VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
On

DATA STRUCTURES (23CS3PCDST)

Submitted by

PREETHI NARASIMHAN (1BM22CS207)

in partial fulfillment for the award of the degree of
BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING

B.M.S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019
Dec 2023- March 2024

B. M. S. College of Engineering,
Bull Temple Road, Bangalore 560019
(Affiliated To Visvesvaraya Technological University, Belgaum)
Department of Computer Science and Engineering

This is to certify that the Lab work entitled **"DATA STRUCTURES"** carried out by PREETHI NARASIMHAN **(1BM22CS207)**, who is a bonafide student of **B. M. S. College of Engineering**. It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum during the year 2023-24. The Lab report has been approved as it satisfies the academic requirements in respect of Data structures Lab - **(23CS3PCDST) work** prescribed for the said degree.

Prof. Sneha S Bagalkot
Assistant Professor
Department of CSE
BMSCE, Bengaluru

Dr. Jyothi S Nayak
Professor and Head
Department of CSE
BMSCE, Bengaluru

Index Sheet

Course outcomes:

CO1	Apply the concept of linear and nonlinear data structures.
CO2	Analyze data structure operations for a given problem
CO3	Design and develop solutions using the operations of linear and nonlinear data structure for a given specification.
CO4	Conduct practical experiments for demonstrating the operations of different data structures.

Lab program 1:

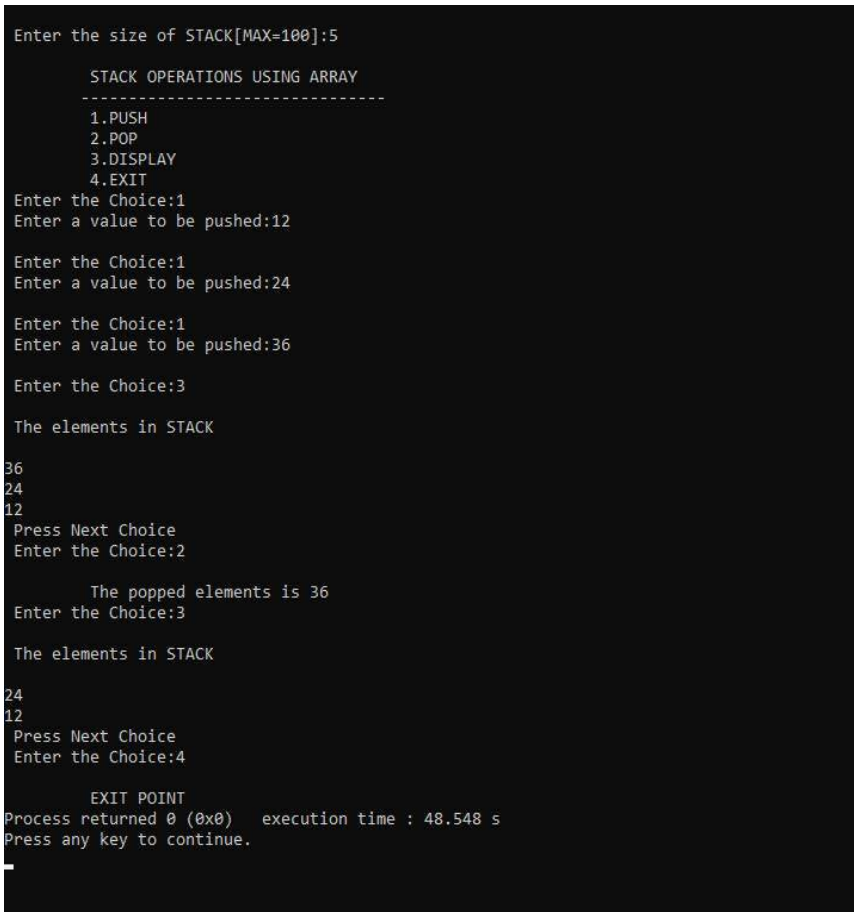
Write a program to simulate the working of stack using an array with the following:
a) Push
b) Pop
c) Display
The program should print appropriate messages for stack overflow, stack underflow.

Code:

```
#include <stdio.h>
#include<stdlib.h>
void push(int st[],int *top)
{
	int item;
	if(*top==SIZE-1)
		printf("Stack overflow\n");
	else
	{
		printf("\nEnter an item :");
		scanf("%d",&item);
		(*top)++;
		st[*top]=item;
	}
}
void pop(int st[],int *top)
{
	if(*top==-1)
		printf("Stack underflow\n");
	else
	{
		printf("\n%d item was deleted",st[(*top)--]);
	}
}
void display(int st[],int *top)
{
	int i;
	if(*top==-1)
		printf("Stack is empty\n");
	for(i=0;i<=*top;i++)
		printf("%d\t",st[i]);
}
int main()
{
	int st[10],top=-1, c,val_del;
	int SIZE;
	printf("Enter the size of STACK[MAX=100]\n");
	scanf("%d",&SIZE);
	printf("STACK OPERATIONS USING ARRAYS\n");
	printf("-----------------------------------------------------\n");
	printf("1. PUSH\n2. POP\n3. DISPLAY\n4.EXIT");
```

```
        do
        {
                printf("\nEnter the choice :");
                scanf("%d",&c);
                switch(c)
                {
                        case 1: push(st,&top);
                                break;
                        case 2: pop(st,&top);
                                break;
                        case 3: display(st,&top);
                                break;
                        case 4: printf("\n EXIT POINT");
                                break;
                        default: printf("\nPlease enter a valid choice (1/2/3/4) ");

                }
        }while(choice!=4);
return 0;
}
```

Output:



Lab Program 2:

2a) Write a program to convert a given valid parenthesized infix arithmetic expression to postfix expression. The expression consists of single character operands and the binary operators + (plus), - (minus), * (multiply) and / (divide).

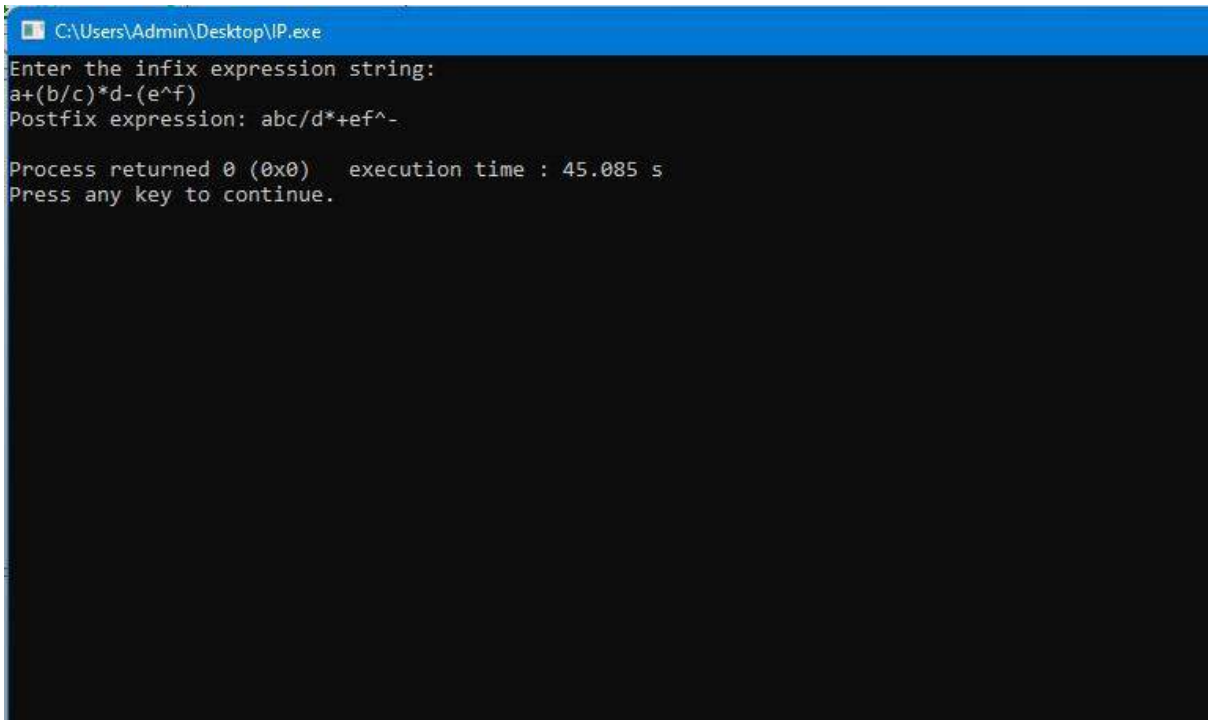
Code:

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

#define MAX 20

char s[MAX];
int top = -1;

int prec(char c) {
   if (c == '^')
      return 5;
   else if (c == '/')
      return 4;
   else if (c == '*')
      return 3;
   else if (c == '+')
      return 2;
   else if (c == '-')
      return 1;
   else
      return -1;
}

int isEmpty() {
   return top == -1;
}

int isFull() {
   return top == MAX - 1;
}

char peek() {
   return s[top];
}

char pop() {
   if (isEmpty()) {
      printf("Stack is empty\n");
      return -1;
   }
   else
```

