**VISVESVARAYA TECHNOLOGICAL UNIVERSITY**

“JnanaSangama”, Belgaum -590014, Karnataka.

****

**LAB REPORT**

On

**DATA STRUCTURES (23CS3PCDST)**

Submitted by

**Polu Rajeswari Vinuthna (1BM22CS193)**

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 **Polu Rajeswari Vinuthna (1BM22CS193)**, 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** **Dr. Jyothi S Nayak**

Assistant Professor Professor and Head

Department of CSE Department of CSE

BMSCE, Bengaluru BMSCE, Bengaluru

**Table of Contents**

|  |  |  |
| --- | --- | --- |
| Sl No | Title | Page number |
| 1 | Lab 1- Stacks | 5 |
| 2 | Lab 2- Infix, Postfix | 12 |
| 3 | Lab 3- Queues | 17 |
| 4 | Lab 4- Singly Linked List Insert, Leet Code 1 | 23 |
| 5 | Lab 5- Singly Linked List Delete, Leet Code 2 | 29 |
| 6 | Lab 6- Singly Linked List -Stacks-Queues | 38 |
| 7 | Lab 7- Doubly Linked List, Leet Code 3 | 49 |
| 8 | Lab 8- Binary Search Tree, Leet Code 4 | 58 |
| 9 | Lab 9-BFS, DFS, Hackerrank | 63 |
| 10 | Lab 10- Hashing | 69 |

**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**

**1. Swapping using Pointers  
2. Dynamic memory allocation [Program should include malloc, free, calloc, realloc]  
3. Stack implementation [Lab Program: push, pop, display functions to be implemented]**

**Swapping using Pointers**

#include <stdio.h>

void swap(int \*a, int \*b) {

int temp = \*a;

\*a = \*b;

\*b = temp;

}

int main() {

int num1, num2;

printf("Enter the first number: ");

scanf("%d", &num1);

printf("Enter the second number: ");

scanf("%d", &num2);

printf("Before swapping: num1 = %d, num2 = %d\n", num1, num2);

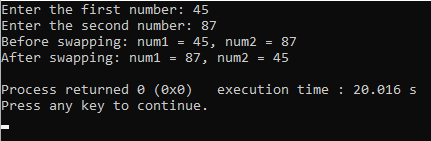
swap(&num1, &num2);

printf("After swapping: num1 = %d, num2 = %d\n", num1, num2);

return 0;

}

**Output**



**Dynamic memory allocation [Program should include malloc, free, calloc, realloc]**

#include <stdio.h>

#include <stdlib.h>

void\* Malloc(size\_t size)

{

return malloc(size);

}

void\* Realloc(void\* ptr, size\_t size)

{

return realloc(ptr, size);

}

void\* Calloc(size\_t num, size\_t size)

{

return calloc(num, size);

}

void Free(void\* ptr)

{

free(ptr);

}

int main()

{

int \*arr1, \*arr2;

size\_t size;

printf("Enter the size of the array: ");

scanf("%d", &size);

arr1 = (int\*)Malloc(size \* sizeof(int));

if (arr1 == NULL)

{

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

return 1;

}

printf("Enter elements of the array:\n");

for (size\_t i = 0; i < size; i++)

{

printf("Element %d: ", i + 1);

scanf("%d", &arr1[i]);

}

printf("Elements of the array (malloc):\n");

for (size\_t i = 0; i < size; i++)

{

printf("%d\n ", arr1[i]);

}

size \*= 2;

arr2 = (int\*)Realloc(arr1, size \* sizeof(int));

if (arr2 == NULL)

{

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

Free(arr1);

return 1;

}

printf("Enter additional elements of the array:\n");

for (size\_t i = size / 2; i < size; i++)

{

printf("Element %d: ", i + 1);

scanf("%d", &arr2[i]);

}

printf("Elements of the array (realloc):\n");

for (size\_t i = 0; i < size; i++)

{

printf("%d ", arr2[i]);

}

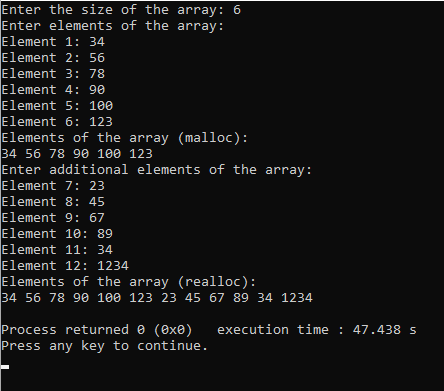
printf("\n");

Free(arr2);

return 0;

}

**Output**



**Stack implementation [Lab Program: push, pop, display functions to be implemented]**

#include <stdio.h>

#include <stdlib.h>

#define MAX\_SIZE 10

struct Stack {

int items[MAX\_SIZE];

int top;

};

void initialize(struct Stack \*stack)

{

stack->top = -1;

}

int isEmpty(struct Stack \*stack)

{

return stack->top == -1;

}

int isFull(struct Stack \*stack)

{

return stack->top == MAX\_SIZE - 1;

}

void push(struct Stack \*stack, int value)

{

if (isFull(stack))

{

printf("Stack overflow. Cannot push %d.\n", value);

} else

{

stack->top++;

stack->items[stack->top] = value;

printf("Pushed %d onto the stack.\n", value);

}

}

int pop(struct Stack \*stack)

{

int poppedValue = -1;

if (isEmpty(stack))

{

printf("Stack underflow. Cannot pop from an empty stack.\n");

} else

{

poppedValue = stack->items[stack->top];

stack->top--;

printf("Popped %d from the stack.\n", poppedValue);

}

return poppedValue;

}

void display(struct Stack \*stack)

{

if (isEmpty(stack))

{

printf("Stack is empty.\n");

} else

{

printf("Elements in the stack: ");

for (int i = 0; i <= stack->top; i++)

{

printf("%d ", stack->items[i]);

}

printf("\n");

}

}

int main()

{

struct Stack stack;

initialize(&stack);

push(&stack, 50);

push(&stack, 100);

push(&stack, 900);

display(&stack);

pop(&stack);

display(&stack);

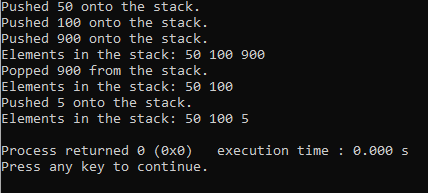
push(&stack, 5);

display(&stack);

return 0;

}

**Output**



**Lab 2**

**1.Infix to Postfix conversion  
2.  Evaluation of Postfix Expression**

**Infix to Postfix conversion**

#include <stdio.h>

#include<ctype.h>

#define SIZE 50

char stack[SIZE];

int top=-1;

push(char ele)

{

stack[++top]=ele;

}

char pop()

{

return(stack[top--]);

}

int pr(char symbol)

{

if (symbol == '^')

{

return(3);

}

else if (symbol == '\*' || symbol == '/')

{

return(2);

}

else if (symbol == '+' || symbol == '-')

