VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
On

DATA STRUCTURES (23CS3PCDST)

Submitted by

Rudraksh Singh (1BM23CS276)

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 **Rudraksh Singh (1BM23CS276)**, 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 Sheet

Course outcomes:

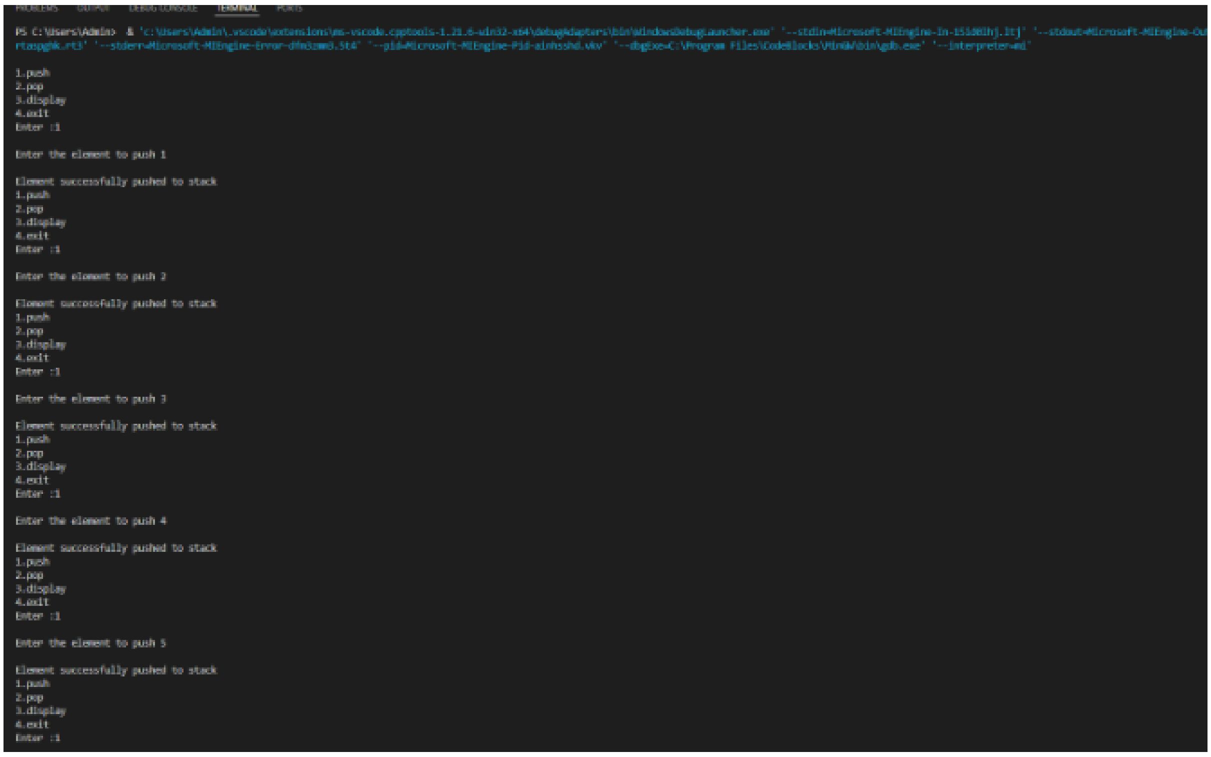
CO1	Apply the concept of linear and nonlinear data structures.
CO2	Analyze data structure operations for a given problem
CO3	Design and develop solutions using the operations of linear and nonlinear data structure for a given specification.
CO4	Conduct practical experiments for demonstrating the operations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

Lab program 1:

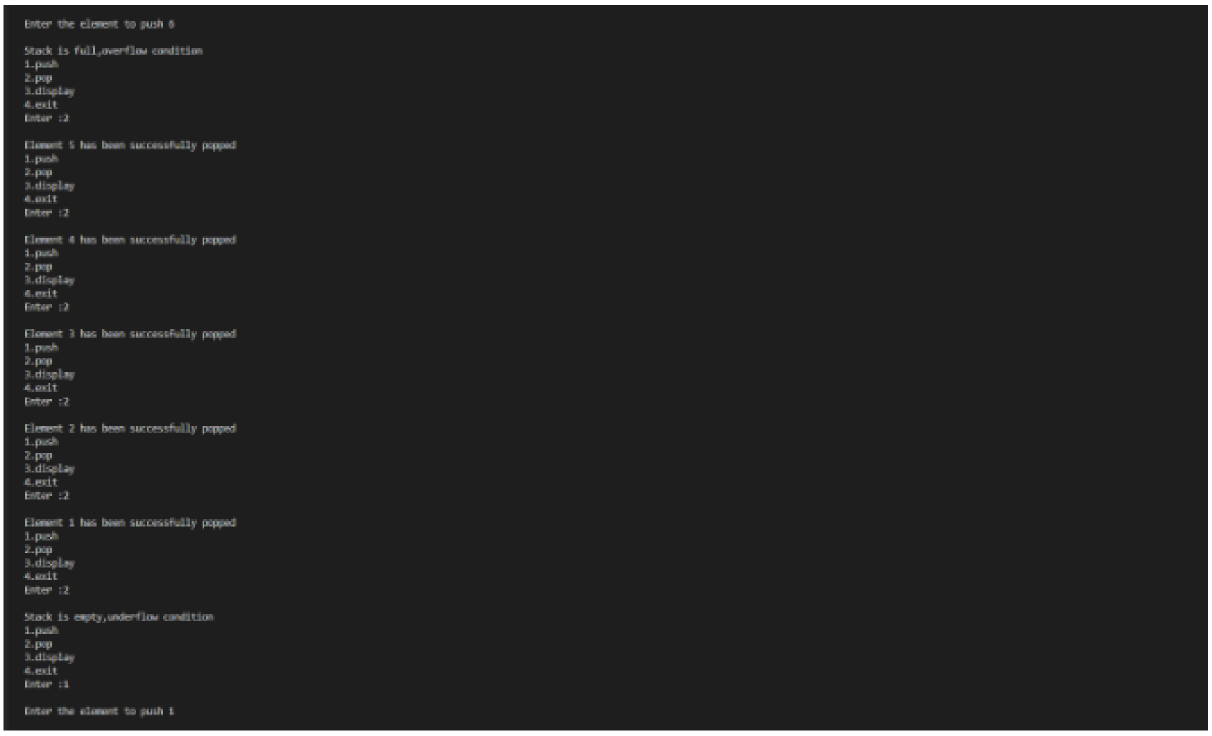
Write a program to simulate the working of stack using an array with the following: a) Push b) Pop c) Display The program should print appropriate messages for stack overflow, stack underflow

```
#include<stdio.h>
#include<stdlib.h>
#define SIZE 5
int stack[SIZE];
int top=-1;

void push(int a)
{
  if(top==SIZE-1)
  {
    printf("\nStack is full,overflow condition");
  }
  else
  {
    top++;
    stack[top]=a;
    printf("\nElement successfully pushed to stack");
  }
}
void pop()
{
  if(top==-1)
  {
    printf("\nStack is empty,underflow condition");
  }
  else
  {
    int ele = stack[top];
    printf("\nElement %d has been successfully popped",ele);
    top--;
  }
}
void display()
{
  if(top==-1)
  {
    printf("\nstack is empty,underflow condition");
  }
  else
  {
    for(int i=top;i>-1;i--)
    {
      printf("%d ",stack[i]);
    }
  }
}
```

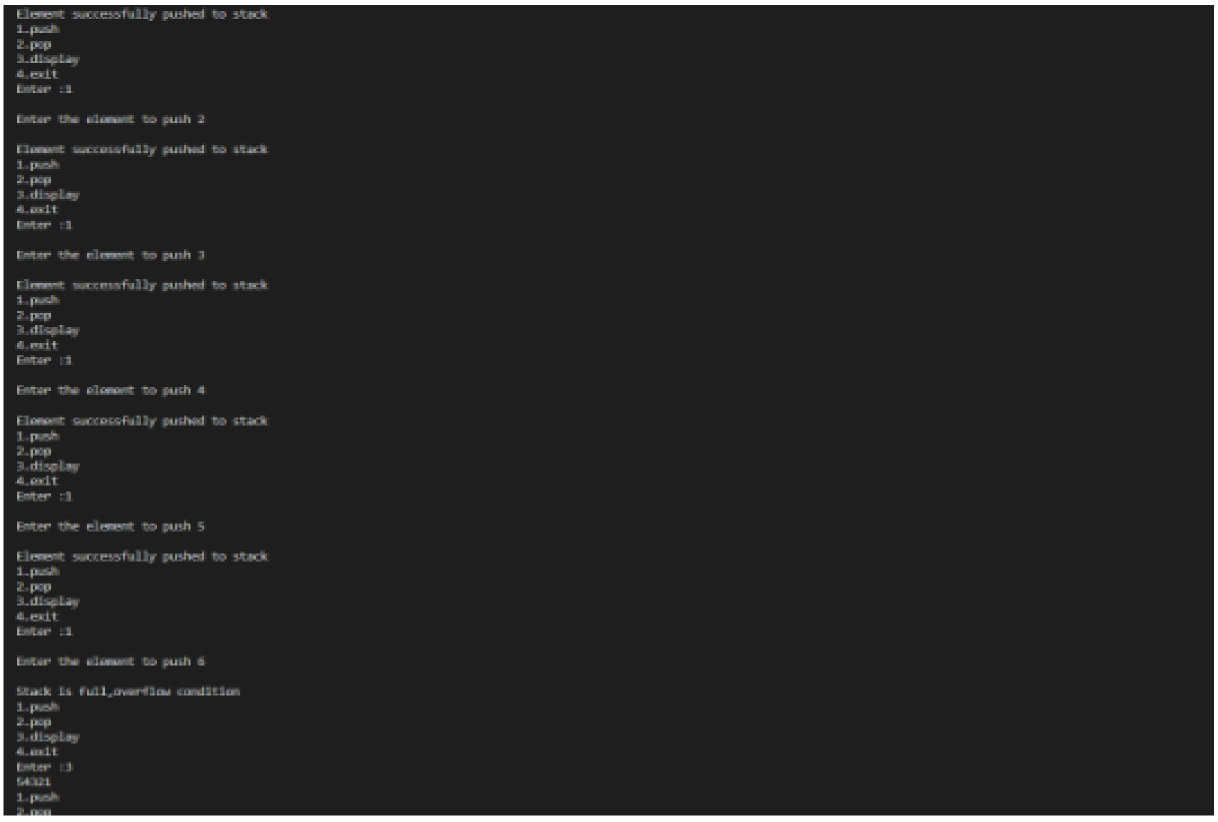
```c
void main()
{
   int c,e;
   while(1)
   {
      printf("\n\n1.push\n2.pop\n3.display\n4.exit\nEnter :");
      scanf("%d",&c);
      switch (c)
      {
         case 1: printf("\nEnter the element to push ");
               scanf("%d",&e);
               push(e);
               break;
         case 2: pop();
               break;
         case 3: display();
               break;
         case 4: exit(1);
         default : printf("\nInvalid input");
      }
   }
}
```



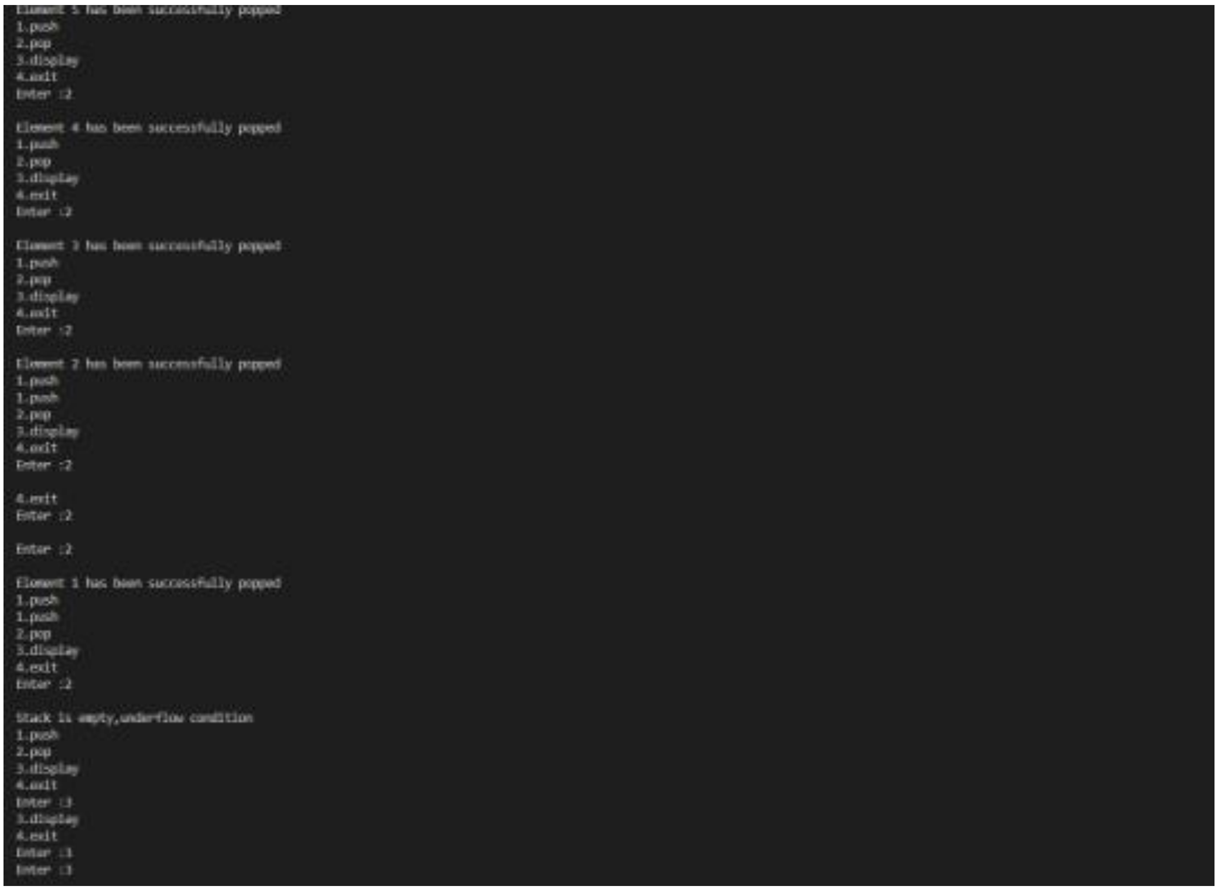
```
PS C:\Users\Admin>  & 'c:\Users\Admin\.vscode\extensions\ms-vscode.cpptools-1.21.6-win32-x64\debugAdapters\bin\WindowsDebugLauncher.exe' '--stdin=Microsoft-MIEngine-In-151d0lhj.1tj' '--stdout=Microsoft-MIEngine-Out
rtaspghk.rt3' '--stderr=Microsoft-MIEngine-Error-dfm3zmd3.5t4' '--pid=Microsoft-MIEngine-Pid-ainhsshd.vkv' '--dbgExe=C:\Program Files\CodeBlocks\MinGW\bin\gdb.exe' '--interpreter=mi'

1.push
2.pop
3.display
4.exit
Enter :1

Enter the element to push 1

Element successfully pushed to stack
1.push
2.pop
3.display
4.exit
Enter :1

Enter the element to push 2

Element successfully pushed to stack
1.push
2.pop
3.display
4.exit
Enter :1

Enter the element to push 3

Element successfully pushed to stack
1.push
2.pop
3.display
4.exit
Enter :1

Enter the element to push 4

Element successfully pushed to stack
1.push
2.pop
3.display
4.exit
Enter :1

Enter the element to push 5

Element successfully pushed to stack
1.push
2.pop
3.display
4.exit
Enter :1
```