```
    {
        char ch = s[top];
        top--;
        return ch;
    }
}

void push(char opr) {
    if (isFull()) {
        printf("Stack is full\n");
    } else {
        top = top + 1;
        s[top] = opr;
    }
}

void infixToPostfix(char infix[], char postfix[]) {
    int i, j;
    char ch;
    for (i = 0, j = 0; i < strlen(infix); i++) {
        ch = infix[i];
        if ((ch >= '0' && ch <= '9') || (ch >= 'A' && ch <= 'Z') || (ch >= 'a' && ch <= 'z')) {
            postfix[j++] = ch;
        } else if (ch == '(') {
            push(ch);
        } else if (ch == ')') {
            while (!isEmpty() && peek() != '(') {
                postfix[j++] = pop();
            }
            if (!isEmpty() && peek() != '(') {
                printf("Invalid expression\n");
                return;
            } else {
                pop();
            }
        } else {
            while (!isEmpty() && prec(ch) <= prec(peek())) {
                postfix[j++] = pop();
            }
            push(ch);
        }
    }

    while (!isEmpty()) {
        postfix[j++] = pop();
    }

    postfix[j] = '\0'; // Null-terminate the postfix expression
}
```

```
int main() {
    char infix[20], postfix[20];
    printf("Enter the infix expression string: \n");
    scanf("%s", infix);
    infixToPostfix(infix, postfix);
    printf("Postfix expression: %s\n", postfix);
    return 0;
}
```

Output:



2b) Write a Leetcode program to implement minstack.

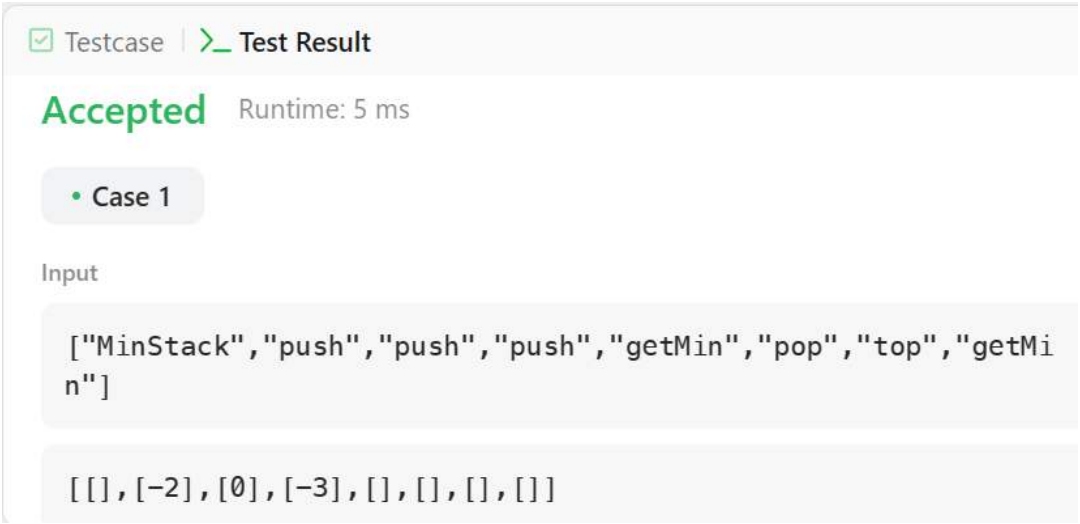
Code:

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

typedef struct MinStackNode
{
    int val;
    struct MinStackNode* next;
} MinStackNode;

typedef struct
{
    MinStackNode* top;
    int min;
```

```
} MinStack;

MinStack* minStackCreate()
{
    MinStack* stack = (MinStack*)malloc(sizeof(MinStack));
    stack->top = NULL;
    stack->min = __INT_MAX__;
    return stack;
}

void minStackPush(MinStack* obj, int val)
{
    MinStackNode* newNode = (MinStackNode*)malloc(sizeof(MinStackNode));
    newNode->val = val;
    newNode->next = obj->top;
    obj->top = newNode;
    if (val < obj->min)
        obj->min = val;
}

void minStackPop(MinStack* obj)
{
    if (obj->top == NULL)
        return;
    MinStackNode* temp = obj->top;
    obj->top = obj->top->next;
    if (temp->val == obj->min) {
        // Recalculate min if necessary
        MinStackNode* current = obj->top;
        obj->min = __INT_MAX__;
        while (current != NULL) {
            if (current->val < obj->min)
                obj->min = current->val;
            current = current->next;
        }
    }
    free(temp);
}

int minStackTop(MinStack* obj)
{
    if (obj->top == NULL)
        return -1; // Stack is empty
    return obj->top->val;
}

int minStackGetMin(MinStack* obj)
{
    return obj->min;
}
```

```
void minStackFree(MinStack* obj)
{
    while (obj->top != NULL) {
        MinStackNode* temp = obj->top;
        obj->top = obj->top->next;
        free(temp);
    }
    free(obj);
}

int main(int argc, char *argv[])
{
    // Test your MinStack implementation here
    MinStack* obj = minStackCreate();

    minStackPush(obj, -2);
    minStackPush(obj, 0);
    minStackPush(obj, -3);
    printf("getMin: %d\n", minStackGetMin(obj)); // return -3
    minStackPop(obj);
    printf("top: %d\n", minStackTop(obj));       // return 0
    printf("getMin: %d\n", minStackGetMin(obj)); // return -2
    minStackFree(obj);
    return 0;
}
```

Output:



Output

```
[null,null,null,null,-3,null,0,-2]
```

Expected

```
[null,null,null,null,-3,null,0,-2]
```

Lab Program 3:

3a) Write a program to simulate the working of a queue of integers using an array. Provide the following operations: Insert, Delete, Display
The program should print appropriate messages for queue empty and queue overflow conditions.

Code:

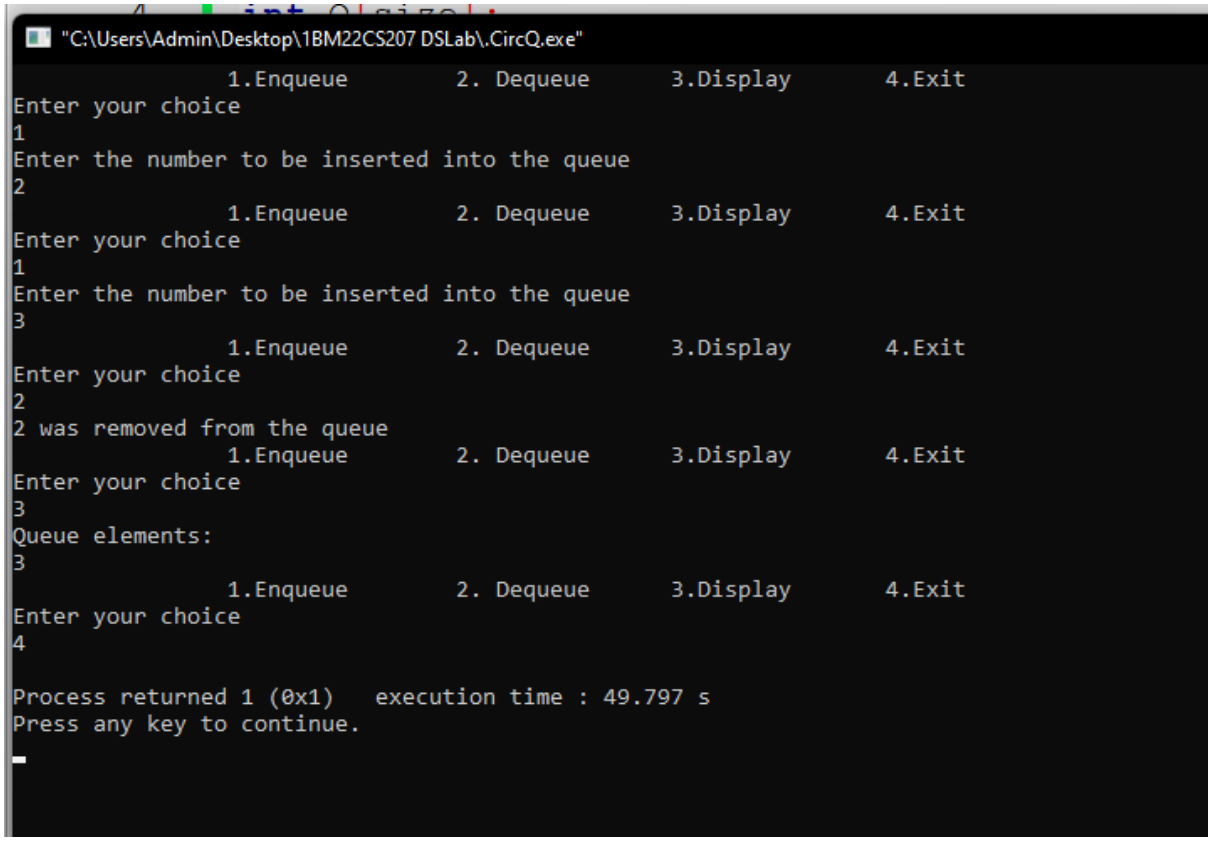
```
#include<stdio.h>
#include<stdlib.h>
#define size 3
int Q[size];
int rear=-1;
int front=-1;
int IsFull()
{
   if(front==(rear+1)%size)
   {
      return 0;
   }
   else
   {
      return -1;
   }
}
int IsEmpty()
{
   if(front==-1 && rear==-1)
   {
      return 0;
   }
   else
```

```
    {
        return -1;
    }
}
void Enqueue(int x)
{
    int item;
    if(IsFull()==0)
    {
        printf("Queue overflow \n");
        return;
    }
    else
    {
        if(IsEmpty()==0)
        {
            front=0;
            rear=0;
        }
        else
        {
          rear=(rear+1)%size;
        }
        Q[rear]=x;
    }
}
int Dequeue()
{
    int x;
    if(IsEmpty()==0)
    {
        printf("Queue underflow \n");
    }
    else
    {
        if(front==rear)
        {
            x=Q[front];
            front=-1;
            rear=-1;
```

```
        }
        else
        {
            x=Q[front];
            front=(front+1)%size;
        }
        return x;
    }
}
void Display()
{
    int i;
    if(IsEmpty()==0)
    {
        printf("Queue is empty \n");
    }
    else
    {
        printf("Queue elements:\n");
        for(i=front; i!=rear; i=(i+1)%size)
        {
            printf("%d \n",Q[i]);
        }
        printf("%d \n",Q[i]);
    }
}
void main()
{
    int choice,x,b;
    while(1)
    {
        printf("1.Enqueue, 2.Dequeue, 3.Display, 4.exit \n");
        printf("Enter your choice:");
        scanf("%d", &choice);
        switch(choice)
        {
        case 1:
            printf("Enter the number to be inserted \n");
            scanf("%d", &x);
            Enqueue(x);
```

```
            break;
        case 2:
            b=Dequeue();
            printf("%d was removed from the queue \n",b);
            break;
        case 3:
            Display();
            break;
        case 4:
            exit(1);
        default:
            printf("Invalid choice \n");

        }
    }
}
```

Output:



3b) Write a program to simulate the working of a circular queue of integers using an array. Provide the following operations: Insert, Delete and Display. The program should print appropriate messages for queue empty and queue overflow conditions.

