VISVESVARAYATECHNOLOGICALUNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
On

DATA STRUCTURES (23CS3PCDST)

Submitted by

SANA T PATHAN (1BM24CS421)

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
September 2024-January 2025

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 **SANA T PATHAN (1BM24CS421)**, who is 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 2024-25. 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. Lakshmi Neelima M	**Dr. Kavitha Sooda**
Assistant Professor	Professor and Head
Department of CSE	Department of CSE
BMSCE, Bengaluru	BMSCE, Bengaluru

INDEX

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.

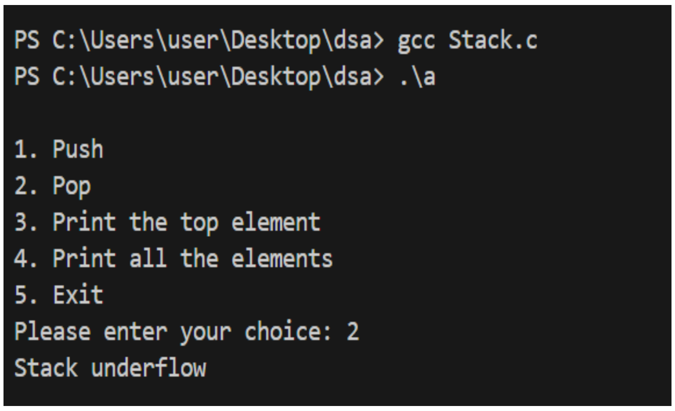
LAB PROGRAM 1:

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.

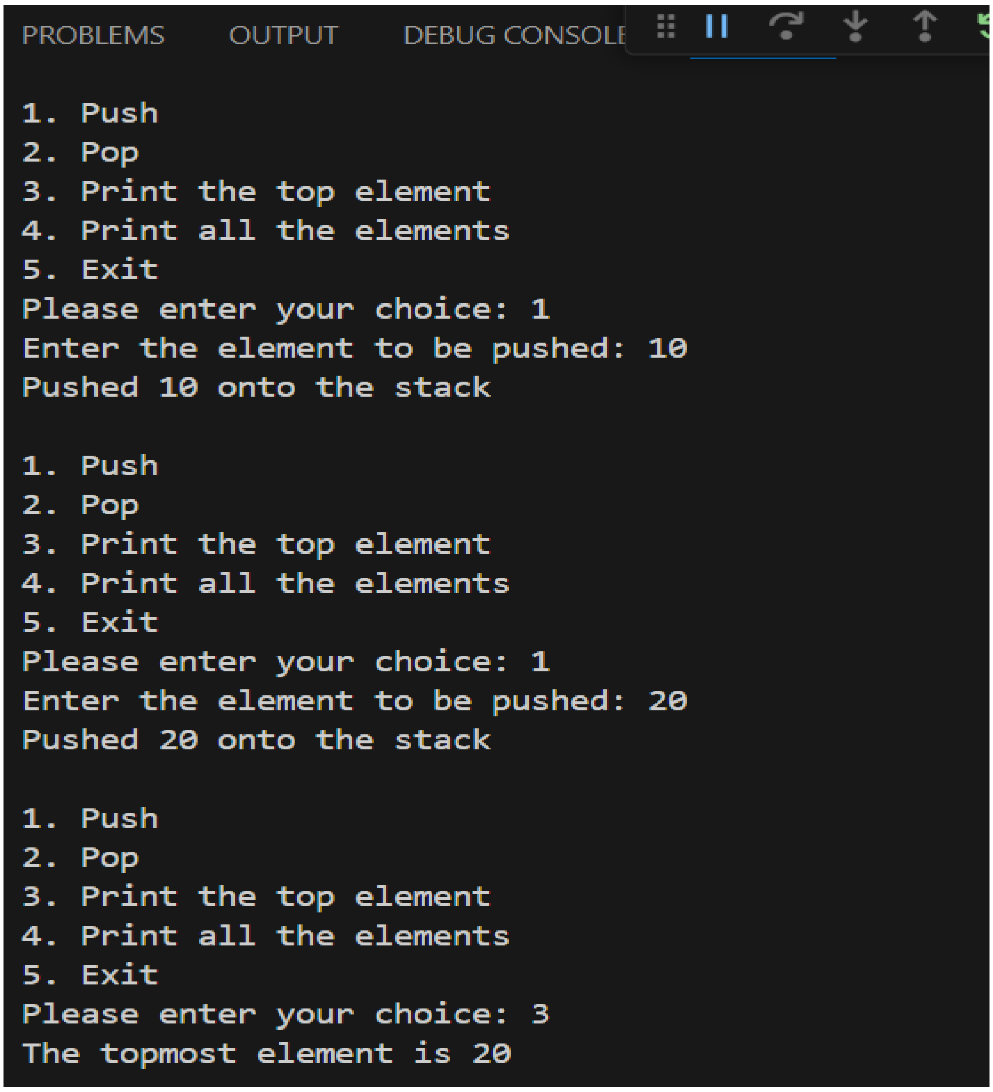
Code:

```c
#include <stdio.h>
#include <stdlib.h>
#define MAX 4

int stack_arr[MAX];
int top = -1;

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

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

void push(int data)
{
    if (isFull())
    {
        printf("Stack overflow\n");
        return;
    }
    top = top + 1;
    stack_arr[top] = data;
    printf("Pushed %d onto the stack\n", data);
}

int pop()
{
    if (isEmpty())
    {
        printf("Stack underflow\n");
        return -1;
    }
    int value = stack_arr[top];
    top = top - 1;
    return value;
}

int peek()
{
    if (isEmpty())
    {
        printf("Stack underflow\n");
        return -1;
    }
    return stack_arr[top];
}

void print()
{
    if (isEmpty())
    {
        printf("Stack is empty\n");
        return;
    }
    printf("Stack elements are:\n");
    for (int i = top; i >= 0; i--)
        printf("%d\n", stack_arr[i]);
}

int main()
{
    int choice, data;

    do
    {
        printf("\n");
        printf("1. Push\n");
        printf("2. Pop\n");
        printf("3. Print the top element\n");
        printf("4. Print all the elements\n");
        printf("5. Exit\n");
        printf("Please enter your choice: ");
        scanf("%d", &choice);

        switch (choice)
        {
        case 1:
            printf("Enter the element to be pushed: ");
            scanf("%d", &data);
            push(data);
            break;

        case 2:
            data = pop();
            if (data != -1)
                printf("The deleted element is %d\n", data);
            break;

        case 3:
            data = peek();
            if (data != -1)
                printf("The topmost element is %d\n", data);
            break;

        case 4:
            print();
            break;

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

        default:
            printf("Wrong choice. Please try again.\n");
        }
    } while (choice != 5);

    return 0;
}
```

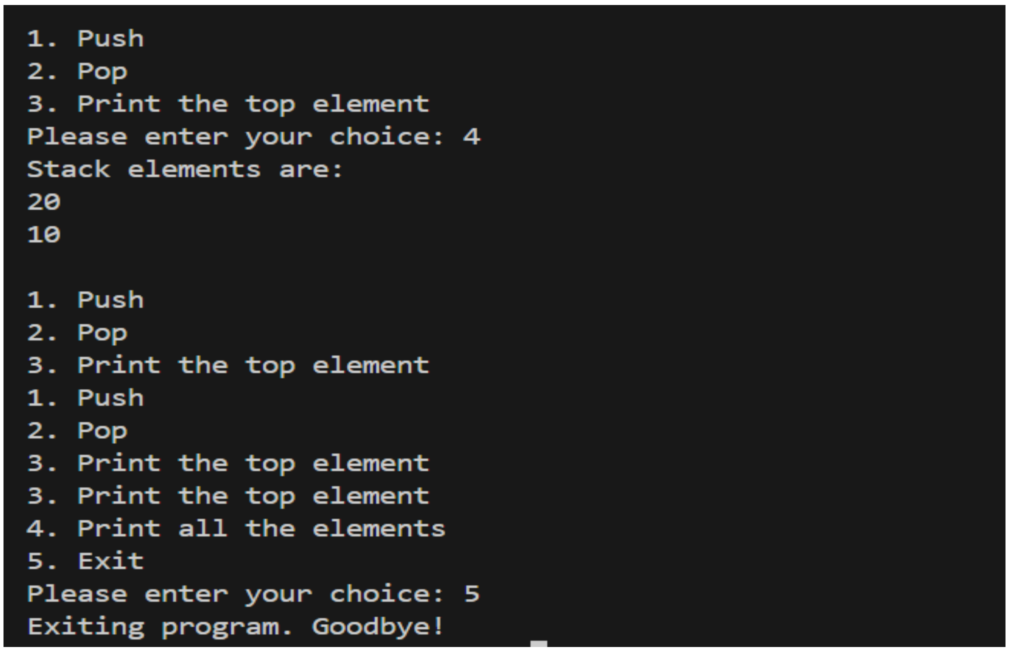
Output:



```
PS C:\Users\user\Desktop\dsa> gcc Stack.c
PS C:\Users\user\Desktop\dsa> .\a

1. Push
2. Pop
3. Print the top element
4. Print all the elements
5. Exit
Please enter your choice: 2
Stack underflow
```



```
1. Push
2. Pop
3. Print the top element
4. Print all the elements
5. Exit
Please enter your choice: 1
Enter the element to be pushed: 10
Pushed 10 onto the stack

1. Push
2. Pop
3. Print the top element
4. Print all the elements
5. Exit
Please enter your choice: 1
Enter the element to be pushed: 20
Pushed 20 onto the stack

1. Push
2. Pop
3. Print the top element
4. Print all the elements
5. Exit
Please enter your choice: 3
The topmost element is 20
```



```
1. Push
2. Pop
3. Print the top element
Please enter your choice: 4
Stack elements are:
20
10

1. Push
2. Pop
3. Print the top element
1. Push
2. Pop
3. Print the top element
3. Print the top element
4. Print all the elements
5. Exit
Please enter your choice: 5
Exiting program. Goodbye!
```

