VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.

LAB REPORT
On

DATA STRUCTURES (23CS3PCDST)

Submitted by

DHRUHI ATYKAR (1BM23CS091)

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 DHRUHI ATYKAR **(1BM23CS091)**, 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.

Dr. Selva kumar S
Associate Professor
Department of CSE
BMSCE, Bengaluru

Dr. Kavitha Sooda
Professor and Head
Department of CSE
BMSCE, Bengaluru

Index Sheet

Course outcomes:

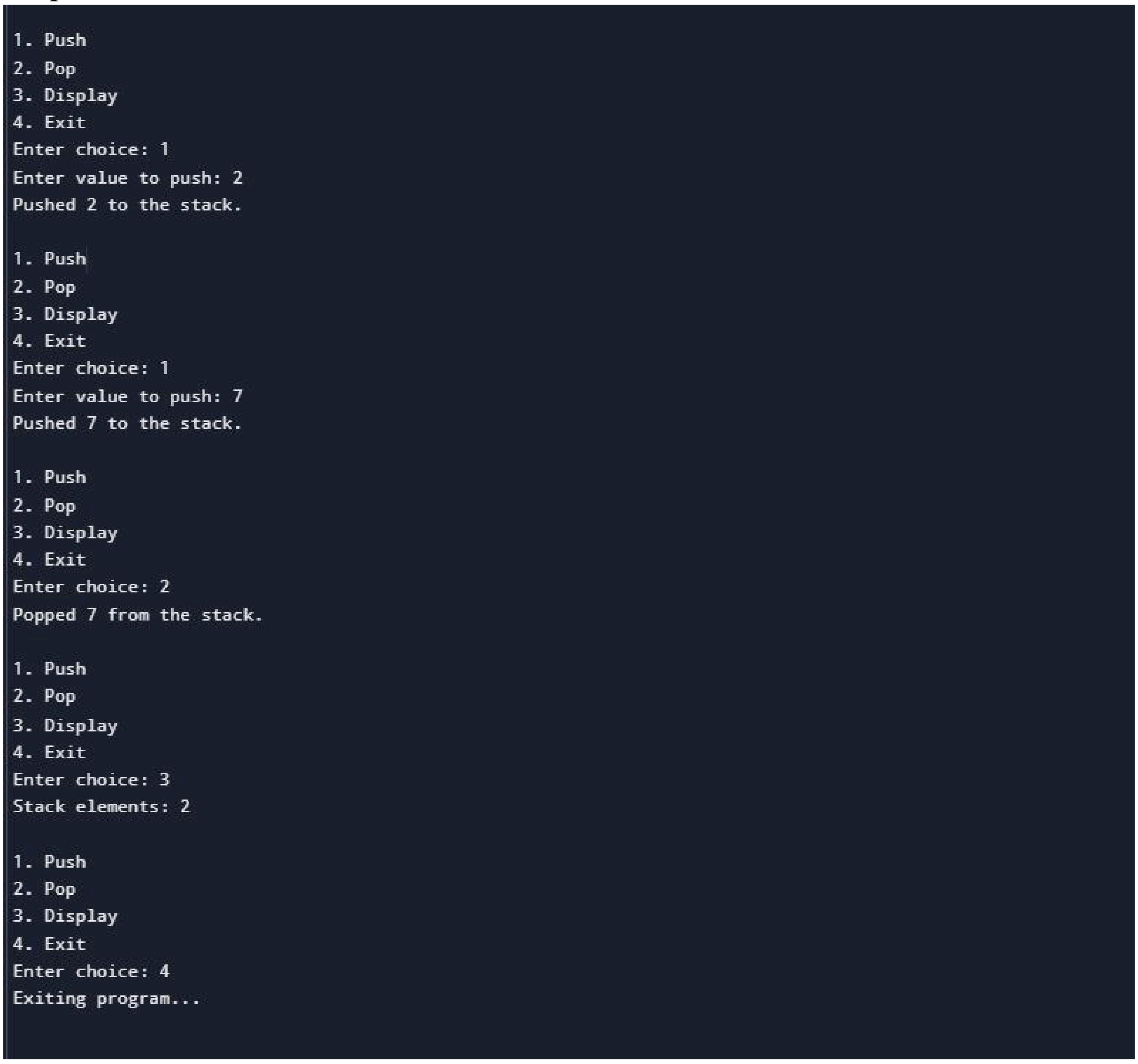
CO1	Apply the concept of linear and nonlinear data structures.
CO2	Analyze data structure operations for a given problem
CO3	Design and develop solutions using the operations of linear and nonlinear data structure for a given specification.
CO4	Conduct practical experiments for demonstrating the operations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

```
#include <stdio.h>
#include<stdlib.h>
#define STACK_SIZE 5
void push(int st[],int *top)
{
        int item;
        if(*top==STACK_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]);
}
void main()
{
        int st[10],top=-1, c,val_del;
        while(1)
        {
                printf("\n1. Push\n2. Pop\n3. Display\n");
                printf("\nEnter your choice :");
                scanf("%d",&c);
                switch(c)
                {
                    case 1: push(st,&top);
                            break;
                    case 2: pop(st,&top);
                            break;
                    case 3: display(st,&top);
                            break;
                    default: printf("\nInvalid choice!!!");
                            exit(0);
            }
        }
}
```

Output:

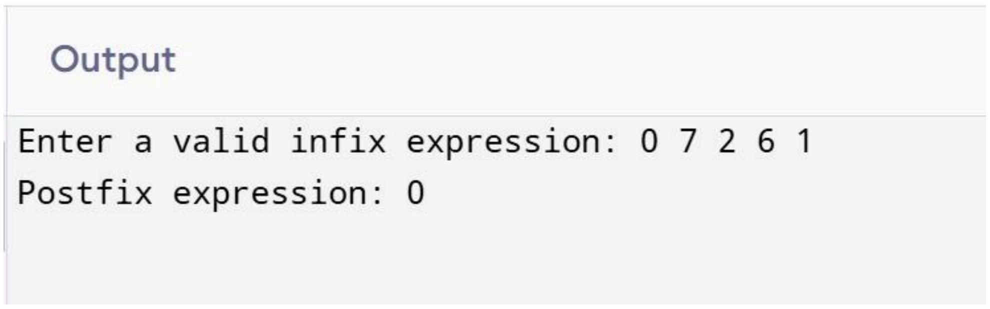


```
1. Push
2. Pop
3. Display
4. Exit
Enter choice: 1
Enter value to push: 2
Pushed 2 to the stack.

1. Push
2. Pop
3. Display
4. Exit
Enter choice: 1
Enter value to push: 7
Pushed 7 to the stack.

1. Push
2. Pop
3. Display
4. Exit
Enter choice: 2
Popped 7 from the stack.

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

1. Push
2. Pop
3. Display
4. Exit
Enter choice: 4
Exiting program...
```

Lab program 2:

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)

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#define MAX 100
typedef struct {
    char arr[MAX];
    int top;
} Stack;
void initializeStack(Stack* stack) {
    stack->top = -1;
}
int isEmpty(Stack* stack) {
    return stack->top == -1;
}
int isFull(Stack* stack) {
    return stack->top == MAX - 1;
}
char peek(Stack* stack) {
    if (!isEmpty(stack)) {
        return stack->arr[stack->top];
    }
    return -1;
}
char pop(Stack* stack) {
    if (!isEmpty(stack)) {
        return stack->arr[stack->top--];
    }
    return -1;
}
void push(Stack* stack, char c) {
    if (isFull(stack)) {
        printf("Stack Overflow!\n");
        return;
    }
    stack->arr[++stack->top] = c;
}
int isOperator(char c) {
    return (c == '+' || c == '-' || c == '*' || c == '/');
}
int precedence(char c) {
    if (c == '+' || c == '-') {
        return 1;
    } else if (c == '*' || c == '/') {
        return 2;
    }
    return 0;
}
void infixToPostfix(char* infix, char* postfix) {
    Stack stack;
    initializeStack(&stack);
    int i = 0, j = 0;

    while (infix[i] != '\0') {
        char current = infix[i];
        if (isalnum(current)) {
            postfix[j++] = current;
        }
        else if (current == '(') {
            push(&stack, current);
        }
        else if (current == ')') {
            while (!isEmpty(&stack) && peek(&stack) != '(') {
                postfix[j++] = pop(&stack);
            }
            pop(&stack);
        }
        else if (isOperator(current)) {
            while (!isEmpty(&stack) && precedence(peek(&stack)) >= precedence(current)) {
                postfix[j++] = pop(&stack);
            }
            push(&stack, current);
        }

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

    postfix[j] = '\0';
}
int main() {
    char infix[MAX], postfix[MAX];
    printf("Enter a valid infix expression: ");
    scanf("%s", infix);
    infixToPostfix(infix, postfix);
    printf("Postfix expression: %s\n", postfix);
    return 0;
}
```



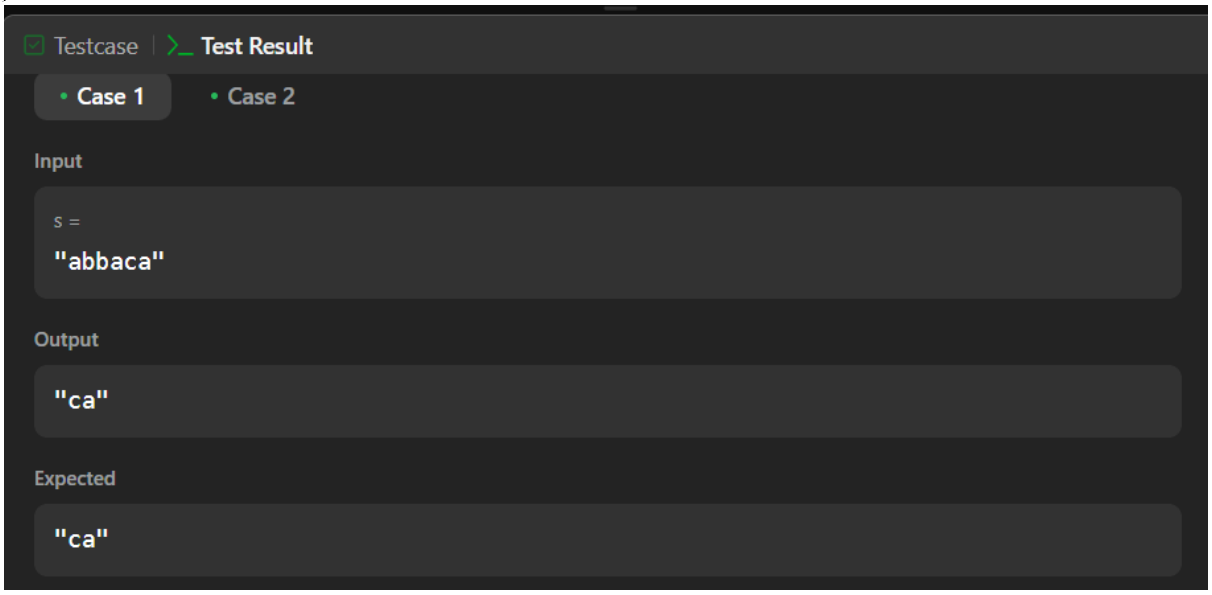
Output

```
Enter a valid infix expression: 0 7 2 6 1
Postfix expression: 0
```

Leet Code 1:

```
#include <stdlib.h>

#include <string.h>


char* removeDuplicates(char* s) {

   int len = strlen(s);

   char* stack = (char*)malloc(len + 1);

   if (!stack) {

      return NULL;

   }

   int top = -1;


   for (int i = 0; i < len; i++) {

      if (top >= 0 && stack[top] == s[i]) {

         top--;

      } else {

         stack[++top] = s[i];

      }

   }


   stack[top + 1] = '\0';

   return stack;
```

}



Testcase | Test Result

• Case 1 • Case 2

Input

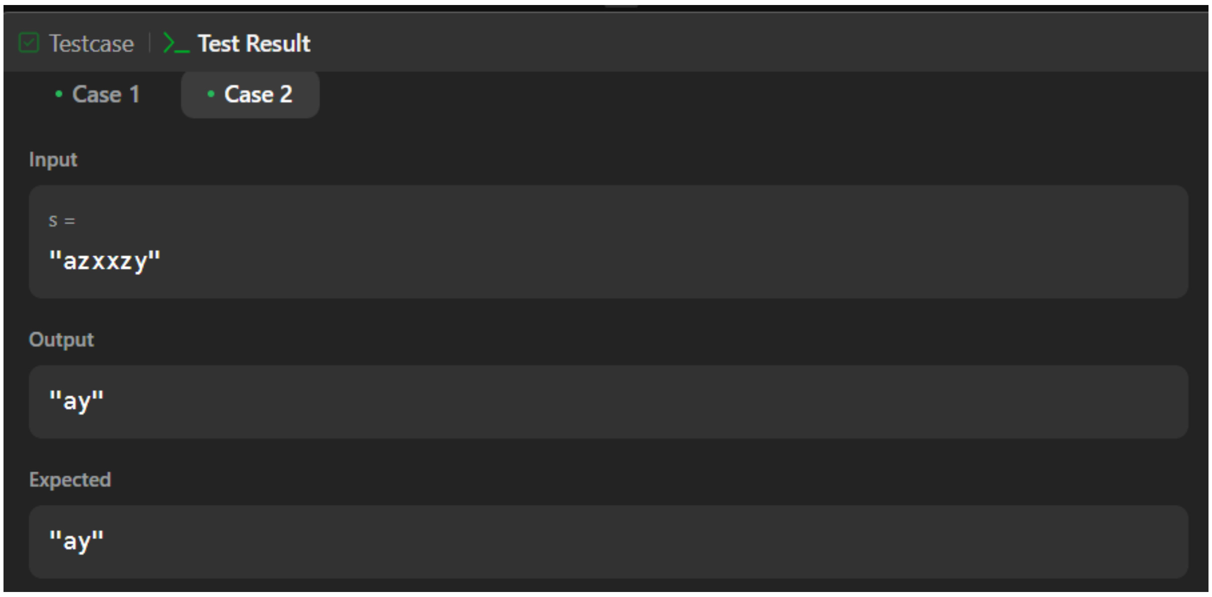
```
s =
"abbaca"
```

Output

