VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
On

DATA STRUCTURES (23CS3PCDST)

Submitted by

RAGHAVENDRA R (1BM22CS214)

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 RAGHAVENDRA R **(1BM22CS214)**, 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 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

```c
#include<stdlib.h>

#include<stdio.h>

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

typedef struct Node Node;

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

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

```c
typedef struct {
    Node *top;
}LinkedList;

void push(LinkedList *stack, int value) {
    Node *newNode = createNode(value);
    newNode -> next = stack -> top;
    stack -> top = newNode;
}

int pop(LinkedList *stack) {
    if(stack -> top == NULL) {
        printf("stack is empty \n");
        return -1;
    }
    int poppedValue = stack -> top -> data;
    Node *temp = stack -> top;
    stack -> top = stack -> top -> next;
    free(temp);
    return poppedValue;
}

void main() {
    LinkedList stack;
    stack.top = NULL;
    printf("stack operations : \n");
     push(&stack, 18);
     push(&stack, 46);
     push(&stack, 16);
     push(&stack, 7);
     display(stack.top);
    printf("popped value : %d\n", pop(&stack));
    printf("popped value : %d\n", pop(&stack));
     display(stack.top);
}
```

output :

```
7 -> 16 -> 46 -> 18 -> NULL
popped value : 7
popped value : 16
46 -> 18 -> NULL
```

LAB 2

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

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#define max 100
char s[max];
int top=-1;
void push(char value)
{
 s[++top]=value;
}
char pop()
{
    char value=' ';
    if (top==-1)
        return value;
    else
        return s[top--];
}
int pre(char value)
{
    if(value=='^')
    {
        return 3;
    }
    else if(value =='*' || value =='/')
    {
        return 2;
    }
    else if(value =='+' || value =='-')
    {
        return 1;
    }
    else
    {
        return 0;
    }

}
void convert(char infix[],char postfix[])
{
    int i=0,j=0;
    char x;
    while(infix[i]!='\0')
    {
        if(infix[i]=='(')
        {
```

```
            push(infix[i]);
            i++;
        }
        else if(infix[i]==')')
        {
            while((s[top]!='(') && (top!=-1))
            {
                postfix[j++]=pop();

            }
            x=pop();
            i++;
        }
        else if(isalnum(infix[i]))
        {
            postfix[j++]=infix[i];
            i++;

        }
        else
        {
            while((s[top]!='(') &&( top!=-1) && (pre(s[top])>=pre(infix[i])))
            {
                postfix[j++]=pop();
}
            push(infix[i]);
            i++;
        }
    }
    while((s[top]!='(') && (top!=-1))
    {
        postfix[j++]=pop();
    }
    postfix[j]='\0';
}
void main()
{
    char infix[100],postfix[100];
    printf("enter the exp:");
    scanf("%s",infix);
    convert(infix,postfix);
    printf("postfix:");
    printf("%s",postfix);
}
```

Output :

```
enter the exp:a+b-*c/d
postfix:ab+c*d/-
```

2b)LEET CODE

Demonstration of account creation on LeetCode platform
Program.

Design a stack that supports push, pop, top, and retrieving the minimum element in constant time.

Implement the MinStack class:
MinStack() initializes the stack object.
void push(int val) pushes the element val onto the stack.
void pop() removes the element on the top of the stack.
int top() gets the top element of the stack.
int getMin() retrieves the minimum element in the stack.
You must implement a solution with O(1) time complexity for each function.

```c
#include<stdio.h>
#include<stdlib.h>
typedef struct
{
    int value;
    int min;

} StackNode;

typedef struct
{
    StackNode *array;
    int capacity;
    int top;
}MinStack;

MinStack* minStackCreate()
{
    MinStack* stack=(MinStack*)malloc(sizeof(MinStack));
    stack->capacity = 10;
    stack->array = (StackNode*)malloc(stack->capacity * sizeof(StackNode));
    stack->top = -1;
    return stack;

}

void minStackPush(MinStack* obj, int val)
{
     if (obj->top == obj->capacity - 1)
    {
        obj->capacity *= 2;
        obj->array = (StackNode*)realloc(obj->array, obj->capacity *
sizeof(StackNode));
    }
```

```
    StackNode newNode;
    newNode.value = val;
    newNode.min = (obj->top == -1) ? val : (val < obj->array[obj->top].min) ? val :
obj->array[obj->top].min;
    obj->array[++(obj->top)] = newNode;
}

void minStackPop(MinStack* obj)
{
     if (obj->top != -1)
     {
        obj->top--;
    }

}

int minStackTop(MinStack* obj)
{
    if (obj->top != -1)
    {
        return obj->array[obj->top].value;
    }
    return -1;

}

int minStackGetMin(MinStack* obj)
{
    if (obj->top != -1)
    {
        return obj->array[obj->top].min;
    }
    return -1;

}

void minStackFree(MinStack* obj)
{
    free(obj->array);
    free(obj);
}
```

