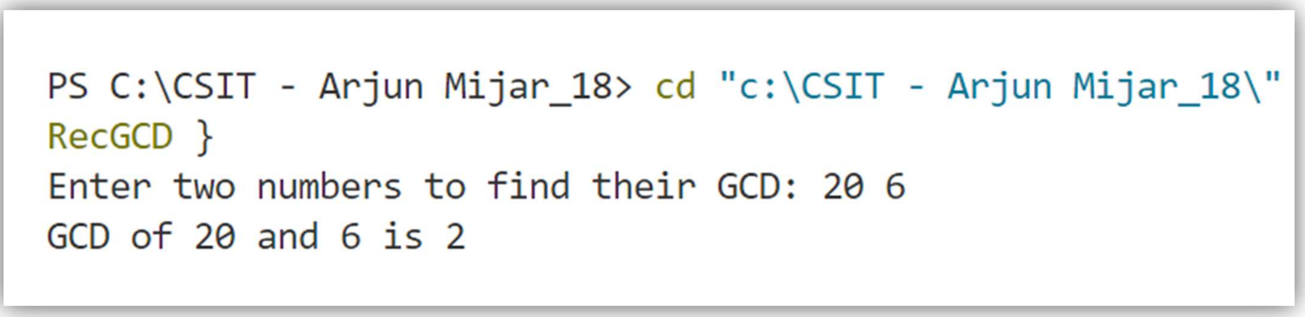


```
PS C:\CSIT - Arjun Mijar_18> cd "c:\CSIT - Arjun Mijar_18\"
s } ; if ($?) { .\fibonacciSeries }
Enter the value of n for Fibonacci sequence: 6
Fibonacci sequence up to 6 terms:
0 1 1 2 3 5
```

Figure: Output

7. Write a recursive program to find GCD of two integers.

```
#include <stdio.h>

// Recursive function to find GCD of two numbers
int gcd(int a, int b) {
    // Base case: If b is 0, return a
    if (b == 0) {
        return a;
    }
    // Recursive case: GCD(a, b) = GCD(b, a % b)
    else {
        return gcd(b, a % b);
    }
}

int main() {
    int num1, num2;
    printf("Enter two numbers to find their GCD: ");
    scanf("%d %d", &num1, &num2);

    // Check for non-positive input
    if (num1 <= 0 || num2 <= 0) {
        printf("GCD is not defined for non-positive numbers.\n");
    } else {
        int result = gcd(num1, num2);
        printf("GCD of %d and %d is %d\n", num1, num2, result);
    }

    return 0;
}
```

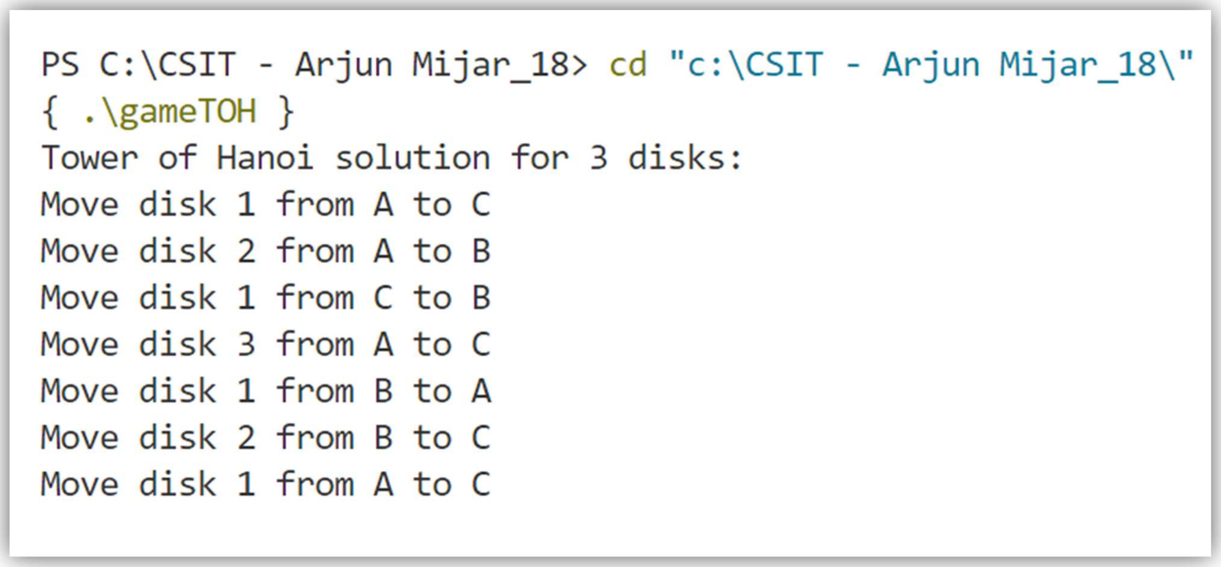


```
PS C:\CSIT - Arjun Mijar_18> cd "c:\CSIT - Arjun Mijar_18\"
RecGCD }
Enter two numbers to find their GCD: 20 6
GCD of 20 and 6 is 2
```

Figure: Output

8. Write a recursive program to implement TOH problem. (Show the output for 3 disks)

```
#include <stdio.h>

// Function to move a disk from source pole to destination pole
void moveDisk(int n, char source, char destination) {
   printf("Move disk %d from %c to %c\n", n, source, destination);
}

// Recursive function to solve Tower of Hanoi problem
void towerOfHanoi(int n, char source, char auxiliary, char destination) {
   if (n == 1) {
      moveDisk(n, source, destination);
      return;
   }
   towerOfHanoi(n - 1, source, destination, auxiliary);
   moveDisk(n, source, destination);
   towerOfHanoi(n - 1, auxiliary, source, destination);
}

int main() {
  int num_disks;
   printf("Enter the number of disks: ");
   scanf("%d", &num_disks);
   printf("Tower of Hanoi solution for %d disks:\n", num_disks);
   towerOfHanoi(num_disks, 'A', 'B', 'C');
   return 0;
}
```



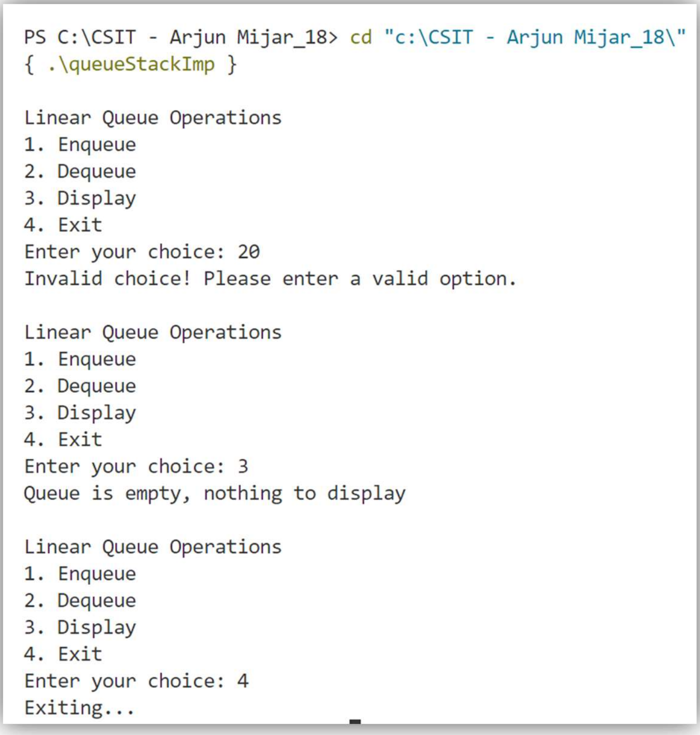
```
PS C:\CSIT - Arjun Mijar_18> cd "c:\CSIT - Arjun Mijar_18\"
{ .\gameTOH }
Tower of Hanoi solution for 3 disks:
Move disk 1 from A to C
Move disk 2 from A to B
Move disk 1 from C to B
Move disk 3 from A to C
Move disk 1 from B to A
Move disk 2 from B to C
Move disk 1 from A to C
```

Figure: Output

9. Write a menu driven program to illustrate basic operations of Linear queue using array implementation and pointer implementation.
a) Enqueue
b) Dequeue
c) Display

Stack Implementation:

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

#define MAX_SIZE 100

// Structure to represent a queue
typedef struct {
    int items[MAX_SIZE];
    int front;
    int rear;
} Queue;

// Function prototypes
void enqueue(Queue *q, int value);
int dequeue(Queue *q);
void display(Queue *q);

int main() {
    Queue q;
    q.front = -1;
    q.rear = -1;
    int choice, value;

    do {
        printf("\nLinear Queue 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 the value to enqueue: ");
                scanf("%d", &value);
                enqueue(&q, value);
                break;
            case 2:
                if (q.front == -1) {
                    printf("Queue is empty, cannot dequeue\n");
                } else {
                    printf("Dequeued element: %d\n", dequeue(&q));
                }
                break;
            case 3:
                display(&q);
                break;
            case 4:
                printf("Exiting...\n");
                exit(0);
            default:
                printf("Invalid choice! Please enter a valid option.\n");
        }
    } while (choice != 4);
    return 0;
}

// Function to enqueue a value into the queue
void enqueue(Queue *q, int value) {
    if (q->rear == MAX_SIZE - 1) {
        printf("Queue is full, cannot enqueue\n");
    } else {
        if (q->front == -1) {
            q->front = 0; // Set front to 0 for the first element
        }
        q->rear++;
        q->items[q->rear] = value;
        printf("Enqueued %d\n", value);
    }
}

// Function to dequeue a value from the queue
int dequeue(Queue *q) {
    int item;
    if (q->front == -1) {
        printf("Queue is empty, cannot dequeue\n");
        return -1;
    } else {
        item = q->items[q->front];
        if (q->front == q->rear) {
            q->front = -1;
            q->rear = -1;
        } else {
            q->front++;
        }
        return item;
    }
}