```
"ca"
```

Expected

```
"ca"
```



Testcase | Test Result

• Case 1 • Case 2

Input

```
s =
"azxxzy"
```

Output

```
"ay"
```

Expected

```
"ay"
```

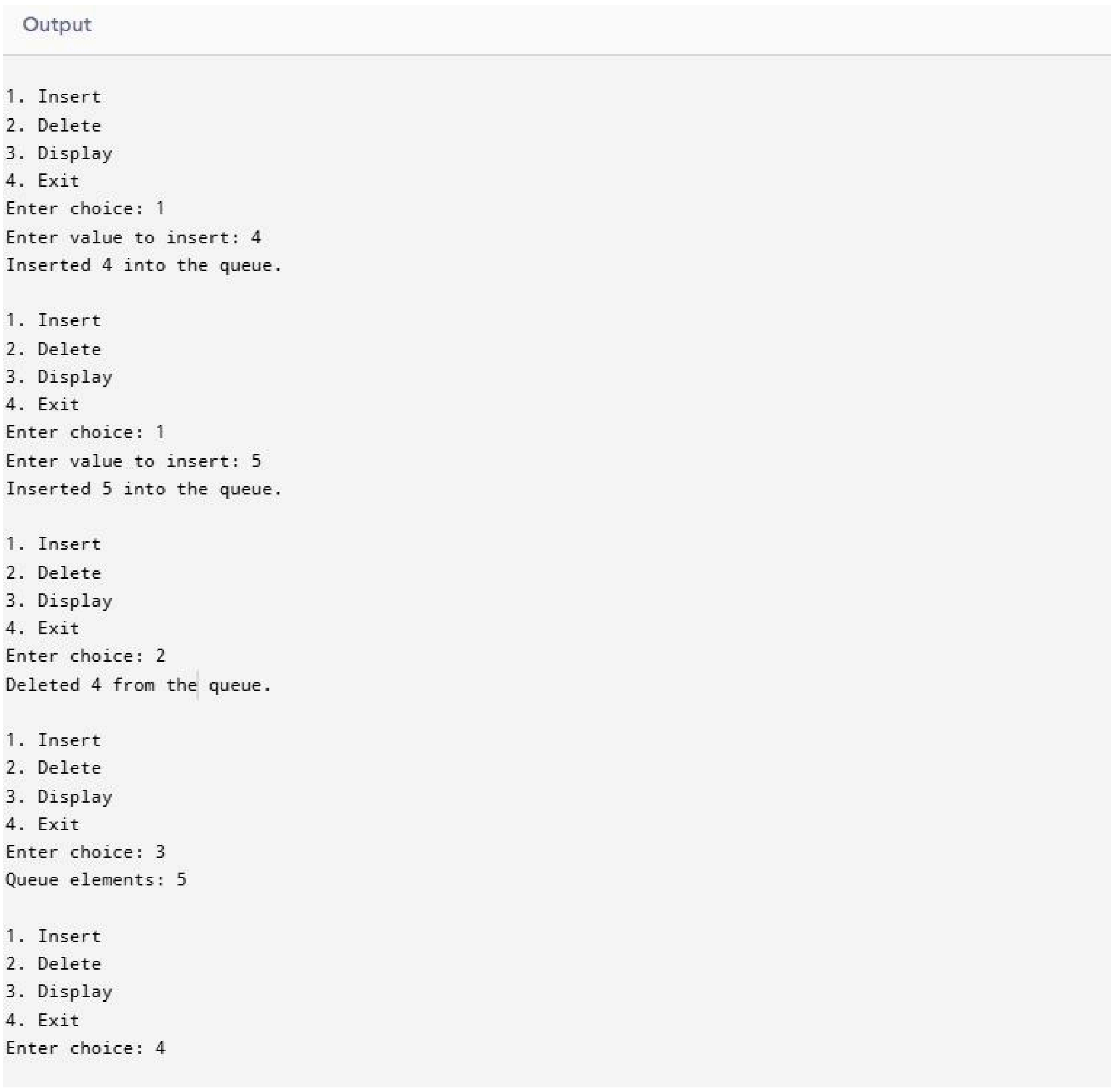
Lab program 3a:

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

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

#define MAX 5

typedef struct {
   int arr[MAX];
   int front, rear;
} Queue;

void initializeQueue(Queue* queue) {
   queue->front = -1;
   queue->rear = -1;
}

int isFull(Queue* queue) {
   return queue->rear == MAX - 1;
}

int isEmpty(Queue* queue) {
   return queue->front == -1 || queue->front > queue->rear;
}

void insert(Queue* queue, int value) {
   if (isFull(queue)) {
       printf("Queue Overflow! Unable to insert %d\n", value);
   } else {
       if (queue->front == -1) queue->front = 0;
       queue->arr[++queue->rear] = value;
       printf("Inserted %d into the queue.\n", value);
   }
}

int delete(Queue* queue) {
   if (isEmpty(queue)) {
       printf("Queue Underflow! No element to delete.\n");
       return -1;
   } else {
       int deletedValue = queue->arr[queue->front++];
       if (queue->front > queue->rear) {
           queue->front = queue->rear = -1;
       }
       return deletedValue;
   }
}

void display(Queue* queue) {
   if (isEmpty(queue)) {
       printf("Queue is empty.\n");
   } else {
       printf("Queue elements: ");
       for (int i = queue->front; i <= queue->rear; i++) {
           printf("%d ", queue->arr[i]);
       }
       printf("\n");
   }
}
int main() {
   Queue queue;
   int choice, value;
   initializeQueue(&queue);
   while (1) {
       printf("\n1. Insert\n2. Delete\n3. Display\n4. Exit\nEnter choice: ");
       scanf("%d", &choice);
       switch (choice) {
           case 1:
               printf("Enter value to insert: ");
               scanf("%d", &value);
               insert(&queue, value);
               break;
           case 2:
               value = delete(&queue);
               if (value != -1) {
                   printf("Deleted %d from the queue.\n", value);
               }
               break;
           case 3:
               display(&queue);
               break;
           case 4:
               exit(0);
           default:
               printf("Invalid choice. Try again.\n");
       }
   }
   return 0;
}
```



Output

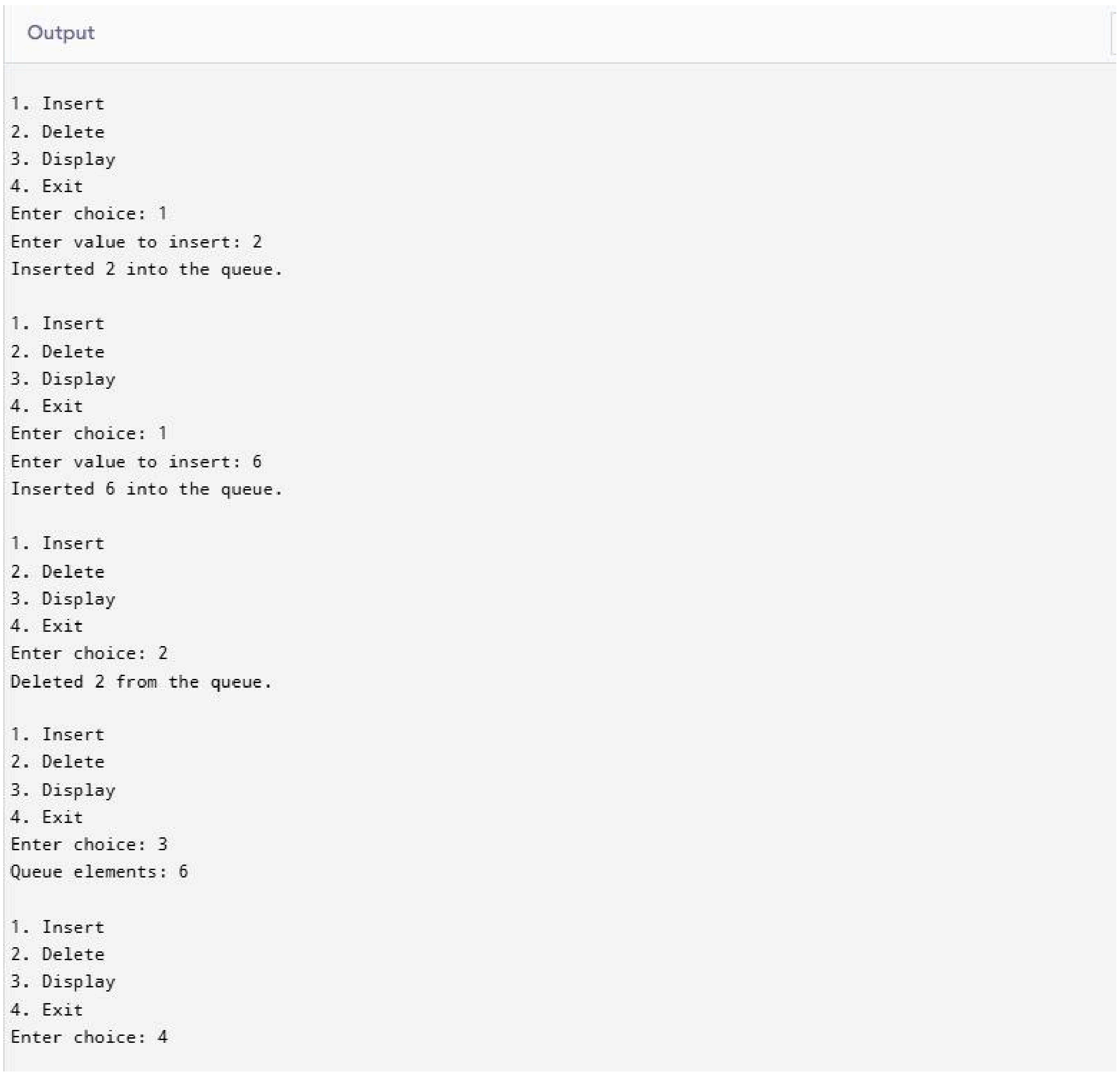
```
1. Insert
2. Delete
3. Display
4. Exit
Enter choice: 1
Enter value to insert: 4
Inserted 4 into the queue.

1. Insert
2. Delete
3. Display
4. Exit
Enter choice: 1
Enter value to insert: 5
Inserted 5 into the queue.

1. Insert
2. Delete
3. Display
4. Exit
Enter choice: 2
Deleted 4 from the queue.

1. Insert
2. Delete
3. Display
4. Exit
Enter choice: 3
Queue elements: 5

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

Lab program 3b:

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

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

#define MAX 5

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

void initializeQueue(CircularQueue* queue) {
   queue->front = queue->rear = -1;
}
int isFull(CircularQueue* queue) {
   return (queue->front == 0 && queue->rear == MAX - 1) || (queue->front == queue->rear +
1);
}
int isEmpty(CircularQueue* queue) {
   return queue->front == -1;
}
void insert(CircularQueue* queue, int value) {
   if (isFull(queue)) {
      printf("Queue Overflow! Unable to insert %d\n", value);
   } else {
      if (queue->front == -1) queue->front = 0;
```

```
        queue->rear = (queue->rear + 1) % MAX;
        queue->arr[queue->rear] = value;
        printf("Inserted %d into the queue.\n", value);
    }
}
int delete(CircularQueue* queue) {
    if (isEmpty(queue)) {
        printf("Queue Underflow! No element to delete.\n");
        return -1;
    } else {
        int deletedValue = queue->arr[queue->front];
        if (queue->front == queue->rear) {
            queue->front = queue->rear = -1;
        } else {
            queue->front = (queue->front + 1) % MAX;
        }
        return deletedValue;
    }
}
void display(CircularQueue* queue) {
    if (isEmpty(queue)) {
        printf("Queue is empty.\n");
    } else {
        printf("Queue elements: ");
        int i = queue->front;
        while (i != queue->rear) {
            printf("%d ", queue->arr[i]);
            i = (i + 1) % MAX;
        }
        printf("%d\n", queue->arr[queue->rear]);
    }
}
int main() {
    CircularQueue queue;
    int choice, value;
    initializeQueue(&queue);
    while (1) {
        printf("\n1. Insert\n2. Delete\n3. Display\n4. Exit\nEnter choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Enter value to insert: ");
                scanf("%d", &value);
                insert(&queue, value);
                break;
            case 2:
                value = delete(&queue);
                if (value != -1) {
                    printf("Deleted %d from the queue.\n", value);
                }
                break;
            case 3:
                display(&queue);
                break;
            case 4:
                exit(0);
            default:
                printf("Invalid choice. Try again.\n");
        }
    }
    return 0;
}
```