LAB 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>
#include <stdlib.h>
#define max 4
int rear=-1;
int front=-1;
int q[max];
void enqueue(int x)
{
    if (rear==max-1)
    {
        printf("queue overflow\n");
    }
    else if(front==-1 && rear==-1)
    {
        front=rear=0;
    }
    else{
        rear++;
    }
    q[rear]=x;
}
int dequeue()
{
    int x=-1;
    if (front==-1 || front>rear)
    {
        printf("\nunderflow");
        return -1;
    }
    else
    {
        x=q[front];
        front++;
        if(front>rear)
            front=rear=-1;
        return x;

    }
}
void display()
{
    if(front==-1 || front>rear)
```

```
    {
        printf("\nunderflow");
    }
    else
    {
        for(int i=front;i<=rear;i++)
        {
            printf("%d\t",q[i]);
        }

    }
}
void main()
{
    int c,no,x;
    while(1)
    {
        printf("enter 1 for insert 2 for delete 3 for display 4 for exit\n");
        printf("enter the choice:");
        scanf("%d",&c);
        switch (c)
        {
        case 1:
            printf("enter the no:");
            scanf("%d",&no);
            enqueue(no);

            break;
        case 2:x=dequeue();
                if (x!=-1)
                {
                    printf("%d is popped\n",x);
                }
                break;
        case 3:display();
                break;
        case 4:exit(0);
                // break;


        default:printf("invalid\n");
            break;
    }
}

}
```

Output :

```
enter the choice:1
enter the no:1
enter 1 for insert 2 for delete 3 for display 4 for exit
enter the choice:1
enter the no:2
enter 1 for insert 2 for delete 3 for display 4 for exit
enter the choice:1
enter the no:3
enter 1 for insert 2 for delete 3 for display 4 for exit
enter the choice:3
1       2       3       enter 1 for insert 2 for delete 3 for display 4 for exit
enter the choice:2
1 is popped
enter 1 for insert 2 for delete 3 for display 4 for exit
enter the choice:2
2 is popped
```

```
LAB 3b

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

#include <stdio.h>
#include <stdlib.h>
#define max 10
int rear=-1;
int front=-1;
int q[max];
int isfull()
{
    if(front==rear+1 || front==0 && rear==max-1)
        return 1;
    return 0;
 }
int is_empty()
{
    if(front==-1 && rear==-1)
        return 1;
    return 0;
}
void enqueue(int x)
```

```
{
    if(isfull())
    {
        printf("overflow\t");
    }
    else if(front==-1 && rear==-1)
    {
        front=0;
        rear=0;
    }
    else
    {
        rear=(rear+1)%max;
    }
    q[rear]=x;
}
int dequeue()
{
    int value=-1;
    if(is_empty())
    {
        printf("underflow\t");
        return -1;
    }
    else
   {
        value=q[front];
    if(front==rear)
    {
        front=-1;
        rear=-1;
    }
    else{
        front=(front+1)%max;
    }
    return value;
   }
}
void display()
{
    int i;
    if(is_empty())
    {
        printf("underflow\t");

    }
    else{
        printf("elements are:");
        for( i=front;i!=rear;i=(i+1)%max)
```

```
        {
            printf("%d\t",q[i]);
        }
        printf("%d",q[i]);
    }
}
void main()
{
    int c,no,x;

    while(1)
    {
        printf("enter 1 for insert 2 for delete 3 for display 4 for exit\n");
        printf("enter the choice:");
        scanf("%d",&c);
        switch (c)
        {
        case 1:
            printf("enter the no:");
            scanf("%d",&no);
            enqueue(no);

            break;
        case 2:x=dequeue();
                if (x!=-1)
                {
                    printf("%d is popped\n",x);
                }
                break;
        case 3:display();
                break;
        case 4:exit(0);
                // break;


        default:printf("invalid\n");
            break;
    }
}
}
```

```
enter the choice:1
enter the no:18
enter 1 for insert 2 for delete 3 for display 4 for exit
enter the choice:1
enter the no:49
enter 1 for insert 2 for delete 3 for display 4 for exit
enter the choice:1
enter the no:33
enter 1 for insert 2 for delete 3 for display 4 for exit
enter the choice:3
elements are:18 49      33enter 1 for insert 2 for delete 3 for display 4 for exit
enter the choice:2
18 is popped
enter 1 for insert 2 for delete 3 for display 4 for exit
enter the choice:3
elements are:49 33enter 1 for insert 2 for delete 3 for display 4 for exit
```

```
LAB 4

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 the end of the list.
Display the contents of the linked list.

