VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.

LAB REPORT
On

DATA STRUCTURES (23CS3PCDST)

Submitted by

TARANG RAJYAVANSHI(1BM22CS305)

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 TARANG **(1BM22CS305)**, 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 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. Lakshmi Neelima	**Dr. Jyothi S Nayak**
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 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!!!
```

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)

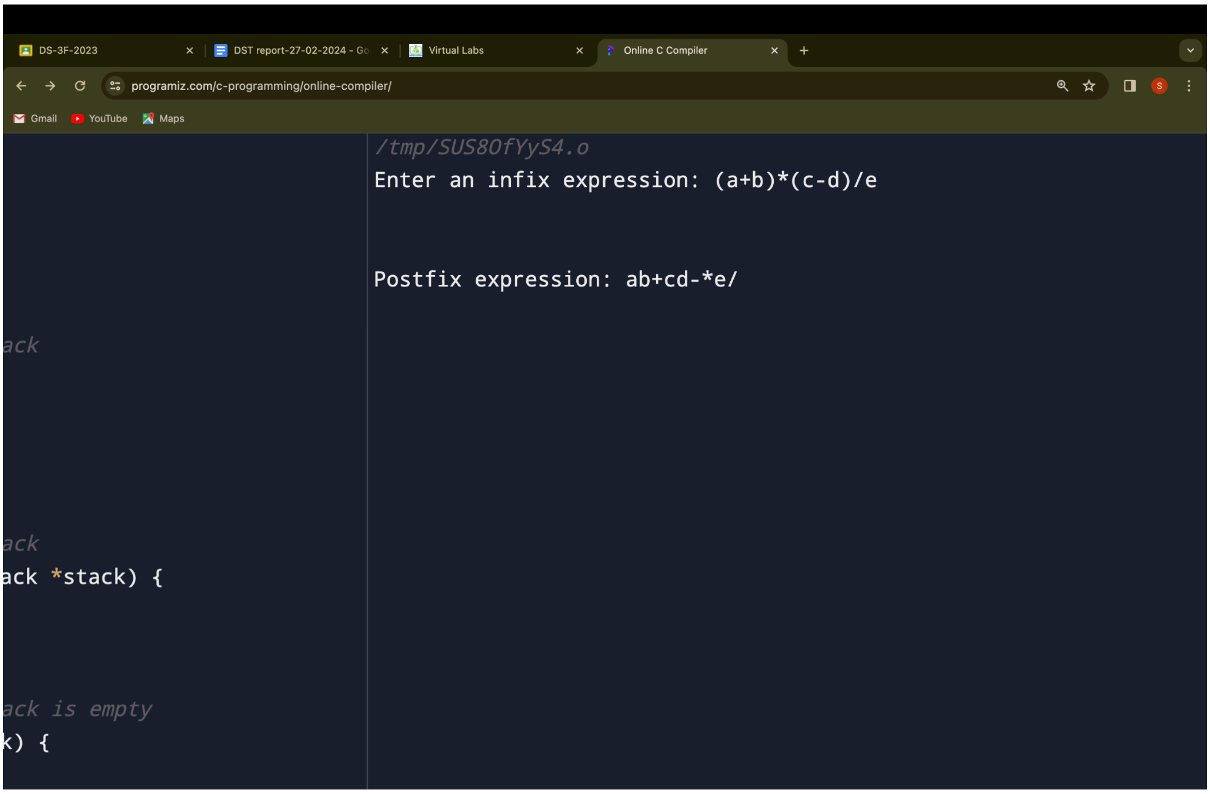
and /(divide)

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

#define MAX_SIZE 100

// Structure to represent a stack
struct Stack {
    int top;
    char items[MAX_SIZE];
};

// Function to initialize a stack
void initializeStack(struct Stack *stack) {
    stack->top = -1;
}

// Function to check if the stack is empty
int isEmpty(struct Stack *stack) {
    return stack->top == -1;
}

// Function to push an element onto the stack
void push(struct Stack *stack, char value) {
    if (stack->top < MAX_SIZE - 1) {
        stack->items[++(stack->top)] = value;
    } else {
        printf("Stack overflow\n");
        exit(EXIT_FAILURE);
    }
}
```

```
// Function to pop an element from the stack
char pop(struct Stack *stack) {
  if (!isEmpty(stack)) {
    return stack->items[(stack->top)--];
  } else {
    printf("Stack underflow\n");
    exit(EXIT_FAILURE);
  }
}

// Function to get the precedence of an operator
int getPrecedence(char operator) {
  switch (operator) {
    case '+':
    case '-':
      return 1;
    case '*':
    case '/':
      return 2;
    default:
      return -1; // Unknown operator
  }
}

// Function to convert infix expression to postfix expression
void infixToPostfix(char infix[], char postfix[]) {
  struct Stack operatorStack;
  initializeStack(&operatorStack);

  int i, j;
  i = j = 0;
```

```
    while (infix[i] != '\0') {
        if ((infix[i] >= 'a' && infix[i] <= 'z') || (infix[i] >= 'A' && infix[i] <= 'Z')) {
            postfix[j++] = infix[i++];
        } else if (infix[i] == '(') {
            push(&operatorStack, infix[i++]);
        } else if (infix[i] == ')') {
            while (!isEmpty(&operatorStack) && operatorStack.items[operatorStack.top] != '(') {
                postfix[j++] = pop(&operatorStack);
            }
            if (!isEmpty(&operatorStack) && operatorStack.items[operatorStack.top] == '(') {
                pop(&operatorStack); // Discard '('
                i++;
            } else {
                printf("Invalid expression: Mismatched parentheses\n");
                exit(EXIT_FAILURE);
            }
        } else {
            while (!isEmpty(&operatorStack) && getPrecedence(infix[i]) <=
getPrecedence(operatorStack.items[operatorStack.top])) {
                postfix[j++] = pop(&operatorStack);
            }
            push(&operatorStack, infix[i++]);
        }
    }

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

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

```c
int main() {
    char infix[MAX_SIZE];
    char postfix[MAX_SIZE];

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

    infixToPostfix(infix, postfix);

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

    return 0;
}
```



Lab program 3:

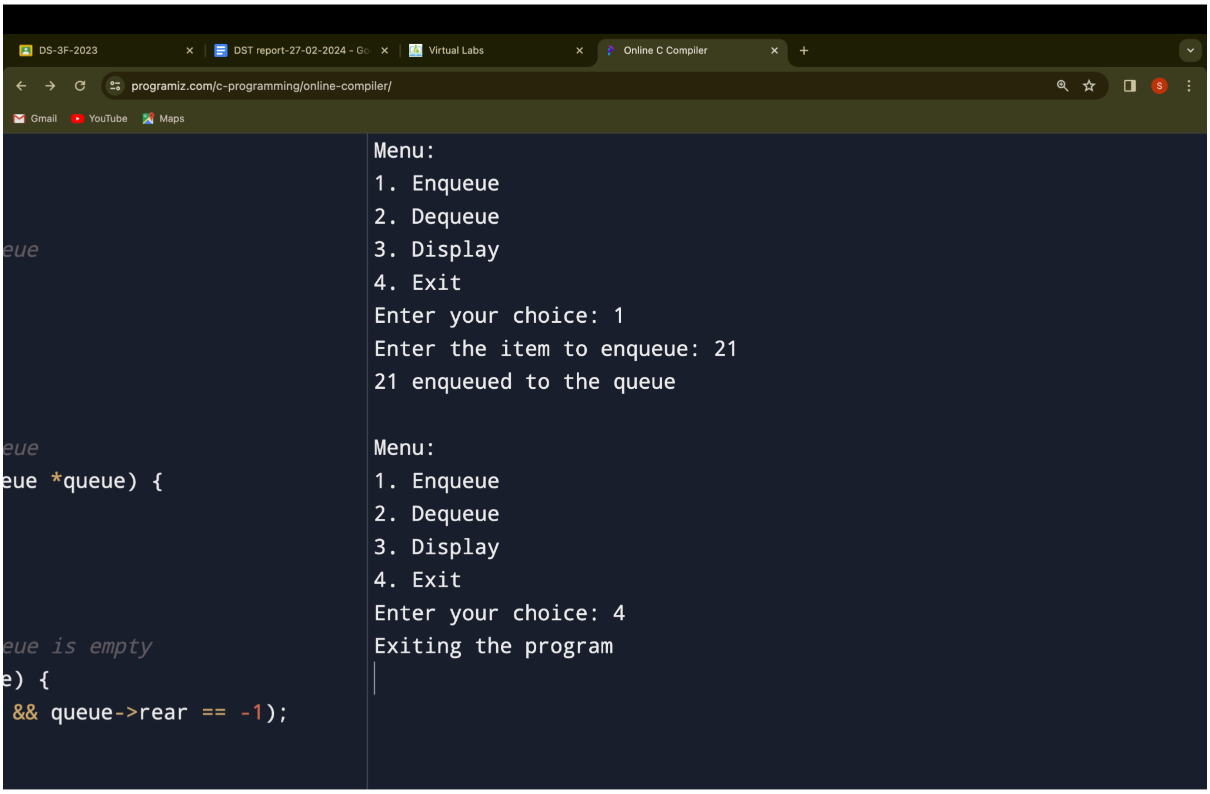
write a program to simulate the working of the queue of integers using an array. Provide the following operations: Insert, delete, display. The program should print appropriate message for overflow and underflow condition.