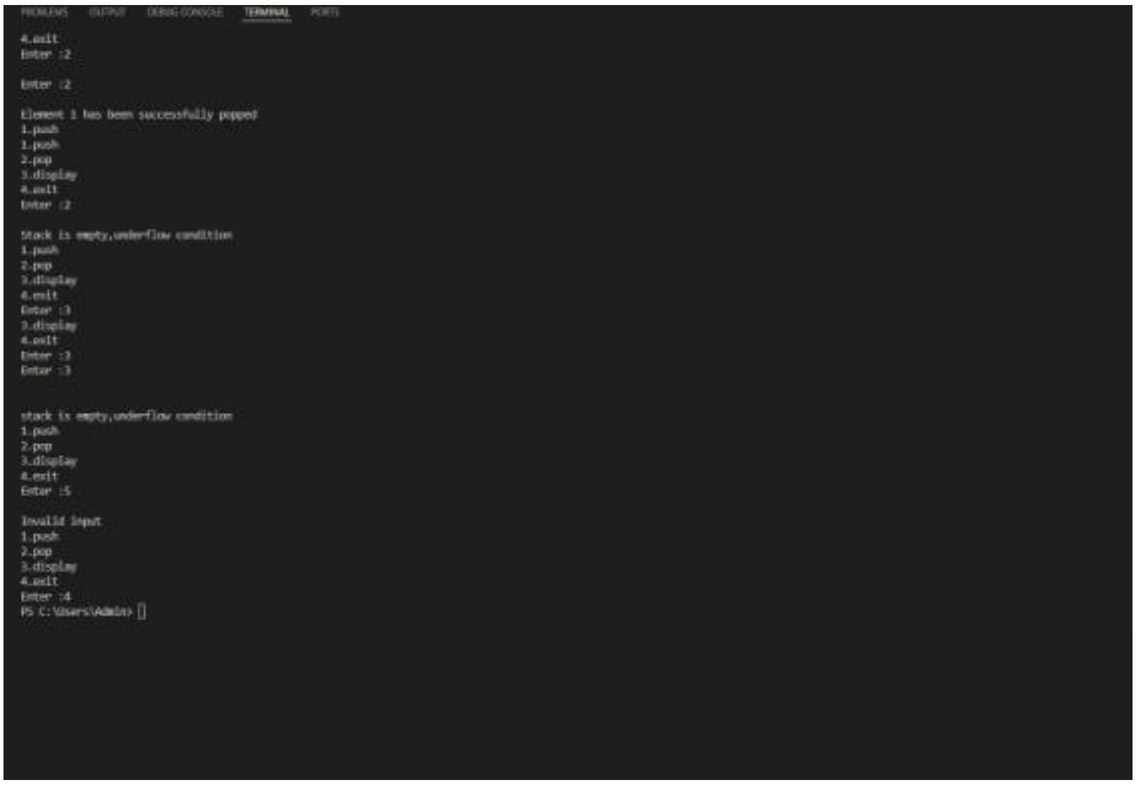
```
Enter the element to push 6

Stack is full,overflow condition
1.push
2.pop
3.display
4.exit
Enter :2

Element 5 has been successfully popped
1.push
2.pop
3.display
4.exit
Enter :2

Element 4 has been successfully popped
1.push
2.pop
3.display
4.exit
Enter :2

Element 3 has been successfully popped
1.push
2.pop
3.display
4.exit
Enter :2

Element 2 has been successfully popped
1.push
2.pop
3.display
4.exit
Enter :2

Element 1 has been successfully popped
1.push
2.pop
3.display
4.exit
Enter :2

Stack is empty,underflow condition
1.push
2.pop
3.display
4.exit
Enter :1

Enter the element to push 1
```



```
Element successfully pushed to stack
1.push
2.pop
3.display
4.exit
Enter :1

Enter the element to push 2

Element successfully pushed to stack
1.push
2.pop
3.display
4.exit
Enter :1

Enter the element to push 3

Element successfully pushed to stack
1.push
2.pop
3.display
4.exit
Enter :1

Enter the element to push 4

Element successfully pushed to stack
1.push
2.pop
3.display
4.exit
Enter :1

Enter the element to push 5

Element successfully pushed to stack
1.push
2.pop
3.display
4.exit
Enter :1

Enter the element to push 6

Stack is full,overflow condition
1.push
2.pop
3.display
4.exit
Enter :3
54321
1.push
2.pop
```



```
Element 5 has been successfully popped
1.push
2.pop
3.display
4.exit
Enter :2

Element 4 has been successfully popped
1.push
2.pop
3.display
4.exit
Enter :2

Element 3 has been successfully popped
1.push
2.pop
3.display
4.exit
Enter :2

Element 2 has been successfully popped
1.push
1.push
2.pop
3.display
4.exit
Enter :2

4.exit
Enter :2

Enter :2

Element 1 has been successfully popped
1.push
1.push
2.pop
3.display
4.exit
Enter :2

Stack is empty,underflow condition
1.push
2.pop
3.display
4.exit
Enter :3
3.display
4.exit
Enter :1
Enter :3
```



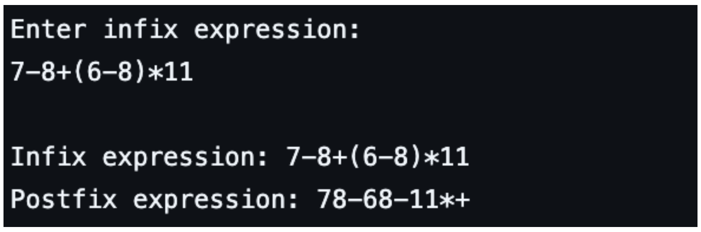
```
4.exit
Enter :2

Enter :2

Element 1 has been successfully popped
1.push
1.push
2.pop
3.display
4.exit
Enter :2

Stack is empty,underflow condition
1.push
2.pop
3.display
4.exit
Enter :3
3.display
4.exit
Enter :3
Enter :3


stack is empty,underflow condition
1.push
2.pop
3.display
4.exit
Enter :5

Invalid input
1.push
2.pop
3.display
4.exit
Enter :4
PS C:\Users\Admin> []
```

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 <string.h>

int index1 = 0, pos = 0, top = -1, length;
char symbol, temp, infix[20], postfix[20], stack[20];
void infixToPostfix();
void push(char symbol);
char pop();
int pred(char symbol);
int main() {
    printf("Enter infix expression:\n");
    scanf("%s", infix);
    infixToPostfix();
    printf("\nInfix expression: %s", infix);
    printf("\nPostfix expression: %s\n", postfix);
    return 0;
}
void infixToPostfix() {
    length = strlen(infix);
    push('#'); // Push an initial dummy character to the stack
    while (index1 < length) {
        symbol = infix[index1];
        switch (symbol) {
            case '(':
                push(symbol);
                break;
            case ')':
                temp = pop();
                while (temp != '(') {
                    postfix[pos++] = temp;
                    temp = pop();
                }
                break;
            case '+':
            case '-':
            case '*':
            case '/':
            case '^':
                while (pred(stack[top]) >= pred(symbol)) {
                    temp = pop();
                    postfix[pos++] = temp;
                }
                push(symbol);
                break;
            default:
                postfix[pos++] = symbol;
        }
        index1++;
    }
    while (top > 0) {
        temp = pop();
        postfix[pos++] = temp;
    }
    postfix[pos] = '\0';
}

void push(char symbol) {
    top = top + 1;
    stack[top] = symbol;
```

```
}

char pop() {
    char symb;
    symb = stack[top];
    top = top - 1;
    return symb;
}

int pred(char symbol) {
    int p;
    switch (symbol) {
        case '^':
            p = 3;
            break;
        case '*':
        case '/':
            p = 2;
            break;
        case '+':
        case '-':
            p = 1;
            break;
        case '(':
            p = 0;
            break;
        case '#':
            p = -1;
            break;
        default:
            p = -1;
    }
    return p;
}
```

Output:

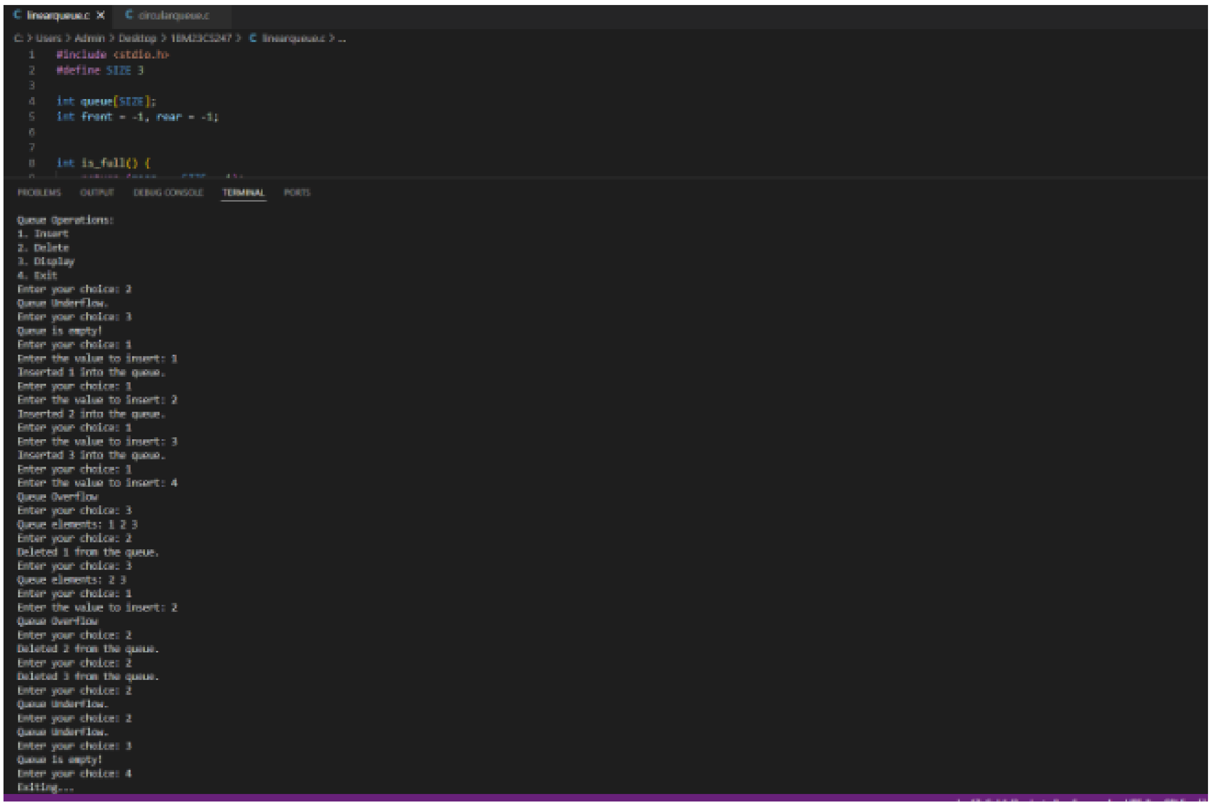


```
Enter infix expression:
7-8+(6-8)*11

Infix expression: 7-8+(6-8)*11
Postfix expression: 78-68-11*+
```

Lab Program 3

a) **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>
#define SIZE 3

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


int is_full() {
  return (rear == SIZE - 1);
}


int is_empty() {
  return (front == -1);
}


void insert(int value) {
  if (is_full()) {
    printf("Queue Overflow\n");
    return;
  }
  if (front == -1)
    front = 0;
  queue[++rear] = value;
  printf("Inserted %d into the queue.\n", value);
}


void delete() {
  if (is_empty()) {
    printf("Queue Underflow.\n");
    return;
  }
  printf("Deleted %d from the queue.\n", queue[front]);
  front++;
  if (front > rear) {
    front = -1;
    rear = -1;
  }
}

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


int main() {
  int choice, value;
  printf("\nQueue Operations:\n");
  printf("1. Insert\n");
  printf("2. Delete\n");
  printf("3. Display\n");
  printf("4. Exit\n");
  while (1) {
    printf("Enter your choice: ");
    scanf("%d", &choice);

    switch (choice) {
        case 1:
            printf("Enter the value to insert: ");
            scanf("%d", &value);
            insert(value);
            break;
        case 2:
            delete();
            break;
        case 3:
            display();
            break;
        case 4:
            printf("Exiting...\n");
            return 0;
        default:
            printf("Invalid choice! Please try again.\n");
    }
  }
}
```