Output

```
1. Insert
2. Delete
3. Display
4. Exit
Enter choice: 1
Enter value to insert: 2
Inserted 2 into the queue.

1. Insert
2. Delete
3. Display
4. Exit
Enter choice: 1
Enter value to insert: 6
Inserted 6 into the queue.

1. Insert
2. Delete
3. Display
4. Exit
Enter choice: 2
Deleted 2 from the queue.

1. Insert
2. Delete
3. Display
4. Exit
Enter choice: 3
Queue elements: 6

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

Lab program 4:

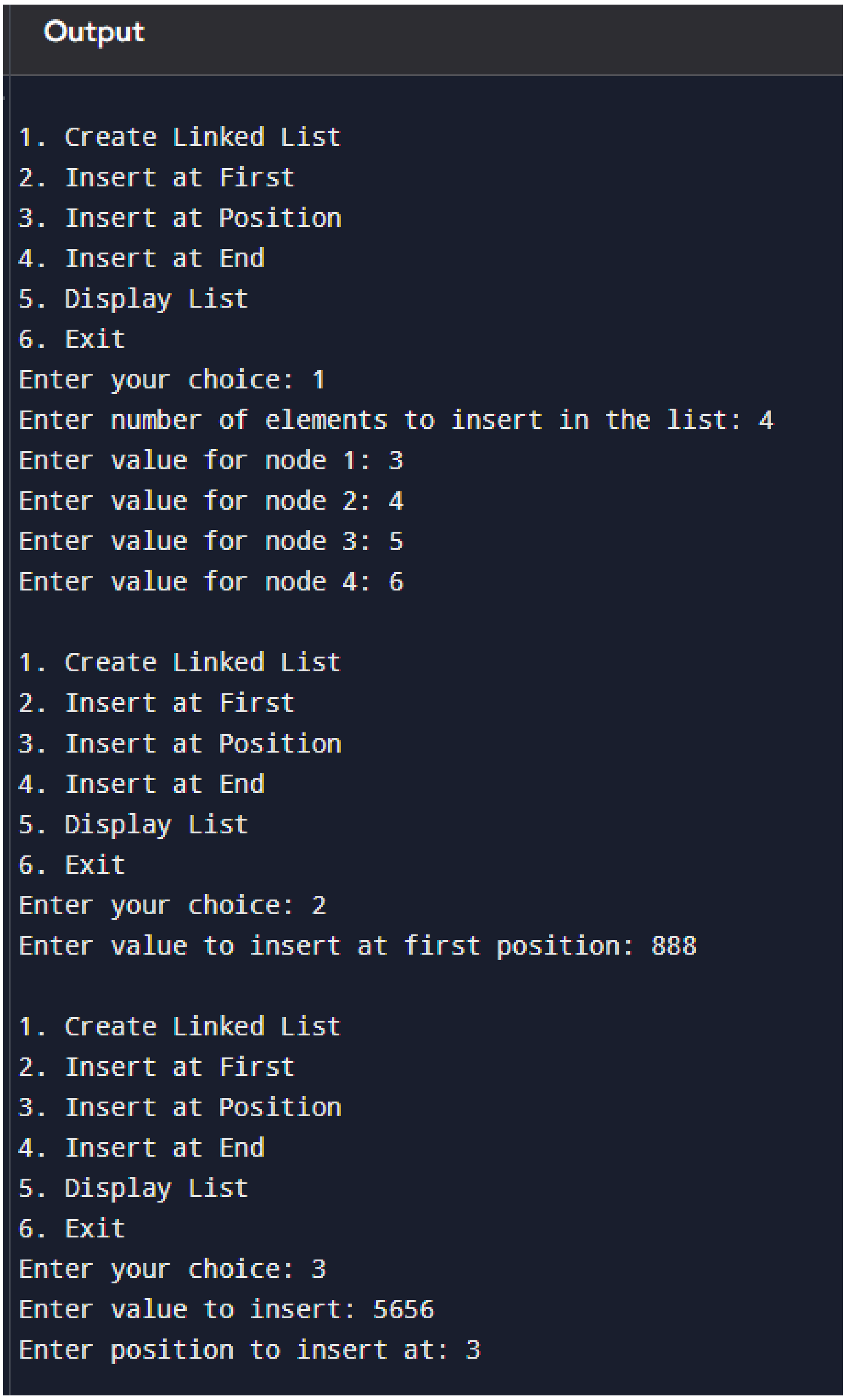
WAP to Implement Singly Linked List with following operations

a) Create a linked list.

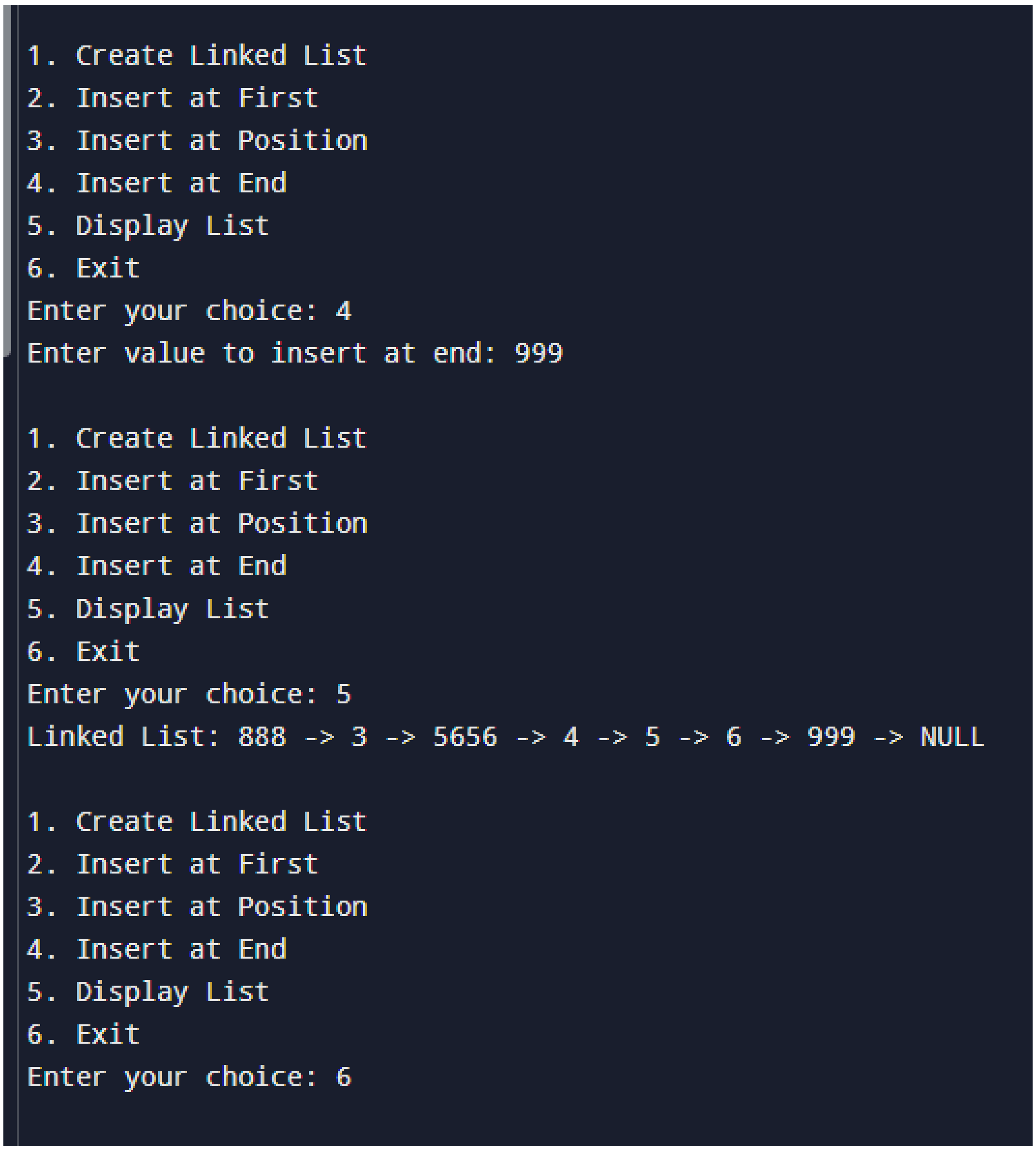
b) Insertion of a node at first position, at any position and at end of list.

Display the contents of the linked list.

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

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

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

struct Node* createLinkedList() {
    struct Node* head = NULL;
    int n, value;
    printf("Enter number of elements to insert in the list: ");
    scanf("%d", &n);

    for (int i = 0; i < n; i++) {
        printf("Enter value for node %d: ", i + 1);
        scanf("%d", &value);
        struct Node* newNode = createNode(value);
        if (head == NULL) {
            head = newNode;  // If the list is empty, the new node becomes the head
        } else {
            struct Node* temp = head;
            while (temp->next != NULL) {
                temp = temp->next;
            }
            temp->next = newNode;  // Append the new node to the end
        }
    }
    return head;
}

struct Node* insertAtFirst(struct Node* head, int value) {
    struct Node* newNode = createNode(value);
    newNode->next = head;
    return newNode;
}
struct Node* insertAtPosition(struct Node* head, int value, int position) {
    struct Node* newNode = createNode(value);
    if (position == 1) {
        newNode->next = head;
        return newNode;
    }

    struct Node* temp = head;
    for (int i = 1; i < position - 1 && temp != NULL; i++) {
        temp = temp->next;
    }

    if (temp == NULL) {
        printf("Invalid position!\n");
        free(newNode);
        return head;
    }

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

struct Node* insertAtEnd(struct Node* head, int value) {
    struct Node* newNode = createNode(value);
    if (head == NULL) {
        return newNode;
    }

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

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

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

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

    while (1) {
        printf("\n1. Create Linked List\n2. Insert at First\n3. Insert at Position\n4. Insert at
End\n5. Display List\n6. Exit\nEnter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                head = createLinkedList();
                break;
            case 2:
                printf("Enter value to insert at first position: ");
                scanf("%d", &value);
                head = insertAtFirst(head, value);
                break;
            case 3:
                printf("Enter value to insert: ");
                scanf("%d", &value);
                printf("Enter position to insert at: ");
                scanf("%d", &position);
                head = insertAtPosition(head, value, position);
                break;
            case 4:
                printf("Enter value to insert at end: ");
                scanf("%d", &value);
                head = insertAtEnd(head, value);
                break;
            case 5:
                displayList(head);
                break;
            case 6:
                exit(0);
            default:
                printf("Invalid choice! Try again.\n");
        }
    }

    return 0;
}
```


Output

```

1. Create Linked List
2. Insert at First
3. Insert at Position
4. Insert at End
5. Display List
6. Exit
Enter your choice: 1
Enter number of elements to insert in the list: 4
Enter value for node 1: 3
Enter value for node 2: 4
Enter value for node 3: 5
Enter value for node 4: 6

1. Create Linked List
2. Insert at First
3. Insert at Position
4. Insert at End
5. Display List
6. Exit
Enter your choice: 2
Enter value to insert at first position: 888

1. Create Linked List
2. Insert at First
3. Insert at Position
4. Insert at End
5. Display List
6. Exit
Enter your choice: 3
Enter value to insert: 5656
Enter position to insert at: 3
```

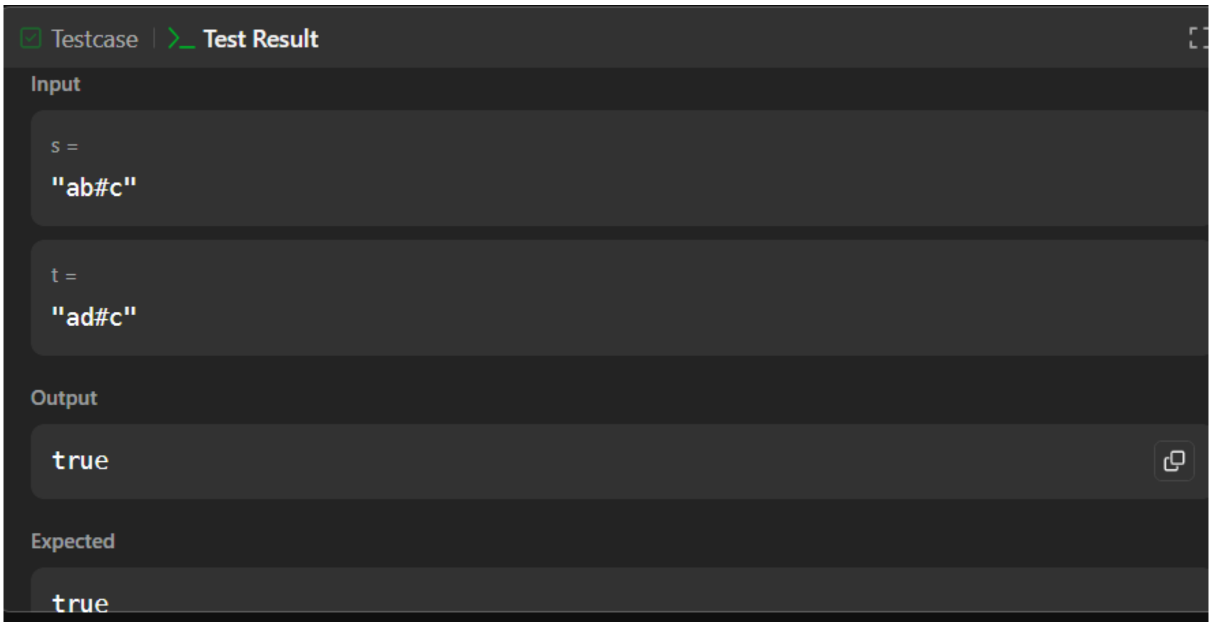


```
1. Create Linked List
2. Insert at First
3. Insert at Position
4. Insert at End
5. Display List
6. Exit
Enter your choice: 4
Enter value to insert at end: 999

1. Create Linked List
2. Insert at First
3. Insert at Position
4. Insert at End
5. Display List
6. Exit
Enter your choice: 5
Linked List: 888 -> 3 -> 5656 -> 4 -> 5 -> 6 -> 999 -> NULL

1. Create Linked List
2. Insert at First
3. Insert at Position
4. Insert at End
5. Display List
6. Exit
Enter your choice: 6
```