{

return(1);

}

else

{

return (0);

}

}

void main()

{

char infix[50],postfix[50],ch,ele;

int i=0,k=0;

printf("enter the infix expression:");

scanf("%s",infix);

push('#');

while( (ch=infix[i++]) != '\0')

{

if(ch=='(') push(ch);

else

if(isalnum(ch)) postfix[k++]=ch;

else

if(ch ==')')

{

while(stack[top]!='(')

postfix[k++]=pop();

ele=pop();

}

else

{

while(pr(stack[top])>=pr(ch))

postfix[k++]=pop();

push(ch);

}

}

while(stack[top]!='#')

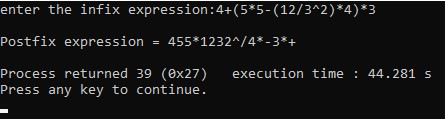
postfix[k++]=pop();

postfix[k]='\0';

printf("\nPostfix expression = %s\n",postfix);

}

**Output**



**Evaluation of Postfix Expression**

#include<stdio.h>

#include <ctype.h>

int stack[20];

int top=-1;

void push(int x)

{

stack[++top]=x;

}

int pop()

{

return stack[top--];

}

int main()

{

char exp[20];

char \*e;

int n1,n2,n3,num;

printf("enter the expresion :: ");

scanf("%s",exp);

e=exp;

while(\*e!= '\0')

{

if(isdigit(\*e))

{

num=\*e-48;

push(num);

}

else

{

n1=pop();

n2=pop();

switch(\*e)

{

case '+':

{

n3=n1+n2;

break;

}

case '-':

{

n3=n1-n2;

break;

}

case '\*':

{

n3=n1\*n2;

break;

}

case '/':

{

n3=n1/n2;

break;

}

}

push(n3);

}

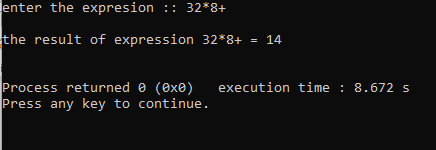
e++;

}

printf("\nthe result of expression %s = %d\n\n",exp,pop());

}

**Output**



**Lab 3:   
3a. Queue Implementation  
3b. Circular Queue Implementation**

**3a.Queue Implementation**

#include<stdio.h>

#define MAX 50

int queue\_array[MAX];

int rear=-1;

int front=-1;

display()

{

int i;

if(front==-1)

printf("queue is empty\n");

else

{

printf("queue is :\n");

for(i=front;i<=rear;i++)

printf("%d",queue\_array[i]);

printf("\n");

}

}

main()

{

int choice;

while(1)

{

printf("1.insert\n");

printf("2.delete\n");

printf("3.display\n");

printf("4.exit\n");

printf("enter your choice:");

scanf("%d",&choice);

switch(choice)

{

case 1:

insert();

break;

case 2:

delete();

break;

case 3:

display();

break;

case 4:

exit(1);

break;

default:

printf("invalid choice\n");

}

}

}

insert()

{

int add\_item;

if(rear==MAX-1)

printf("queue overflow\n");

else

{

if(front==-1)

front=0;

printf("insert the element in the queue:");

scanf("%d",&add\_item);

rear+=1;

queue\_array[rear]=add\_item;

}

}

delete()

{

if(front==-1 || front>rear)

{

printf("queue underflow\n");

return;

}

else

{

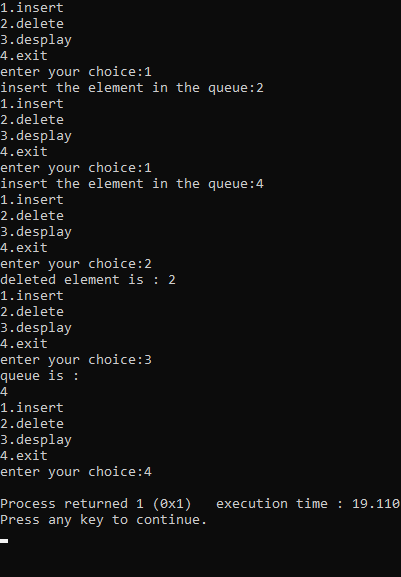
printf("deleted element is : %d\n",queue\_array[front]);

front+=1;

}

}

**Output**



**3b. Circular Queue Implementation**

#include<stdio.h>

#define SIZE 5

int items[SIZE];

int front = -1, rear = -1;

int isFull() {

if ((front == rear + 1) || (front == 0 && rear == SIZE - 1))

return 1;

return 0;

}

int isEmpty() {

if (front == -1)

return 1;

return 0;

}

void enQueue(int element) {

if (isFull())

printf("\nQueue is full");

else {

if (front == -1)

front = 0;

rear = (rear + 1) % SIZE;

items[rear] = element;

printf("\nInserted -> %d", element);

}

}

int deQueue() {

int element;

if (isEmpty()) {

printf("\nQueue is empty");

return -1;

} else {

element = items[front];

if (front == rear) {

front = -1;

rear = -1;

} else {

front = (front + 1) % SIZE;

}

printf("\nDeleted element -> %d\n", element);

return element;

}

}

void display() {

int i;

if (isEmpty())

printf("\nEmpty queue\n");

else {

printf("\nFront -> %d", front);

printf("\nItems -> ");

for (i = front; i != rear; i = (i + 1) % SIZE) {

printf("%d ", items[i]);

}

printf("%d", items[i]);

printf("\nRear -> %d\n", rear);

}

}

int main() {

enQueue(1);

enQueue(2);

enQueue(3);

enQueue(4);

enQueue(5);

display();

deQueue();

deQueue();

display();

enQueue(6);

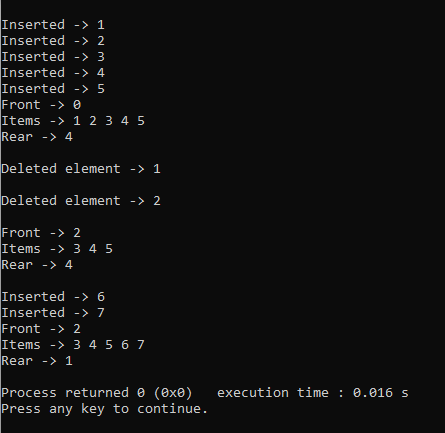
enQueue(7);

display();

return 0;