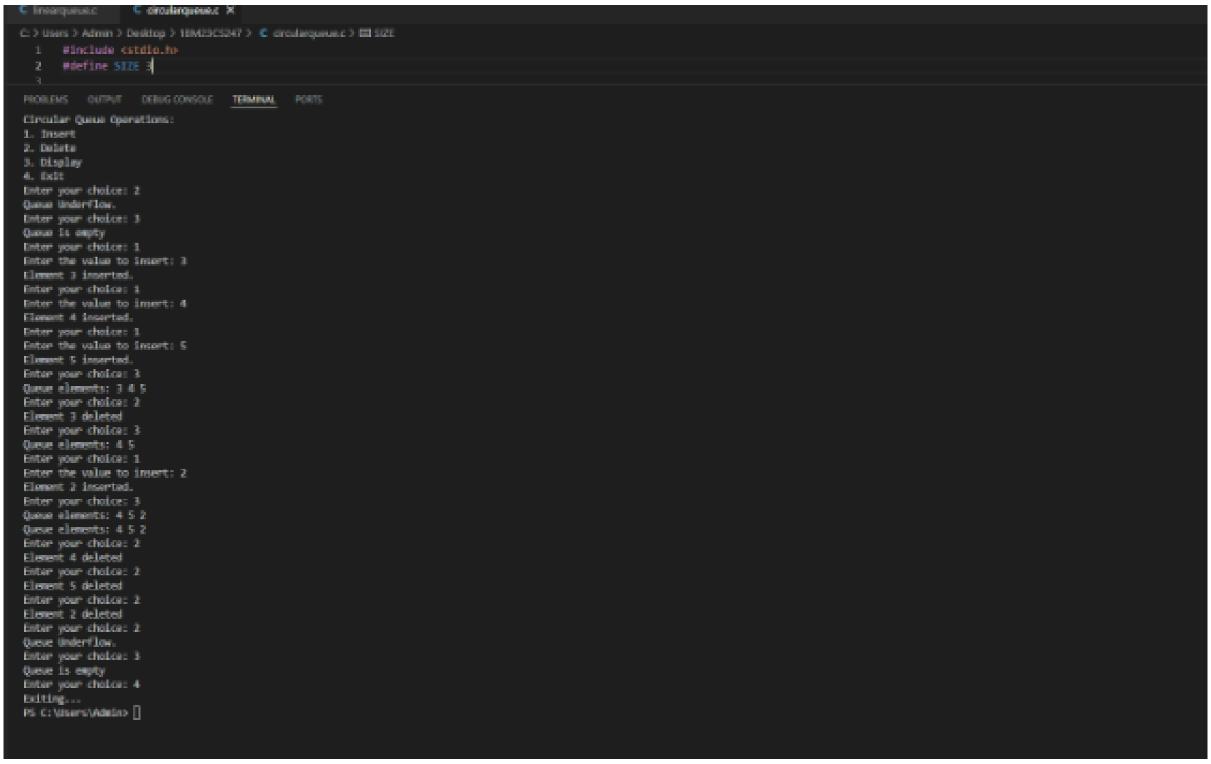
```c
#include <stdio.h>
#define SIZE 3

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


int is_full() {
```

```
Queue Operations:
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 2
Queue Underflow.
Enter your choice: 3
Queue is empty!
Enter your choice: 1
Enter the value to insert: 1
Inserted 1 into the queue.
Enter your choice: 1
Enter the value to insert: 2
Inserted 2 into the queue.
Enter your choice: 1
Enter the value to insert: 3
Inserted 3 into the queue.
Enter your choice: 1
Enter the value to insert: 4
Queue Overflow
Enter your choice: 3
Queue elements: 1 2 3
Enter your choice: 2
Deleted 1 from the queue.
Enter your choice: 3
Queue elements: 2 3
Enter your choice: 1
Enter the value to insert: 2
Queue Overflow
Enter your choice: 2
Deleted 2 from the queue.
Enter your choice: 2
Deleted 3 from the queue.
Enter your choice: 2
Queue Underflow.
Enter your choice: 2
Queue Underflow.
Enter your choice: 3
Queue is empty!
Enter your choice: 4
Exiting...
```

b) 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

```
#include <stdio.h>
#define SIZE 3

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


int is_full() {
   return (front == (rear + 1) % SIZE);
}


int is_empty() {
   return (front == -1);
}


void insert(int n) {
   if (is_full()) {
      printf("Queue Overflow\n");
      return;
   }
   if (is_empty())
      front = 0;
      rear = 0;
   else
      rear = (rear + 1) % SIZE;
   queue[rear] = n;
   printf("Element %d inserted.\n", n);
}


void delete() {
   if (is_empty()) {
      printf("Queue Underflow.\n");
      return;
   }
   printf("Element %d deleted\n", queue[front]);
   if (front == rear){
      front = -1;
      rear = -1;
   }
   else
      front = (front + 1) % SIZE;
}
```

```
void display() {
   if (is_empty()) {
      printf("Queue is empty\n");
      return;
   }
   printf("Queue elements: ");
   int i = front;
   while (1) {
      printf("%d ", queue[i]);
      if (i == rear)
         break;
      i = (i + 1) % SIZE;
   }
   printf("\n");
}

int main() {
   int choice, value;
   printf("\nCircular Queue Operations:\n");
   printf("1. Insert\n");
   printf("2. Delete\n");
   printf("3. Display\n");
   printf("4. Exit\n");
   while (1) {
      printf("Enter your choice: ");
      scanf("%d", &choice);

      switch (choice) {
         case 1:
            printf("Enter the value to insert: ");
            scanf("%d", &value);
            insert(value);
            break;
         case 2:
            delete();
            break;
         case 3:
            display();
            break;
         case 4:
            printf("Exiting...\n");
            return 0;
         default:
            printf("Invalid choice! Please try again.\n");
      }
   }
}
```



```c
#include <stdio.h>
#define SIZE 3
```

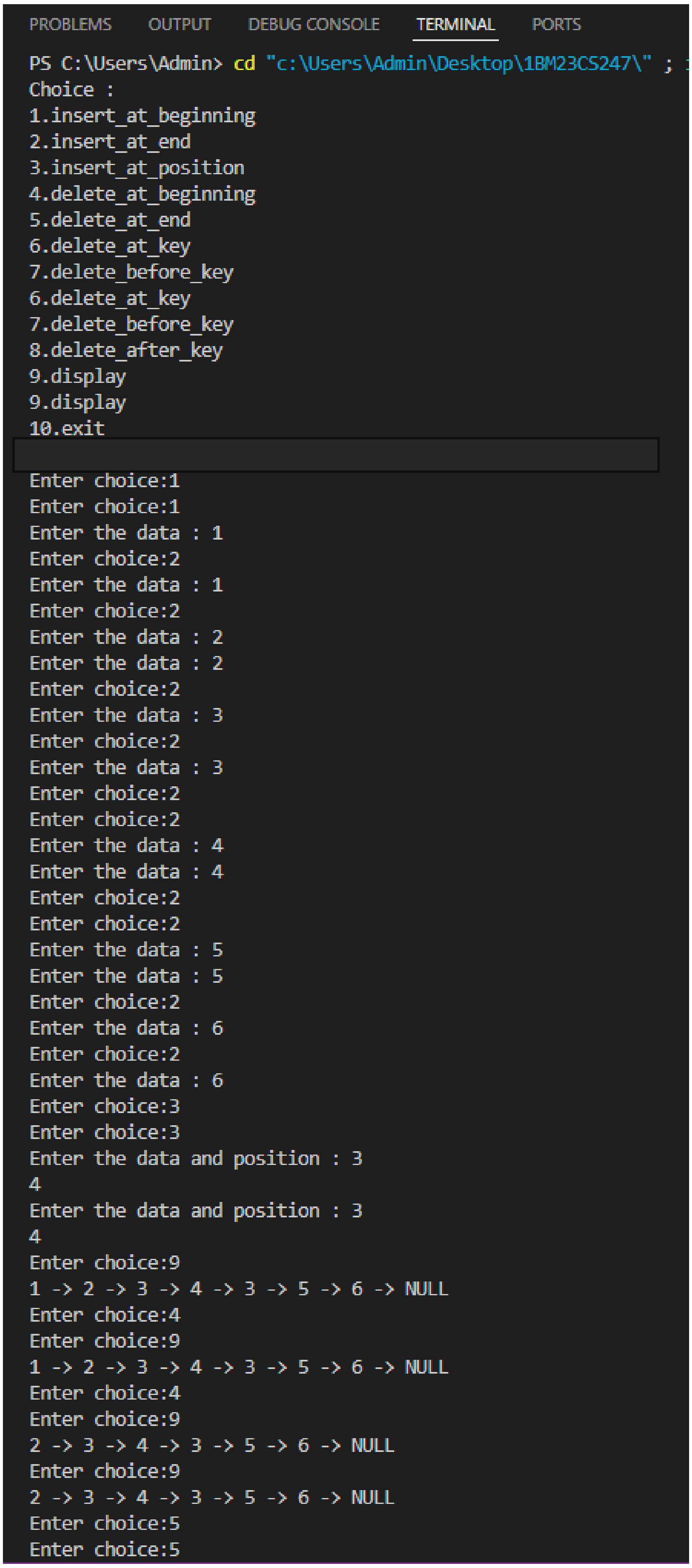
```
Circular Queue Operations:
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 2
Queue Underflow.
Enter your choice: 3
Queue is empty
Enter your choice: 1
Enter the value to insert: 3
Element 3 inserted.
Enter your choice: 1
Enter the value to insert: 4
Element 4 inserted.
Enter your choice: 1
Enter the value to insert: 5
Element 5 inserted.
Enter your choice: 3
Queue elements: 3 4 5
Enter your choice: 2
Element 3 deleted
Enter your choice: 3
Queue elements: 4 5
Enter your choice: 1
Enter the value to insert: 2
Element 2 inserted.
Enter your choice: 3
Queue elements: 4 5 2
Queue elements: 4 5 2
Enter your choice: 2
Element 4 deleted
Enter your choice: 2
Element 5 deleted
Enter your choice: 2
Element 2 deleted
Enter your choice: 2
Queue Underflow.
Enter your choice: 3
Queue is empty
Enter your choice: 4
Exiting...
PS C:\Users\Admin> 
```

Lab Program 4
WAP to Implement Singly Linked List with following operations
a) Create LinkedList.
b) Insertion of a node at first position, at any position and at end of list.
c) Deletion of first element, specified element and last element in the list.
Display the contents of the linked list.