Leet Code-2:

```
bool backspaceCompare(char* s, char* t) {
    int i = strlen(s) - 1;
    int j = strlen(t) - 1;
    while (i >= 0 || j >= 0) {
        int backspaceCount = 0;
        while (i >= 0 && (s[i] == '#' || backspaceCount > 0)) {
            if (s[i] == '#') {
                backspaceCount++;
            } else {
                backspaceCount--;
            }
            i--;
       }
        int backspaceCountT = 0;
        while (j >= 0 && (t[j] == '#' || backspaceCountT > 0)) {
            if (t[j] == '#') {
                backspaceCountT++;
            } else {
                backspaceCountT--;
            }
            j--;
        }
        if (i >= 0 && j >= 0 && s[i] != t[j]) {
            return false;
        }
        if ((i >= 0) != (j >= 0)) {
            return false;
        }
```

```
            i--;
            j--;
        }
        return true;
    }
```



Testcase | Test Result

Input

```
s =
"ab#c"
```

```
t =
"ad#c"
```

Output

```
true
```

Expected

```
true
```

Lab program 5:

WAP to Implement Singly Linked List with following operations

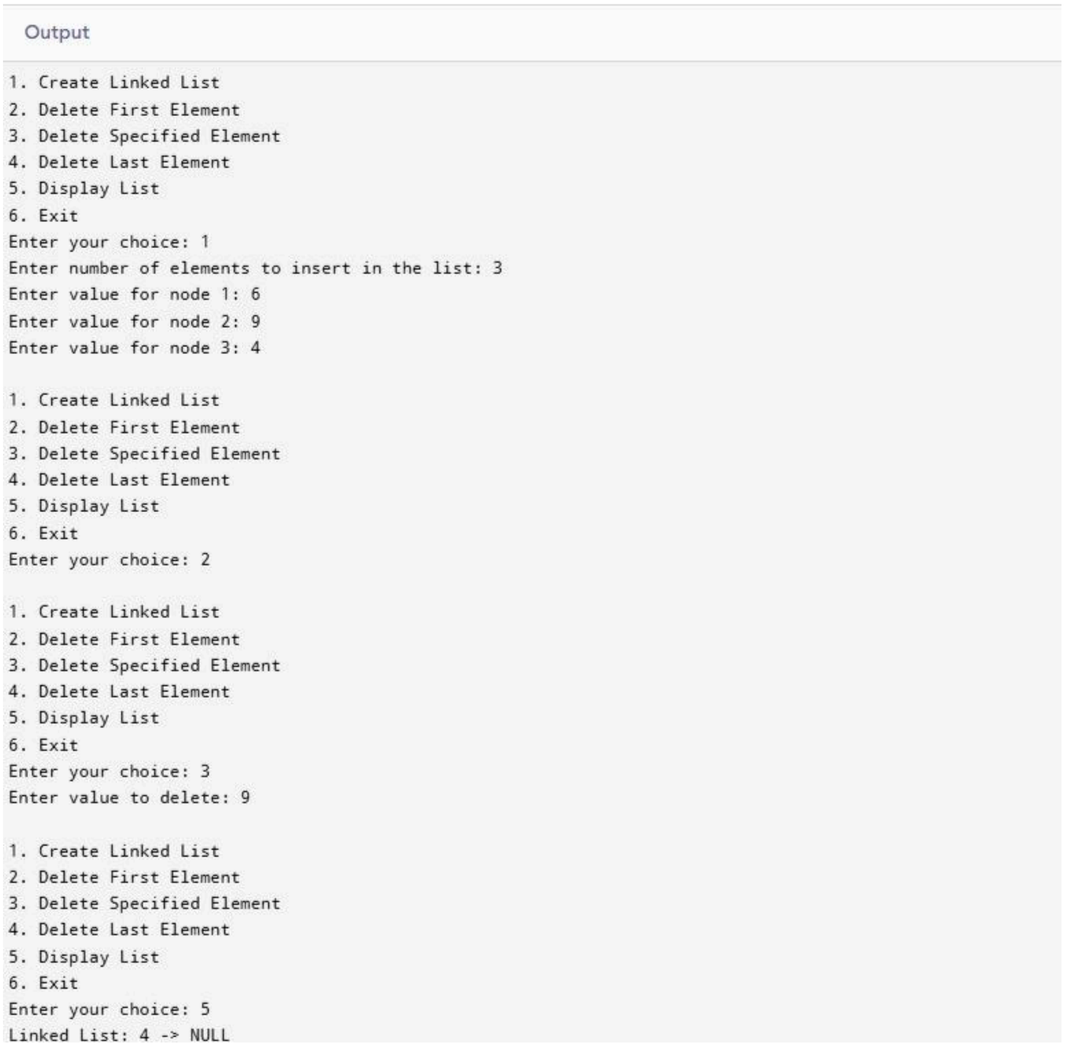
a) Create a linked list.

b) Deletion of first element, specified element and last

element in the list.

c) Display the contents of the linked list.

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

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

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

struct Node* createLinkedList() {
    struct Node* head = NULL;
    int n, value;
    printf("Enter number of elements to insert in the list: ");
    scanf("%d", &n);

    for (int i = 0; i < n; i++) {
        printf("Enter value for node %d: ", i + 1);
        scanf("%d", &value);
        struct Node* newNode = createNode(value);
        if (head == NULL) {
            head = newNode;
        } else {
            struct Node* temp = head;
            while (temp->next != NULL) {
                temp = temp->next;
            }
            temp->next = newNode;
        }
    }
    return head;
}

struct Node* deleteFirst(struct Node* head) {
    if (head == NULL) {
        printf("List is empty. Cannot delete the first element.\n");
        return head;
    }
    struct Node* temp = head;
    head = head->next;
    free(temp);
    return head;
}

struct Node* deleteElement(struct Node* head, int value) {
    if (head == NULL) {
        printf("List is empty. Cannot delete element.\n");
        return head;
    }

    struct Node* temp = head;
    if (head->data == value) {
        head = head->next;
        free(temp);
        return head;
    }

    struct Node* prev = NULL;
    while (temp != NULL && temp->data != value) {
        prev = temp;
        temp = temp->next;
    }

    if (temp == NULL) {
        printf("Element %d not found in the list.\n", value);
        return head;
    }

    prev->next = temp->next;
    free(temp);
    return head;
}

struct Node* deleteLast(struct Node* head) {
    if (head == NULL) {
        printf("List is empty. Cannot delete the last element.\n");
        return head;
    }

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

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

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

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

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

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

    while (1) {
        printf("\n1. Create Linked List\n2. Delete First Element\n3. Delete Specified
Element\n4. Delete Last Element\n5. Display List\n6. Exit\nEnter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                head = createLinkedList();
                break;
            case 2:
                head = deleteFirst(head);
                break;
            case 3:
                printf("Enter value to delete: ");
                scanf("%d", &value);
                head = deleteElement(head, value);
                break;
            case 4:
                head = deleteLast(head);
                break;
            case 5:
                displayList(head);
                break;
            case 6:
                exit(0);
            default:
                printf("Invalid choice! Try again.\n");
        }
    }
    return 0;
}
```

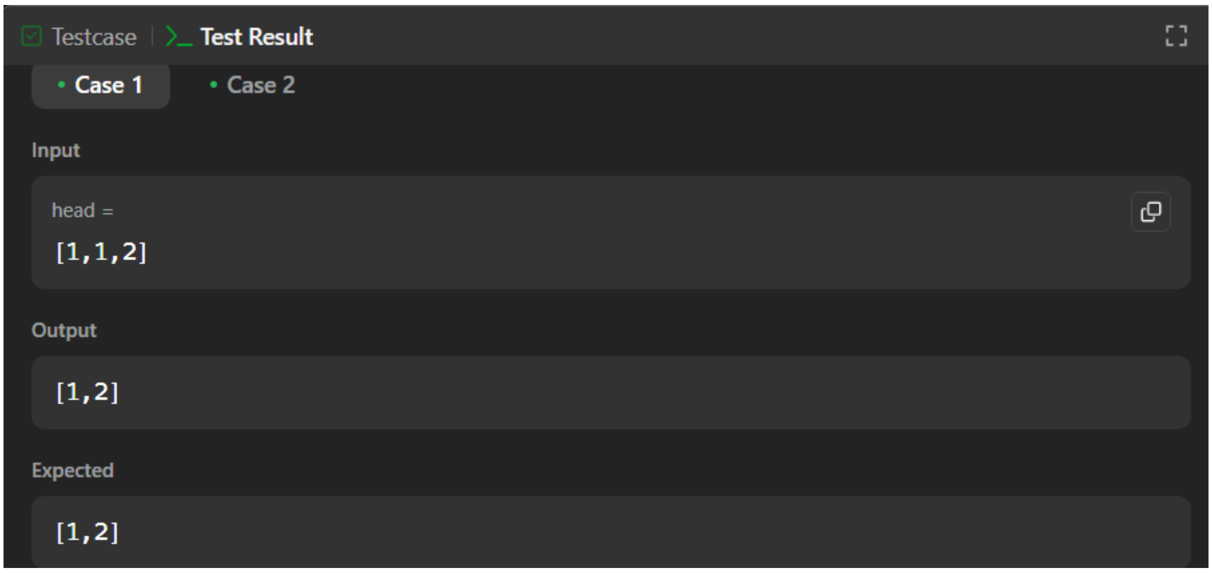


Output

```
1. Create Linked List
2. Delete First Element
3. Delete Specified Element
4. Delete Last Element
5. Display List
6. Exit
Enter your choice: 1
Enter number of elements to insert in the list: 3
Enter value for node 1: 6
Enter value for node 2: 9
Enter value for node 3: 4

1. Create Linked List
2. Delete First Element
3. Delete Specified Element
4. Delete Last Element
5. Display List
6. Exit
Enter your choice: 2

1. Create Linked List
2. Delete First Element
3. Delete Specified Element
4. Delete Last Element
5. Display List
6. Exit
Enter your choice: 3
Enter value to delete: 9

1. Create Linked List
2. Delete First Element
3. Delete Specified Element
4. Delete Last Element
5. Display List
6. Exit
Enter your choice: 5
Linked List: 4 -> NULL
```

Leet Code-3:

```
struct ListNode* deleteDuplicates(struct ListNode* head) {
    struct ListNode* current = head;
    while (current && current->next) {
        if (current->val == current->next->val) {
            struct ListNode* temp = current->next;
            current->next = current->next->next;
            free(temp);
        } else {
            current = current->next;
        }
    }
    return head;
}
```



Input

head =

```
[1,1,2]
```

Output

```
[1,2]
```

Expected

```
[1,2]
```

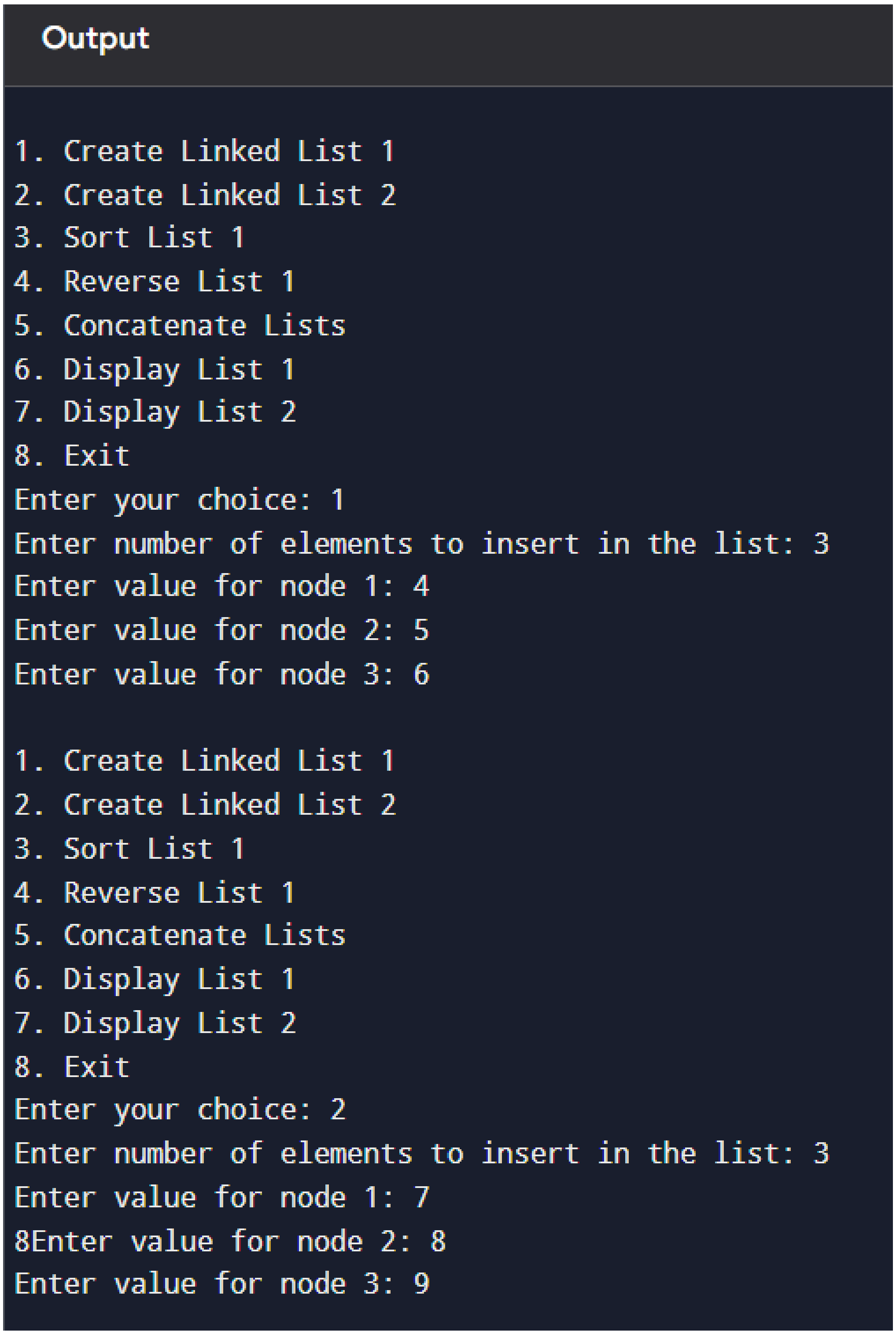
Lab program 6a:

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

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

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

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

struct Node* createLinkedList() {
    struct Node* head = NULL;
    int n, value;
    printf("Enter number of elements to insert in the list: ");
    scanf("%d", &n);

    for (int i = 0; i < n; i++) {
        printf("Enter value for node %d: ", i + 1);
        scanf("%d", &value);
        struct Node* newNode = createNode(value);
        if (head == NULL) {
            head = newNode;
        } else {
            struct Node* temp = head;
            while (temp->next != NULL) {
                temp = temp->next;
            }
            temp->next = newNode;
        }
    }
    return head;
}

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

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

void sortList(struct Node* head) {
    if (head == NULL) {
        printf("List is empty. Cannot sort.\n");
        return;
    }

    struct Node* i = head;
    struct Node* j = NULL;
    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;
    }
}

struct Node* 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;

    return head;

}


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

    if (head1 == NULL) {

        return head2;

    }


    struct Node* temp = head1;

    while (temp->next != NULL) {

        temp = temp->next;

    }


    temp->next = head2;

    return head1;

}

int main() {

    struct Node* head1 = NULL;

    struct Node* head2 = NULL;

    int choice, value

    while (1) {

        printf("\n1. Create Linked List 1\n2. Create Linked List 2\n3. Sort List 1\n4. Reverse
List 1\n5. Concatenate Lists\n6. Display List 1\n7. Display List 2\n8. Exit\nEnter your
choice: ");
```