// Function to display all values in the queue
void display(Queue *q) {
    if (q->front == -1) {
        printf("Queue is empty, nothing to display\n");
    } else {
        printf("Elements in the queue: ");
        for (int i = q->front; i <= q->rear; i++) {
            printf("%d ", q->items[i]);
        }
        printf("\n");
    }
}
```



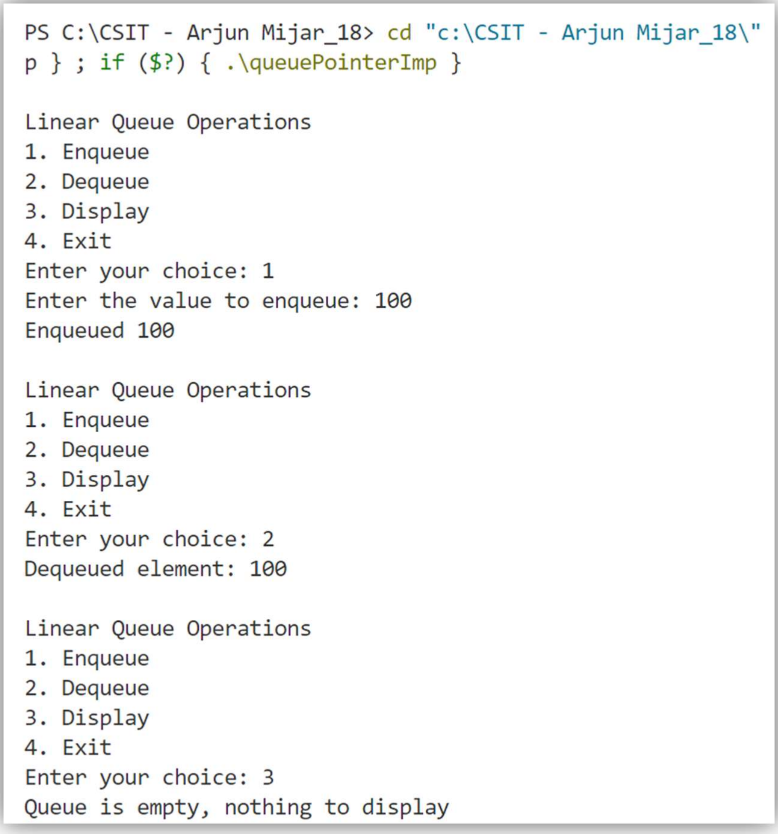
```
PS C:\CSIT - Arjun Mijar_18> cd "c:\CSIT - Arjun Mijar_18\"
{ .\queueStackImp }

Linear Queue Operations
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 20
Invalid choice! Please enter a valid option.

Linear Queue Operations
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 3
Queue is empty, nothing to display

Linear Queue Operations
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 4
Exiting...
```

Figure: Output

Pointer Implementation:

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

#define MAX_SIZE 100

// Structure to represent a node in the queue
typedef struct Node {
    int data;
    struct Node *next;
} Node;

// Structure to represent a queue
typedef struct {
    Node *front;
    Node *rear;
} Queue;

// Function prototypes
void enqueue(Queue *q, int value);
int dequeue(Queue *q);
void display(Queue *q);

int main() {
    Queue q;
    q.front = NULL;
    q.rear = NULL;
    int choice, value;

    do {
        printf("\nLinear Queue 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 the value to enqueue: ");
                scanf("%d", &value);
                enqueue(&q, value);
                break;
            case 2:
                value = dequeue(&q);
                if (value != -1) {
                    printf("Dequeued element: %d\n", value);
                }
                break;
            case 3:
                display(&q);
                break;
            case 4:
                printf("Exiting...\n");
                exit(0);
            default:
                printf("Invalid choice! Please enter a valid option.\n");
        }
    } while (choice != 4);

    return 0;
}
```

```
// Function to enqueue a value into the queue
void enqueue(Queue *q, int value) {
    Node *newNode = (Node *)malloc(sizeof(Node));
    if (newNode == NULL) {
        printf("Memory allocation failed\n");
        return;
    }
    newNode->data = value;
    newNode->next = NULL;

    if (q->rear == NULL) {
        q->front = newNode;
        q->rear = newNode;
    } else {
        q->rear->next = newNode;
        q->rear = newNode;
    }

    printf("Enqueued %d\n", value);
}

// Function to dequeue a value from the queue
int dequeue(Queue *q) {
    if (q->front == NULL) {
        printf("Queue is empty, cannot dequeue\n");
        return -1;
    }

    int value = q->front->data;
    Node *temp = q->front;
    q->front = q->front->next;
    free(temp);

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

    return value;
}

// Function to display all values in the queue
void display(Queue *q) {
    if (q->front == NULL) {
        printf("Queue is empty, nothing to display\n");
    } else {
        printf("Elements in the queue: ");
        Node *current = q->front;
        while (current != NULL) {
            printf("%d ", current->data);
            current = current->next;
        }
        printf("\n");
    }
}
```



```
PS C:\CSIT - Arjun Mijar_18> cd "c:\CSIT - Arjun Mijar_18\"
p } ; if ($?) { .\queuePointerImp }

Linear Queue Operations
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 1
Enter the value to enqueue: 100
Enqueued 100

Linear Queue Operations
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 2
Dequeued element: 100

Linear Queue Operations
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 3
Queue is empty, nothing to display
```

Figure: Output

10. Write a menu driven program to illustrate basic operations of circular queue having following menu:
a) Enqueue
b) Dequeue
c) Traverse
d) Exit

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

#define MAX_SIZE 5

// Structure to represent a circular queue
typedef struct {
    int items[MAX_SIZE];
    int front, rear;
} CircularQueue;

// Function prototypes
void enqueue(CircularQueue *cq, int value);
int dequeue(CircularQueue *cq);
void traverse(CircularQueue *cq);

int main() {
    CircularQueue cq;
    cq.front = -1;
    cq.rear = -1;
    int choice, value;

    do {
        printf("\nCircular Queue Operations\n");
        printf("a) Enqueue\n");
        printf("b) Dequeue\n");
        printf("c) Traverse\n");
        printf("d) Exit\n");
        printf("Enter your choice: ");
        scanf(" %c", &choice);

        switch (choice) {
            case 'a':
                printf("Enter the value to enqueue: ");
                scanf("%d", &value);
                enqueue(&cq, value);
                break;
            case 'b':
                value = dequeue(&cq);
                if (value != -1) {
                    printf("Dequeued element: %d\n", value);
                }
                break;
            case 'c':
                traverse(&cq);
                break;
            case 'd':
                printf("Exiting...\n");
                exit(0);
            default:
                printf("Invalid choice! Please enter a valid option.\n");
        }
    } while (choice != 'd');

    return 0;
}
```

```
// Function to check if the queue is full
int isFull(CircularQueue *cq) {
    return (cq->front == (cq->rear + 1) % MAX_SIZE);
}

// Function to check if the queue is empty
int isEmpty(CircularQueue *cq) {
    return (cq->front == -1);
}

// Function to enqueue a value into the circular queue
void enqueue(CircularQueue *cq, int value) {
    if (isFull(cq)) {
        printf("Queue is full, cannot enqueue\n");
    } else {
        if (cq->front == -1) {
            cq->front = 0;
        }
        cq->rear = (cq->rear + 1) % MAX_SIZE;
        cq->items[cq->rear] = value;
        printf("Enqueued %d\n", value);
    }
}

// Function to dequeue a value from the circular queue
int dequeue(CircularQueue *cq) {
    int item;
    if (isEmpty(cq)) {
        printf("Queue is empty, cannot dequeue\n");
        return -1;
    } else {
        item = cq->items[cq->front];
        if (cq->front == cq->rear) {
            cq->front = -1;
            cq->rear = -1;
        } else {
            cq->front = (cq->front + 1) % MAX_SIZE;
        }
        return item;
    }
}

// Function to traverse and display
void traverse(CircularQueue *cq) {
    if (isEmpty(cq)) {
        printf("Queue is empty, nothing to display\n");
    } else {
        printf("Elements in the circular queue: ");
        int i = cq->front;
        while (i != cq->rear) {
            printf("%d ", cq->items[i]);
            i = (i + 1) % MAX_SIZE;
        }
        printf("%d\n", cq->items[i]);
    }
}
```

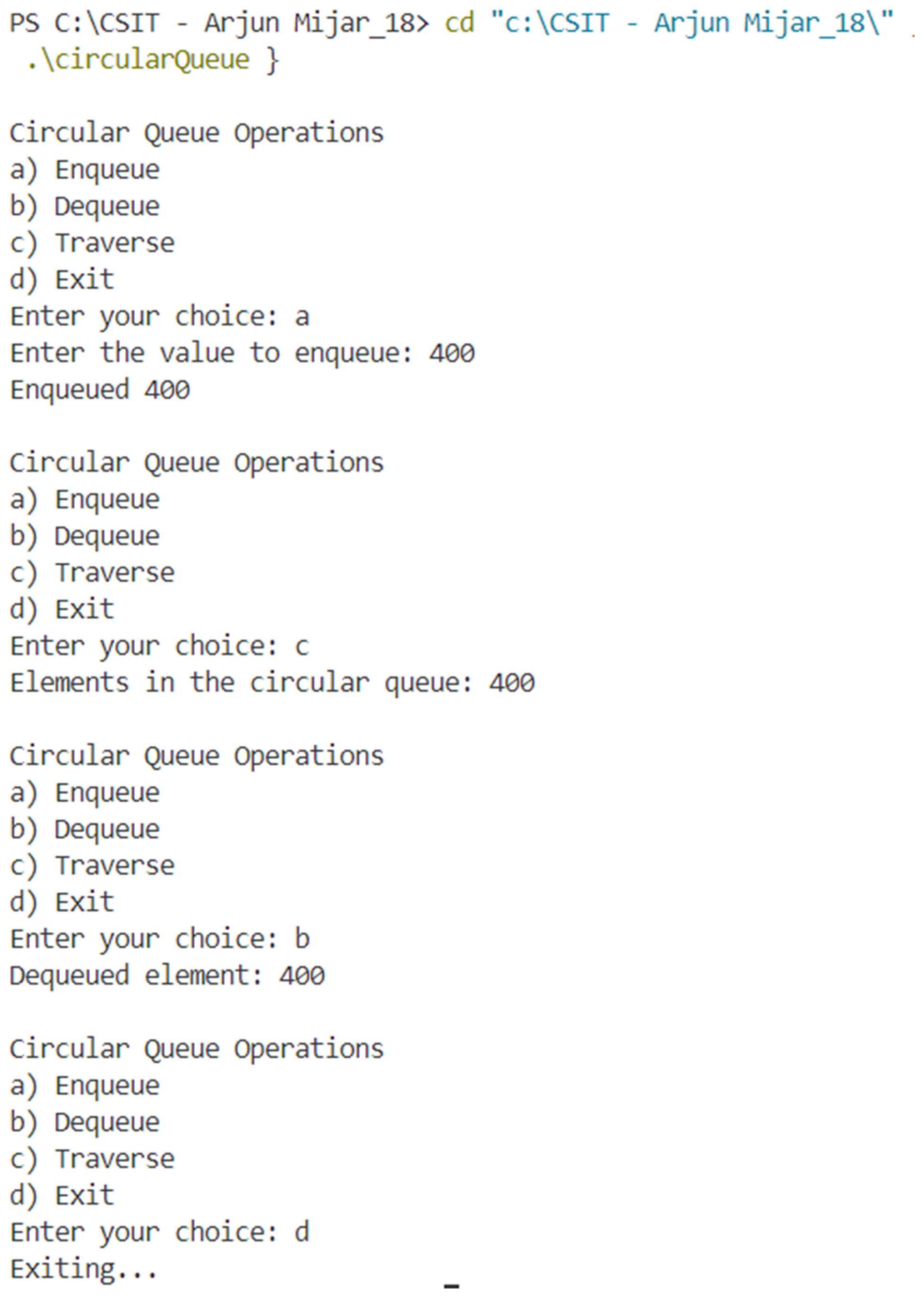
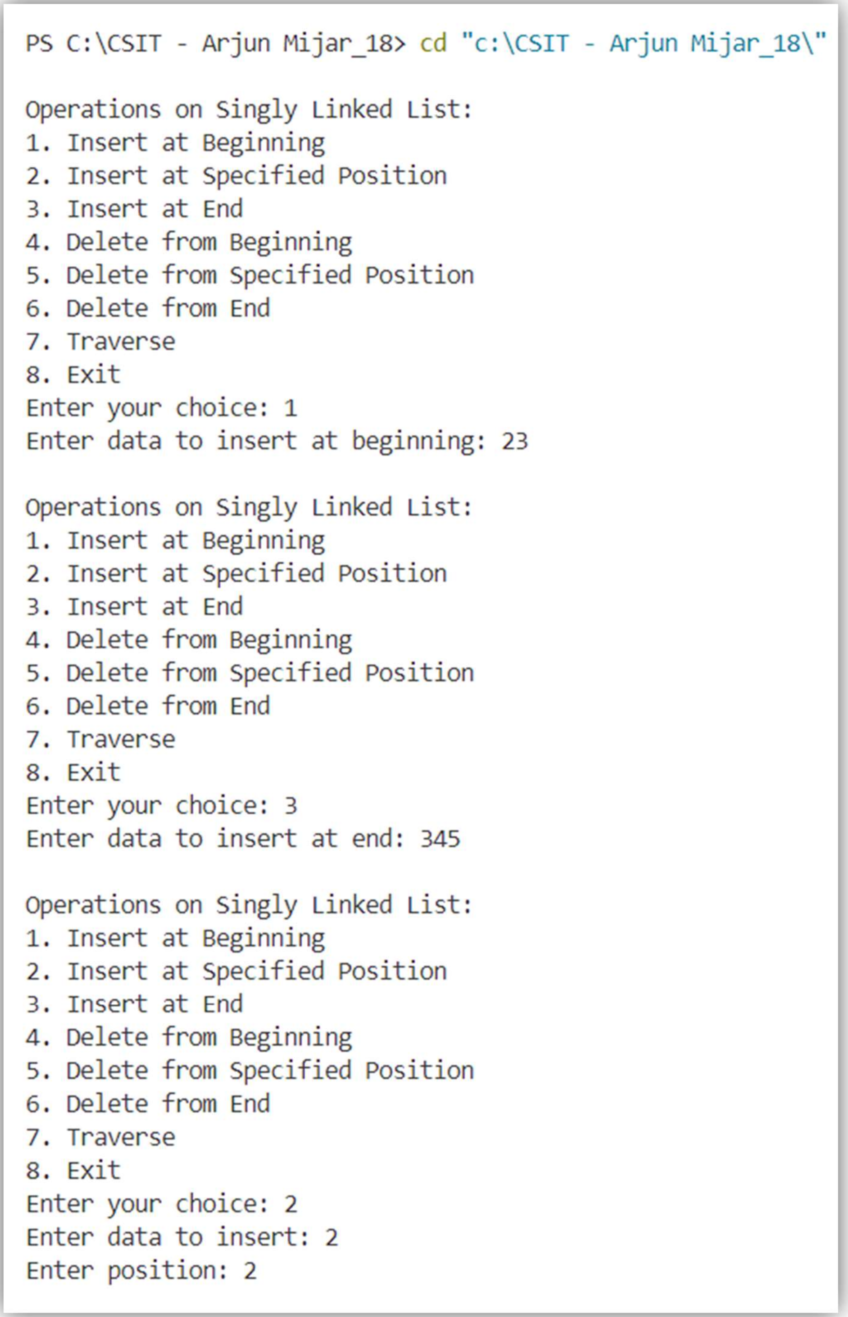

