VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
On

DATA STRUCTURES (23CS3PCDST)

Submitted by

RISHABH KUMAR (1BM22CS221)

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 RISHABH KUMAR**(1BM22CS221)**, 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

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 (Rishabh Kumar, 1BM22CS221)

Q) 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_SIZE 5

int stack[MAX_SIZE];
int top = -1;

void push(int value) {
	if (top == MAX_SIZE - 1) {
	printf("Stack Overflow: Cannot push %d, stack is full.\n", value);
	} else {
	top++;
	stack[top] = value;
	printf("Pushed %d onto the stack.\n", value);
	}
}
```

```c
void pop() {
    if (top == -1) {
    printf("Stack Underflow: Cannot pop, stack is empty.\n");
    } else {
    printf("Popped %d from the stack.\n", stack[top]);
    top--;
    }
}

void display() {
    if (top == -1) {
    printf("Stack is empty.\n");
    } else {
    printf("Stack elements: ");
    for (int i = 0; i <= top; i++) {
    printf("%d ", stack[i]);
    }
    printf("\n");
    }
}

int main() {
    push(10);
    push(20);
    push(30);
    display();

    pop();
    display();
```

```
        push(40);

        push(50);

        push(60); // This will cause a stack overflow

        display();


        pop();

        pop();

        pop();

        pop(); // This will cause a stack underflow

        display();


        return 0;

}
```

```
PS C:\Users\risku\coding> cd "c:\Users\risku\coding\" ; if ($?) { gcc rough.c -o rough } ; if ($?) { .\rough }
Pushed 10 onto the stack.
Pushed 20 onto the stack.
Pushed 30 onto the stack.
Stack elements: 10 20 30
Popped 30 from the stack.
Stack elements: 10 20
Pushed 40 onto the stack.
Pushed 50 onto the stack.
Pushed 60 onto the stack.
Stack elements: 10 20 40 50 60
Popped 60 from the stack.
Popped 50 from the stack.
Popped 40 from the stack.
Popped 20 from the stack.
Stack elements: 10
```

LAB Program – 2 (Rishabh Kumar, 1BM22CS221)

WAP to convert a given valid parenthesized infix arithmetic expression to postfix expression. The expression consists of single character operands and the binary operators + (plus), - (minus), * (multiply) and / (divide)

```
#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>


#define MAX_SIZE 50


char stack[MAX_SIZE];

int top = -1;


// Function to check if the given character is an operator

int isOperator(char ch) {

	return (ch == '+' || ch == '-' || ch == '*' || ch == '/');

}


// Function to get the precedence of an operator

int getPrecedence(char ch) {

	switch (ch) {

	case '+':

	case '-':

	return 1;

	case '*':

	case '/':

	return 2;

	default:

	return 0;

	}

}


// Function to push a character onto the stack
```

```
void push(char ch) {
	if (top == MAX_SIZE - 1) {
	printf("Stack Overflow: Cannot push %c, stack is full.\n", ch);
	exit(1);
	} else {
	top++;
	stack[top] = ch;
	}
}

// Function to pop a character from the stack
char pop() {
	if (top == -1) {
	printf("Stack Underflow: Cannot pop, stack is empty.\n");
	exit(1);
	} else {
	return stack[top--];
	}
}

// Function to convert infix to postfix
void infixToPostfix(char infix[]) {
	printf("Infix Expression: %s\n", infix);
	printf("Postfix Expression: ");

	for (int i = 0; infix[i] != '\0'; i++) {
	if (isalnum(infix[i])) {
	printf("%c", infix[i]);
	} else if (infix[i] == '(') {
```

```
push(infix[i]);
} else if (infix[i] == ')') {
while (top != -1 && stack[top] != '(') {
printf("%c", pop());
}
if (top != -1 && stack[top] == '(') {
pop(); // Discard '('
}
} else if (isOperator(infix[i])) {
while (top != -1 && getPrecedence(stack[top]) >= getPrecedence(infix[i])) {
printf("%c", pop());
}
push(infix[i]);
}
}

// Pop remaining operators from the stack
while (top != -1) {
printf("%c", pop());
}

printf("\n");
}

int main() {
char infixExpression[MAX_SIZE];

// Get infix expression as input
printf("Enter a valid parenthesized infix arithmetic expression: ");
```

```
fgets(infixExpression, sizeof(infixExpression), stdin);

// Remove newline character from the input
for (int i = 0; infixExpression[i] != '\0'; i++) {
if (infixExpression[i] == '\n') {
infixExpression[i] = '\0';
break;
}
}

// Convert infix to postfix and display the result
infixToPostfix(infixExpression);

return 0;
}
```

```
PS C:\Users\risku\coding> cd "c:\Users\risku\coding\" ; if ($?) { gcc rough.c -o rough } ; if ($?) { .\rough }
Enter a valid parenthesized infix arithmetic expression: a+b-c+d/c
Infix Expression: a+b-c+d/c
Postfix Expression: ab+c-dc/+
```

LAB Program – 3 (Rishabh Kumar, 1BM22CS221)

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_SIZE 5
```

```
int queue[MAX_SIZE];

int front = -1, rear = -1;


// Function to check if the queue is empty

int isEmpty() {

        return front == -1;

}


// Function to check if the queue is full

int isFull() {

        return (rear == MAX_SIZE - 1 && front == 0) || (rear + 1 == front);

}


// Function to insert an element into the queue

void insert(int value) {

        if (isFull()) {

        printf("Queue Overflow: Cannot insert %d, queue is full.\n", value);

        } else {

        if (front == -1) {

        front = 0;

        }

        rear = (rear + 1) % MAX_SIZE;

        queue[rear] = value;

        printf("Inserted %d into the queue.\n", value);

        }

}


// Function to delete an element from the queue

void delete() {

        if (isEmpty()) {

        printf("Queue Underflow: Cannot delete, queue is empty.\n");
```

```
        } else {
        printf("Deleted %d from the queue.\n", queue[front]);
        if (front == rear) {
        // If there was only one element in the queue
        front = rear = -1;
        } else {
        front = (front + 1) % MAX_SIZE;
        }
        }
}

// Function to display the elements in the queue
void display() {
        if (isEmpty()) {
        printf("Queue is empty.\n");
        } else {
        printf("Queue elements: ");
        int i = front;
        do {
        printf("%d ", queue[i]);
        i = (i + 1) % MAX_SIZE;
        } while (i != (rear + 1) % MAX_SIZE);
        printf("\n");
        }
}

int main() {
        insert(10);
        insert(20);
        insert(30);
        display();
```

```
    delete();

    display();


    insert(40);

    insert(50);

    insert(60); // This will cause a queue overflow

    display();


    delete();

    delete();

    delete();

    delete(); // This will cause a queue underflow

    display();


    return 0;

}
```

```
PS C:\Users\risku\coding> cd "c:\Users\risku\coding\" ; if ($?) { gcc rough.c -o rough } ; if ($?) { .\rough }
Inserted 10 into the queue.
Inserted 20 into the queue.
Inserted 30 into the queue.
Queue elements: 10 20 30
Deleted 10 from the queue.
Queue elements: 20 30
Inserted 40 into the queue.
Inserted 50 into the queue.
Inserted 60 into the queue.
Queue elements: 20 30 40 50 60
Deleted 20 from the queue.
Deleted 30 from the queue.
Deleted 40 from the queue.
Deleted 50 from the queue.
Queue elements: 60
```