Code:

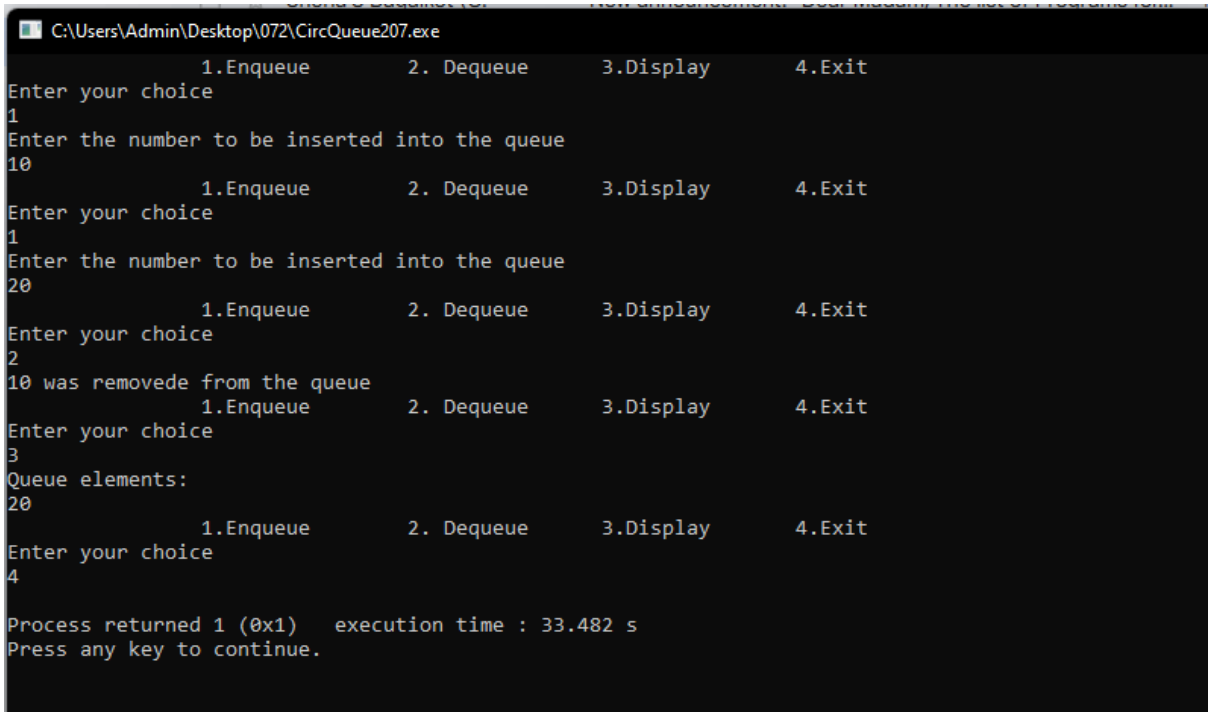
```
#include <stdio.h>
#include <stdlib.h>
#define size 50
int Q[size];
int rear=-1;
int front=-1;
int IsFull()
{
   if(front==(rear+1)%size)
   {
      return 0;
   }
   else
   {
      return -1;
   }
}
int IsEmpty()
{
   if(front==-1&&rear==-1)
   {
      return 0;
   }
   else
   {
      return -1;
   }
}
void Enqueue(int x)
{
   int item;
   if(IsFull()==0)
   {
```

```
        printf("Queue Overflow\n");
    }
    else
    {
        if(IsEmpty()==0)
        {
            front=0;
            rear=0;
        }
        else
        {
            rear=(rear+1)%size;
        }
        Q[rear]=x;
    }
}
int Dequeue()
{
    int x;
    if(IsEmpty()==0)
    {
        printf("Queue underflow\n");
    }
    else
    {
        if(front==rear)
        {
            x=Q[front];
            front=-1;
            rear=-1;
        }
        else
        {
            x=Q[front];
            front=(front+1)%size;
        }
        return x;
    }
}
void Display()
```

```
{
   int i;
   if(IsEmpty()==0)
   {
      printf("Queue is empty\n");
   }
   else
   {
      printf("Queue elements:\n");
      for(i=front;i!=rear;i=(i+1)%size)
      {
         printf("%d\n",Q[i]);
      }
      printf("%d \n",Q[i]);
   }
}
void main()
{
   int choice,x,b;
   while(1)
   {
      printf("\t\t1.Enqueue\t 2. Dequeue\t 3.Display\t 4.Exit\n");
      printf("Enter your choice\n");
      scanf("%d",&choice);
      switch(choice)
      {
      case 1:
         printf("Enter the number to be inserted into the queue\n");
         scanf("%d",&x);
         Enqueue(x);
         break;
      case 2:
         b=Dequeue();
         printf("%d was removed from the queue\n",b);
         break;
      case 3:
         Display();
         break;
      case 4:
         exit(1);
```

```
        default:
            printf("Invalid input\n");
        }
    }
}
```

Output:



Lab Program 4:

4a) Write a program to Implement Singly Linked List with following operations

a) Create a linked list.

b) Insertion of a node at first position, at any position and at end of list.

Display the contents of the linked list.

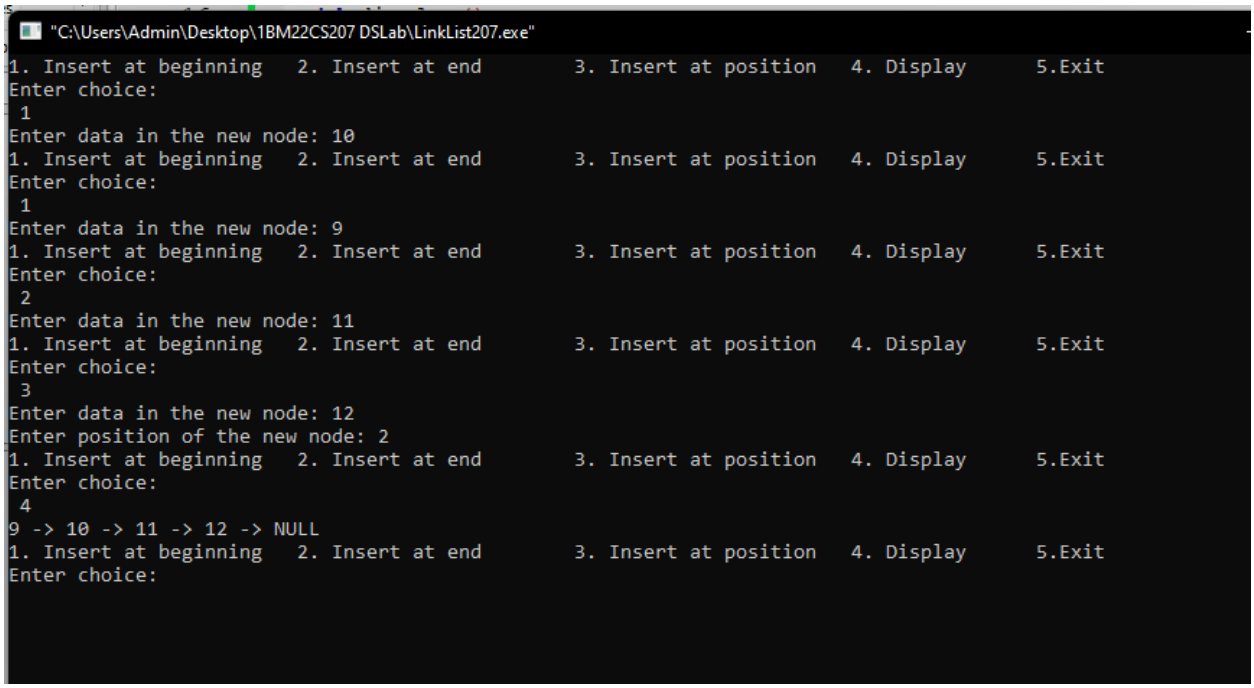
Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
typedef struct Node
 {
   int data;
   struct Node* next;
 } Node;
Node* head = NULL;
void push();
void append();
void insert();
void display();
void main()
{
   int choice;
   while (1) {
      printf("1. Insert at beginning\t 2. Insert at end\t 3. Insert at position\t 4. Display\t
5.Exit\t\n");
      printf("Enter choice: \n ");
      scanf("%d", &choice);
      switch (choice)
      {
         case 1:
            push();
            break;
         case 2:
            append();
            break;
         case 3:
            insert();
```

```
            break;
        case 4:
            display();
            break;
        default:
            printf("Exiting the program");
      }
   }
}

void push()
 {
   Node* temp = (Node*)malloc(sizeof(Node));
   int new_data;
   printf("Enter data in the new node: ");
   scanf("%d", &new_data);
   temp->data = new_data;
   temp->next = head;
   head = temp;
}

void append()
 {
   Node* temp = (Node*)malloc(sizeof(Node));
   int new_data;
   printf("Enter data in the new node: ");
   scanf("%d", &new_data);
   temp->data = new_data;
   temp->next = NULL;
   if (head == NULL) {
      head = temp;
      return;
   }
   Node* temp1 = head;
   while (temp1->next != NULL) {
      temp1 = temp1->next;
   }
   temp1->next = temp;
}
```