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

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

typedef struct Node Node;

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

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

typedef struct {
    Node *front;
    Node *rear;
}LinkedList;

void enqueue(LinkedList *queue , int value)
{
    Node *newNode = createNode(value);
    if(queue -> front ==NULL)
    {
        queue -> front = newNode;
```

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

int dequeue(LinkedList *queue)
{
    if(queue -> front == NULL)
    {
        printf("queue is empty : \n");
        return -1;
    }
    int dequeue_value = queue -> front -> data;
    Node *temp = queue -> front;
    queue -> front = queue -> front -> next;
    free(temp);
    return dequeue_value;
}

void main()
{
    LinkedList queue;
    queue.front = NULL;
    queue.rear = NULL;

    printf("\n queue operations : \n");
    enqueue(&queue,40);
    enqueue(&queue,50);
    enqueue(&queue,60);
    display(queue.front);
    display(queue.front);
}
```

```
 queue operations :
40 -> 50 -> 60 -> NULL
40 -> 50 -> 60 -> NULL
```

LAB 4b

Program - Leetcode platform-Reverse Linked List II

Given the head of a singly linked list and two integers left and right where left <= right, reverse the nodes of the list from position left to position right, and return the reversed list.

```
struct ListNode* reverseBetween(struct ListNode* head, int left, int right) {

    if (head == NULL) return NULL;

    if (left == right) return head;

    struct ListNode* prev = NULL;
    struct ListNode* curr = head;

    int index = 1;
    while (index < left)
    {
        prev = curr;
        curr = curr->next;
        index++;
    }

    struct ListNode* leftMinusOneNode = prev;

    struct ListNode* leftNode = curr;
    struct ListNode* next = NULL;

    while (left <= right)
    {
        next = curr->next;

        curr->next = prev;

        prev = curr;
        curr = next;
        left++;
    }

    if (leftMinusOneNode == NULL) // means head changes
        head = prev;
    else
        leftMinusOneNode->next = prev;

    leftNode->next = curr;

    return head;
}
```

```
LAB 5a

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>

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

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

void insertAtEnd(struct Node** head,int value)
{

    struct Node* newNode = createNode(value);
    if(*head == NULL)
    {
        *head = newNode;
    }
    else
    {
        struct Node* temp = *head;
        while(temp -> next !=NULL)
        {
            temp = temp -> next;
        }
        temp -> next = newNode;
    }
}
```

```c
//function to delete the first element from the linked list
void deleteFirst(struct Node** head)
{
    if(*head == NULL)
    {
        struct Node* temp = *head;
        *head = (*head) -> next;
        free(temp);
    }
}

//to delete a specified element
void deleteEle(struct Node** head,int value)
{
    struct Node* current  = *head;
    struct Node* prev = NULL;

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

    if(current == NULL)
    {
        printf("empty");
    }
    if(prev =  NULL)
    {
        *head = current -> next;
    }
    else
    {
        prev -> next = current -> next;
    }
    free(current);
}

//to delete the last element
void deleteLast(struct Node** head)
{
    if(*head == NULL)
    {
        printf("empty");
    }
    struct Node* temp = *head;
    struct Node* prev = NULL;
    while(temp -> next != NULL)
    {
```

```
        prev = temp;
        temp = temp -> next;
    }
    if(prev == NULL)
    {
        *head = NULL;
    }
    else
    {
        prev -> next = NULL;
    }
    free(temp);
}

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

//main function
void main()
{
    struct Node* head = NULL;

    insertAtEnd(&head,1);
    insertAtEnd(&head,2);
    insertAtEnd(&head,3);
    insertAtEnd(&head,4);

    printf("initial linked list : ");
    display(head);

    deleteFirst(&head,1);
    printf("\nAfter deleting the first element : ");
    display(head,1);

    deleteEle(&head,2);
    printf("\nafter deleting the specified element : ");
    display(head,2);

    deleteLast(&head);
    printf("\nafter deleting the last element : ");
```

```
    display(head);

}
```

Output :

```
Initial Linked List: 1 -> 2 -> 3 -> NULL

After deleting the first element: 2 -> 3 -> NULL

After deleting the specified element (2): 3 -> NULL

After deleting the last element: NULL

Process returned 0 (0x0)   execution time : 0.052 s
Press any key to continue.
```

LAB 5b - leetcode

Given the head of a singly linked list and an integer k, split the linked list into k consecutive linked list parts.
The length of each part should be as equal as possible: no two parts should have a size differing by more than one. This may lead to some parts being null.
The parts should be in the order of occurrence in the input list, and parts occurring earlier should always have a size greater than or equal to parts occurring later.
Return an array of the k parts.