3b) 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_SIZE 5

int circularQueue[MAX_SIZE];
int front = -1, rear = -1;

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

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

// Function to insert an element into the circular queue
void insert(int value) {
	if (isFull()) {
	printf("Circular Queue Overflow: Cannot insert %d, queue is full.\n", value);
	} else {
	if (front == -1) {
	front = 0;
	}
	rear = (rear + 1) % MAX_SIZE;
	circularQueue[rear] = value;
	printf("Inserted %d into the circular queue.\n", value);
	}
}
```

```c
// Function to delete an element from the circular queue
void delete() {
    if (isEmpty()) {
    printf("Circular Queue Underflow: Cannot delete, queue is empty.\n");
    } else {
    printf("Deleted %d from the circular queue.\n", circularQueue[front]);
    if (front == rear) {
    // If there was only one element in the circular queue
    front = rear = -1;
    } else {
    front = (front + 1) % MAX_SIZE;
    }
    }
}

// Function to display the elements in the circular queue
void display() {
    if (isEmpty()) {
    printf("Circular Queue is empty.\n");
    } else {
    printf("Circular Queue elements: ");
    int i = front;
    do {
    printf("%d ", circularQueue[i]);
    i = (i + 1) % MAX_SIZE;
    } while (i != (rear + 1) % MAX_SIZE);
    printf("\n");
    }
}

int main() {
```

```
insert(10);

insert(20);

insert(30);

display();


delete();

display();


insert(40);

insert(50);

insert(60); // This will cause a circular queue overflow

display();


delete();

delete();

delete();

delete(); // This will cause a circular queue underflow

display();


return 0;

}
```

```
PS C:\Users\risku\coding> cd "c:\Users\risku\coding\" ; if ($?) { gcc rough.c -o rough } ; if ($?) { .\rough }
Inserted 10 into the circular queue.
Inserted 20 into the circular queue.
Inserted 30 into the circular queue.
Circular Queue elements: 10 20 30
Deleted 10 from the circular queue.
Circular Queue elements: 20 30
Inserted 40 into the circular queue.
Inserted 50 into the circular queue.
Inserted 60 into the circular queue.
Circular Queue elements: 20 30 40 50 60
Deleted 20 from the circular queue.
Deleted 30 from the circular queue.
Deleted 40 from the circular queue.
Deleted 50 from the circular queue.
Circular Queue elements: 60
```

LAB Program – 4 (Rishabh Kumar, 1BM22CS221)

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>


// Node structure

struct Node {

        int data;

        struct Node* next;

};


// 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(1);

        }

        newNode->data = value;

        newNode->next = NULL;

        return newNode;

}


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

struct Node* insertFirst(struct Node* head, int value) {
```

```
struct Node* newNode = createNode(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 = createNode(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 for insertion.\n");
free(newNode);
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* insertEnd(struct Node* head, int value) {
	struct Node* newNode = createNode(value);

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

// Function to free the memory allocated for the linked list
void freeList(struct Node* head) {
	struct Node* current = head;
	struct Node* nextNode;
```

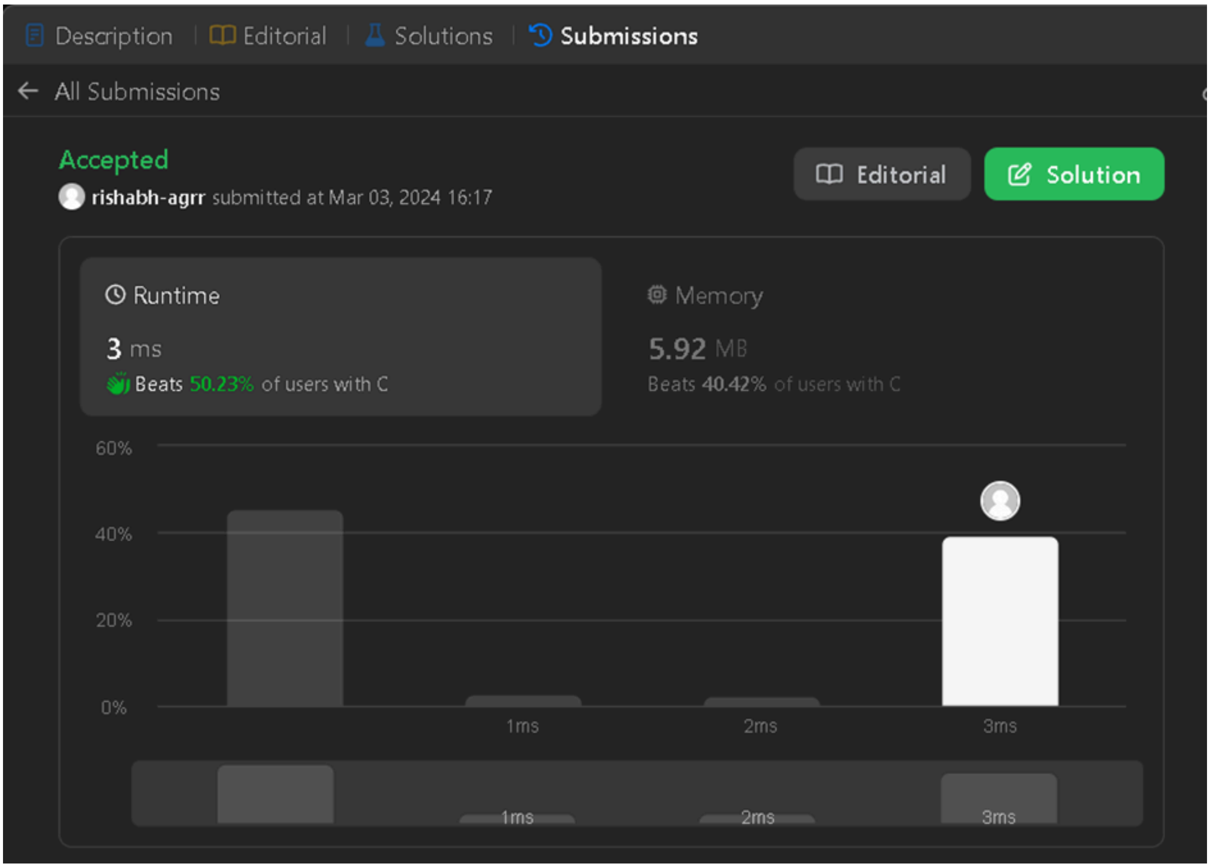
```
        while (current != NULL) {
        nextNode = current->next;
        free(current);
        current = nextNode;
        }
}

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

        // Inserting nodes at various positions
        head = insertEnd(head, 10);
        head = insertEnd(head, 20);
        head = insertFirst(head, 5);
        head = insertAtPosition(head, 15, 2);

        // Displaying the linked list
        display(head);

        // Freeing the memory allocated for the linked list
        freeList(head);

        return 0;
}
```

```
PS C:\Users\risku\coding> cd "c:\Users\risku\coding\" ; if ($?) { gcc rough.c -o rough } ; if ($?) { .\rough }
Linked List: 5 -> 15 -> 10 -> 20 -> NULL
PS C:\Users\risku\coding> 
```

