# VISVESVARAYA TECHNOLOGICAL UNIVERSITY "JnanaSangama", Belgaum -590014, Karnataka.

**LAB REPORT**
**On**

**DATA STRUCTURES (23CS3PCDST)**

**Submitted by**

**DHANUSH S(1BM23CS089)**

**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 **Dhanush S (1BM23CS089)**, 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. Selva kumar** Assistant Professor Professor and Head Department of CSE Department of CSE BMSCE, Bengaluru BMSCE, Bengaluru

## Index Sheet

| | | |
|---|---|---|
| 9 | 9a. Write a program to traverse a graph using the BFS method<br>9b. Write a program to check whether a given graph is connected or not using the DFS method. | 58-70 |
| 10 | Hashing Method Linear Probing | 70-73 |

**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>

#define MAX 100


void push(int *stack, int *top, int value) {

    if (*top == MAX - 1) {

        printf("Stack Overflow\n");

        return;

    }

    stack[++(*top)] = value;

}

void pop(int *stack, int *top) {

    if (*top == -1) {
```

```
        printf("Stack Underflow\n");
        return;
    }
    printf("Popped element: %d\n", stack[(*top)--]);
}
void display(int *stack, int *top) {
    if (*top == -1) {
        printf("Stack is empty\n");
        return;
    }
    for (int i = *top; i >= 0; i--) {
        printf("%d\n", stack[i]);
    }
}
int main() {
    int stack[MAX], top = -1, choice, value;
    do {
        printf("1. Push\n2. Pop\n3. Display\n4. Exit\nEnter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Enter value to push: ");
                scanf("%d", &value);
                push(stack, &top, value);
                break;
            case 2:
                pop(stack, &top);
                break;
            case 3:
                display(stack, &top);
                break;
```

```
            case 4:
                break;
            default:
                printf("Invalid choice\n");
        }
    } while (choice != 4);
    return 0;
}
```

**Output**

```
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 1
Enter value to push: 56
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 1
Enter value to push: 78
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 3
78
56
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 2
Popped element: 78
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 3
56
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 4


=== Code Execution Successful ===
```

**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 <ctype.h>
#define MAX 100

void push(char *stack, int *top, char value) {
    if (*top == MAX - 1) {
        printf("Stack Overflow\n");
        return;
    }
    stack[++(*top)] = value;
}

char pop(char *stack, int *top) {
    if (*top == -1) {
        printf("Stack Underflow\n");
        return '\0';
    }
    return stack[(*top)--];
}

int precedence(char op) {
    if (op == '+' || op == '-') return 1;
    if (op == '*' || op == '/') return 2;
    if (op == '^') return 3;
```

```
    return 0;
}

int isRightAssociative(char op) {
    if (op == '^') return 1;
    return 0;
}

void infixToPostfix(char *infix, char *postfix) {
    char stack[MAX];
    int top = -1, k = 0;

    for (int i = 0; infix[i] != '\0'; i++) {
        if (isalnum(infix[i])) {
            postfix[k++] = infix[i];
        } else if (infix[i] == '(') {
            push(stack, &top, infix[i]);
        } else if (infix[i] == ')') {
            while (top != -1 && stack[top] != '(') {
                postfix[k++] = pop(stack, &top);
            }
            pop(stack, &top);
        } else {
            while (top != -1 &&
                   ((isRightAssociative(infix[i]) && precedence(stack[top]) > precedence(infix[i]))
||
                   (!isRightAssociative(infix[i]) && precedence(stack[top]) >=
precedence(infix[i])))) {
                postfix[k++] = pop(stack, &top);
            }
push(stack, &top, infix[i]);
```

```
}
}

while (top != -1) {
postfix[k++] = pop(stack, &top);
}

postfix[k] = '\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: A+(B*C-(D/E^F)*G)*H
Postfix expression: ABC*DEF^/G*-H*+
```

**Lab Program 3:**

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

```
#include <stdio.h>
#define MAX 100

void insert(int *queue, int *front, int *rear, int value) {
    if ((*rear + 1) % MAX == *front) {
        printf("Queue Overflow\n");
        return;
    }
    if (*front == -1) {
        *front = 0;
    }
    *rear = (*rear + 1) % MAX;
    queue[*rear] = value;
}

int delete(int *queue, int *front, int *rear) {
    if (*front == -1) {
        printf("Queue Empty\n");
        return -1;
    }
    int value = queue[*front];
    if (*front == *rear) {
        *front = *rear = -1;
    } else {


        *front = (*front + 1) % MAX;
    }
    return value;
}
```

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

int main() {
    int queue[MAX], front = -1, rear = -
    1; int choice, value;

    do {
        printf("\nMenu:\n");
        printf("1. Insert\n");
        printf("2. Delete\n");
        printf("3. Display\n");


        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
```

```
        case 1:
            printf("Enter value to insert: ");
            scanf("%d", &value);
            insert(queue, &front, &rear, value);
            break;
        case 2:
            value = delete(queue, &front,
            &rear); if (value != -1) {
                printf("Deleted value: %d\n",
            value); }
            break;
        case 3:
            display(queue, front, rear);
            break;
        case 4:
            printf("Exiting...\n");
            break;
        default:
            printf("Invalid choice!\n");
    }
  } while (choice != 4);