```
PS C:\CSIT - Arjun Mijar_18> cd "c:\CSIT - Arjun Mijar_18\" .
 .\circularQueue }

Circular Queue Operations
a) Enqueue
b) Dequeue
c) Traverse
d) Exit
Enter your choice: a
Enter the value to enqueue: 400
Enqueued 400

Circular Queue Operations
a) Enqueue
b) Dequeue
c) Traverse
d) Exit
Enter your choice: c
Elements in the circular queue: 400

Circular Queue Operations
a) Enqueue
b) Dequeue
c) Traverse
d) Exit
Enter your choice: b
Dequeued element: 400

Circular Queue Operations
a) Enqueue
b) Dequeue
c) Traverse
d) Exit
Enter your choice: d
Exiting...
```

Figure: Output

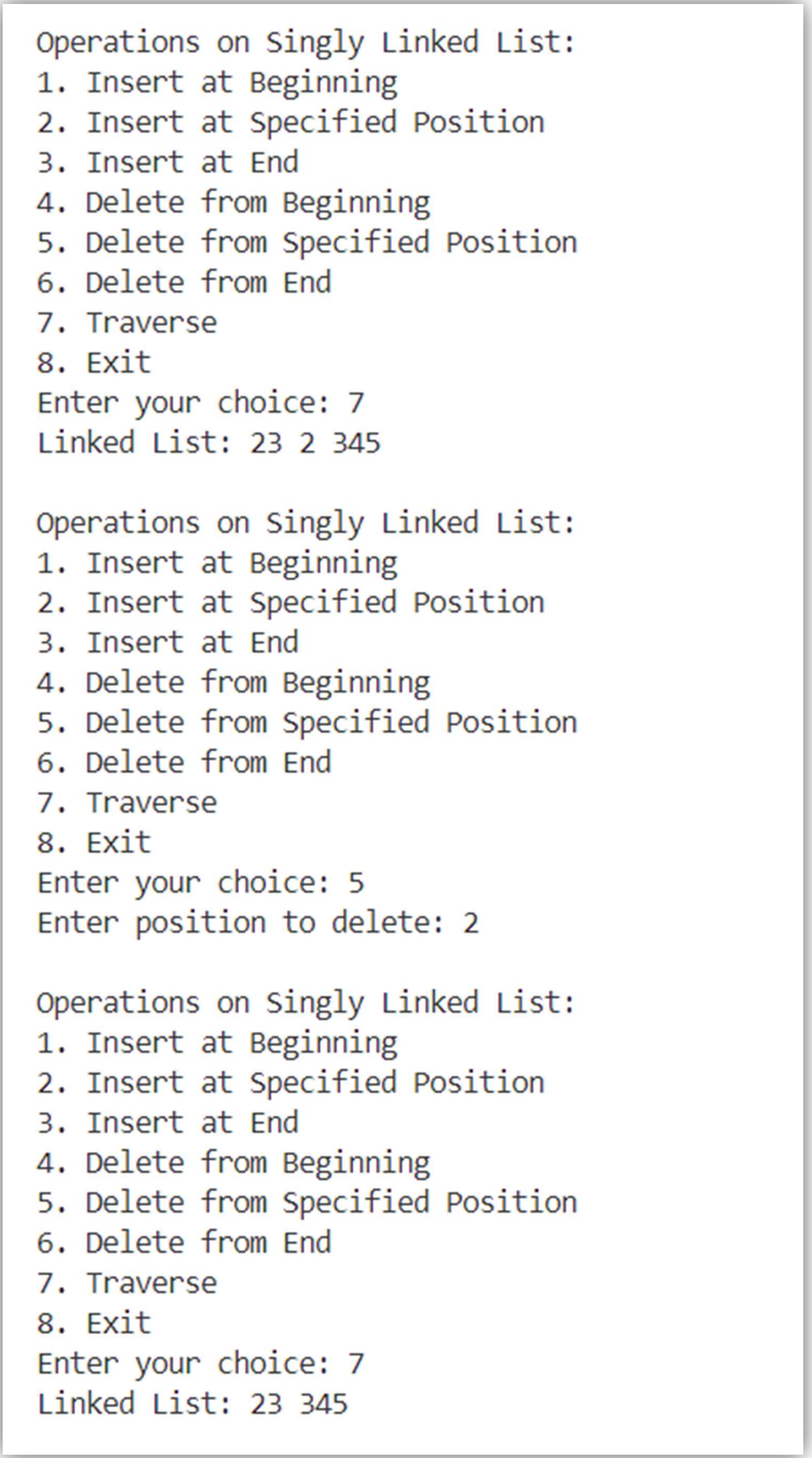
11. Write a program that uses functions to perform the following operations on singly linked list

- **a) Creation**
- **b) Insertion**
  - **1) Insertion at beginning**
  - **2) Insertion at specified position**
  - **3) Insertion at end**
- **c) Deletion**
  - **1) Deletion from the beginning**
  - **2) Deletion from the specified position**
  - **3) Deletion from the end**
- **d) Traversal.**
- **e) Exit**

```
#include <stdio.h>
#include <stdlib.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 = NULL;
    return newNode;
}

void insertBeginning(struct Node** headRef, int data) {
    struct Node* newNode = createNode(data);
    newNode->next = *headRef;
    *headRef = newNode;
}

void insertAtPosition(struct Node** headRef, int data, int position) {
    if (position < 1) {
        printf("Invalid position.\n");
        return;
    }
    if (position == 1) {
        insertBeginning(headRef, data);
        return;
    }
    struct Node* newNode = createNode(data);
    struct Node* current = *headRef;
    for (int i = 1; i < position - 1 && current != NULL; i++) {
        current = current->next;
    }
    if (current == NULL) {
        printf("Position out of range.\n");
        return;
    }
    newNode->next = current->next;
    current->next = newNode;
}

void insertEnd(struct Node** headRef, int data) {
    struct Node* newNode = createNode(data);
    if (*headRef == NULL) {
        *headRef = newNode;
        return;
    }
    struct Node* current = *headRef;
    while (current->next != NULL) {
        current = current->next;
    }
    current->next = newNode;
}

void deleteBeginning(struct Node** headRef) {
    if (*headRef == NULL) {
        printf("List is empty. Deletion not possible.\n");
        return;
    }
    struct Node* temp = *headRef;
    *headRef = (*headRef)->next;
    free(temp);
}

void deleteAtPosition(struct Node** headRef, int position) {
    if (*headRef == NULL) {
        printf("List is empty. Deletion not possible.\n");
        return;
    }
    if (position < 1) {
        printf("Invalid position.\n");
        return;
    }
    if (position == 1) {
        deleteBeginning(headRef);
        return;
    }
    struct Node* current = *headRef;
    struct Node* prev = NULL;
    for (int i = 1; i < position && current != NULL; i++) {
        prev = current;
        current = current->next;
    }
    if (current == NULL) {
        printf("Position out of range.\n");
        return;
    }
    prev->next = current->next;
    free(current);
}

void deleteEnd(struct Node** headRef) {
    if (*headRef == NULL) {
        printf("List is empty. Deletion not possible.\n");
        return;
    }
    if ((*headRef)->next == NULL) {
        free(*headRef);
        *headRef = NULL;
        return;
    }
    struct Node* current = *headRef;
    struct Node* prev = NULL;
    while (current->next != NULL) {
        prev = current;
        current = current->next;
    }
    prev->next = NULL;
    free(current);
}

