VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
On

DATA STRUCTURES (23CS3PCDST)

Submitted by

RUSHI HUNDIWALA (1BM22CS224)

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

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

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

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

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

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

Index Sheet

Sl. No.	Experiment Title	Page No.
1	Swapping Using Pointers, Dynamic Memory Allocation, Stack Implementation	4
2	Infix to Postfix conversion AND Evaluation of Postfix Expression	7
3	Queue Implementation and Circular Queue Implementation	9
4	Singly Linked List Insert and display Implementation(LeetCode)	13
5	Singly Linked List delete and display Implementation (LeetCodes)	17
6	Sort the linked list, Reverse the linked list, Concatenation of two linked lists. Implement Single Linked List to simulate Stack & Queue Operations	17
7	Implement doubly link list with primitive operations (LeetCode)	26
8	To construct a binary Search Tree, To traverse the tree using all the methods i.e., in-order, preorder and postorder and To display the elements in the tree.	34
9	Write a program to traverse a graph using BFS method.Write a program to check whether given graph is connected or not using DFS method. (HackerRank)	42
10	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.	48

Course outcomes:

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

Lab program 1:

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

```
#include <stdio.h>
#include<stdlib.h>
#define STACK_SIZE 5
void push(int st[],int *top)
{
        int item;
        if(*top==STACK_SIZE-1)
                printf("Stack overflow\n");
        else
        {
                printf("\nEnter an item :");
                scanf("%d",&item);
                (*top)++;
                st[*top]=item;
        }
}
void pop(int st[],int *top)
{
        if(*top==-1)
                printf("Stack underflow\n");
        else
        {
                printf("\n%d item was deleted",st[(*top)--]);
        }
}
void display(int st[],int *top)
{
        int i;
        if(*top==-1)
                printf("Stack is empty\n");
        for(i=0;i<=*top;i++)
                printf("%d\t",st[i]);
}
void main()
{
        int st[10],top=-1, c,val_del;
        while(1)
        {
                printf("\n1. Push\n2. Pop\n3. Display\n");
                printf("\nEnter your choice :");
                scanf("%d",&c);
```

```
                switch(c)
                {
                        case 1: push(st,&top);
                                break;
                        case 2: pop(st,&top);
                                break;
                        case 3: display(st,&top);
                                break;
                        default: printf("\nInvalid choice!!!");
                                exit(0);
                }
        }
}
```

Output:

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL

PS D:\jyothika\DST> cd "d:\jyothika\DST\" ; if ($?) { gcc 1.c -o 1 } ; if ($?) { .\1 }

1. Push
2. Pop
3. Display

Enter your choice :1

Enter an item :12

1. Push
2. Pop
3. Display

Enter your choice :1

Enter an item :65

1. Push
2. Pop
3. Display

Enter your choice :1

Enter an item :45

1. Push
2. Pop
3. Display

Enter your choice :1
Stack overflow
```

```
1. Push
2. Pop
3. Display

Enter your choice :2

45 item was deleted
1. Push
2. Pop
3. Display

Enter your choice :2

65 item was deleted
1. Push
2. Pop
3. Display

Enter your choice :3
12
1. Push
2. Pop
3. Display

Enter your choice :2

12 item was deleted
1. Push
2. Pop
3. Display

Enter your choice :2
Stack underflow

1. Push
2. Pop
3. Display

Enter your choice :4

Invalid choice!!!
```

NOTE: SIMILARLY INCLUDE FOR ALL THE LAB PROGRAMS (ALL 10). LEETCODE AND HACKERRANK PROGRAMS ALSO TO BE INCLUDED.

LAB 2

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

#define MAX_SIZE 100

typedef struct {
    char items[MAX_SIZE];
    int top;
} Stack;

void push(Stack *s, char item) {
    if (s->top == MAX_SIZE - 1) {
        printf("Stack Overflow\n");
        exit(1);
    }
    s->items[++(s->top)] = item;
}

char pop(Stack *s) {
    if (s->top == -1) {
        printf("Stack Underflow\n");
        exit(1);
    }
    return s->items[(s->top)--];
}

int is_operator(char c) {
    return (c == '+' || c == '-' || c == '*' || c == '/');
```