}

**Output**

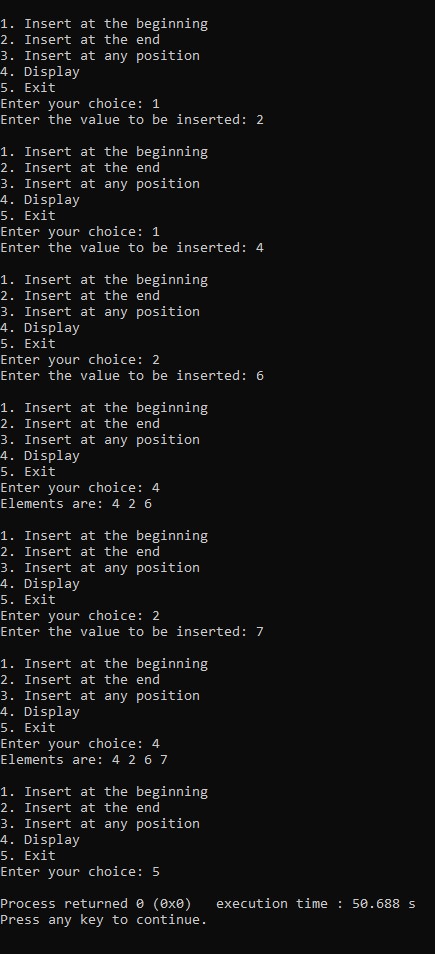
****

**Lab 4**

**Singly Linked List Insert and display Implementation**

#include<stdio.h>  
#include<stdlib.h>  
struct node {  
    int data;  
    struct node \*next;  
};  
struct node \*head = NULL;  
void display() {  
    struct node \*ptr = head;  
    if (ptr == NULL) {  
        printf("List is empty\n");  
        return;  
    }  
    printf("Elements are: ");  
    while (ptr != NULL) {  
        printf("%d ", ptr->data);  
        ptr = ptr->next;  
    }  
    printf("\n");  
}  
void insert\_begin() {  
    struct node \*temp;  
    temp = (struct node \*)malloc(sizeof(struct node));  
    printf("Enter the value to be inserted: ");  
    scanf("%d", &temp->data);  
    temp->next = head;  
    head = temp;  
}  
void insert\_end() {  
    struct node \*temp, \*ptr;  
    temp = (struct node \*)malloc(sizeof(struct node));  
    printf("Enter the value to be inserted: ");  
    scanf("%d", &temp->data);  
    temp->next = NULL;  
    if (head == NULL) {  
        head = temp;  
    } else {  
        ptr = head;  
        while (ptr->next != NULL) {  
            ptr = ptr->next;  
        }  
        ptr->next = temp;  
    }  
}  
void insert\_pos() {  
    int pos, i;  
    struct node \*temp, \*ptr;  
    temp = (struct node \*)malloc(sizeof(struct node));  
    printf("Enter the position to insert: ");  
    scanf("%d", &pos);  
    printf("Enter the value to be inserted: ");  
    scanf("%d", &temp->data);  
    temp->next = NULL;  
  
    if (pos == 0) {  
        temp->next = head;  
        head = temp;  
    } else {  
        ptr = head;  
        for (i = 0; i < pos - 1; i++) {  
            ptr = ptr->next;  
            if (ptr == NULL) {  
                printf("Position not found\n");  
                return;  
            }  
        }  
        temp->next = ptr->next;  
        ptr->next = temp;  
    }  
}  
int main() {  
    int choice;  
    while(1) {  
        printf("\n1. Insert at the beginning\n2. Insert at the end\n3. Insert at any position\n4. Display\n5. Exit\n");  
        printf("Enter your choice: ");  
        scanf("%d", &choice);  
  
        switch(choice) {  
            case 1:  
                insert\_begin();  
                break;  
            case 2:  
                insert\_end();  
                break;  
            case 3:  
                insert\_pos();  
                break;  
            case 4:  
                display();  
                break;  
            case 5:  
                exit(0);  
                break;  
            default:  
                printf("Enter the correct choice\n");  
        }  
    }  
    return 0;  
}

**Output(next page)**



**Leet code- Min stack**

#include <stdlib.h>

typedef struct {

int \*stack;

int \*minStack;

int top

} MinStack;

MinStack\* minStackCreate() {

MinStack\* stack = (MinStack\*)malloc(sizeof(MinStack));

stack->stack = (int\*)malloc(sizeof(int) \* 50);

stack->minStack = (int\*)malloc(sizeof(int) \* 50);

stack->top = -1;

return stack;

}

void minStackPush(MinStack\* obj, int val) {

obj->top++;

obj->stack[obj->top] = val;

if (obj->top == 0 || val <= obj->minStack[obj->top - 1]) {

obj->minStack[obj->top] = val;

} else {

obj->minStack[obj->top] = obj->minStack[obj->top - 1];

}

}

void minStackPop(MinStack\* obj) {

obj->top--;

}

int minStackTop(MinStack\* obj) {

return obj->stack[obj->top];

}

int minStackGetMin(MinStack\* obj) {

return obj->minStack[obj->top];

}

void minStackFree(MinStack\* obj) {

free(obj->stack);

free(obj->minStack);

free(obj);

}

/\*\*

 \* Your MinStack struct will be instantiated and called as such:

 \* MinStack\* obj = minStackCreate();

\* minStackPush(obj, val);

 \* minStackPop(obj);

 \* int param\_3 = minStackTop(obj);

 \* int param\_4 = minStackGetMin(obj);

 \* minStackFree(obj);

\*/

**Output**

****

**Lab 5**

**Singly Linked List delete and display Implementation**

#include<stdio.h>

#include<stdlib.h>

struct node {

int data;

struct node \*next;

};

struct node \*head = NULL;

void display() {

printf("Elements are: ");

struct node \*ptr = head;

while (ptr != NULL) {

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

ptr = ptr->next;

}

printf("NULL\n");

}

void insert\_begin() {

struct node \*temp = (struct node\*)malloc(sizeof(struct node));

printf("Enter the value to be inserted: ");

scanf("%d", &temp->data);

temp->next = head;

head = temp;

}

void delete\_begin() {

if (head == NULL) {

printf("List is empty. Deletion not possible.\n");

return;

}

struct node \*temp = head;

head = head->next;

printf("Element deleted from the beginning: %d\n", temp->data);

free(temp);

}

void delete\_end() {

if (head == NULL) {

printf("List is empty. Deletion not possible.\n");

return;

}

struct node \*temp, \*prev;

temp = head;

while (temp->next != NULL) {

prev = temp;

temp = temp->next;

}

if (temp == head) {

head = NULL;

} else {

prev->next = NULL;

}

printf("Element deleted from the end: %d\n", temp->data);

free(temp);

}

void delete\_at\_position() {

int position;

printf("Enter the position to delete: ");

scanf("%d", &position);

if (head == NULL) {

printf("List is empty. Deletion not possible.\n");

return;

}

struct node \*temp, \*prev;

temp = head;

if (position == 0) {

head = head->next;

printf("Element at position %d deleted successfully.\n", position);

free(temp);

return;

}

for (int i = 0; temp != NULL && i < position; i++) {

prev = temp;

temp = temp->next;

}

if (temp == NULL) {

printf("Position %d is out of bounds.\n", position);

return;

}

prev->next = temp->next;

printf("Element at position %d deleted successfully.\n", position);

free(temp);