```
void insert() {
   Node* temp = (Node*)malloc(sizeof(Node));
   int new_data, pos;
   printf("Enter data in the new node: ");
   scanf("%d", &new_data);
   printf("Enter position of the new node: ");
   scanf("%d", &pos);
   temp->data = new_data;
   temp->next = NULL;
   if (pos == 0) {
      temp->next = head;
      head = temp;
      return;
   }
   Node* temp1 = head;
   while (pos--) {
      temp1 = temp1->next;
   }
   Node* temp2 = temp1->next;
   temp->next = temp2;
   temp1->next = temp;
}

void display()
{
   Node* temp1 = head;
   while (temp1 != NULL) {
      printf("%d -> ", temp1->data);
      temp1 = temp1->next;
   }
   printf("NULL\n");
}
```

Output:



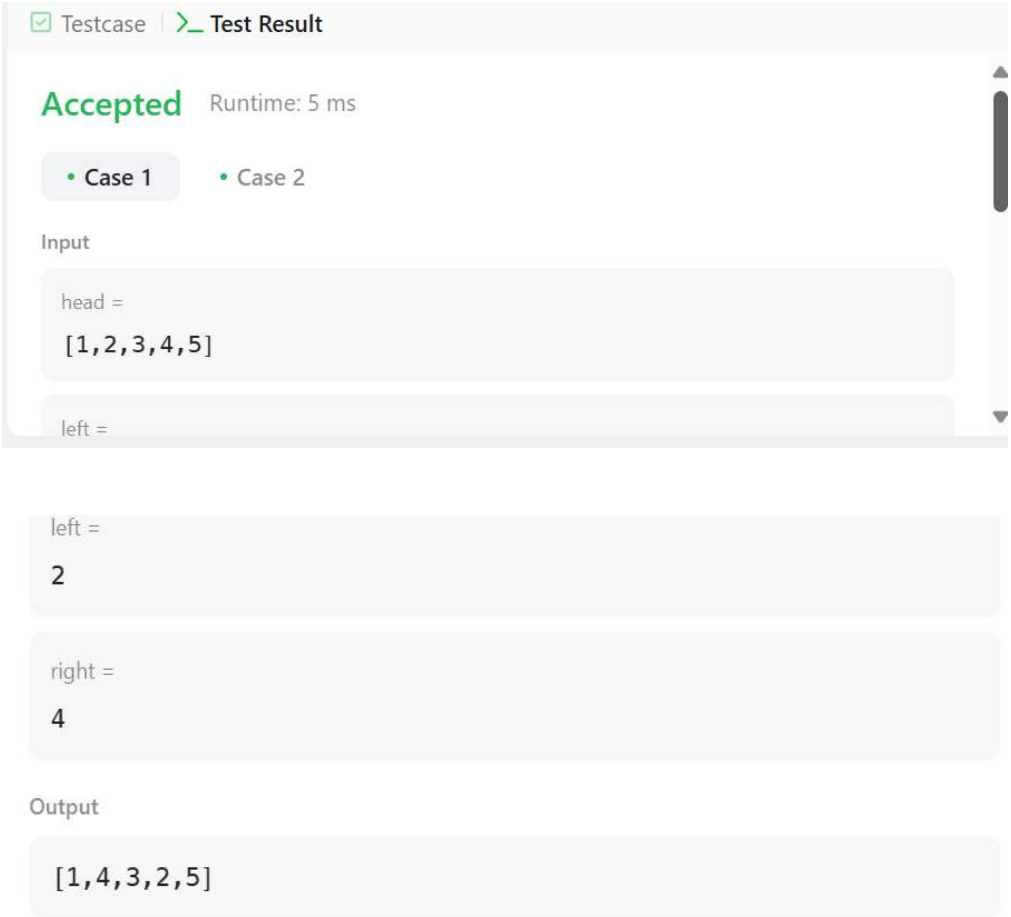
4b) Write a Leetcode program to reverse a Linked List.

Code:

```
struct ListNode* reverseBetween(struct ListNode* head, int left, int right)
{
    if (head == NULL || left == right) {
        return head;
    }
    struct ListNode* dummy = (struct ListNode*)malloc(sizeof(struct ListNode));
    dummy->next = head;
    struct ListNode* pre = dummy;
    // Move to the node just before the left position
    for (int i = 1; i < left; ++i)
   {
        pre = pre->next;
    }
    // Reverse the nodes from left to right
    struct ListNode* current = pre->next;
    struct ListNode* next = NULL;
```

```
    struct ListNode* prev = NULL;
    for (int i = 0; i <= right - left; ++i) {
        next = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }
    // Connect the reversed portion back to the original list
    pre->next->next = current;
    pre->next = prev;
    struct ListNode* result = dummy->next;
    free(dummy);
    return result;
}
```

Output:



Lab Program 5:

5a) Write a program to Implement Singly Linked List with following operations

a) Create a linked list.

b) Deletion of first element, specified element and last element in the list.

c) Display the contents of the linked list.

Code:

```
#include<stdio.h>
#include<stdlib.h>
struct node{
int data;
struct node*next;
};
void display();
void insert_begin();
void insert_end();
void insert_pos();
void begin_delete();
struct node *head=NULL;
void display()
{

   printf("elements are :\n");
        struct node *ptr;
        if(head==NULL)
        {
        printf("list is empty");
        return;



        }
```

```
        else{
        ptr=head;
        while(ptr !=NULL)
        {
        printf("%d\n", ptr->data);
        ptr=ptr->next;
        }
        }
}
void insert_begin()
{
        struct node*temp;
        temp =(struct node*)malloc(sizeof(struct node));
        printf("enter the value to be inserted\n");
        scanf("%d",&temp->data);
        temp->next=NULL;
        if(head==NULL)
        head=temp;
        else{
      temp->next=head;
        head=temp;
        }
}
void insert_end()
{
        struct node *temp,*ptr;
        temp=(struct node*)malloc(sizeof(struct node));
        printf("enter the value to be inserted \n");
        scanf("%d",&temp->data);
   temp->next=NULL;
        if(head==NULL)
        {
```

```
            head=temp;
            }
            else
            {
            ptr=head;
            while(ptr->next != NULL)
            {
            ptr=ptr->next;
            }
        ptr->next=temp;
            }
}
void insert_pos()
{

            int pos,i;
            struct node*temp,*ptr;
            printf("enter the position");
    scanf("%d",&pos);
            temp=(struct node*)malloc(sizeof(struct node));
            printf("enter the value to be inserted\n");
    scanf("%d",&temp->data);
    temp->next=NULL;
            if(pos==0)
            {

        temp->next=head;
            head=temp;

            }
            else
```

```
        {

        for(i=0, ptr=head; i<pos-1;i++)
        {

        ptr=ptr->next;
        }
    temp->next=ptr->next;
    ptr->next=temp;
        }
}

void begin_delete()
{
        struct node *ptr;
        if(head == NULL)
        {
    printf("\nList is empty\n");
        }
        else
        {
        ptr = head;
        head = ptr->next;
        free(ptr);
         printf("\nNode deleted from the begining ...\n");
        }
}

void last_delete()
{
        struct node *ptr,*ptr1;
        if(head == NULL)
```

```
        {
    printf("\nlist is empty");
        }
        else if(head -> next == NULL)
        {
        head = NULL;
        free(head);
    printf("\nOnly node of the list deleted ...\n");
        }
        else
        {
        ptr = head;
    while(ptr->next != NULL)
        {
        ptr1 = ptr;
        ptr = ptr ->next;
        }
        ptr1->next = NULL;
        free(ptr);
    printf("\nDeleted Node from the last ...\n");
        }
}

void random_delete()
{
        struct node *ptr,*ptr1;
        int loc,i;
        printf("\n Enter the location of the node after which you want to perform deletion \n");
        scanf("%d",&loc);
        ptr=head;
        for(i=0;i<loc;i++)
        {
```

```
        ptr1 = ptr;
        ptr = ptr->next;

        if(ptr == NULL)
        {
        printf("\nCan't delete");
        return;
        }
        }
        ptr1 ->next = ptr ->next;
        free(ptr);
        printf("\nDeleted node %d ",loc+1);
}

void main()
{
        int choice;
        while(1)
        {

        printf("\n 1.to insert at the beginning\n"
        " 2.to insert at the end\n "
          "3.to insert at the position\n "
          "4.to display\n "
          "5.delete from beginning\n"
          "6.delete from end\n"
          "7.random delete\n"
          "8.exit\n");
         printf("enter you choice:\n");
      scanf("%d",&choice);
        switch(choice)
        {
```

```
        case 1: insert_begin();
                break;
        case 2: insert_end();
                break;
        case 3: insert_pos();
                break;
        case 4: display();
                break;
        case 5: begin_delete();
                break;
        case 6: last_delete();
                break;
        case 7: random_delete();
                break;
        case 8: exit(0);
                break;
        default: printf("invalid choice\n");
                break;
        }
        }
}
```

Output:

```
 1.to insert at the beginning
 2.to insert at the end
 3.to insert at the position
 4.to display
 5.delete from beginning
6.delete from end
7.random delete
8.exit
enter you choice:
1
enter the value to be inserted
2

 1.to insert at the beginning
 2.to insert at the end
 3.to insert at the position
 4.to display
 5.delete from beginning
6.delete from end
7.random delete
8.exit
enter you choice:
1
enter the value to be inserted
3

 1.to insert at the beginning
 2.to insert at the end
 3.to insert at the position
 4.to display
 5.delete from beginning
6.delete from end
7.random delete
8.exit
enter you choice:
2
enter the value to be inserted
5