LEETCODE – 206 Reverse Linked list

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *          int val;
 *          struct ListNode *next;
 * };
 */

//Reverse the linked list from node head, and link it with other
//nodes which came from the original one's tail
struct ListNode* reverse(struct ListNode* head,int len){
        if(len == 1){
        return head;
        }

        int count = 0;
        struct ListNode* p = head;
        while(count < len-1){
        p = p->next;
        count++;
        }
        struct ListNode* pEnd = p->next;

        struct ListNode* pPre = head;
        p = head->next;
        struct ListNode* pNext;

        count = 0;
        while(count < len-1){
        pNext = p->next;
        p->next = pPre;
```

```
        pPre = p;
        p = pNext;
        count++;
        }
        head->next = pEnd;

        return pPre;
}

struct ListNode* reverseBetween(struct ListNode* head, int left, int right){
        struct ListNode* p = head;struct ListNode* pPre = NULL;
        int count = 1;
        while(count < left){
        pPre = p;
        p = p->next;
        count++;
        }
        if(pPre){
        pPre->next = reverse(p,right-left+1);
        }
        else{
        head = reverse(p,right-left+1);
        }

        return head;
}
```



LAB Program – 5 (Rishabh Kumar, 1BM22CS221)

WAP to Implement Singly Linked List with following operations

a. Create a linked list.

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

c. Display the contents of the linked list.

```
#include <stdio.h>

#include <stdlib.h>


// Node structure

struct Node {

        int data;

        struct Node* next;
```

```
};

// 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(1);
    }
    newNode->data = value;
    newNode->next = NULL;
    return newNode;
}

// Function to insert a node at the end of the linked list
struct Node* insertEnd(struct Node* head, int value) {
    struct Node* newNode = createNode(value);

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

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

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

```
// Function to delete the first element from the linked list
struct Node* deleteFirst(struct Node* head) {
	if (head == NULL) {
	printf("List is empty. Cannot delete.\n");
	return NULL;
	}

	struct Node* newHead = head->next;
	free(head);
	return newHead;
}

// Function to delete a specified element from the linked list
struct Node* deleteElement(struct Node* head, int value) {
	if (head == NULL) {
	printf("List is empty. Cannot delete.\n");
	return NULL;
	}

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

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

	if (current->next == NULL) {
```

```
        printf("Element %d not found in the list. Cannot delete.\n", value);
        return head;
        }

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

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

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

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

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

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

// Function to free the memory allocated for the linked list
void freeList(struct Node* head) {
        struct Node* current = head;
        struct Node* nextNode;

        while (current != NULL) {
        nextNode = current->next;
        free(current);
        current = nextNode;
        }
}

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

        // Inserting nodes at the end of the linked list
        head = insertEnd(head, 10);
        head = insertEnd(head, 20);
        head = insertEnd(head, 30);
```

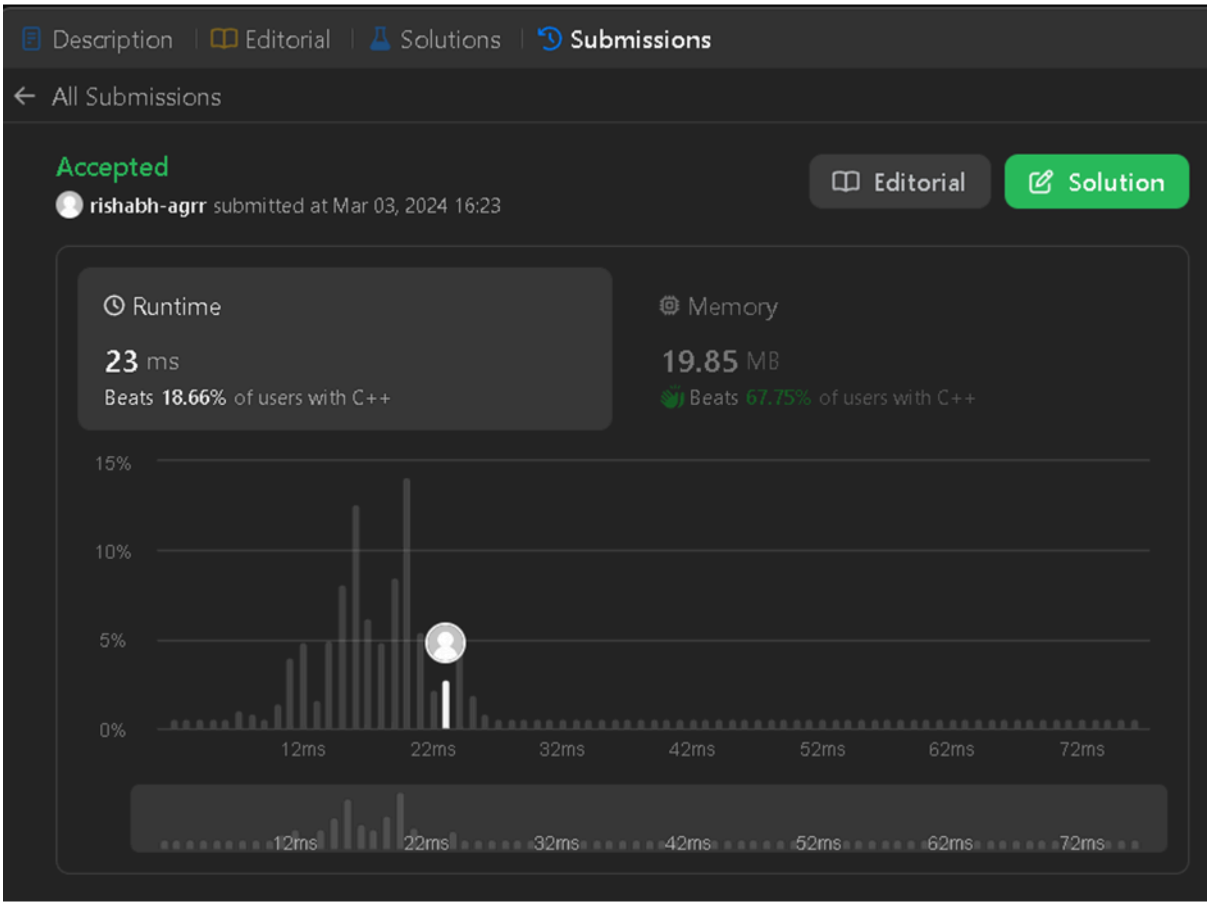
```
    // Displaying the linked list
    display(head);

    // Deleting the first element
    head = deleteFirst(head);
    display(head);

    // Deleting a specified element
    head = deleteElement(head, 20);
    display(head);

    // Deleting the last element
    head = deleteLast(head);
    display(head);

    // Freeing the memory allocated for the linked list
    freeList(head);

    return 0;
}
```

```
PS C:\Users\risku\coding> cd "c:\Users\risku\coding\" ; if ($?) { gcc rough.c -o rough } ; if ($?) { .\rough }
Linked List: 10 -> 20 -> 30 -> NULL
Linked List: 20 -> 30 -> NULL
Linked List: 30 -> NULL
Linked List: NULL
PS C:\Users\risku\coding>
```