```
#include <stdio.h>

#include <stdlib.h>


#define MAX_SIZE 5


// Structure to represent a queue

struct Queue {

   int front, rear;

   int items[MAX_SIZE];

};


// Function to initialize a queue

void initializeQueue(struct Queue *queue) {

   queue->front = -1;

   queue->rear = -1;

}


// Function to check if the queue is empty

int isEmpty(struct Queue *queue) {

   return (queue->front == -1 && queue->rear == -1);

}


// Function to check if the queue is full

int isFull(struct Queue *queue) {

   return ((queue->rear + 1) % MAX_SIZE == queue->front);

}


// Function to insert an element into the queue
```

```
void enqueue(struct Queue *queue, int item) {
    if (isFull(queue)) {
        printf("Queue overflow. Cannot enqueue %d\n", item);
        return;
    }

    if (isEmpty(queue)) {
        queue->front = queue->rear = 0;
    } else {
        queue->rear = (queue->rear + 1) % MAX_SIZE;
    }

    queue->items[queue->rear] = item;
    printf("%d enqueued to the queue\n", item);
}

// Function to delete an element from the queue
int dequeue(struct Queue *queue) {
    int item;

    if (isEmpty(queue)) {
        printf("Queue underflow. Cannot dequeue.\n");
        return -1; // Return a special value to indicate underflow
    }

    item = queue->items[queue->front];

    if (queue->front == queue->rear) {
        // Reset front and rear as it's the last element in the queue
        queue->front = queue->rear = -1;
    } else {
        queue->front = (queue->front + 1) % MAX_SIZE;
```

```
    }

    return item;
}

// Function to display the elements of the queue
void display(struct Queue *queue) {
    if (isEmpty(queue)) {
        printf("Queue is empty\n");
        return;
    }

    printf("Queue elements: ");
    int i = queue->front;
    do {
        printf("%d ", queue->items[i]);
        i = (i + 1) % MAX_SIZE;
    } while (i != (queue->rear + 1) % MAX_SIZE);

    printf("\n");
}

int main() {
    struct Queue myQueue;
    initializeQueue(&myQueue);

    int choice, item;

    do {
        printf("\nMenu:\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 the item to enqueue: ");
                scanf("%d", &item);
                enqueue(&myQueue, item);
                break;
            case 2:
                item = dequeue(&myQueue);
                if (item != -1) {
                    printf("Dequeued item: %d\n", item);
                }
                break;
            case 3:
                display(&myQueue);
                break;
            case 4:
                printf("Exiting the program\n");
                break;
            default:
                printf("Invalid choice. Please enter a valid option.\n");
        }

    } while (choice != 4);

    return 0;
}
```



Lab program 4:

write a program to simulate the working of a circular queue using an array. Provide the following operations: insert, delete& display. The program should print appropriate message for queue empty and queue overflow conditions.

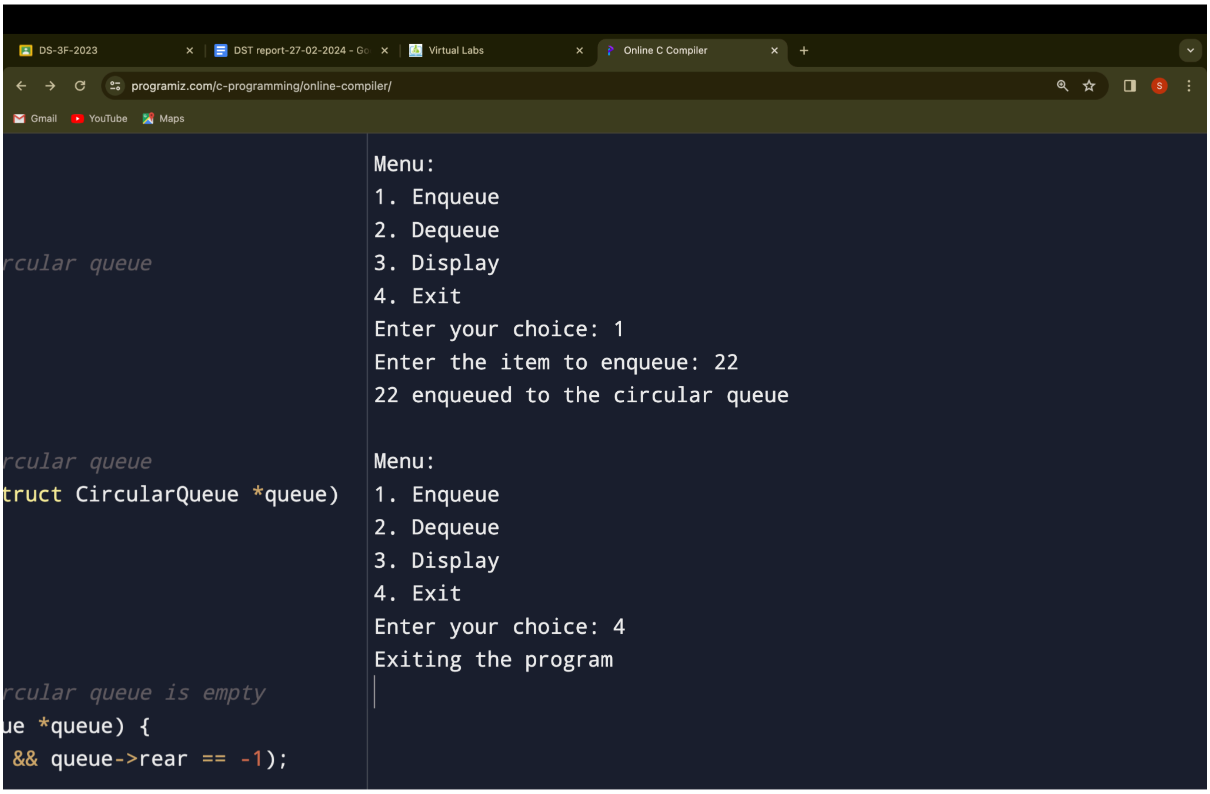
```
#include <stdio.h>

#include <stdlib.h>



#define MAX_SIZE 5



// Structure to represent a circular queue

struct CircularQueue {

    int front, rear;

    int items[MAX_SIZE];

};



// Function to initialize a circular queue

void initializeCircularQueue(struct CircularQueue *queue) {

    queue->front = -1;
```

```
    queue->rear = -1;
}

// Function to check if the circular queue is empty
int isEmpty(struct CircularQueue *queue) {
    return (queue->front == -1 && queue->rear == -1);
}

// Function to check if the circular queue is full
int isFull(struct CircularQueue *queue) {
    return ((queue->rear + 1) % MAX_SIZE == queue->front);
}

// Function to insert an element into the circular queue
void enqueue(struct CircularQueue *queue, int item) {
    if (isFull(queue)) {
        printf("Queue overflow. Cannot enqueue %d\n", item);
        return;
    }

    if (isEmpty(queue)) {
        queue->front = queue->rear = 0;
    } else {
        queue->rear = (queue->rear + 1) % MAX_SIZE;
    }

    queue->items[queue->rear] = item;
    printf("%d enqueued to the circular queue\n", item);
}

// Function to delete an element from the circular queue
int dequeue(struct CircularQueue *queue) {
```