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


struct ListNode** splitListToParts(struct ListNode* head, int k, int* returnSize) {
    int length = getLength(head);
    int partSize = length / k;
    int remainder = length % k;

    struct ListNode** result = (struct ListNode**)malloc(k * sizeof(struct
ListNode*));
    *returnSize = k;

    for (int i = 0; i < k; i++) {
        int currentPartSize = partSize + (i < remainder ? 1 : 0);

        if (currentPartSize == 0) {
            result[i] = NULL;
        } else {
            result[i] = head;
            for (int j = 0; j < currentPartSize - 1; j++) {
                head = head->next;
            }

            struct ListNode* temp = head->next;
```

```
            head->next = NULL;
            head = temp;
        }
    }

    return result;
}
```

Lab 6

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

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

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

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

Node *sortList(Node *head)
{
    if(head == NULL || head -> next == NULL)
        return head;
    int swapped;
    Node *temp;
    Node *end = NULL;
    do
    {
        swapped = 0;
        temp = head;
        while(temp -> next != end)
        {
            if(temp -> data > temp -> next ->data)
            {
                int tempData = temp -> data;
```

```
                temp -> data = temp -> next -> data;
                temp -> next -> data =tempData;
                swapped = -1;
            }
            temp = temp -> next;
        }
        end = temp;
    }while(swapped);
    return head;
}

Node *reverseList(Node *head)
{
    Node *prev = NULL;
    Node *current = head;
    Node *nextNode = NULL;
    while(current != NULL)
    {
        nextNode = current -> next;
        current -> next = prev;
        prev = current;
        current = nextNode;
    }
    return prev;
}

Node *concatLists(Node *list1 , Node *list2)
{
    if(list1 == list2)
        return list2;
    Node *temp = list1;
    while(temp -> next != NULL)
    {
        temp = temp -> next;
    }
    temp -> next = list2;
    return list1;
}

void main()
{
    Node *list1 = createNode(3);
    list1 -> next = createNode(1);
    list1 -> next -> next = createNode(4);

    Node *list2 = createNode(2);
    list2 -> next = createNode(5);

    printf("original list 1 : ");
```

```
    display(list1);
    printf("original list 2 : ");
    display(list2);

    list1 = sortList(list1);
    printf("sorted list 1 : ");
    display(list1);

    list1 = reverseList(list1);
    printf("reversed list 1 : ");
    display(list1);

    Node *concatenated = concatLists(list1,list2);
    printf("concatenated list : ");
    display(concatenated);
}
```

Output :

```
original list 2 : 2 -> 5 -> NULL
sorted list 1 : 1 -> 3 -> 4 -> NULL
reversed list 1 : 4 -> 3 -> 1 -> NULL
concatenated list : 4 -> 3 -> 1 -> 2 -> 5 -> NULL
```

Lab 6B

```
WAP to Implement Single Link List to simulate Stack and Queue
Operations.
using stack

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

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

typedef struct Node Node;

Node *createNode(int value)
```

```
{
    Node *newNode = (Node*)malloc(sizeof(Node));
    newNode -> data = value;
    newNode -> next = NULL;
    return newNode;
}

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

typedef struct {
    Node *top;
}LinkedList;

void push(LinkedList *stack, int value) {
    Node *newNode = createNode(value);
    newNode -> next = stack -> top;
    stack -> top = newNode;
}

int pop(LinkedList *stack) {
    if(stack -> top == NULL) {
        printf("stack is empty \n");
        return -1;
    }
    int poppedValue = stack -> top -> data;
    Node *temp = stack -> top;
    stack -> top = stack -> top -> next;
    free(temp);
    return poppedValue;
}

void main() {
    LinkedList stack;
```

```
    stack.top = NULL;
    printf("stack operations : \n");
     push(&stack, 10);
     push(&stack, 20);
     push(&stack, 25);
     push(&stack, 30);
     display(stack.top);
    printf("popped value : %d\n", pop(&stack));
    printf("popped value : %d\n", pop(&stack));
     display(stack.top);
}
```

```
using queue

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

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

typedef struct Node Node;

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

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

typedef struct {
    Node *front;
    Node *rear;
}LinkedList;

void enqueue(LinkedList *queue , int value)
```

```
{
    Node *newNode = createNode(value);
    if(queue -> front ==NULL)
    {
        queue -> front = newNode;
        queue -> rear = newNode;
    }
    else
    {
        queue -> rear -> next = newNode;
        queue -> rear = newNode;
    }
}

int dequeue(LinkedList *queue)
{
    if(queue -> front == NULL)
    {
        printf("queue is empty : \n");
        return -1;
    }
    int dequeuedvalue = queue -> front -> data;
    Node *temp = queue -> front;
    queue -> front = queue -> front -> next;
    free(temp);
    return dequeuedvalue;
}

void main()
{
    LinkedList queue;
    queue.front = NULL;
    queue.rear = NULL;

    printf("\n queue operations : \n");
    enqueue(&queue,40);
    enqueue(&queue,50);
    enqueue(&queue,60);
```

```
    display(queue.front);
    printf("dequeued from queue : %d\n",dequeue(&queue));
    printf("dequeued from queue : %d\n",dequeue(&queue));
    display(queue.front);
}
```