LEETCODE – 155 (MIN STACK)

```cpp
#include <stack>

#include <climits>


class MinStack {

private:

        std::stack<int> mainStack;

        std::stack<int> minStack;


public:

        /** initialize your data structure here */

        MinStack() {}


        void push(int val) {

        mainStack.push(val);

        if (minStack.empty() || val <= minStack.top()) {

        minStack.push(val);

        }

        }


        void pop() {

        if (mainStack.top() == minStack.top()) {

        minStack.pop();

        }

        mainStack.pop();

        }


        int top() {

        return mainStack.top();

        }


        int getMin() {
```

```
        return minStack.top();
        }
};

// Example usage:
int main() {
        MinStack minStack;
        minStack.push(-2);
        minStack.push(0);
        minStack.push(-3);

        // Output: -3
        std::cout << minStack.getMin() << std::endl;

        minStack.pop();

        // Output: 0
        std::cout << minStack.top() << std::endl;

        // Output: -2
        std::cout << minStack.getMin() << std::endl;

        return 0;
}
```



LAB Program – 6 (Rishabh Kumar, 1BM22CS221)

6a) WAP to Implement Single Link List with following operations: Sort the linked list, Reverse the linked list, Concatenation of two linked lists.

```
#include <stdio.h>

#include <stdlib.h>


// Node structure

struct Node {

        int data;

        struct Node* next;

};
```

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

// Function to insert a node at the end of the linked list
struct Node* insertEnd(struct Node* head, int value) {
        struct Node* newNode = createNode(value);

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

// Function to sort the linked list (bubble sort)
struct Node* sortLinkedList(struct Node* head) {
    if (head == NULL || head->next == NULL) {
    return head;
    }

    int swapped;
    struct Node* temp;
    struct Node* current;

    do {
    swapped = 0;
    current = head;

    while (current->next != NULL) {
    if (current->data > current->next->data) {
    // Swap the data of current and next nodes
    temp = createNode(current->data);
    current->data = current->next->data;
    current->next->data = temp->data;
    free(temp);
```

```
            swapped = 1;
        }

        current = current->next;
    }
} while (swapped);

return head;
}

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

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

    head = prev;
    return head;
}

// Function to concatenate two linked lists
struct Node* concatenateLinkedLists(struct Node* list1, struct Node* list2) {
    if (list1 == NULL) {
```

```
        return list2;
        }

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

        current->next = list2;
        return list1;
}

// Function to free the memory allocated for the linked list
void freeList(struct Node* head) {
        struct Node* current = head;
        struct Node* nextNode;

        while (current != NULL) {
        nextNode = current->next;
        free(current);
        current = nextNode;
        }
}

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

        // Inserting nodes into list1
        list1 = insertEnd(list1, 10);
        list1 = insertEnd(list1, 30);
```

```
list1 = insertEnd(list1, 20);
list1 = insertEnd(list1, 40);

// Inserting nodes into list2
list2 = insertEnd(list2, 50);
list2 = insertEnd(list2, 70);
list2 = insertEnd(list2, 60);
list2 = insertEnd(list2, 80);

// Displaying the original linked lists
printf("Original List 1:\n");
display(list1);
printf("Original List 2:\n");
display(list2);

// Sorting list1
list1 = sortLinkedList(list1);
printf("Sorted List 1:\n");
display(list1);

// Reversing list2
list2 = reverseLinkedList(list2);
printf("Reversed List 2:\n");
display(list2);

// Concatenating list1 and list2
list1 = concatenateLinkedLists(list1, list2);
printf("Concatenated List 1 and List 2:\n");
display(list1);

// Freeing the memory allocated for the linked lists
```

```
	freeList(list1);

	return 0;
}
```

```
PS C:\Users\risku\coding> cd "c:\Users\risku\coding\" ; if ($?) { gcc rough.c -o rough } ; if ($?) { .\rough }
Original List 1:
Linked List: 10 -> 30 -> 20 -> 40 -> NULL
Original List 2:
Linked List: 50 -> 70 -> 60 -> 80 -> NULL
Sorted List 1:
Linked List: 10 -> 20 -> 30 -> 40 -> NULL
Reversed List 2:
Linked List: 80 -> 60 -> 70 -> 50 -> NULL
Concatenated List 1 and List 2:
Linked List: 10 -> 20 -> 30 -> 40 -> 80 -> 60 -> 70 -> 50 -> NULL
```

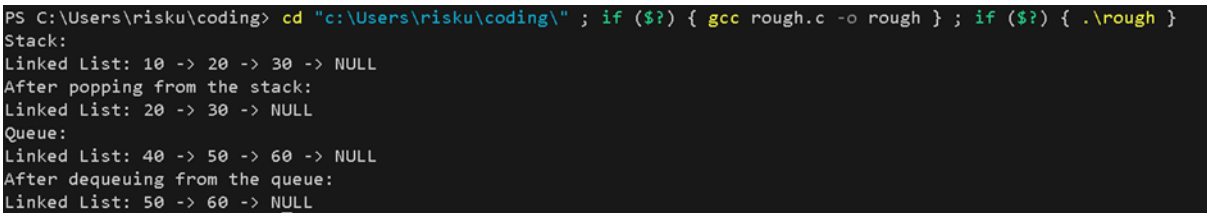
6b) WAP to Implement Single Link List to simulate Stack & Queue Operations