}

int main() {

int choice;

while (1) {

printf("\n 1. to insert at the beginning\n 2. to delete beginning\n 3. to delete at end\n 4. to delete at any position\n 5. to display\n 6. to exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

insert\_begin();

break;

case 2:

delete\_begin();

break;

case 3:

delete\_end();

break;

case 4:

delete\_at\_position();

break;

case 5:

display();

break;

case 6:

exit(0);

break;

default:

printf("Enter the correct choice\n");

break;

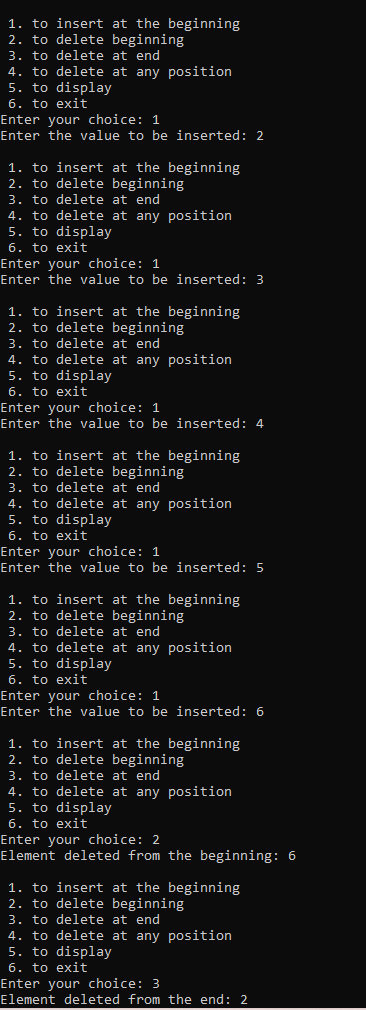
}

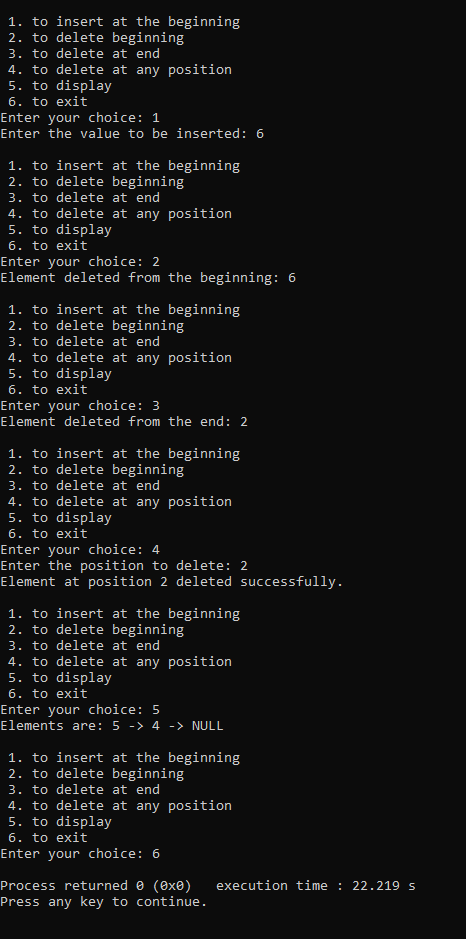
}

return 0;

}

**Output(next page)**





**Leet code- reverse linked list ll**

/\*\*

\* Definition for singly-linked list.

\* struct ListNode {

\* int val;

\* struct ListNode \*next;

\* };

\*/

struct ListNode\* reverseBetween(struct ListNode\* start, int a, int b)

{

a-=1;

b-=1;

struct ListNode \*node1=NULL,\*node2=NULL,\*nodeb=NULL,\*nodea=NULL,\*ptr=start;

int c=0;

while(ptr!=NULL)

{

if(c==a-1)

nodeb=ptr;

else if(c==a)

node1=ptr;

else if(c==b)

node2=ptr;

else if(c==b+1)

{

nodea=ptr;

break;

}

c+=1;

ptr=ptr->next;

}

struct ListNode\*pre=nodea,\*temp;

ptr=start;

c=0;

while(ptr!=NULL)

{

if(c>=a && c<b)

{

temp=ptr->next;

ptr->next=pre;

pre=ptr;

ptr=temp;

}

else if(c==b)

{

ptr->next=pre;

if(a==0)

start=ptr;

else

nodeb->next=ptr;

break;

}

else

ptr=ptr->next;

c+=1;

}

return start;

}

**Output**

****

****

**Lab 6**

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

#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 insertEnd(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;

}

}

void display(struct Node\* head) {

struct Node\* temp = head;

while (temp != NULL) {

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

temp = temp->next;

}

printf("NULL\n");

}

void sortLinkedList(struct Node\* head) {

int swapped, i;

struct Node\* ptr;

struct Node\* lptr = NULL;

if (head == NULL)

return;

do {

swapped = 0;

ptr = head;

while (ptr->next != lptr) {

if (ptr->data > ptr->next->data) {

int temp = ptr->data;

ptr->data = ptr->next->data;

ptr->next->data = temp;

swapped = 1;

}

ptr = ptr->next;

}

lptr = ptr;

} while (swapped);

}

struct Node\* reverseLinkedList(struct Node\* head) {

struct Node \*prev = NULL, \*current = head, \*next = NULL;

while (current != NULL) {

next = current->next;

current->next = prev;

prev = current;

current = next;

}

return prev;

}

void concatenateLinkedLists(struct Node\*\* list1, struct Node\* list2) {

if (\*list1 == NULL) {

\*list1 = list2;

} else {

struct Node\* temp = \*list1;

while (temp->next != NULL) {

temp = temp->next;

}

temp->next = list2;

}

}

int main() {

struct Node\* list1 = NULL;

struct Node\* list2 = NULL;

int n, value;

printf("Enter the number of elements for list 1: ");

scanf("%d", &n);

printf("Enter the elements for list 1:\n");

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

scanf("%d", &value);

insertEnd(&list1, value);

}

printf("Enter the number of elements for list 2: ");

scanf("%d", &n);

printf("Enter the elements for list 2:\n");

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

scanf("%d", &value);

insertEnd(&list2, value);

}

sortLinkedList(list1);

printf("Sorted List 1: ");

display(list1);

list2 = reverseLinkedList(list2);

printf("Reversed List 2: ");

display(list2);

concatenateLinkedLists(&list1, list2);

printf("Concatenated List: ");

display(list1);

struct Node\* temp;

while (list1 != NULL) {

temp = list1;

list1 = list1->next;

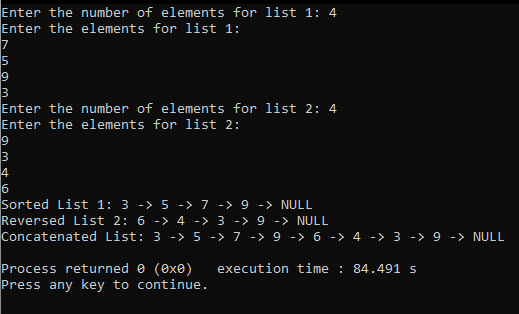
free(temp);

}

return 0;