  return 0;
}
```

**Output**

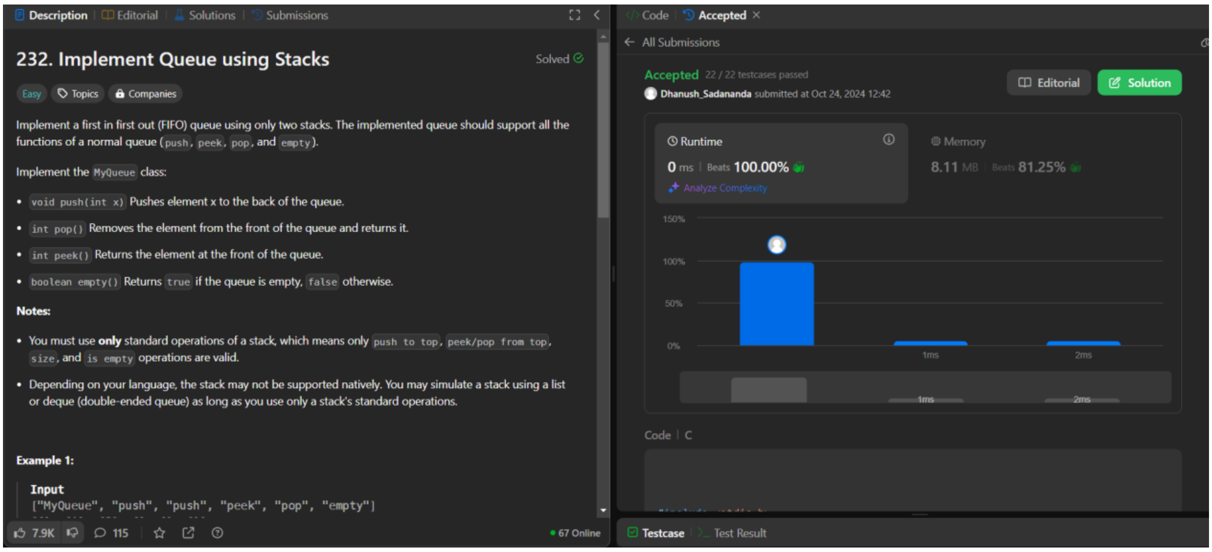
```
Menu:
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 1
Enter value to insert: 45

Menu:
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 1
Enter value to insert: 78

Menu:
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 3
Queue elements: 45 78

Menu:
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 2
Deleted value: 45

Menu:
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 3
Queue elements: 78
```

**LeetCode 1:**

**Implement Queue using Stacks.**

```
class MyQueue:
    def __init__(self):
        self.stack_1=[]
        self.stack_2=[]

    def push(self, x: int) -> None:
        self.stack_1.append(x)

    def pop(self) -> int:
        if len(self.stack_1)==0:
            return None
        for i in range(len(self.stack_1)):
            self.stack_2.append(self.stack_1.pop(
        )) temp=(self.stack_2.pop())
        for i in range(len(self.stack_2)):
            self.stack_1.append(self.stack_2.pop(
        )) return temp

    def peek(self) -> int:
        return self.stack_1[0]

    def empty(self) -> bool:
        return len(self.stack_1)==0
```

**Output**




**4. 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 MAX 100

void insert(int *queue, int *front, int *rear, int value) {
    if ((*rear + 1) % MAX == *front) {
        printf("Queue Overflow\n");
        return;
    }
    if (*front == -1) {
        *front = 0;
    }
    *rear = (*rear + 1) % MAX;
    queue[*rear] = value;
}

int delete(int *queue, int *front, int *rear) {
```

```
    if (*front == -1) {
        printf("Queue Empty\n");
        return -1;
    }
    int value = queue[*front];
    if (*front == *rear) {
        *front = *rear = -1;
    } else {
        *front = (*front + 1) % MAX;

    }
    return value;
}

void display(int *queue, int front, int rear) {
    if (front == -1) {
        printf("Queue Empty\n");
        return;
    }
    printf("Queue elements: ");
    int i = front;
    while (1) {
        printf("%d ", queue[i]);
        if (i == rear) break;
        i = (i + 1) % MAX;
    }
    printf("\n");
}

int main() {
    int queue[MAX], front = -1, rear = -1;
```

```
int choice, value;

do {
    printf("\nMenu:\n");
    printf("1. Insert\n");
    printf("2. Delete\n");
    printf("3. Display\n");
    printf("4. Exit\n");

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

    switch (choice) {
        case 1:
            printf("Enter value to insert: ");
            scanf("%d", &value);
            insert(queue, &front, &rear, value);
            break;
        case 2:
            value = delete(queue, &front, &rear);
            if (value != -1) {
                printf("Deleted value: %d\n", value);
            }
            break;
        case 3:
            display(queue, front, rear);
            break;
        case 4:
            printf("Exiting...\n");
            break;
```

```
            default:
                printf("Invalid choice!\n");
        }
    } while (choice != 4);

    return 0;
}
```

**Output**

```
Menu:
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 1
Enter value to insert: 45

Menu:
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 1
Enter value to insert: 78

Menu:
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 3
Queue elements: 45 78

Menu:
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 2
Deleted value: 45

Menu:
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 3
Queue elements: 78
```

**5.Linked List (Create, Insert, Delete, Display).**

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

typedef struct Node {
    int value;
    struct Node* next;
} Node;

typedef struct LinkedList {
    Node* head;
    Node* tail;
    int length;
} LinkedList;

Node* create_node(int value) {
    Node* new_node = (Node*)malloc(sizeof(Node));
    new_node->value = value;
    new_node->next = NULL;
    return new_node;
}