Output :

```
30 -> 25 -> 20 -> 10 -> NULL
popped value : 30
popped value : 25
20 -> 10 -> NULL
```

```
40 -> 50 -> 60 -> NULL
dequeued from queue : 40
dequeued from queue : 50
60 -> NULL
```

Lab 7

WAP to Implement doubly link list with primitive operations

a) Create a doubly linked list.
b) Insert a new node to the left of the node.
c) Delete the node based on a specific value

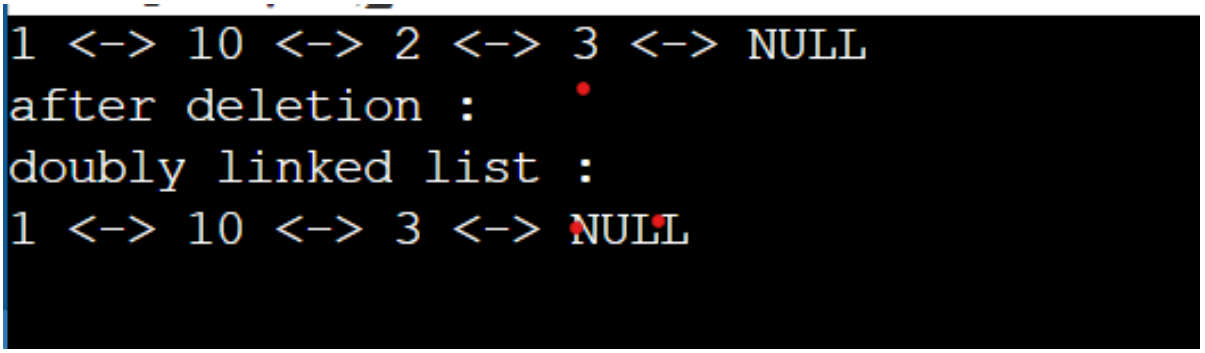
creation and insertion and deletion

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

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

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;
};

void insertNode(struct Node *head,struct Node *forget,int data)
{
    struct Node *newNode = createNode(data);
    if(forget -> prev != NULL)
    {
        forget -> prev -> next = newNode;
        newNode -> prev = forget -> prev;
    }
    else
    {
        head = newNode;
    }
    newNode -> next = forget;
```

```
    forget -> prev = newNode;
}

void deleteNode(struct Node *head,int value)
{
    struct Node *current = head;
    while(current != NULL)
    {
        if(current -> data ==value)
        {
            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;
        }
        current = current -> next;
    }
    printf("node with value %d not found \n",value);
}

void display(struct Node *head)
{

    printf("doubly linked list : \n");
    while(head != NULL)
    {
        printf("%d <-> ",head -> data);
        head = head -> next;
    }
    printf("NULL\n");
}

void main()
{
    struct Node *head = NULL;
    head = createNode(1);
    head -> next = createNode(2);
    head -> next -> prev = head;
    head -> next -> next = createNode(3);
```

```
    head -> next -> next -> prev = head -> next;

    display(head);

    insertNode(head,head -> next,10);
    printf("after insertion : \n");
    display(head);

    deleteNode(head,2);
    printf("after deletion : \n");
    display(head);
    return 0;
}
```

Output :



LAB 7b

```
Program - Leetcode platform-Rotate List
Given the head of a linked list, rotate the list to the right by k places.

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;

    if (k > length)
    {
        int t = k / length;
        k = k - (t * length);
    }

    while (k > 0)
    {
        int tempLength = length;
        struct ListNode* secondLastNode = head;
        while (tempLength - 2 > 0)
        {
            secondLastNode = secondLastNode->next;
            tempLength--;
        }
        secondLastNode->next->next = head;
        head = secondLastNode->next;
        secondLastNode->next = NULL;
        k--;
    }
    return head;
}
```

LAB 8

Write a program

a) To construct a binary Search tree.
b) To traverse the tree using all the methods i.e., in-order, preorder and post order
c) To display the elements in the tree.