LAB PROGRAM 2

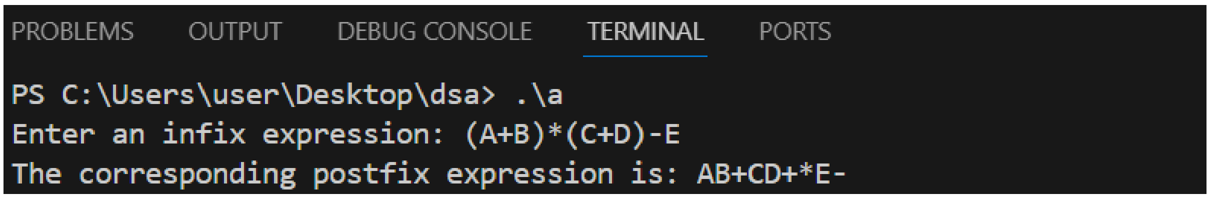
2A) WAP to convert a given valid parenthesized infix arithmetic expression to postfix expression. The expression consists of single character operands and the binary operators + (plus), - (minus), * (multiply) and / (divide)

Code:

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

#define MAX 100

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

// Function to push an element to the stack
void push(char c)
{
    if (top == MAX - 1)
    {
        printf("Stack overflow\n");
        exit(1);
    }
    stack[++top] = c;
}

// Function to pop an element from the stack
char pop()
{
    if (top == -1)
    {
        printf("Stack underflow\n");
        exit(1);
    }
    return stack[top--];
}

// Function to return the precedence of an operator
int precedence(char op)
{
    switch (op)
    {
    case '+':
    case '-':
        return 1;
    case '*':
    case '/':
        return 2;
    case '^':
        return 3;
    default:
        return 0;
    }
}

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

// Function to convert infix expression to postfix expression
void infixToPostfix(char infix[], char postfix[])
{
    int i, j = 0;
    char c;
    for (i = 0; infix[i] != '\0'; i++)
    {
        c = infix[i];

        // If the scanned character is an operand, add it to the postfix
expression
        if (isalnum(c))
        {
            postfix[j++] = c;
        }
        // If the scanned character is '(', push it to the stack
        else if (c == '(')
        {
            push(c);
        }
        // If the scanned character is ')', pop and output until '(' is
encountered
        else if (c == ')')
        {
            while (top != -1 && stack[top] != '(')
            {
                postfix[j++] = pop();
            }
```

```c
            if (top == -1)
            {
                printf("Invalid expression: unmatched parentheses\n");
                exit(1);
            }
            pop(); // Pop '(' from the stack
        }
        // If the scanned character is an operator
        else if (isOperator(c))
        {
            while (top != -1 && precedence(stack[top]) >= precedence(c))
            {
                postfix[j++] = pop();
            }
            push(c);
        }
        else
        {
            printf("Invalid character in expression: %c\n", c);
            exit(1);
        }
    }

    // Pop all the remaining operators from the stack
    while (top != -1)
    {
        if (stack[top] == '(')
        {
            printf("Invalid expression: unmatched parentheses\n");
            exit(1);
        }
        postfix[j++] = pop();
    }

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

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

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

    infixToPostfix(infix, postfix);

    printf("The corresponding postfix expression is: %s\n", postfix);
    return 0;
}
```

Output:



```
PS C:\Users\user\Desktop\dsa> .\a
Enter an infix expression: (A+B)*(C+D)-E
The corresponding postfix expression is: AB+CD+*E-
```

LEET CODE PROGRAM

2B) Given an array nums of size n, return *the majority element*. The majority element is the element that appears more than ⌊n / 2⌋ times. You may assume that the majority element always exists in the array.

Example 1:

Input: nums = [3,2,3]

Output: 3

Example 2:

Input: nums = [2,2,1,1,1,2,2]

Output: 2

Constraints:

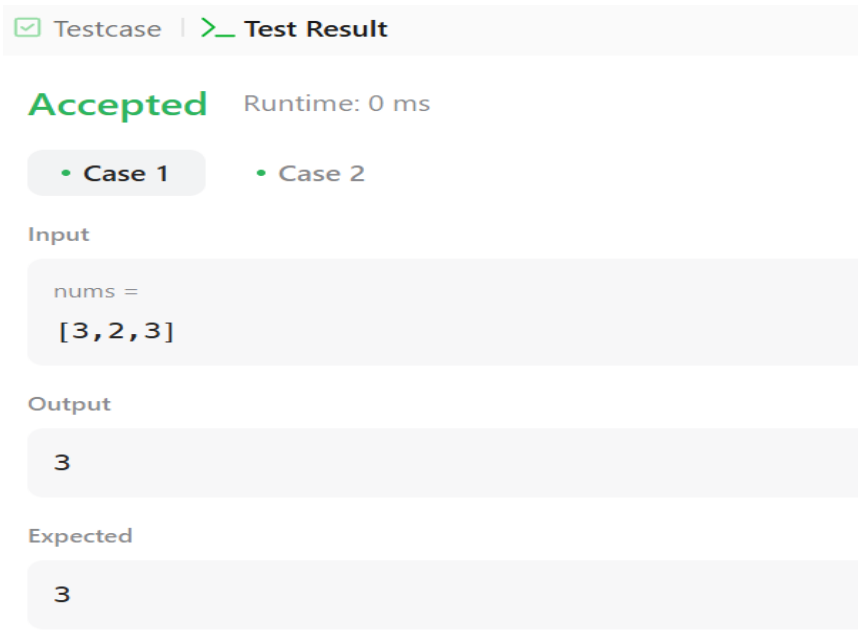
- n == nums.length
- 1 <= n <= 5 * 10^4
- -10^9 <= nums[i] <= 10^9

Code:

```c
int majorityElement(int* nums, int size) {
    for (int i = 0; i < size; i++) {
        int count = 0;
        for (int j = 0; j < size; j++) {
            if (nums[j] == nums[i]) {
                count++;
            }
        }
        if (count > size / 2) {
            return nums[i];
        }
    }
    return -1;
}
```

Output:

Case 1:


Testcase | >_ Test Result

Accepted Runtime: 0 ms

• Case 1 • Case 2

Input

```
nums =
[3,2,3]
```

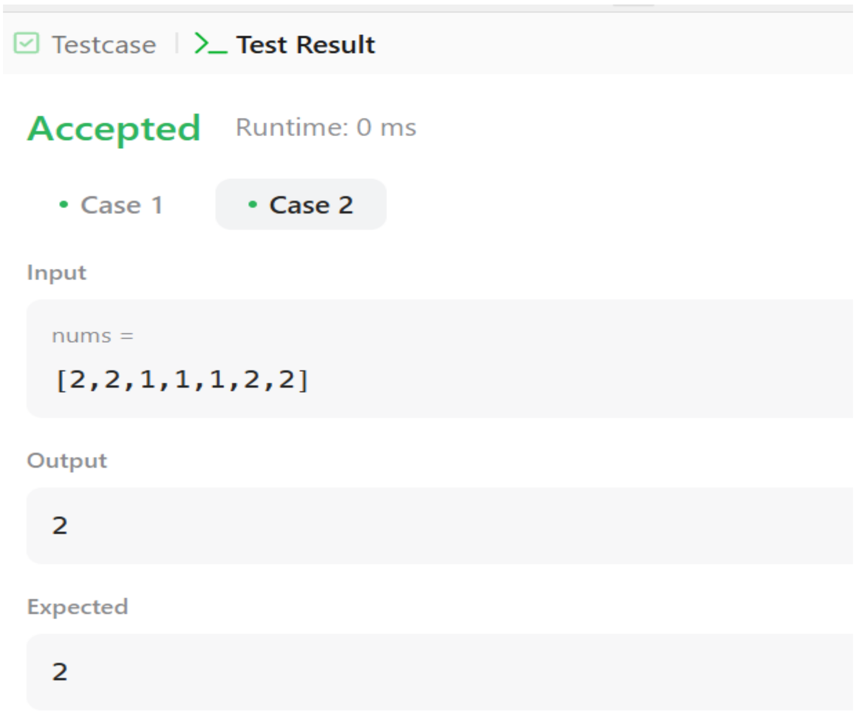
Output

```
3
```

Expected

```
3
```

Case 2:


Testcase | >_ Test Result

Accepted Runtime: 0 ms

• Case 1 • Case 2

Input