LinkedList* create_linked_list() {
    LinkedList* list = (LinkedList*)malloc(sizeof(LinkedList));
    list->head = NULL;
    list->tail = NULL;
    list->length = 0;
    return list;
}

void print_list(LinkedList* list) {
    Node* temp = list->head;
```

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

void append(LinkedList* list, int value) { Node*
    new_node = create_node(value); if (list->head ==
    NULL && list->tail == NULL) { list->head =
    new_node;
        list->tail = new_node;
    } else {
        list->tail->next = new_node;
        list->tail = new_node;
    }
    list->length++;
}

Node* pop(LinkedList* list) {
    if (list->length == 0) return NULL;

    Node* temp = list->head;
    Node* pre = list->head;


    while (temp->next != NULL) {
        pre = temp;
        temp = temp->next;
    }
    list->tail = pre;
```

```
    pre->next = NULL;
    list->length--;

    if (list->length == 0) {
        list->head = NULL;
        list->tail = NULL;
    }
    return temp;
}

void prepend(LinkedList* list, int value) {
    Node* new_node = create_node(value);
    if (list->length == 0) {
        list->head = new_node;
        list->tail = new_node;
    } else {
        new_node->next = list->head;
        list->head = new_node;
    }
    list->length++;
}

Node* pop_first(LinkedList* list) { if
    (list->length == 0) return NULL;

    Node* temp = list->head;
    list->head = temp->next;
    temp->next = NULL;
    list->length--;
```

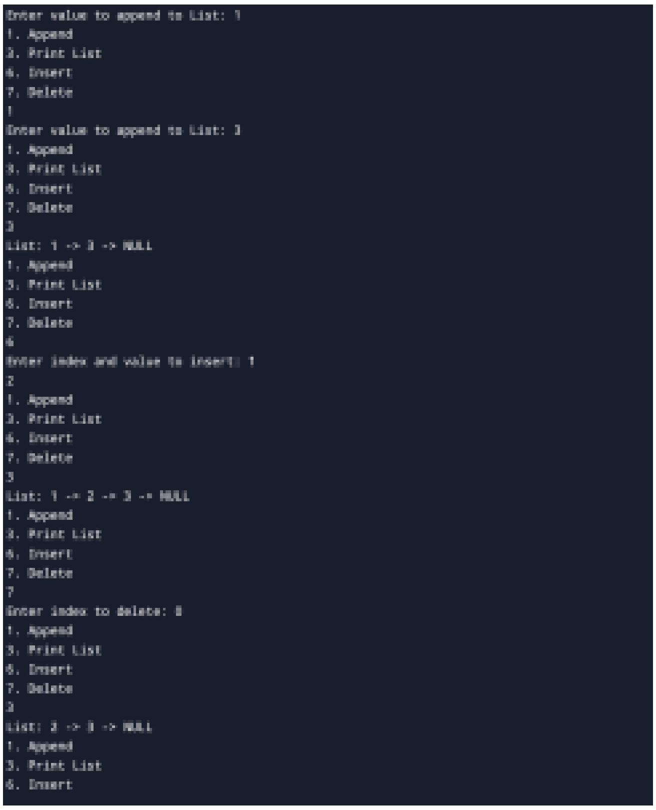
```
    if (list->length == 0) list->tail = NULL;

    return temp;
}

Node* get(LinkedList* list, int index) {
    if (index < 0 || index >= list->length) return NULL;

    Node* temp = list->head;
    for (int i = 0; i < index; i++) {
        temp = temp->next;
    }
    return temp;
}

int set_value(LinkedList* list, int index, int value) {
    Node* temp = get(list, index);
    if (temp) {
        temp->value = value;
        return 1;
    }
    return 0;
}

int insert(LinkedList* list, int index, int value) {
    if (index < 0 || index > list->length) return 0;
    if (index == 0) {
        prepend(list, value);
        return 1;
    }
```

```
    if (index == list->length) {
        append(list, value);
        return 1;
    }

    Node* new_node = create_node(value);
    Node* prev = get(list, index - 1);
    new_node->next = prev->next;
    prev->next = new_node;
    list->length++;
    return 1;
}

Node* remove_node(LinkedList* list, int index) { if
    (index < 0 || index >= list->length) return NULL; if
    (index == 0) return pop_first(list);
    if (index == list->length - 1) return pop(list);

    Node* prev = get(list, index - 1);
    Node* temp = prev->next;
    prev->next = temp->next;
    temp->next = NULL;
    list->length--;

    return temp;
}

int main() {
    LinkedList* list1 = create_linked_list();
    int choice, value, index;
```

```
do {
    printf("1. Append\n");
    printf("2. Pop\n");
    printf("3. Print List\n");
    printf("4. Prepend\n");
    printf("5. Pop_first\n");
    printf("6. Insert\n");
    printf("7. Delete\n");
    printf("8. Exit\n");

    scanf("%d", &choice);

    switch (choice) {
        case 1:
            printf("Enter value to append to List: ");
            scanf("%d", &value);
            append(list1, value);
            break;
        case 2:
            pop(list1);
            break;


        case 3:
            printf("List: ");
            print_list(list1);
            break;
        case 4:
            printf("Enter value to prepend to List: ");
            scanf("%d", &value);
            prepend(list1, value);
```

```
            break;
        case 5:
            pop_first(list1);
            break;
        case 6:
            printf("Enter index and value to insert: ");
            scanf("%d %d", &index, &value);
            insert(list1, index, value);
            break;
        case 7:
            printf("Enter index to delete: ");
            scanf("%d", &index);
            remove_node(list1, index);
            break;
        case 8:
            printf("Exiting...\n");
            break;
        default:
            printf("Invalid choice, please try again.\n");
        }
    } while (choice != 8);
while (list1->length > 0) {
Node* temp = pop(list1);
free(temp);
}
free(list1);

return 0;
}
```

**Output**



```
Enter value to append to List: 1
1. Append
3. Print List
6. Insert
7. Delete
1
Enter value to append to List: 3
1. Append
3. Print List
6. Insert
7. Delete
3
List: 1 -> 3 -> NULL
1. Append
3. Print List
6. Insert
7. Delete
6
Enter index and value to insert: 1
2
1. Append
3. Print List
6. Insert
7. Delete
3
List: 1 -> 2 -> 3 -> NULL
1. Append
3. Print List
6. Insert
7. Delete
7
Enter index to delete: 0
1. Append
3. Print List
6. Insert
7. Delete
3
List: 2 -> 3 -> NULL
1. Append
3. Print List
6. Insert
```