}

**Output:**

****

**WAP to Implement Single Linked List to simulate Stack**

#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 push(struct Node\*\* top, int value) {

struct Node\* newNode = createNode(value);

newNode->next = \*top;

\*top = newNode;

}

int pop(struct Node\*\* top) {

if (\*top == NULL) {

printf("Stack underflow!\n");

return -1;

}

struct Node\* temp = \*top;

int poppedValue = temp->data;

\*top = temp->next;

free(temp);

return poppedValue;

}

void displayStack(struct Node\* top) {

printf("Stack: ");

while (top != NULL) {

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

top = top->next;

}

printf("\n");

}

int main() {

struct Node\* top = NULL;

int choice, value;

do {

printf("\nStack Operations:\n");

printf("1. Push\n");

printf("2. Pop\n");

printf("3. Display\n");

printf("4. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter the value to push: ");

scanf("%d", &value);

push(&top, value);

break;

case 2:

value = pop(&top);

if (value != -1) {

printf("Popped value: %d\n", value);

}

break;

case 3:

displayStack(top);

break;

case 4:

printf("Exiting the program.\n");

break;

default:

printf("Invalid choice! Please enter a valid option.\n");

}

} while (choice != 4);

struct Node\* temp;

while (top != NULL) {

temp = top;

top = top->next;

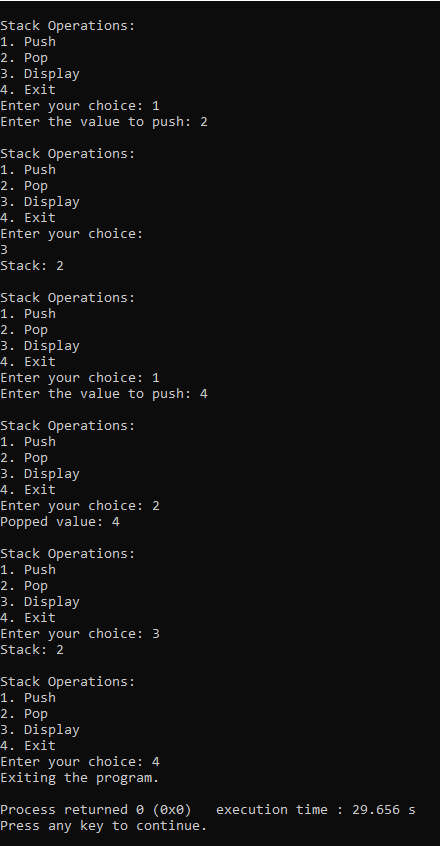
free(temp);

}

return 0;

}

**Output (next page)**

****

**WAP to Implement Single Linked List to simulate Queue Operations.**

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* next;

};

struct Queue {

struct Node\* front;

struct Node\* rear;

};

struct Node\* createNode(int value) {

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

newNode->data = value;

newNode->next = NULL;

return newNode;

}

struct Queue\* createQueue() {

struct Queue\* queue = (struct Queue\*)malloc(sizeof(struct Queue));

queue->front = queue->rear = NULL;

return queue;

}

void enqueue(struct Queue\* queue, int value) {

struct Node\* newNode = createNode(value);

if (queue->rear == NULL) {

queue->front = queue->rear = newNode;

return;

}

queue->rear->next = newNode;

queue->rear = newNode;

}

int dequeue(struct Queue\* queue) {

if (queue->front == NULL) {

printf("Queue underflow!\n");

return -1;

}

struct Node\* temp = queue->front;

int dequeuedValue = temp->data;

queue->front = temp->next;

if (queue->front == NULL) {

queue->rear = NULL;

}

free(temp);

return dequeuedValue;

}

void displayQueue(struct Queue\* queue) {

struct Node\* temp = queue->front;

printf("Queue: ");

while (temp != NULL) {

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

temp = temp->next;

}

printf("\n");

}

int main() {

struct Queue\* queue = createQueue();

int choice, value;

do {

printf("\nQueue Operations:\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 value to enqueue: ");

scanf("%d", &value);

enqueue(queue, value);

break;

case 2:

value = dequeue(queue);

if (value != -1) {

printf("Dequeued value: %d\n", value);

}

break;

case 3:

displayQueue(queue);

break;

case 4:

printf("Exiting the program.\n");

break;

default:

printf("Invalid choice! Please enter a valid option.\n");

}

} while (choice != 4);

struct Node\* temp;

while (queue->front != NULL) {

temp = queue->front;

queue->front = queue->front->next;

free(temp);

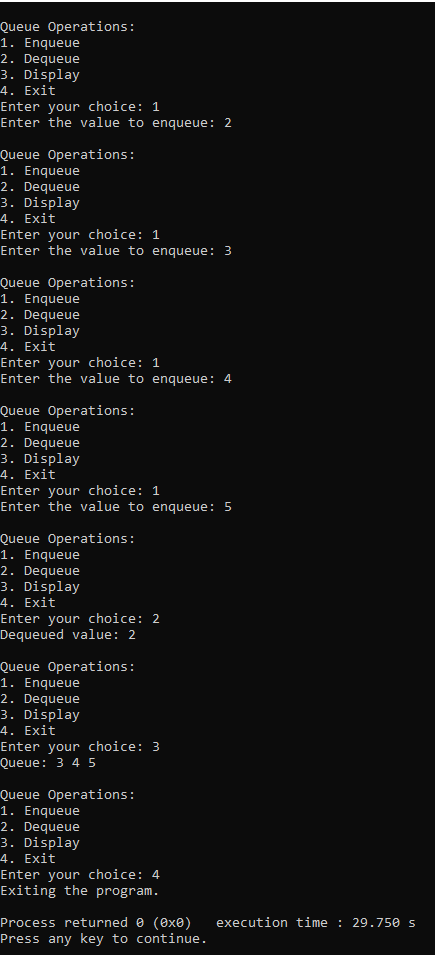
}

free(queue);

return 0;

}

**Output (next page)**



**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  
d) Display the contents of the list**

#include<stdio.h>

#include<stdlib.h>

struct node {

int data;

struct node \*prev;

struct node \*next;

};

struct node \*s1 = NULL;

struct node \*createNode(int value) {

struct node \*temp = (struct node \*)malloc(sizeof(struct node));

temp->data = value;

temp->next = NULL;

temp->prev = NULL;

return temp;

}

struct node \*insert\_left(struct node \*start) {

int value, key;

struct node \*temp = createNode(0);

printf("Enter the value to be inserted: ");

scanf("%d", &temp->data);

printf("Enter the value to the left of which the node has to be inserted: ");

scanf("%d", &key);

struct node \*ptr = start;

while (ptr != NULL && ptr->data != key) {

ptr = ptr->next;

}

if (ptr == NULL) {

printf("Node with value %d not found\n", key);

free(temp);

} else {

temp->next = ptr;

temp->prev = ptr->prev;

if (ptr->prev != NULL) {

ptr->prev->next = temp;

}

ptr->prev = temp;

if (ptr == start) {

start = temp;

}

}

return start;