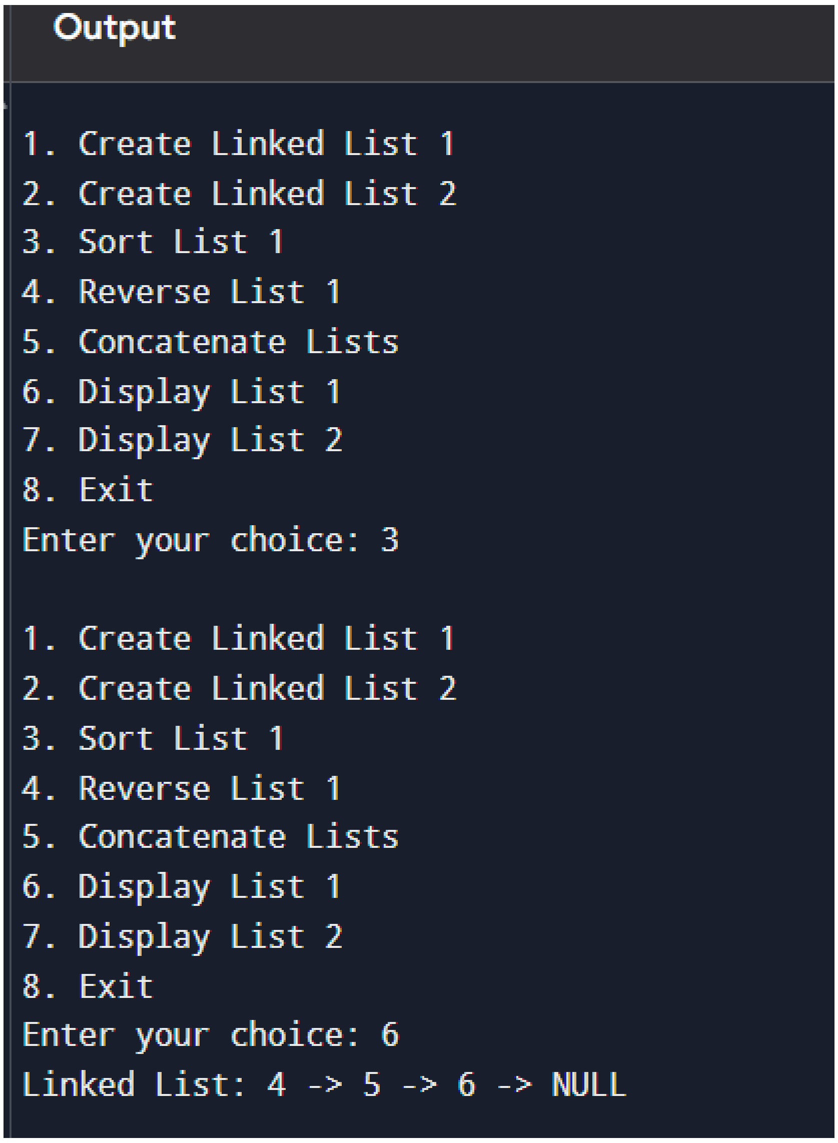
```
    scanf("%d", &choice);
    switch (choice) {
        case 1:
            head1 = createLinkedList();
            break;
        case 2:
            head2 = createLinkedList();
            break;
        case 3:
            sortList(head1);
            break;
        case 4:
            head1 = reverseList(head1);
            break;
        case 5:
            head1 = concatenateLists(head1, head2);
            break;
        case 6:
            displayList(head1);
            break;
        case 7:
            displayList(head2);
            break;
        case 8:
            exit(0);
        default:
            printf("Invalid choice! Try again.\n");
    }
}
```

```
    return 0;
}
```

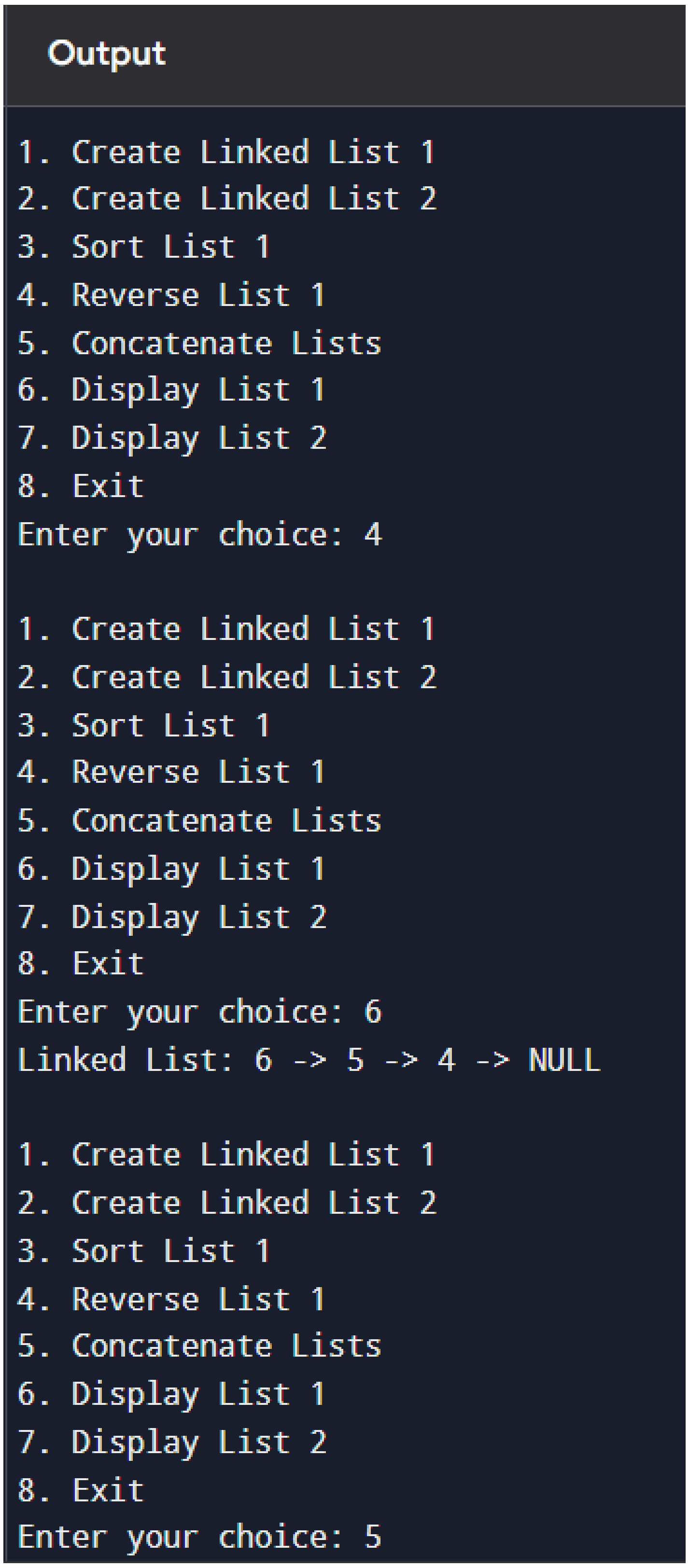


Output

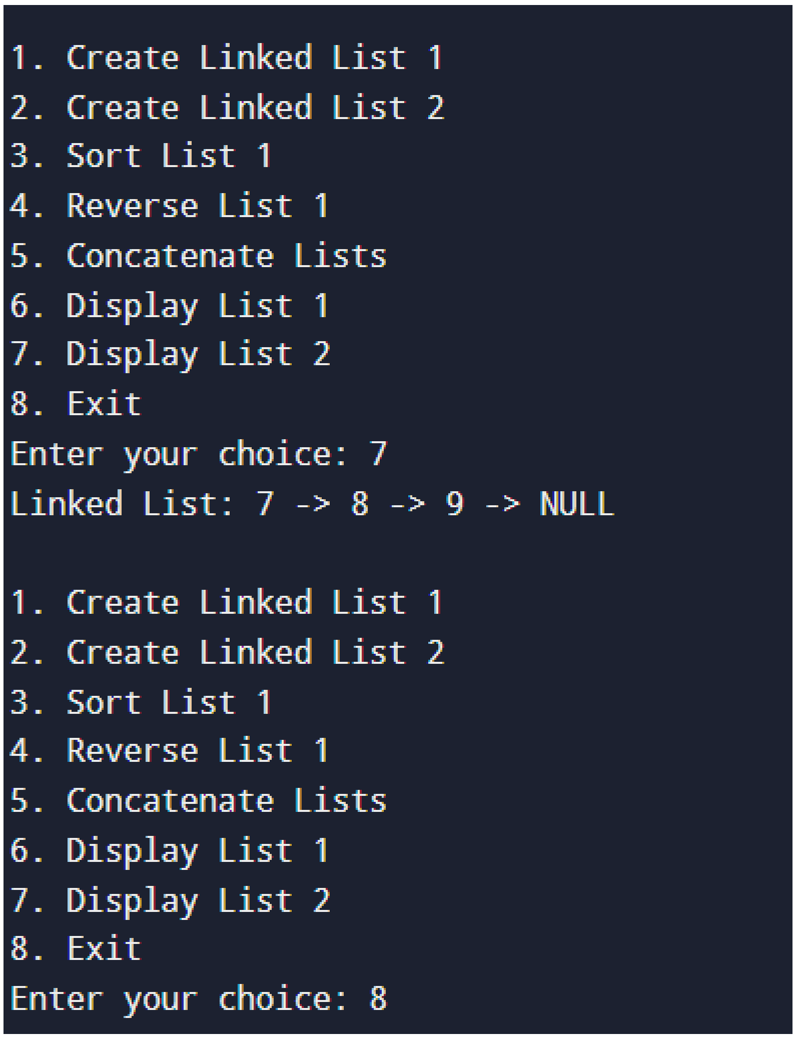
```
1. Create Linked List 1
2. Create Linked List 2
3. Sort List 1
4. Reverse List 1
5. Concatenate Lists
6. Display List 1
7. Display List 2
8. Exit
Enter your choice: 1
Enter number of elements to insert in the list: 3
Enter value for node 1: 4
Enter value for node 2: 5
Enter value for node 3: 6

1. Create Linked List 1
2. Create Linked List 2
3. Sort List 1
4. Reverse List 1
5. Concatenate Lists
6. Display List 1
7. Display List 2
8. Exit
Enter your choice: 2
Enter number of elements to insert in the list: 3
Enter value for node 1: 7
8Enter value for node 2: 8
Enter value for node 3: 9
```



```
1. Create Linked List 1
2. Create Linked List 2
3. Sort List 1
4. Reverse List 1
5. Concatenate Lists
6. Display List 1
7. Display List 2
8. Exit
Enter your choice: 3

1. Create Linked List 1
2. Create Linked List 2
3. Sort List 1
4. Reverse List 1
5. Concatenate Lists
6. Display List 1
7. Display List 2
8. Exit
Enter your choice: 6
Linked List: 4 -> 5 -> 6 -> NULL
```


Output

```
1. Create Linked List 1
2. Create Linked List 2
3. Sort List 1
4. Reverse List 1
5. Concatenate Lists
6. Display List 1
7. Display List 2
8. Exit
Enter your choice: 4

1. Create Linked List 1
2. Create Linked List 2
3. Sort List 1
4. Reverse List 1
5. Concatenate Lists
6. Display List 1
7. Display List 2
8. Exit
Enter your choice: 6
Linked List: 6 -> 5 -> 4 -> NULL

1. Create Linked List 1
2. Create Linked List 2
3. Sort List 1
4. Reverse List 1
5. Concatenate Lists
6. Display List 1
7. Display List 2
8. Exit
Enter your choice: 5
```