**Lab Program-6:**

**WAP to Implement Singly Linked List with following operations**

**a) Create a linked list.**

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

**c) Display the contents of the linked list.**

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

typedef struct Node {
    int value;
    struct Node* next;
} Node;
```

```c
typedef struct LinkedList {
    Node* head;
    Node* tail;
    int length;
} LinkedList;

Node* create_node(int value) {
    Node* new_node = (Node*)malloc(sizeof(Node));
    new_node->value = value;
    new_node->next = NULL;
    return new_node;
}

LinkedList* create_linked_list() {
    LinkedList* list = (LinkedList*)malloc(sizeof(LinkedList));


    list->head = NULL;
    list->tail = NULL;
    list->length = 0;
    return list;
}

void print_list(LinkedList* list) {
    Node* temp = list->head;
    while (temp != NULL) {
        printf("%d -> ", temp->value);
        temp = temp->next;
    }
    printf("NULL\n");
}
```

```
void append(LinkedList* list, int value) { Node*
   new_node = create_node(value); if (list->head ==
   NULL && list->tail == NULL) { list->head =
   new_node;
      list->tail = new_node;
    } else {
      list->tail->next = new_node;
      list->tail = new_node;
    }
   list->length++;
}

Node* pop(LinkedList* list) {
   if (list->length == 0) return NULL;

Node* temp = list->head;
   Node* pre = list->head;
   while (temp->next != NULL) {
      pre = temp;
      temp = temp->next;
    }
   list->tail = pre;
   pre->next = NULL;
   list->length--;

   if (list->length == 0) {
      list->head = NULL;
      list->tail = NULL;
    }
   return temp;
}
```

```
void prepend(LinkedList* list, int value) {
    Node* new_node = create_node(value);
    if (list->length == 0) {
        list->head = new_node;
        list->tail = new_node;
    } else {
        new_node->next = list->head;
        list->head = new_node;
    }
    list->length++;
}


Node* pop_first(LinkedList* list) {
    if (list->length == 0) return NULL;

    Node* temp = list->head;
    list->head = temp->next;
    temp->next = NULL;
    list->length--;

    if (list->length == 0) list->tail = NULL;

    return temp;
}

Node* get(LinkedList* list, int index) {
    if (index < 0 || index >= list->length) return NULL;

    Node* temp = list->head;
    for (int i = 0; i < index; i++) {
```

```
        temp = temp->next;
    }
    return temp;
}

int set_value(LinkedList* list, int index, int value) {
    Node* temp = get(list, index);
    if (temp) {
        temp->value = value;
        return 1;


    }
    return 0;
}

int insert(LinkedList* list, int index, int value) {
    if (index < 0 || index > list->length) return 0;
    if (index == 0) {
        prepend(list, value);
        return 1;
    }
    if (index == list->length) {
        append(list, value);
        return 1;
    }

    Node* new_node = create_node(value);
    Node* prev = get(list, index - 1);
    new_node->next = prev->next;
    prev->next = new_node;
```

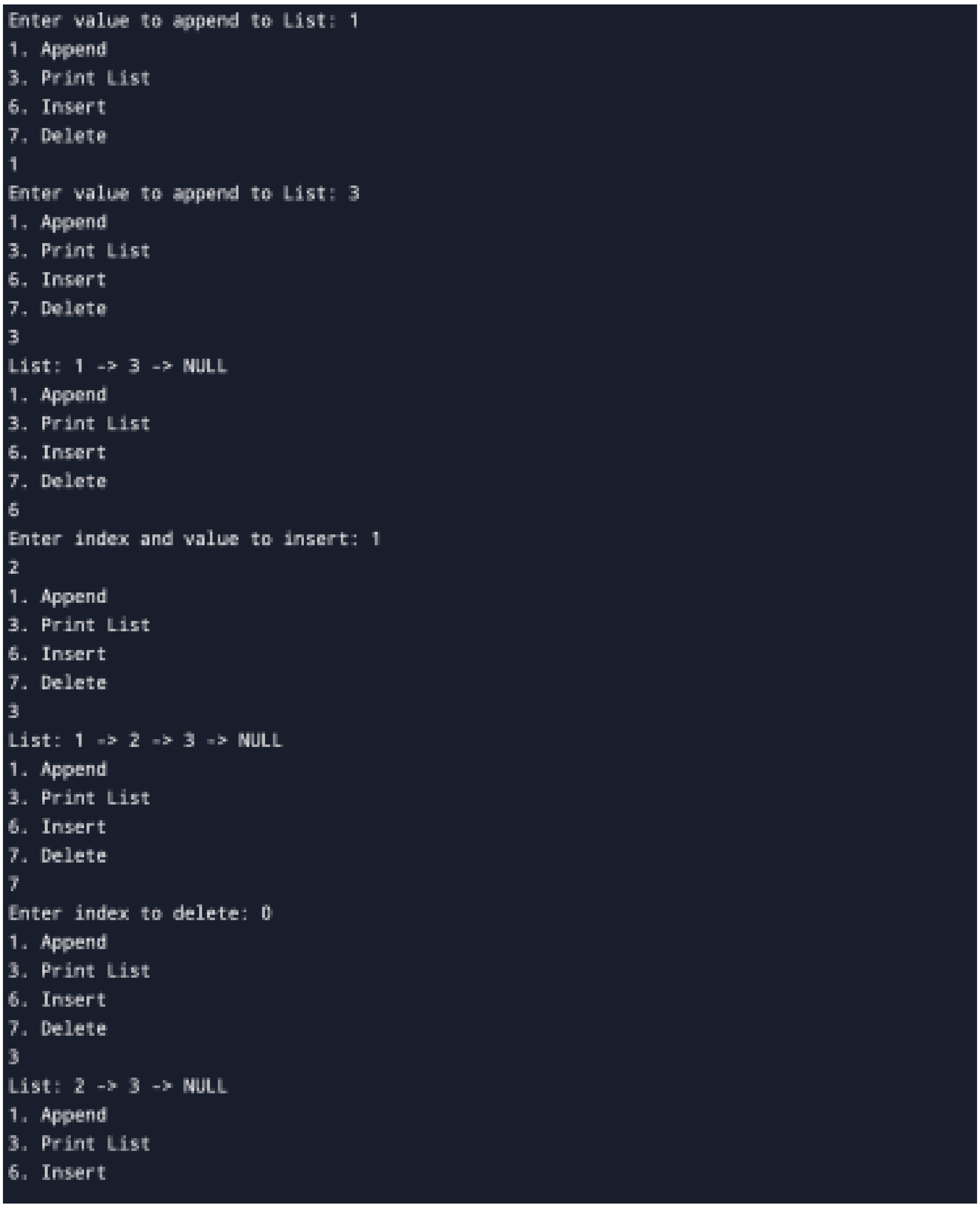
```
    list->length++;
    return 1;
}