}

struct node \*delete\_value(struct node \*start) {

int value;

printf("Enter the value to be deleted: ");

scanf("%d", &value);

struct node \*ptr = start;

while (ptr != NULL && ptr->data != value) {

ptr = ptr->next;

}

if (ptr == NULL) {

printf("Node with value %d not found\n", value);

} else {

if (ptr->prev != NULL) {

ptr->prev->next = ptr->next;

} else {

start = ptr->next;

}

if (ptr->next != NULL) {

ptr->next->prev = ptr->prev;

}

printf("Node with value %d deleted\n", value);

free(ptr);

}

return start;

}

void display(struct node \*start) {

struct node \*ptr = start;

if (start == NULL) {

printf("List is empty\n");

} else {

printf("List contents:\n");

while (ptr != NULL) {

printf("%d\n", ptr->data);

ptr = ptr->next;

}

}

}

int main() {

int choice;

while (1) {

printf("\n1. Create a doubly linked list\n2. Insert to the left of a node\n3. Delete based on a specific value\n4. Display the contents\n5. Exit\n");

scanf("%d", &choice);

switch (choice) {

case 1:

s1 = createNode(0);

printf("Doubly linked list created\n");

break;

case 2:

s1 = insert\_left(s1);

break;

case 3:

s1 = delete\_value(s1);

break;

case 4:

display(s1);

break;

case 5:

printf("Exiting the program\n");

exit(0);

default:

printf("Invalid choice\n");

}

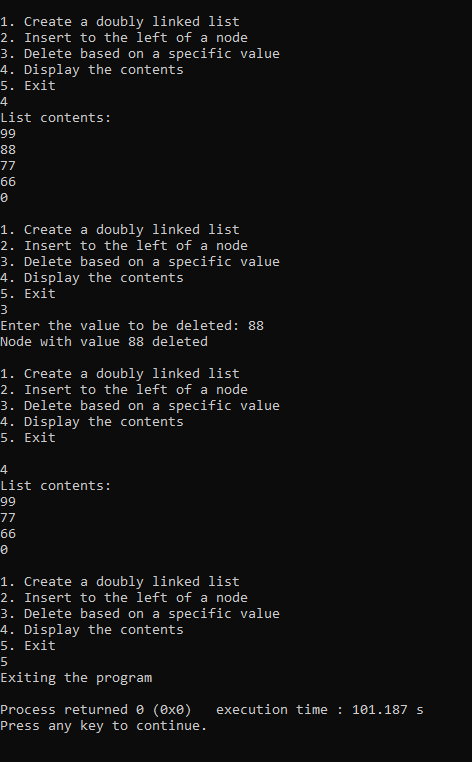
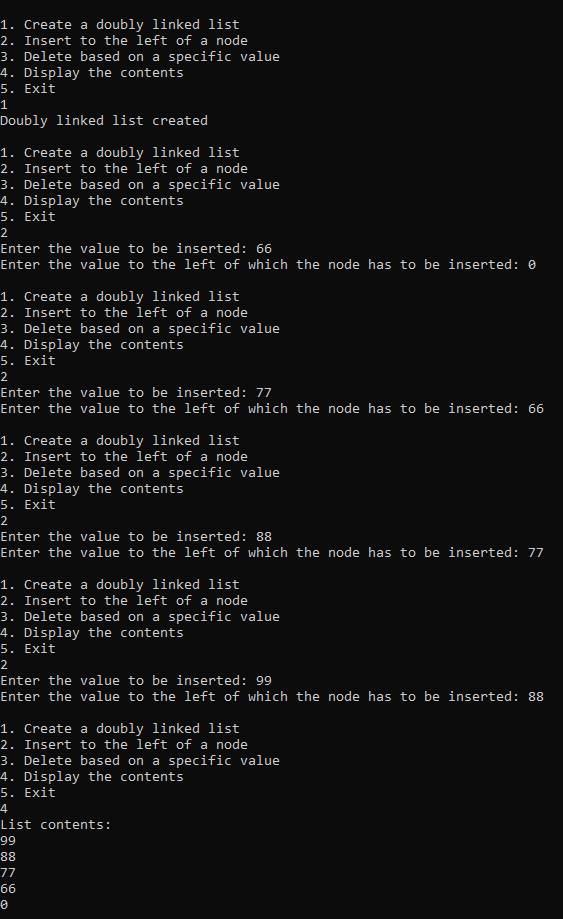
}

return 0;

}

**Output**

**(next page)**



**Leet code**

/\*\* \* Definition for singly-linked list.

\* struct ListNode {

\*     int val;

\*     struct ListNode

\*next;

\* };

\*/

/\*\*

\* Note: The returned array must be malloced, assume caller calls free().

\*/

struct ListNode\*\* splitListToParts(struct ListNode\* head, int k, int\* returnSize) {

    struct ListNode\* current = head;

    int length = 0;

    while (current) {

        length++;

        current = current->next;

    }

    int part\_size = length / k;

    int extra\_nodes = length % k;

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

    current = head;

    for (int i = 0; i < k; i++) {

        struct ListNode\* part\_head = current;

        int part\_length = part\_size + (i < extra\_nodes ? 1 : 0);

        for (int j = 0; j < part\_length - 1 && current; j++) {

            current = current->next;

        }

        if (current) {

            struct ListNode\* next\_node = current->next;

            current->next = NULL;

            result[i] = part\_head;

            current = next\_node;

        } else {

            result[i] = NULL;

        }

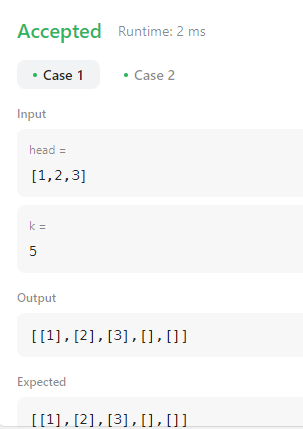
    }

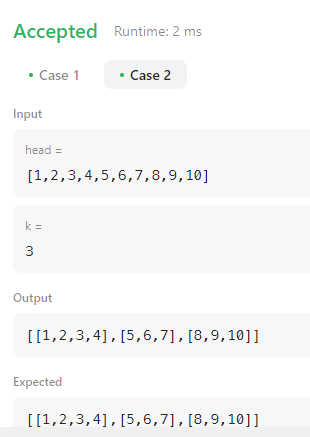
    \*returnSize = k;

    return result;

}

**output**



******

**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 postorder  
c. To display the elements in the tree.**

#include <stdio.h>

#include <stdlib.h>

struct TreeNode {

int data;

struct TreeNode\* left;

struct TreeNode\* right;

};

struct TreeNode\* createNode(int data) {

struct TreeNode\* newNode = (struct TreeNode\*)malloc(sizeof(struct TreeNode));

newNode->data = data;

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

return newNode;

}

struct TreeNode\* insertNode(struct TreeNode\* 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;

}

void inOrderTraversal(struct TreeNode\* root) {

if (root != NULL) {

inOrderTraversal(root->left);