```
nums =
[2,2,1,1,1,2,2]
```

Output

```
2
```

Expected

```
2
```

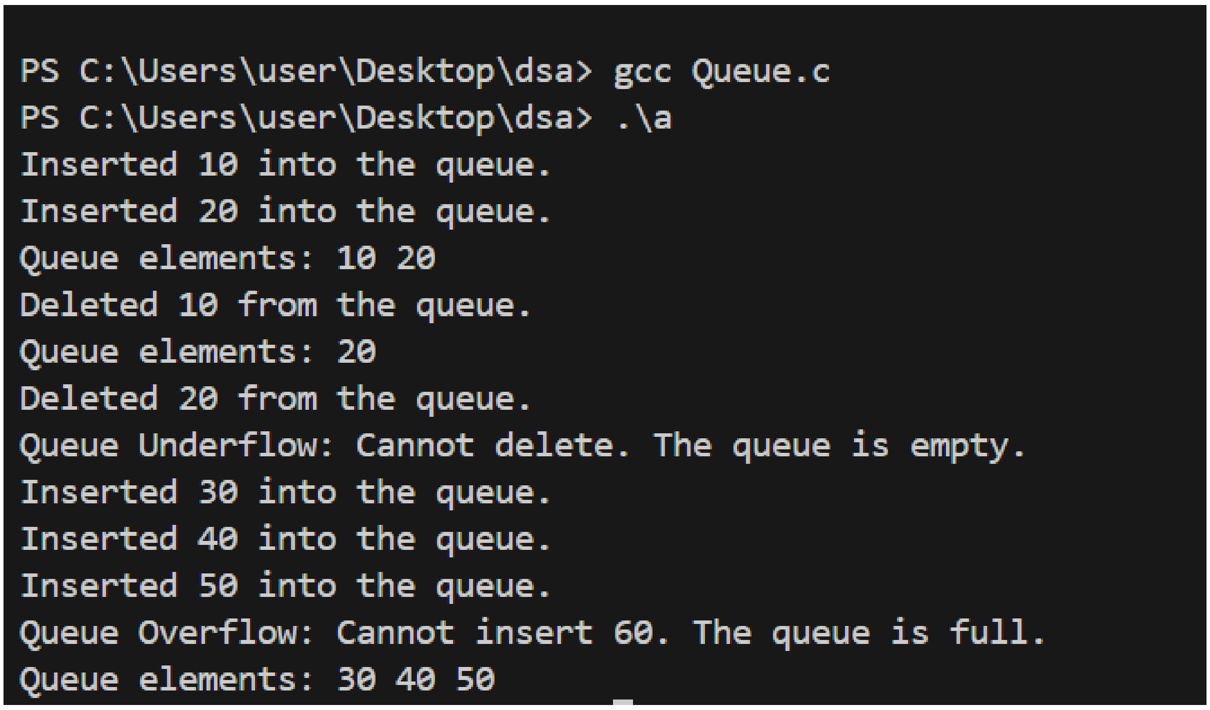
LAB PROGRAM 3

3A) WAP 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:

```c
#include <stdio.h>
#define MAX 5 // Maximum size of the queue

// Queue structure
typedef struct {
    int items[MAX];
    int front;
    int rear;
} Queue;

// Function to initialize the queue
void initializeQueue(Queue *q) {
    q->front = -1;
    q->rear = -1;
}

// Function to check if the queue is full
int isFull(Queue *q) {
    return (q->rear == MAX - 1);
}

// Function to check if the queue is empty
int isEmpty(Queue *q) {
    return (q->front == -1 || q->front > q->rear);
}

// Function to insert an element into the queue
void enqueue(Queue *q, int value) {
    if (isFull(q)) {
        printf("Queue Overflow: Cannot insert %d. The queue is full.\n",
value);
    } else {
        if (q->front == -1) {
            q->front = 0; // Initialize front when first element is added
        }
        q->rear++;
        q->items[q->rear] = value;
        printf("Inserted %d into the queue.\n", value);
    }
}

// Function to delete an element from the queue
void dequeue(Queue *q) {
    if (isEmpty(q)) {
        printf("Queue Underflow: Cannot delete. The queue is empty.\n");
    } else {
        printf("Deleted %d from the queue.\n", q->items[q->front]);
        q->front++;
    }
}

// Function to display the queue
void display(Queue *q) {
    if (isEmpty(q)) {
        printf("The queue is empty.\n");
    } else {
        printf("Queue elements: ");
        for (int i = q->front; i <= q->rear; i++) {
            printf("%d ", q->items[i]);
        }
        printf("\n");
    }
}

int main() {
    Queue q;
    initializeQueue(&q);

    enqueue(&q, 10);
    enqueue(&q, 20);
    display(&q);

    dequeue(&q);
    display(&q);
    dequeue(&q);
    dequeue(&q);


    enqueue(&q,30);
    enqueue(&q, 40);
    enqueue(&q, 50);
    enqueue(&q, 60); // This should trigger overflow
    display(&q);
    return 0;
}
```

Output:



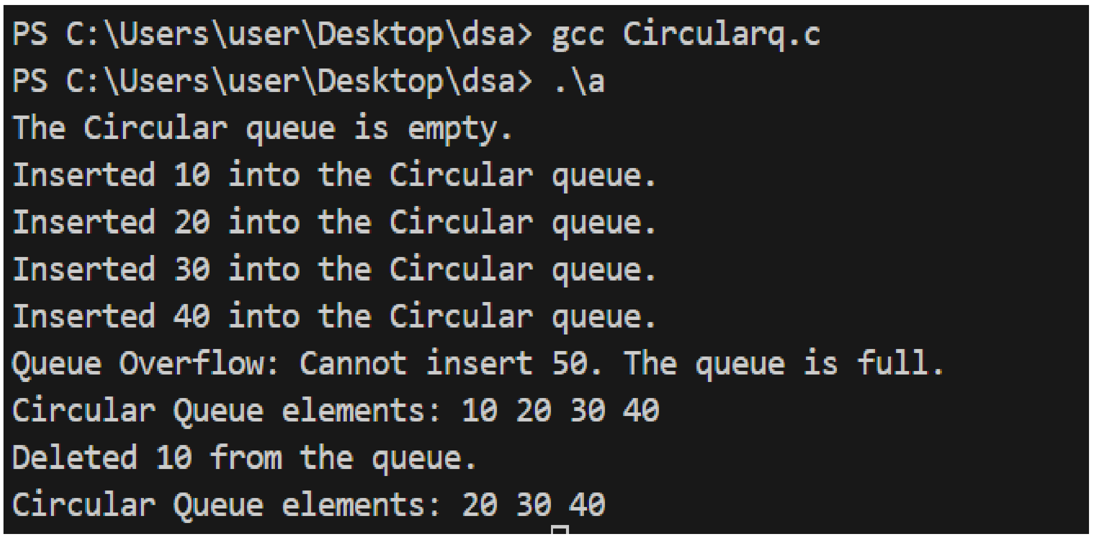
```
PS C:\Users\user\Desktop\dsa> gcc Queue.c
PS C:\Users\user\Desktop\dsa> .\a
Inserted 10 into the queue.
Inserted 20 into the queue.
Queue elements: 10 20
Deleted 10 from the queue.
Queue elements: 20
Deleted 20 from the queue.
Queue Underflow: Cannot delete. The queue is empty.
Inserted 30 into the queue.
Inserted 40 into the queue.
Inserted 50 into the queue.
Queue Overflow: Cannot insert 60. The queue is full.
Queue elements: 30 40 50
```

3B) WAP to simulate the working of a circular 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:

```c
#include <stdio.h>
#define MAX 4

typedef struct {
    int items[MAX];
    int front;
    int rear;
} CircularQueue;

void initializeQueue(CircularQueue *q) {
    q->front = -1;
    q->rear = -1;
}


int isFull(CircularQueue *q) {
    return (q->front == (q->rear + 1) % MAX);
}

int isEmpty(CircularQueue *q) {
    return (q->front == -1);
}

void enqueue(CircularQueue *q, int value) {
    if (isFull(q)) {
        printf("Queue Overflow: Cannot insert %d. The queue is full.\n",
value);
    } else {
        if (q->front == -1) {
            q->front = 0;
        }
        q->rear = (q->rear + 1) % MAX;
        q->items[q->rear] = value;
        printf("Inserted %d into the Circular queue.\n", value);
    }
}


void dequeue(CircularQueue *q) {
    if (isEmpty(q)) {
        printf("Queue Underflow: Cannot delete. The queue is empty.\n");
    } else {
        printf("Deleted %d from the queue.\n", q->items[q->front]);
        if (q->front == q->rear) {

            q->front = -1;
            q->rear = -1;
        } else {
            q->front = (q->front + 1) % MAX;
        }
    }
}


void display(CircularQueue *q) {
    if (isEmpty(q)) {
        printf("The Circular queue is empty.\n");
    } else {
        printf("Circular Queue elements: ");
        for (int i = q->front; ; i = (i + 1) % MAX) {
            printf("%d ", q->items[i]);
            if (i == q->rear) break;
        }
        printf("\n");
    }
}

int main() {
    CircularQueue q;
    initializeQueue(&q);

    display(&q);
    enqueue(&q, 10);
    enqueue(&q, 20);
    enqueue(&q, 30);

    enqueue(&q, 40);
    enqueue(&q, 50);
    display(&q);
    dequeue(&q);
    display(&q);

    return 0;
}
```

Output:



```
PS C:\Users\user\Desktop\dsa> gcc Circularq.c
PS C:\Users\user\Desktop\dsa> .\a
The Circular queue is empty.
Inserted 10 into the Circular queue.
Inserted 20 into the Circular queue.
Inserted 30 into the Circular queue.
Inserted 40 into the Circular queue.
Queue Overflow: Cannot insert 50. The queue is full.
Circular Queue elements: 10 20 30 40
Deleted 10 from the queue.
Circular Queue elements: 20 30 40
```

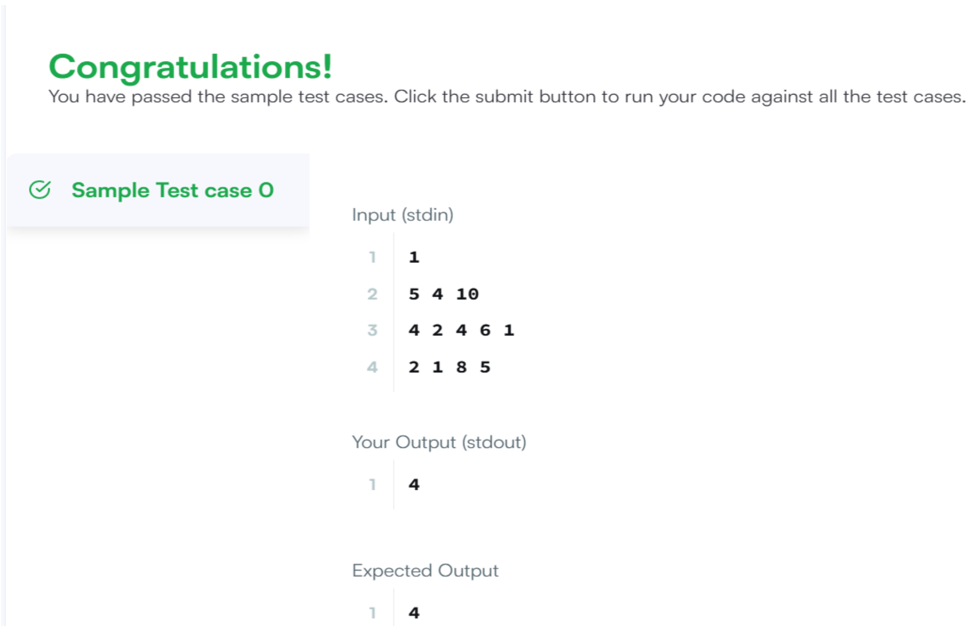
LAB PROGRAM 4

Hacker Rank Program

Code:

```
#include <stdio.h>

int twoStacks(int maxSum, int a[], int n, int b[], int m) {
    int sum = 0, count = 0, i = 0, j = 0;

    // Take as many elements as possible from stack 'a' first
    while (i < n && sum + a[i] <= maxSum) {
        sum += a[i];
        i++;
    }
    count = i;

    // Try adding elements from stack 'b' and adjust from 'a' if
necessary
    while (j < m) {
        sum += b[j];
        j++;

        // If sum exceeds maxSum, remove elements from 'a'
        while (sum > maxSum && i > 0) {
            i--;
            sum -= a[i];
        }

        // Update the maximum count if valid
        if (sum <= maxSum && i + j > count) {
            count = i + j;
        }
    }

    return count;
}

int main() {
    int g;
    scanf("%d", &g); // Number of test cases
    for (int t = 0; t < g; t++) {
        int n, m, maxSum;
        scanf("%d %d %d", &n, &m, &maxSum);

        int a[n], b[m];

        // Input array 'a'
        for (int i = 0; i < n; i++) {
            scanf("%d", &a[i]);
        }

        // Input array 'b'
        for (int i = 0; i < m; i++) {
            scanf("%d", &b[i]);
        }

        // Calculate the result
        int result = twoStacks(maxSum, a, n, b, m);
        printf("%d\n", result); // Print the result for this test
case
    }

    return 0;
}
```

Output:



Congratulations!

You have passed the sample test cases. Click the submit button to run your code against all the test cases.

Sample Test case 0

Input (stdin)

```
1
5 4 10
4 2 4 6 1
2 1 8 5
```

Your Output (stdout)

```
4
```

Expected Output

```
4
```

LAB PROGRAM 5

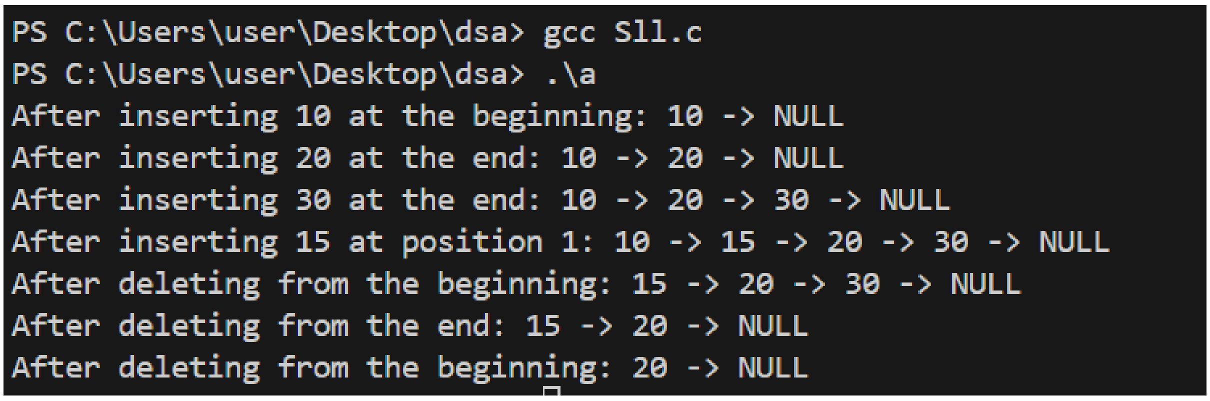
WAP to implement insertion [Beginning, Middle, End] and Deletion [Beginning, Middle, End] in a Singly linked list.

Code:

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

// Define the structure for a node
struct Node {
    int data;
    struct Node* next;
};

// Function prototype for printList
void printList(struct Node* head);

// 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 print the linked list
void printList(struct Node* head) {
    struct Node* temp = head;
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}

// Insert at the beginning
void insertAtBeginning(struct Node** head, int data) {
    struct Node* newNode = createNode(data);
    newNode->next = *head;
    *head = newNode;
    printf("After inserting %d at the beginning: ", data);
    printList(*head);
```

```c
}

// Insert at the end
void insertAtEnd(struct Node** head, int data) {
    struct Node* newNode = createNode(data);
    if (*head == NULL) {
        *head = newNode;
    } else {
        struct Node* temp = *head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newNode;
    }
    printf("After inserting %d at the end: ", data);
    printList(*head);
}

// Insert at a specific position (Middle)
void insertAtMiddle(struct Node** head, int data, int position) {
    struct Node* newNode = createNode(data);
    if (position == 0) {
        insertAtBeginning(head, data);
        return;
    }
    struct Node* temp = *head;
    for (int i = 0; temp != NULL && i < position - 1; i++) {
        temp = temp->next;
    }
    if (temp == NULL) {
        printf("Position out of range.\n");
        return;
    }
    newNode->next = temp->next;
    temp->next = newNode;
    printf("After inserting %d at position %d: ", data, position);
    printList(*head);
}

// Delete from the beginning
void deleteFromBeginning(struct Node** head) {
    if (*head == NULL) {
        printf("The list is empty.\n");
        return;
    }
    struct Node* temp = *head;
    *head = (*head)->next;
    free(temp);
    printf("After deleting from the beginning: ");
    printList(*head);
}

// Delete from the end
void deleteFromEnd(struct Node** head) {
    if (*head == NULL) {
        printf("The list is empty.\n");
        return;
    }
    if ((*head)->next == NULL) {
        free(*head);
        *head = NULL;
        printf("After deleting from the end: ");
        printList(*head);
        return;
    }
    struct Node* temp = *head;
    while (temp->next && temp->next->next != NULL) {
        temp = temp->next;
    }
    free(temp->next);
    temp->next = NULL;
    printf("After deleting from the end: ");
    printList(*head);
}

// Delete from a specific position (Middle)
void deleteFromMiddle(struct Node** head, int position) {
    if (*head == NULL) {
        printf("The list is empty.\n");
        return;
    }
    if (position == 0) {
        deleteFromBeginning(head);
        return;
    }
    struct Node* temp = *head;
    for (int i = 0; temp != NULL && i < position - 1; i++) {
        temp = temp->next;
    }
    if (temp == NULL || temp->next == NULL) {
        printf("Position out of range.\n");
        return;
    }
    struct Node* nodeToDelete = temp->next;
    temp->next = temp->next->next;
    free(nodeToDelete);
    printf("After deleting from position %d: ", position);
    printList(*head);
}

int main() {
    struct Node* head = NULL;

    // Test cases to insert and delete
    insertAtBeginning(&head, 10);
    insertAtEnd(&head, 20);
    insertAtEnd(&head, 30);
    insertAtMiddle(&head, 15, 1);  // Insert at position 1 (middle)

    deleteFromBeginning(&head);  // Delete from beginning
    deleteFromEnd(&head);        // Delete from end
    deleteFromMiddle(&head, 0);  // Delete from position 0 (beginning)

    return 0;
}
```

Output



```
PS C:\Users\user\Desktop\dsa> gcc Sll.c
PS C:\Users\user\Desktop\dsa> .\a
After inserting 10 at the beginning: 10 -> NULL
After inserting 20 at the end: 10 -> 20 -> NULL
After inserting 30 at the end: 10 -> 20 -> 30 -> NULL
After inserting 15 at position 1: 10 -> 15 -> 20 -> 30 -> NULL
After deleting from the beginning: 15 -> 20 -> 30 -> NULL
After deleting from the end: 15 -> 20 -> NULL
After deleting from the beginning: 20 -> NULL
```

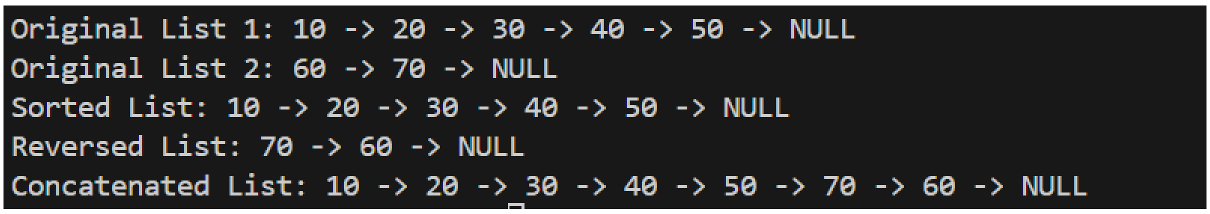
LAB PROGRAM 6

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

Code:

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

// Structure of the node
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));
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

// Function to insert at the end
void insertAtEnd(struct Node** head, int data) {
    struct Node* newNode = createNode(data);
    if (*head == NULL) {
        *head = newNode;
    } else {
        struct Node* temp = *head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newNode;
    }
}

// Function to print the linked list
void printList(struct Node* head) {
    struct Node* temp = head;
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}

// Function to sort the linked list in ascending order (Bubble Sort)
void sortList(struct Node* head) {
    if (head == NULL) return;

    struct Node* i = head;
    struct Node* j;
    int temp;

    while (i != NULL) {
        j = i->next;
        while (j != NULL) {
            if (i->data > j->data) {
                temp = i->data;
                i->data = j->data;
                j->data = temp;
            }
            j = j->next;
        }
        i = i->next;
    }
    printf("Sorted List: ");
    printList(head);
}