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

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

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

    while (1) {
        printf("\nOperations on Singly Linked List:\n");
        printf("1. Insert at Beginning\n");
        printf("2. Insert at Specified Position\n");
        printf("3. Insert at End\n");
        printf("4. Delete from Beginning\n");
        printf("5. Delete from Specified Position\n");
        printf("6. Delete from End\n");
        printf("7. Traverse\n");
        printf("8. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter data to insert at beginning: ");
                scanf("%d", &data);
                insertBeginning(&head, data);
                break;
            case 2:
                printf("Enter data to insert: ");
                scanf("%d", &data);
                printf("Enter position: ");
                scanf("%d", &position);
                insertAtPosition(&head, data, position);
                break;
            case 3:
                printf("Enter data to insert at end: ");
                scanf("%d", &data);
                insertEnd(&head, data);
                break;
            case 4:
                deleteBeginning(&head);
                break;
            case 5:
                printf("Enter position to delete: ");
                scanf("%d", &position);
                deleteAtPosition(&head, position);
                break;
            case 6:
                deleteEnd(&head);
                break;
            case 7:
                traverse(head);
                break;
            case 8:
                freeList(&head);
                printf("Exiting program.\n");
                exit(0);
            default:
                printf("Invalid choice. Please enter a valid choice.\n");
        }
    }

    return 0;
}
```



```
PS C:\CSIT - Arjun Mijar_18> cd "c:\CSIT - Arjun Mijar_18\"

Operations on Singly Linked List:
1. Insert at Beginning
2. Insert at Specified Position
3. Insert at End
4. Delete from Beginning
5. Delete from Specified Position
6. Delete from End
7. Traverse
8. Exit
Enter your choice: 1
Enter data to insert at beginning: 23

Operations on Singly Linked List:
1. Insert at Beginning
2. Insert at Specified Position
3. Insert at End
4. Delete from Beginning
5. Delete from Specified Position
6. Delete from End
7. Traverse
8. Exit
Enter your choice: 3
Enter data to insert at end: 345

Operations on Singly Linked List:
1. Insert at Beginning
2. Insert at Specified Position
3. Insert at End
4. Delete from Beginning
5. Delete from Specified Position
6. Delete from End
7. Traverse
8. Exit
Enter your choice: 2
Enter data to insert: 2
Enter position: 2
```



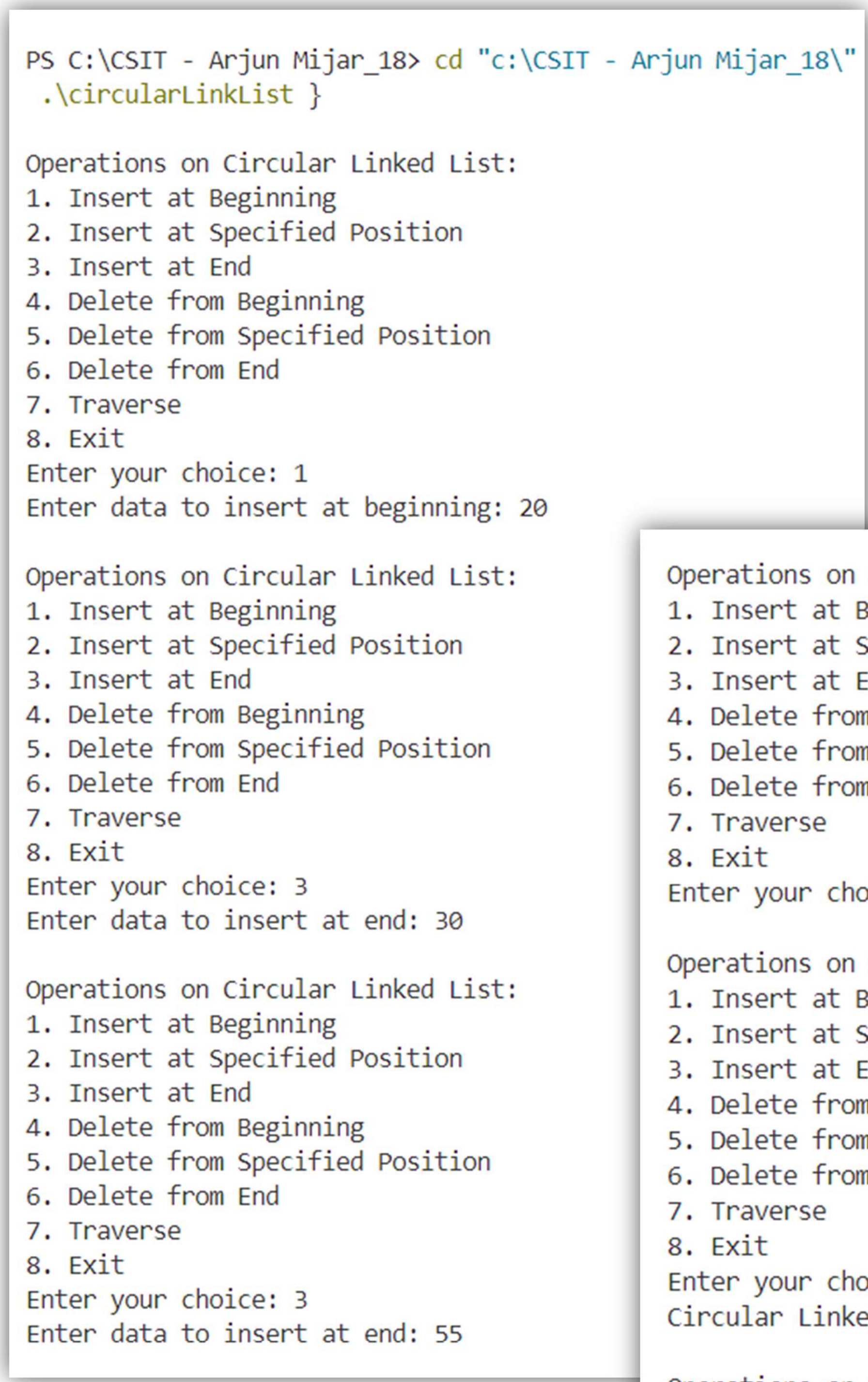
```
Operations on Singly Linked List:
1. Insert at Beginning
2. Insert at Specified Position
3. Insert at End
4. Delete from Beginning
5. Delete from Specified Position
6. Delete from End
7. Traverse
8. Exit
Enter your choice: 7
Linked List: 23 2 345

Operations on Singly Linked List:
1. Insert at Beginning
2. Insert at Specified Position
3. Insert at End
4. Delete from Beginning
5. Delete from Specified Position
6. Delete from End
7. Traverse
8. Exit
Enter your choice: 5
Enter position to delete: 2

Operations on Singly Linked List:
1. Insert at Beginning
2. Insert at Specified Position
3. Insert at End
4. Delete from Beginning
5. Delete from Specified Position
6. Delete from End
7. Traverse
8. Exit
Enter your choice: 7
Linked List: 23 345
```

Figure: Output

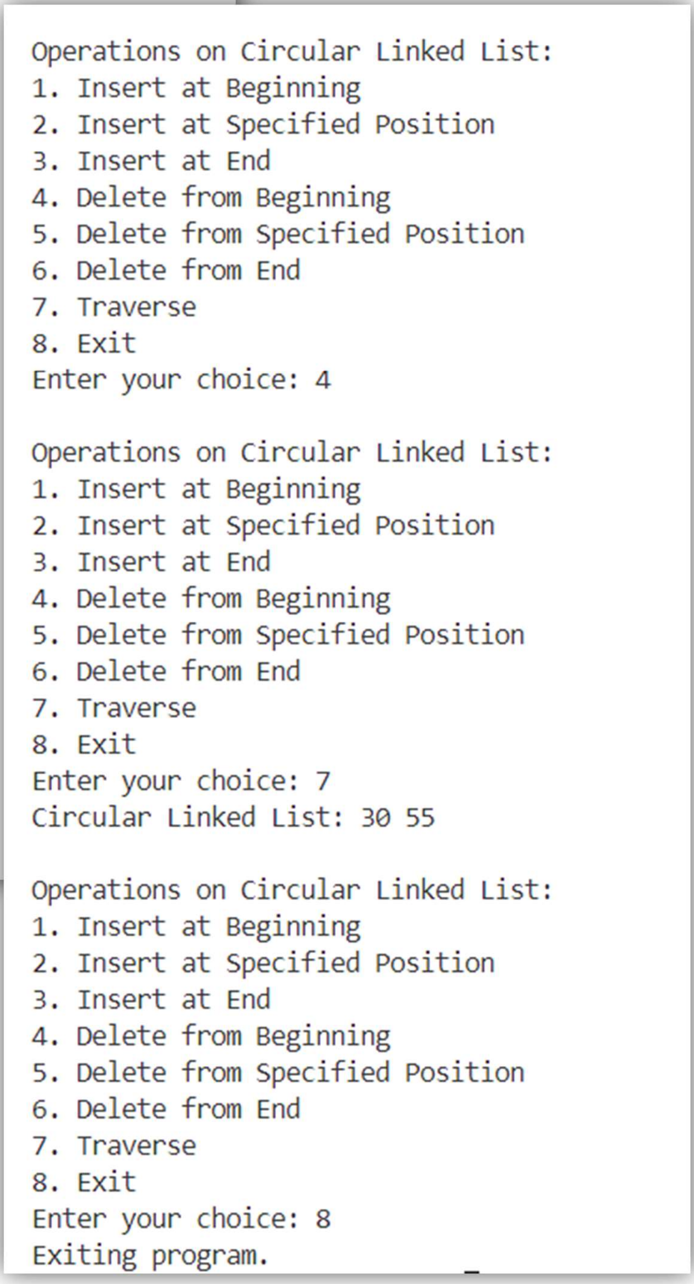
12. Write a program that uses functions to perform the following operations on circular linked List

- **a) Creation**
- **b) Insertion**
    - **1) Insertion at beginning**
    - **2) Insertion at specified position**
    - **3) Insertion at end**
- **c) Deletion**
    - **1) Deletion from the beginning**
    - **2) Deletion from the specified position**
    - **3) Deletion from the end**
- **d) Traversal.**
- **e) Exit**