```
#include <stdio.h>
#include <stdlib.h>
struct node{
    int data;
    struct node *left;
    struct node *right;
    }*root=NULL;
struct node *create(struct node *t,int ele){

    if(t==NULL){
        struct node *temp=(struct node *) malloc (sizeof(struct node));
        temp->data=ele;
        temp->left=temp->right=NULL;
        return temp;
    }
    else{
            if(ele<t->data){
                t->left=create(t->left,ele);
            }
            else{
                t->right=create(t->right,ele);
            }

    }
    return t;
}
void pre(struct node *root){
    struct node *r;
    r=root;
    if(r!=NULL){
        printf("%d\t",r->data);
        pre(r->left);
        pre(r->right);
    }
}
void in(struct node *root){
    struct node *r;
    r=root;
    if(r!=NULL){
```

```
        in(r->left);
        printf("%d\t",r->data);
        in(r->right);
    }
}

void post(struct node *root){
    struct node *r;
    r=root;
    if(r!=NULL){

        post(r->left);
        post(r->right);
        printf("%d\t",r->data);
    }
}
void main(){
    int n,ele;
    printf("enter the no of elements:");
    scanf("%d",&n);
    for(int i=0;i<n;i++){
        printf("enter the element %d:",i+1);
        scanf("%d",&ele);
        root=create(root,ele);
    }
    printf("display the elements in preorder traversal:");
    pre(root);
    printf("\ndisplay the elements in inorder traversal:");
    in(root);
    printf("\ndisplay the elements in postorder traversal:");
    post(root);

}
```

output:

```
enter the no of elements:5
enter the element 1:18
enter the element 2:17
enter the element 3:46
enter the element 4:7
enter the element 5:33
display the elements in preorder traversal:18   17      7       46      33
display the elements in inorder traversal:7     17      18      33      46
display the elements in postorder traversal:7   17      33      46      18
```

```
LAB 8b

Program - Hacker Rank platform
Swapping subtrees of a node means that if initially node has left subtree L
and right subtree R, then after swapping, the left subtree will be R and the
right subtree, L.

Complete the swapNodes function in the editor below. It should return a
two-dimensional array where each element is an array of integers representing
the node indices of an in-order traversal after a swap operation.

swapNodes has the following parameter(s):
- indexes: an array of integers representing index values of each , beginning
with the first element, as the root.
- queries: an array of integers, each representing a  value

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

struct node
{
    int id;
    int depth;
    struct node *left, *right;
};

void
inorder(struct node* tree)
{
    if(tree == NULL)
        return;

    inorder(tree->left);
    printf("%d ",tree->id);
    inorder((tree->right));

}


int
main(void)
{
    int no_of_nodes, i = 0;
    int l,r, max_depth,k;
    struct node* temp = NULL;
    scanf("%d",&no_of_nodes);
    struct node* tree = (struct node *) calloc(no_of_nodes , sizeof(struct node));
    tree[0].depth = 1;
    while(i <  no_of_nodes )
    {
        tree[i].id = i+1;
        scanf("%d %d",&l,&r);
        if(l ==  -1)
```

```
            tree[i].left = NULL;
        else
             {
                 tree[i].left = &tree[l-1];
                 tree[i].left->depth = tree[i].depth + 1;
                 max_depth = tree[i].left->depth;
             }

         if(r ==  -1)
            tree[i].right = NULL;
        else
             {
                 tree[i].right = &tree[r-1];
                 tree[i].right->depth = tree[i].depth + 1;
                 max_depth = tree[i].right->depth+2;
             }

    i++;
    }

    scanf("%d", &i);
    while(i--)
    {
        scanf("%d",&l);
        r = l;
        while(l <= max_depth)
        {
            for(k = 0;k < no_of_nodes; ++k)
            {
                if(tree[k].depth == l)
                {
                    temp = tree[k].left;
                    tree[k].left = tree[k].right;
                    tree[k].right = temp;
                }
            }
            l = l + r;
        }
        inorder(tree);
        printf("\n");
    }