// Function to reverse the linked list
void reverseList(struct Node** head) {
    struct Node* prev = NULL;
    struct Node* current = *head;
    struct Node* next = NULL;

    while (current != NULL) {
        next = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }
    *head = prev;
    printf("Reversed List: ");
    printList(*head);
}

// Function to concatenate two linked lists
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;
    printf("Concatenated List: ");
    printList(*head1);
}

int main() {
    struct Node* list1 = NULL;
    struct Node* list2 = NULL;

    // Insert elements into list1
    insertAtEnd(&list1, 10);
    insertAtEnd(&list1, 20);
    insertAtEnd(&list1, 30);
    insertAtEnd(&list1, 40);
    insertAtEnd(&list1, 50);

    // Insert elements into list2
    insertAtEnd(&list2, 60);
    insertAtEnd(&list2, 70);

    // Print initial lists
    printf("Original List 1: ");
    printList(list1);
    printf("Original List 2: ");
    printList(list2);

    // Sort the first list
    sortList(list1);

    // Reverse the second list
    reverseList(&list2);

    // Concatenate list1 and list2
    concatenateLists(&list1, &list2);

    return 0;
}
```

Output:



```
Original List 1: 10 -> 20 -> 30 -> 40 -> 50 -> NULL
Original List 2: 60 -> 70 -> NULL
Sorted List: 10 -> 20 -> 30 -> 40 -> 50 -> NULL
Reversed List: 70 -> 60 -> NULL
Concatenated List: 10 -> 20 -> 30 -> 40 -> 50 -> 70 -> 60 -> NULL
```

6B) WAP to Implement Single Link List to simulate Stack & Queue Operations.

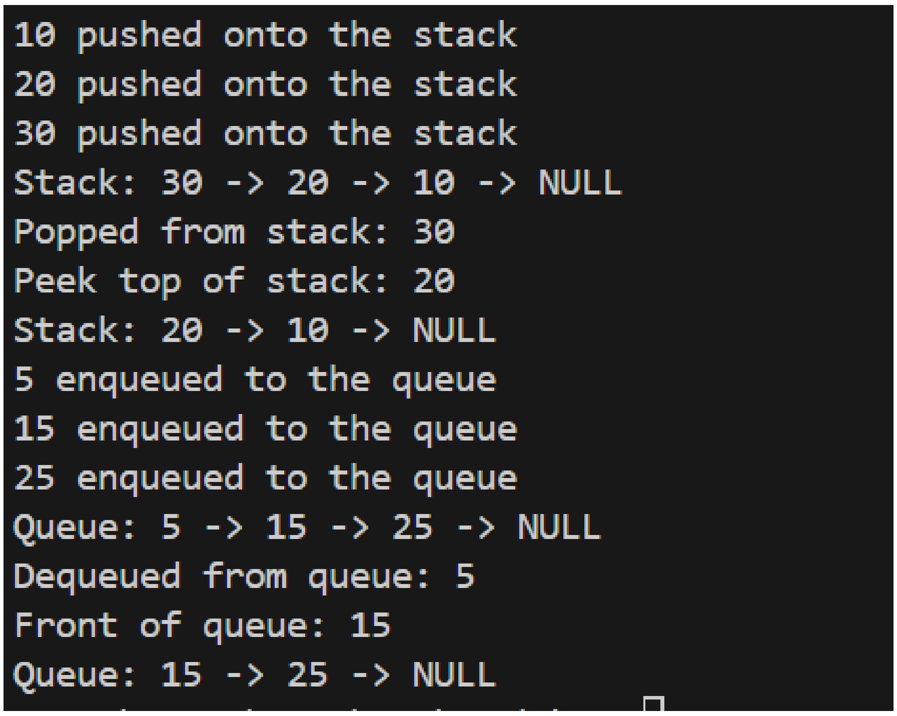
Code:

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

// Structure for the node
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));
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

// Stack Operations

// Function to push an element onto the stack (insert at beginning)
void push(struct Node** top, int data) {
    struct Node* newNode = createNode(data);
    newNode->next = *top;
    *top = newNode;
    printf("%d pushed onto the stack\n", data);
}

// Function to pop an element from the stack (remove from beginning)
int pop(struct Node** top) {
    if (*top == NULL) {
        printf("Stack Underflow\n");
        return -1;
    }
    struct Node* temp = *top;
    int data = temp->data;
    *top = (*top)->next;
    free(temp);
    return data;
}

// Function to peek the top element of the stack
int peek(struct Node* top) {
    if (top == NULL) {
        printf("Stack is empty\n");
        return -1;
    }
    return top->data;
}

// Queue Operations

// Function to enqueue an element into the queue (insert at the end)
void enqueue(struct Node** front, struct Node** rear, int data) {
    struct Node* newNode = createNode(data);
    if (*rear == NULL) {
        *front = *rear = newNode;
        printf("%d enqueued to the queue\n", data);
        return;
    }
    (*rear)->next = newNode;
    *rear = newNode;
    printf("%d enqueued to the queue\n", data);
}

// Function to dequeue an element from the queue (remove from beginning)
int dequeue(struct Node** front) {
    if (*front == NULL) {
        printf("Queue Underflow\n");
        return -1;
    }
    struct Node* temp = *front;
    int data = temp->data;
    *front = (*front)->next;
    if (*front == NULL) {
        printf("Queue is now empty\n");
    }
    free(temp);
    return data;
```

```
}

// Function to get the front element of the queue
int front(struct Node* front) {
    if (front == NULL) {
        printf("Queue is empty\n");
        return -1;
    }
    return front->data;
}

// Function to print the stack
void printStack(struct Node* top) {
    if (top == NULL) {
        printf("Stack is empty\n");
        return;
    }
    printf("Stack: ");
    struct Node* temp = top;
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}

// Function to print the queue
void printQueue(struct Node* front) {
    if (front == NULL) {
        printf("Queue is empty\n");
        return;
    }
    printf("Queue: ");
    struct Node* temp = front;
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}

int main() {
    struct Node* stackTop = NULL;  // Top of the stack
    struct Node* queueFront = NULL;  // Front of the queue
    struct Node* queueRear = NULL;   // Rear of the queue

    // Stack Operations
    push(&stackTop, 10);
```

```c
    push(&stackTop, 20);
    push(&stackTop, 30);
    printStack(stackTop);

    printf("Popped from stack: %d\n", pop(&stackTop));
    printf("Peek top of stack: %d\n", peek(stackTop));
    printStack(stackTop);

    // Queue Operations
    enqueue(&queueFront, &queueRear, 5);
    enqueue(&queueFront, &queueRear, 15);
    enqueue(&queueFront, &queueRear, 25);
    printQueue(queueFront);

    printf("Dequeued from queue: %d\n", dequeue(&queueFront));
    printf("Front of queue: %d\n", front(queueFront));
    printQueue(queueFront);

    return 0;
}
```

Output:



```
10 pushed onto the stack
20 pushed onto the stack
30 pushed onto the stack
Stack: 30 -> 20 -> 10 -> NULL
Popped from stack: 30
Peek top of stack: 20
Stack: 20 -> 10 -> NULL
5 enqueued to the queue
15 enqueued to the queue
25 enqueued to the queue
Queue: 5 -> 15 -> 25 -> NULL
Dequeued from queue: 5
Front of queue: 15
Queue: 15 -> 25 -> NULL
```

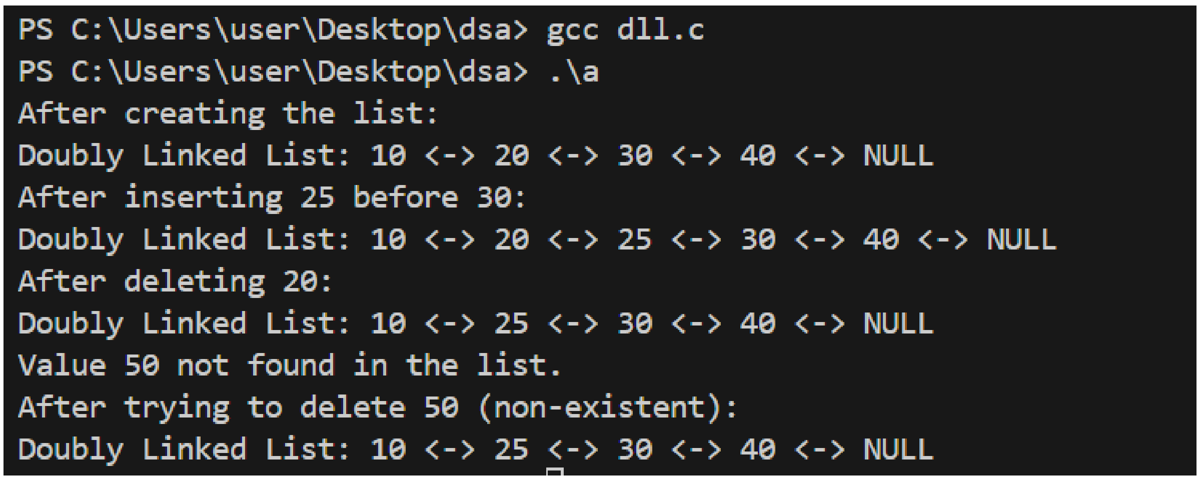
LAB PROGRAM 7

7A) WAP 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:

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

// Structure for the node of the doubly linked list
struct Node {
    int data;
    struct Node* next;
    struct Node* prev;
};

// 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;
    newNode->prev = NULL;
    return newNode;
}

// Function to insert a new node at the end of the doubly linked list
void insertEnd(struct Node** head, int data) {
    struct Node* newNode = createNode(data);
    if (*head == NULL) {
        *head = newNode;
        return;
    }
    struct Node* temp = *head;
    while (temp->next != NULL) {
        temp = temp->next;
    }
    temp->next = newNode;
    newNode->prev = temp;
}