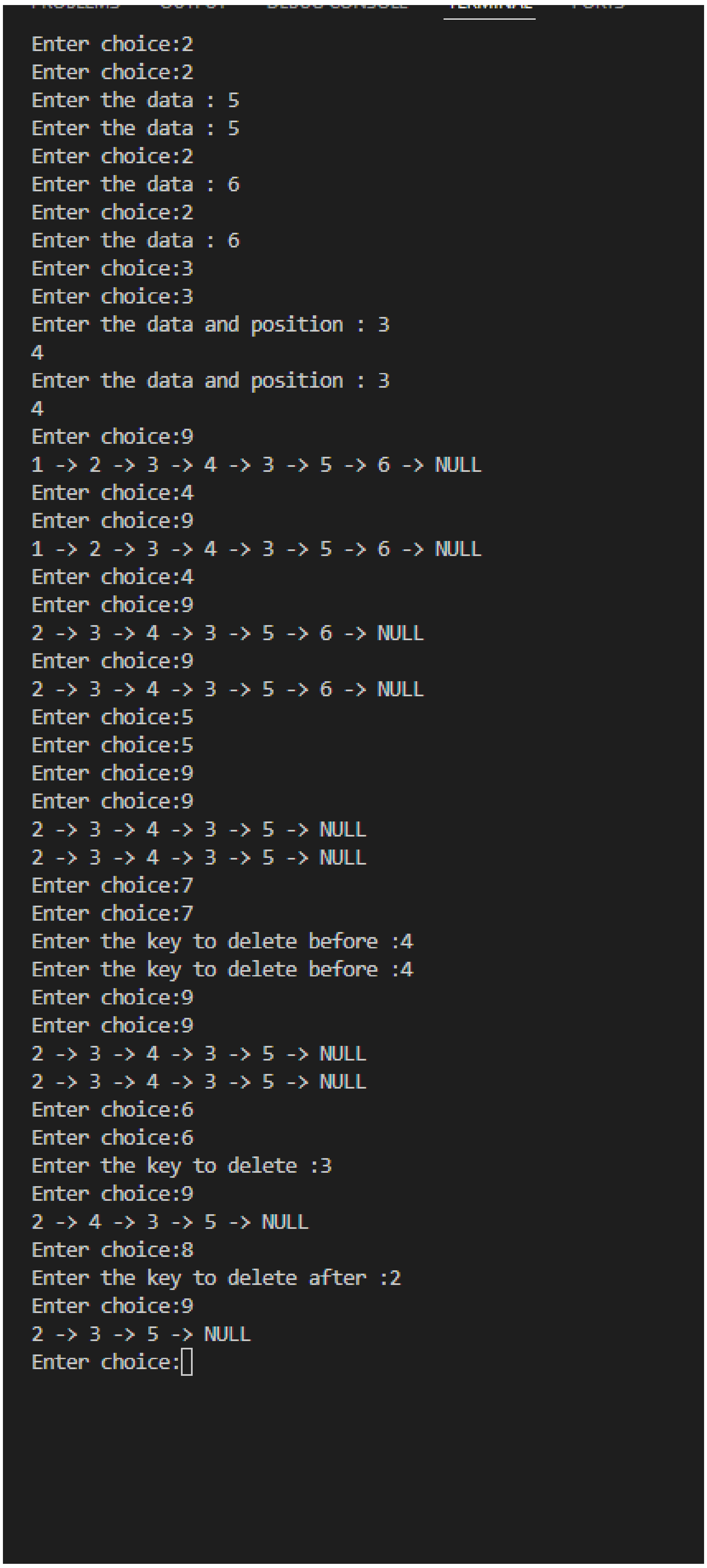
```
#include <stdio.h>
#include <stdlib.h>

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

// Create a new node
struct Node* create_node(int data) {
   struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
   new_node->data = data;
   new_node->next = NULL;
   return new_node;
}

// Insert node at the beginning
void insert_at_beginning(struct Node** head, int data) {
   struct Node* new_node = create_node(data);
   new_node->next = *head;
   *head = new_node;
}

// Insert node at the end
void insert_at_end(struct Node** head, int data) {
   struct Node* new_node = create_node(data);
   if (*head == NULL) {
      *head = new_node;
      return;
   }
   struct Node* temp = *head;
   while (temp->next != NULL) {
      temp = temp->next;
   }
   temp->next = new_node;
}

// Insert node at a specific position
void insert_at_position(struct Node** head, int data, int position) {
   if (position < 0) return;  // Invalid position
   if (position == 0) {
      insert_at_beginning(head, data);
      return;
   }
```

```
    struct Node* new_node = create_node(data);
    struct Node* temp = *head;
    for (int i = 0; i < position - 1; i++) {
        if (temp == NULL) return;  // Position out of range
        temp = temp->next;
    }
    new_node->next = temp->next;
    temp->next = new_node;
}

// Delete node at the beginning
void delete_at_beginning(struct Node** head) {
    if (*head != NULL) {
        struct Node* temp = *head;
        *head = (*head)->next;
        free(temp);
    }
}

// Delete node at the end
void delete_at_end(struct Node** head) {
    if (*head == NULL) return;
    if ((*head)->next == NULL) {
        free(*head);
        *head = NULL;
        return;
    }
    struct Node* temp = *head;
    while (temp->next && temp->next->next != NULL) {
        temp = temp->next;
    }
    free(temp->next);
    temp->next = NULL;
}

// Delete node with a specific key
void delete_at_key(struct Node** head, int key) {
    if (*head == NULL) return;
    if ((*head)->data == key) {
        struct Node* temp = *head;
        *head = (*head)->next;
        free(temp);
        return;
    }
    struct Node* temp = *head;
    while (temp->next != NULL && temp->next->data != key) {
        temp = temp->next;
    }
    if (temp->next == NULL) return;
    struct Node* to_delete = temp->next;
    temp->next = temp->next->next;
    free(to_delete);
}

// Delete node before the key
void delete_before_key(struct Node** head, int key) {
    if (*head == NULL || (*head)->next == NULL) return;
    if ((*head)->next->data == key) {
        struct Node* temp = *head;
        *head = (*head)->next;
        free(temp);
        return;
    }
    struct Node* temp = *head;
    while (temp->next != NULL && temp->next->next != NULL) {
        if (temp->next->next->data == key) {
            struct Node* to_delete = temp;
            temp = temp->next;
            free(to_delete);
            return;
        }
        temp = temp->next;
    }
}

// Delete node after the key
void delete_after_key(struct Node** head, int key) {
    struct Node* temp = *head;
    while (temp != NULL && temp->data != key) {
        temp = temp->next;
    }
    if (temp != NULL && temp->next != NULL) {
        struct Node* to_delete = temp->next;
        temp->next = temp->next->next;
        free(to_delete);
    }
}

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

// Free all nodes to avoid memory leaks
void free_list(struct Node** head) {
    struct Node* temp;
    while (*head != NULL) {
        temp = *head;
        *head = (*head)->next;
        free(temp);
    }
}

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

    printf("Choice
:\n1.insert_at_beginning\n2.insert_at_end\n3.insert_at_position\n4.delete_at_beginning\n5.delete
_at_end\n6.delete_at_key\n7.delete_before_key\n8.delete_after_key\n9.display\n10.exit\n");

    int c;
    while (1) {
        printf("Enter choice: ");
        scanf("%d", &c);
        switch (c) {
            case 1:
                printf("Enter the data: ");
                scanf("%d", &data);
                insert_at_beginning(&head, data);
                break;
            case 2:
                printf("Enter the data: ");
                scanf("%d", &data);
                insert_at_end(&head, data);
                break;
            case 3:
                printf("Enter the data and position: ");
                scanf("%d%d", &data, &key);
                insert_at_position(&head, data, key);
                break;
            case 4:
                delete_at_beginning(&head);
                break;
            case 5:
                delete_at_end(&head);
                break;
            case 6:
                printf("Enter the key to delete: ");
                scanf("%d", &key);
                delete_at_key(&head, key);
                break;
            case 7:
                printf("Enter the key to delete before: ");
                scanf("%d", &key);
                delete_before_key(&head, key);
                break;
            case 8:
```

```
            printf("Enter the key to delete after: ");
            scanf("%d", &key);
            delete_after_key(&head, key);
            break;
         case 9:
            display(head);
            break;
         case 10:
            exit(0);
         default:
            printf("Invalid choice...\n");
      }
   }
   return 0;
}
```



```
PS C:\Users\Admin> cd "c:\Users\Admin\Desktop\1BM23CS247\" ;
Choice :
1.insert_at_beginning
2.insert_at_end
3.insert_at_position
4.delete_at_beginning
5.delete_at_end
6.delete_at_key
7.delete_before_key
6.delete_at_key
7.delete_before_key
8.delete_after_key
9.display
9.display
10.exit

Enter choice:1
Enter choice:1
Enter the data : 1
Enter choice:2
Enter the data : 1
Enter choice:2
Enter the data : 2
Enter the data : 2
Enter choice:2
Enter the data : 3
Enter choice:2
Enter the data : 3
Enter choice:2
Enter choice:2
Enter the data : 4
Enter the data : 4
Enter choice:2
Enter choice:2
Enter the data : 5
Enter the data : 5
Enter choice:2
Enter the data : 6
Enter choice:2
Enter the data : 6
Enter choice:3
Enter choice:3
Enter the data and position : 3
4
Enter the data and position : 3
4
Enter choice:9
1 -> 2 -> 3 -> 4 -> 3 -> 5 -> 6 -> NULL
Enter choice:4
Enter choice:9
1 -> 2 -> 3 -> 4 -> 3 -> 5 -> 6 -> NULL
Enter choice:4
Enter choice:9
2 -> 3 -> 4 -> 3 -> 5 -> 6 -> NULL
Enter choice:9
2 -> 3 -> 4 -> 3 -> 5 -> 6 -> NULL
Enter choice:5
Enter choice:5
```

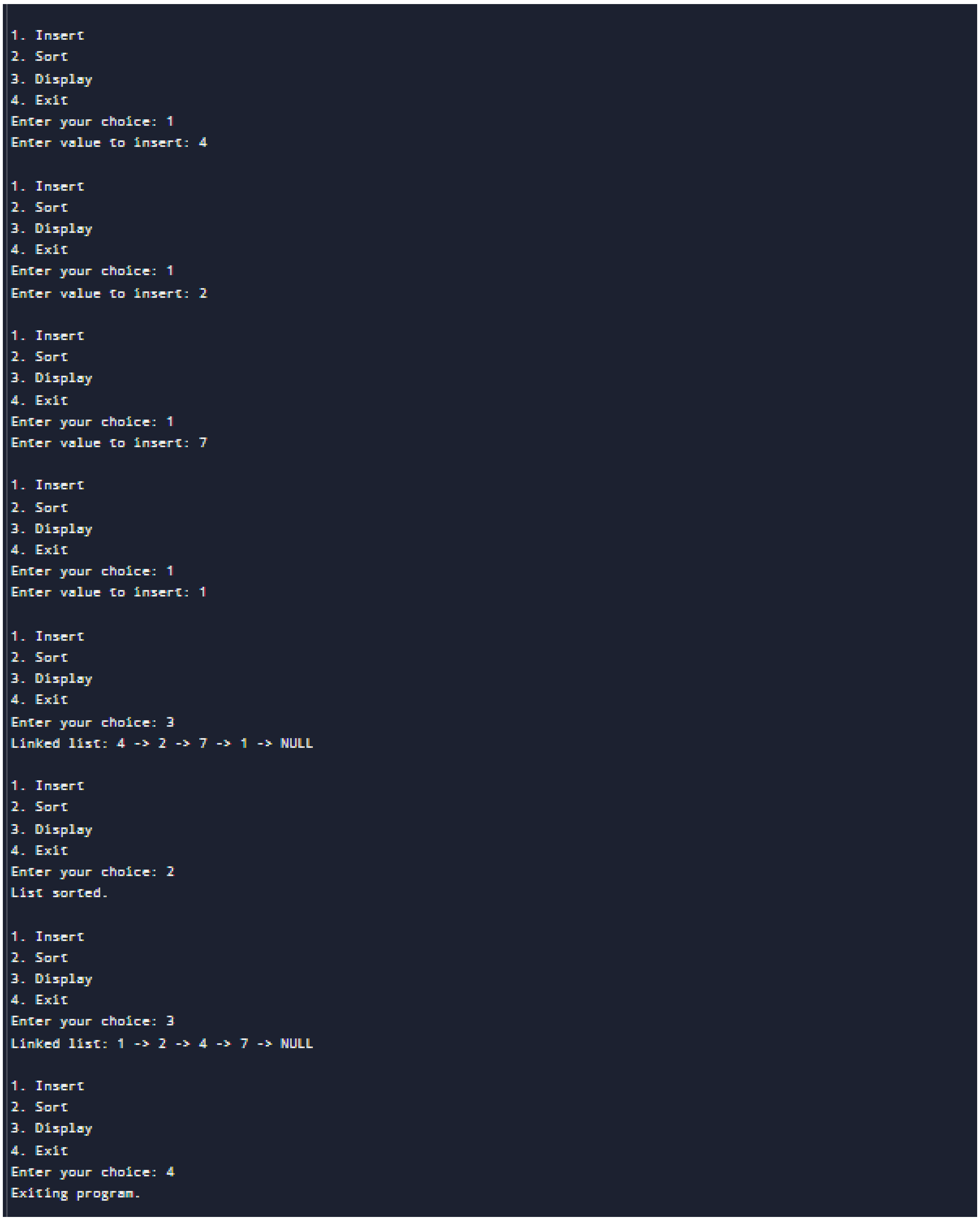


```
Enter choice:2
Enter choice:2
Enter the data : 5
Enter the data : 5
Enter choice:2
Enter the data : 6
Enter choice:2
Enter the data : 6
Enter choice:3
Enter choice:3
Enter the data and position : 3
4
Enter the data and position : 3
4
Enter choice:9
1 -> 2 -> 3 -> 4 -> 3 -> 5 -> 6 -> NULL
Enter choice:4
Enter choice:9
1 -> 2 -> 3 -> 4 -> 3 -> 5 -> 6 -> NULL
Enter choice:4
Enter choice:9
2 -> 3 -> 4 -> 3 -> 5 -> 6 -> NULL
Enter choice:9
2 -> 3 -> 4 -> 3 -> 5 -> 6 -> NULL
Enter choice:5
Enter choice:5
Enter choice:9
Enter choice:9
2 -> 3 -> 4 -> 3 -> 5 -> NULL
2 -> 3 -> 4 -> 3 -> 5 -> NULL
Enter choice:7
Enter choice:7
Enter the key to delete before :4
Enter the key to delete before :4
Enter choice:9
Enter choice:9
2 -> 3 -> 4 -> 3 -> 5 -> NULL
2 -> 3 -> 4 -> 3 -> 5 -> NULL
Enter choice:6
Enter choice:6
Enter the key to delete :3
Enter choice:9
2 -> 4 -> 3 -> 5 -> NULL
Enter choice:8
Enter the key to delete after :2
Enter choice:9
2 -> 3 -> 5 -> NULL
Enter choice:
```

Lab Program 5
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* insertAtEnd(struct Node* head, int value) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = value;
   newNode->next = NULL;
   if (!head) return newNode;

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

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

struct Node* sortList(struct Node* head) {
   if (!head || !head->next) return head;

   struct Node* current = head;
   while (current) {
      struct Node* next = current->next;
      while (next) {
         if (current->data > next->data) {
            int temp = current->data;
            current->data = next->data;
            next->data = temp;
         }
         next = next->next;
      }
      current = current->next;
   }
   return head;
}
```

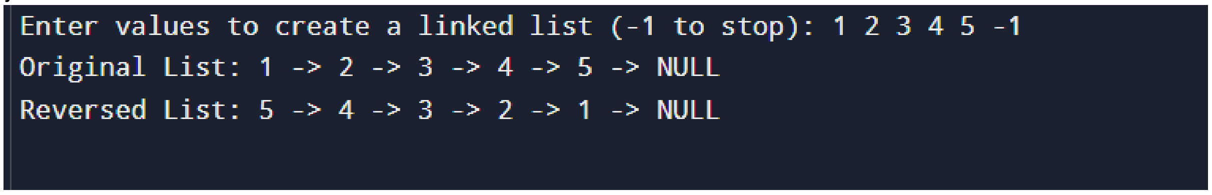
```
int main() {
   struct Node* head = NULL;
   int choice, value;

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

      switch (choice) {
         case 1:
            printf("Enter value to insert: ");
            scanf("%d", &value);
            head = insertAtEnd(head, value);
            break;
         case 2:
            head = sortList(head);
            printf("List sorted.\n");
            break;
         case 3:
            printf("Linked list: ");
            printList(head);
            break;
         case 4:
            printf("Exiting program.\n");
            break;
         default:
            printf("Invalid choice. Try again.\n");
      }
   } while (choice != 4);

   return 0;
}
```



```
1. Insert
2. Sort
3. Display
4. Exit
Enter your choice: 1
Enter value to insert: 4

1. Insert
2. Sort
3. Display
4. Exit
Enter your choice: 1
Enter value to insert: 2

1. Insert
2. Sort
3. Display
4. Exit
Enter your choice: 1
Enter value to insert: 7

1. Insert
2. Sort
3. Display
4. Exit
Enter your choice: 1
Enter value to insert: 1

1. Insert
2. Sort
3. Display
4. Exit
Enter your choice: 3
Linked list: 4 -> 2 -> 7 -> 1 -> NULL

1. Insert
2. Sort
3. Display
4. Exit
Enter your choice: 2
List sorted.

1. Insert
2. Sort
3. Display
4. Exit
Enter your choice: 3
Linked list: 1 -> 2 -> 4 -> 7 -> NULL

1. Insert
2. Sort
3. Display
4. Exit
Enter your choice: 4
Exiting program.
```

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

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

struct Node* insertAtEnd(struct Node* head, int value) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = value;
   newNode->next = NULL;
   if (!head) return newNode;

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

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

struct Node* reverseList(struct Node* head) {
   struct Node* prev = NULL;
   struct Node* current = head;
   struct Node* next = NULL;

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

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

   printf("Enter values to create a linked list (-1 to stop): ");
   do {
      scanf("%d", &value);
```