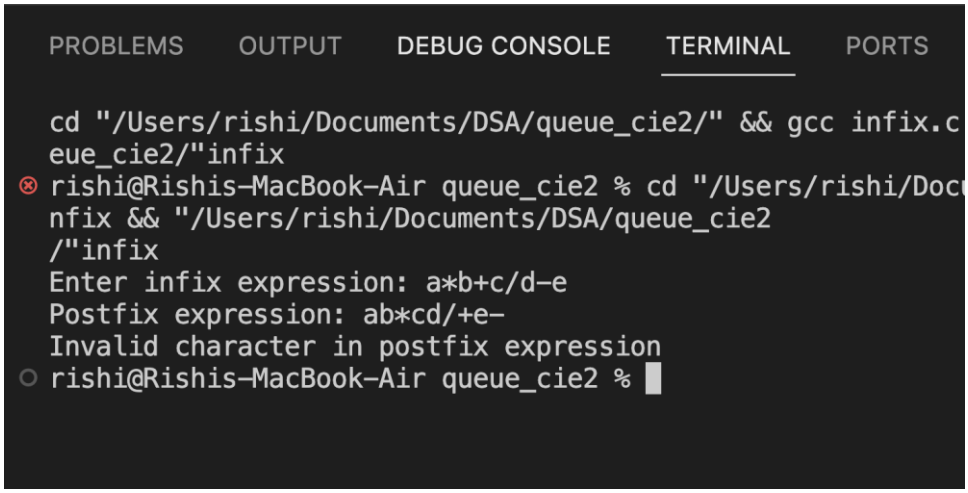
```
}

int precedence(char c) {
    if (c == '+' || c == '-')
        return 1;
    else if (c == '*' || c == '/')
        return 2;
    return 0;
}

void infix_to_postfix(char infix[], char postfix[]) {
    Stack s;
    s.top = -1;
    int i = 0, j = 0;

    while (infix[i] != '\0') {
        if (isdigit(infix[i]) || isalpha(infix[i]))
            postfix[j++] = infix[i];
        else if (infix[i] == '(')
            push(&s, '(');
        else if (infix[i] == ')') {
            while (s.top != -1 && s.items[s.top] != '(')
                postfix[j++] = pop(&s);
            if (s.top == -1) {
                printf("Invalid expression: Unmatched parenthesis\n");
                exit(1);
            }
            pop(&s); // Discard '('
        }
        else if (is_operator(infix[i])) {
            while (s.top != -1 && precedence(s.items[s.top]) >= precedence(infix[i]))
                postfix[j++] = pop(&s);
```

```
            push(&s, infix[i]);
        }
        else {
            printf("Invalid character in infix expression\n");
            exit(1);
        }
        i++;
    }

    while (s.top != -1) {
        if (s.items[s.top] == '(') {
            printf("Invalid expression: Unmatched parenthesis\n");
            exit(1);
        }
        postfix[j++] = pop(&s);
    }
    postfix[j] = '\0';
}

int evaluate_postfix(char postfix[]) {
    Stack s;
    s.top = -1;
    int i = 0, operand1, operand2, result;

    while (postfix[i] != '\0') {
        if (isdigit(postfix[i])) {
            push(&s, postfix[i] - '0');
        }
        else if (is_operator(postfix[i])) {
            operand2 = pop(&s);
            operand1 = pop(&s);
            switch (postfix[i]) {
```

```
            case '+':
                push(&s, operand1 + operand2);
                break;
            case '-':
                push(&s, operand1 - operand2);
                break;
            case '*':
                push(&s, operand1 * operand2);
                break;
            case '/':
                if (operand2 == 0) {
                    printf("Division by zero\n");
                    exit(1);
                }
                push(&s, operand1 / operand2);
                break;
        }
    }
    else {
        printf("Invalid character in postfix expression\n");
        exit(1);
    }
    i++;
}

result = pop(&s);
if (s.top != -1) {
    printf("Invalid expression: Too many operands\n");
    exit(1);
}
return result;
}
```

```
int main() {
    char infix[MAX_SIZE], postfix[MAX_SIZE];
    printf("Enter infix expression: ");
    scanf("%s", infix);

    infix_to_postfix(infix, postfix);
    printf("Postfix expression: %s\n", postfix);

    int result = evaluate_postfix(postfix);
    printf("Result of evaluation: %d\n", result);

    return 0;
}
```