```
#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 value) {
	struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
	if (newNode == NULL) {
	printf("Memory allocation failed.\n");
	exit(1);
```

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

// Function to insert a node at the end of the linked list (for both stack and queue)
struct Node* insertEnd(struct Node* head, int value) {
struct Node* newNode = createNode(value);

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

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

current->next = newNode;
return head;
}

// Function to delete the first node (for both stack and queue)
struct Node* deleteFirst(struct Node* head) {
if (head == NULL) {
printf("List is empty.\n");
return NULL;
}

struct Node* temp = head;
```

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

// Function to display the contents of the linked list
void display(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* stack = NULL;
        struct Node* queue = NULL;

        // Pushing elements onto the stack
        stack = insertEnd(stack, 10);
        stack = insertEnd(stack, 20);
        stack = insertEnd(stack, 30);

        // Displaying the stack
        printf("Stack:\n");
        display(stack);

        // Popping an element from the stack
        stack = deleteFirst(stack);
```

```
    printf("After popping from the stack:\n");
    display(stack);

    // Enqueuing elements into the queue
    queue = insertEnd(queue, 40);
    queue = insertEnd(queue, 50);
    queue = insertEnd(queue, 60);

    // Displaying the queue
    printf("Queue:\n");
    display(queue);

    // Dequeuing an element from the queue
    queue = deleteFirst(queue);
    printf("After dequeuing from the queue:\n");
    display(queue);

    // Freeing the memory allocated for the linked lists
    while (stack != NULL) {
    stack = deleteFirst(stack);
    }

    while (queue != NULL) {
    queue = deleteFirst(queue);
    }

    return 0;
}
```



```
PS C:\Users\risku\coding> cd "c:\Users\risku\coding\" ; if ($?) { gcc rough.c -o rough } ; if ($?) { .\rough }
Stack:
Linked List: 10 -> 20 -> 30 -> NULL
After popping from the stack:
Linked List: 20 -> 30 -> NULL
Queue:
Linked List: 40 -> 50 -> 60 -> NULL
After dequeuing from the queue:
Linked List: 50 -> 60 -> NULL
```

<u>LAB Program – 7 (Rishabh Kumar, 1BM22CS221)</u>

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>


// Node structure

struct Node {

        int data;

        struct Node* prev;

        struct Node* next;

};


// 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(1);
```

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

// Function to insert a new node to the left of a specific node
struct Node* insertLeft(struct Node* head, int value, int targetValue) {
        struct Node* newNode = createNode(value);

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

        struct Node* current = head;

        // Find the target node
        while (current != NULL && current->data != targetValue) {
        current = current->next;
        }

        if (current == NULL) {
        printf("Target node not found.\n");
        free(newNode);
        return head;
        }

        // Insert the new node to the left
        if (current->prev != NULL) {
        current->prev->next = newNode;
```

```
newNode->prev = current->prev;
} else {
head = newNode;
}

newNode->next = current;
current->prev = newNode;

return head;
}

// Function to delete a node based on a specific value
struct Node* deleteNode(struct Node* head, int value) {
if (head == NULL) {
printf("List is empty.\n");
return NULL;
}

struct Node* current = head;

// Find the node with the specified value
while (current != NULL && current->data != value) {
current = current->next;
}

if (current == NULL) {
printf("Node with specified value not found.\n");
return head;
}

// Update the pointers of the adjacent nodes
```

```
        if (current->prev != NULL) {
        current->prev->next = current->next;
        } else {
        head = current->next;
        }

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

        free(current);
        return head;
}

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

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

        // Inserting nodes into the doubly linked list
        doublyLinkedList = insertLeft(doublyLinkedList, 20, 0);
        doublyLinkedList = insertLeft(doublyLinkedList, 30, 20);
```

```c
doublyLinkedList = insertLeft(doublyLinkedList, 40, 30);
doublyLinkedList = insertLeft(doublyLinkedList, 50, 0);

// Displaying the original doubly linked list
display(doublyLinkedList);

// Deleting a node with a specific value
doublyLinkedList = deleteNode(doublyLinkedList, 30);

// Displaying the doubly linked list after deletion
display(doublyLinkedList);

// Freeing the memory allocated for the doubly linked list
while (doublyLinkedList != NULL) {
struct Node* temp = doublyLinkedList;
doublyLinkedList = doublyLinkedList->next;
free(temp);
}