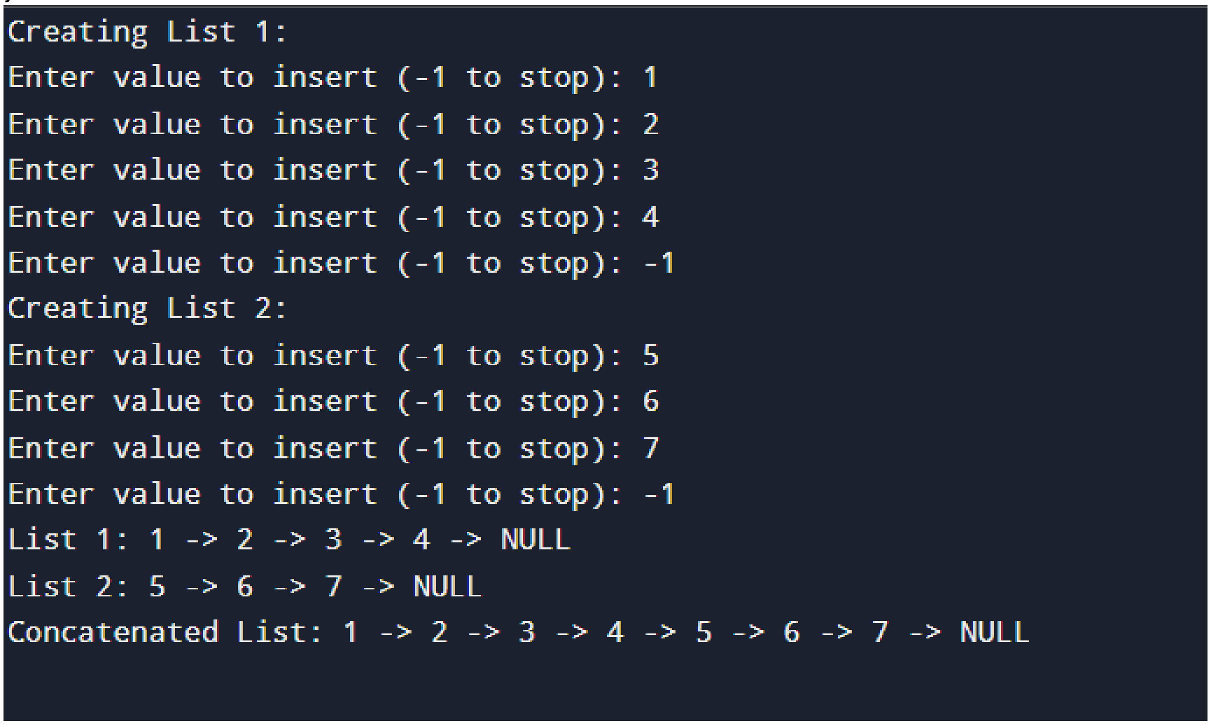
```
        if (value != -1) head = insertAtEnd(head, value);
    } while (value != -1);

    printf("Original List: ");
    printList(head);

    head = reverseList(head);

    printf("Reversed List: ");
    printList(head);

    return 0;
}
```



```
Enter values to create a linked list (-1 to stop): 1 2 3 4 5 -1
Original List: 1 -> 2 -> 3 -> 4 -> 5 -> NULL
Reversed List: 5 -> 4 -> 3 -> 2 -> 1 -> NULL
```

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

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

struct Node* insertAtEnd(struct Node* head, int value) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = value;
   newNode->next = NULL;
   if (!head) return newNode;

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

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

struct Node* concatenateLists(struct Node* head1, struct Node* head2) {
   if (!head1) return head2;
   if (!head2) return head1;

   struct Node* temp = head1;
   while (temp->next) temp = temp->next;
   temp->next = head2;
   return head1;
}

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

   int choice, value;

   printf("Creating List 1:\n");
   do {
      printf("Enter value to insert (-1 to stop): ");
      scanf("%d", &value);
      if (value != -1) list1 = insertAtEnd(list1, value);
```

```
    } while (value != -1);

    printf("Creating List 2:\n");
    do {
        printf("Enter value to insert (-1 to stop): ");
        scanf("%d", &value);
        if (value != -1) list2 = insertAtEnd(list2, value);
    } while (value != -1);

    printf("List 1: ");
    printList(list1);

    printf("List 2: ");
    printList(list2);

    list1 = concatenateLists(list1, list2);

    printf("Concatenated List: ");
    printList(list1);

    return 0;
}
```

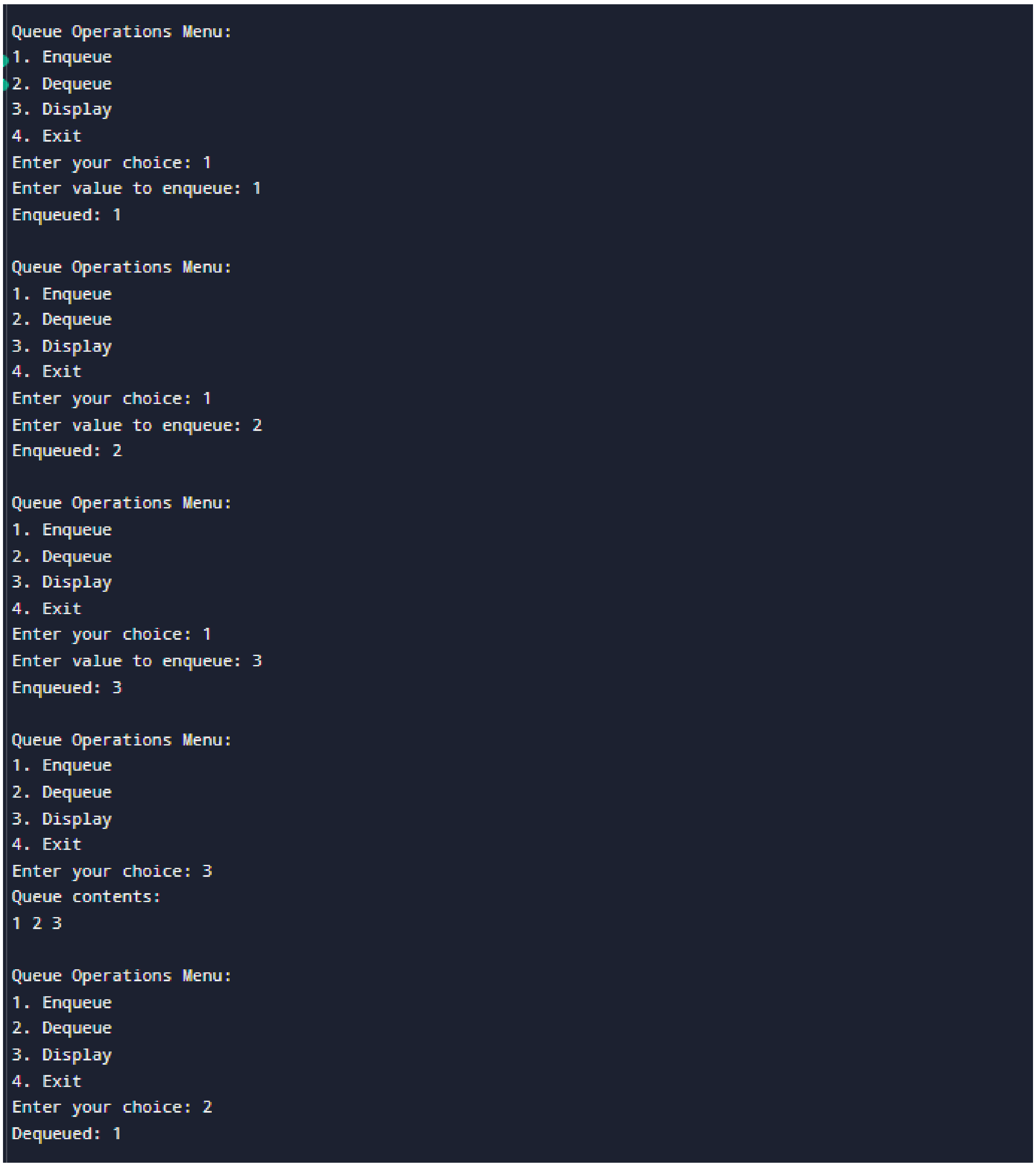


```
Creating List 1:
Enter value to insert (-1 to stop): 1
Enter value to insert (-1 to stop): 2
Enter value to insert (-1 to stop): 3
Enter value to insert (-1 to stop): 4
Enter value to insert (-1 to stop): -1
Creating List 2:
Enter value to insert (-1 to stop): 5
Enter value to insert (-1 to stop): 6
Enter value to insert (-1 to stop): 7
Enter value to insert (-1 to stop): -1
List 1: 1 -> 2 -> 3 -> 4 -> NULL
List 2: 5 -> 6 -> 7 -> NULL
Concatenated List: 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> NULL
```

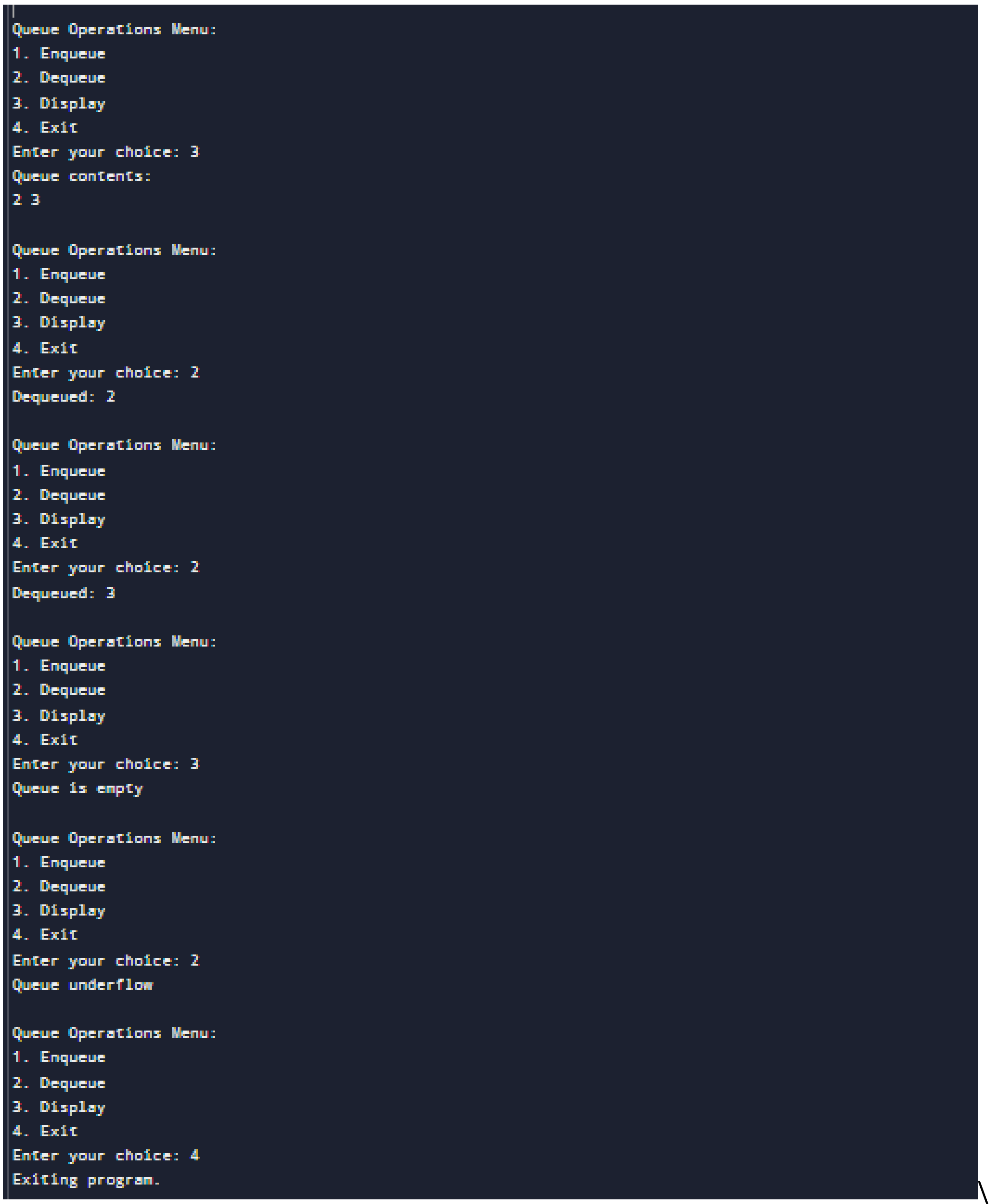
WAP to Implement Single Link List to simulate Stack & Queue Operations.

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

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

struct Queue {
   struct Node* front;
   struct Node* rear;
};

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

struct Queue* createQueue() {
   struct Queue* queue = (struct Queue*)malloc(sizeof(struct Queue));
   queue->front = NULL;
   queue->rear = NULL;
   return queue;
}

int isEmpty(struct Queue* queue) {
   return queue->front == NULL;
}

void enqueue(struct Queue* queue, int data) {
   struct Node* node = createNode(data);
   if (queue->rear == NULL) {
      queue->front = queue->rear = node;
      return;
   }
   queue->rear->next = node;
   queue->rear = node;
}

int dequeue(struct Queue* queue) {
   if (isEmpty(queue)) {
      printf("Queue underflow\n");
      return NULL;
   }
   struct Node* temp = queue->front;
   int data = temp->data;
   queue->front = queue->front->next;
```

```
    if (queue->front == NULL) queue->rear = NULL;
    free(temp);
    return data;
}


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

int main() {
    struct Queue* queue = createQueue();
    int choice, value;

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

        switch (choice) {
            case 1:
                printf("Enter value to enqueue: ");
                scanf("%d", &value);
                enqueue(queue, value);
                printf("Enqueued: %d\n", value);
                break;
            case 2:
                value = dequeue(queue);
                if (value != NULL) {
                    printf("Dequeued: %d\n", value);
                }
                break;
            case 3:
                display(queue);
                break;
            case 4:
                printf("Exiting program.\n");
                exit(0);
            default:
                printf("Invalid choice! Please try again.\n");
        }
    }
    return 0;
}
```



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

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

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

Queue Operations Menu:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 3
Queue contents:
1 2 3