 1.to insert at the beginning
```

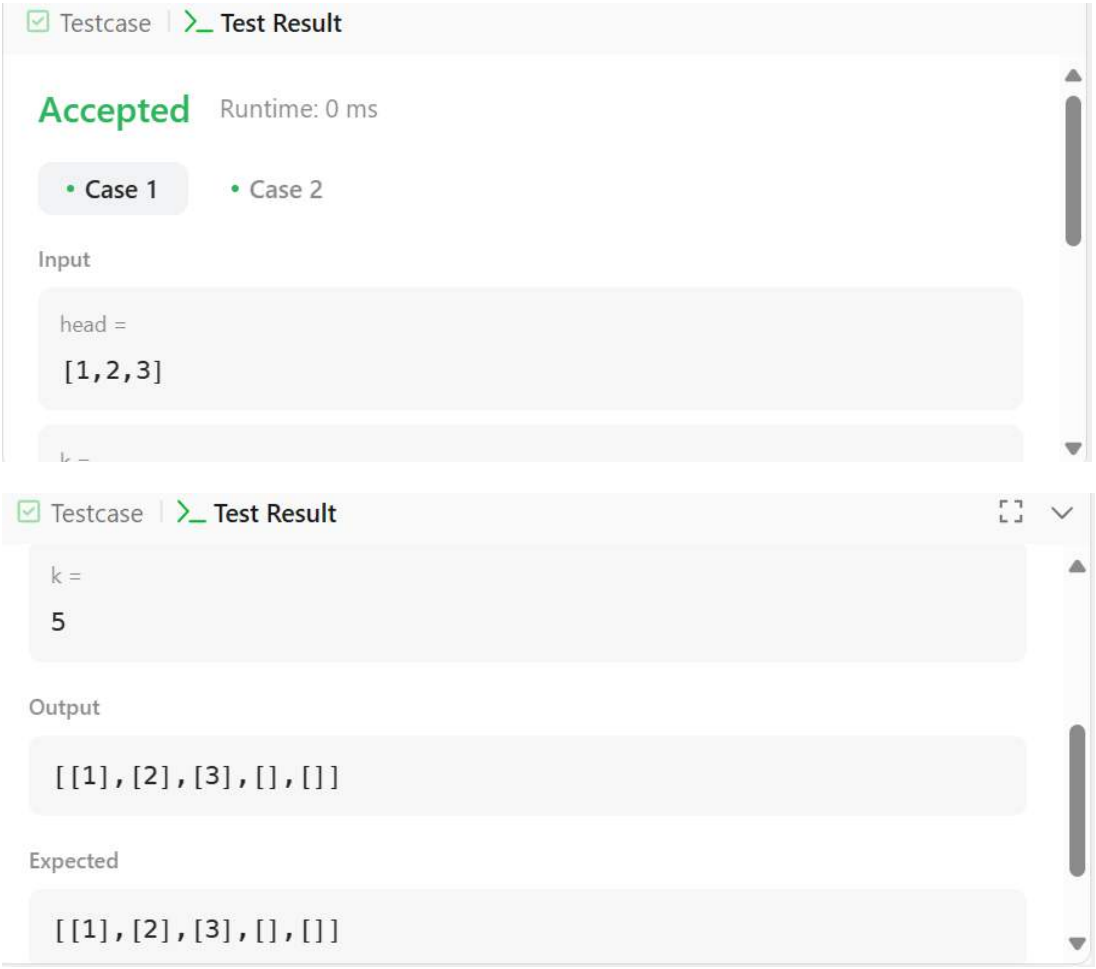
5b) Leetcode Program to split a Linked List into parts.

Code:

```
 int getLength(struct ListNode* head)
{
    int length = 0;
    while (head != NULL) {
        length++;
        head = head->next;
    }
```

```
    return length;
}


    struct ListNode** splitListToParts(struct ListNode* head, int k, int* returnSize)
{
    int length = getLength(head);
    int partSize = length / k;
    int remainder = length % k;

    struct ListNode** result = (struct ListNode**)malloc(k * sizeof(struct ListNode*));
    *returnSize = k;

    for (int i = 0; i < k; i++) {
        int currentPartSize = partSize + (i < remainder ? 1 : 0);

        if (currentPartSize == 0) {
            result[i] = NULL;
        } else {
            result[i] = head;
            for (int j = 0; j < currentPartSize - 1; j++) {
                head = head->next;
            }

            struct ListNode* temp = head->next;
            head->next = NULL;
            head = temp;
        }
    }

    return result;
}
```

Output:



Lab Program 6:

6a) Write a program to Implement Single Link List with following operations: Sort the linked list, Reverse the linked list, Concatenation of two linked lists.

Code:

```
 #include <stdio.h>
#include <stdlib.h>
struct Node {
        int data;
        struct Node* next;
};
```

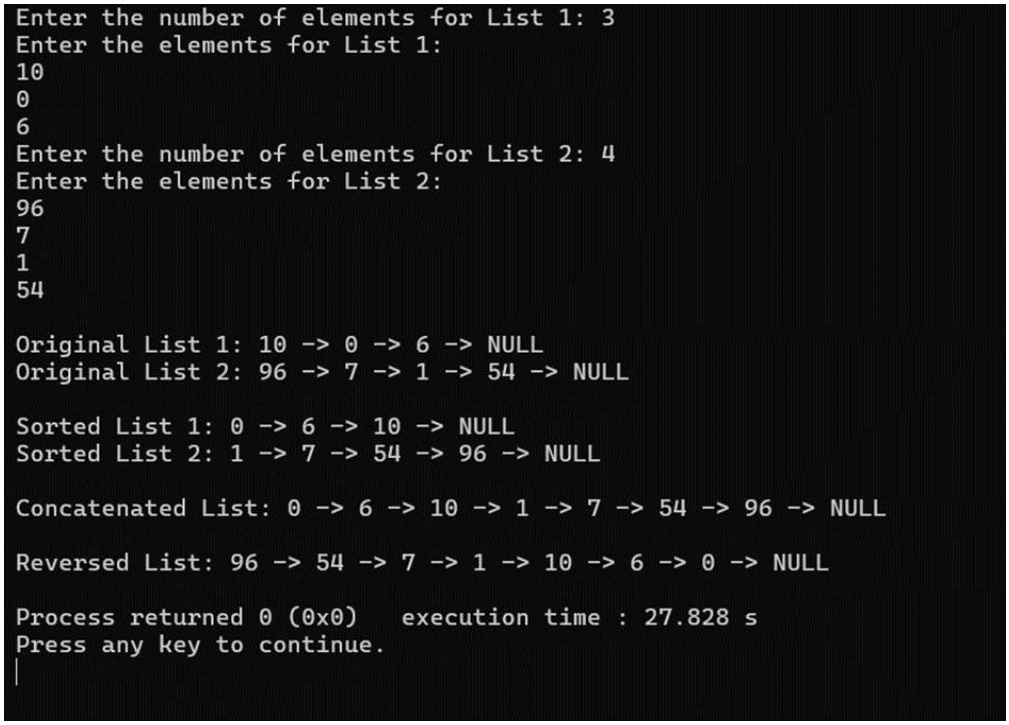
```
void append(struct Node** head_ref, int new_data)
{
        struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
        struct Node* last = *head_ref;
        new_node->data = new_data;
        new_node->next = NULL;
        if (*head_ref == NULL) {
        *head_ref = new_node;
        return;
        }
        while (last->next != NULL) {
        last = last->next;
        }
        last->next = new_node;
}

void printList(struct Node* node) {
        while (node != NULL) {
      printf("%d -> ", node->data);
        node = node->next;
        }
   printf("NULL\n");
}

void sortList(struct Node** head_ref)
{
        if (*head_ref == NULL) {
        return;
        }
        int swapped, temp;
        struct Node* ptr1;
```

```
        struct Node* lptr = NULL;
        do {
        swapped = 0;
        ptr1 = *head_ref;
        while (ptr1->next != lptr) {
        if (ptr1->data > ptr1->next->data) {
                temp = ptr1->data;
           ptr1->data = ptr1->next->data;
           ptr1->next->data = temp;
           swapped = 1;
        }
        ptr1 = ptr1->next;
        }
        lptr = ptr1;
        } while (swapped);
}

void reverseList(struct Node** head_ref) {
        struct Node* prev = NULL;
        struct Node* current = *head_ref;
        struct Node* next = NULL;
        while (current != NULL)
        {
        next = current->next;
        current->next = prev;
        prev = current;
        current = next;
        }
        *head_ref = prev;
}

void concatenateLists(struct Node** head1, struct Node* head2) {
```

```
        if (*head1 == NULL) {
        *head1 = head2;
        return;
        }

        struct Node* temp = *head1;
        while (temp->next != NULL) {
        temp = temp->next;
        }
        temp->next = head2;
}
int main() {
        struct Node* list1 = NULL;
        struct Node* list2 = NULL;
        int n, data;
        printf("Enter the number of elements for List 1: ");
        scanf("%d", &n);
        printf("Enter the elements for List 1:\n");
        for (int i = 0; i < n; ++i)
        {
                scanf("%d", &data);
                append(&list1, data);
        }
        printf("Enter the number of elements for List 2: ");
         scanf("%d", &n);
        printf("Enter the elements for List 2:\n");
        for (int i = 0; i < n; ++i)
        {
                scanf("%d", &data);
                append(&list2, data);
        }
   printf("\nOriginal List 1: ");
```

```
    printList(list1);
    printf("Original List 2: ");
    printList(list2);
    sortList(&list1);
    sortList(&list2);
    printf("\nSorted List 1: ");
    printList(list1);
    printf("Sorted List 2: ");
    printList(list2);
    concatenateLists(&list1, list2);
    printf("\nConcatenated List: ");
    printList(list1);
    reverseList(&list1);
    printf("\nReversed List: ");
    printList(list1);
    return 0;
}
```

Output:



6b) Write a program to implement Single Link List to simulate Stack and Queue Operations.

Code:

Stack using linked list:

```
#include <stdio.h>
#include <stdlib.h>
void push();
void pop();
void display();
struct node
{
int val;
struct node *next;
};
struct node *head;

void main ()
{
        int choice=0;
   printf("\nStack operations using linked list\n");
        while(choice != 4)
        {
      printf("\n\nChoose one from the below options...\n");
      printf("\n1.Push\n2.Pop\n3.Show\n4.Exit");
      printf("\n Enter your choice \n");
      scanf("%d",&choice);
      switch(choice)
        {
        case 1:
        {
           push();
                break;
        }
        case 2:
        {
```

```
                pop();
                break;
        }
        case 3:
        {
           display();
                break;
        }
        case 4:
        {
           printf("Exiting....");
                break;
        }
        default:
        {
           printf("Please Enter valid choice ");
        }
        };
}
}
void push ()
{
        int val;
        struct node *ptr = (struct node*)malloc(sizeof(struct node));
        if(ptr == NULL)
        {
      printf("Not able to push the element");
        }
        else
        {
      printf("Enter the value");
      scanf("%d",&val);
        if(head==NULL)
        {
        ptr->val = val;
        ptr -> next = NULL;
        head=ptr;
        }
        else
        {
```

```
            ptr->val = val;
            ptr->next = head;
            head=ptr;

            }
        printf("Item pushed");

            }
}

void pop()
{
            int item;
            struct node *ptr;
            if (head == NULL)
            {
        printf("Underflow");
            }
            else
            {
            item = head->val;
            ptr = head;
            head = head->next;
            free(ptr);
        printf("Item popped");

            }
}
void display()
{
            int i;
            struct node *ptr;
            ptr=head;
            if(ptr == NULL)
            {
        printf("Stack is empty\n");
            }
            else
            {
        printf("Printing Stack elements \n");
```

```
        while(ptr!=NULL)
            {
            printf("%d\n",ptr->val);
            ptr = ptr->next;
            }
            }
}
```

Output:

```
Stack operations using linked list


Chose one from the below options...