return 0;
}
```

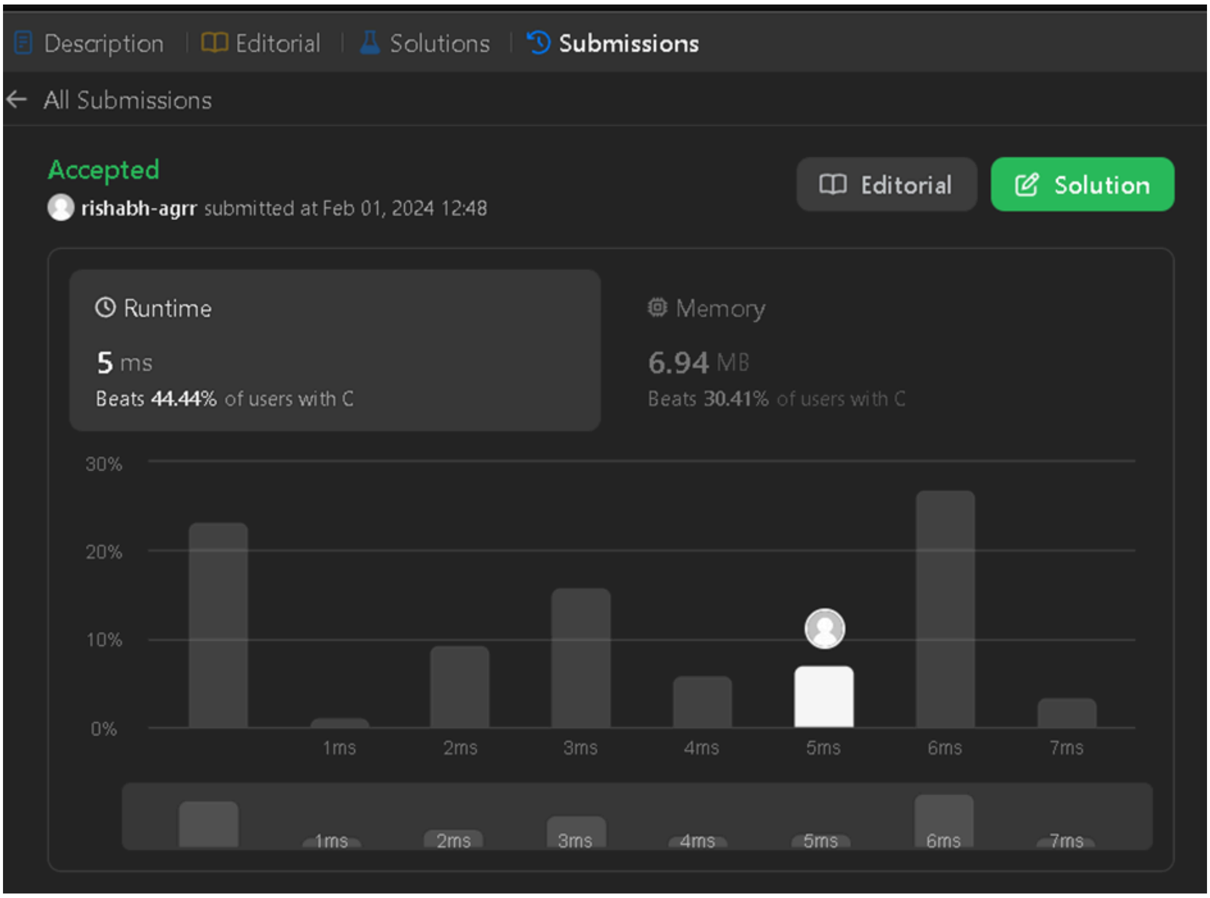
```
PS C:\Users\risku\coding> cd "c:\Users\risku\coding\" ; if ($?) { gcc rough.c -o rough } ; if ($?) { .\rough }
Target node not found.
Doubly Linked List: 40 <-> 30 <-> 20 <-> NULL
Doubly Linked List: 40 <-> 20 <-> NULL
PS C:\Users\risku\coding>
```

LEETCODE _ 725 (split linked list into parts)

```c
/**
 * Definition for singly-linked list.
```

```
 * struct ListNode {
 *     int val;
 *     struct ListNode *next;
 * };
 */
/**
 * Note: The returned array must be malloced, assume caller calls free().
 */
void append(struct ListNode **head, int val){
    struct ListNode* new=(struct ListNode*)malloc(sizeof(struct ListNode));
    struct ListNode* prev=*head;
    new->val=val;
    new->next=NULL;
    if(*head==NULL)
    *head=new;

    else{
    while(prev->next!=NULL)
    prev=prev->next;
    prev->next=new;
    }
}

int length(struct ListNode *head){
    struct ListNode *prev=head;
    int len=0;
    while(prev!=NULL){
    prev=prev->next;
    len++;
    }
    return len;
```

```
}

struct ListNode** splitListToParts(struct ListNode* head, int k, int* returnSize) {
        struct ListNode** heads=(struct ListNode**)malloc(sizeof(struct ListNode*)*k);
        int len=length(head);
        struct ListNode* prev=head;
        for(int i=0;i<k;i++){
        struct ListNode* nhead=NULL;
        heads[i]=nhead;
        }

        int common=len/k;
        int extra=len%k;

        int iter;
        int i=0;
        while(prev!=NULL){
        for(int j=0;j<common+((extra>0)?1:0);j++){
      append(&heads[i],prev->val);
        prev=prev->next;
        }
        i++;
        extra--;
        }
        *returnSize=k;
        return heads;
}
```



LAB Program – 8 (Rishabh Kumar, 1BM22CS221)

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>


// Node structure for the binary search tree

struct Node {

    int data;
```

```
struct Node* left;

struct Node* right;

};


// 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(1);

}

newNode->data = value;

newNode->left = newNode->right = NULL;

return newNode;

}


// Function to insert a value into the binary search tree

struct Node* insert(struct Node* root, int value) {

if (root == NULL) {

return createNode(value);

}


if (value < root->data) {

root->left = insert(root->left, value);

} else if (value > root->data) {

root->right = insert(root->right, value);

}


return root;

}
```

```
// Function to perform in-order traversal of the binary search tree

void inOrderTraversal(struct Node* root) {

        if (root != NULL) {

    inOrderTraversal(root->left);

        printf("%d ", root->data);

    inOrderTraversal(root->right);

        }

}


// Function to perform pre-order traversal of the binary search tree

void preOrderTraversal(struct Node* root) {

        if (root != NULL) {

        printf("%d ", root->data);

    preOrderTraversal(root->left);

    preOrderTraversal(root->right);

        }

}


// Function to perform post-order traversal of the binary search tree

void postOrderTraversal(struct Node* root) {

        if (root != NULL) {

    postOrderTraversal(root->left);

    postOrderTraversal(root->right);

        printf("%d ", root->data);

        }

}


// Function to free the memory allocated for the binary search tree

void freeTree(struct Node* root) {

        if (root != NULL) {

        freeTree(root->left);
```

```
        freeTree(root->right);
        free(root);
        }
}

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

        // Constructing a binary search tree
        root = insert(root, 50);
        insert(root, 30);
        insert(root, 20);
        insert(root, 40);
        insert(root, 70);
        insert(root, 60);
        insert(root, 80);

        // Displaying the elements in the tree using in-order traversal
        printf("In-order Traversal: ");
        inOrderTraversal(root);
        printf("\n");

        // Displaying the elements in the tree using pre-order traversal
        printf("Pre-order Traversal: ");
        preOrderTraversal(root);
        printf("\n");

        // Displaying the elements in the tree using post-order traversal
        printf("Post-order Traversal: ");
        postOrderTraversal(root);
        printf("\n");
```