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

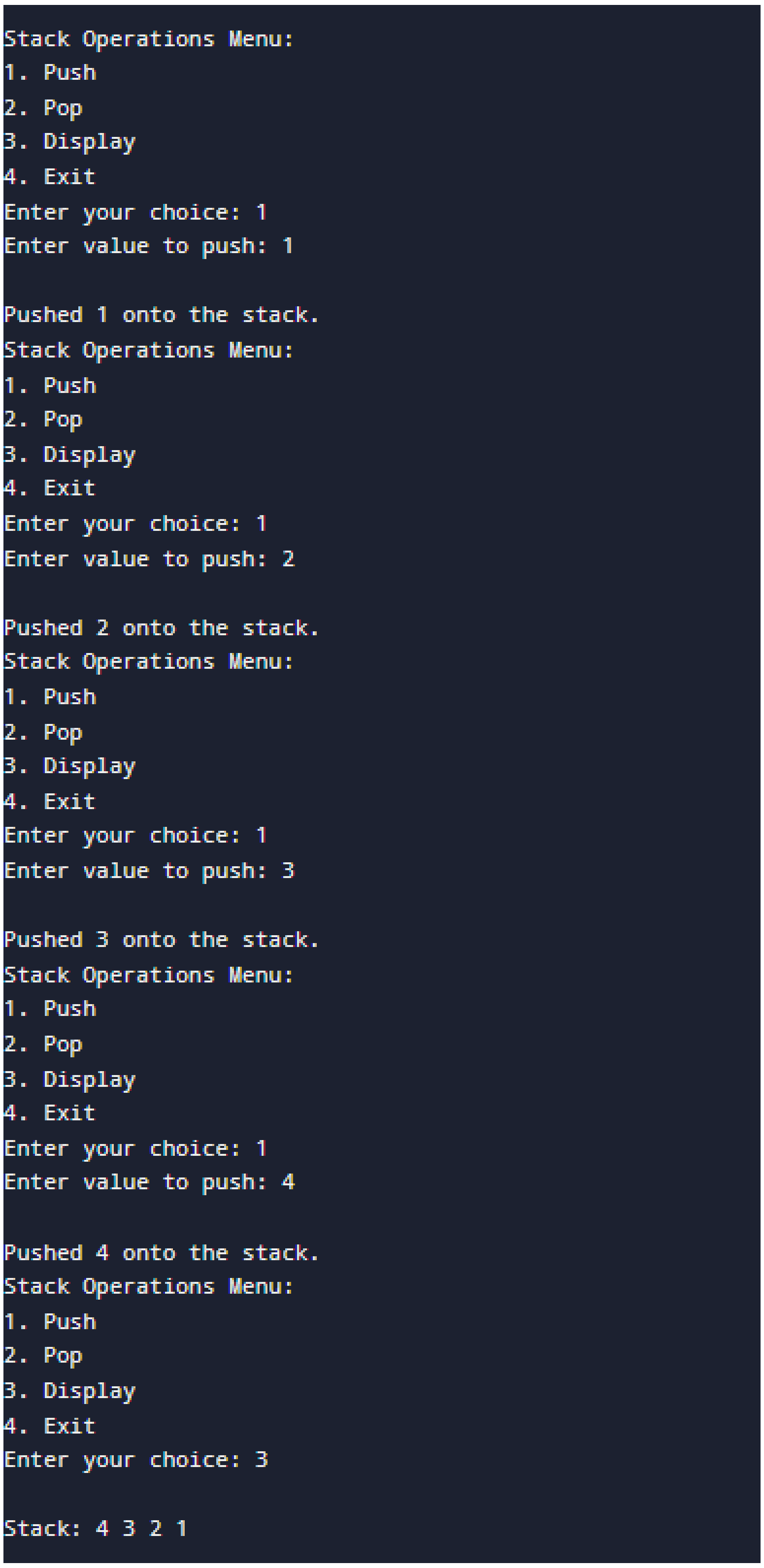


```
Queue Operations Menu:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 3
Queue contents:
2 3

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

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

Queue Operations Menu:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 3
Queue is empty

Queue Operations Menu:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 2
Queue underflow

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

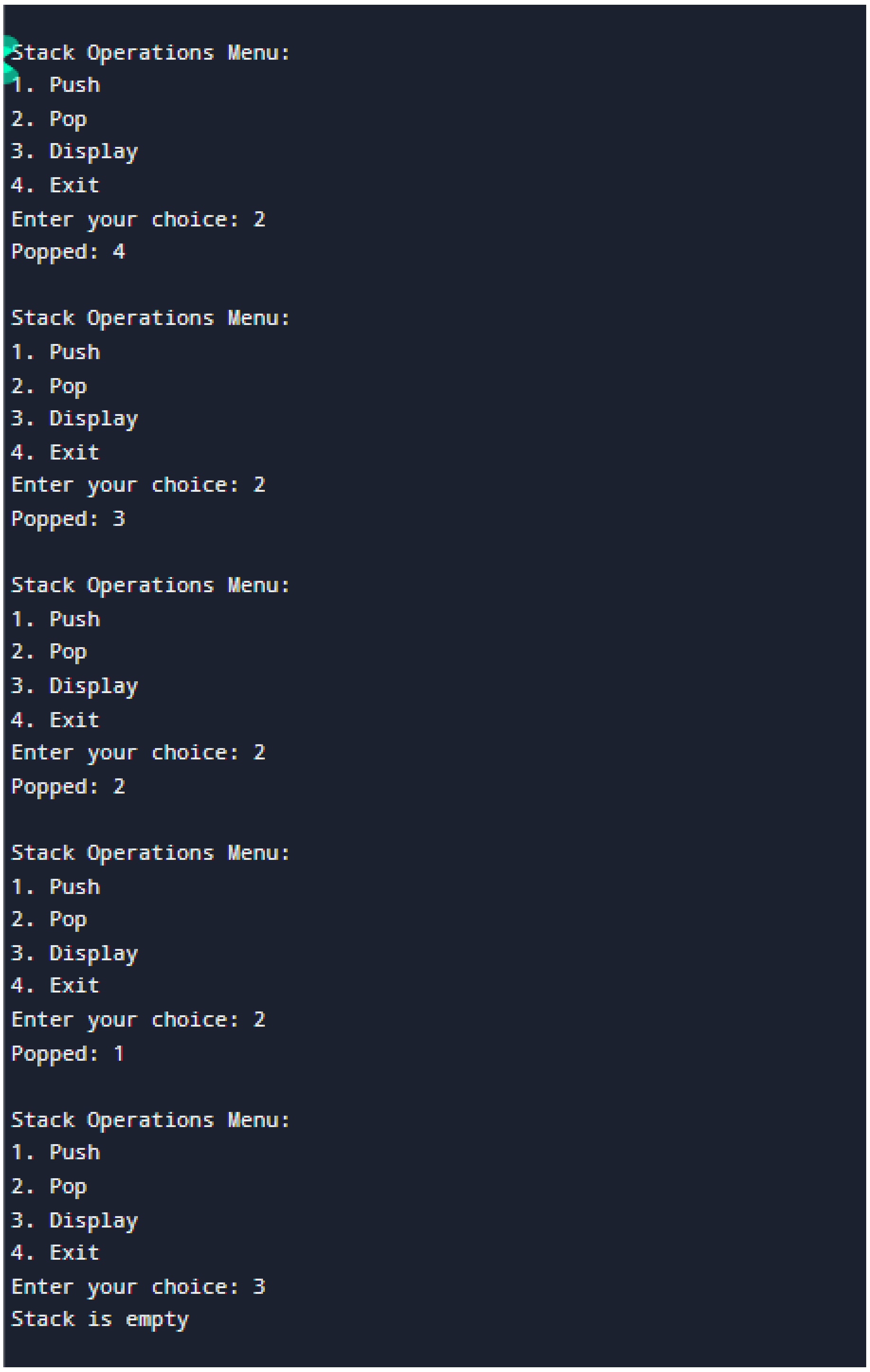
\

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

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

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

// Function to check if the stack is empty
int isEmpty(struct Node* top) {
   return top == NULL;
}

// Function to push an element onto the stack
void push(struct Node** top, int data) {
   struct Node* node = createNode(data);
   node->next = *top;
   *top = node;
   printf("\nPushed %d onto the stack.", data);
}

// Function to pop an element from the stack
int pop(struct Node** top) {
   if (isEmpty(*top)) {
      printf("Stack underflow\n");
      return -1; // Return -1 to indicate the stack is empty
   }
   struct Node* temp = *top;
   int data = temp->data;
   *top = (*top)->next;
   free(temp);
   return data;
}

// Function to display the elements in the stack
void display(struct Node* top) {
   if (isEmpty(top)) {
      printf("Stack is empty\n");
      return;
   }
   struct Node* temp = top;
   printf("\nStack: ");
```

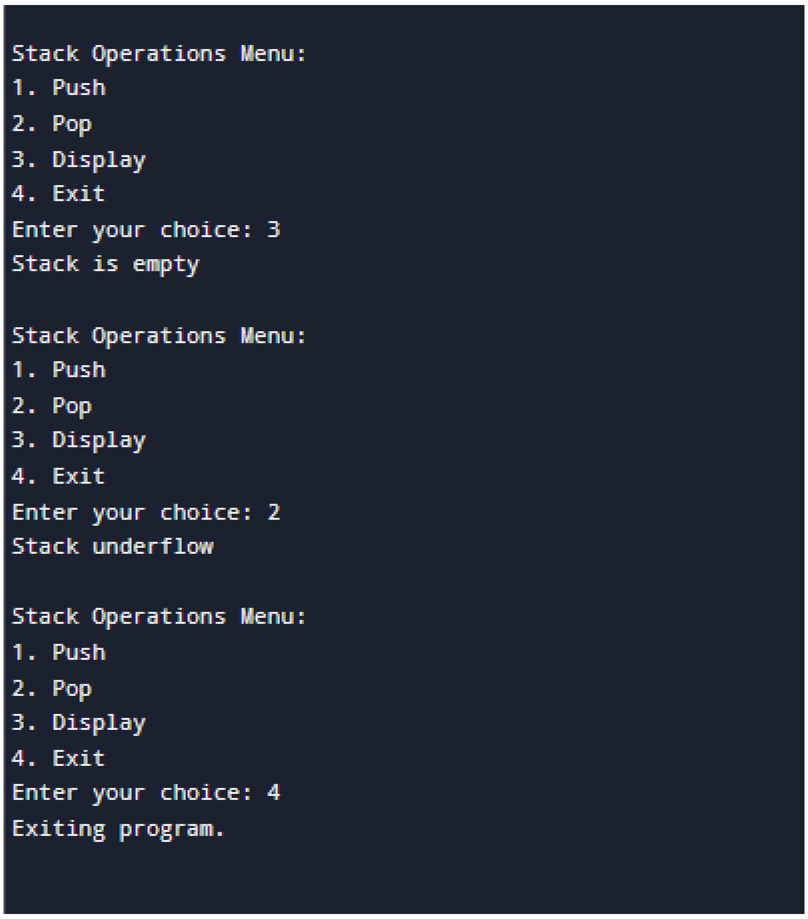
```
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

// Main function with switch-based menu
int main() {
    struct Node* stack = NULL;
    int choice, value;

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

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

    return 0;
}
```



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

Pushed 1 onto the stack.
Stack Operations Menu:
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 1
Enter value to push: 2

Pushed 2 onto the stack.
Stack Operations Menu:
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 1
Enter value to push: 3

Pushed 3 onto the stack.
Stack Operations Menu:
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 1
Enter value to push: 4

Pushed 4 onto the stack.
Stack Operations Menu:
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 3

Stack: 4 3 2 1
```



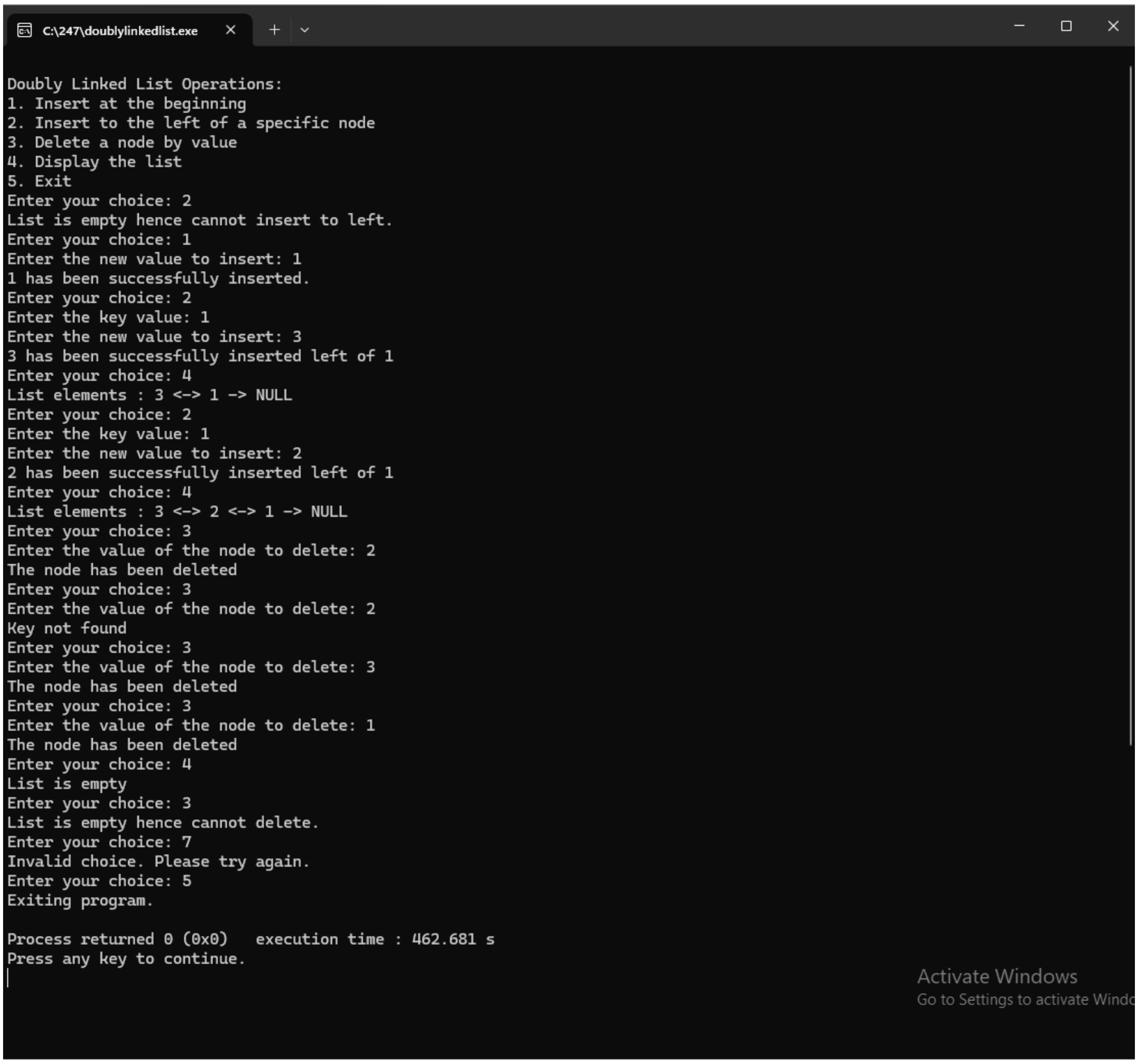
```
Stack Operations Menu:
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 2
Popped: 4