```
#include <stdio.h>
#include <stdlib.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 = NULL;
    return newNode;
}

void insertBeginning(struct Node** headRef, int data) {
    struct Node* newNode = createNode(data);
    if (*headRef == NULL) {
        newNode->next = newNode;
        *headRef = newNode;
    } else {
        struct Node* last = *headRef;
        while (last->next != *headRef) {
            last = last->next;
        }
        newNode->next = *headRef;
        last->next = newNode;
        *headRef = newNode;
    }
}

void insertAtPosition(struct Node** headRef, int data, int position) {
    if (position < 1) {
        printf("Invalid position.\n");
        return;
    }
    if (position == 1) {
        insertBeginning(headRef, data);
        return;
    }
    struct Node* newNode = createNode(data);
    struct Node* current = *headRef;
    for (int i = 1; i < position - 1 && current->next != *headRef; i++) {
        current = current->next;
    }
    if (current->next == *headRef && position > 1) {
        printf("Position out of range.\n");
        return;
    }
    newNode->next = current->next;
    current->next = newNode;
}

void insertEnd(struct Node** headRef, int data) {
    struct Node* newNode = createNode(data);
    if (*headRef == NULL) {
        newNode->next = newNode;
        *headRef = newNode;
    } else {
        struct Node* last = *headRef;
        while (last->next != *headRef) {
            last = last->next;
        }
        newNode->next = *headRef;
        last->next = newNode;
    }
}

void deleteBeginning(struct Node** headRef) {
    if (*headRef == NULL) {
        printf("List is empty. Deletion not possible.\n");
        return;
    }
    if ((*headRef)->next == *headRef) {
        free(*headRef);
        *headRef = NULL;
    } else {
        struct Node* last = *headRef;
        while (last->next != *headRef) {
            last = last->next;
        }
        struct Node* temp = *headRef;
        *headRef = (*headRef)->next;
        last->next = *headRef;
        free(temp);
    }
}

void deleteAtPosition(struct Node** headRef, int position) {
    if (*headRef == NULL) {
        printf("List is empty. Deletion not possible.\n");
        return;
    }
    if (position < 1) {
        printf("Invalid position.\n");
        return;
    }
    if (position == 1) {
        deleteBeginning(headRef);
        return;
    }
    struct Node* current = *headRef;
    struct Node* prev = NULL;
    for (int i = 1; i < position && current->next != *headRef; i++) {
        prev = current;
        current = current->next;
    }
    if (current->next == *headRef && position > 1) {
        printf("Position out of range.\n");
        return;
    }
    prev->next = current->next;
    free(current);
}

void deleteEnd(struct Node** headRef) {
    if (*headRef == NULL) {
        printf("List is empty. Deletion not possible.\n");
        return;
    }
    if ((*headRef)->next == *headRef) {
        free(*headRef);
        *headRef = NULL;
    } else {
        struct Node* current = *headRef;
        struct Node* prev = NULL;
        while (current->next != *headRef) {
            prev = current;
            current = current->next;
        }
        prev->next = *headRef;
        free(current);
    }
}

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

void freeList(struct Node** headRef) {
    if (*headRef == NULL) {
        return;
    }
    struct Node* temp = *headRef;
    while (temp->next != *headRef) {
        struct Node* del = temp;
        temp = temp->next;
        free(del);
    }
    free(temp);
    *headRef = NULL;
}

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

    while (1) {
        printf("\nOperations on Circular Linked List:\n");
        printf("1. Insert at Beginning\n");
        printf("2. Insert at Specified Position\n");
        printf("3. Insert at End\n");
        printf("4. Delete from Beginning\n");
        printf("5. Delete from Specified Position\n");
        printf("6. Delete from End\n");
        printf("7. Traverse\n");
        printf("8. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter data to insert at beginning: ");
                scanf("%d", &data);
                insertBeginning(&head, data);
                break;
            case 2:
                printf("Enter data to insert: ");
                scanf("%d", &data);
                printf("Enter position: ");
                scanf("%d", &position);
                insertAtPosition(&head, data, position);
                break;
            case 3:
                printf("Enter data to insert at end: ");
                scanf("%d", &data);
                insertEnd(&head, data);
                break;
            case 4:
                deleteBeginning(&head);
                break;
            case 5:
                printf("Enter position to delete: ");
                scanf("%d", &position);
                deleteAtPosition(&head, position);
                break;
            case 6:
                deleteEnd(&head);
                break;
            case 7:
                traverse(head);
                break;
            case 8:
                freeList(&head);
                printf("Exiting program.\n");
                exit(0);
            default:
                printf("Invalid choice. Please enter a valid choice.\n");
        }
    }

    return 0;
}
```



```
PS C:\CSIT - Arjun Mijar_18> cd "c:\CSIT - Arjun Mijar_18\"
 .\circularLinkList }

Operations on Circular Linked List:
1. Insert at Beginning
2. Insert at Specified Position
3. Insert at End
4. Delete from Beginning
5. Delete from Specified Position
6. Delete from End
7. Traverse
8. Exit
Enter your choice: 1
Enter data to insert at beginning: 20

Operations on Circular Linked List:
1. Insert at Beginning
2. Insert at Specified Position
3. Insert at End
4. Delete from Beginning
5. Delete from Specified Position
6. Delete from End
7. Traverse
8. Exit
Enter your choice: 3
Enter data to insert at end: 30

Operations on Circular Linked List:
1. Insert at Beginning
2. Insert at Specified Position
3. Insert at End
4. Delete from Beginning
5. Delete from Specified Position
6. Delete from End
7. Traverse
8. Exit
Enter your choice: 3
Enter data to insert at end: 55
```

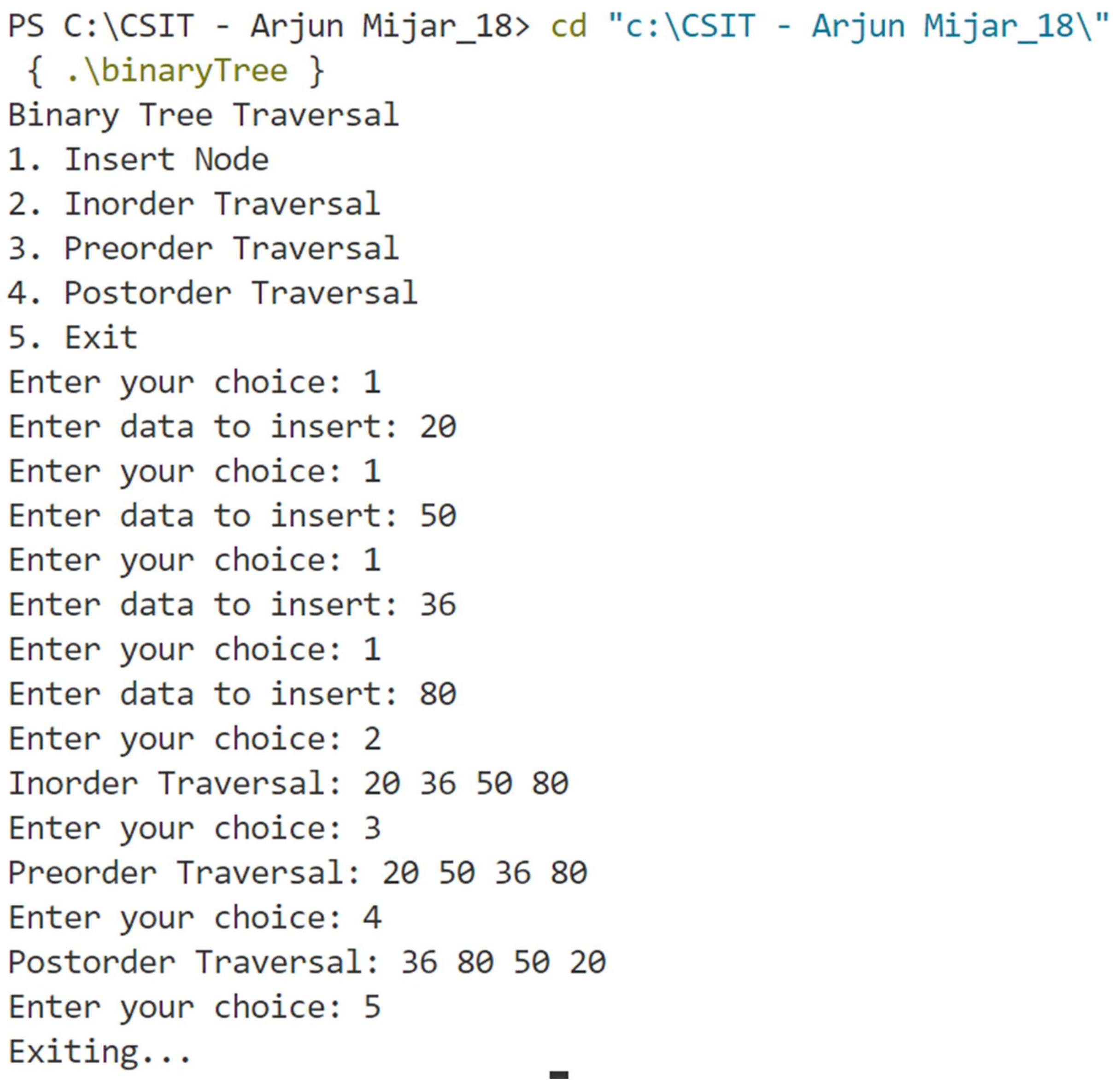


```
Operations on Circular Linked List:
1. Insert at Beginning
2. Insert at Specified Position
3. Insert at End
4. Delete from Beginning
5. Delete from Specified Position
6. Delete from End
7. Traverse
8. Exit
Enter your choice: 4

Operations on Circular Linked List:
1. Insert at Beginning
2. Insert at Specified Position
3. Insert at End
4. Delete from Beginning
5. Delete from Specified Position
6. Delete from End
7. Traverse
8. Exit
Enter your choice: 7
Circular Linked List: 30 55

Operations on Circular Linked List:
1. Insert at Beginning
2. Insert at Specified Position
3. Insert at End
4. Delete from Beginning
5. Delete from Specified Position
6. Delete from End
7. Traverse
8. Exit
Enter your choice: 8
Exiting program.
```

Figure: Output

13. Write a program to Implement binary tree and traverse tree with user's choice (Inorder, Preorder, Postorder).

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

// Structure for a binary tree node
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

// Function to create a new node
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (!newNode) {
        printf("Memory allocation failed\n");
        exit(1);
    }
    newNode->data = data;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}

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

// Function to perform inorder traversal
void inorder(struct Node* root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right);
    }
}

// Function to perform preorder traversal
void preorder(struct Node* root) {
    if (root != NULL) {
        printf("%d ", root->data);
        preorder(root->left);
        preorder(root->right);
    }
}

// Function to perform postorder traversal
void postorder(struct Node* root) {
    if (root != NULL) {
        postorder(root->left);
        postorder(root->right);
        printf("%d ", root->data);
    }
}

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

    // Menu for user choice
    printf("Binary Tree Traversal\n");
    printf("1. Insert Node\n");
    printf("2. Inorder Traversal\n");
    printf("3. Preorder Traversal\n");
    printf("4. Postorder Traversal\n");
    printf("5. Exit\n");

    do {
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter data to insert: ");
                scanf("%d", &data);
                root = insert(root, data);
                break;
            case 2:
                printf("Inorder Traversal: ");
                inorder(root);
                printf("\n");
                break;
            case 3:
                printf("Preorder Traversal: ");
                preorder(root);
                printf("\n");
                break;
            case 4:
                printf("Postorder Traversal: ");
                postorder(root);
                printf("\n");
                break;
            case 5:
                printf("Exiting...\n");
                break;
            default:
                printf("Invalid choice!\n");
        }
    } while (choice != 5);

    return 0;
}
```