Node* remove_node(LinkedList* list, int index) { if
    (index < 0 || index >= list->length) return NULL; if
    (index == 0) return pop_first(list);
    if (index == list->length - 1) return pop(list);

    Node* prev = get(list, index - 1);
    Node* temp = prev->next;
    prev->next = temp->next;
    temp->next = NULL;
    list->length--;

    return temp;
}

int main() {
    LinkedList* list1 = create_linked_list();
    int choice, value, index;

    do {
        printf("1. Append\n");
        printf("2. Pop\n");
        printf("3. Print List\n");
        printf("4. Prepend\n");
        printf("5. Pop_first\n");
        printf("6. Insert\n");
        printf("7. Delete\n");
        printf("8. Exit\n");
```

```
scanf("%d", &choice);

switch (choice) {
  case 1:
    printf("Enter value to append to List: ");
    scanf("%d", &value);
    append(list1, value);


    break;
  case 2:
    pop(list1);
    break;
  case 3:
    printf("List: ");
    print_list(list1);
    break;
  case 4:
    printf("Enter value to prepend to List: ");
    scanf("%d", &value);
    prepend(list1, value);
    break;
  case 5:
    pop_first(list1);
    break;
  case 6:
    printf("Enter index and value to insert: ");
    scanf("%d %d", &index, &value);
    insert(list1, index, value);
    break;
  case 7:
```

```
            printf("Enter index to delete: ");
            scanf("%d", &index);
            remove_node(list1, index);
            break;
        case 8:
            printf("Exiting...\n");
            break;


        default:
            printf("Invalid choice, please try again.\n");
    }
  } while (choice != 8);

  while (list1->length > 0) {
      Node* temp = pop(list1);
      free(temp);
  }
  free(list1);

  return 0;
}
```

## Output

```
7
Enter index to delete: 0
1. Append
2. Pop
3. Print List
5. Pop_first
7. Delete
3
List: 3 -> NULL
1. Append
2. Pop
3. Print List
5. Pop_first
7. Delete
2
1. Append
2. Pop
3. Print List
5. Pop_first
7. Delete
3
List: NULL
1. Append
2. Pop
3. Print List
5. Pop_first
7. Delete
1
Enter value to append to List: 0
1. Append
2. Pop
3. Print List
5. Pop_first
7. Delete
3
List: 0 -> NULL
```

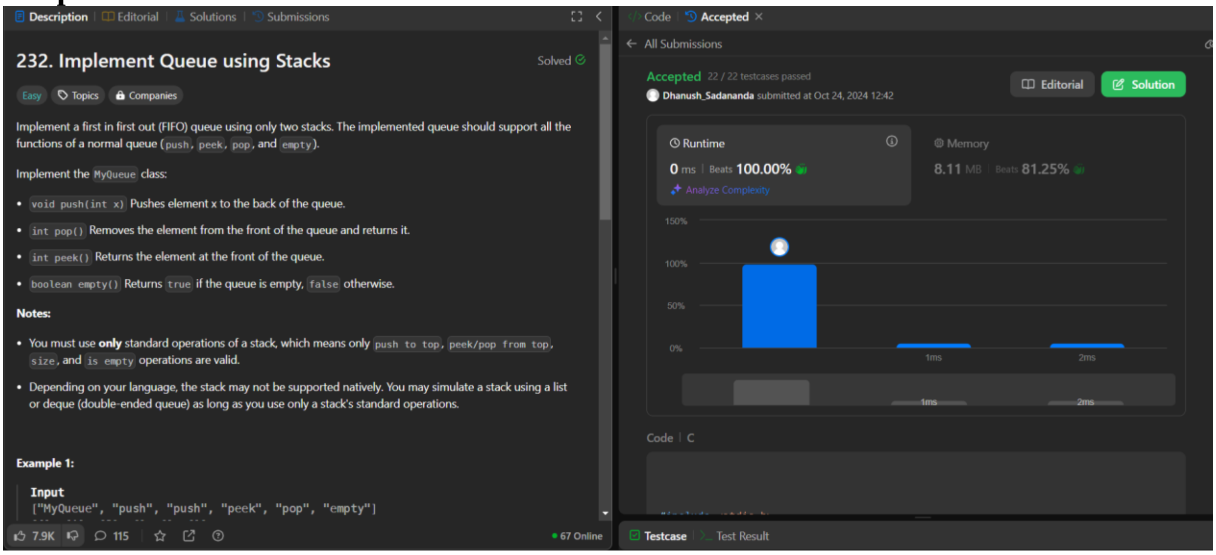


```
Enter value to append to List: 1
1. Append
3. Print List
6. Insert
7. Delete
1
Enter value to append to List: 3
1. Append
3. Print List
6. Insert
7. Delete
3
List: 1 -> 3 -> NULL
1. Append
3. Print List
6. Insert
7. Delete
6
Enter index and value to insert: 1
2
1. Append
3. Print List
6. Insert
7. Delete
3
List: 1 -> 2 -> 3 -> NULL
1. Append
3. Print List
6. Insert
7. Delete
7
Enter index to delete: 0
1. Append
3. Print List
6. Insert
7. Delete
3
List: 2 -> 3 -> NULL
1. Append
3. Print List
6. Insert
```

**Leetcode 83**

```
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;
}
```

**Output:**




## Lab Program-7:

### 7a. WAP to Implement Single Link List with following operations: Sort the linked list, Reverse the linked list, Concatenation of two linked lists.