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

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

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

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


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

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

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

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>

typedef struct node {
   struct node* prev;
   int data;
   struct node* next;
}node;


node* createnode(int data){
   node* newnode=(node*)malloc(sizeof(node));
   newnode->prev=NULL;
   newnode->next=NULL;
   newnode->data=data;
   return newnode;
}

struct node* createDoublyLinkedList() {
   return NULL;
}
int isempty(node* head){
   return head==NULL;
}

node* insert_at_beginning(int data, node* head){
   node* newnode=createnode(data);
   newnode->next=head;
   if(head!=NULL)
      head->prev=newnode;
   head=newnode;
   printf("%d has been successfully inserted.\n",data);
   return head;
}

node* insert_to_left(int data, int key, node* head) {
   node* temp = head;

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

   if (temp == NULL) {
      printf("Key not found\n");
```

```
        return head;
    }

    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;

    printf("%d has been successfully inserted left of %d\n", data, key);
    return head;
}


node* deletenode(int key,node* head){
    if(isempty(head)){
        printf("List is empty hence cannot delete a node\n ");
        return head;
    }
    node* temp=head;
    while(temp!=NULL && temp->data!=key){
        temp=temp->next;
    }
    if(temp==NULL){
        printf("Key not found\n");
        return head;
    }
    if (temp->prev != NULL) {
        temp->prev->next = temp->next;
    } else {
        head = temp->next;
    }

    if (temp->next != NULL) {
        temp->next->prev = temp->prev;
    }
    printf("The node has been deleted\n");
    free(temp);
    return head;
}

void display(node* head){
    node* temp=head;
    if(isempty(head)){
        printf("List is empty\n");
        return;
    }
    printf("List elements : ");
    while(temp->next!=NULL){
        printf("%d <-> ",temp->data);
        temp=temp->next;
    }
    printf("%d -> NULL\n",temp->data);
}

int main() {
    node* head = createDoublyLinkedList();
    int choice, value, key;
    printf("\nDoubly Linked List Operations:\n");
    printf("1. Insert at the beginning\n");
    printf("2. Insert to the left of a specific node\n");
    printf("3. Delete a node by value\n");
    printf("4. Display the list\n");
    printf("5. Exit\n");
    while (1) {
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter the new value to insert: ");
                scanf("%d", &value);
                head=insert_at_beginning(value,head);
                break;
            case 2:
                if(!isempty(head)){
                    printf("Enter the key value: ");
                    scanf("%d", &key);
                    printf("Enter the new value to insert: ");
                    scanf("%d", &value);
                    head=insert_to_left(value,key,head);
                }
                else
                    printf("List is empty hence cannot insert to left.\n");
                break;

            case 3:
                if(!isempty(head)){
                    printf("Enter the value of the node to delete: ");
                    scanf("%d", &value);
                    head=deletenode(value,head);
                }
                else
                    printf("List is empty hence cannot delete.\n");
                break;
            case 4:
                display(head);
                break;

            case 5:
                printf("Exiting program.\n");
                exit(0);

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

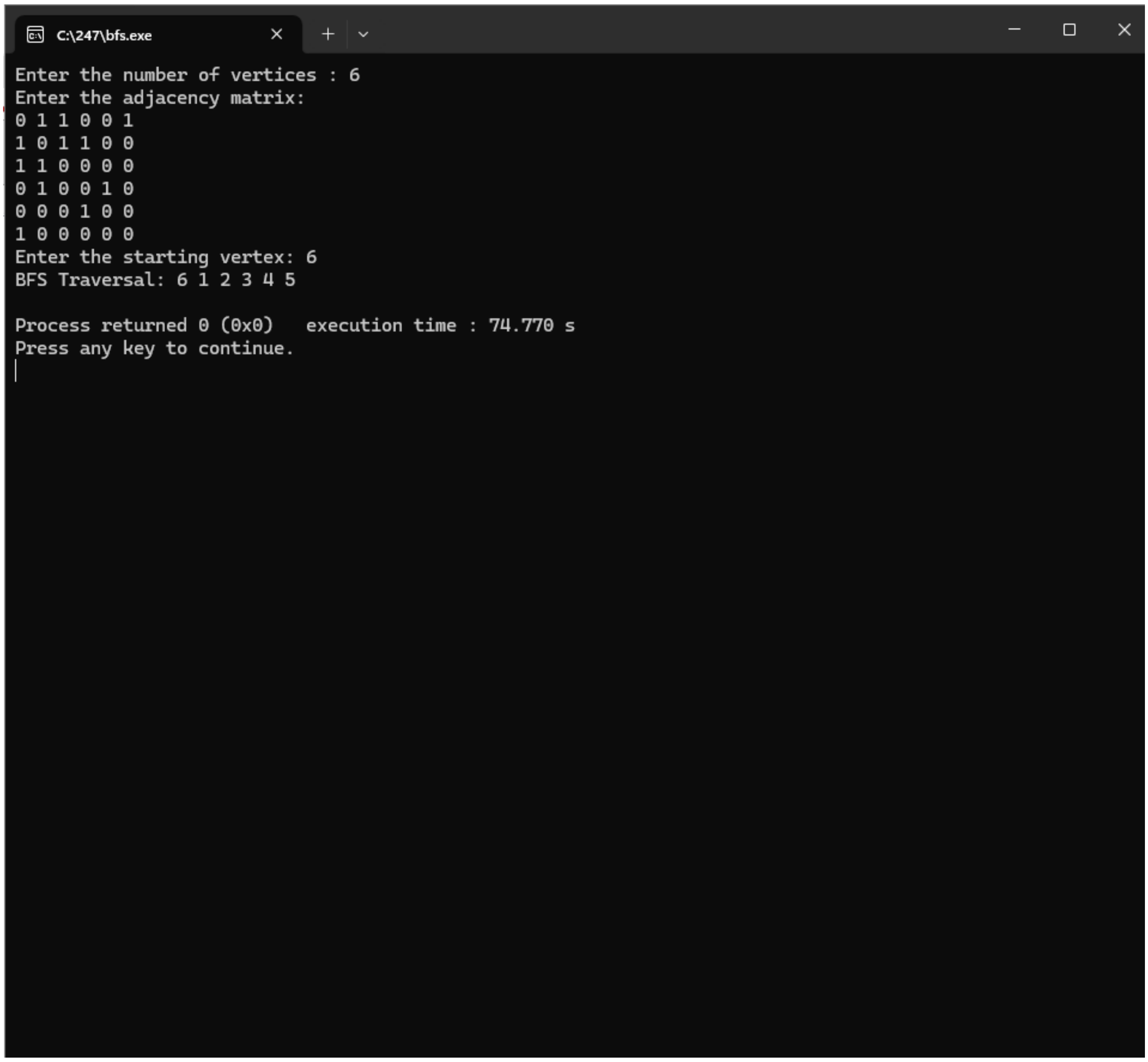


```
Doubly Linked List Operations:
1. Insert at the beginning
2. Insert to the left of a specific node
3. Delete a node by value
4. Display the list
5. Exit
Enter your choice: 2
List is empty hence cannot insert to left.
Enter your choice: 1
Enter the new value to insert: 1
1 has been successfully inserted.
Enter your choice: 2
Enter the key value: 1
Enter the new value to insert: 3
3 has been successfully inserted left of 1
Enter your choice: 4
List elements : 3 <-> 1 -> NULL
Enter your choice: 2
Enter the key value: 1
Enter the new value to insert: 2
2 has been successfully inserted left of 1
Enter your choice: 4
List elements : 3 <-> 2 <-> 1 -> NULL
Enter your choice: 3
Enter the value of the node to delete: 2
The node has been deleted
Enter your choice: 3
Enter the value of the node to delete: 2
Key not found
Enter your choice: 3
Enter the value of the node to delete: 3
The node has been deleted
Enter your choice: 3
Enter the value of the node to delete: 1
The node has been deleted
Enter your choice: 4
List is empty
Enter your choice: 3
List is empty hence cannot delete.
Enter your choice: 7
Invalid choice. Please try again.
Enter your choice: 5
Exiting program.

Process returned 0 (0x0)   execution time : 462.681 s
Press any key to continue.
```

Lab Program 8
Write a program
a) To construct a binary Search tree.
b) To traverse the tree using all the methods i.e., inorder, 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 data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->left = newNode->right = NULL;
    return newNode;
}

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

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

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

```
void postorderTraversal(struct Node* root) {
   if (root == NULL) {
      return;
   }
   postorderTraversal(root->left);
   postorderTraversal(root->right);
   printf("%d ", root->data);
}

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

   while (1) {
      printf("\nBinary Search Tree Operations:\n");
      printf("1. Insert a node\n");
      printf("2. In-order traversal\n");
      printf("3. Pre-order traversal\n");
      printf("4. Post-order traversal\n");
      printf("5. Exit\n");
      printf("Enter your choice: ");
      scanf("%d", &choice);

      switch (choice) {
         case 1:
            printf("Enter the value to insert: ");
            scanf("%d", &data);
            root = insert(root, data);
            printf("Node %d inserted.\n", data);
            break;

         case 2:
            printf("In-order traversal: ");
            inorderTraversal(root);
            printf("\n");
            break;

         case 3:
            printf("Pre-order traversal: ");
            preorderTraversal(root);
            printf("\n");
            break;

         case 4:
            printf("Post-order traversal: ");
            postorderTraversal(root);
            printf("\n");
            break;

         case 5:
                printf("Exiting...\n");
                exit(0);

            default:
                printf("Invalid choice, please try again.\n");
        }
    }

    return 0;
}
```



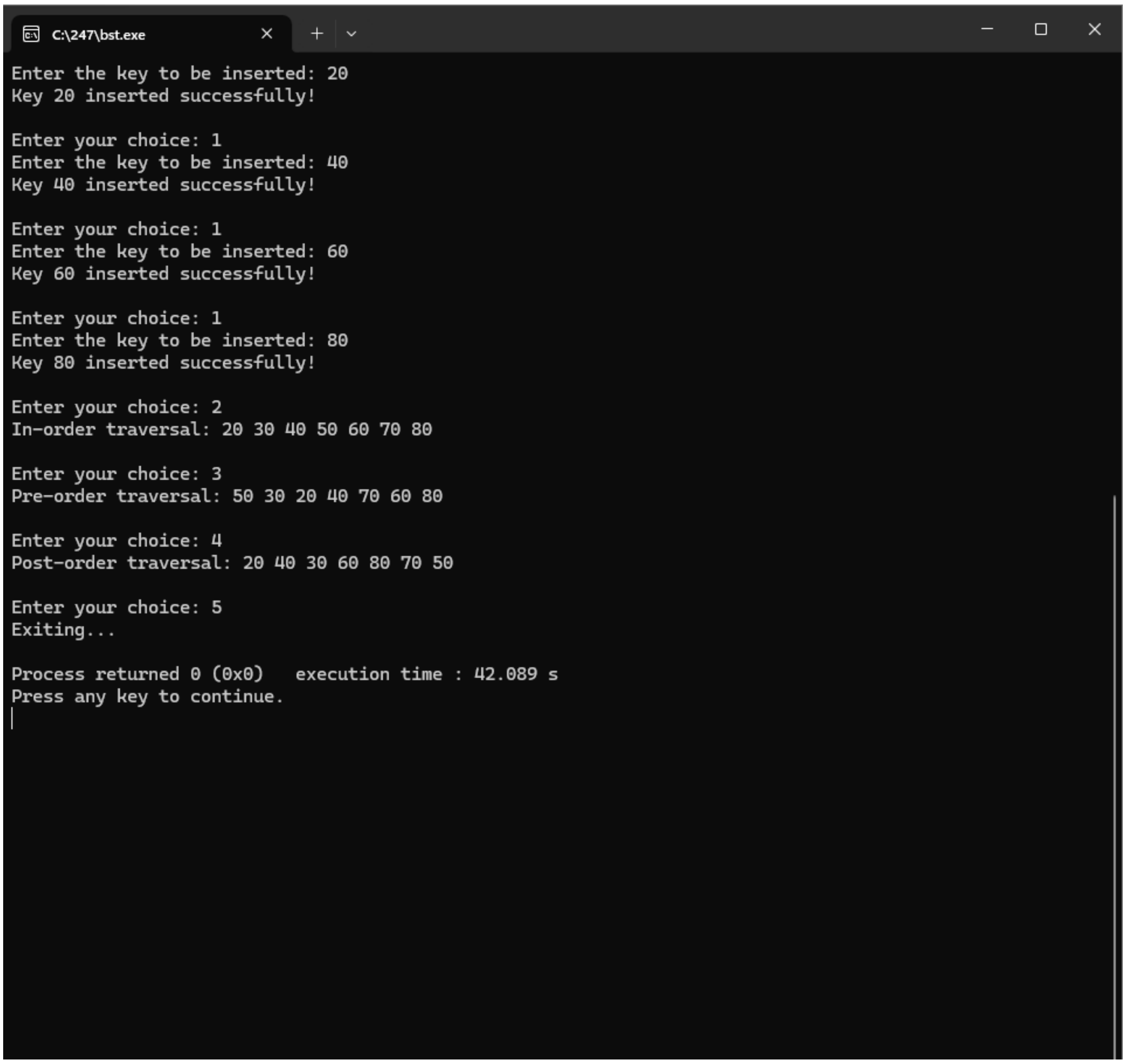
```
Enter the number of vertices : 6
Enter the adjacency matrix:
0 1 1 0 0 1
1 0 1 1 0 0
1 1 0 0 0 0
0 1 0 0 1 0
0 0 0 1 0 0
1 0 0 0 0 0
Enter the starting vertex: 6
BFS Traversal: 6 1 2 3 4 5

Process returned 0 (0x0)   execution time : 74.770 s
Press any key to continue.
```

Lab Program 9

Write a program to traverse a graph using BFS method

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

#define MAX 100

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

struct Queue* createQueue() {
   struct Queue* q = (struct Queue*)malloc(sizeof(struct Queue));
   q->front = -1;
   q->rear = -1;
   return q;
}