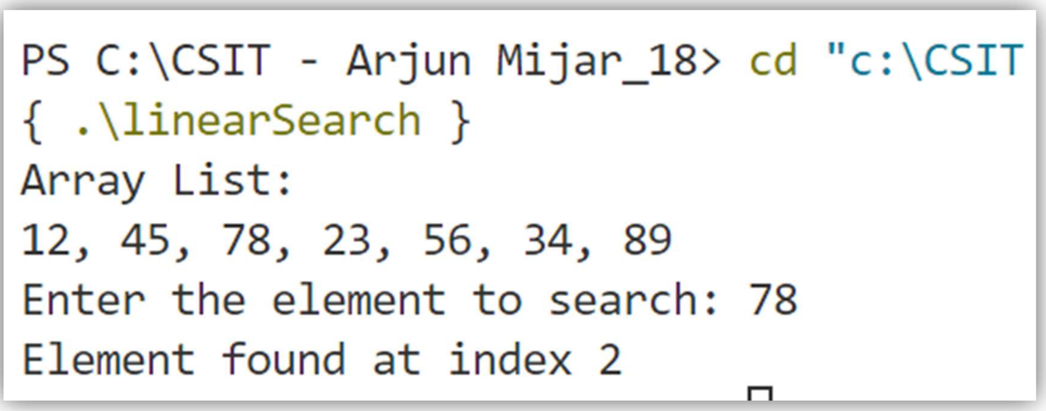


```
PS C:\CSIT - Arjun Mijar_18> cd "c:\CSIT - Arjun Mijar_18\"
 { .\binaryTree }
Binary Tree Traversal
1. Insert Node
2. Inorder Traversal
3. Preorder Traversal
4. Postorder Traversal
5. Exit
Enter your choice: 1
Enter data to insert: 20
Enter your choice: 1
Enter data to insert: 50
Enter your choice: 1
Enter data to insert: 36
Enter your choice: 1
Enter data to insert: 80
Enter your choice: 2
Inorder Traversal: 20 36 50 80
Enter your choice: 3
Preorder Traversal: 20 50 36 80
Enter your choice: 4
Postorder Traversal: 36 80 50 20
Enter your choice: 5
Exiting...
```

Figure: Output

14. Write a program to implement linear search.

```
#include <stdio.h>
// Function to perform linear search
int linearSearch(int arr[], int n, int key) {
   for (int i = 0; i < n; i++) {
      if (arr[i] == key) {
         return i; // Return the index if found
      }
   }
   return -1; // Return -1 if not found
}

int main() {
   int arr[]={12,45,78,23,56,34,89};
   int n = sizeof(arr) / sizeof(arr[0]);
   int key, result;
   printf("Array List: \n12,45,78,23,56,34,89");
   printf("Enter the element to search: ");
   scanf("%d", &key);

   result = linearSearch(arr, n, key);

   if (result != -1) {
      printf("Element found at index %d\n", result);
   } else {
      printf("Element not found in the array\n");
   }

   return 0;
}
```

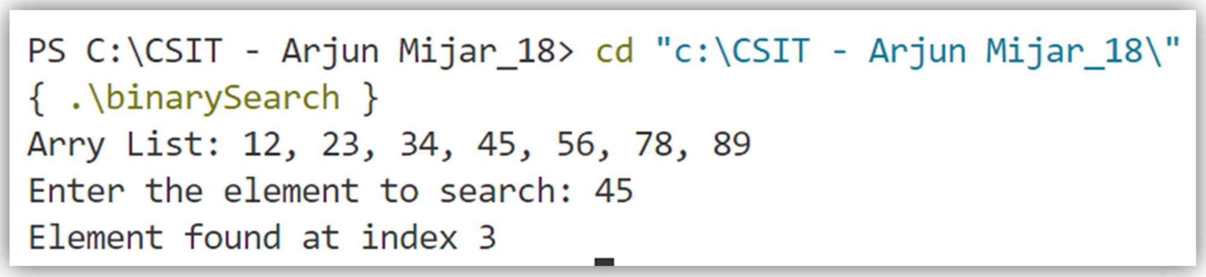


```
PS C:\CSIT - Arjun Mijar_18> cd "c:\CSIT
{ .\linearSearch }
Array List:
12, 45, 78, 23, 56, 34, 89
Enter the element to search: 78
Element found at index 2
```

Figure: Output

15. Write a program to implement binary search.

```
#include <stdio.h>

// Function to perform binary search
int binarySearch(int arr[], int low, int high, int key) {
    while (low <= high) {
        int mid = low + (high - low) / 2;

        if (arr[mid] == key) {
            return mid; // Return the index if found
        } else if (arr[mid] < key) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }
    return -1; // Return -1 if not found
}

int main() {
    int arr[] = {12, 23, 34, 45, 56, 78, 89};
    int n = sizeof(arr) / sizeof(arr[0]);
    int key, result;

    printf("Enter the element to search: ");
    scanf("%d", &key);

    result = binarySearch(arr, 0, n - 1, key);

    if (result != -1) {
        printf("Element found at index %d\n", result);
    } else {
        printf("Element not found in the array\n");
    }

    return 0;
}
```

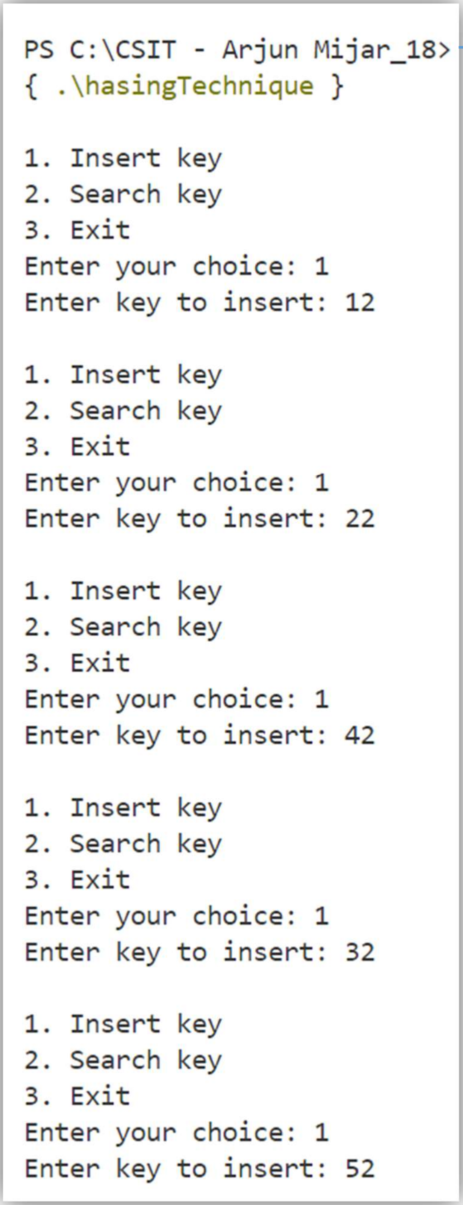


```
PS C:\CSIT - Arjun Mijar_18> cd "c:\CSIT - Arjun Mijar_18\"
{ .\binarySearch }
Arry List: 12, 23, 34, 45, 56, 78, 89
Enter the element to search: 45
Element found at index 3
```

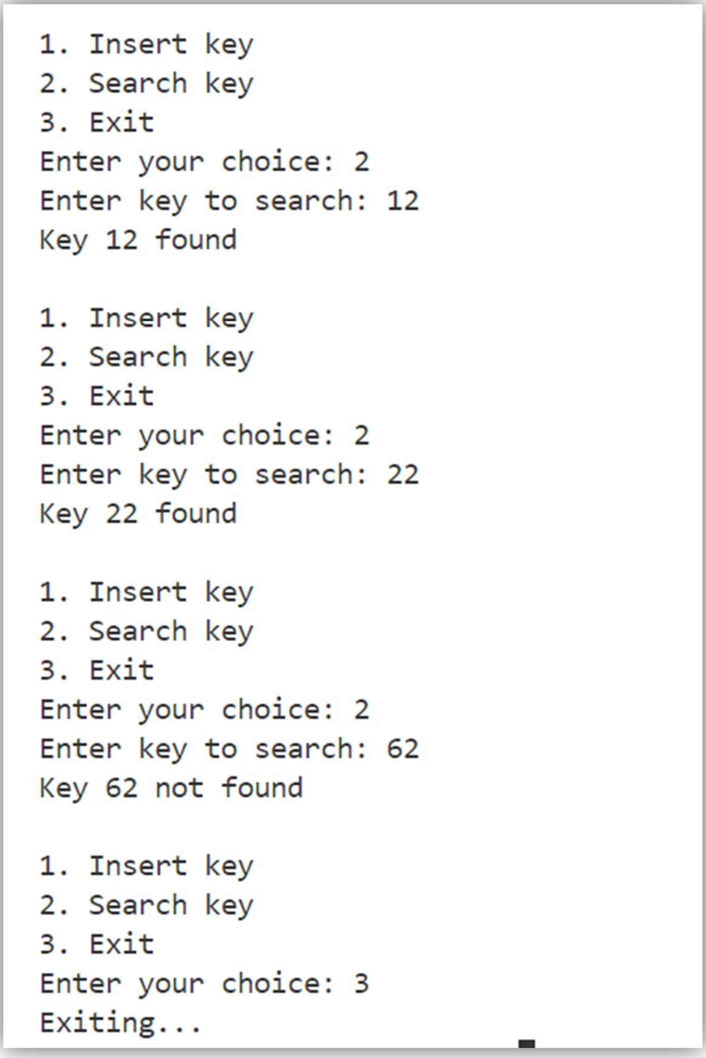
Figure: Output

16. Write a program to implement the hashing techniques.

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

#define SIZE 10

// Structure for a node in the hash table
struct Node {
    int data;
    struct Node* next;
};

// Function to create a new node
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (!newNode) {
        printf("Memory allocation failed\n");
        exit(1);
    }
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

// Function to initialize hash table
struct Node** initializeHashTable() {
    struct Node** hashTable = (struct Node**)malloc(SIZE * sizeof(struct Node*));
    if (!hashTable) {
        printf("Memory allocation failed\n");
        exit(1);
    }
    for (int i = 0; i < SIZE; i++) {
        hashTable[i] = NULL;
    }
    return hashTable;
}