```
    int item;

    if (isEmpty(queue)) {
        printf("Queue underflow. Cannot dequeue.\n");
        return -1; // Return a special value to indicate underflow
    }

    item = queue->items[queue->front];

    if (queue->front == queue->rear) {
        // Reset front and rear as it's the last element in the circular queue
        queue->front = queue->rear = -1;
    } else {
        queue->front = (queue->front + 1) % MAX_SIZE;
    }

    return item;
}

// Function to display the elements of the circular queue
void display(struct CircularQueue *queue) {
    if (isEmpty(queue)) {
        printf("Circular queue is empty\n");
        return;
    }

    printf("Circular Queue elements: ");
    int i = queue->front;
    do {
        printf("%d ", queue->items[i]);
        i = (i + 1) % MAX_SIZE;
    } while (i != (queue->rear + 1) % MAX_SIZE);
```

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

int main() {
    struct CircularQueue myCircularQueue;
    initializeCircularQueue(&myCircularQueue);

    int choice, item;

    do {
        printf("\nMenu:\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 the item to enqueue: ");
                scanf("%d", &item);
                enqueue(&myCircularQueue, item);
                break;
            case 2:
                item = dequeue(&myCircularQueue);
                if (item != -1) {
                    printf("Dequeued item: %d\n", item);
                }
                break;
```

```
            case 3:
                display(&myCircularQueue);
                break;
            case 4:
                printf("Exiting the program\n");
                break;
            default:
                printf("Invalid choice. Please enter a valid option.\n");
        }

    } while (choice != 4);

    return 0;
}
```



Lab program 5:

WAP to Implement Singly Linked List with following operations.

a)

Create a

linked list.

b)

Insertion of a node at first position, at any

position and at end of list.

Display the contents of the linked

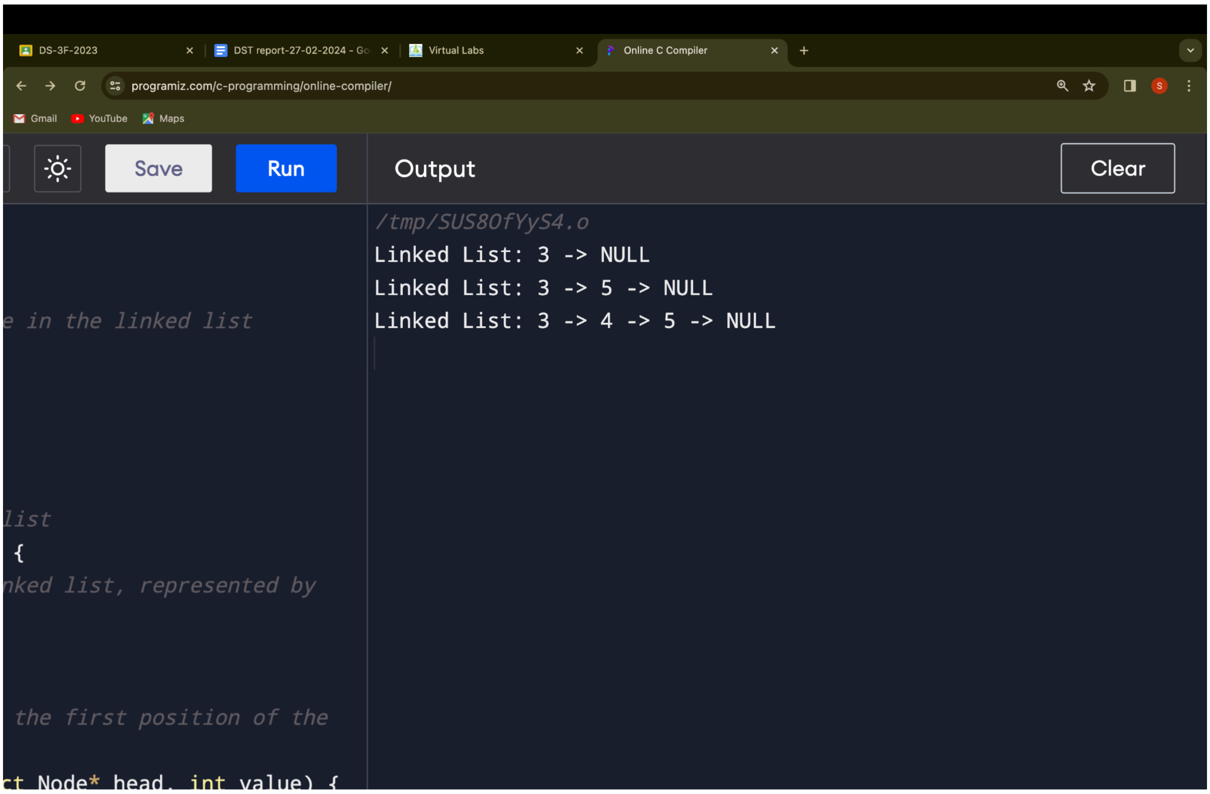
list.

```
#include <stdio.h>

#include <stdlib.h>

// Structure to represent a node in the linked list

struct Node {

   int data;

   struct Node* next;

};

// Function to create a linked list

struct Node* createLinkedList() {

   return NULL; // An empty linked list, represented by NULL

}

// Function to insert a node at the first position of the linked list
```

```c
struct Node* insertAtFirst(struct Node* head, int value) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = value;
   newNode->next = head;
   return newNode;
}

// Function to insert a node at any position of the linked list
struct Node* insertAtPosition(struct Node* head, int value, int position) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = value;

   if (position == 1) {
      newNode->next = head;
      return newNode;
   }

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

   if (current == NULL) {
      printf("Invalid position. Cannot insert at position %d\n", position);
      return head;
   }

   newNode->next = current->next;
   current->next = newNode;
   return head;
}
```

```
// Function to insert a node at the end of the linked list
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 == NULL) {
        return newNode;
    }

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

    current->next = newNode;
    return head;
}

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

int main() {
    struct Node* linkedList = createLinkedList();
```

```
    // Insert at first position
    linkedList = insertAtFirst(linkedList, 3);
    displayLinkedList(linkedList);

    // Insert at end
    linkedList = insertAtEnd(linkedList, 5);
    displayLinkedList(linkedList);

    // Insert at any position
    linkedList = insertAtPosition(linkedList, 4, 2);
    displayLinkedList(linkedList);

    return 0;
}
```



Lab program 6:

WAP to Implement Singly Linked List with following operations.

a)

Create a linked

list.

b)

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

Display the contents of the linked

list.

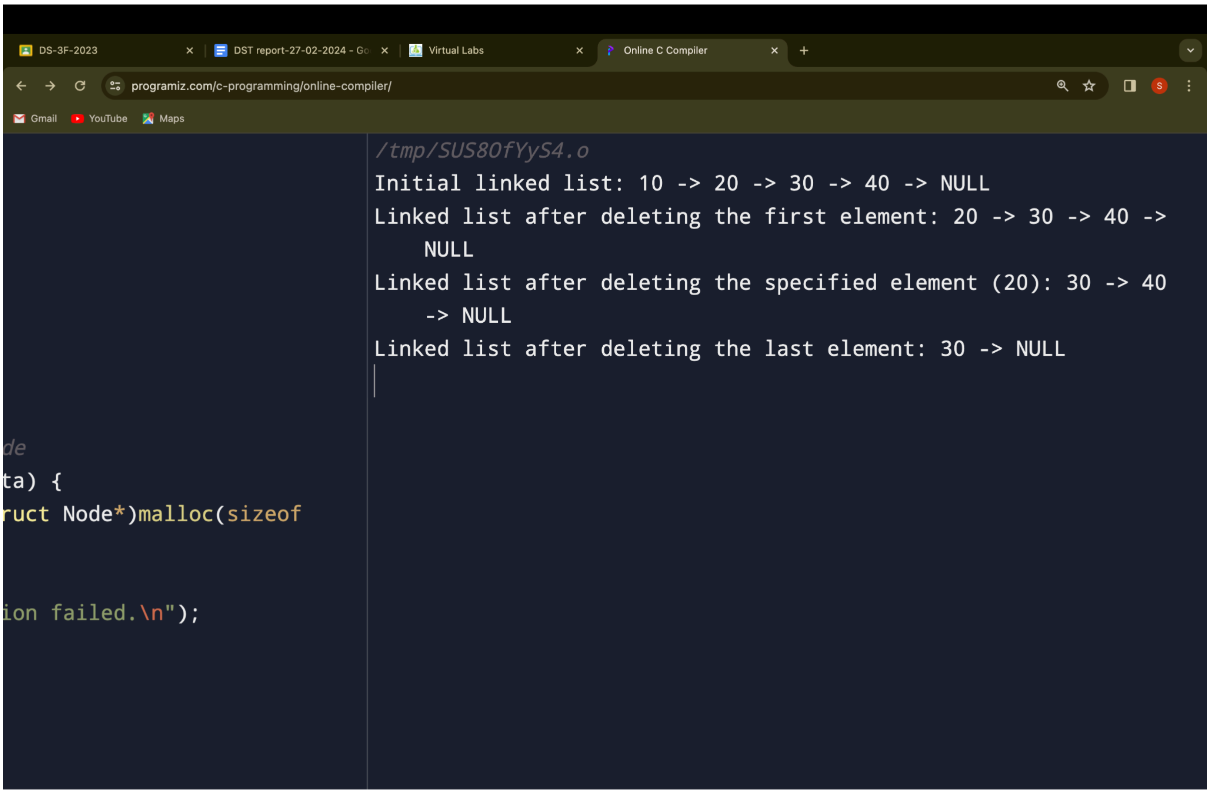
```
#include <stdio.h>

#include <stdlib.h>


// Node structure

struct Node {

    int data;

    struct Node* next;

};


// Function to create a new node

struct Node* createNode(int data) {

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

    if (newNode == NULL) {

        printf("Memory allocation failed.\n");

        exit(1);

    }
```

```
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

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

// Function to delete the first node in the list
void deleteFirst(struct Node** head) {
    if (*head == NULL) {
        printf("List is empty. Cannot delete.\n");
        return;
    }
    struct Node* temp = *head;
    *head = (*head)->next;
    free(temp);
}
```

```
// Function to delete a specified node in the list
void deleteNode(struct Node** head, int key) {
    if (*head == NULL) {
        printf("List is empty. Cannot delete.\n");
        return;
    }
    struct Node* temp = *head;
    struct Node* prev = NULL;
    while (temp != NULL && temp->data != key) {
        prev = temp;
        temp = temp->next;
    }
    if (temp == NULL) {
        printf("Element not found in the list.\n");
        return;
    }
    if (prev == NULL) {
        *head = temp->next;
    } else {
        prev->next = temp->next;
    }
    free(temp);
}

// Function to delete the last node in the list
void deleteLast(struct Node** head) {
    if (*head == NULL) {
        printf("List is empty. Cannot delete.\n");
```

```
        return;
    }
    if ((*head)->next == NULL) {
        free(*head);
        *head = NULL;
        return;
    }
    struct Node* temp = *head;
    struct Node* prev = NULL;
    while (temp->next != NULL) {
        prev = temp;
        temp = temp->next;
    }
    prev->next = NULL;
    free(temp);
}

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

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