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

inOrderTraversal(root->right);

}

}

void preOrderTraversal(struct TreeNode\* root) {

if (root != NULL) {

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

preOrderTraversal(root->left);

preOrderTraversal(root->right);

}

}

void postOrderTraversal(struct TreeNode\* root) {

if (root != NULL) {

postOrderTraversal(root->left);

postOrderTraversal(root->right);

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

}

}

void displayTree(struct TreeNode\* root) {

printf("In-order traversal: ");

inOrderTraversal(root);

printf("\n");

printf("Pre-order traversal: ");

preOrderTraversal(root);

printf("\n");

printf("Post-order traversal: ");

postOrderTraversal(root);

printf("\n");

}

int main() {

struct TreeNode\* root = NULL;

int choice, data;

do {

printf("1. Insert a node\n");

printf("2. Display tree\n");

printf("3. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter data to insert: ");

scanf("%d", &data);

root = insertNode(root, data);

break;

case 2:

if (root == NULL) {

printf("Tree is empty.\n");

} else {

displayTree(root);

}

break;

case 3:

printf("Exiting program.\n");

break;

default:

printf("Invalid choice. Please try again.\n");

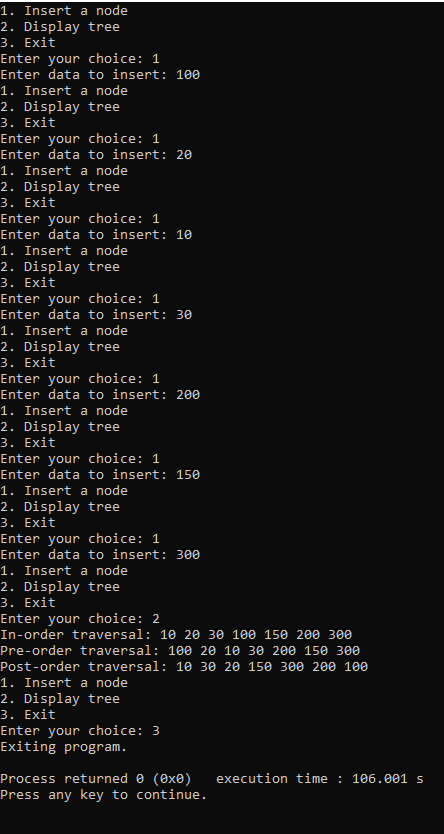
}

} while (choice != 3);

return 0;

}

**Output**

****

**Leet code**

/\*\*

\* Definition for singly-linked list.

\* struct ListNode {

\* int val;

\* struct ListNode \*next;

\* };

\*/

struct ListNode\* rotateRight(struct ListNode\* head, int k) {

if (head == NULL || k == 0) {

return head;

}

struct ListNode\* current = head;

int length = 1;

while (current->next != NULL) {

current = current->next;

length++;

}

k = k % length;

if (k == 0) {

return head;

}

current = head;

for (int i = 1; i < length - k; i++) {

current = current->next;

}

struct ListNode\* newHead = current->next;

current->next = NULL;

current = newHead;

while (current->next != NULL) {

current = current->next;

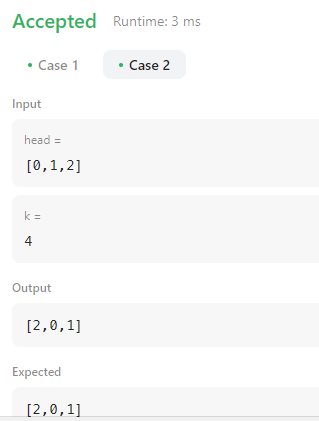
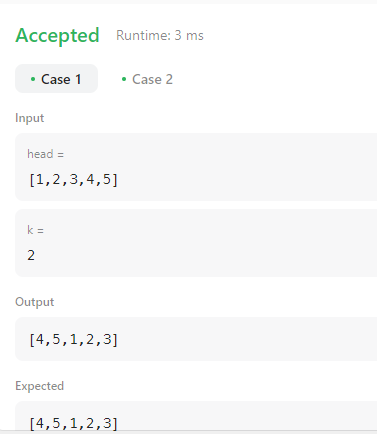
}

current->next = head;

return newHead;

}

**Output**

****

**Lab 9**

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

**b) Write a program to check whether given graph is connected or not using**

**DFS method.**

**A)**

#include <stdio.h>

#include <stdlib.h>

#define MAX 100

void BFS(int adjacency\_matrix[MAX][MAX], int visited[MAX], int n, int start) {

int queue[MAX], front = -1, rear = -1;

visited[start] = 1;

queue[++rear] = start;

while (front != rear) {

int current = queue[++front];

printf("%d ", current + 1);

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

if (adjacency\_matrix[current][i] == 1 && !visited[i]) {

visited[i] = 1;

queue[++rear] = i;

}

}

}

}

int main() {

int adjacency\_matrix[MAX][MAX], visited[MAX], n, i, j;

printf("Enter the number of nodes: ");

scanf("%d", &n);

printf("Enter the adjacency matrix:\n");

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

visited[i] = 0; // Initialize visited array

for (j = 0; j < n; j++) {

scanf("%d", &adjacency\_matrix[i][j]);

}

}

int start\_node;

printf("Enter the starting node for BFS traversal (1 to %d): ", n);

scanf("%d", &start\_node);

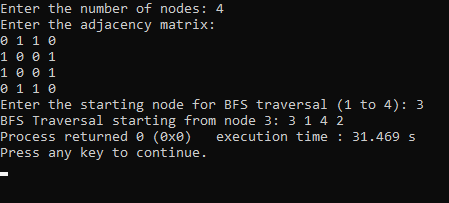
printf("BFS Traversal starting from node %d: ", start\_node);

BFS(adjacency\_matrix, visited, n, start\_node - 1);

return 0;

}

**Output:**

****

**B)**

#include <stdio.h>

#include <stdlib.h>

#define MAX 100

void DFS(int adjacency\_matrix[MAX][MAX], int visited[MAX], int n, int current) {

visited[current] = 1;

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

if (adjacency\_matrix[current][i] == 1 && !visited[i]) {

DFS(adjacency\_matrix, visited, n, i);

}

}

}

int isConnected(int adjacency\_matrix[MAX][MAX], int visited[MAX], int n) {

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

visited[i] = 0;

}

DFS(adjacency\_matrix, visited, n, 0);

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

if (!visited[i]) {

return 0;

}

}

return 1;

}

int main() {

int adjacency\_matrix[MAX][MAX], visited[MAX], n, i, j;

printf("Enter the number of nodes: ");

scanf("%d", &n);

printf("Enter the adjacency matrix:\n");

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

for (j = 0; j < n; j++) {

scanf("%d", &adjacency\_matrix[i][j]);

}

}

if (isConnected(adjacency\_matrix, visited, n)) {

printf("The graph is connected.\n");