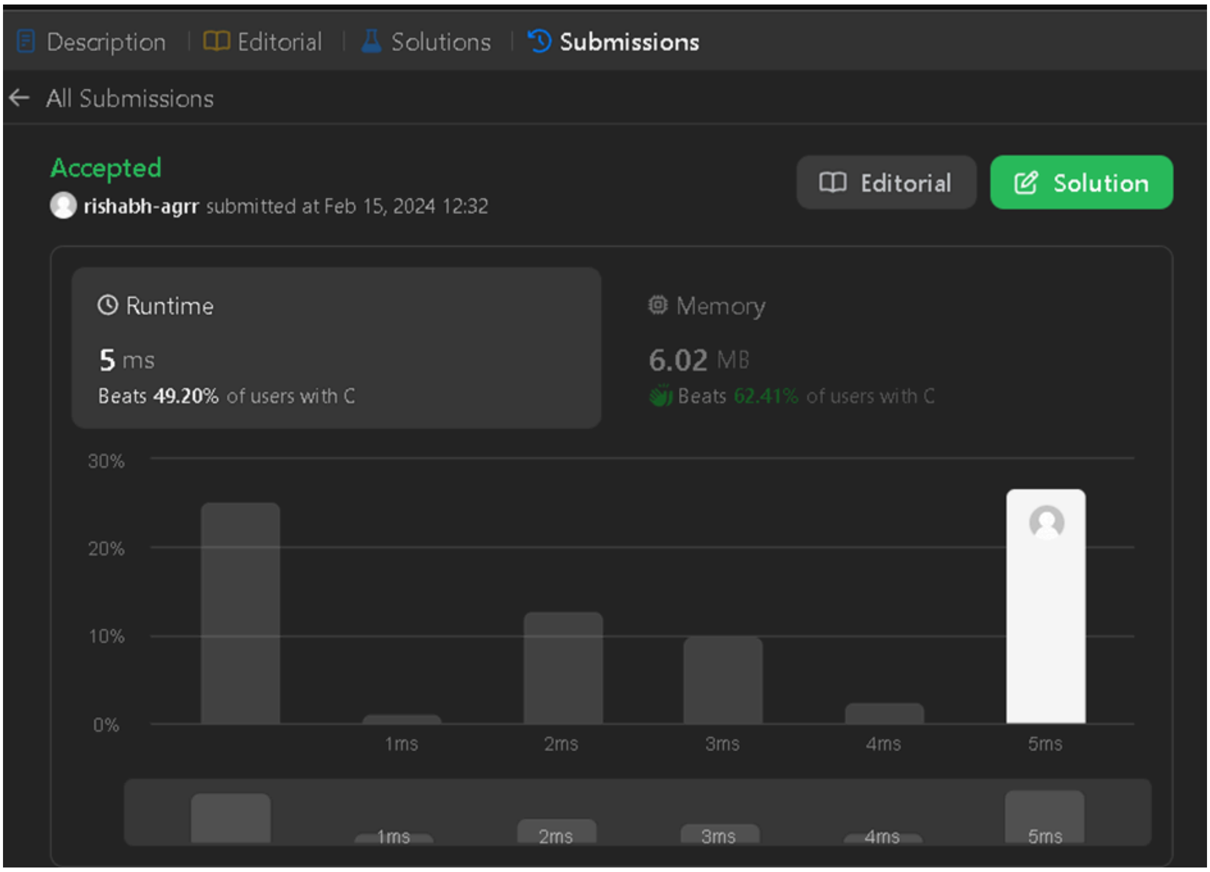
```
        // Freeing the memory allocated for the binary search tree
        freeTree(root);

        return 0;
}
```

```
PS C:\Users\risku\coding> cd "c:\Users\risku\coding\" ; if ($?) { gcc rough.c -o rough } ; if ($?) { .\rough }
In-order Traversal: 20 30 40 50 60 70 80
Pre-order Traversal: 50 30 20 40 70 60 80
Post-order Traversal: 20 40 30 60 80 70 50
```

Leetcode 61 – rotate the list

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *      int val;
 *      struct ListNode *next;
 * };
 */
 int GetLength(struct ListNode* head)
{
                if (head == NULL)
                        return 0;

                return 1 + GetLength(head->next);
}
struct ListNode* rotateRight(struct ListNode* head, int k){
        if (head == NULL || k == 0)
                        return head;
        int length = GetLength(head);
```

```
                if (length == 1)
                        return head;
        for(int i=0;i<k%length;i++)
        {
        struct ListNode *p=head;
    while(p->next->next!=NULL)
        {
        p=p->next;
        }
        struct ListNode *a=(struct ListNode *)malloc(sizeof(struct ListNode));
    a->val=p->next->val;
        a->next=head;
        head=a;
        p->next=NULL;
        }
        return head;

}
```



LAB Program – 9 (Rishabh Kumar, 1BM22CS221)

9a) Write a program to traverse a graph using BFS method.

```
#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>


#define MAX_VERTICES 100


// Queue structure for BFS
```

```
struct Queue {
	int front, rear;
	int capacity;
	int* array;
};

// Function to create a new queue
struct Queue* createQueue(int capacity) {
	struct Queue* queue = (struct Queue*)malloc(sizeof(struct Queue));
	if (queue == NULL) {
	printf("Memory allocation failed.\n");
	exit(1);
	}
	queue->capacity = capacity;
	queue->front = queue->rear = -1;
	queue->array = (int*)malloc(capacity * sizeof(int));
	if (queue->array == NULL) {
	printf("Memory allocation failed.\n");
	exit(1);
	}
	return queue;
}

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

// Function to enqueue a vertex into the queue
void enqueue(struct Queue* queue, int vertex) {
	if (queue->rear == queue->capacity - 1) {
```

```
        printf("Queue overflow.\n");
        exit(1);
        }
        if (isEmpty(queue)) {
        queue->front = 0;
        }
        queue->rear++;
    queue->array[queue->rear] = vertex;
}

// Function to dequeue a vertex from the queue
int dequeue(struct Queue* queue) {
        if (isEmpty(queue)) {
        printf("Queue underflow.\n");
        exit(1);
        }
        int vertex = queue->array[queue->front];
        if (queue->front == queue->rear) {
        queue->front = queue->rear = -1;
        } else {
        queue->front++;
        }
        return vertex;
}

// Function to perform BFS traversal on the graph
void BFS(int graph[MAX_VERTICES][MAX_VERTICES], int vertices, int startVertex) {
        struct Queue* queue = createQueue(vertices);
        bool visited[MAX_VERTICES] = {false};

        visited[startVertex] = true;
```

```
        enqueue(queue, startVertex);

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

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

        for (int adjacentVertex = 0; adjacentVertex < vertices; adjacentVertex++) {
        if (graph[currentVertex][adjacentVertex] == 1 && !visited[adjacentVertex]) {
        visited[adjacentVertex] = true;
        enqueue(queue, adjacentVertex);
        }
        }
        }

        printf("\n");

        free(queue->array);
        free(queue);
}

int main() {
        int vertices, edges;

        // Get the number of vertices and edges in the graph
        printf("Enter the number of vertices: ");
        scanf("%d", &vertices);
        printf("Enter the number of edges: ");
        scanf("%d", &edges);
```

```
        // Initialize the adjacency matrix with zeros

        int graph[MAX_VERTICES][MAX_VERTICES] = {0};