```
    // Create a linked list
    insertEnd(&head, 10);
    insertEnd(&head, 20);
    insertEnd(&head, 30);
    insertEnd(&head, 40);

    // Display the initial list
    printf("Initial linked list: ");
    displayList(head);

    // Delete the first element
    deleteFirst(&head);
    printf("Linked list after deleting the first element: ");
    displayList(head);

    // Delete a specified element (e.g., 20)
    deleteNode(&head, 20);
    printf("Linked list after deleting the specified element (20): ");
    displayList(head);

    // Delete the last element
    deleteLast(&head);
    printf("Linked list after deleting the last element: ");
    displayList(head);

    return 0;
}
```



Lab program 7:

sll-sort,reverse,concatination

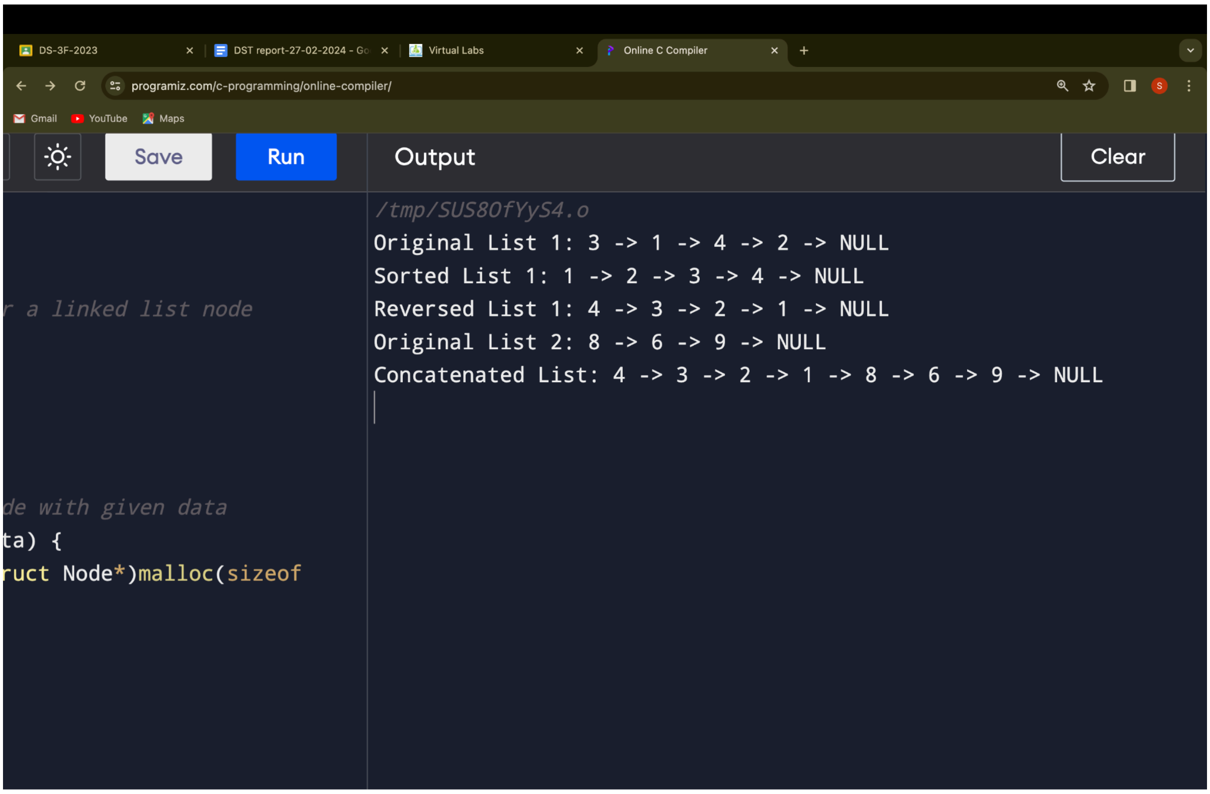
```
#include <stdio.h>

#include <stdlib.h>


// Define a basic structure for a linked list node

struct Node {

    int data;

    struct Node* next;

};


// Function to create a new node with given data

struct Node* createNode(int data) {

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

    newNode->data = data;

    newNode->next = NULL;

    return newNode;
```

```c
}

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

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

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

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

    return prev;
}

// Function to merge two linked lists
struct Node* concatenateLists(struct Node* list1, struct Node* list2) {
    if (list1 == NULL) {
        return list2;
    }

    struct Node* temp = list1;
    while (temp->next != NULL) {
        temp = temp->next;
    }

    temp->next = list2;
    return list1;
}

// Function to perform a bubble sort on the linked list
void bubbleSort(struct Node* head) {
    int swapped;
    struct Node* ptr1;
    struct Node* lptr = NULL;

    // Checking for empty list
```

```
    if (head == NULL) {
        return;
    }

    do {
        swapped = 0;
        ptr1 = head;

        while (ptr1->next != lptr) {
            if (ptr1->data > ptr1->next->data) {
                // Swap the data of the nodes
                int temp = ptr1->data;
                ptr1->data = ptr1->next->data;
                ptr1->next->data = temp;

                swapped = 1;
            }
            ptr1 = ptr1->next;
        }
        lptr = ptr1;
    } while (swapped);
}

int main() {
    // Creating and displaying the first linked list
    struct Node* list1 = NULL;
    insertAtEnd(&list1, 3);
    insertAtEnd(&list1, 1);
    insertAtEnd(&list1, 4);
    insertAtEnd(&list1, 2);
    printf("Original List 1: ");
    printList(list1);
```

```
    // Sorting the first linked list
    bubbleSort(list1);
    printf("Sorted List 1: ");
    printList(list1);

    // Reversing the first linked list
    list1 = reverseList(list1);
    printf("Reversed List 1: ");
    printList(list1);

    // Creating and displaying the second linked list
    struct Node* list2 = NULL;
    insertAtEnd(&list2, 8);
    insertAtEnd(&list2, 6);
    insertAtEnd(&list2, 9);
    printf("Original List 2: ");
    printList(list2);