} else {

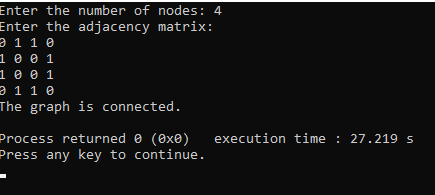
printf("The graph is not connected.\n");

}

return 0;

}

**Output:**



**Hacker Rank on Tree**

#include <assert.h>

#include <stdbool.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

typedef struct Node {

    int data;

    struct Node\* left;

    struct Node\* right;

} Node;

Node\* createNode(int data) {

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

    newNode->data = data;

    newNode->left = NULL;

    newNode->right = NULL;

    return newNode;

}

void inOrderTraversal(Node\* root, int\* result, int\* index) {

    if (root == NULL) return;

    inOrderTraversal(root->left, result, index);

    result[(\*index)++] = root->data;

    inOrderTraversal(root->right, result, index);

}

void swapAtLevel(Node\* root, int k, int level) {

    if (root == NULL) return;

    if (level % k == 0) {

        Node\* temp = root->left;

        root->left = root->right;

        root->right = temp;

    }

    swapAtLevel(root->left, k, level + 1);

    swapAtLevel(root->right, k, level + 1);

}

int\*\* swapNodes(int indexes\_rows, int indexes\_columns, int\*\* indexes, int queries\_count, int\* queries, int\* result\_rows, int\* result\_columns) {

    Node\*\* nodes = (Node\*\*)malloc((indexes\_rows + 1) \* sizeof(Node\*));

    for (int i = 1; i <= indexes\_rows; i++) {

        nodes[i] = createNode(i);

    }

    for (int i = 0; i < indexes\_rows; i++) {

        int leftIndex = indexes[i][0];

        int rightIndex = indexes[i][1];

        if (leftIndex != -1) nodes[i + 1]->left = nodes[leftIndex];

        if (rightIndex != -1) nodes[i + 1]->right = nodes[rightIndex];

    }

    int\*\* result = (int\*\*)malloc(queries\_count \* sizeof(int\*));

    \*result\_rows = queries\_count;

    \*result\_columns = indexes\_rows;

    for (int i = 0; i < queries\_count; i++) {

        swapAtLevel(nodes[1], queries[i], 1);

        int\* traversalResult = (int\*)malloc(indexes\_rows \* sizeof(int));

        int index = 0;

        inOrderTraversal(nodes[1], traversalResult, &index);

        result[i] = traversalResult;

    }

    free(nodes);

    return result;

}

int main() {

    int n;

    scanf("%d", &n);

    int\*\* indexes = malloc(n \* sizeof(int\*));

    for (int i = 0; i < n; i++) {

        indexes[i] = malloc(2 \* sizeof(int));

        scanf("%d %d", &indexes[i][0], &indexes[i][1]);

    }

    int queries\_count;

    scanf("%d", &queries\_count);

    int\* queries = malloc(queries\_count \* sizeof(int));

    for (int i = 0; i < queries\_count; i++) {

        scanf("%d", &queries[i]);

    }

    int result\_rows;

    int result\_columns;

    int\*\* result = swapNodes(n, 2, indexes, queries\_count, queries, &result\_rows, &result\_columns);

    for (int i = 0; i < result\_rows; i++) {

        for (int j = 0; j < result\_columns; j++) {

            printf("%d ", result[i][j]);

        }

        printf("\n");

        free(result[i]);

    }

    free(result);

    for (int i = 0; i < n; i++) {

        free(indexes[i]);

    }

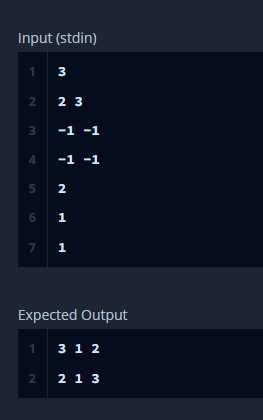
    free(indexes);

    free(queries);

    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.**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define TABLE\_SIZE 100

#define KEY\_LENGTH 5

#define MAX\_NAME\_LENGTH 50

#define MAX\_DESIGNATION\_LENGTH 50

struct Employee {

char key[KEY\_LENGTH];

char name[MAX\_NAME\_LENGTH];

char designation[MAX\_DESIGNATION\_LENGTH];

float salary;

};

struct HashTable {

struct Employee\* table[TABLE\_SIZE];

};

int hash\_function(const char\* key, int m) {

int sum = 0;

for (int i = 0; key[i] != '\0'; i++) {

sum += key[i];

}

return sum % m;

}

void insert(struct HashTable\* ht, struct Employee\* emp) {

int index = hash\_function(emp->key, TABLE\_SIZE);

while (ht->table[index] != NULL) {

index = (index + 1) % TABLE\_SIZE;

}

ht->table[index] = emp;

}

struct Employee\* search(struct HashTable\* ht, const char\* key) {

int index = hash\_function(key, TABLE\_SIZE);

while (ht->table[index] != NULL) {

if (strcmp(ht->table[index]->key, key) == 0) {

return ht->table[index];

}

index = (index + 1) % TABLE\_SIZE;

}

return NULL;

}

int main() {

struct HashTable ht;

struct Employee\* emp;

char key[KEY\_LENGTH];

FILE\* file;

char filename[100];

char line[100];

for (int i = 0; i < TABLE\_SIZE; i++) {

ht.table[i] = NULL;

}

printf("Enter the filename containing employee records: ");

scanf("%s", filename);

file = fopen(filename, "r");

if (file == NULL) {

printf("Error opening file.\n");

return 1;

}

while (fgets(line, sizeof(line), file)) {

emp = (struct Employee\*)malloc(sizeof(struct Employee));

sscanf(line, "%s %s %s %f", emp->key, emp->name, emp->designation, &emp->salary);

insert(&ht, emp);

}

fclose(file);

printf("Enter the key to search: ");

scanf("%s", key);

emp = search(&ht, key);

if (emp != NULL) {

printf("Employee record found with key %s:\n", emp->key);

printf("Name: %s\n", emp->name);

printf("Designation: %s\n", emp->designation);

printf("Salary: %.2f\n", emp->salary);

} else {

printf("Employee record not found for key %s\n", key);

}

for (int i = 0; i < TABLE\_SIZE; i++) {

if (ht.table[i] != NULL) {

free(ht.table[i]);

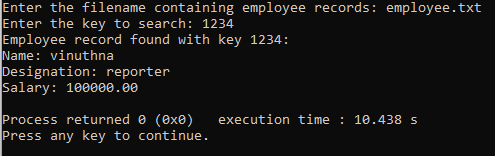
}

}

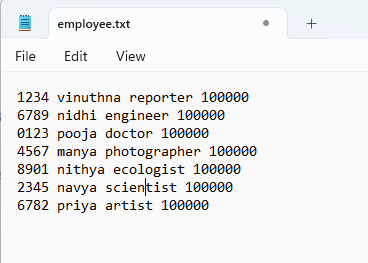
return 0;

}

**Output:**

****

**File:**

****