OUTPUT:


PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
cd "/Users/rishi/Documents/DSA/queue_cie2/" && gcc infix.c
eue_cie2/"infix
rishi@Rishis-MacBook-Air queue_cie2 % cd "/Users/rishi/Doc
nfix && "/Users/rishi/Documents/DSA/queue_cie2
/"infix
Enter infix expression: a*b+c/d-e
Postfix expression: ab*cd/+e-
Invalid character in postfix expression
rishi@Rishis-MacBook-Air queue_cie2 %

LAB 3

A)

QUEUE IMPLEMENTATION

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

typedef struct {
    int items[MAX_SIZE];
    int front;
    int rear;
} Queue;

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

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

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

void enqueue(Queue *q, int data) {
    if (isFull(q)) {
        printf("Queue is full\n");
        return;
    }
    if (isEmpty(q)) {
        q->front = 0;
        q->rear = 0;
```

```
    } else {
        q->rear = (q->rear + 1) % MAX_SIZE;
    }
    q->items[q->rear] = data;
}

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

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

    enqueue(&q, 1);
    enqueue(&q, 2);
    enqueue(&q, 3);
    enqueue(&q, 4);

    printf("Dequeued: %d\n", dequeue(&q));
    printf("Dequeued: %d\n", dequeue(&q));
    printf("Dequeued: %d\n", dequeue(&q));
    printf("Dequeued: %d\n", dequeue(&q));
```

```
    return 0;
}
```

OUTPUT:

B)

Circular Queue Implementation

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

#define MAX_SIZE 5

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

```
void initCircularQueue(CircularQueue *cq) {
    cq->front = -1;
    cq->rear = -1;
}

int isEmptyCircularQueue(CircularQueue *cq) {
    return cq->front == -1;
}

int isFullCircularQueue(CircularQueue *cq) {
    return (cq->rear + 1) % MAX_SIZE == cq->front;
}

void enqueueCircularQueue(CircularQueue *cq, int data) {
    if (isFullCircularQueue(cq)) {
        printf("Queue is full\n");
        return;
    }
    if (isEmptyCircularQueue(cq)) {
        cq->front = 0;
    }
    cq->rear = (cq->rear + 1) % MAX_SIZE;
    cq->items[cq->rear] = data;
}

int dequeueCircularQueue(CircularQueue *cq) {
    if (isEmptyCircularQueue(cq)) {
        printf("Queue is empty\n");
        exit(1);
    }
    int data = cq->items[cq->front];
```

```
    if (cq->front == cq->rear) {
        cq->front = -1;
        cq->rear = -1;
    } else {
        cq->front = (cq->front + 1) % MAX_SIZE;
    }
    return data;
}

int main() {
    CircularQueue cq;
    initCircularQueue(&cq);

    enqueueCircularQueue(&cq, 1);
    enqueueCircularQueue(&cq, 2);
    enqueueCircularQueue(&cq, 3);
    enqueueCircularQueue(&cq, 4);

    printf("Dequeued: %d\n", dequeueCircularQueue(&cq));
    printf("Dequeued: %d\n", dequeueCircularQueue(&cq));

    enqueueCircularQueue(&cq, 5);
    enqueueCircularQueue(&cq, 6);

    printf("Dequeued: %d\n", dequeueCircularQueue(&cq));
    printf("Dequeued: %d\n", dequeueCircularQueue(&cq));
    printf("Dequeued: %d\n", dequeueCircularQueue(&cq));