    // Concatenating the two linked lists
    list1 = concatenateLists(list1, list2);
    printf("Concatenated List: ");
    printList(list1);

    return 0;
}
```



Lab program 8:

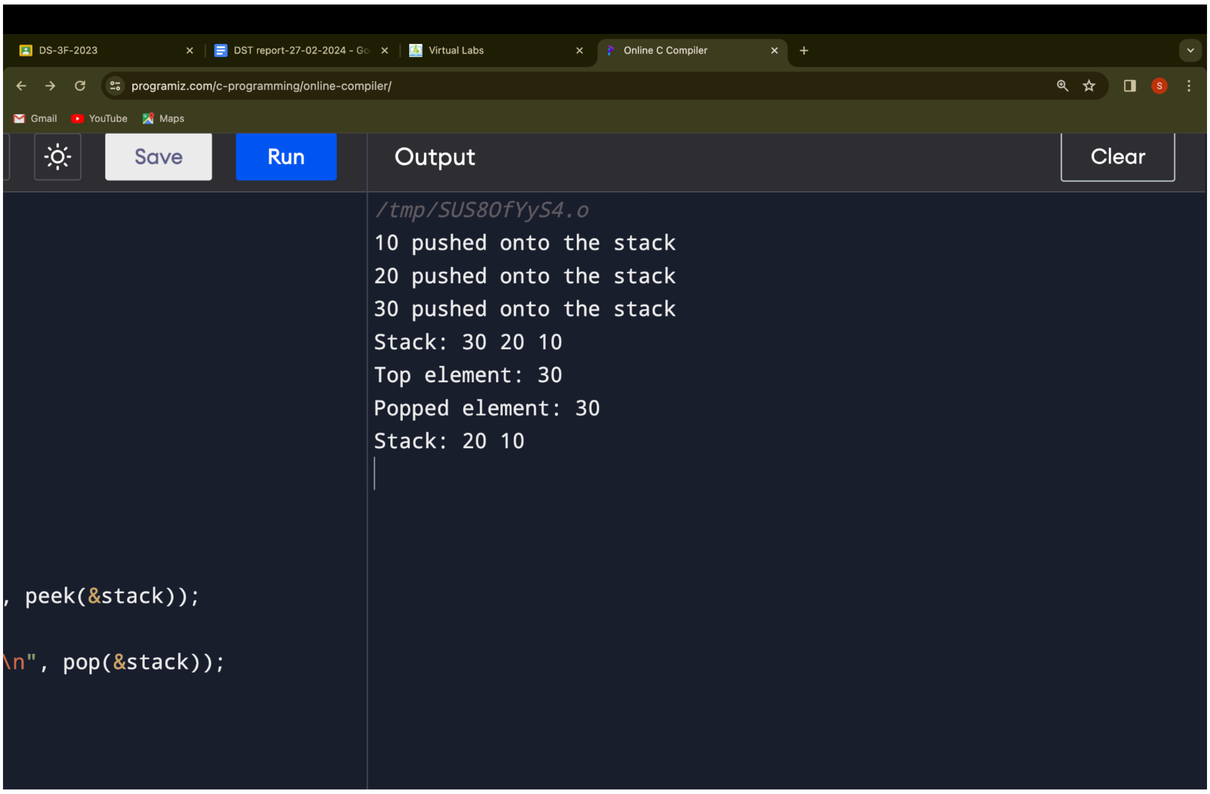
.Stack implementation using single linked list

```
#include <stdio.h>

#include <stdlib.h>


// Node structure for the linked list

typedef struct Node {

    int data;

    struct Node* next;

} Node;


// Stack structure

typedef struct Stack {

    Node* top;

} Stack;


// Function to initialize an empty stack

void initialize(Stack* stack) {
```

```
    stack->top = NULL;
}

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

// Function to push an element onto the stack
void push(Stack* stack, int data) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    if (newNode == NULL) {
        printf("Memory allocation failed\n");
        return;
    }

    newNode->data = data;
    newNode->next = stack->top;
    stack->top = newNode;

    printf("%d pushed onto the stack\n", data);
}

// Function to pop an element from the stack
int pop(Stack* stack) {
    if (isEmpty(stack)) {
        printf("Stack underflow\n");
        return -1; // Return an invalid value indicating stack underflow
    }

    Node* temp = stack->top;
    int poppedData = temp->data;
```

```
    stack->top = temp->next;
    free(temp);

    return poppedData;
}

// Function to get the top element of the stack without removing it
int peek(Stack* stack) {
    if (isEmpty(stack)) {
        printf("Stack is empty\n");
        return -1; // Return an invalid value indicating an empty stack
    }

    return stack->top->data;
}

// Function to display the elements of the stack
void display(Stack* stack) {
    if (isEmpty(stack)) {
        printf("Stack is empty\n");
        return;
    }

    Node* current = stack->top;
    printf("Stack: ");
    while (current != NULL) {
        printf("%d ", current->data);
        current = current->next;
    }
    printf("\n");
}
```

```c
// Function to free the memory used by the stack
void destroy(Stack* stack) {
    while (!isEmpty(stack)) {
        pop(stack);
    }
}

int main() {
    Stack stack;
    initialize(&stack);

    push(&stack, 10);
    push(&stack, 20);
    push(&stack, 30);

    display(&stack);

    printf("Top element: %d\n", peek(&stack));

    printf("Popped element: %d\n", pop(&stack));
    display(&stack);

    destroy(&stack);

    return 0;
}
```



Lab program 9:

.Queue implementation using single linked list

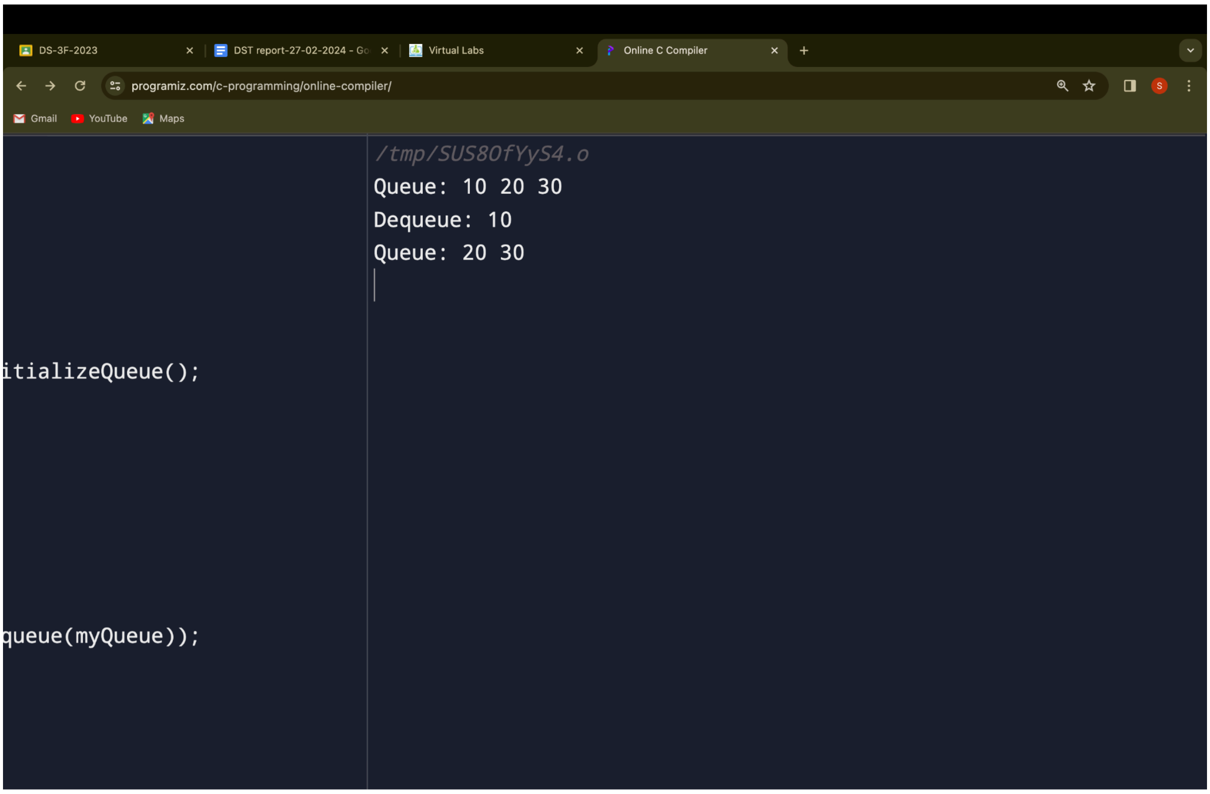
```
#include <stdio.h>