1.Push
2.Pop
3.Show
4.Exit
 Enter your choice
1
Enter the value23
Item pushed

Chose one from the below options...

1.Push
2.Pop
3.Show
4.Exit
 Enter your choice
1
Enter the value45
Item pushed

Chose one from the below options...

1.Push
2.Pop
3.Show
4.Exit
 Enter your choice
1
Enter the value75
Item pushed

Chose one from the below options...

1.Push
2.Pop
```

Queue using linked list:

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
        int data;
        struct node *next;
};
struct node *front;
struct node *rear;
void insert();
void delete();
void display();
void main ()
{
        int choice;
        while(choice != 4)
        {
            printf("\nQueue operation using linked list\n");
            printf("\n1.insert an element\n2.Delete an element\n3.Display the
        queue\n4.Exit\n");
            printf("\nEnter your choice ");
            scanf("%d",& choice);
            switch(choice)
            {
               case 1: insert();
                       break;
               case 2: delete();
                       break;
               case 3: display();
                       break;
               case 4: exit(0);
                       break;
               default: printf("\nEnter valid choice??\n");
               }
        }
}
void insert()
{
```

```
        struct node *ptr;
        int item;

        ptr = (struct node *) malloc (sizeof(struct node));
        if(ptr == NULL)
        {
        printf("\nOVERFLOW\n");
        return;
        }
        else
        {
        printf("\nEnter value?\n");
        scanf("%d",&item);
        ptr -> data = item;
        if(front == NULL)
        {
        front = ptr;
        rear = ptr;
        front -> next = NULL;
        rear -> next = NULL;
        }
        else
        {
        rear -> next = ptr;
        rear = ptr;
        rear->next = NULL;
        }
        }
}
void delete ()
{
        struct node *ptr;
        if(front == NULL)
        {
        printf("\nUNDERFLOW\n");
        return;
        }
        else
        {
        ptr = front;
```

```
            front = front -> next;
            free(ptr);
            }
}
void display()
{
            struct node *ptr;
            ptr = front;
            if(front == NULL)
            {
        printf("\nEmpty queue\n");
            }
            else
            {   printf("\nprinting values .....\n");
            while(ptr != NULL)
            {
            printf("\n%d\n",ptr -> data);
            ptr = ptr -> next;
            }
            }
}
```

Output:

```
Queue operation using linked list

1.insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice 2

Queue operation using linked list

1.insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice 2

Queue operation using linked list

1.insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice 3

printing values .....

54

76

100

Queue operation using linked list

1.insert an element
2.Delete an element
3.Display the queue
4.Exit
```

Lab Program 7:

7a) Write a program to implement doubly link list with primitive operations

a) Create a doubly linked list.
b) Insert a new node to the left of the node.
c) Delete the node based on a specific value.
d) Display the contents of the list.

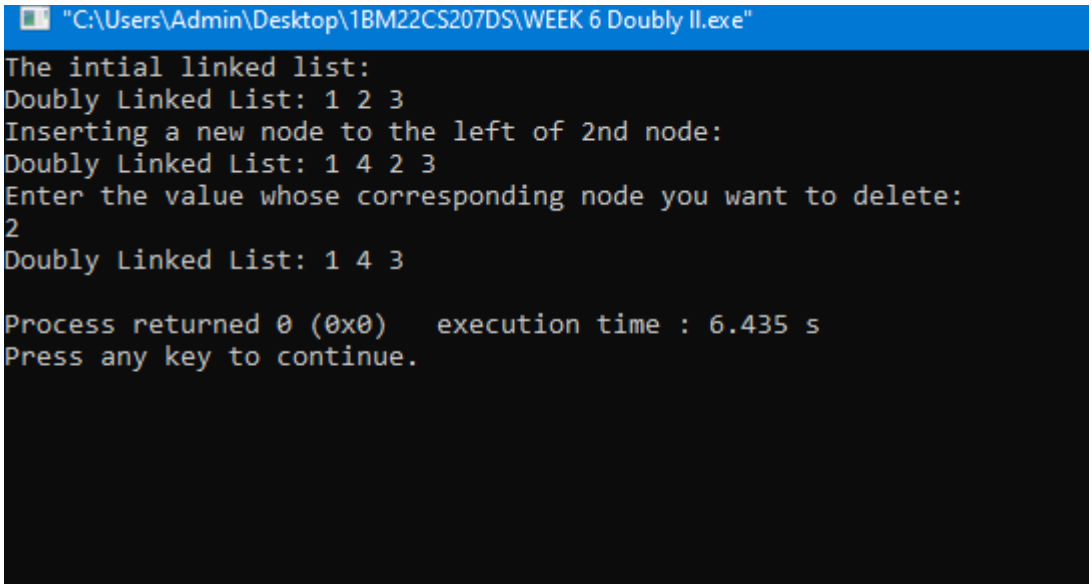
Code:

```
#include <stdio.h>
#include <stdlib.h>
struct Node
{
    int data;
    struct Node* prev;
    struct Node* next;
};

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

// Function to insert a new node to the left of the given node
void insertNodeToLeft(struct Node** head, struct Node* targetNode, int value)
{
    struct Node* newNode = createNode(value);
    if (targetNode->prev != NULL) {
```

```
        targetNode->prev->next = newNode;
        newNode->prev = targetNode->prev;
    }
    else
    {
        *head = newNode;
    }
    newNode->next = targetNode;
    targetNode->prev = newNode;
}

void deleteNodeByValue(struct Node** head, int value)
{
    struct Node* current = *head;
    while (current != NULL)
    {
        if (current->data == value)
        {
            if (current->prev != NULL)
            {
                current->prev->next = current->next;
            }
            else
            {
                *head = current->next;
            }
            if (current->next != NULL)
            {
                current->next->prev = current->prev;
            }
            free(current);
            return;
        }
        current = current->next;
    }
    printf("Node with value %d not found in the list.\n", value);
}
```

```
void displayList(struct Node* head)
{
      printf("Doubly Linked List: ");
      while (head != NULL)
   {
       printf("%d ", head->data);
       head = head->next;
   }
   printf("\n");
}
int main()
{
   struct Node* head = NULL;
   head = createNode(1);
   head->next = createNode(2);
   head->next->prev = head;
   head->next->next = createNode(3);
   head->next->next->prev = head->next;
   // Displaying the initial list
   printf("The intial linked list:\n");
   displayList(head);
   printf("Inserting a new node to the left of 2nd node:\n");
   insertNodeToLeft(&head, head->next, 4);
   displayList(head);
   int value;
   printf("Enter the value whose corresponding node you want to delete:\n");
   scanf("%d",&value);
   deleteNodeByValue(&head,value);
   displayList(head);
   return 0;
}
```

Output:



7b) Leetcode program to rotate a linked list.

Code:

```
struct ListNode* rotateRight(struct ListNode* head, int k)
{
    struct ListNode *temp = head;
    if (head == NULL) return NULL;
    if (head->next == NULL) return head;
    if (k == 0) return head;
    int size = 1;
    for(; temp->next != NULL; temp=temp->next, size++);
    k %= size;
    if (k == 0) return head;
    temp->next = head;
    struct ListNode *temp1 = head;
    for(int i = 0; i < (size-k-1); temp1 = temp1->next, i++);
    head = temp1->next;
    temp1->next = NULL;
    return head;
}
```

Output:

Lab Program 8:

8a) Write a program

a) To construct a binary Search tree.

b) To traverse the tree using all the methods i.e., in-order, preorder and post order.

c) To display the elements in the tree.

Code:

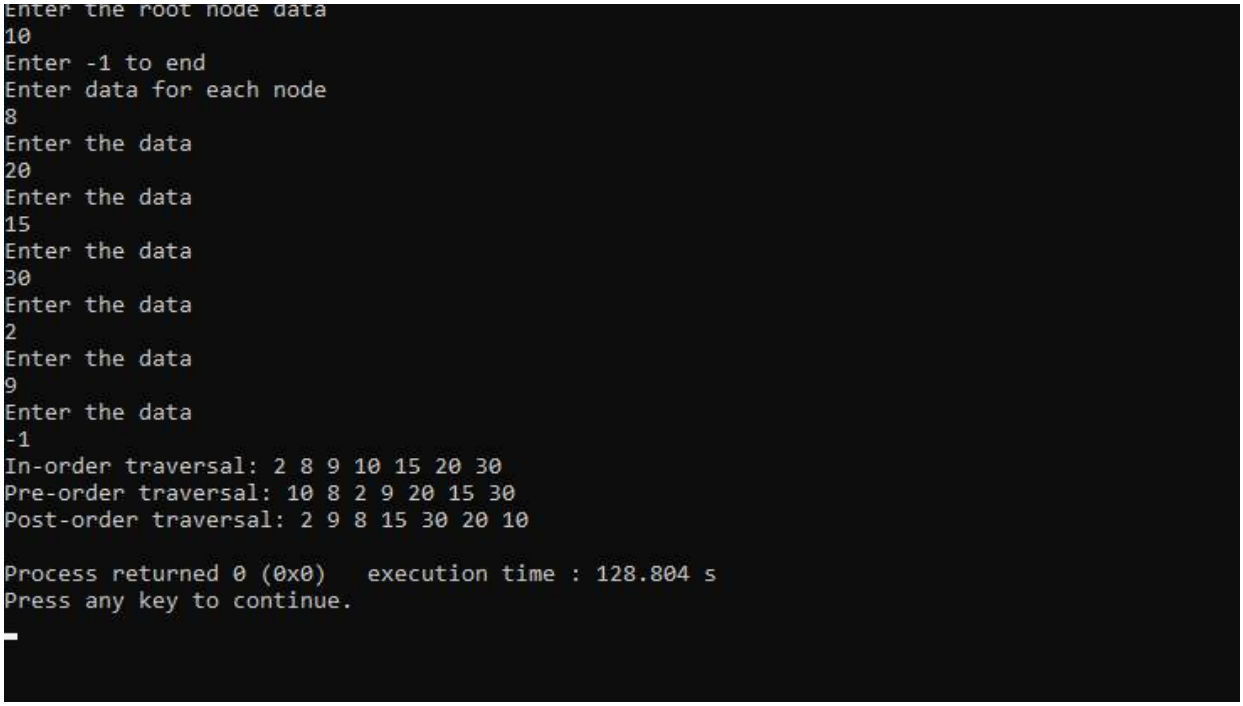
```
#include <stdio.h>
#include <stdlib.h>
struct TreeNode {
    int data;
    struct TreeNode* left;
    struct TreeNode* right;
};
struct TreeNode* createNode(int data)
{
```

```
    struct TreeNode* newNode = (struct TreeNode*)malloc(sizeof(struct TreeNode));
    newNode->data = data;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}

struct TreeNode* insert(struct TreeNode* 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;
}

void inorderTraversal(struct TreeNode* root)
{
    if (root != NULL)
   {
        inorderTraversal(root->left);
        printf("%d ", root->data);
        inorderTraversal(root->right);
    }
}

void preorderTraversal(struct TreeNode* root)
{
    if (root != NULL)
   {
        printf("%d ", root->data);
        preorderTraversal(root->left);
        preorderTraversal(root->right);
    }
}

void postorderTraversal(struct TreeNode* root)
```