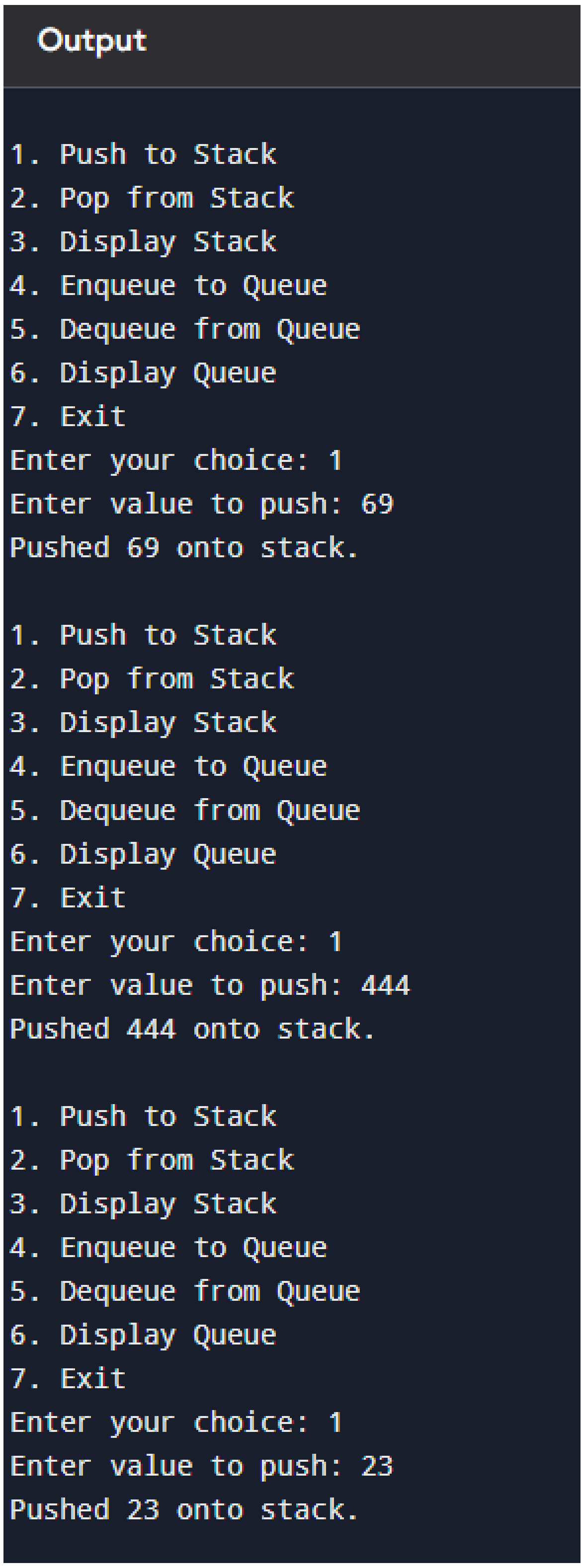


```
1. Create Linked List 1
2. Create Linked List 2
3. Sort List 1
4. Reverse List 1
5. Concatenate Lists
6. Display List 1
7. Display List 2
8. Exit
Enter your choice: 7
Linked List: 7 -> 8 -> 9 -> NULL

1. Create Linked List 1
2. Create Linked List 2
3. Sort List 1
4. Reverse List 1
5. Concatenate Lists
6. Display List 1
7. Display List 2
8. Exit
Enter your choice: 8
```

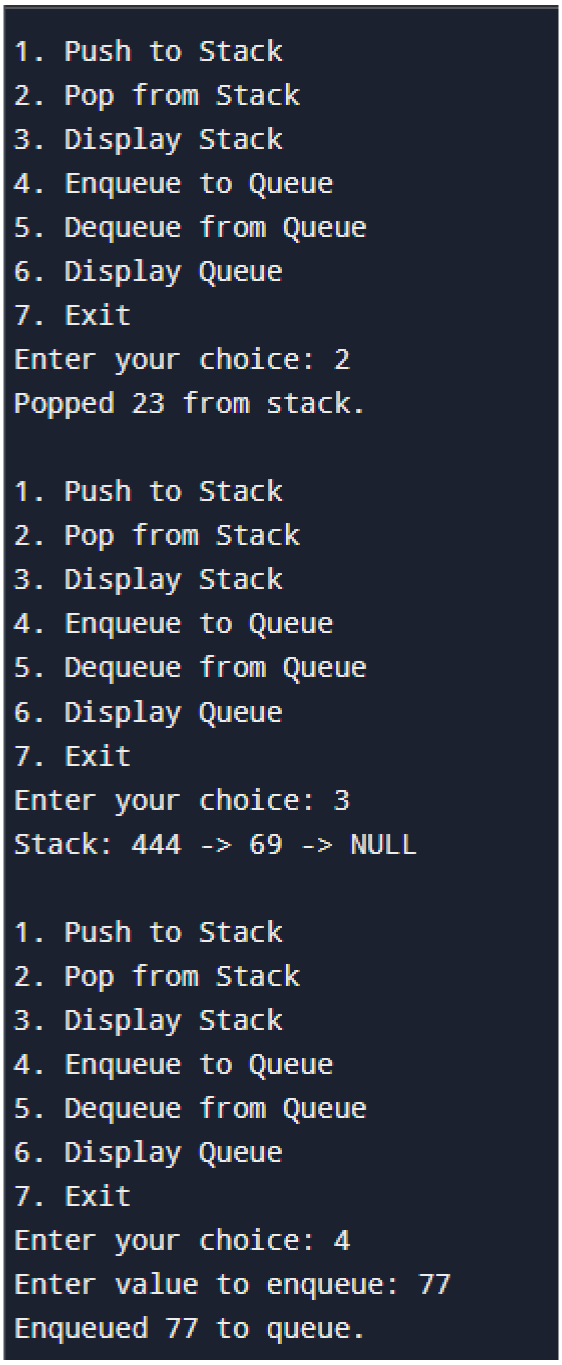
Lab program 6b:

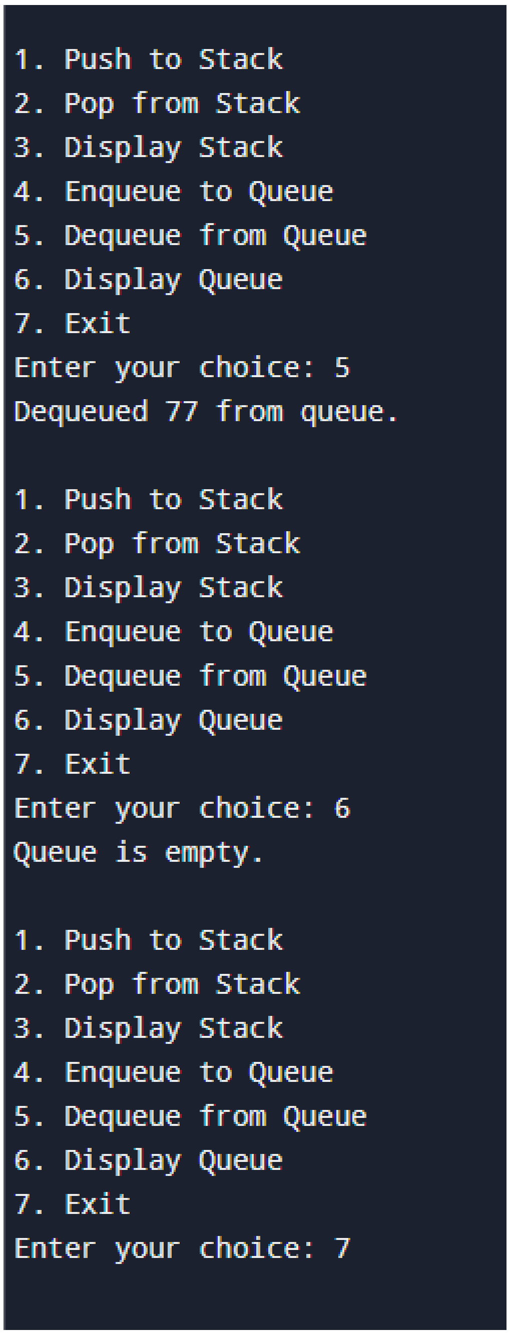
WAP to Implement Single Link List to simulate Stack &amp; Queue Operations.

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

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

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

void push(struct Node** top, int value) {
    struct Node* newNode = createNode(value);
    newNode->next = *top;
    *top = newNode;
    printf("Pushed %d onto stack.\n", value);
}

void pop(struct Node** top) {
    if (*top == NULL) {
        printf("Stack is empty. Underflow condition.\n");
        return;
    }
    struct Node* temp = *top;
    *top = (*top)->next;
    printf("Popped %d from stack.\n", temp->data);
    free(temp);
}

void displayStack(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");
}

void enqueue(struct Node** front, struct Node** rear, int value) {
    struct Node* newNode = createNode(value);
    if (*rear == NULL) {
        *front = *rear = newNode;
        printf("Enqueued %d to queue.\n", value);
        return;
    }
    (*rear)->next = newNode;
    *rear = newNode;
    printf("Enqueued %d to queue.\n", value);
}

void dequeue(struct Node** front, struct Node** rear) {
    if (*front == NULL) {
        printf("Queue is empty. Underflow condition.\n");
        return;
    }
    struct Node* temp = *front;
    *front = (*front)->next;
    if (*front == NULL) {
        *rear = NULL;
    }
    printf("Dequeued %d from queue.\n", temp->data);
    free(temp);
}

void displayQueue(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;
    struct Node* queueFront = NULL;
    struct Node* queueRear = NULL;
    int choice, value;

    while (1) {
        printf("\n1. Push to Stack\n2. Pop from Stack\n3. Display Stack\n4. Enqueue to
Queue\n5. Dequeue from Queue\n6. Display Queue\n7. Exit\nEnter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter value to push: ");
                scanf("%d", &value);
                push(&stackTop, value);
                break;
            case 2:
                pop(&stackTop);
                break;
            case 3:
                displayStack(stackTop);
                break;
            case 4:
                printf("Enter value to enqueue: ");
                scanf("%d", &value);
                enqueue(&queueFront, &queueRear, value);
                break;
            case 5:
                dequeue(&queueFront, &queueRear);
                break;
            case 6:
                displayQueue(queueFront);
                break;
            case 7:
                exit(0);
            default:
                printf("Invalid choice! Try again.\n");
        }
    }

    return 0;
}
```


Output

```
1. Push to Stack
2. Pop from Stack
3. Display Stack
4. Enqueue to Queue
5. Dequeue from Queue
6. Display Queue
7. Exit
Enter your choice: 1
Enter value to push: 69
Pushed 69 onto stack.

1. Push to Stack
2. Pop from Stack
3. Display Stack
4. Enqueue to Queue
5. Dequeue from Queue
6. Display Queue
7. Exit
Enter your choice: 1
Enter value to push: 444
Pushed 444 onto stack.

1. Push to Stack
2. Pop from Stack
3. Display Stack
4. Enqueue to Queue
5. Dequeue from Queue
6. Display Queue
7. Exit
Enter your choice: 1
Enter value to push: 23
Pushed 23 onto stack.
```


```
1. Push to Stack
2. Pop from Stack
3. Display Stack
4. Enqueue to Queue
5. Dequeue from Queue
6. Display Queue
7. Exit
Enter your choice: 2
Popped 23 from stack.

1. Push to Stack
2. Pop from Stack
3. Display Stack
4. Enqueue to Queue
5. Dequeue from Queue
6. Display Queue
7. Exit
Enter your choice: 3
Stack: 444 -> 69 -> NULL

1. Push to Stack
2. Pop from Stack
3. Display Stack
4. Enqueue to Queue
5. Dequeue from Queue
6. Display Queue
7. Exit
Enter your choice: 4
Enter value to enqueue: 77
Enqueued 77 to queue.
```



```
1. Push to Stack
2. Pop from Stack
3. Display Stack
4. Enqueue to Queue
5. Dequeue from Queue
6. Display Queue
7. Exit
Enter your choice: 5
Dequeued 77 from queue.

1. Push to Stack
2. Pop from Stack
3. Display Stack
4. Enqueue to Queue
5. Dequeue from Queue
6. Display Queue
7. Exit
Enter your choice: 6
Queue is empty.