#include <stdlib.h>


// Node structure

struct Node {

    int data;

    struct Node* next;

};


// Queue structure

struct Queue {

    struct Node* front;

    struct Node* rear;

};


// Function to create a new node
```

```
struct Node* createNode(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation failed.\n");
        exit(EXIT_FAILURE);
    }
    newNode->data = value;
    newNode->next = NULL;
    return newNode;
}

// Function to initialize a queue
struct Queue* initializeQueue() {
    struct Queue* queue = (struct Queue*)malloc(sizeof(struct Queue));
    if (queue == NULL) {
        printf("Memory allocation failed.\n");
        exit(EXIT_FAILURE);
    }
    queue->front = queue->rear = NULL;
    return queue;
}

// Function to check if the queue is empty
int isEmpty(struct Queue* queue) {
    return (queue->front == NULL);
}

// Function to enqueue (insert) an element into the queue
void enqueue(struct Queue* queue, int value) {
    struct Node* newNode = createNode(value);
    if (isEmpty(queue)) {
        queue->front = queue->rear = newNode;
```

```
  } else {
    queue->rear->next = newNode;
    queue->rear = newNode;
  }
}

// Function to dequeue (remove) an element from the queue
int dequeue(struct Queue* queue) {
  if (isEmpty(queue)) {
    printf("Queue is empty. Cannot dequeue.\n");
    exit(EXIT_FAILURE);
  }

  struct Node* temp = queue->front;
  int value = temp->data;

  queue->front = temp->next;
  free(temp);

  if (queue->front == NULL) {
    queue->rear = NULL; // If the last element is dequeued, update the rear pointer
  }

  return value;
}

// Function to display the elements in the queue
void displayQueue(struct Queue* queue) {
  if (isEmpty(queue)) {
    printf("Queue is empty.\n");
    return;
  }
```

```
    struct Node* current = queue->front;
    printf("Queue: ");
    while (current != NULL) {
        printf("%d ", current->data);
        current = current->next;
    }
    printf("\n");
}

// Function to free the allocated memory for the queue
void freeQueue(struct Queue* queue) {
    while (!isEmpty(queue)) {
        dequeue(queue);
    }
    free(queue);
}

int main() {
    struct Queue* myQueue = initializeQueue();

    enqueue(myQueue, 10);
    enqueue(myQueue, 20);
    enqueue(myQueue, 30);

    displayQueue(myQueue);

    printf("Dequeue: %d\n", dequeue(myQueue));

    displayQueue(myQueue);

    freeQueue(myQueue);
```

```
    return 0;
}
```



Lab program 10:

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**

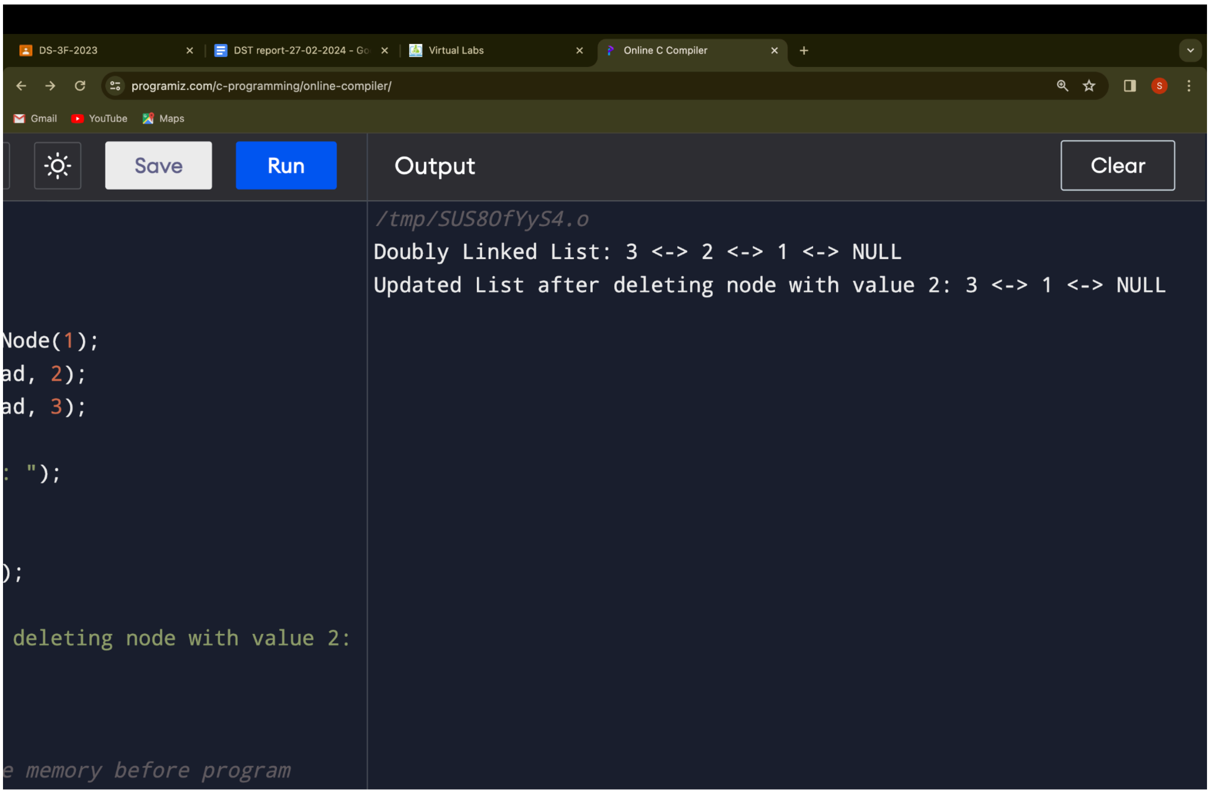
```
#include <stdio.h>
#include <stdlib.h>

// Node structure for doubly linked list
struct Node {
    int data;
    struct Node* prev;
```

```c
    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 new node to the left of the given node
void insertNodeToLeft(struct Node** head, struct Node* target, int data) {
    struct Node* newNode = createNode(data);

    // If the target node is the head
    if (*head == target) {
        newNode->next = *head;
        (*head)->prev = newNode;
        *head = newNode;
    } else {
        newNode->next = target;
        newNode->prev = target->prev;
        target->prev->next = newNode;
        target->prev = newNode;
    }
}
```

```
// Function to delete a node based on a specific value
void deleteNodeByValue(struct Node** head, int value) {
  struct Node* current = *head;

  // Traverse the list to find the node with the given value
  while (current != NULL && current->data != value) {
    current = current->next;
  }

  // If the node with the given value is found
  if (current != NULL) {
    // If the node is the head
    if (current->prev == NULL) {
      *head = current->next;
      if (*head != NULL) {
        (*head)->prev = NULL;
      }
    } else {
      current->prev->next = current->next;
      if (current->next != NULL) {
        current->next->prev = current->prev;
      }
    }

    // Free the memory occupied by the deleted node
    free(current);
  } else {
    printf("Node with value %d not found.\n", value);
  }
}
```

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

// Function to free the memory occupied by the doubly linked list
void freeList(struct Node* head) {
    struct Node* temp;
    while (head != NULL) {
        temp = head;
        head = head->next;
        free(temp);
    }
}

int main() {
    struct Node* head = createNode(1);
    insertNodeToLeft(&head, head, 2);
    insertNodeToLeft(&head, head, 3);

    printf("Doubly Linked List: ");
    printList(head);

    deleteNodeByValue(&head, 2);

    printf("Updated List after deleting node with value 2: ");
    printList(head);
```