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

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

int dequeue(struct Queue* q) {
   int item;
   if (isEmpty(q)) {
      printf("Queue is empty\n");
      return -1;
   } else {
      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 startVertex, int n) {
   int visited[MAX] = {0};
   struct Queue* q = createQueue();

   visited[startVertex] = 1;
   enqueue(q, startVertex);

   printf("BFS Traversal: ");

   while (!isEmpty(q)) {
      int currentVertex = dequeue(q);
      printf("%d ", currentVertex);

      for (int i = 1; i <= n; i++) {
         if (graph[currentVertex][i] == 1 && !visited[i]) {
            visited[i] = 1;
            enqueue(q, i);
         }
      }
   }

   printf("\n");
}

int main() {
   int n, startVertex;
   int graph[MAX][MAX];

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

   printf("Enter the adjacency matrix:\n");
   for (int i = 1; i <= n; i++) {
      for (int j = 1; j <= n; j++) {
         scanf("%d", &graph[i][j]);
      }
   }

   printf("Enter the starting vertex: ");
   scanf("%d", &startVertex);

   bfs(graph, startVertex, n);

   return 0;
}
```

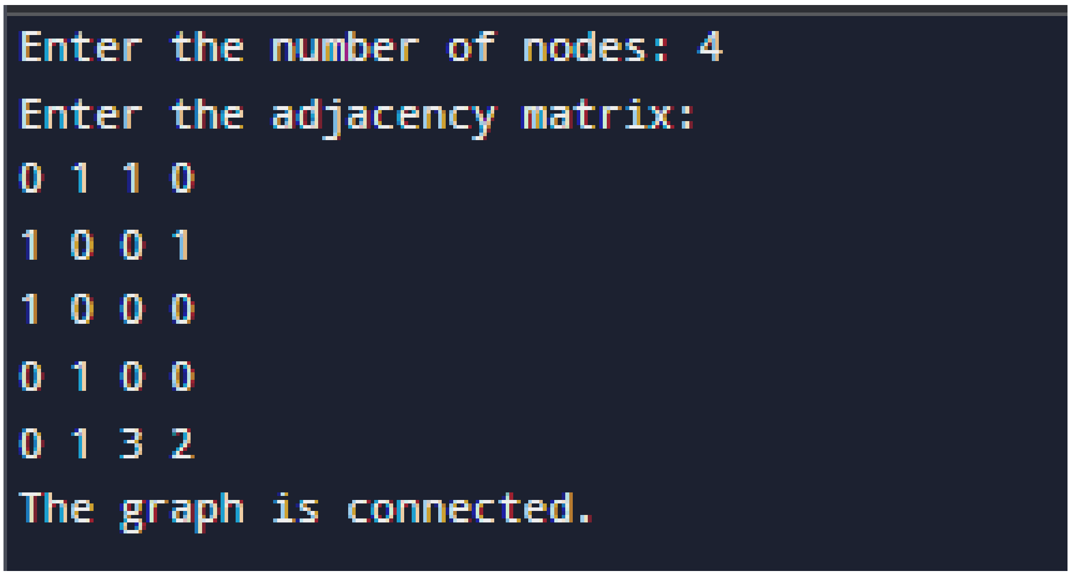


```
Enter the key to be inserted: 20
Key 20 inserted successfully!

Enter your choice: 1
Enter the key to be inserted: 40
Key 40 inserted successfully!

Enter your choice: 1
Enter the key to be inserted: 60
Key 60 inserted successfully!

Enter your choice: 1
Enter the key to be inserted: 80
Key 80 inserted successfully!

Enter your choice: 2
In-order traversal: 20 30 40 50 60 70 80

Enter your choice: 3
Pre-order traversal: 50 30 20 40 70 60 80

Enter your choice: 4
Post-order traversal: 20 40 30 60 80 70 50

Enter your choice: 5
Exiting...

Process returned 0 (0x0)   execution time : 42.089 s
Press any key to continue.
```

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

```
#include <stdio.h>

#define MAX_NODES 100

int adjacencyMatrix[MAX_NODES][MAX_NODES];
int visited[MAX_NODES];
int nodes;

// Function for DFS
void DFS(int vertex) {
   visited[vertex] = 1;
   printf("%d ", vertex);  // Print visited node

   for (int i = 0; i < nodes; i++) {
      if (adjacencyMatrix[vertex][i] == 1 && !visited[i]) {
         DFS(i);
      }
   }
}

// Function to check if the graph is connected
int isConnected() {
   // Initialize visited array to 0
   for (int i = 0; i < nodes; i++) {
      visited[i] = 0;
   }

   // Start DFS from node 0
   DFS(0);

   // Check if all nodes are visited
   for (int i = 0; i < nodes; i++) {
      if (!visited[i]) {
         return 0;  // Graph is not connected
      }
   }
   return 1;  // Graph is connected
}

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

   printf("Enter the adjacency matrix:\n");
   for (int i = 0; i < nodes; i++) {
      for (int j = 0; j < nodes; j++) {
         scanf("%d", &adjacencyMatrix[i][j]);
      }
```

```
    }

    // Check connectivity
    if (isConnected()) {
        printf("\nThe graph is connected.\n");
    } else {
        printf("\nThe graph is not connected.\n");
    }

    return 0;
}
```



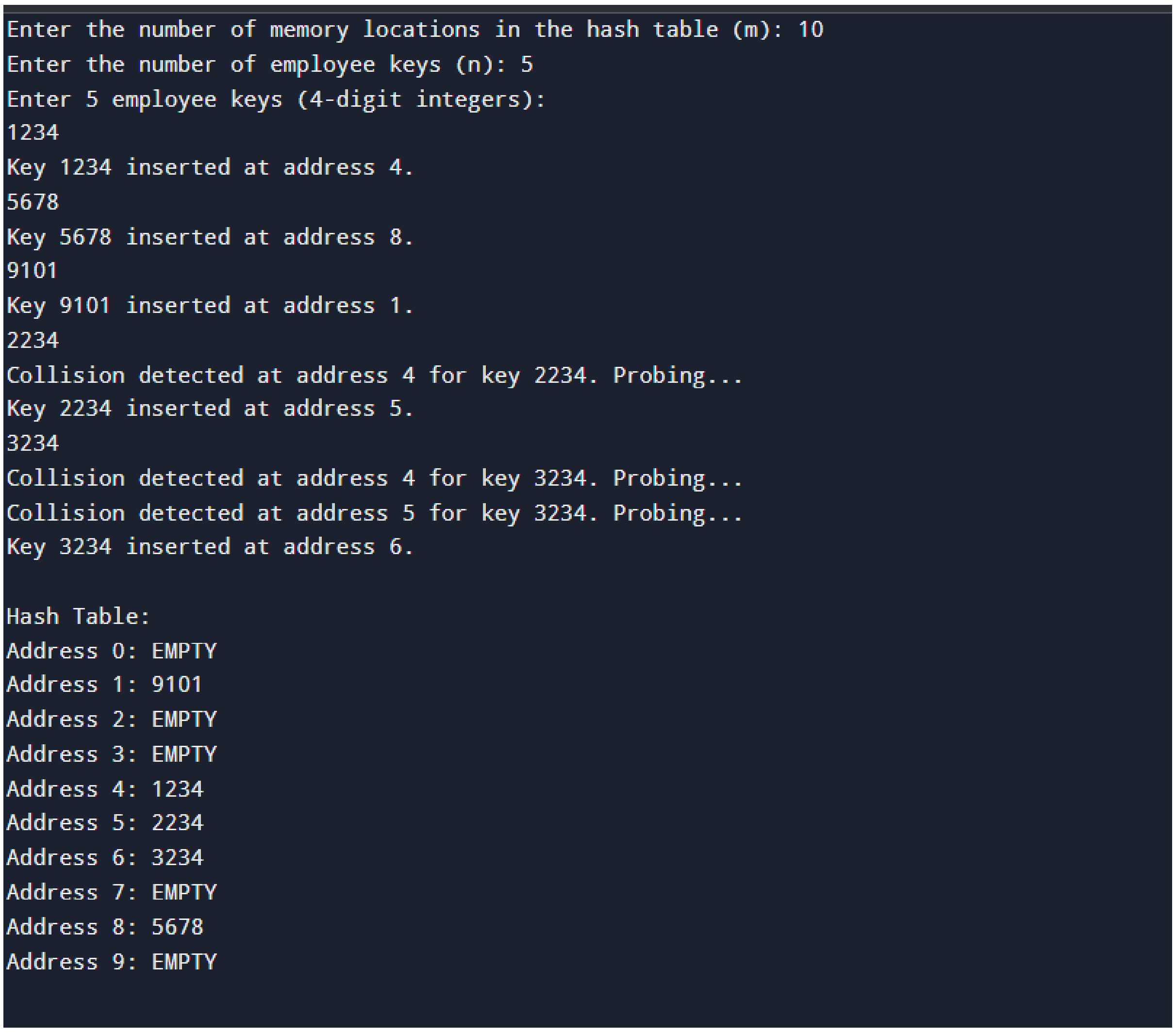
```
Enter the number of nodes: 4
Enter the adjacency matrix:
0 1 1 0
1 0 0 1
1 0 0 0
0 1 0 0
0 1 3 2
The graph is connected.
```

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

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

#define MAX_SIZE 100  // Maximum size of the hash table

int hashTable[MAX_SIZE];  // Hash table to store employee keys
int m;  // Number of memory locations in the hash table

// Function to initialize the hash table
void initializeHashTable() {
   for (int i = 0; i < m; i++) {
      hashTable[i] = -1;  // Initialize all locations as empty (-1 indicates empty)
   }
}

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

// Function to insert a key into the hash table using linear probing
void insertKey(int key) {
   int address = hashFunction(key);  // Compute the initial hash address

   // Linear probing to resolve collision
   int originalAddress = address;
   while (hashTable[address] != -1) {  // While the location is occupied
      printf("Collision detected at address %d for key %d. Probing...\n", address, key);
      address = (address + 1) % m;  // Move to the next location
      if (address == originalAddress) {  // If we come back to the start, table is full
         printf("Hash table is full! Cannot insert key %d.\n", key);
         return;
      }
   }
   hashTable[address] = key;  // Place the key at the resolved address
   printf("Key %d inserted at address %d.\n", key, address);
}

// Function to display the hash table
void displayHashTable() {
   printf("\nHash Table:\n");
   for (int i = 0; i < m; i++) {
      if (hashTable[i] == -1)
```

```
            printf("Address %d: EMPTY\n", i);
        else
            printf("Address %d: %d\n", i, hashTable[i]);
    }
}

int main() {
    int n;  // Number of employee keys
    int key;

    printf("Enter the number of memory locations in the hash table (m): ");
    scanf("%d", &m);
    if (m > MAX_SIZE) {
        printf("Error: m exceeds the maximum size %d.\n", MAX_SIZE);
        return 1;
    }

    initializeHashTable();  // Initialize the hash table

    printf("Enter the number of employee keys (n): ");
    scanf("%d", &n);

    printf("Enter %d employee keys (4-digit integers):\n", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &key);
        insertKey(key);  // Insert each key into the hash table
    }

    displayHashTable();  // Display the final hash table

    return 0;
}
```



```
Enter the number of memory locations in the hash table (m): 10
Enter the number of employee keys (n): 5
Enter 5 employee keys (4-digit integers):
1234
Key 1234 inserted at address 4.
5678
Key 5678 inserted at address 8.
9101
Key 9101 inserted at address 1.
2234
Collision detected at address 4 for key 2234. Probing...
Key 2234 inserted at address 5.
3234
Collision detected at address 4 for key 3234. Probing...
Collision detected at address 5 for key 3234. Probing...
Key 3234 inserted at address 6.

Hash Table:
Address 0: EMPTY
Address 1: 9101
Address 2: EMPTY
Address 3: EMPTY
Address 4: 1234
Address 5: 2234
Address 6: 3234
Address 7: EMPTY
Address 8: 5678
Address 9: EMPTY
```

Leet Code Programs:
Move_Zeroes

```
void moveZeroes(int* nums, int numsSize) {
 int index = 0;
   for (int i = 0; i < numsSize; i++) {
       if (nums[i] != 0) {
           nums[index] = nums[i];
           index++;
       }
   }
   while (index < numsSize) {
       nums[index] = 0;
       index++;
   }
}
```

leetcode_majority_ele

```
int majorityElement(int* nums, int numsSize) {
   int ele=0;
   int c=0;
   for(int i=0;i<numsSize;i++)
   {
      if(c==0)
      {
         ele=nums[i];
         c++;
      }
      else if(nums[i]==ele)
         c++;
      else
         c--;
   }
   return ele;
}
```

Game of two stacks

```
int twoStacks(int maxSum, int a_count, int* a, int b_count, int* b) {
   int sum_a = 0, score_a = 0;
   int sum_b = 0, score_b = 0;
   int max_score = 0;

   for (int i = 0; i < a_count; i++) {
      if (sum_a + a[i] > maxSum) break;
      sum_a += a[i];
      score_a++;
   }

   max_score = score_a;
   for (int j = 0; j < b_count; j++) {
      sum_b += b[j];
      score_b++;

      while (sum_a + sum_b > maxSum && score_a > 0) {
         sum_a -= a[score_a - 1];
         score_a--;
      }

      if (sum_a + sum_b <= maxSum) {
         int current_score = score_a + score_b;
         if (current_score > max_score) {
            max_score = current_score;
         }
      } else {
         break;
      }
   }

   return max_score;
}
```

leetcode-234_palindrome

```
bool isPalindrome(struct ListNode* head) {
   struct ListNode* temp=head;
   if(temp->next==NULL) return true;

   struct ListNode *slow = head, *fast = head;
   while (fast->next && fast->next->next) {
      slow = slow->next;
      fast = fast->next->next;
   }

   temp=slow;
   struct ListNode* prev = NULL;
   struct ListNode* curr = temp;
   struct ListNode* next;

   while (curr) {
      next = curr->next;
      curr->next = prev;
      prev = curr;
      curr = next;
   }
   temp=prev;
   struct ListNode* n1=head;
   struct ListNode* n2=temp;
   for(;n1&&n2;n1=n1->next,n2=n2->next)
   {
      if(n1->val!=n2->val)return false;
   }
   return true;
}
```

Path Sum -112

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     struct TreeNode *left;
 *     struct TreeNode *right;
 * };
 */

bool hasPathSum(struct TreeNode* root, int targetSum) {
   if(root==NULL)
      return false;

   if(root->val==targetSum && root->left==NULL && root->right==NULL)
      return true;

   else{
      return(hasPathSum(root->left,targetSum-root->val)  || hasPathSum(root->right,targetSum-root-
>val));
   }
}
```