    return 0;
}
```

OUTPUT:

LAB 4

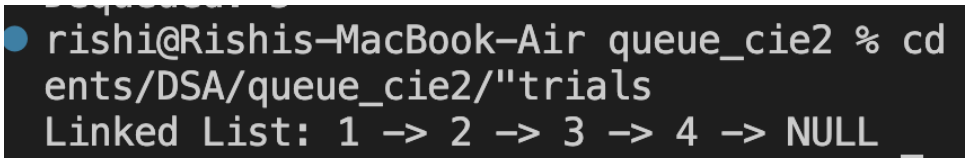
Singly Linked List insertion

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

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

// Function to create a new node
Node* createNode(int data) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    if (newNode == NULL) {
        printf("Memory allocation failed\n");
```

```
        exit(1);
    }
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

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

// Function to display the linked list
void display(Node* head) {
    if (head == NULL) {
        printf("List is empty\n");
    } else {
        printf("Linked List: ");
        while (head != NULL) {
            printf("%d -> ", head->data);
            head = head->next;
        }
```

```
        printf("NULL\n");
    }
}

int main() {
    Node* head = NULL;

    // Insert some elements at the end of the linked list
    insertEnd(&head, 1);
    insertEnd(&head, 2);
    insertEnd(&head, 3);
    insertEnd(&head, 4);

    // Display the linked list
    display(head);

    return 0;
}
```

OUTPUT:



LAB 5

Singly Linked List deletion

name= rushi hunwdiwala

usn=1BM22CS224

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

struct node *head = NULL;
void pop() {
    if (head == NULL) {
        printf("EMPTY LIST");
        return;
    }
```

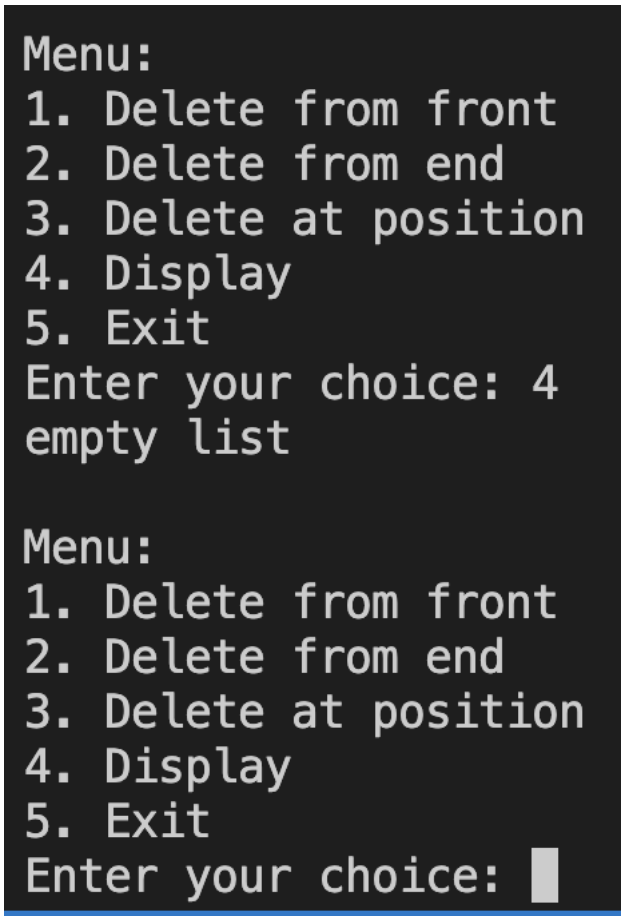
```
    struct node *ptr = head;
    head = head->next;
    free(ptr);
    printf("node deleted\n");
}

void end_delete() {
    if (head == NULL) {
        printf("list is empty\n");
        return;
    }
    if (head->next == NULL) {
        free(head);
        head = NULL;
        printf("node deleted from end\n");
        return;
    }

    struct node *ptr = head;
    struct node *ptr1 = NULL;