// Function to insert a new node before a specific node
void insertBefore(struct Node** head, int value, int data) {
    struct Node* temp = *head;
    while (temp != NULL && temp->data != value) {
        temp = temp->next;
    }
    if (temp == NULL) {
        printf("Value %d not found in the list.\n", value);
        return;
    }
    struct Node* newNode = createNode(data);
    newNode->next = temp;
    newNode->prev = temp->prev;
    if (temp->prev != NULL) {
        temp->prev->next = newNode;
    } else {
        *head = newNode;
    }
    temp->prev = newNode;
}

// Function to delete a node based on a specific value
void deleteNode(struct Node** head, int value) {
    struct Node* temp = *head;
    while (temp != NULL && temp->data != value) {
        temp = temp->next;
    }
    if (temp == NULL) {
        printf("Value %d not found in the list.\n", value);
        return;
    }
    if (temp->prev != NULL) {
        temp->prev->next = temp->next;
    } else {
        *head = temp->next;  // If deleting the head node
    }
    if (temp->next != NULL) {
        temp->next->prev = temp->prev;
    }
    free(temp);
}

// Function to display the contents of the doubly linked list
void display(struct Node* head) {
    if (head == NULL) {
        printf("The list is empty.\n");
        return;
    }
    struct Node* temp = head;
    printf("Doubly Linked List: ");
    while (temp != NULL) {
        printf("%d <-> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}

int main() {
    struct Node* head = NULL;

    // Creating a doubly linked list with hardcoded values
    insertEnd(&head, 10);
    insertEnd(&head, 20);
    insertEnd(&head, 30);
    insertEnd(&head, 40);

    // Display the list after creation
    printf("After creating the list:\n");
    display(head);

    // Insert 25 before the node with value 30
    insertBefore(&head, 30, 25);
    printf("After inserting 25 before 30:\n");
    display(head);

    // Delete the node with value 20
    deleteNode(&head, 20);
    printf("After deleting 20:\n");
    display(head);

    // Delete a non-existent node with value 50
    deleteNode(&head, 50);
    printf("After trying to delete 50 (non-existent):\n");
    display(head);

    return 0;
}
```

Output:

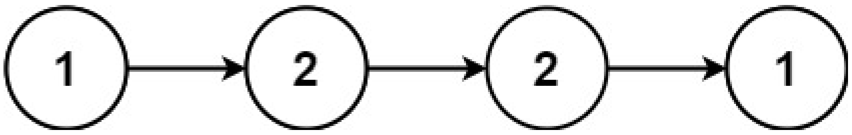


```
PS C:\Users\user\Desktop\dsa> gcc dll.c
PS C:\Users\user\Desktop\dsa> .\a
After creating the list:
Doubly Linked List: 10 <-> 20 <-> 30 <-> 40 <-> NULL
After inserting 25 before 30:
Doubly Linked List: 10 <-> 20 <-> 25 <-> 30 <-> 40 <-> NULL
After deleting 20:
Doubly Linked List: 10 <-> 25 <-> 30 <-> 40 <-> NULL
Value 50 not found in the list.
After trying to delete 50 (non-existent):
Doubly Linked List: 10 <-> 25 <-> 30 <-> 40 <-> NULL
```

LEET CODE PROGRAM

7B) Given the head of a singly linked list, return true if it is a palindrome or false otherwise.

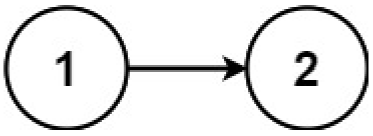
Example 1:



Input: head = [1,2,2,1]

Output: true

Example 2:



Input: head = [1,2]

Output: false

Constraints:

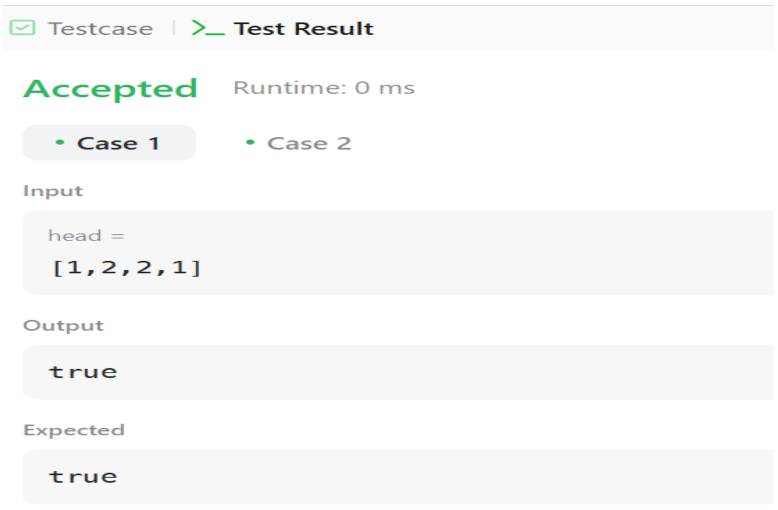
- The number of nodes in the list is in the range [1, 10^5].
- 0 <= Node.val <= 9

Code:

```c
#include <stdbool.h>


/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     struct ListNode *next;
 * };
 */

// Function to reverse a linked list
struct ListNode* reverseList(struct ListNode* head) {
    struct ListNode* prev = NULL;
    struct ListNode* current = head;
    struct ListNode* next = NULL;

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

    return prev;
}

// Function to check if the linked list is a palindrome
bool isPalindrome(struct ListNode* head) {
    if (head == NULL || head->next == NULL) {
        return true; // Empty list or single node list is a palindrome
    }

    // Step 1: Find the middle of the linked list using slow and fast pointers
    struct ListNode *slow = head, *fast = head;

    // Move slow to the middle and fast to the end
    while (fast != NULL && fast->next != NULL) {
        slow = slow->next;
        fast = fast->next->next;
    }

    // Step 2: Reverse the second half of the list starting from slow
    struct ListNode* secondHalf = reverseList(slow);

    // Step 3: Compare the first and second halves
    struct ListNode* firstHalf = head;
    while (secondHalf != NULL) {
        if (firstHalf->val != secondHalf->val) {
            return false; // If any values don't match, return false
        }
        firstHalf = firstHalf->next;
        secondHalf = secondHalf->next;
    }

    return true; // All values matched
}
```

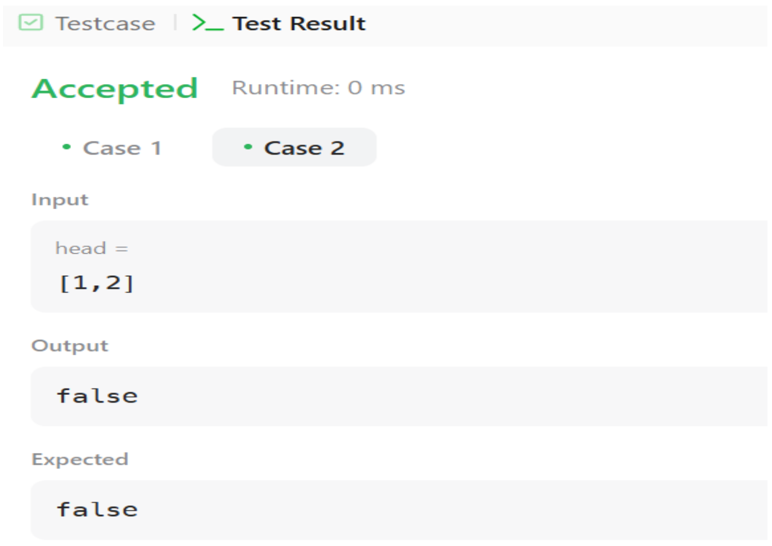
Output:

Case 1:



```
Testcase | >_ Test Result

Accepted    Runtime: 0 ms

• Case 1    • Case 2

Input
head =
[1,2,2,1]

Output
true

Expected
true
```

Case 2:



```
Testcase | >_ Test Result

Accepted    Runtime: 0 ms

• Case 1    • Case 2

Input
head =
[1,2]

Output
false

Expected
false
```

LAB PROGRAM 8

Write a program

a) To construct a binary Search tree.

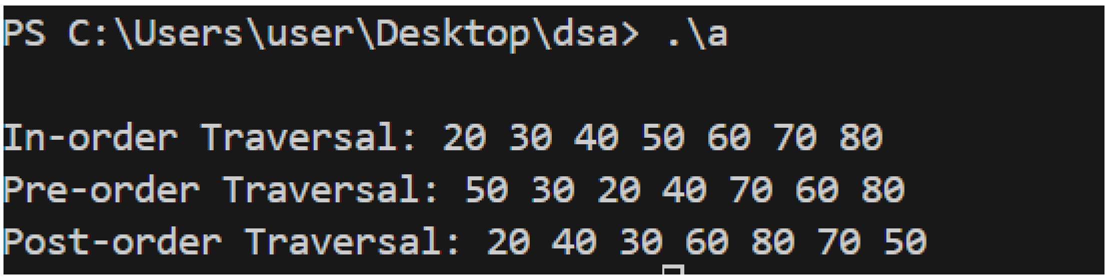
b) To traverse the tree using all the methods i.e., in-order, preorder and post order

c) To display the elements in the tree.

Code:

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

// Structure for a node in the binary search tree
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));
    newNode->data = data;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}

// Function to insert a new node in the BST
struct Node* insert(struct Node* root, int data) {
    if (root == NULL) {
        return createNode(data);
    }

    // Insert in the left subtree if the data is less than root
    if (data < root->data) {
        root->left = insert(root->left, data);
    }
    // Insert in the right subtree if the data is greater than root
    else if (data > root->data) {
        root->right = insert(root->right, data);
    }

    return root;
}

// In-order traversal (Left, Root, Right)
void inOrder(struct Node* root) {
    if (root != NULL) {
        inOrder(root->left);
        printf("%d ", root->data);
        inOrder(root->right);
    }
}

// Pre-order traversal (Root, Left, Right)
void preOrder(struct Node* root) {
    if (root != NULL) {
        printf("%d ", root->data);
        preOrder(root->left);
        preOrder(root->right);
    }
}

// Post-order traversal (Left, Right, Root)
void postOrder(struct Node* root) {
    if (root != NULL) {
        postOrder(root->left);
        postOrder(root->right);
        printf("%d ", root->data);
    }
}

// Function to display the tree elements using In-order, Pre-order, and Post-
order
void display(struct Node* root) {
    printf("\nIn-order Traversal: ");
    inOrder(root);

    printf("\nPre-order Traversal: ");
    preOrder(root);

    printf("\nPost-order Traversal: ");
    postOrder(root);
}

// Main function
int main() {
    struct Node* root = NULL;
    // Construct the binary search tree
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);
    insert(root, 60);
    insert(root, 80);

    // Display the elements in the tree
    display(root);

    return 0;
}
```