```c
#include <stdio.h>

#include <stdlib.h>


struct Node {

    int data;

    struct Node* next;

};


void insert(struct Node** head, int data) {

    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));

    struct Node* temp = *head;

    newNode->data = data;

    newNode->next = NULL;

    if (*head == NULL) {

        *head = newNode;
```

```
        return;
    }
    while (temp->next != NULL) {
        temp = temp->next;
    }
    temp->next = newNode;
}

void display(struct Node* head) {
    struct Node* temp = head;

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

void sort(struct Node* head) {
    struct Node *current, *index;
    int temp;
    if (head == NULL) {
        return;
    }
    for (current = head; current != NULL; current = current->next) { for
        (index = current->next; index != NULL; index = index->next) { if
        (current->data > index->data) {
                temp = current->data;
                current->data = index->data;
                index->data = temp;
            }
```

```
        }
    }
}

void reverse(struct Node** head) {
    struct Node *prev = NULL, *current = *head, *next = NULL;
    while (current != NULL) {
        next = current->next;
        current->next = prev;


        prev = current;
        current = next;
    }
    *head = prev;
}

void concatenate(struct Node** head1, struct Node* head2) {
    if (*head1 == NULL) {
        *head1 = head2;
        return;
    }
    struct Node* temp = *head1;
    while (temp->next != NULL) {
        temp = temp->next;
    }
    temp->next = head2;
}

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

```
    struct Node* list2 = NULL;

    insert(&list1, 3);
    insert(&list1, 1);
    insert(&list1, 4);
    insert(&list1, 2);
    printf("Original List:\n");
    display(list1);



    sort(list1);
    printf("Sorted List:\n");
    display(list1);

    reverse(&list1);
    printf("Reversed List:\n");
    display(list1);

    insert(&list2, 5);
    insert(&list2, 6);
    printf("Second List:\n");
    display(list2);

    concatenate(&list1, list2);
    printf("Concatenated List:\n");
    display(list1);

    return 0;
}
```

**Output**

```
Original List:
3 -> 1 -> 4 -> 2 -> NULL
Sorted List:
1 -> 2 -> 3 -> 4 -> NULL
Reversed List:
4 -> 3 -> 2 -> 1 -> NULL
Second List:
5 -> 6 -> NULL
Concatenated List:
4 -> 3 -> 2 -> 1 -> 5 -> 6 -> NULL
```

**7b. WAP to Implement Single Link List to simulate Stack & Queue Operations.**

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

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

void push(struct Node** top, int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = *top;
    *top = newNode;
}

int pop(struct Node** top) {
    if (*top == NULL) {
        printf("Stack is empty!\n");
        return -1;
```

```
    }
    struct Node* temp = *top;
    int data = temp->data;
    *top = (*top)->next;
    free(temp);
    return data;
}



void enqueue(struct Node** front, struct Node** rear, int data) { struct
    Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = NULL;
    if (*rear == NULL) {
        *front = *rear = newNode;
        return;
    }
    (*rear)->next = newNode;
    *rear = newNode;
}

int dequeue(struct Node** front, struct Node** rear) {
    if (*front == NULL) {
        printf("Queue is empty!\n");
        return -1;
    }
    struct Node* temp = *front;
    int data = temp->data;
    *front = (*front)->next;
    if (*front == NULL) {
```

```
        *rear = NULL;
    }
    free(temp);
    return data;
}

void display(struct Node* head) {
    struct Node* temp = head;


    if (temp == NULL) {
        printf("Empty list.\n");
        return;
    }
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}

int main() {
    struct Node* stack = NULL;
    struct Node* front = NULL;
    struct Node* rear = NULL;
    int choice, value;

    while (1) {
        printf("\nChoose an operation:\n");
        printf("1. Push (Stack)\n");
        printf("2. Pop (Stack)\n");
```

```
printf("3. Enqueue (Queue)\n");
printf("4. Dequeue (Queue)\n");
printf("5. Display (Stack)\n");
printf("6. Display (Queue)\n");
printf("7. 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) {
            printf("Popped value: %d\n", value);
        }
        break;
    case 3:
        printf("Enter value to enqueue: ");
        scanf("%d", &value);
        enqueue(&front, &rear, value);
        break;
    case 4:
        value = dequeue(&front, &rear);
        if (value != -1) {
            printf("Dequeued value: %d\n", value);
```

```
            }
            break;
        case 5:
            printf("Stack: ");
            display(stack);
            break;
        case 6:
            printf("Queue: ");


            display(front);
            break;
        case 7:
            printf("Exiting...\n");
            exit(0);
        default:
            printf("Invalid choice, please try again.\n");
      }
   }
   return 0;
}
```

**Output**

```
Choose an operation:
1. Push (Stack)
2. Pop (Stack)
3. Enqueue (Queue)
4. Dequeue (Queue)
5. Display (Stack)
6. Display (Queue)
7. Exit
Enter your choice: 1
Enter value to push: 1