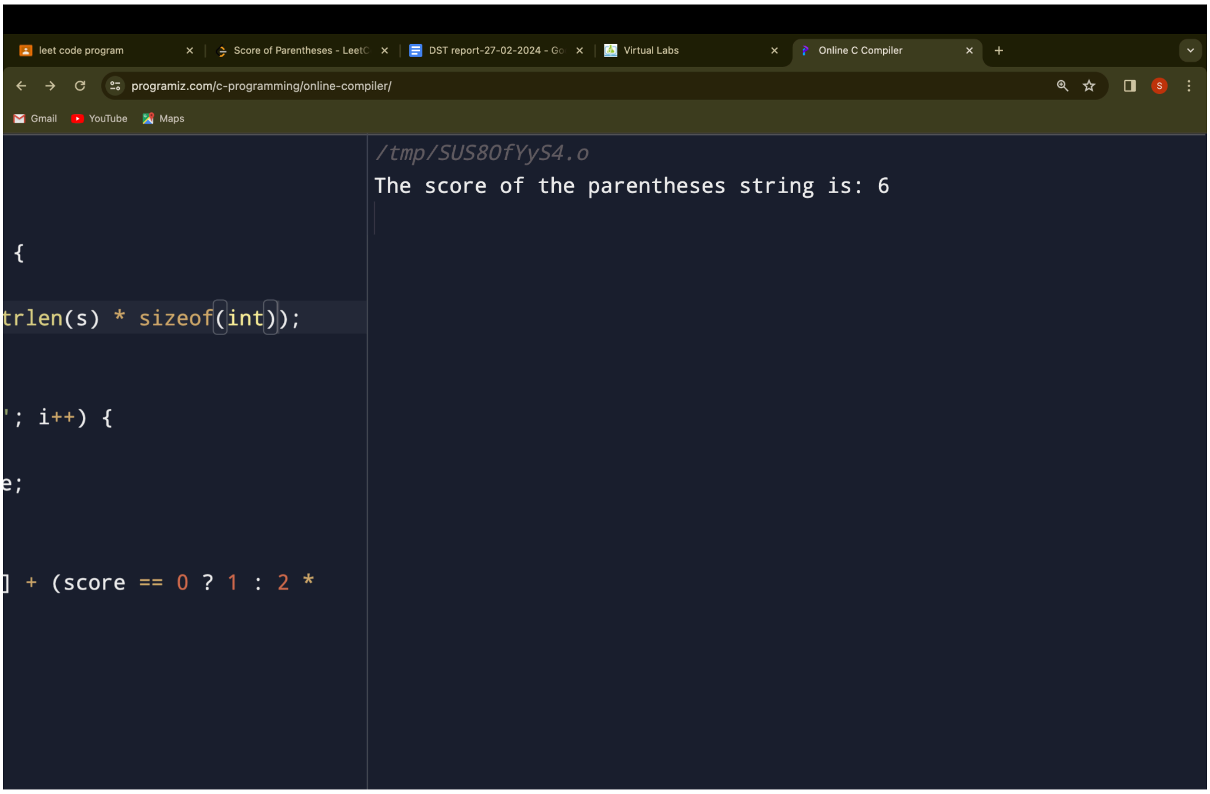
```
    freeList(head); // Free the memory before program termination

    return 0;

}
```



Lab program 11:

```
#include <stdio.h>
#include <stdlib.h>
int scoreOfParentheses(char* s) {
    int score = 0;
    int* stack = (int*)malloc(strlen(s) * sizeof(int));
    int top = -1;
    for (int i = 0; s[i] != '\0'; i++) {
        if (s[i] == '(') {
            stack[++top] = score;
            score = 0;
        } else {
            score = stack[top--] + (score == 0 ? 1 : 2
* score);
        }
    }
```

```
    free(stack);
    return score;
}
int main() {
    char input[] = "(()(()))";
    int result = scoreOfParentheses(input);
    printf("The score of the parentheses string is:
%d\n", result);
    return 0;
}
```



Lab program 12:

```
struct ListNode* oddEvenList(struct ListNode* head) {
   if (head == NULL || head->next == NULL) {
      return head;
   }

   struct ListNode* odd = head;
   struct ListNode* even = head->next;
```

```
    struct ListNode* evenHead = even;

    while (even != NULL && even->next != NULL) {
        odd->next = even->next;
        odd = odd->next;
        even->next = odd->next;
        even = even->next;
    }

    odd->next = evenHead;

    return head;
}
```

Lab program 13:

```
struct ListNode* deleteMiddle(struct ListNode* head) {
    struct ListNode *ptr,*temp;
    int count=0,n=0;
    ptr=head;
    while(ptr!=NULL){
        n++;
        ptr=ptr->next;
    }
    ptr=head;
    if(n==1) {
        return 0;
    }
    while(count!=n/2) {
        temp=ptr;
        count++;
```

```
        ptr=ptr->next;
    }
    temp->next=ptr->next;
    ptr=NULL;
    free(ptr);
    return head;

}
```

Lab program 14:

Write a program.

a. To construct Binary Search tree

b. Traverse the tree using inorder , postorder, preorder.

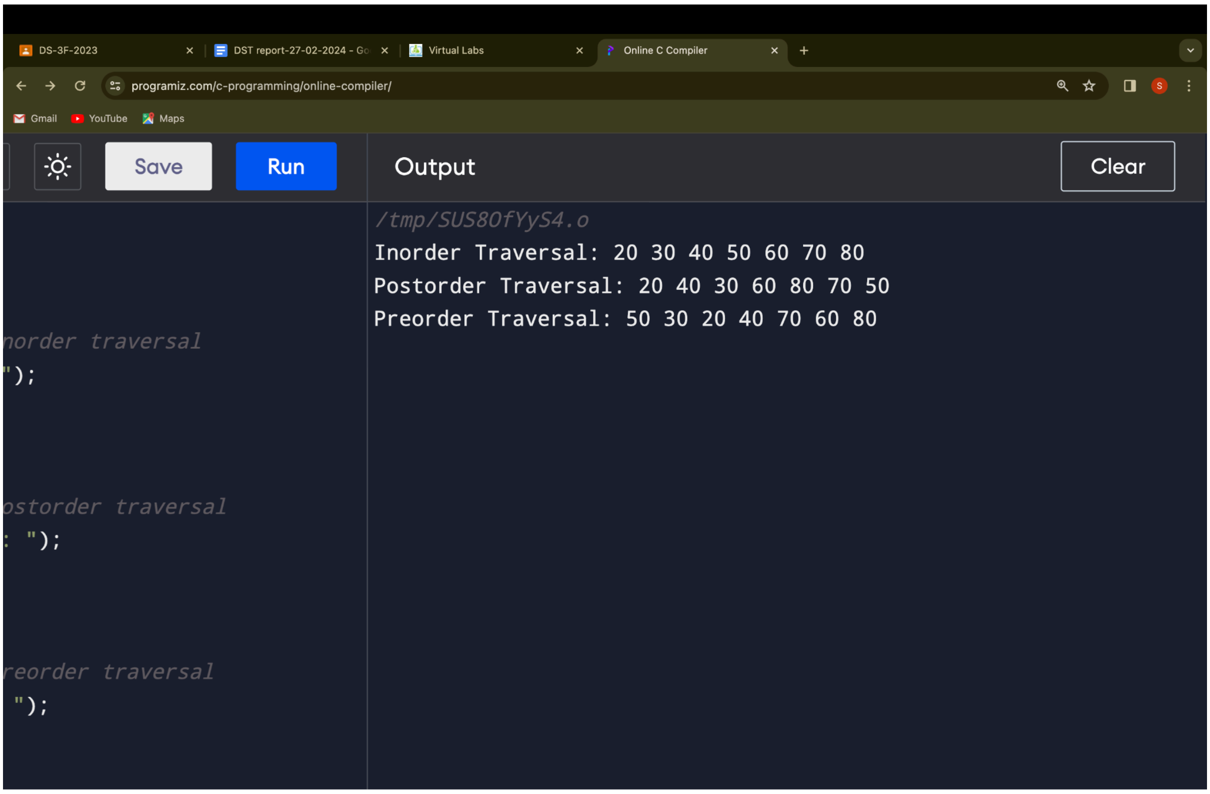
c. Display the elements in the tree.