    while (ptr->next != NULL) {
        ptr1 = ptr;
        ptr = ptr->next;
    }

    ptr1->next = NULL;
    free(ptr);
    printf("node deleted from end\n");
}
```

```
void delete_at_pos(int position) {
    struct node *ptr = head;
    struct node *ptr1 = NULL;
    if (head == NULL) {
        printf("EMPTY LIST\n");
        return;
    }

for (int i = 1; ptr != NULL && i < position; i++) {
    ptr1 = ptr;
    ptr = ptr->next;
}

    if (ptr == NULL) {
        printf("there are fewer elements\n");
        return;
    }

    if (ptr1 == NULL) {
        head = ptr->next;
    } else {
        ptr1->next = ptr->next;
    }

    free(ptr);
    printf("node deleted from position\n");
}

void display() {
    if (head == NULL) {
        printf("empty list\n");
```

```
        return;
    }
    struct node *current = head;
    printf("linked list: ");
    while (current != NULL) {
        printf("%d ", current->data);
        current = current->next;
    }
    printf("\n");
}

int main() {
    int choice;

    do {
        printf("\nMenu:\n");
        printf("1. Delete from front\n");
        printf("2. Delete from end\n");
        printf("3. Delete at position\n");
        printf("4. Display\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                pop();
                break;
            case 2:
                end_delete();
                break;
```

```
            case 3: {
                int position;
                printf("Enter the position from where you want to delete: ");
                scanf("%d", &position);
                delete_at_pos(position);
                break;
            }
            case 4:
                display();
                break;
            case 5:
                printf("Exiting program.\n");
                break;
            default:
                printf("Invalid choice. Please try again.\n");
        }
    } while (choice != 5);

    return 0;
}
```

OUTPUT:


Menu:
1. Delete from front
2. Delete from end
3. Delete at position
4. Display
5. Exit
Enter your choice: 4
empty list
Menu:
1. Delete from front
2. Delete from end
3. Delete at position
4. Display
5. Exit
Enter your choice:

LAB 6

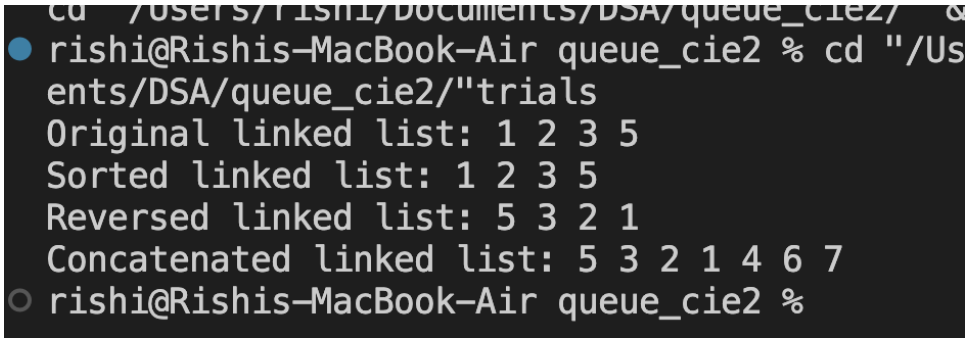
```
#include <stdio.h>
#include <stdlib.h>
// Node structure
struct Node {
int data;
struct Node *next;
};
// Function to create a new node
struct Node *newNode(int data) {
struct Node *node = (struct Node *)malloc(sizeof(struct Node));
node->data = data;
node->next = NULL;
return node;
}
// Function to insert a node at the beginning of the linked list
void insertAtBeginning(struct Node **head, int data) {
struct Node *node = newNode(data);
node->next = *head;
*head = node;
}
// Function to print the linked list
void printLinkedList(struct Node *head) {
struct Node *temp = head;
while (temp != NULL) {
printf("%d ", temp->data);
temp = temp->next;
}
printf("\n");
}
// Function to sort the linked list using the bubble sort algorithm
void sortLinkedList(struct Node **head) {
struct Node *current = *head;
struct Node *next = NULL;
int swapped;
do {
```