// Function to insert data into hash table
void insert(struct Node** hashTable, int key) {
    int index = key % SIZE;
    struct Node* newNode = createNode(key);
    if (!hashTable[index]) {
        hashTable[index] = newNode;
    } else {
        struct Node* temp = hashTable[index];
        while (temp->next) {
            temp = temp->next;
        }
        temp->next = newNode;
    }
}

// Function to search for data in hash table
int search(struct Node** hashTable, int key) {
    int index = key % SIZE;
    struct Node* temp = hashTable[index];
    while (temp) {
        if (temp->data == key) {
            return 1; // Key found
        }
        temp = temp->next;
    }
    return 0; // Key not found
}

int main() {
    struct Node** hashTable = initializeHashTable();

    // Inserting elements into the hash table
    insert(hashTable, 12);
    insert(hashTable, 22);
    insert(hashTable, 42);
    insert(hashTable, 32);
    insert(hashTable, 52);

    // Searching for elements in the hash table
    printf("%d\n", search(hashTable, 12)); // Should print 1 (true)
    printf("%d\n", search(hashTable, 22)); // Should print 1 (true)
    printf("%d\n", search(hashTable, 42)); // Should print 1 (true)
    printf("%d\n", search(hashTable, 62)); // Should print 0 (false)

    return 0;
}
```



```
PS C:\CSIT - Arjun Mijar_18>
{ .\hasingTechnique }

1. Insert key
2. Search key
3. Exit
Enter your choice: 1
Enter key to insert: 12

1. Insert key
2. Search key
3. Exit
Enter your choice: 1
Enter key to insert: 22

1. Insert key
2. Search key
3. Exit
Enter your choice: 1
Enter key to insert: 42

1. Insert key
2. Search key
3. Exit
Enter your choice: 1
Enter key to insert: 32

1. Insert key
2. Search key
3. Exit
Enter your choice: 1
Enter key to insert: 52
```

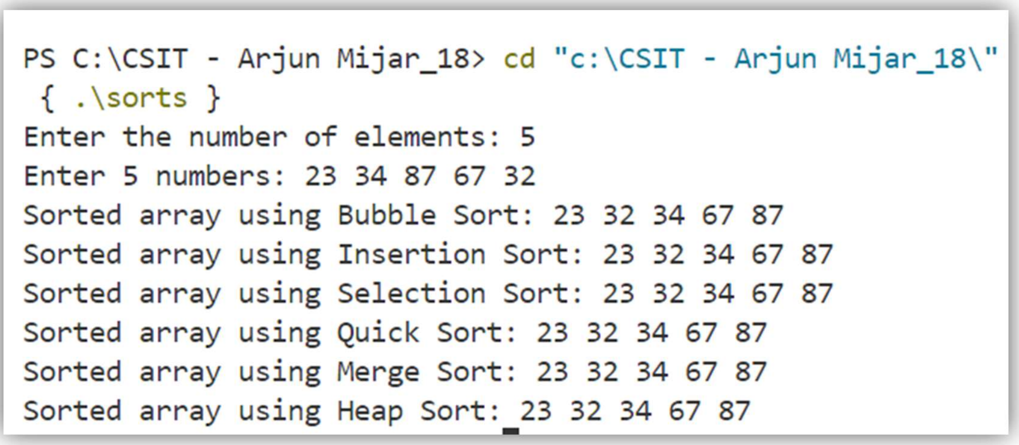


```
1. Insert key
2. Search key
3. Exit
Enter your choice: 2
Enter key to search: 12
Key 12 found

1. Insert key
2. Search key
3. Exit
Enter your choice: 2
Enter key to search: 22
Key 22 found

1. Insert key
2. Search key
3. Exit
Enter your choice: 2
Enter key to search: 62
Key 62 not found

1. Insert key
2. Search key
3. Exit
Enter your choice: 3
Exiting...
```

17. Write a program to enter n numbers and sort according to

- **a) Bubble sort**
- **b) Insertion sort**
- **c) Selection sort**
- **d) Quick sort**
- **e) Merge sort**
- **f) Heap sort**

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

// Function to swap two elements
void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Bubble Sort
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(&arr[j], &arr[j + 1]);
            }
        }
    }
}

// Insertion Sort
void insertionSort(int arr[], int n) {
    int i, key, j;
    for (i = 1; i < n; i++) {
        key = arr[i];
        j = i - 1;

        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}

// Selection Sort
void selectionSort(int arr[], int n) {
    int i, j, min_idx;
    for (i = 0; i < n - 1; i++) {
        min_idx = i;
        for (j = i + 1; j < n; j++) {
            if (arr[j] < arr[min_idx]) {
                min_idx = j;
            }
        }
        swap(&arr[min_idx], &arr[i]);
    }
}

// Quick Sort
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);
    }
}

// Merge Sort
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);
    }
}

// Heap Sort
void heapify(int arr[], int n, int i) {
    int largest = i;
    int l = 2 * i + 1;
    int r = 2 * i + 2;

    if (l < n && arr[l] > arr[largest])
        largest = l;

    if (r < n && arr[r] > arr[largest])
        largest = r;

    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);
    }
}

// Function to print an array
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

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

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

    // Bubble Sort
    int arr_bubble[n];
    for (int i = 0; i < n; i++) {
        arr_bubble[i] = arr[i];
    }
    bubbleSort(arr_bubble, n);
    printf("Sorted array using Bubble Sort: ");
    printArray(arr_bubble, n);

    // Insertion Sort
    int arr_insertion[n];
    for (int i = 0; i < n; i++) {
        arr_insertion[i] = arr[i];
    }
    insertionSort(arr_insertion, n);
    printf("Sorted array using Insertion Sort: ");
    printArray(arr_insertion, n);

    // Selection Sort
    int arr_selection[n];
    for (int i = 0; i < n; i++) {
        arr_selection[i] = arr[i];
    }
    selectionSort(arr_selection, n);
    printf("Sorted array using Selection Sort: ");
    printArray(arr_selection, n);

    // Quick Sort
    int arr_quick[n];
    for (int i = 0; i < n; i++) {
        arr_quick[i] = arr[i];
    }
    quickSort(arr_quick, 0, n - 1);
    printf("Sorted array using Quick Sort: ");
    printArray(arr_quick, n);

    // Merge Sort
    int arr_merge[n];
    for (int i = 0; i < n; i++) {
        arr_merge[i] = arr[i];
    }
    mergeSort(arr_merge, 0, n - 1);
    printf("Sorted array using Merge Sort: ");
    printArray(arr_merge, n);

    // Heap Sort
    int arr_heap[n];
    for (int i = 0; i < n; i++) {
        arr_heap[i] = arr[i];
    }
    heapSort(arr_heap, n);
    printf("Sorted array using Heap Sort: ");
    printArray(arr_heap, n);

    return 0;
}
```



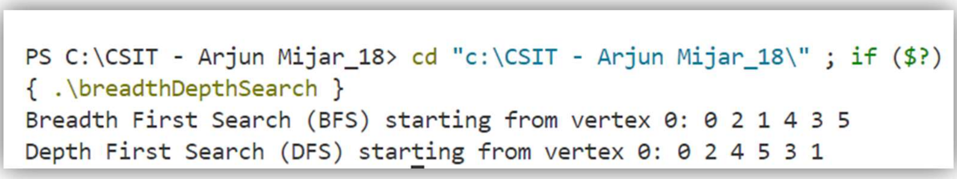
```
PS C:\CSIT - Arjun Mijar_18> cd "c:\CSIT - Arjun Mijar_18\"
 { .\sorts }
Enter the number of elements: 5
Enter 5 numbers: 23 34 87 67 32
Sorted array using Bubble Sort: 23 32 34 67 87
Sorted array using Insertion Sort: 23 32 34 67 87
Sorted array using Selection Sort: 23 32 34 67 87
Sorted array using Quick Sort: 23 32 34 67 87
Sorted array using Merge Sort: 23 32 34 67 87
Sorted array using Heap Sort: 23 32 34 67 87
```

Figure: Output

18. Write a program to implement Breadth First Search and Depth First Search in graph.

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

// Structure for a node in adjacency list
struct Node {
    int data;
    struct Node* next;
};

// Structure for adjacency list
struct Graph {
    int numVertices;
    struct Node** adjLists;
    int* visited;
};

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

// Function to create a graph with 'numVertices' vertices
struct Graph* createGraph(int numVertices) {
    struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
    graph->numVertices = numVertices;

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

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

// Function to add an edge to the graph
void addEdge(struct Graph* graph, int src, int dest) {
    // Add edge from src to dest
    struct Node* newNode = createNode(dest);
    newNode->next = graph->adjLists[src];
    graph->adjLists[src] = newNode;

    // Add edge from dest to src (since the graph is undirected)
    newNode = createNode(src);
    newNode->next = graph->adjLists[dest];
    graph->adjLists[dest] = newNode;
}

// Function for Breadth First Search (BFS)
void BFS(struct Graph* graph, int startVertex) {
    // Create a queue for BFS
    int queue[graph->numVertices];
    int front = 0, rear = 0;

    // Mark the startVertex as visited and enqueue it
    graph->visited[startVertex] = 1;
    queue[rear++] = startVertex;
```

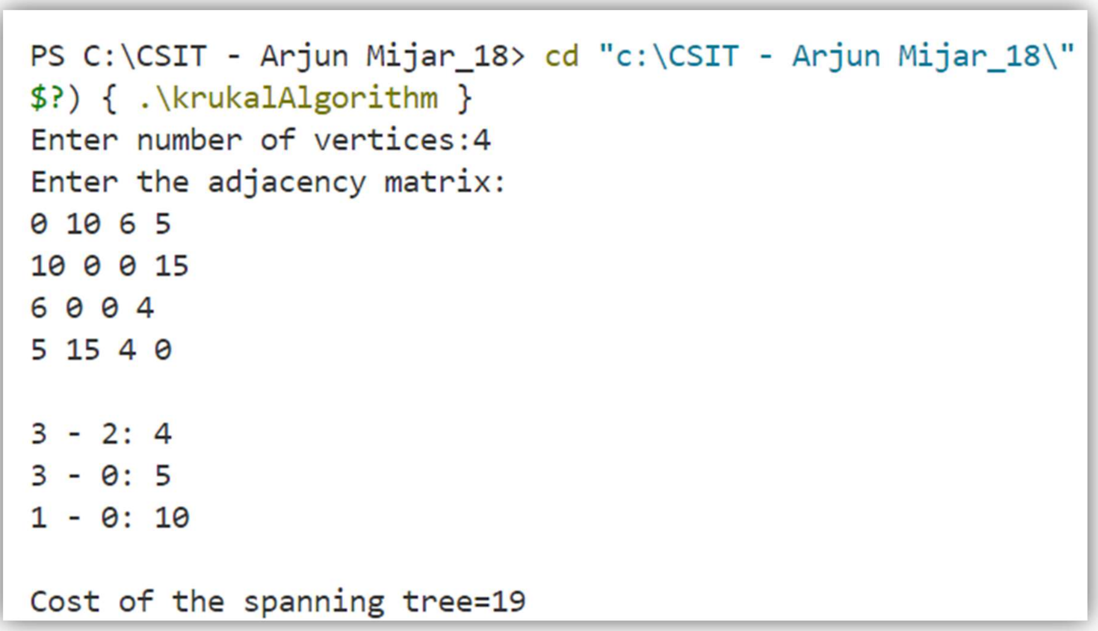
```c
    while (front < rear) {
        // Dequeue a vertex from queue and print it
        int currentVertex = queue[front++];
        printf("%d ", currentVertex);