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

// Node structure for the Binary Search Tree
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

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

```c
// Function to insert a new node into the Binary Search Tree
struct Node* insertNode(struct Node* root, int data) {
   if (root == NULL)
      return createNode(data);

   if (data < root->data)
      root->left = insertNode(root->left, data);
   else if (data > root->data)
      root->right = insertNode(root->right, data);

   return root;
}

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

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

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

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

  // Insert elements into the BST
  root = insertNode(root, 50);
  insertNode(root, 30);
  insertNode(root, 20);
  insertNode(root, 40);
  insertNode(root, 70);
  insertNode(root, 60);
  insertNode(root, 80);

  // Display elements using inorder traversal
  printf("Inorder Traversal: ");
  inorderTraversal(root);
  printf("\n");

  // Display elements using postorder traversal
  printf("Postorder Traversal: ");
  postorderTraversal(root);
  printf("\n");

  // Display elements using preorder traversal
```

```
    printf("Preorder Traversal: ");
    preorderTraversal(root);
    printf("\n");

    return 0;
}
```



Lab program 15:

1.BFS

2.DFS

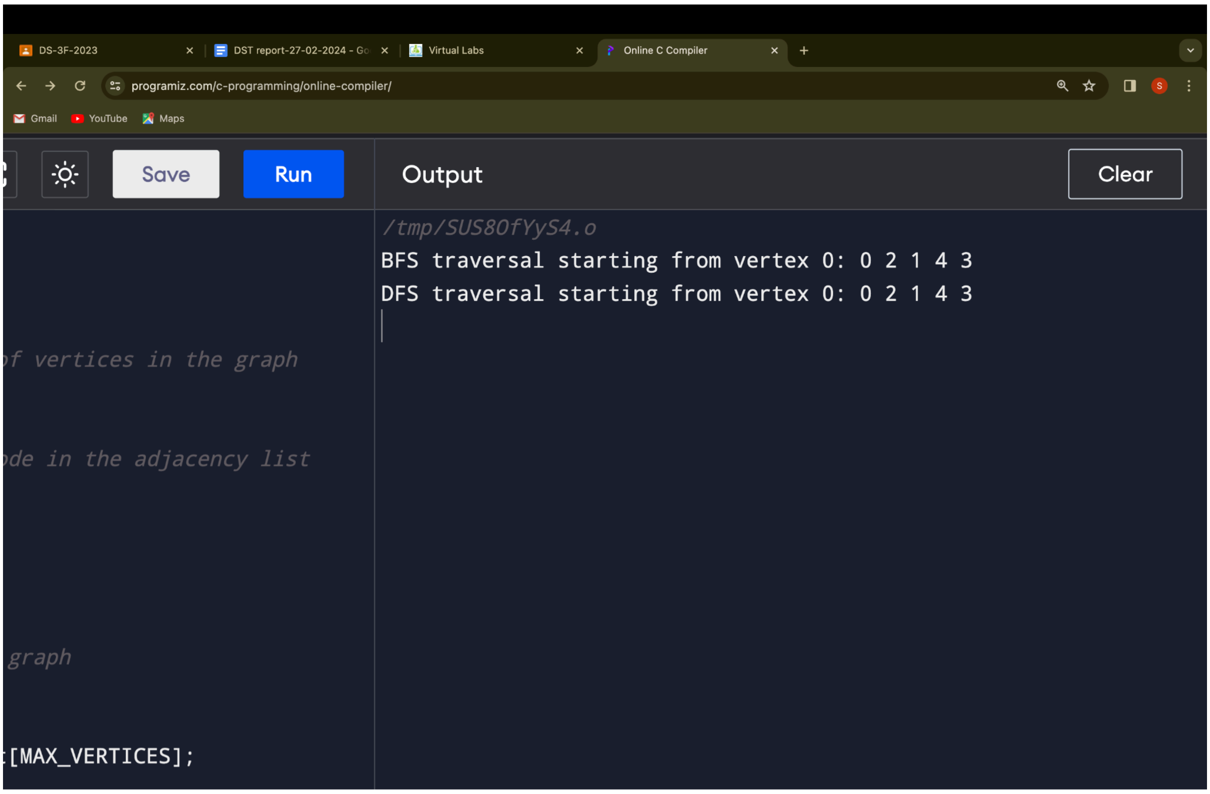
```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

// Define the maximum number of vertices in the graph
#define MAX_VERTICES 100

// Structure to represent a node in the adjacency list
```

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

// Structure to represent the graph
struct Graph {
    int numVertices;
    struct Node* adjacencyList[MAX_VERTICES];
};

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

// Function to create a graph with the given number of vertices
struct Graph* createGraph(int numVertices) {
    struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
    graph->numVertices = numVertices;

    // Initialize adjacency list
    for (int i = 0; i < numVertices; ++i) {
        graph->adjacencyList[i] = NULL;
    }

    return graph;
}
```

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

   // Add edge from dest to src (assuming undirected graph)
   newNode = createNode(src);
   newNode->next = graph->adjacencyList[dest];
   graph->adjacencyList[dest] = newNode;
}

// BFS traversal of the graph
void BFS(struct Graph* graph, int startVertex) {
   // Array to keep track of visited vertices
   bool visited[MAX_VERTICES] = {false};

   // Queue for BFS
   int queue[MAX_VERTICES];
   int front = 0, rear = 0;

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

   // BFS loop
   while (front < rear) {
      // Dequeue a vertex from the queue and print it
      int currentVertex = queue[front++];
      printf("%d ", currentVertex);
```

```
        // Explore adjacent vertices
        struct Node* temp = graph->adjacencyList[currentVertex];
        while (temp != NULL) {
            int adjVertex = temp->data;
            if (!visited[adjVertex]) {
                visited[adjVertex] = true;
                queue[rear++] = adjVertex;
            }
            temp = temp->next;
        }
    }
}

// DFS traversal of the graph
void DFSUtil(struct Graph* graph, int currentVertex, bool visited[]) {
    // Mark the current vertex as visited and print it
    visited[currentVertex] = true;
    printf("%d ", currentVertex);

    // Recur for all the adjacent vertices
    struct Node* temp = graph->adjacencyList[currentVertex];
    while (temp != NULL) {
        int adjVertex = temp->data;
        if (!visited[adjVertex]) {
            DFSUtil(graph, adjVertex, visited);
        }
        temp = temp->next;
    }
}

void DFS(struct Graph* graph, int startVertex) {
    // Array to keep track of visited vertices
```

```
    bool visited[MAX_VERTICES] = {false};

    // Call the DFS utility function
    DFSUtil(graph, startVertex, visited);
}

// Driver program
int main() {
    // Create a graph with 5 vertices
    struct Graph* graph = createGraph(5);

    // Add edges
    addEdge(graph, 0, 1);
    addEdge(graph, 0, 2);
    addEdge(graph, 1, 3);
    addEdge(graph, 1, 4);

    // Print BFS traversal starting from vertex 0
    printf("BFS traversal starting from vertex 0: ");
    BFS(graph, 0);
    printf("\n");

    // Reset visited array for DFS
    bool visited[MAX_VERTICES] = {false};

    // Print DFS traversal starting from vertex 0
    printf("DFS traversal starting from vertex 0: ");
    DFS(graph, 0);
    printf("\n");

    return 0;
}
```



Lab program 15:

Leetcode -Delete a node in BST

```
struct TreeNode* findMinValueNode(struct TreeNode* node) {
    while (node->left != NULL) {
        node = node->left;
    }
    return node;
}

struct TreeNode* deleteNode(struct TreeNode* root, int key) {
    struct TreeNode* current = root;
    struct TreeNode* parent = NULL;

    // Find the node to be deleted and its parent
    while (current != NULL && current->val != key) {
        parent = current;
        if (key < current->val) {
            current = current->left;
```

```
    } else {
        current = current->right;
    }
}

// If the key is not found
if (current == NULL) {
    return root;
}

// Case 1: Node with only one child or no child
if (current->left == NULL) {
    struct TreeNode* temp = current->right;
    if (parent == NULL) {
        // If the node to be deleted is the root
        free(current);
        return temp;
    } else {
        // Adjust the parent's pointer
        if (parent->left == current) {
            parent->left = temp;
        } else {
            parent->right = temp;
        }
        free(current);
        return root;
    }
} else if (current->right == NULL) {
    struct TreeNode* temp = current->left;
    if (parent == NULL) {
        // If the node to be deleted is the root
        free(current);
```

```
        return temp;
    } else {
        // Adjust the parent's pointer
        if (parent->left == current) {
            parent->left = temp;
        } else {
            parent->right = temp;
        }
        free(current);
        return root;
    }
}

// Case 3: Node with two children
struct TreeNode* minValueNode = findMinValueNode(current->right);
current->val = minValueNode->val;

// Delete the in-order successor
current->right = deleteNode(current->right, minValueNode->val);

return root;
}
```

Lab program 16:

```
void findBottomLeftValueHelper(struct TreeNode* root, int depth, int* maxDepth, int* result) {
    if (root == NULL) {
        return;
    }

    // Check if the current node is at a deeper level
    if (depth > *maxDepth) {
```

```
        *maxDepth = depth;
        *result = root->val;
    }

    // Recursive calls for the left and right subtrees
    findBottomLeftValueHelper(root->left, depth + 1, maxDepth, result);
    findBottomLeftValueHelper(root->right, depth + 1, maxDepth, result);
}

int findBottomLeftValue(struct TreeNode* root) {
   if (root == NULL) {
        return 0; // Or any suitable default value indicating an empty tree
    }

    int maxDepth = 0;
    int result = 0;

    findBottomLeftValueHelper(root, 1, &maxDepth, &result);

    return result;
}
```