```
swapped = 0;
current = *head;
while (current->next != NULL) {
next = current->next;
if (current->data > next->data) {
int temp = current->data;
current->data = next->data;
next->data = temp;
swapped = 1;
}
current = current->next;
}
} while (swapped);
}
// Function to reverse the linked list
void reverseLinkedList(struct Node **head) {
struct Node *previous = NULL;
struct Node *current = *head;
struct Node *next = NULL;
while (current != NULL) {
next = current->next;
current->next = previous;
previous = current;
current = next;
}
*head = previous;
}
// Function to concatenate two linked lists
struct Node *concatenateLinkedLists(struct Node *head1, struct Node
*head2) {
struct Node *temp = head1;
while (temp->next != NULL) {
temp = temp->next;
}
temp->next = head2;
return head1;
}
// Function to implement a stack using a singly linked list
void push(struct Node **head, int data) {
```

```
insertAtBeginning(head, data);
}
int pop(struct Node **head) {
if (*head == NULL) {
printf("Stack is empty\n");
return -1;
}
int data = (*head)->data;
*head = (*head)->next;
return data;
}
// Function to implement a queue using a singly linked list
void enqueue(struct Node **head, int data) {
insertAtBeginning(head, data);
}
int dequeue(struct Node **head) {
if (*head == NULL) {
printf("Queue is empty\n");
return -1;
}
struct Node *temp = *head;
while (temp->next != NULL) {
temp = temp->next;
}
int data = temp->data;
free(temp);
return data;
}
// Main function
int main() {
// Create a linked list
struct Node *head = NULL;
insertAtBeginning(&head, 5);
insertAtBeginning(&head, 3);
insertAtBeginning(&head, 2);
insertAtBeginning(&head, 1);
// Print the linked list
printLinkedList(head);
// Sort the linked list
```

```
sortLinkedList(&head);
// Print the sorted linked list
printLinkedList(head);
// Reverse the linked list
reverseLinkedList(&head);
// Print the reversed linked list
printLinkedList(head);
// Concatenate two linked lists
struct Node *head2 = NULL;
insertAtBeginning(&head2, 7);
insertAtBeginning(&head2, 6);
insertAtBeginning(&head2, 4);
head = concatenateLinkedLists(head, head2);
```

OUTPUT:



LAB 7

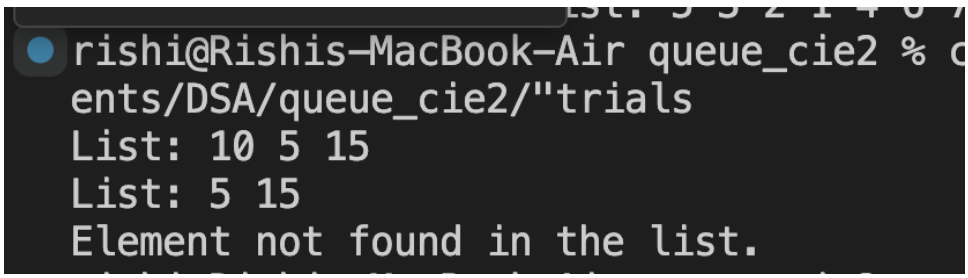
```
#include <stdio.h>
#include <stdlib.h>

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

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

// Function to insert a node at the beginning
void insertAtBeginning(struct Node** head, int data) {
    struct Node* newNode = createNode(data);
    if (*head == NULL) {
        *head = newNode;
    } else {
```

```
        newNode->next = *head;
        (*head)->prev = newNode;
        *head = newNode;
    }
}

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

// Function to delete a node by value
void deleteNode(struct Node** head, int data) {
    struct Node* temp = *head;
    if (temp == NULL) {
        printf("List is empty.\n");
        return;
    }
    if (temp->data == data) {
        *head = temp->next;
        if (*head != NULL)
```

```
            (*head)->prev = NULL;
        free(temp);
        return;
    }
    while (temp != NULL && temp->data != data) {
        temp = temp->next;
    }
    if (temp == NULL) {
        printf("Element not found in the list.\n");
        return;
    }
    if (temp->prev != NULL)
        temp->prev->next = temp->next;
    if (temp->next != NULL)
        temp->next->prev = temp->prev;
    free(temp);
}