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

void display(struct TreeNode* root)
{
   printf("In-order traversal: ");
   inorderTraversal(root);
   printf("\nPre-order traversal: ");
   preorderTraversal(root);
   printf("\nPost-order traversal: ");
   postorderTraversal(root);
   printf("\n");
}

int main()
{
   struct TreeNode* root = NULL;
   int num_root;
   int num;
   // Constructing the binary search tree
   printf("Enter the root node data\n");
   scanf("%d",&num_root);
   root = insert(root, num_root);
   printf("Enter -1 to end\n");
   printf("Enter data for each node\n");
   scanf("%d",&num);
   while(num!=-1)
   {
      insert(root,num);
      printf("Enter the data\n");
      scanf("%d",&num);
   }
   // Displaying the elements in the binary search tree
   display(root);
```

```
    return 0;
}
```

Output:



8b) Hackerrank program to swap nodes of a binary search tree at a level.

Code:

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

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

Node* createNode(int data)
{
```

```
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = data;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}

void inOrderTraversal(Node* root, int* result, int* index)
{
    if (root == NULL) return;
    inOrderTraversal(root->left, result, index);
    result[(*index)++] = root->data;
    inOrderTraversal(root->right, result, index);
}

void swapAtLevel(Node* root, int k, int level)
{
    if (root == NULL) return;
    if (level % k == 0)
   {
        Node* temp = root->left;
        root->left = root->right;
        root->right = temp;
    }
    swapAtLevel(root->left, k, level + 1);
    swapAtLevel(root->right, k, level + 1);
}

int** swapNodes(int indexes_rows, int indexes_columns, int** indexes, int queries_count,
 int* queries, int* result_rows, int* result_columns)
{
    // Build the tree
    Node** nodes = (Node*)malloc((indexes_rows + 1) * sizeof(Node));
    for (int i = 1; i <= indexes_rows; i++)
   {
        nodes[i] = createNode(i);
    }

    for (int i = 0; i < indexes_rows; i++)
   {
```

```
        int leftIndex = indexes[i][0];
        int rightIndex = indexes[i][1];
        if (leftIndex != -1) nodes[i + 1]->left = nodes[leftIndex];
        if (rightIndex != -1) nodes[i + 1]->right = nodes[rightIndex];
    }

    // Perform swaps and store results
    int** result = (int*)malloc(queries_count * sizeof(int));
    *result_rows = queries_count;
    *result_columns = indexes_rows;
     for (int i = 0; i < queries_count; i++)
    {
        swapAtLevel(nodes[1], queries[i], 1);
        int* traversalResult = (int*)malloc(indexes_rows * sizeof(int));
        int index = 0;
        inOrderTraversal(nodes[1], traversalResult, &index);
        result[i] = traversalResult;
    }

    free(nodes);
    return result;
}

int main()
{
            int n;
            scanf("%d", &n);
            int** indexes = malloc(n * sizeof(int*));
            for (int i = 0; i < n; i++)
            {
                indexes[i] = malloc(2 * sizeof(int));
                scanf("%d %d", &indexes[i][0], &indexes[i][1]);
            }
            int queries_count;
            scanf("%d", &queries_count);

            int* queries = malloc(queries_count * sizeof(int));
            for (int i = 0; i < queries_count; i++)
            {
                scanf("%d", &queries[i]);
```

```
    }
    int result_rows;
    int result_columns;
    int** result = swapNodes(n, 2, indexes, queries_count, queries, &result_rows,
    &result_columns);

    for (int i = 0; i < result_rows; i++)
    {
        for (int j = 0; j < result_columns; j++)
       {
            printf("%d ", result[i][j]);
        }
        printf("\n");
        free(result[i]); // Free memory allocated for each row
    }
    free(result); // Free memory allocated for the result array

    // Free memory allocated for indexes and queries arrays
    for (int i = 0; i < n; i++) {
        free(indexes[i]);
    }
    free(indexes);
    free(queries);
    return 0;
}
```

Output:

Lab Program 9:

9a) Write a program to traverse a graph using BFS method.

Code:

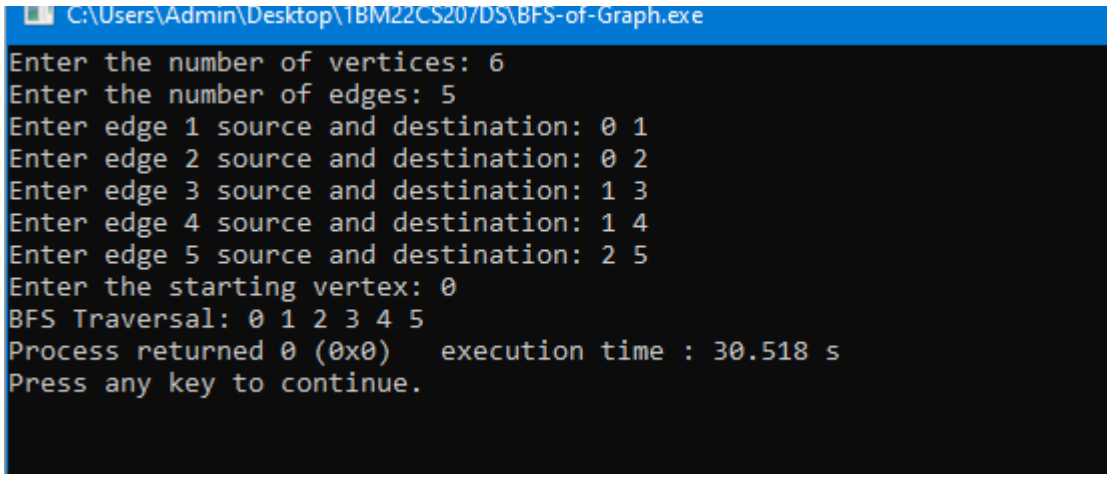
```
#include <stdio.h>
#include <stdbool.h>
#define MAX_SIZE 100
struct Queue {
    int items[MAX_SIZE];
    int front;
    int rear;
};

// Graph structure
struct Graph
{
```

```
    int vertices;
    bool adjMatrix[MAX_SIZE][MAX_SIZE];
};
struct Queue* createQueue()
{
    struct Queue* queue = (struct Queue*)malloc(sizeof(struct Queue));
    queue->front = -1;
    queue->rear = -1;
    return queue;
}

bool isEmpty(struct Queue* queue)
{
    if (queue->rear == -1)
        return true;
    else
        return false;
}

void enqueue(struct Queue* queue, int value)
{
    if (queue->rear == MAX_SIZE - 1)
        printf("\nQueue is full!");
    else
   {
        if (queue->front == -1)
            queue->front = 0;
        queue->rear++;
        queue->items[queue->rear] = value;
    }
}

int dequeue(struct Queue* queue)
{
    int item;
    if (isEmpty(queue))
    {
        printf("\nQueue is empty!");
        item = -1;
    }
```

```
    else
    {
        item = queue->items[queue->front];
        queue->front++;
        if (queue->front > queue->rear)
        {
            queue->front = queue->rear = -1;
        }
    }
    return item;
}

void createGraph(struct Graph* graph, int vertices)
{
    graph->vertices = vertices;
    for (int i = 0; i < vertices; i++)
    {
        for (int j = 0; j < vertices; j++)
        {
            graph->adjMatrix[i][j] = false;
        }
    }
}

void addEdge(struct Graph* graph, int src, int dest)
{
    graph->adjMatrix[src][dest] = true;
    graph->adjMatrix[dest][src] = true;
}

// Function to perform BFS traversal
void BFS(struct Graph* graph, int startVertex)
{
    bool visited[MAX_SIZE] = {false};
    struct Queue* queue = createQueue();
    visited[startVertex] = true;
    enqueue(queue, startVertex);
    while (!isEmpty(queue))
   {
        int currentVertex = dequeue(queue);
```

```
        printf("%d ", currentVertex);

        for (int i = 0; i < graph->vertices; i++)
        {
            if (graph->adjMatrix[currentVertex][i] && !visited[i])
            {
                visited[i] = true;
                enqueue(queue, i);
            }
        }
    }
}


int main()
{
    struct Graph graph;
    int vertices, edges, startVertex;
    printf("Enter the number of vertices: ");
    scanf("%d", &vertices);
    createGraph(&graph, vertices);
    printf("Enter the number of edges: ");
    scanf("%d", &edges);
    for (int i = 0; i < edges; i++)
    {
        int src, dest;
        printf("Enter edge %d source and destination: ", i+1);
        scanf("%d %d", &src, &dest);
        addEdge(&graph, src, dest);
    }

    printf("Enter the starting vertex: ");
    scanf("%d", &startVertex);
    printf("BFS Traversal: ");
    BFS(&graph, startVertex);
    return 0;
}
```

Output:



9b) Write a program to check whether given graph is connected or not using DFS method.