1. Push to Stack
2. Pop from Stack
3. Display Stack
4. Enqueue to Queue
5. Dequeue from Queue
6. Display Queue
7. Exit
Enter your choice: 7
```

Lab program 7:

WAP to Implement doubly link list with primitive operations

a) Create a doubly linked list.

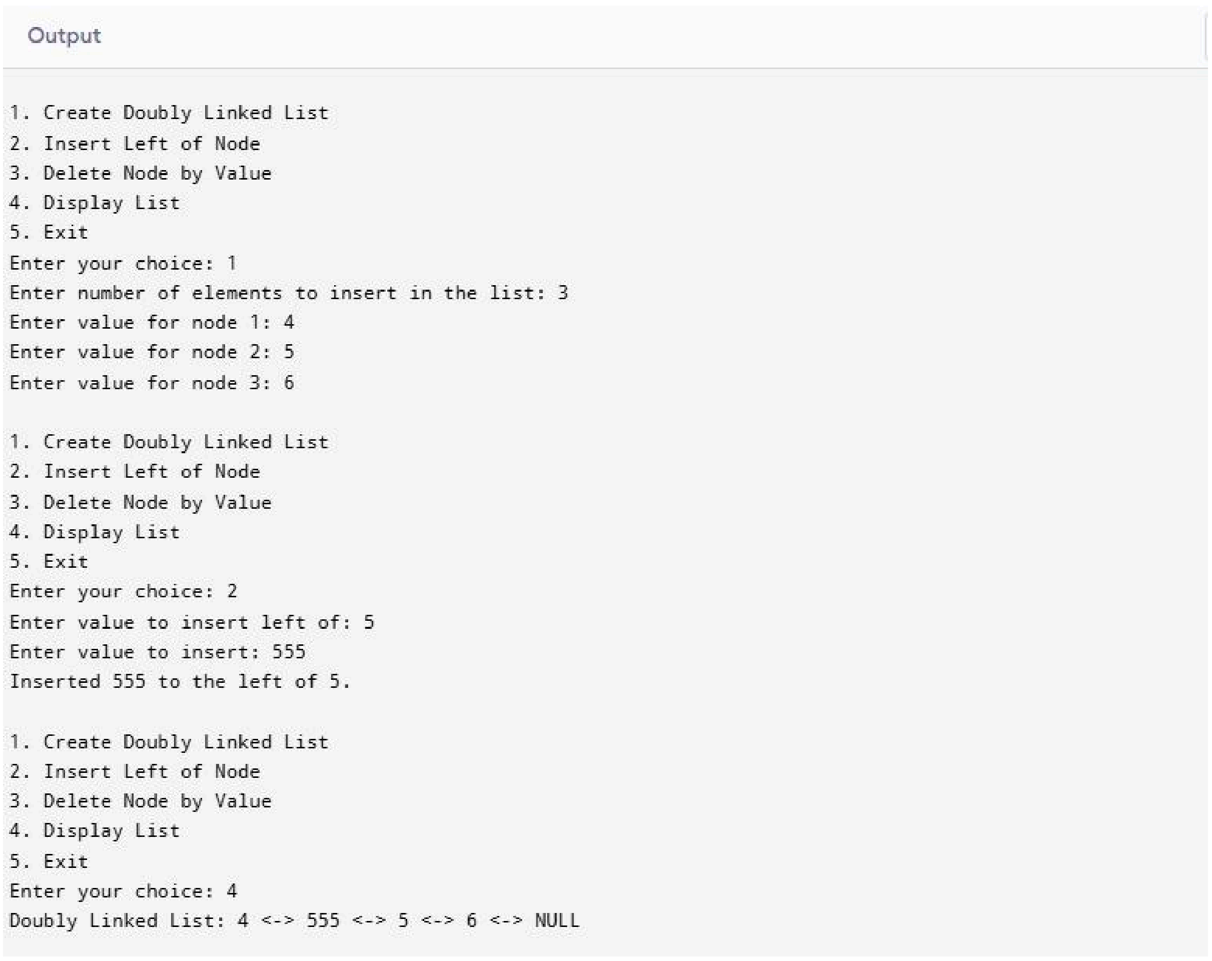
b) Insert a new node to the left of the node.

c) Delete the node based on a specific value

d) Display the contents of the list

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
  int data;
  struct Node* prev;
  struct Node* next;
};
struct Node* createNode(int value) {
  struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
  newNode->data = value;
  newNode->prev = NULL;
  newNode->next = NULL;
  return newNode;
}
struct Node* createDoublyLinkedList() {
  struct Node* head = NULL;
  int n, value;
  printf("Enter number of elements to insert in the list: ");
  scanf("%d", &n);
  for (int i = 0; i < n; i++) {
    printf("Enter value for node %d: ", i + 1);
    scanf("%d", &value);
    struct Node* newNode = createNode(value);
```

```
        if (head == NULL) {
            head = newNode;
        } else {
            struct Node* temp = head;
            while (temp->next != NULL) {
                temp = temp->next;
            }
            temp->next = newNode;
            newNode->prev = temp;
        }
    }
    return head;
}
void displayList(struct Node* head) {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }
    struct Node* temp = head;
    printf("Doubly Linked List: ");
    while (temp != NULL) {
        printf("%d <-> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}
void insertLeft(struct Node** head, int value, int newValue) {
    struct Node* newNode = createNode(newValue);
    struct Node* temp = *head;
    while (temp != NULL) {
        if (temp->data == value) {
            if (temp->prev != NULL) {
                temp->prev->next = newNode;
                newNode->prev = temp->prev;
            } else {
                *head = newNode;
            }
            newNode->next = temp;
            temp->prev = newNode;
            printf("Inserted %d to the left of %d.\n", newValue, value);
            return;
        }
        temp = temp->next;
    }
    printf("Node with value %d not found.\n", value);
}
void deleteNode(struct Node** head, int value) {
    struct Node* temp = *head;
    while (temp != NULL) {
        if (temp->data == value) {
            if (temp->prev != NULL) {
                temp->prev->next = temp->next;
            } else {
                *head = temp->next;
            }
            if (temp->next != NULL) {
                temp->next->prev = temp->prev;
            }
            free(temp);
            printf("Deleted node with value %d.\n", value);
            return;
        }
        temp = temp->next;
    }
    printf("Node with value %d not found.\n", value);
}
int main() {
    struct Node* head = NULL;
    int choice, value, newValue;
    while (1) {
        printf("\n1. Create Doubly Linked List\n2. Insert Left of Node\n3. Delete Node by 
Value\n4. Display List\n5. Exit\nEnter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                head = createDoublyLinkedList();
                break;
            case 2:
                printf("Enter value to insert left of: ");
                scanf("%d", &value);
                printf("Enter value to insert: ");
                scanf("%d", &newValue);
                insertLeft(&head, value, newValue);
                break;
            case 3:
                printf("Enter value to delete: ");
                scanf("%d", &value);
                deleteNode(&head, value);
                break;
            case 4:
                displayList(head);
                break;
            case 5:
                exit(0);
            default:
                printf("Invalid choice! Try again.\n");
        }
    }
    return 0;
}
```

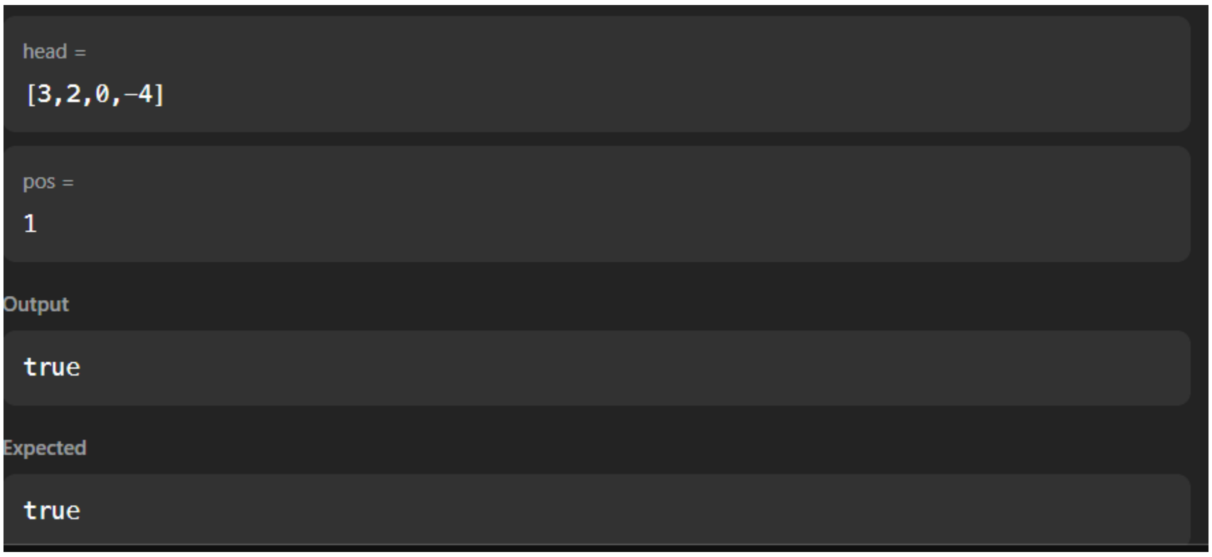


Output

```
1. Create Doubly Linked List
2. Insert Left of Node
3. Delete Node by Value
4. Display List
5. Exit
Enter your choice: 1
Enter number of elements to insert in the list: 3
Enter value for node 1: 4
Enter value for node 2: 5
Enter value for node 3: 6

1. Create Doubly Linked List
2. Insert Left of Node
3. Delete Node by Value
4. Display List
5. Exit
Enter your choice: 2
Enter value to insert left of: 5
Enter value to insert: 555
Inserted 555 to the left of 5.

1. Create Doubly Linked List
2. Insert Left of Node
3. Delete Node by Value
4. Display List
5. Exit
Enter your choice: 4
Doubly Linked List: 4 <-> 555 <-> 5 <-> 6 <-> NULL
```

Leet Code 4:

```
bool hasCycle(struct ListNode *head) {
    if (head == NULL || head->next == NULL) {
        return false;
    }
    struct ListNode *slow = head;
    struct ListNode *fast = head;
    while (fast != NULL && fast->next != NULL) {
        slow = slow->next;
        fast = fast->next->next;
        if (slow == fast) {
            return true;
        }
    }
    return false;
}
```



```
head =
[3,2,0,-4]

pos =
1

Output
true

Expected
true
```

Lab program 8:

Write a program

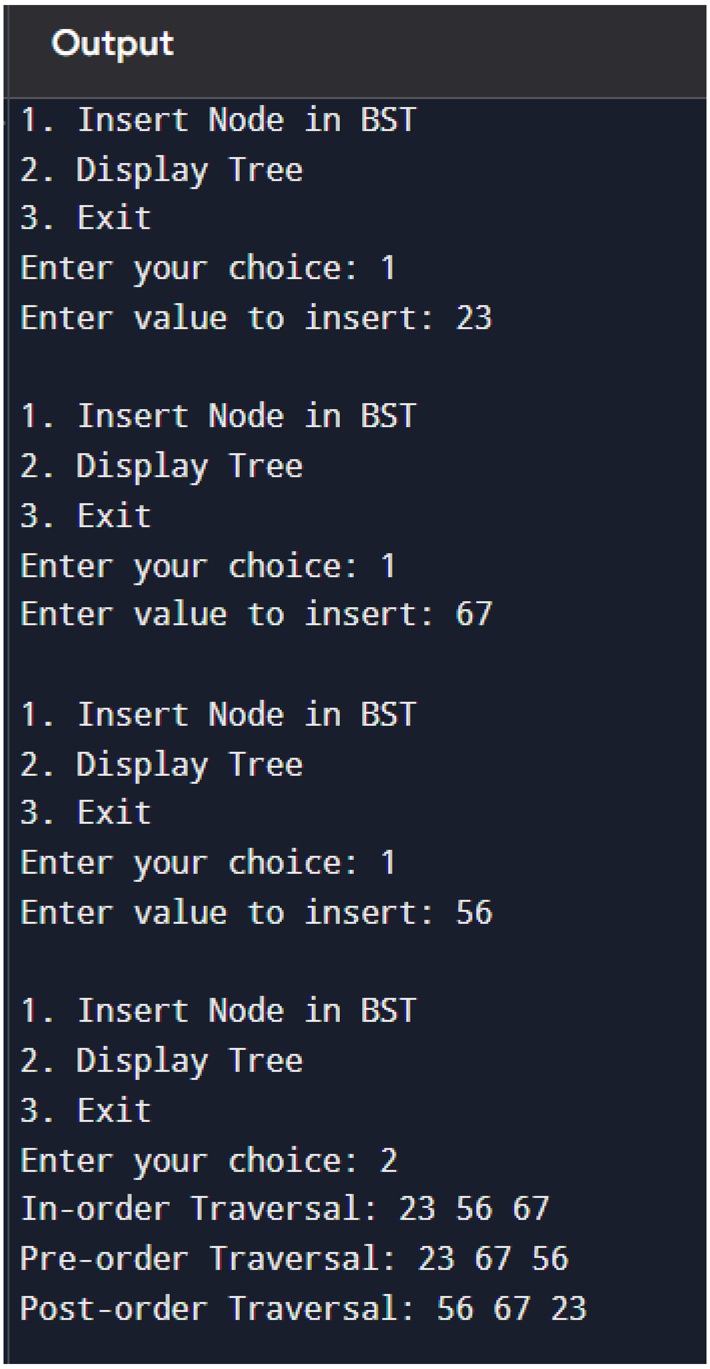
a) To construct a binary Search tree.

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.

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
   int data;
   struct Node* left;
   struct Node* right;
};
struct Node* createNode(int value) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = value;
   newNode->left = newNode->right = NULL;
   return newNode;
}
struct Node* insert(struct Node* root, int value) {
   if (root == NULL) {
      return createNode(value);
   }
   if (value < root->data) {
      root->left = insert(root->left, value);
   } else if (value > root->data) {
      root->right = insert(root->right, value);
   }
   return root;
```

```
}
void inorder(struct Node* root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right);
    }
}
void preorder(struct Node* root) {
    if (root != NULL) {
        printf("%d ", root->data);
        preorder(root->left);
        preorder(root->right);
    }
}
void postorder(struct Node* root) {
    if (root != NULL) {
        postorder(root->left);
        postorder(root->right);
        printf("%d ", root->data);
    }
}
void displayTree(struct Node* root) {
    if (root == NULL) {
        printf("Tree is empty.\n");
        return;
    }
    printf("In-order Traversal: ");
    inorder(root);
    printf("\n");
    printf("Pre-order Traversal: ");
    preorder(root);
    printf("\n");
    printf("Post-order Traversal: ");
    postorder(root);
    printf("\n");
}
int main() {
    struct Node* root = NULL;
    int choice, value;
    while (1) {
        printf("\n1. Insert Node in BST\n2. Display Tree\n3. Exit\nEnter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter value to insert: ");
                scanf("%d", &value);
                root = insert(root, value);
                break;
            case 2:
                displayTree(root);
                break;
            case 3:
                exit(0);
            default:
                printf("Invalid choice! Try again.\n");
        }
    }
    return 0;
}
```



Output

```
1. Insert Node in BST
2. Display Tree
3. Exit
Enter your choice: 1
Enter value to insert: 23

1. Insert Node in BST
2. Display Tree
3. Exit
Enter your choice: 1
Enter value to insert: 67

1. Insert Node in BST
2. Display Tree
3. Exit
Enter your choice: 1
Enter value to insert: 56

1. Insert Node in BST
2. Display Tree
3. Exit
Enter your choice: 2
In-order Traversal: 23 56 67
Pre-order Traversal: 23 67 56
Post-order Traversal: 56 67 23
```

Leet Code 5:

```
bool isPalindrome(struct ListNode* head) {
   if (head == NULL || head->next == NULL) {
      return true;
   }
   struct ListNode* slow = head;
   struct ListNode* fast = head;
   while (fast != NULL && fast->next != NULL) {
      slow = slow->next;
      fast = fast->next->next;
   }
   struct ListNode* prev = NULL;
   struct ListNode* curr = slow;
   struct ListNode* next = NULL;
   while (curr != NULL) {
      next = curr->next;
      curr->next = prev;
      prev = curr;
      curr = next;
   }
struct ListNode* firstHalf = head;
   struct ListNode* secondHalf = prev;
   while (secondHalf != NULL) {
      if (firstHalf->val != secondHalf->val) {
         return false;
      }
      firstHalf = firstHalf->next;
      secondHalf = secondHalf->next;
   }
```

```
    return true;
}
```

Input

```
head =
[1,2,2,1]
```

Output

```
true
```

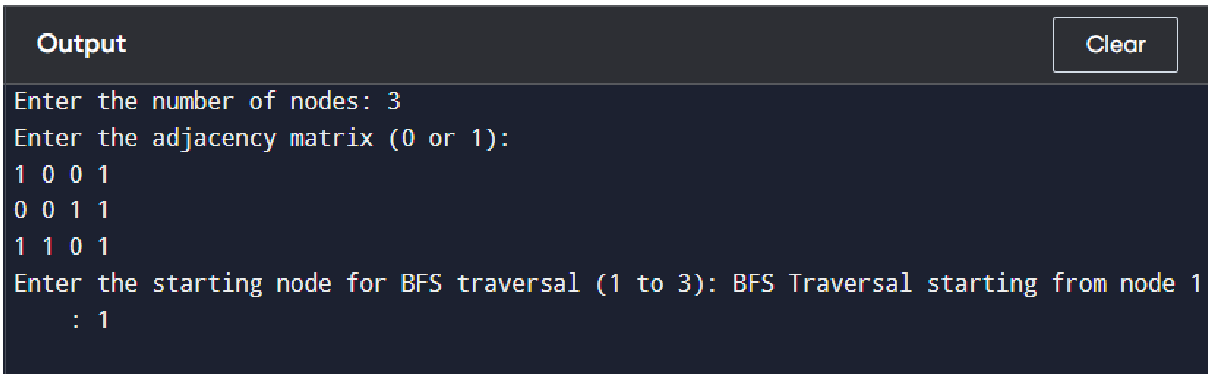
Expected

```
true
```

Lab program 9a:

Write a program to traverse a graph using BFS method.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define MAX 50

struct Queue {
    int items[MAX];
    int front, rear;
};

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

bool isEmpty(struct Queue* q) {
    return q->front == -1;
}

bool isFull(struct Queue* q) {
    return q->rear == MAX - 1;
}

void enqueue(struct Queue* q, int value) {
    if (isFull(q)) {
        printf("Queue is full.\n");
        return;
    }
    if (q->front == -1) {
        q->front = 0;
    }
    q->rear++;
    q->items[q->rear] = value;
}

int dequeue(struct Queue* q) {
    if (isEmpty(q)) {
        printf("Queue is empty.\n");
        return -1;
    }
    int item = q->items[q->front];
    if (q->front == q->rear) {
        q->front = q->rear = -1;
    } else {
        q->front++;
    }
    return item;
}

void bfs(int graph[MAX][MAX], int start, int n) {
    bool visited[MAX] = {false};
    struct Queue q;
    initQueue(&q);

    visited[start] = true;
    enqueue(&q, start);

    printf("BFS Traversal starting from node %d: ", start + 1);
    while (!isEmpty(&q)) {
        int node = dequeue(&q);
        printf("%d ", node + 1);

        for (int i = 0; i < n; i++) {
            if (graph[node][i] == 1 && !visited[i]) {
                visited[i] = true;
                enqueue(&q, i);
            }
        }
    }
    printf("\n");
}

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

    if (n > MAX) {
        printf("Error: Number of nodes exceeds maximum limit (%d).\n", MAX);
        return 1;
    }

    int graph[MAX][MAX];
    printf("Enter the adjacency matrix (0 or 1):\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            scanf("%d", &graph[i][j]);
        }
    }
    printf("Enter the starting node for BFS traversal (1 to %d): ", n);
    scanf("%d", &start);

    if (start < 1 || start > n) {
        printf("Error: Invalid starting node.\n");
        return 1;
    }
    bfs(graph, start - 1, n);
    return 0;
}
```

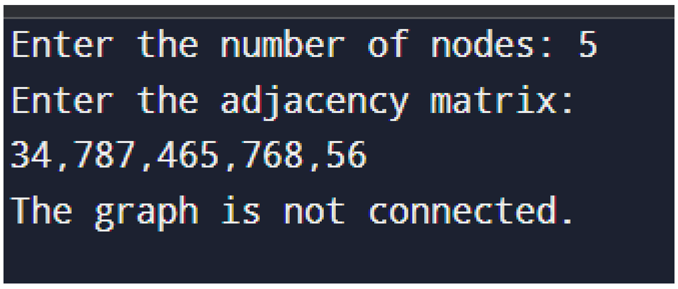


```
Enter the number of nodes: 3
Enter the adjacency matrix (0 or 1):
1 0 0 1
0 0 1 1
1 1 0 1
Enter the starting node for BFS traversal (1 to 3): BFS Traversal starting from node 1
    : 1
```

Lab program 9b:

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

```
#include <stdio.h>
#include <stdbool.h>
#define MAX 50
void dfs(int graph[MAX][MAX], bool visited[MAX], int node, int n) {
    visited[node] = true;
    for (int i = 0; i < n; i++) {
        if (graph[node][i] == 1 && !visited[i]) {
            dfs(graph, visited, i, n);
        }
    }
}
bool isConnected(int graph[MAX][MAX], int n) {
    bool visited[MAX] = {false};
    dfs(graph, visited, 0, n);
    for (int i = 0; i < n; i++) {
        if (!visited[i]) {
            return false;
        }
    }
    return true;
}
int main() {
    int n;
    printf("Enter the number of nodes: ");
    scanf("%d", &n);
    int graph[MAX][MAX];
    printf("Enter the adjacency matrix:\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            scanf("%d", &graph[i][j]);
        }
    }
    if (isConnected(graph, n)) {
        printf("The graph is connected.\n");
    } else {
        printf("The graph is not connected.\n");
    }
    return 0;
}
```



```
Enter the number of nodes: 5
Enter the adjacency matrix:
34,787,465,768,56
The graph is not 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 -&gt; L as

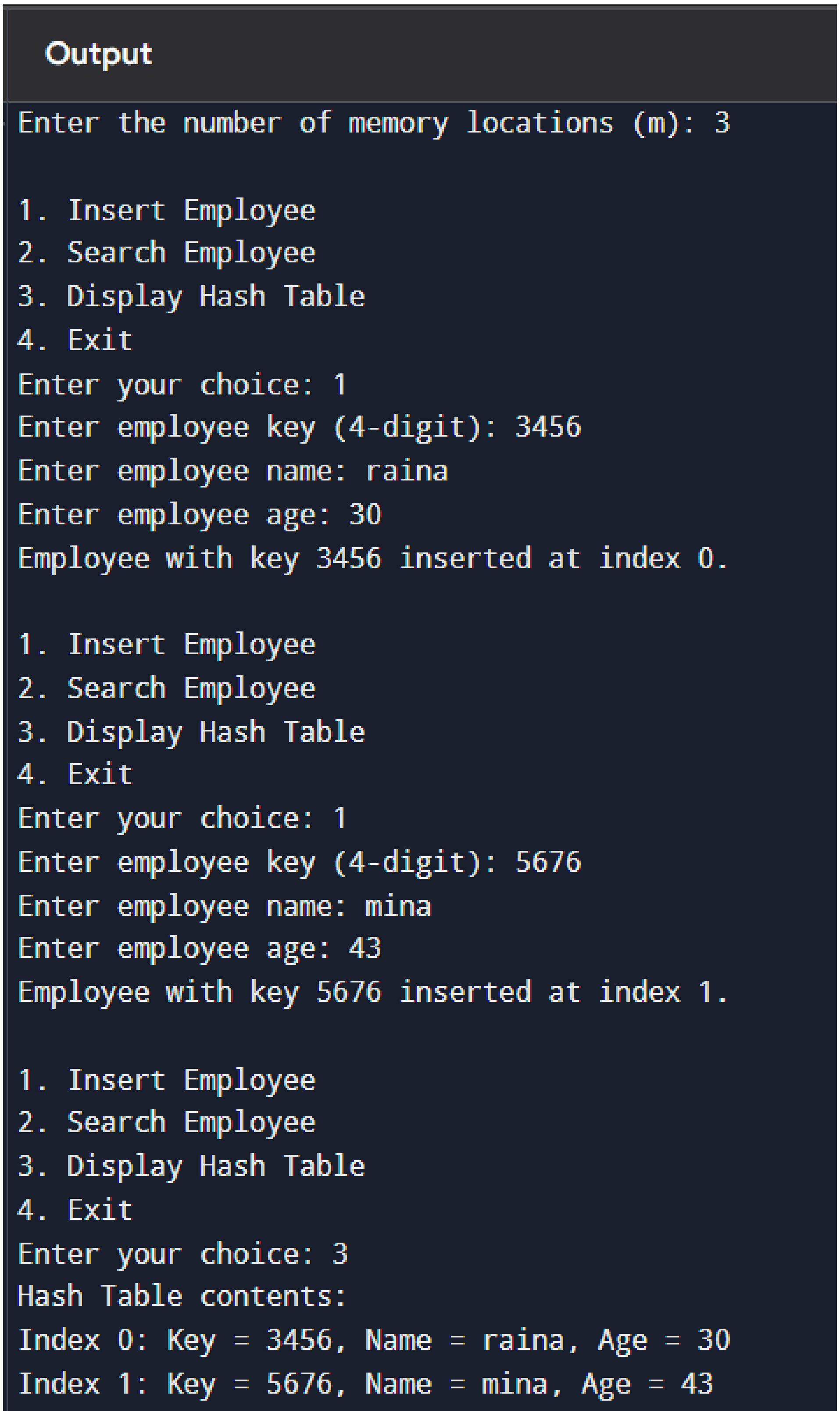
H(K)=K mod m (remainder method), and implement hashing technique to

map a given key K to the address space L.

Resolve the collision (if any) using linear probing.

```
#include <stdio.h>

#include <stdlib.h>


#define MAX 50


struct Employee {

   int key;

   char name[50];

   int age;

};


struct Employee hashTable[MAX];

void initializeHashTable() {

   for (int i = 0; i < MAX; i++) {

      hashTable[i].key = -1;

   }

}
```

```
int hashFunction(int key, int m) {
    return key % m;
}

int linearProbing(int key, int m) {
    int index = hashFunction(key, m);
    int i = 0;
    while (hashTable[(index + i) % m].key != -1) {
        i++;
    }
    return (index + i) % m;
}

void insertEmployee(int key, const char* name, int age, int m) {
    int index = hashFunction(key, m);
    if (hashTable[index].key != -1) {
        index = linearProbing(key, m);
    }

    hashTable[index].key = key;
    snprintf(hashTable[index].name, sizeof(hashTable[index].name), "%s", name);
    hashTable[index].age = age;
    printf("Employee with key %d inserted at index %d.\n", key, index);
}

void searchEmployee(int key, int m) {
    int index = hashFunction(key, m);
    int i = 0;
    while (hashTable[(index + i) % m].key != -1) {
        if (hashTable[(index + i) % m].key == key) {
            printf("Employee found at index %d: Key = %d, Name = %s, Age = %d\n",
                   (index + i) % m, hashTable[(index + i) % m].key,
                   hashTable[(index + i) % m].name, hashTable[(index + i) % m].age);
            return;
        }
        i++;
    }
    printf("Employee with key %d not found.\n", key);
}

void displayHashTable(int m) {
    printf("Hash Table contents:\n");
    for (int i = 0; i < m; i++) {
        if (hashTable[i].key != -1) {
            printf("Index %d: Key = %d, Name = %s, Age = %d\n", i, hashTable[i].key,
                   hashTable[i].name, hashTable[i].age);
        }
    }
}

int main() {
    int m;
    printf("Enter the number of memory locations (m): ");
    scanf("%d", &m);
    initializeHashTable();
    int choice, key, age;
    char name[50];
    while (1) {
        printf("\n1. Insert Employee\n2. Search Employee\n3. Display Hash Table\n4.
Exit\nEnter your choice: ");

        scanf("%d", &choice);

        switch (choice) {

            case 1:

                printf("Enter employee key (4-digit): ");

                scanf("%d", &key);

                printf("Enter employee name: ");

                scanf("%s", name);

                printf("Enter employee age: ");

                scanf("%d", &age);

                insertEmployee(key, name, age, m);

                break;

            case 2:

                printf("Enter employee key to search: ");

                scanf("%d", &key);

                searchEmployee(key, m);

                break;

            case 3:

                displayHashTable(m);

                break;

            case 4:

                exit(0);

            default:

                printf("Invalid choice! Please try again.\n");

        }

    }


    return 0;

}
```


Output

```
Enter the number of memory locations (m): 3

1. Insert Employee
2. Search Employee
3. Display Hash Table
4. Exit
Enter your choice: 1
Enter employee key (4-digit): 3456
Enter employee name: raina
Enter employee age: 30
Employee with key 3456 inserted at index 0.

1. Insert Employee
2. Search Employee
3. Display Hash Table
4. Exit
Enter your choice: 1
Enter employee key (4-digit): 5676
Enter employee name: mina
Enter employee age: 43
Employee with key 5676 inserted at index 1.

1. Insert Employee
2. Search Employee
3. Display Hash Table
4. Exit
Enter your choice: 3
Hash Table contents:
Index 0: Key = 3456, Name = raina, Age = 30
Index 1: Key = 5676, Name = mina, Age = 43
```