    return 0;
}
```

LAB 9

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

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

#define MAX_SIZE 100

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

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

// Check if the queue is empty
int isEmpty(struct Queue* queue) {
    if (queue->rear == -1)
        return 1;
    else
        return 0;
}

// Check if the queue is full
int isFull(struct Queue* queue) {
    if (queue->rear == MAX_SIZE - 1)
        return 1;
    else
        return 0;
}

// Add an item to the queue
void enqueue(struct Queue* queue, int value) {
    if (isFull(queue))
        printf("\nQueue is Full!!");
    else {
        if (queue->front == -1)
            queue->front = 0;
        queue->rear++;
```

```
        queue->items[queue->rear] = value;
    }
}

// Remove an item from the queue
int dequeue(struct Queue* queue) {
    int item;
    if (isEmpty(queue)) {
        printf("Queue is empty");
        item = -1;
    } else {
        item = queue->items[queue->front];
        queue->front++;
        if (queue->front > queue->rear) {
            queue->front = queue->rear = -1;
        }
    }
    return item;
}

// Graph implementation
struct Graph {
    int vertices;
    int** adjMatrix;
};

// Create a graph with 'vertices' number of vertices
struct Graph* createGraph(int vertices) {
    struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
    graph->vertices = vertices;

    graph->adjMatrix = (int**)malloc(vertices * sizeof(int*));
    for (int i = 0; i < vertices; i++) {
        graph->adjMatrix[i] = (int*)malloc(vertices * sizeof(int));
        for (int j = 0; j < vertices; j++)
            graph->adjMatrix[i][j] = 0;
    }
    return graph;
}

// Add an edge to the graph
void addEdge(struct Graph* graph, int src, int dest) {
    graph->adjMatrix[src][dest] = 1;
    graph->adjMatrix[dest][src] = 1; // Uncomment if the graph is undirected
}

// Breadth First Search traversal
void BFS(struct Graph* graph, int startVertex) {
    int visited[MAX_SIZE] = {0}; // Initialize all vertices as not visited
```

```
    struct Queue* queue = createQueue();

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

    printf("Breadth First Search Traversal: ");

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

        for (int i = 0; i < graph->vertices; i++) {
            if (graph->adjMatrix[currentVertex][i] == 1 && visited[i] == 0) {
                visited[i] = 1;
                enqueue(queue, i);
            }
        }
    }
    printf("\n");
}

int main() {
    int vertices, edges, src, dest;

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

    struct Graph* graph = createGraph(vertices);

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

    for (int i = 0; i < edges; i++) {
        printf("Enter edge %d (source destination): ", i + 1);
        scanf("%d%d", &src, &dest);
        addEdge(graph, src, dest);
    }

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

    BFS(graph, startVertex);

    return 0;
}
```

output :

```
Enter the number of edges: 3
Enter edge 1 (source destination): 0 1
Enter edge 2 (source destination): 0 3
Enter edge 3 (source destination): 1 2
Enter the starting vertex for BFS: 2
Breadth First Search Traversal: 2 1 0 3
```

LAB 9b

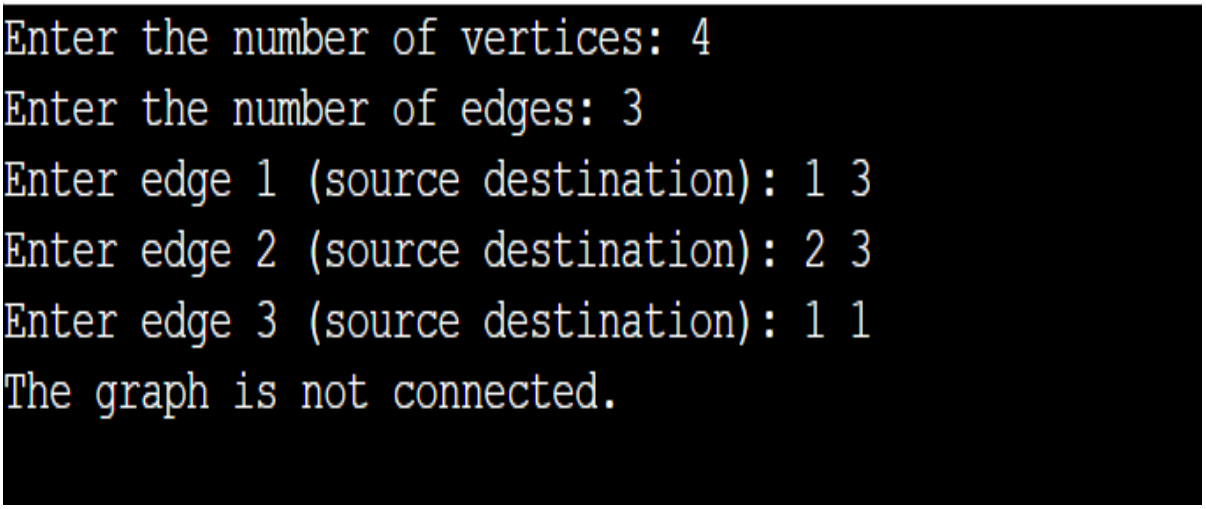
Write a program to check whether given graph is connected or not using the DFS method.

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

#define MAX_SIZE 100

// Graph implementation
struct Graph
{
    int vertices;
    int** adjMatrix;
};

// Create a graph with 'vertices' number of vertices
struct Graph* createGraph(int vertices)
{
    struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
    graph->vertices = vertices;

    graph->adjMatrix = (int**)malloc(vertices * sizeof(int*));
    for (int i = 0; i < vertices; i++)
    {
        graph->adjMatrix[i] = (int*)malloc(vertices * sizeof(int));
        for (int j = 0; j < vertices; j++)
            graph->adjMatrix[i][j] = 0;
    }
    return graph;
}

// Add an edge to the graph
void addEdge(struct Graph* graph, int src, int dest)
{
```

```
    graph->adjMatrix[src][dest] = 1;
    graph->adjMatrix[dest][src] = 1; // Uncomment if the graph is undirected
}