        // Get all adjacent vertices of the dequeued vertex
        struct Node* temp = graph->adjLists[currentVertex];
        while (temp != NULL) {
            int adjVertex = temp->data;
            if (!graph->visited[adjVertex]) {
                graph->visited[adjVertex] = 1;
                queue[rear++] = adjVertex;
            }
            temp = temp->next;
        }
    }
}

// Function for Depth First Search (DFS)
void DFS(struct Graph* graph, int vertex) {
    // Mark the current vertex as visited
    graph->visited[vertex] = 1;
    printf("%d ", vertex);

    // Recur for all the vertices adjacent to this vertex
    struct Node* temp = graph->adjLists[vertex];
    while (temp != NULL) {
        int adjVertex = temp->data;
        if (!graph->visited[adjVertex]) {
            DFS(graph, adjVertex);
        }
        temp = temp->next;
    }
}
int main() {
    int numVertices = 6; // Example graph has 6 vertices
    struct Graph* graph = createGraph(numVertices);

    // Adding edges to the graph
    addEdge(graph, 0, 1);
    addEdge(graph, 0, 2);
    addEdge(graph, 1, 3);
    addEdge(graph, 1, 4);
    addEdge(graph, 2, 4);
    addEdge(graph, 3, 4);
    addEdge(graph, 3, 5);
    addEdge(graph, 4, 5);

    printf("Breadth First Search (BFS) starting from vertex 0: ");
    BFS(graph, 0);
    printf("\n");
    // Resetting visited array
    for (int i = 0; i < numVertices; i++) {
        graph->visited[i] = 0;
    }

    printf("Depth First Search (DFS) starting from vertex 0: ");
    DFS(graph, 0);
    printf("\n");

    return 0;
}
```



```
PS C:\CSIT - Arjun Mijar_18> cd "c:\CSIT - Arjun Mijar_18\" ; if ($?)
{ .\breadthDepthSearch }
Breadth First Search (BFS) starting from vertex 0: 0 2 1 4 3 5
Depth First Search (DFS) starting from vertex 0: 0 2 4 5 3 1
```

19. Write a program to implement Kruskal's algorithm.

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

#define MAX 30

typedef struct Edge
{
   int u, v, w;
} Edge;

typedef struct EdgeList
{
   Edge data[MAX];
   int n;
} EdgeList;
EdgeList elist;
int G[MAX][MAX], n;
EdgeList spanlist;

void Kruskal();
int find(int belongs[], int vertexno);
void union1(int belongs[], int c1, int c2);
void sort();
void print();

int main()
{
   int i, j, total_cost;
   printf("Enter number of vertices:");
   scanf("%d", &n);
   printf("Enter the adjacency matrix:\n");

   for (i = 0; i < n; i++)
      for (j = 0; j < n; j++)
         scanf("%d", &G[i][j]);

   Kruskal();
   print();
   return 0;
}

void Kruskal()
{
   int belongs[MAX], i, j, cno1, cno2;
   elist.n = 0;

   for (i = 1; i < n; i++)
   {
      for (j = 0; j < i; j++)
      {
            if (G[i][j] != 0)
            {
                elist.data[elist.n].u = i;
                elist.data[elist.n].v = j;
                elist.data[elist.n].w = G[i][j];
                elist.n++;
            }
        }
    }

    sort();

    for (i = 0; i < n; i++)
        belongs[i] = i;

    spanlist.n = 0;

    for (i = 0; i < elist.n; i++)
    {
        cno1 = find(belongs, elist.data[i].u);
        cno2 = find(belongs, elist.data[i].v);

        if (cno1 != cno2)
        {
            spanlist.data[spanlist.n] = elist.data[i];
            spanlist.n++;
            union1(belongs, cno1, cno2);
        }
    }
}

int find(int belongs[], int vertexno)
{
    return (belongs[vertexno]);
}

void union1(int belongs[], int c1, int c2)
{
    int i;

    for (i = 0; i < n; i++)
    {
        if (belongs[i] == c2)
            belongs[i] = c1;
    }
}

void sort()
{
    int i, j;
    Edge temp;
    for (i = 1; i < elist.n; i++)
    {
        for (j = 0; j < elist.n - 1; j++)
        {
            if (elist.data[j].w > elist.data[j + 1].w)
            {
                temp = elist.data[j];
                elist.data[j] = elist.data[j + 1];
                elist.data[j + 1] = temp;
            }
        }
    }
}

void print()
{
    int i, cost = 0;

    for (i = 0; i < spanlist.n; i++)
    {
        printf("\n%d - %d: %d", spanlist.data[i].u, spanlist.data[i].v, spanlist.data[i].w);
        cost += spanlist.data[i].w;
    }

    printf("\n\nCost of the spanning tree=%d", cost);
}
```

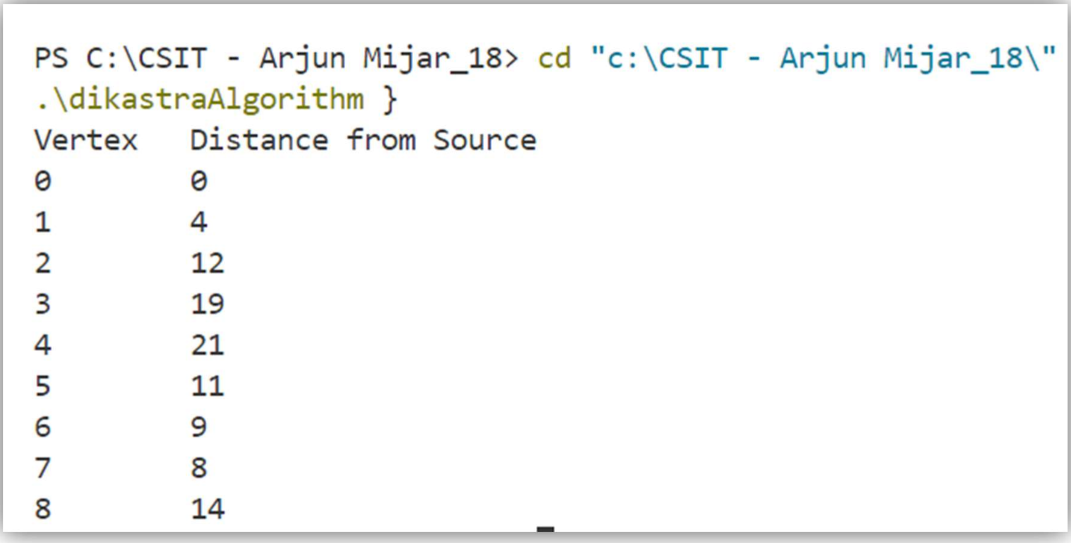


```
PS C:\CSIT - Arjun Mijar_18> cd "c:\CSIT - Arjun Mijar_18\"
$?) { .\krukalAlgorithm }
Enter number of vertices:4
Enter the adjacency matrix:
0 10 6 5
10 0 0 15
6 0 0 4
5 15 4 0

3 - 2: 4
3 - 0: 5
1 - 0: 10

Cost of the spanning tree=19
```

Figure: Output

20. Write a program to implement Dijkastra's algorithm.

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

#define V 9 // Number of vertices in the graph

// Function to find the vertex with the minimum distance value, from the set of vertices not yet included in the shortest
path tree
int minDistance(int dist[], int sptSet[]) {
    int min = INT_MAX, min_index;

    for (int v = 0; v < V; v++) {
        if (sptSet[v] == 0 && dist[v] <= min) {
            min = dist[v];
            min_index = v;
        }
    }

    return min_index;
}

// Function to print the constructed distance array
void printSolution(int dist[]) {
    printf("Vertex \t Distance from Source\n");
    for (int i = 0; i < V; i++) {
        printf("%d \t %d\n", i, dist[i]);
    }
}

// Function to implement Dijkstra's algorithm for a given graph represented as an adjacency matrix
void dijkstra(int graph[V][V], int src) {
    int dist[V]; // The output array. dist[i] will hold the shortest distance from src to i

    int sptSet[V]; // sptSet[i] will be true if vertex i is included in the shortest path tree or shortest distance from src to i is
finalized

    // Initialize all distances as INFINITE and stpSet[] as false
    for (int i = 0; i < V; i++) {
        dist[i] = INT_MAX;
        sptSet[i] = 0;
    }

    // Distance of source vertex from itself is always 0
    dist[src] = 0;

    // Find shortest path for all vertices
    for (int count = 0; count < V - 1; count++) {
        // Pick the minimum distance vertex from the set of vertices not yet processed. u is always equal to src in the first
iteration.
        int u = minDistance(dist, sptSet);

        // Mark the picked vertex as processed
        sptSet[u] = 1;

        // Update dist value of the adjacent vertices of the picked vertex
        for (int v = 0; v < V; v++) {
            // Update dist[v] only if it is not in sptSet, there is an edge from u to v, and total weight of path from src to v
through u is smaller than current value of dist[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];
            }
        }
    }

    // Print the constructed distance array
    printSolution(dist);
}

int main() {
    // Graph representation using adjacency matrix
    int graph[V][V] = {
        {0, 4, 0, 0, 0, 0, 0, 8, 0},
        {4, 0, 8, 0, 0, 0, 0, 11, 0},
        {0, 8, 0, 7, 0, 4, 0, 0, 2},
        {0, 0, 7, 0, 9, 14, 0, 0, 0},
        {0, 0, 0, 9, 0, 10, 0, 0, 0},
        {0, 0, 4, 14, 10, 0, 2, 0, 0},
        {0, 0, 0, 0, 0, 2, 0, 1, 6},
        {8, 11, 0, 0, 0, 0, 1, 0, 7},
        {0, 0, 2, 0, 0, 0, 6, 7, 0}
    };

    dijkstra(graph, 0);

    return 0;
}
```



```
PS C:\CSIT - Arjun Mijar_18> cd "c:\CSIT - Arjun Mijar_18\"
.\dikastraAlgorithm }
Vertex   Distance from Source
0        0
1        4
2        12
3        19
4        21
5        11
6        9
7        8
8        14
```

Figure: Output