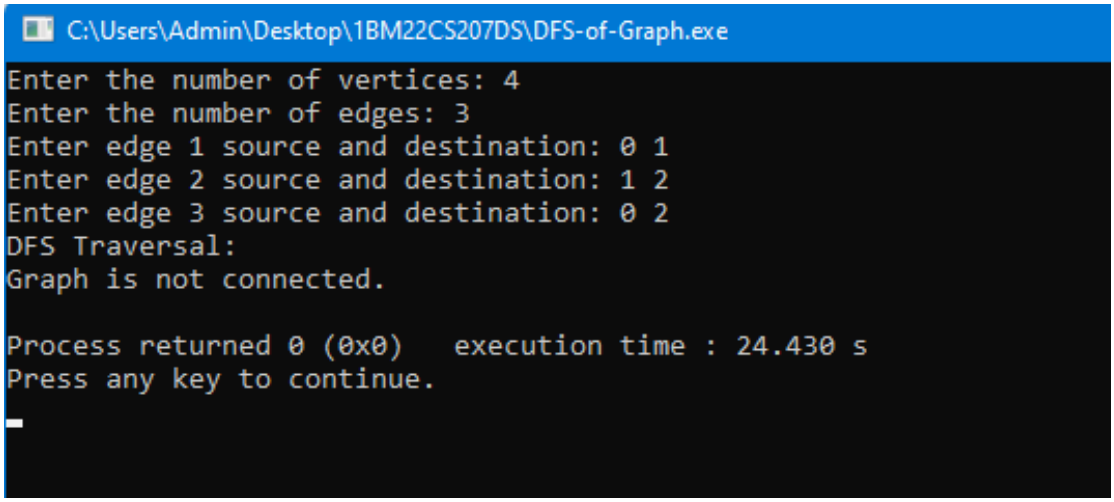
Code:

```
#include <stdio.h>
#include <stdbool.h>
#define MAX_SIZE 100

// Graph structure
struct Graph {
   int vertices;
   bool adjMatrix[MAX_SIZE][MAX_SIZE];
   bool visited[MAX_SIZE];
};

void createGraph(struct Graph* graph, int vertices)
{
    graph->vertices = vertices;

    for (int i = 0; i < vertices; i++)
    {
      graph->visited[i] = false;
      for (int j = 0; j < vertices; j++)
      {
          graph->adjMatrix[i][j] = false;
       }
   }
}
```

```
void addEdge(struct Graph* graph, int src, int dest)
{
   graph->adjMatrix[src][dest] = true;
   graph->adjMatrix[dest][src] = true;
}

// Function to perform DFS traversal
void DFS(struct Graph* graph, int vertex, int* count)
{
    (*count)++;
    graph->visited[vertex] = true;
    for (int i = 0; i < graph->vertices; i++)
   {
        if (graph->adjMatrix[vertex][i] && !graph->visited[i])
       {
           DFS(graph, i, count);
       }
   }
}

int main()
{
              struct Graph graph;
              int vertices, edges;
              printf("Enter the number of vertices: ");
              scanf("%d", &vertices);
              createGraph(&graph, vertices);
              printf("Enter the number of edges: ");
              scanf("%d", &edges);

              for (int i = 0; i < edges; i++)
              {
                  int src, dest;
                  printf("Enter edge %d source and destination: ", i+1);
                  scanf("%d %d", &src, &dest);
                  addEdge(&graph, src, dest);
              }
              int count = 0;
              printf("DFS Traversal: ");
              DFS(&graph, 0, &count); // Starting DFS from vertex 0
              if (count == vertices)
                  printf("\nGraph is connected.\n");
              else
                  printf("\nGraph is not connected.\n");
```

```
    return 0;
}
```

Output:



Lab Program 10:

Given a File of N employee records with a set K of Keys(4-digit) which uniquely determine the records in file F.
Assume that file F is maintained in memory by a Hash Table (HT) of m memory locations with L as the set of memory addresses (2-digit) of locations in HT.
Let the keys in K and addresses in L are integers.
Design and develop a Program in C that uses Hash function H: K -> L as H(K)=K mod m (remainder method), and implement hashing technique to map a given key K to the address space L.
Resolve the collision (if any) using linear probing.

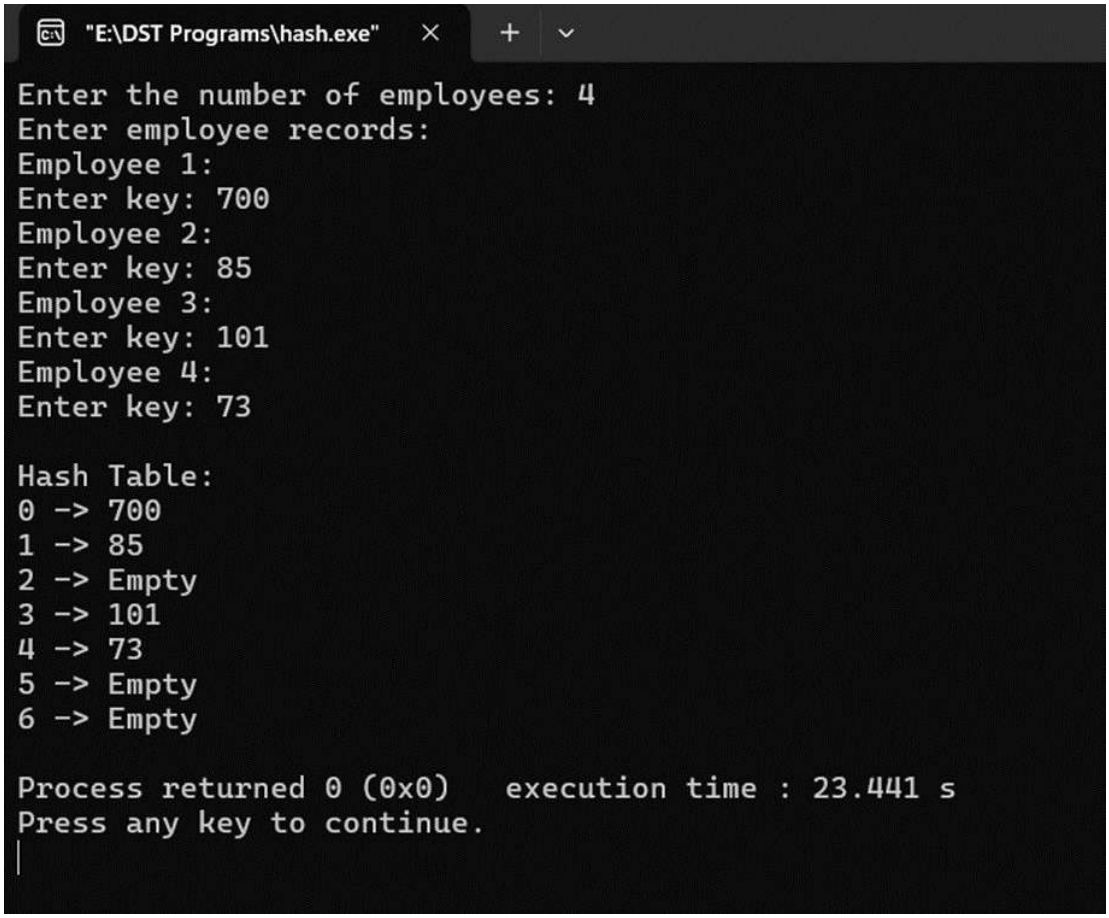
Code:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_EMPLOYEES 100 // Maximum number of employees
#define HASH_TABLE_SIZE 7 // Size of the hash table

// Structure for employee record
struct Employee {
    int key; // 4-digit key
    // Other employee details can be added here
```

```
};

// Function prototypes
int hashFunction(int key);
void insertEmployee(struct Employee employees[], int hashTable[], struct Employee emp);
void displayHashTable(int hashTable[]);

int main()
{
	struct Employee employees[MAX_EMPLOYEES]; // Array to hold employee records
	int hashTable[HASH_TABLE_SIZE] = {0}; // Hash table initialized with 0
	int n, m, i;
	// Input the number of employees
	printf("Enter the number of employees: ");
	scanf("%d", &n);

	// Input employee records
	printf("Enter employee records:\n");
	for (i = 0; i < n; ++i)
	{
		printf("Employee %d:\n", i + 1);
		printf("Enter key: ");
		scanf("%d", &employees[i].key);
		// Additional details can be input here
		insertEmployee(employees, hashTable, employees[i]);
	 }

	// Display the hash table
	 printf("\nHash Table:\n");
	 displayHashTable(hashTable);
	return 0;
}

// Hash function: H(K) = K mod m
int hashFunction(int key)
{
	return key % HASH_TABLE_SIZE;
}

// Function to insert an employee into the hash table
```

```
void insertEmployee(struct Employee employees[], int hashTable[], struct Employee emp)
{
        int index = hashFunction(emp.key);
        // Linear probing to resolve collisions
        while (hashTable[index] != 0)
        {
                index = (index + 1) % HASH_TABLE_SIZE;
        }
        // Insert the employee key into the hash table
        hashTable[index] = emp.key;
}

// Function to display the hash table
void displayHashTable(int hashTable[])
{
        int i;
        for (i = 0; i < HASH_TABLE_SIZE; ++i) {
      printf("%d -> ", i);
        if (hashTable[i] == 0) {
        printf("Empty\n");
        } else {
        printf("%d\n", hashTable[i]);
        }
        }
}
```

Output:


"E:\DST Programs\hash.exe"
Enter the number of employees: 4
Enter employee records:
Employee 1:
Enter key: 700
Employee 2:
Enter key: 85
Employee 3:
Enter key: 101
Employee 4:
Enter key: 73
Hash Table:
0 -> 700
1 -> 85
2 -> Empty
3 -> 101
4 -> 73
5 -> Empty
6 -> Empty
Process returned 0 (0x0) execution time : 23.441 s
Press any key to continue.