// Depth First Search traversal
void DFS(struct Graph* graph, int startVertex, int visited[])
{
    visited[startVertex] = 1;

    for (int i = 0; i < graph->vertices; i++)
    {
        if (graph->adjMatrix[startVertex][i] == 1 && visited[i] == 0)
            DFS(graph, i, visited);
    }
}

// Check if the graph is connected
int isConnected(struct Graph* graph)
{
    int* visited = (int*)malloc(graph->vertices * sizeof(int));

    for (int i = 0; i < graph->vertices; i++)
        visited[i] = 0;

    DFS(graph, 0, visited);

    for (int i = 0; i < graph->vertices; i++) {
        if (visited[i] == 0)
            return 0; // Graph is not connected
    }
    return 1; // Graph is connected
}

int main()
{
    int vertices, edges, src, dest;

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

    struct Graph* graph = createGraph(vertices);

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

    for (int i = 0; i < edges; i++)
    {
        printf("Enter edge %d (source destination): ", i + 1);
        scanf("%d%d", &src, &dest);
```

```
        addEdge(graph, src, dest);
    }

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

    return 0;
}
```

output :



LAB 10

Given a File of N employee records with a set K of Keys(4-digit) which uniquely determine the records in file F.
Assume that file F is maintained in memory by a Hash Table (HT) of m memory locations with L as the set of memory addresses (2-digit) of locations in HT.
Let the keys in K and addresses in L are integers.
Design and develop a Program in C that uses Hash function H: K -> L as H(K)=K mod m (remainder method), and implement hashing technique to map a given key K to the address space L.
Resolve the collision (if any) using linear probing.

```c
#include <stdio.h>
#define TABLE_SIZE 10
int hashFunction(int key) {
 return key % TABLE_SIZE;
}
void insertValue(int hashTable[], int key) {
 int i = 0;
 int hkey = hashFunction(key);
 int index;
 do {
 index = (hkey + i) % TABLE_SIZE;
 if (hashTable[index] == -1) {
 hashTable[index] = key;
 printf("Inserted key %d at index %d\n", key, index);
 return;
 }
 i++;
 } while (i < TABLE_SIZE);
 printf("Unable to insert key %d. Hash table is full.\n", key);
}
int searchValue(int hashTable[], int key) {
 int i = 0;
 int hkey = hashFunction(key);
 int index;
 do {
 index = (hkey + i) % TABLE_SIZE;
 if (hashTable[index] == key) {
 printf("Key %d found at index %d\n", key, index);
 return index;
 }
```

```
 i++;
 } while (i < TABLE_SIZE);
 printf("Key %d not found in hash table.\n", key);
 return -1;
}
int main() {
 int hashTable[TABLE_SIZE];
 for (int i = 0; i < TABLE_SIZE; i++) {
 hashTable[i] = -1;
 }
 insertValue(hashTable, 1234);
 insertValue(hashTable, 5678);
 insertValue(hashTable, 9876);
 searchValue(hashTable, 5678);
 searchValue(hashTable, 1111);
 return 0;
}

#include <stdio.h>
#define TABLE_SIZE 10
int hashFunction(int key) {
 return key % TABLE_SIZE;
}
void insertValue(int hashTable[], int key) {
 int i = 0;
 int hkey = hashFunction(key);
 int index;
 do {
 index = (hkey + i) % TABLE_SIZE;
 if (hashTable[index] == -1) {
 hashTable[index] = key;
 printf("Inserted key %d at index %d\n", key, index);
 return;
 }
 i++;
 } while (i < TABLE_SIZE);
 printf("Unable to insert key %d. Hash table is full.\n", key);
}
int searchValue(int hashTable[], int key) {
 int i = 0;
 int hkey = hashFunction(key);
 int index;
```

```
 do {
 index = (hkey + i) % TABLE_SIZE;
 if (hashTable[index] == key) {
 printf("Key %d found at index %d\n", key, index);
 return index;
 }
 i++;
 } while (i < TABLE_SIZE);
 printf("Key %d not found in hash table.\n", key);
 return -1;
}
int main() {
 int hashTable[TABLE_SIZE];
 for (int i = 0; i < TABLE_SIZE; i++) {
 hashTable[i] = -1;
 }
 insertValue(hashTable, 1234);
 insertValue(hashTable, 5678);
 insertValue(hashTable, 9876);
 searchValue(hashTable, 5678);
 searchValue(hashTable, 1111);
 return 0;
}
```

Output :

```
Inserted key 1234 at index 4
Inserted key 5678 at index 8
Inserted key 9876 at index 6
Key 5678 found at index 8
Key 1111 not found in hash table.

Process returned 0 (0x0)   execution time : 0.074 s
Press any key to continue.
```