Choose an operation:
1. Push (Stack)
2. Pop (Stack)
3. Enqueue (Queue)
4. Dequeue (Queue)
5. Display (Stack)
6. Display (Queue)
7. Exit
Enter your choice: 1
Enter value to push: 2

Choose an operation:
1. Push (Stack)
2. Pop (Stack)
3. Enqueue (Queue)
4. Dequeue (Queue)
5. Display (Stack)
6. Display (Queue)
7. Exit
Enter your choice: 5
Stack: 2 -> 1 -> NULL
```

```
Choose an operation:
1. Push (Stack)
2. Pop (Stack)
3. Enqueue (Queue)
4. Dequeue (Queue)
5. Display (Stack)
6. Display (Queue)
7. Exit
Enter your choice: 2
Popped value: 2

Choose an operation:
1. Push (Stack)
2. Pop (Stack)
3. Enqueue (Queue)
4. Dequeue (Queue)
5. Display (Stack)
6. Display (Queue)
7. Exit
Enter your choice: 5
Stack: 1 -> NULL

Choose an operation:
1. Push (Stack)
2. Pop (Stack)
3. Enqueue (Queue)
4. Dequeue (Queue)
5. Display (Stack)
6. Display (Queue)
7. Exit
Enter your choice: 3
Enter value to enqueue: 1
```

**Lab program-8:**

**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;
};

void createList(struct Node** head) {
    *head = NULL;
}

void insertLeft(struct Node** head, int data, int value) { struct Node*
    newNode = (struct Node*)malloc(sizeof(struct Node)); newNode-
    >data = data;
    newNode->prev = NULL;
    newNode->next = NULL;

    if (*head == NULL) {
        *head = newNode;
        return;


    }
```

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

    if (temp != NULL) {
        newNode->next = temp;
        newNode->prev = temp->prev;
        if (temp->prev != NULL) {
            temp->prev->next = newNode;
        }
        temp->prev = newNode;

        if (*head == temp) {
            *head = newNode;
        }
    } else {
        printf("Node with value %d not found.\n", value);
    }
}

void deleteNode(struct Node** head, int value) {
    struct Node* temp = *head;

    if (temp != NULL && temp->data == value) {
        *head = temp->next;
        if (*head != NULL) {


            (*head)->prev = NULL;
        }
```

```
        free(temp);
        return;
    }

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

    if (temp == NULL) {
        printf("Node with value %d not found.\n", value);
        return;
    }

    if (temp->next != NULL) {
        temp->next->prev = temp->prev;
    }
    if (temp->prev != NULL) {
        temp->prev->next = temp->next;
    }
    free(temp);
}

void displayList(struct Node* head) {
    struct Node* temp = head;

    if (temp == NULL) {
        printf("The list is empty.\n");


        return;
    }
```

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

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

    createList(&head);

    while (1) {
        printf("\nChoose an operation:\n");
        printf("1. Insert node to the left of a node\n");
        printf("2. Delete node based on a specific value\n");
        printf("3. Display the list\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter the value to insert: ");


                scanf("%d", &data);
                printf("Enter the value to insert to the left of: ");
                scanf("%d", &value);
```

```
            insertLeft(&head, data, value);
            break;
        case 2:
            printf("Enter the value to delete: ");
            scanf("%d", &value);
            deleteNode(&head, value);
            break;
        case 3:
            displayList(head);
            break;
        case 4:
            printf("Exiting...\n");
            exit(0);
        default:
            printf("Invalid choice, please try again.\n");
        }
    }

    return 0;
}
```
**Output**

```
Choose an operation:
1. Insert node to the left of a node
2. Delete node based on a specific value
3. Display the list
4. Exit
Enter your choice: 1
Enter the value to insert: 20
Enter the value to insert to the left of: 20

Choose an operation:
1. Insert node to the left of a node
2. Delete node based on a specific value
3. Display the list
4. Exit
Enter your choice: 1
Enter the value to insert: 30
Enter the value to insert to the left of: 20

Choose an operation:
1. Insert node to the left of a node
2. Delete node based on a specific value
3. Display the list
4. Exit
Enter your choice: 3
Doubly Linked List: 30 <-> 20 <-> NULL
```

**Lab program 9:**

**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 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 {
        root->right = insert(root->right, data);
    }
    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 display(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("\nChoose an operation:\n");
        printf("1. Insert node into BST\n");
        printf("2. Display the tree (In-order, Pre-order, Post-order)\n");
        printf("3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter value to insert: ");
                scanf("%d", &value);
                root = insert(root, value);
                break;
            case 2:
                display(root);
                break;
            case 3:
                printf("Exiting...\n");
                exit(0);
            default:
                printf("Invalid choice, please try again.\n");
        }
    }

    return 0;
```

}

**Output**

```
2. Display the tree (In-order, Pre-order, Post-order)
3. Exit
Enter your choice: 1
Enter value to insert: 20

Choose an operation:
1. Insert node into BST
2. Display the tree (In-order, Pre-order, Post-order)
3. Exit
Enter your choice: 1
Enter value to insert: 30

Choose an operation:
1. Insert node into BST
2. Display the tree (In-order, Pre-order, Post-order)
3. Exit
Enter your choice: 1
Enter value to insert: 30