        // Get the edges of the graph

        printf("Enter the edges (vertex1 vertex2):\n");

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

        int vertex1, vertex2;

        scanf("%d %d", &vertex1, &vertex2);

        graph[vertex1][vertex2] = 1;

        graph[vertex2][vertex1] = 1; // For undirected graph

        }


        // Perform BFS traversal starting from vertex 0

        BFS(graph, vertices, 0);


        return 0;

}
```

```
PS C:\Users\risku\coding> cd "c:\Users\risku\coding\" ; if ($?) { gcc rough.c -o rough } ; if ($?) { .\rough }
Enter the number of vertices: 2
Enter the number of edges: 4
Enter the edges (vertex1 vertex2):
9
5
2
1
5
6
2
5
BFS Traversal starting from vertex 0: 0
```

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

```
#include <stdio.h>

#include <stdlib.h>
```

```
#include <stdbool.h>

#define MAX_VERTICES 100

// Function to perform DFS traversal on the graph

void DFS(int graph[MAX_VERTICES][MAX_VERTICES], int vertices, int startVertex, bool
visited[MAX_VERTICES]) {

    visited[startVertex] = true;


    for (int adjacentVertex = 0; adjacentVertex < vertices; adjacentVertex++) {

    if (graph[startVertex][adjacentVertex] == 1 && !visited[adjacentVertex]) {

    DFS(graph, vertices, adjacentVertex, visited);

    }

    }

}


// Function to check whether the graph is connected or not

bool isConnected(int graph[MAX_VERTICES][MAX_VERTICES], int vertices) {

    bool visited[MAX_VERTICES] = {false};


    // Perform DFS traversal starting from the first vertex

    DFS(graph, vertices, 0, visited);


    // Check if all vertices are visited

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

    if (!visited[i]) {

    return false;

    }

    }


    return true;

}
```

```
int main() {
	int vertices, edges;

	// Get the number of vertices and edges in the graph
	printf("Enter the number of vertices: ");
	scanf("%d", &vertices);
	printf("Enter the number of edges: ");
	scanf("%d", &edges);

	// Initialize the adjacency matrix with zeros
	int graph[MAX_VERTICES][MAX_VERTICES] = {0};

	// Get the edges of the graph
	printf("Enter the edges (vertex1 vertex2):\n");
	for (int i = 0; i < edges; i++) {
	int vertex1, vertex2;
	scanf("%d %d", &vertex1, &vertex2);
	graph[vertex1][vertex2] = 1;
	graph[vertex2][vertex1] = 1; // For undirected graph
	}

	// Check whether the graph is connected or not
	if (isConnected(graph, vertices)) {
	printf("The graph is connected.\n");
	} else {
	printf("The graph is not connected.\n");
	}

	return 0;
}
```

```
PS C:\Users\risku\coding> cd "c:\Users\risku\coding\" ; if ($?) { gcc rough.c -o rough } ; if ($?) { .\rough }
Enter the number of vertices: 2
Enter the number of edges: 4
Enter the edges (vertex1 vertex2):
5 5
4 2
8 9
7 6
The graph is not connected.
```

<u>LAB Program – 10 (Rishabh Kumar, 1BM22CS221)</u>

Given a File of N employee records with a set K of Keys(4-digit) which uniquely determine the records in file F.

Assume that file F is maintained in memory by a Hash Table (HT) of m memory locations with L as the set of memory addresses (2-digit) of locations in HT.

Let the keys in K and addresses in L are integers.

Design and develop a Program in C that uses Hash function H: K -> L as H(K)=K mod m (remainder method), and implement hashing technique to map a given key K to the address space L.

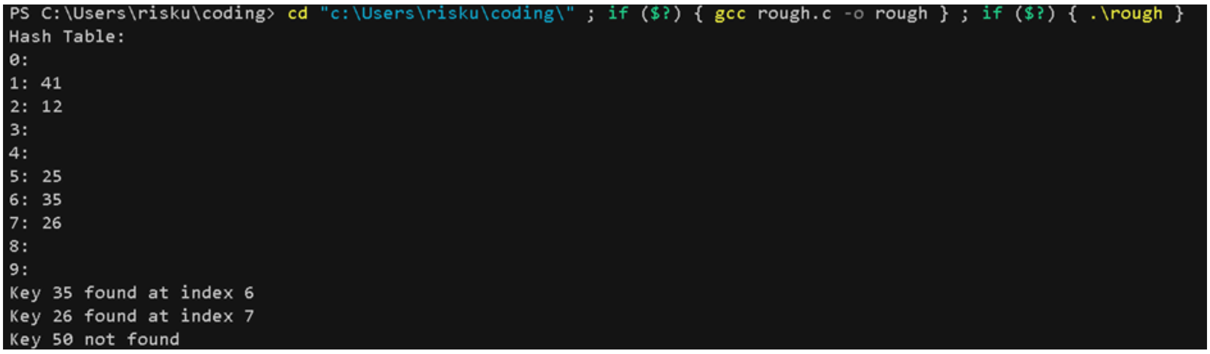
Resolve the collision (if any) using linear probing.

```
#include <stdio.h>

#define TABLE_SIZE 10

int hash_table[TABLE_SIZE];

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

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

// Function to insert a value into the hash table using linear probing
void insert(int key) {
        int hkey = hash(key);
        int index = hkey;
        int i = 0;

        while (hash_table[index] != -1) {
        i++;
        index = (hkey + i) % TABLE_SIZE;
        if (i == TABLE_SIZE) {
        printf("Hash table is full. Unable to insert key %d\n", key);
        return;
        }
        }

        hash_table[index] = key;
}

// Function to search for a value in the hash table using linear probing
void search(int key) {
        int hkey = hash(key);
        int index = hkey;
        int i = 0;

        while (hash_table[index] != key) {
```

```
        i++;
        index = (hkey + i) % TABLE_SIZE;
        if (hash_table[index] == -1 || i == TABLE_SIZE) {
        printf("Key %d not found\n", key);
        return;
        }
        }

        printf("Key %d found at index %d\n", key, index);
}

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

int main() {
        initializeHashTable();

        insert(12);
        insert(25);
        insert(35);
        insert(26);
        insert(41);
```

```c
        displayHashTable();

        search(35);
        search(26);
        search(50);

        return 0;
}
```



```
PS C:\Users\risku\coding> cd "c:\Users\risku\coding\" ; if ($?) { gcc rough.c -o rough } ; if ($?) { .\rough }
Hash Table:
0:
1: 41
2: 12
3:
4:
5: 25
6: 35
7: 26
8:
9:
Key 35 found at index 6
Key 26 found at index 7
Key 50 not found
```