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

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

```
    insertAtBeginning(&head, 5);
    insertAtBeginning(&head, 10);
    insertAtEnd(&head, 15);
    displayList(head);

    deleteNode(&head, 10);
    displayList(head);

    deleteNode(&head, 20);

    return 0;
}
```

OUTPUT:



LAB 8

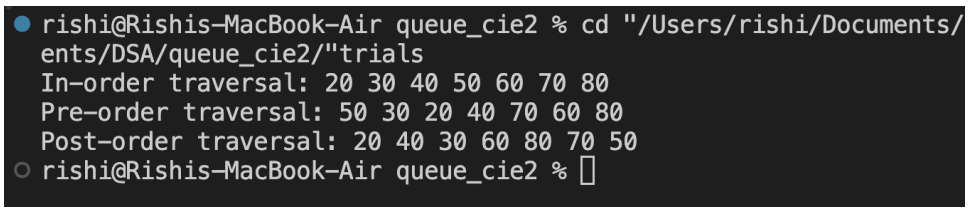
```
#include <stdio.h>
#include <stdlib.h>
```

```
// Define a structure for a binary search tree node
struct TreeNode {
    int data;
    struct TreeNode *left;
    struct TreeNode *right;
};

// Function to create a new node
struct TreeNode *createNode(int value) {
    struct TreeNode *newNode = (struct TreeNode *)malloc(sizeof(struct TreeNode));
    newNode->data = value;
    newNode->left = newNode->right = NULL;
    return newNode;
}

// Function to insert a value into a BST
struct TreeNode *insert(struct TreeNode *root, int value) {
    // If the tree is empty, create a new node and return it
    if (root == NULL) {
        return createNode(value);
    }

    // Otherwise, recur down the tree
    if (value < root->data) {
        root->left = insert(root->left, value);
    } else if (value > root->data) {
        root->right = insert(root->right, value);
    }

    // Return the (unchanged) node pointer
```

```
    return root;
}

// Function to perform in-order traversal of BST
void inorderTraversal(struct TreeNode *root) {
    if (root != NULL) {
        inorderTraversal(root->left);
        printf("%d ", root->data);
        inorderTraversal(root->right);
    }
}

// Function to perform pre-order traversal of BST
void preorderTraversal(struct TreeNode *root) {
    if (root != NULL) {
        printf("%d ", root->data);
        preorderTraversal(root->left);
        preorderTraversal(root->right);
    }
}

// Function to perform post-order traversal of BST
void postorderTraversal(struct TreeNode *root) {
    if (root != NULL) {
        postorderTraversal(root->left);
        postorderTraversal(root->right);
        printf("%d ", root->data);
    }
}

// Function to display the elements of the BST
```

```
void display(struct TreeNode *root) {
    printf("In-order traversal: ");
    inorderTraversal(root);
    printf("\n");

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

    printf("Post-order traversal: ");
    postorderTraversal(root);
    printf("\n");
}

int main() {
    struct TreeNode *root = NULL;

    // Insert elements into the binary search tree
    root = insert(root, 50);
    root = insert(root, 30);
    root = insert(root, 20);
    root = insert(root, 40);
    root = insert(root, 70);
    root = insert(root, 60);
    root = insert(root, 80);

    // Display the elements of the binary search tree
    display(root);

    return 0;
}
```

OUTPUT:



LAB 9

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

#define MAX_VERTICES 100

int graph[MAX_VERTICES][MAX_VERTICES];
```

```
int visited[MAX_VERTICES];
int queue[MAX_VERTICES];
int front = -1, rear = -1;
int vertices, edges;

// Function to add an edge to the graph
void addEdge(int v1, int v2) {
    graph[v1][v2] = 1;
    graph[v2][v1] = 1; // Assuming undirected graph
}

// Function to perform Breadth First Search traversal of the graph
void BFS(int start) {
    printf("BFS traversal starting from vertex %d: ", start);
    visited[start] = true;
    enqueue(start);