Choose an operation:
1. Insert node into BST
2. Display the tree (In-order, Pre-order, Post-order)
3. Exit
Enter your choice: 2
In-order traversal: 20 30 30
Pre-order traversal: 20 30 30
Post-order traversal: 30 30 20
```

**Lab Program 9:**

**9a. Write a program to traverse a graph using the BFS method.**

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

#define MAX 20

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

struct Graph {
    int vertices;
    struct Node* adjList[MAX];
};

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

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

void addEdge(struct Graph* graph, int src, int dest) {
```

```
        struct Node* newNode = createNode(dest);

    newNode->next = graph->adjList[src];
    graph->adjList[src] = newNode;

    newNode = createNode(src);
    newNode->next = graph->adjList[dest];
    graph->adjList[dest] = newNode;
}

void bfs(struct Graph* graph, int startVertex) {
    int visited[MAX] = {0};
    int queue[MAX], front = -1, rear = -1;

    rear++;
    queue[rear] = startVertex;
    visited[startVertex] = 1;

    printf("BFS Traversal starting from vertex %d: ", startVertex);

    while (front != rear) {
        front++;
        int currentVertex = queue[front];
        printf("%d ", currentVertex);

        struct Node* temp = graph-
        >adjList[currentVertex]; while (temp != NULL) {
            int adjVertex = temp->vertex;
            if (!visited[adjVertex]) {
                rear++;
                queue[rear] = adjVertex;
                visited[adjVertex] = 1;
            }
            temp = temp->next;
        }
```

```
    }

    printf("\n");
}

int main() {
    struct Graph graph;
    int vertices, edges, src, dest, startVertex;

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

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

    for (int i = 0; i < edges; i++) {
        printf("Enter source vertex: ");
        scanf("%d", &src);

        printf("Enter destination vertices (separate by space, end with -1): ");
        while (1) {
            scanf("%d", &dest);
            if (dest == -1) {
                break;
            }
            addEdge(&graph, src, dest);
        }
    }

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

    bfs(&graph, startVertex);
```

```
    return 0;
}
```

**Output**

```
Enter the number of vertices: 5
Enter the number of edges: 4
Enter source vertex: 0
Enter destination vertices (separate by space, end with -1): 1 -1
Enter source vertex: 0
Enter destination vertices (separate by space, end with -1): 2 -1
Enter source vertex: 1
Enter destination vertices (separate by space, end with -1): 3 -1
Enter source vertex: 2
Enter destination vertices (separate by space, end with -1): 4 -1
Enter the starting vertex for BFS: 0
BFS Traversal starting from vertex 0: 0 2 1 4 3
```

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

```
#include <stdio.h>

#include <stdlib.h>


#define MAX 20


struct Node {

    int vertex;

    struct Node* next;

};


struct Graph {

    int vertices;

    struct Node* adjList[MAX];

};
```

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

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

}

void addEdge(struct Graph* graph, int src, int dest) {
    struct Node* newNode = createNode(dest);
    newNode->next = graph->adjList[src];
    graph->adjList[src] = newNode;

    newNode = createNode(src);
    newNode->next = graph->adjList[dest];
    graph->adjList[dest] = newNode;
}

void dfs(struct Graph* graph, int vertex, int visited[]) {
    visited[vertex] = 1;

    struct Node* temp = graph->adjList[vertex];
    while (temp != NULL) {
        int adjVertex = temp->vertex;
```

```
        if (!visited[adjVertex]) {
            dfs(graph, adjVertex, visited);
        }
        temp = temp->next;
    }
}

int isConnected(struct Graph* graph) {
    int visited[MAX] = {0};

    dfs(graph, 0, visited);

    for (int i = 0; i < graph->vertices; i++) {
        if (!visited[i]) {
            return 0;
        }
    }
    return 1;
}

int main() {
    struct Graph graph;
    int vertices, edges, src, dest;

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

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

```
    for (int i = 0; i < edges; i++) {
        printf("Enter source vertex: ");
        scanf("%d", &src);
        printf("Enter destination vertex: ");
        scanf("%d", &dest);
        addEdge(&graph, src, dest);
    }

    if (isConnected(&graph)) {
printf("The graph is connected.\n");
} else {
printf("The graph is not connected.\n");
}
printf("Name:Dhanush S\n");
printf("USN:1BM23CS089");

return 0;
}
```

**Output**

```
Enter the number of vertices: 5
Enter the number of edges: 4
Enter source vertex: 0
Enter destination vertex: 1
Enter source vertex: 0
Enter destination vertex: 2
Enter source vertex: 1
Enter destination vertex: 3
Enter source vertex: 3
Enter destination vertex: 4
The graph is connected.
Name:Dhanush S
USN:1BM23CS089
```

}

**Lab Program 10:**

**Linear Probing**

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

int key[20], n, m;
int *ht, index;
int count = 0;

void insert(int key) {
    index = key % m;
    while (ht[index] != -1) {
        index = (index + 1) % m;
    }
    ht[index] = key;
    count++;
}

void display() {
```

```
    int i;
    if (count == 0) {
        printf("\nHash Table is empty");
        return;
    }
    printf("\nHash Table contents are:\n");
    for (i = 0; i < m; i++) {
        printf("\nT[%d] --> %d", i, ht[i]);
    }
}

void main() {
    int i;
    printf("\nEnter the number of employee records (N): ");
    scanf("%d", &n);

    printf("\nEnter the two-digit memory locations (m) for hash table: ");
    scanf("%d", &m);

    ht = (int *)malloc(m * sizeof(int));
    for (i = 0; i < m; i++) {
        ht[i] = -1;
    }

    printf("\nEnter the four-digit key values (K) for N Employee Records:\n");
    for (i = 0; i < n; i++) {
        scanf("%d", &key[i]);
    }

    for (i = 0; i < n; i++) {
```

```
        if (count == m) {
            printf("\nHash table is full. Cannot insert the record %d key", i + 1);
            break;
        }
        insert(key[i]);
    }

    display();
}
```

**Output**

```
Enter the number of employee records (N): 5

Enter the two-digit memory locations (m) for hash table: 7

Enter the four-digit key values (K) for N Employee Records:
1234
5678
9011
5441
8765

Hash Table contents are:

T[0] --> -1
T[1] --> 5678
T[2] --> 1234
T[3] --> 9011
T[4] --> 5441
T[5] --> 8765
T[6] --> -1
```