Output:



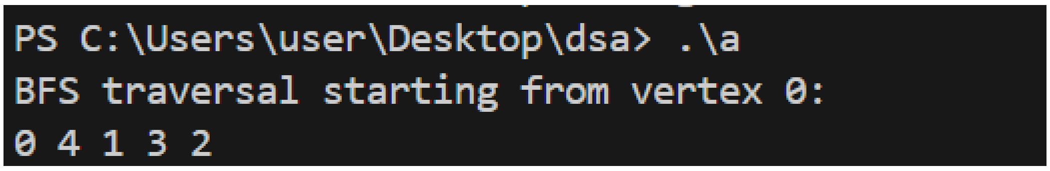
```
PS C:\Users\user\Desktop\dsa> .\a

In-order Traversal: 20 30 40 50 60 70 80
Pre-order Traversal: 50 30 20 40 70 60 80
Post-order Traversal: 20 40 30 60 80 70 50
```

8b) Write a program to traverse a graph using BFS method.

Code:

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

#define MAX_VERTICES 10

// Structure for an adjacency list node
struct Node {
    int vertex;
    struct Node* next;
};

// Structure for the graph
struct Graph {
    int numVertices;
    struct Node* adjLists[MAX_VERTICES];
    bool visited[MAX_VERTICES];
};

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

// Function to initialize a graph
void initGraph(struct Graph* graph, int vertices) {
    graph->numVertices = vertices;
    for (int i = 0; i < vertices; i++) {
        graph->adjLists[i] = NULL;
        graph->visited[i] = false;
    }
}

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

    // Since the graph is undirected, add the reverse edge as well
    newNode = createNode(src);
    newNode->next = graph->adjLists[dest];
    graph->adjLists[dest] = newNode;
}

// Function to perform BFS traversal
void BFS(struct Graph* graph, int startVertex) {
    int queue[MAX_VERTICES];
    int front = -1, rear = -1;

    // Mark the start vertex as visited and enqueue it
    graph->visited[startVertex] = true;
    queue[++rear] = startVertex;

    printf("BFS traversal starting from vertex %d:\n", startVertex);

    while (front != rear) {
        // Dequeue a vertex from the queue
        int currentVertex = queue[++front];

        printf("%d ", currentVertex);

        // Get all the adjacent vertices of the dequeued vertex
        struct Node* temp = graph->adjLists[currentVertex];
        while (temp) {
            int adjVertex = temp->vertex;
            if (!graph->visited[adjVertex]) {
                // Mark the adjacent vertex as visited and enqueue it
                graph->visited[adjVertex] = true;
                queue[++rear] = adjVertex;
            }
            temp = temp->next;
        }
    }
    printf("\n");
}

int main() {
    struct Graph graph;
    int vertices = 5;

    initGraph(&graph, vertices);

    // Add edges to the graph
    addEdge(&graph, 0, 1);
    addEdge(&graph, 0, 4);
    addEdge(&graph, 1, 2);
    addEdge(&graph, 1, 3);
    addEdge(&graph, 2, 3);
    addEdge(&graph, 3, 4);
    // Perform BFS starting from vertex 0
    BFS(&graph, 0);

    return 0;
}
```

Output:



```
PS C:\Users\user\Desktop\dsa> .\a
BFS traversal starting from vertex 0:
0 4 1 3 2
```

LAB PROGRAM 9

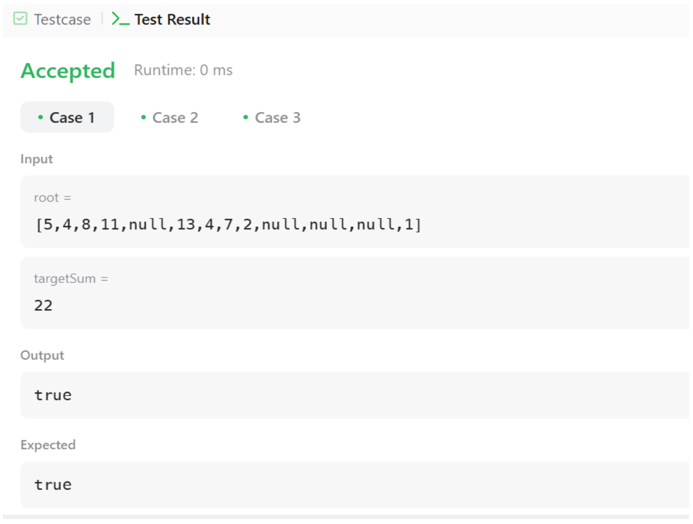
9A) Leet code program: Given the root of a binary tree and an integer target Sum, return true if the tree has a root-to-leaf path such that adding up all the values along the path equals targetSum.

Code:

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     struct TreeNode *left;
 *     struct TreeNode *right;
 * };
 */
bool hasPathSum(struct TreeNode* root, int targetSum) {
    // Base case: if the root is NULL, return false (no path exists)
    if (root == NULL) {
        return false;
    }

    // If we've reached a leaf node, check if the target sum is achieved
    if (root->left == NULL && root->right == NULL) {
        return (root->val == targetSum);
    }

    // Subtract the current node's value from the target sum
    targetSum -= root->val;

    // Recursively check the left and right subtrees
    return hasPathSum(root->left, targetSum) || hasPathSum(root->right,
targetSum);
}
```

Output:

Case 1:



Testcase | >_ Test Result

Accepted Runtime: 0 ms

• Case 1 • Case 2 • Case 3

Input

```
root =
[5,4,8,11,null,13,4,7,2,null,null,null,1]
```

```
targetSum =
22
```

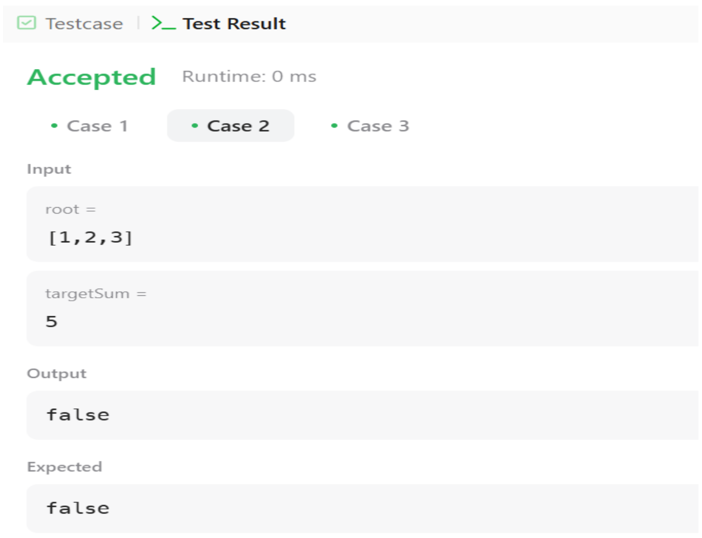
Output

```
true
```

Expected

```
true
```

Case 2:



Testcase | >_ Test Result

Accepted Runtime: 0 ms

• Case 1 • Case 2 • Case 3

Input

```
root =
[1,2,3]
```

```
targetSum =
5
```

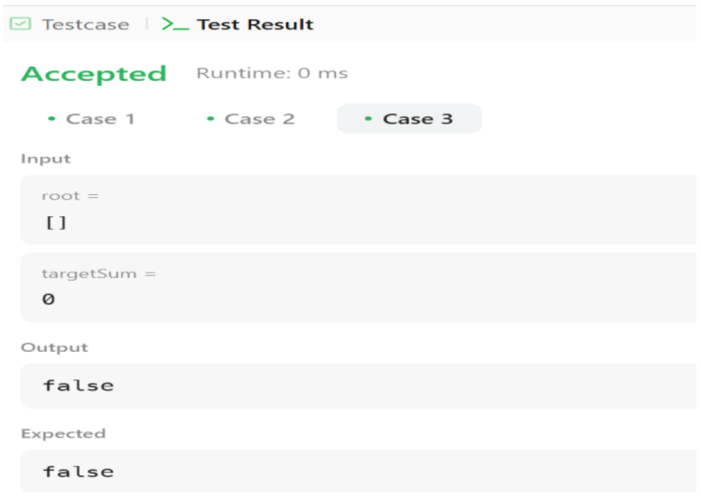
Output

```
false
```

Expected

```
false
```

Case 3:



Testcase | >_ Test Result

Accepted Runtime: 0 ms

• Case 1 • Case 2 • Case 3

Input

```
root =
[]
```

```
targetSum =
0
```

Output

```
false
```

Expected

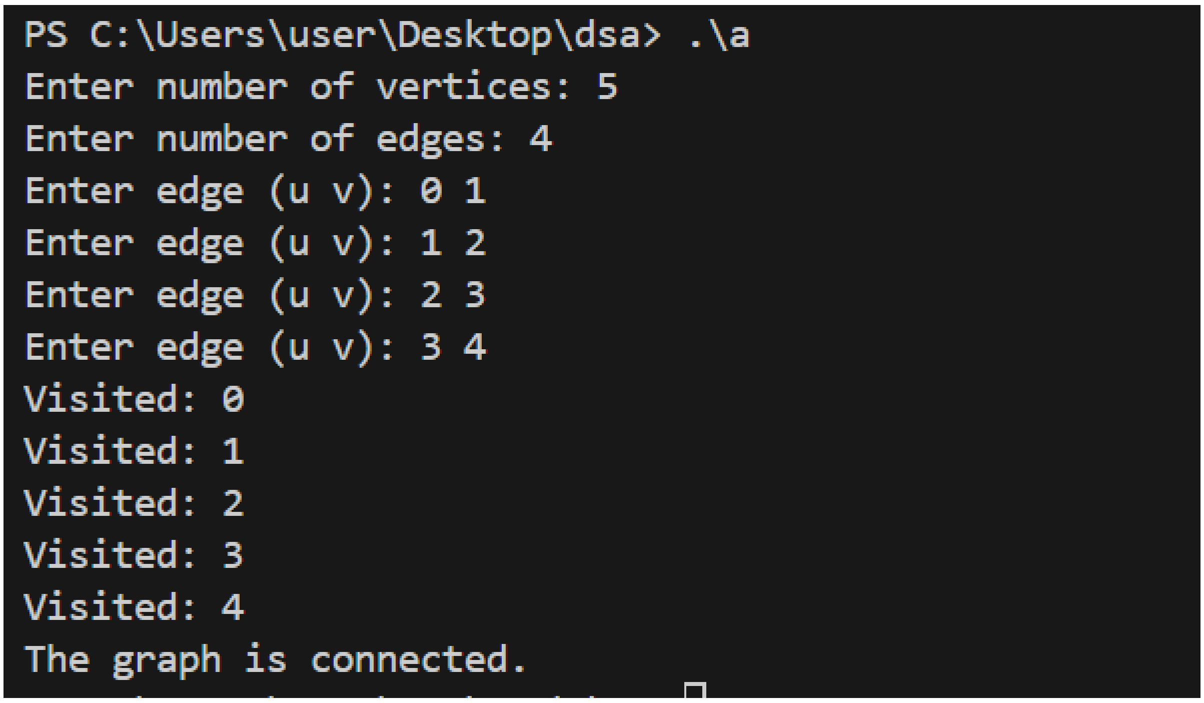
```
false
```

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

Code:

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

#define MAX_VERTICES 10 // Max number of vertices in the graph

// Structure to represent the graph
struct Graph {
    int vertices;
    int adj[MAX_VERTICES][MAX_VERTICES];
};

// Function to initialize the graph
void initGraph(struct Graph* g, int vertices) {
    g->vertices = vertices;
    for (int i = 0; i < vertices; i++) {
        for (int j = 0; j < vertices; j++) {
            g->adj[i][j] = 0; // Initialize all edges as 0 (no edge)
        }
    }
}

// Function to add an edge in the graph (undirected graph)
void addEdge(struct Graph* g, int u, int v) {
    g->adj[u][v] = 1;
    g->adj[v][u] = 1; // Since the graph is undirected
}

// DFS function to traverse the graph
void DFS(struct Graph* g, int vertex, int visited[]) {
    visited[vertex] = 1;
    printf("Visited: %d\n", vertex);

    for (int i = 0; i < g->vertices; i++) {
        if (g->adj[vertex][i] == 1 && !visited[i]) {
            DFS(g, i, visited);
        }
    }
}

// Function to check if the graph is connected
int isConnected(struct Graph* g) {
    int visited[MAX_VERTICES] = {0}; // Array to track visited vertices
    // Start DFS from the first vertex (0th vertex)
    DFS(g, 0, visited);

    // Check if all vertices are visited
    for (int i = 0; i < g->vertices; i++) {
        if (!visited[i]) {
            return 0; // If any vertex is not visited, the graph is not
connected
        }
    }

    return 1; // All vertices are visited, graph is connected
}

int main() {
    struct Graph g;
    int vertices, edges, u, v;

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

    // Initialize the graph
    initGraph(&g, vertices);

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

    // Add edges to the graph
    for (int i = 0; i < edges; i++) {
        printf("Enter edge (u v): ");
        scanf("%d %d", &u, &v);
        addEdge(&g, u, v);
    }

    // Check if the graph is connected
    if (isConnected(&g)) {
        printf("The graph is connected.\n");
    } else {
        printf("The graph is not connected.\n");
    }

    return 0;
}
```

Output:



```
PS C:\Users\user\Desktop\dsa> .\a
Enter number of vertices: 5
Enter number of edges: 4
Enter edge (u v): 0 1
Enter edge (u v): 1 2
Enter edge (u v): 2 3
Enter edge (u v): 3 4
Visited: 0
Visited: 1
Visited: 2
Visited: 3
Visited: 4
The graph is connected.
```

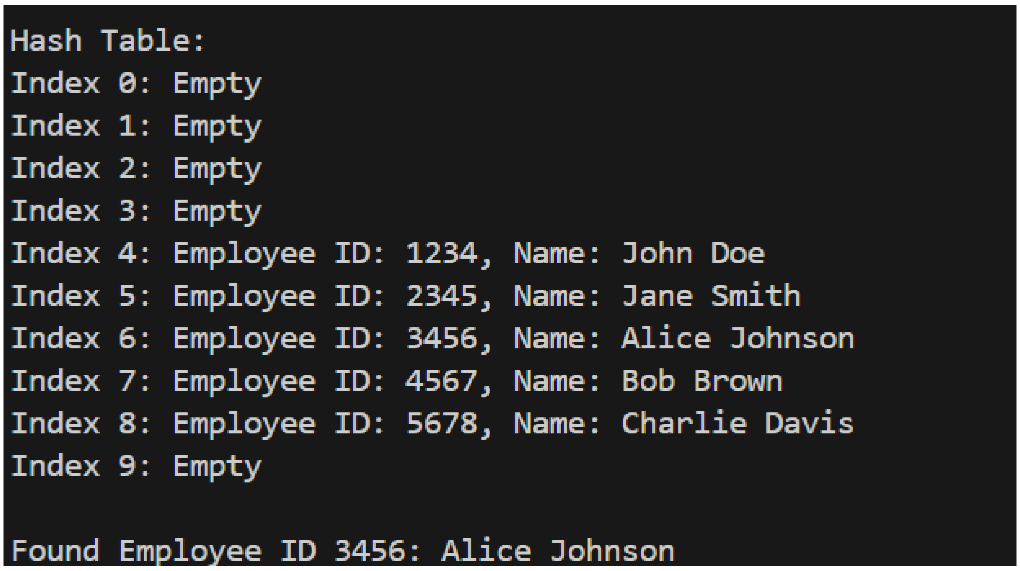
LAB PROGRAM 10

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:

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

#define M 10  // Size of the hash table
#define N 5   // Number of employee records (example)

typedef struct {
    int empID;     // 4-digit unique employee ID (key)
    char name[30]; // Employee name
} Employee;

// Hash Table declaration
Employee *hashTable[M];

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

// Linear probing to resolve collisions
int linearProbing(int key) {
    int index = hashFunction(key);
    int i = 0;

    // Search for an empty slot
    while (hashTable[(index + i) % M] != NULL) {
        i++;
    }

    return (index + i) % M;
}

// Insert an employee record into the hash table
void insert(Employee *emp) {
    int index = hashFunction(emp->empID);

    // If the slot is empty, insert directly
    if (hashTable[index] == NULL) {
        hashTable[index] = emp;
    } else {
        // Collision occurs, resolve using linear probing
        int newIndex = linearProbing(emp->empID);
        hashTable[newIndex] = emp;
    }
}

// Search for an employee by their ID
Employee* search(int empID) {
    int index = hashFunction(empID);
    int i = 0;

    // Linear probing search for the employee
    while (hashTable[(index + i) % M] != NULL) {
        if (hashTable[(index + i) % M]->empID == empID) {
            return hashTable[(index + i) % M];
        }
        i++;
    }

    return NULL;  // Employee not found
}

// Display the hash table
void display() {
    printf("\nHash Table:\n");
    for (int i = 0; i < M; i++) {
        if (hashTable[i] != NULL) {
            printf("Index %d: Employee ID: %d, Name: %s\n", i, hashTable[i]-
>empID, hashTable[i]->name);
        } else {
            printf("Index %d: Empty\n", i);
        }
    }
}

int main() {
    // Initialize hash table to NULL
    for (int i = 0; i < M; i++) {
        hashTable[i] = NULL;
    }

    // Employee records to insert
    Employee emp1 = {1234, "John Doe"};
    Employee emp2 = {2345, "Jane Smith"};
    Employee emp3 = {3456, "Alice Johnson"};
    Employee emp4 = {4567, "Bob Brown"};
    Employee emp5 = {5678, "Charlie Davis"};

    // Insert employees into the hash table
    insert(&emp1);
    insert(&emp2);
    insert(&emp3);
    insert(&emp4);
    insert(&emp5);

    // Display the hash table
    display();

    // Search for an employee
    int searchID = 3456;
    Employee *searchedEmp = search(searchID);
    if (searchedEmp != NULL) {
        printf("\nFound Employee ID %d: %s\n", searchedEmp->empID,
searchedEmp->name);
    } else {
        printf("\nEmployee ID %d not found.\n", searchID);
    }

    return 0;
}
```

Output:



```
Hash Table:
Index 0: Empty
Index 1: Empty
Index 2: Empty
Index 3: Empty
Index 4: Employee ID: 1234, Name: John Doe
Index 5: Employee ID: 2345, Name: Jane Smith
Index 6: Employee ID: 3456, Name: Alice Johnson
Index 7: Employee ID: 4567, Name: Bob Brown
Index 8: Employee ID: 5678, Name: Charlie Davis
Index 9: Empty

Found Employee ID 3456: Alice Johnson
```