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

        for (int i = 0; i < vertices; i++) {
            if (graph[currentVertex][i] && !visited[i]) {
                visited[i] = true;
                enqueue(i);
            }
        }
    }
    printf("\n");
}
```

```
// Function to add an element to the queue
void enqueue(int vertex) {
    if (rear == MAX_VERTICES - 1)
        printf("Queue Overflow\n");
    else {
        if (front == -1)
            front = 0;
        rear++;
        queue[rear] = vertex;
    }
}

// Function to remove an element from the queue
int dequeue() {
    int deletedVertex;
    if (front == -1)
        printf("Queue Underflow\n");
    else {
        deletedVertex = queue[front];
        front++;
        if (front > rear)
            front = rear = -1;
        return deletedVertex;
    }
}

// Function to check if the queue is empty
bool isEmpty() {
    return front == -1;
}
```

```
// Function to perform Depth First Search traversal of the graph
void DFS(int vertex) {
    visited[vertex] = true;
    for (int i = 0; i < vertices; i++) {
        if (graph[vertex][i] && !visited[i]) {
            DFS(i);
        }
    }
}

// Function to check if the given graph is connected or not using DFS
bool isConnected() {
    for (int i = 0; i < vertices; i++)
        visited[i] = false;

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

    for (int i = 0; i < vertices; i++) {
        if (!visited[i]) {
            return false;
        }
    }
    return true;
}

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

    printf("Enter the edges (vertex pairs):\n");
    for (int i = 0; i < edges; i++) {
```

```
        int v1, v2;
        scanf("%d %d", &v1, &v2);
        addEdge(v1, v2);
    }

    if (isConnected())
        printf("The graph is connected.\n");
    else
        printf("The graph is not connected.\n");

    printf("BFS traversal of the graph:\n");
    for (int i = 0; i < vertices; i++) {
        if (!visited[i])
            BFS(i);
    }

    return 0;
}
```

OUTPUT:

```
Enter the number of vertices and edges: 5 4
Enter the edges (vertex pairs):
0 1
0 2
1 3
3 4
The graph is connected.
BFS traversal of the graph:
BFS traversal starting from vertex 0: 0 1 2 3 4
```

LAB 10

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

#define SIZE 10 // Size of the hash table

int hash_table[SIZE];

// Function to initialize hash table
void initializeHashTable() {
    for (int i = 0; i < SIZE; i++) {
        hash_table[i] = -1; // -1 indicates empty slot
    }
}
```

```
// Function to calculate hash value using remainder method
int hash(int key) {
    return key % SIZE;
}

// Function to insert a key into the hash table using linear probing
void insert(int key) {
    int index = hash(key);

    // If slot is empty, insert key
    if (hash_table[index] == -1) {
        hash_table[index] = key;
    } else {
        // Collision occurred, find next empty slot using linear probing
        while (hash_table[index] != -1) {
            index = (index + 1) % SIZE;
        }
        hash_table[index] = key;
    }
}

// Function to search for a key in the hash table
int search(int key) {
    int index = hash(key);

    // Search until an empty slot is found or key is found
    while (hash_table[index] != -1) {
        if (hash_table[index] == key) {
            return index; // Key found
        }
        index = (index + 1) % SIZE; // Move to next slot
```

```
    }

    return -1; // Key not found
}

// Function to display the hash table
void displayHashTable() {
    printf("Hash Table:\n");
    for (int i = 0; i < SIZE; i++) {
        printf("%d: ", i);
        if (hash_table[i] != -1) {
            printf("%d", hash_table[i]);
        }
        printf("\n");
    }
}

int main() {
    initializeHashTable();

    // Insert keys into the hash table
    insert(12);
    insert(25);
    insert(35);
    insert(26);
    insert(41);
    insert(15);
    insert(76);

    displayHashTable();
```

```
    // Search for a key
    int key_to_search = 26;
    int index = search(key_to_search);
    if (index != -1) {
        printf("Key %d found at index %d\n", key_to_search, index);
    } else {
        printf("Key %d not found\n", key_to_search);
    }

    return 0;
}
```

OUTPUT: