

VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.



LAB REPORT

On

DATA STRUCTURES

(23CS3PCDST)

Submitted by

DISHA D S (1BM23CS094)

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

September 2024-January 2025

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 **DISHA D S (1BM23CS094)**, who is bonafide student of **B. M. S. College of Engineering**. It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum during the year 2024-25. 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.

Dr. Selva Kumar S

Associate Professor

Department of CSE

BMSCE, Bengaluru

Dr. Kavitha Sooda

Professor and Head

Department of CSE

BMSCE, Bengaluru

INDEX SHEET

Sl. No.	Experiment Title	Page No.
1	Stimulate working of Stacks using array	4
2	Convert infix to postfix Leetcode: Remove all adjacent duplicates in a string	8 10
3	3a. Stimulate working of Queues 3b. Stimulate working of circular queues	12 15
4	Implementation of Singly Linked Lists Leetcode 1: Remove digit from number to maximize result Leetcode 2: Backspace string compare	20 25 27
5	Implementation of singly linked list to delete node at end, beginning and specified position and display Leetcode 1: Remove duplicates from sorted linked list Leetcode 2: Linked list cycle Leetcode 3: Palindrome linked list	30 37 40 45
6	6a. Program to sort, reverse and concatenate linked lists 6b. Program to stimulate stack and queue implementation using linked lists	50 57
7	Implement doubly linked lists	63
8	Binary Search Tree construction, traversal and display contents	69
9	9a. Program to traverse a graph using BFS method 9b. Program to check whether a given graph is connected using DFS method	76 80
10	Implementation of hash table to store employee records using remainder based hash function & resolving collisions with linear probing	84

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.

PROGRAM 1

Write a program to simulate the working of stack using an array with the following :

a) Push

b) Pop

c) Display

The program should print appropriate messages for stack overflow, stack underflow

Program:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 5
int stack[MAX];
int top = -1;
int isFull() {
    if (top == MAX - 1) {
        return 1;
    }
    return 0;
}
int isEmpty() {
    if (top == -1) {
        return 1;
    }
    return 0;
}
void push(int value) {
    if (isFull()) {
        printf("Stack Overflow! Cannot push %d. The stack is full.\n", value);
    } else {
        stack[++top] = value;
        printf("Pushed %d onto the stack.\n", value);
    }
}
```

```

void pop() {
    if (isEmpty()) {
        printf("Stack Underflow! Cannot pop. The stack is empty.\n");
    } else {
        printf("Popped %d from the stack.\n", stack[top--]);
    }
}

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

int main() {
    int choice, value;
    while (1) {
        printf("\nMenu:\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(value);
        break;
case 2:
    pop();
    break;
case 3:
    display();
    break;
case 4:
    printf("Exiting the program.\n");
    exit(0);
default:
    printf("Invalid choice! Please try again.\n");
    }
}

return 0;
}
```

OUTPUT:

```
DISHA D S, 1BM23CS094
Menu:
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 1
Enter the value to push: 1
Pushed 1 onto the stack.
DISHA D S, 1BM23CS094
Menu:
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 1
Enter the value to push: 2
Pushed 2 onto the stack.
DISHA D S, 1BM23CS094
Menu:
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 1
Enter the value to push: 3
Pushed 3 onto the stack.
DISHA D S, 1BM23CS094
Menu:
1. Push
```

```
2. Pop
3. Display
4. Exit
Enter your choice: 3
Current stack: 1 2 3
DISHA D S, 1BM23CS094
Menu:
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 2
Popped 3 from the stack.
DISHA D S, 1BM23CS094
Menu:
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 3
Current stack: 1 2
DISHA D S, 1BM23CS094
Menu:
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 4
Exiting the program.
```

PROGRAM 2:

Write a program 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)

PROGRAM:

```
#include<stdio.h>

#include<ctype.h>

#define size 50

char S[size];

int top = -1;

push(char elem){
    S[++top]=elem;
}

char pop(){
    return(S[top--]);
}

int pr(char sym){
    if(sym == '^'){
        return(3);}
    else if(sym == '*' || sym == '/'){
        return(2);
    }
    else if(sym == '+' || sym == '-'){
        return(1);
    }
    else{
        return(0);
    }
}

void main(){
```

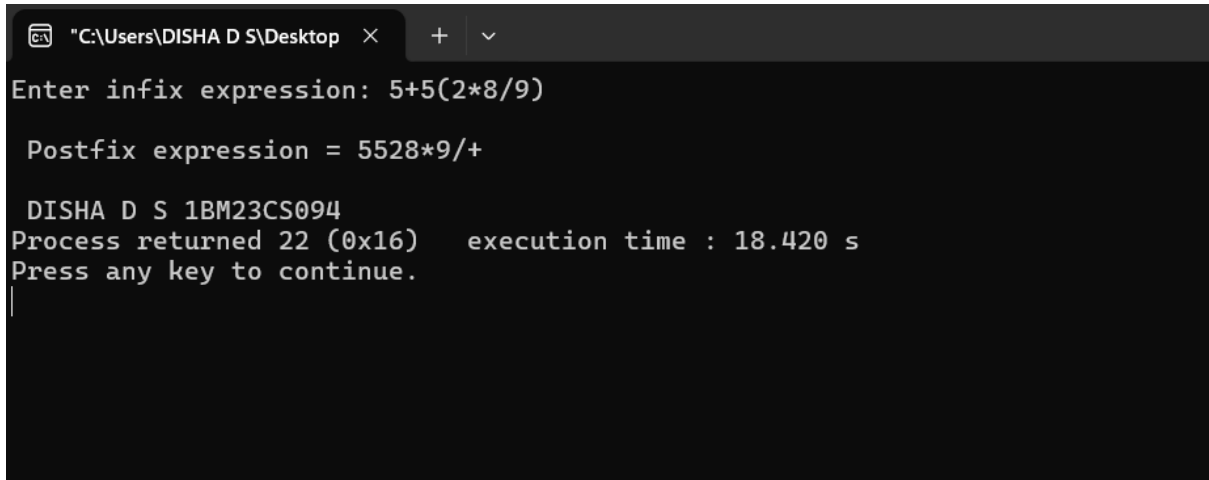


```

char inx[50], pox[50],ch,elem;
int i=0, k=0;
printf("Enter infix expression: ");
scanf("%s",inx);
push('#');
while( (ch=inx[i++]) != '\0'){
    if( ch == '(')
        push(ch);
    else
        if(isalnum(ch)) pox[k++]=ch;
        else
            if( ch ==')'){
                while(S[top] != '(')
                    pox[k++]=pop();
                elem= pop();
            }
            else{
                while(pr(S[top]) >= pr(ch))
                    pox[k++]=pop();
                push(ch);
            }
        }
    while ( S[top] != '#'){
        pox[k++] = pop();
    }
    pox[k]='\0';
    printf("\n Postfix expression = %s\n",pox);
    printf("\n DISHA D S, 1BM23CS094");
}

```

OUTPUT:



```
"C:\Users\DISHA D S\Desktop" × + v
Enter infix expression: 5+5(2*8/9)

Postfix expression = 5528*9/+

DISHA D S IBM23CS094
Process returned 22 (0x16) execution time : 18.420 s
Press any key to continue.
|
```

LEETCODE:

Remove all adjacent duplicates in a string:

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

```
#include <string.h>
```

```
void removeAdjacentDuplicates(char* str) {
```

```
    int len = strlen(str);
```

```
    int top = -1;
```

```
    char result[len + 1];
```

```
    for (int i = 0; i < len; i++) {
```

```
        if (top != -1 && result[top] == str[i]) {
```

```
            top--;
```

```
        } else {
```

```
            result[++top] = str[i];
```

```
        }
```

```
    }
```

```

    result[top + 1] = '\0';

    strcpy(str, result);
}

int main() {
    char str[100];
    printf("DISHA D S, 1BM23CS094");

    printf("Enter a string: ");
    fgets(str, sizeof(str), stdin);

    str[strcspn(str, "\n")] = '\0';

    removeAdjacentDuplicates(str);

    printf("String after removing adjacent duplicates: %s\n", str);

    return 0;
}

```

OUTPUT:

```

DISHA D S, 1BM23CS094
Enter a string: abbaca
String after removing adjacent duplicates: ca

```

```

DISHA D S, 1BM23CS094
Enter a string: azxxzy
String after removing adjacent duplicates: ay

```

PROGRAM 3:

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 50

int qarr[max];

int r = -1;

int f = -1;

display()
{
    int i;
    if(f == -1)
        printf("Queue is empty \n");
    else
    {
        printf("Queue is : \n");
        for( i=f; i<=r; i++){
            printf("%d",qarr[i]);
            printf("\n");}
    }
}

main()
{
    int ch;

    printf("DISHA D S, 1BM23CS094\n");

    while(1)
    {
        printf("1.INSERT\n");
```

```

printf("2.DELETE\n");
printf("3.DISPLAY\n");
printf("4.EXIT\n");
printf("Enter your choice: ");
scanf("%d",&ch);
switch(ch)
{
    case 1: insert(); break;
    case 2: deleting(); break;
    case 3: display(); break;
    case 4: exit(10);
    default: printf("INVALID CHOICE \n");
}
}
}
insert()
{
    int add;
    if(r == max-1)
        printf("QUEUE OVERFLOW \n");
    else
    {
        if( f == -1)
            f=0;

        printf("Insert the element in queue: ");
        scanf("%d",&add);
        r = r+1;
        qarr[r] = add;
    }
}

```

```

deleting()
{
    if(f == -1 || f>r)
    {
        printf("QUEUE UNDERFLOW \n");
        return;
    }
    else
    {
        printf("Deleted element is: %d\n", qarr[f]);
        f++;
    }
}

```

Output:

```

DISHA D S, 1BM23CS094
1.INSERT
2.DELETE
3.DISPLAY
4.EXIT
Enter your choice: 2
QUEUE UNDERFLOW
1.INSERT
2.DELETE
3.DISPLAY
4.EXIT
Enter your choice: 1
Insert the element in queue: 1
1.INSERT
2.DELETE
3.DISPLAY
4.EXIT
Enter your choice: 1
Insert the element in queue: 2
1.INSERT
2.DELETE
3.DISPLAY
4.EXIT
Enter your choice: 1
Insert the element in queue: 3

```

```
"C:\Users\DISHA D S\Desktop" × + v
Enter your choice: 1
Insert the element in queue: 3
1.INSERT
2.DELETE
3.DISPLAY
4.EXIT
Enter your choice: 3
Queue is :
1
2
3
1.INSERT
2.DELETE
3.DISPLAY
4.EXIT
Enter your choice: 2
Deleted element is: 1
1.INSERT
2.DELETE
3.DISPLAY
4.EXIT
Enter your choice: 3
Queue is :
2
3
1.INSERT
2.DELETE
3.DISPLAY
4.EXIT
Enter your choice: 4
Process returned 10 (0xA)   execution time : 51.557 s
```

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 size 5

int items[size];

int f = -1, r = -1;

int isFull(){
    if(( f == r+1) || (f==0 && r==size -1))
        return 1;
    return 0;
}
```

```

}

int isEmpty(){
    if(f== -1)
        return 1;
    return 0;
}

void enqueue(int ele){
    if (isFull())
        printf("\n QUEUE IS FULL !!\n");
    else{
        if(f==-1)
            f=0;
        r = (r+1) % size;
        items[r] = ele;
        printf("\n INSERTED -> %d",ele);
    }
}

int dequeue(){
    int ele;
    if(isEmpty()){
        printf("\nQueue is empty!!\n");
        return(-1);
    }
    else{
        ele = items[f];
        if( f==r){
            f = -1;
            r = -1;
        }
        else{

```



```

        f = (f+1)%size;
    }

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

void display(){
    int i;
    if(isEmpty())
        printf("\n Empty Queue\n");
    else{
        printf("\n Front = %d",f);
        printf("\n Items = ");
        for(i=f; i!= r; i=(i+1)%size){
            printf("%d",items[i]);
        }
        printf("%d",items[i]);
        printf("\nRear = %d \n",r);
    }

}

main()
{
    int ch,x;
    printf("DISHA D S, 1BM23CS094\n");
    while(1)
    {
        printf("1.INSERT\n");

```

```

printf("2.DELETE\n");
printf("3.DISPLAY\n");
printf("4.EXIT\n");
printf("Enter your choice: ");
scanf("%d",&ch);
switch(ch)
{
    case 1:
        printf("\nENTER THE ELEMENT TO BE INSERTED: ");
        scanf("%d",&x);
        enqueue(x); break;
    case 2: dequeue(); break;
    case 3: display(); break;
    case 4: exit(10);
    default: printf("INVALID CHOICE \n");
}
}
}

```

OUTPUT:

```

C:\Users\DISHA D S\Desktop
DISHA D S, 1BM23CS094
1.INSERT
2.DELETE
3.DISPLAY
4.EXIT
Enter your choice: 1
ENTER THE ELEMENT TO BE INSERTED: 1
    INSERTED -> 1
1.INSERT
2.DELETE
3.DISPLAY
4.EXIT
Enter your choice: 1
ENTER THE ELEMENT TO BE INSERTED: 2
    INSERTED -> 2
1.INSERT
2.DELETE
3.DISPLAY
4.EXIT
Enter your choice: 1
ENTER THE ELEMENT TO BE INSERTED: 3
    INSERTED -> 3

```

```
"C:\Users\DISHA D S\Desktop" X + v

INSERTED -> 3
1.INSERT
2.DELETE
3.DISPLAY
4.EXIT
Enter your choice: 3

Front = 0
Items = 123
Rear = 2

1.INSERT
2.DELETE
3.DISPLAY
4.EXIT
Enter your choice: 2

Deleted element -> 1

1.INSERT
2.DELETE
3.DISPLAY
4.EXIT
Enter your choice: 3

Front = 1
Items = 23
Rear = 2
```

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 end of list.

Display the contents of the linked list.

Program:

```
#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 insertAtFirst(struct Node** head, int value) {  
    struct Node* newNode = createNode(value);  
    newNode->next = *head;  
    *head = newNode;  
    printf("Node with value %d inserted at the beginning.\n", value);  
}
```

```

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;
    }
    printf("Node with value %d inserted at the end.\n", value);
}

```

```

void insertAtPosition(struct Node** head, int value, int position) {
    struct Node* newNode = createNode(value);

    if (position == 1) {
        newNode->next = *head;
        *head = newNode;
        printf("Node with value %d inserted at position %d.\n", value, position);
        return;
    }

    struct Node* temp = *head;
    int count = 1;

    while (temp != NULL && count < position - 1) {
        temp = temp->next;
    }

```

```

        count++;
    }

    if (temp == NULL) {
        printf("Position %d is out of range.\n", position);
        return;
    }

    newNode->next = temp->next;
    temp->next = newNode;
    printf("Node with value %d inserted at position %d.\n", value, position);
}

void displayList(struct Node* head) {
    if (head == NULL) {
        printf("The list is empty.\n");
        return;
    }

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

int main() {
    struct Node* head = NULL;
    int choice, value, position;
    printf("DISHA D S, 1BM23CS094");
}

```

```

while (1) {

    printf("\nMenu:\n");

    printf("1. Insert at the beginning\n");
    printf("2. Insert at the end\n");
    printf("3. Insert at a specific position\n");
    printf("4. Display the list\n");
    printf("5. Exit\n");


    printf("Enter your choice: ");
    scanf("%d", &choice);


    switch (choice) {
        case 1:
            printf("Enter the value to insert at the beginning: ");
            scanf("%d", &value);
            insertAtFirst(&head, value);
            break;
        case 2:
            printf("Enter the value to insert at the end: ");
            scanf("%d", &value);
            insertAtEnd(&head, value);
            break;
        case 3:
            printf("Enter the position and value to insert: ");
            scanf("%d %d", &position, &value);
            insertAtPosition(&head, value, position);
            break;
        case 4:
            displayList(head);
            break;
        case 5:
            printf("Exiting the program.\n");

```

```

        exit(0);
    default:
        printf("Invalid choice! Please try again.\n");
    }
}

return 0;
}

```

OUTPUT:

```

DISHA D S, 1BM23CS094
Menu:
1. Insert at the beginning
2. Insert at the end
3. Insert at a specific position
4. Display the list
5. Exit
Enter your choice: 1
Enter the value to insert at the beginning: 1
Node with value 1 inserted at the beginning.

Menu:
1. Insert at the beginning
2. Insert at the end
3. Insert at a specific position
4. Display the list
5. Exit
Enter your choice: 1
Enter the value to insert at the beginning: 2
Node with value 2 inserted at the beginning.

Menu:
1. Insert at the beginning
2. Insert at the end
3. Insert at a specific position
4. Display the list
5. Exit
Enter your choice: 1
Enter the value to insert at the beginning: 3
Node with value 3 inserted at the beginning.

```



```

Menu:
1. Insert at the beginning
2. Insert at the end
3. Insert at a specific position
4. Display the list
5. Exit
Enter your choice: 2
Enter the value to insert at the end: 4
Node with value 4 inserted at the end.

Menu:
1. Insert at the beginning
2. Insert at the end
3. Insert at a specific position
4. Display the list
5. Exit
Enter your choice: 3
Enter the position and value to insert: 5 23
Node with value 23 inserted at position 5.

Menu:
1. Insert at the beginning
2. Insert at the end
3. Insert at a specific position
4. Display the list
5. Exit
Enter your choice: 4
Linked List: 3 2 1 4 23

```

LEETCODE 1

Remove digit from number to maximize result

```

#include <stdio.h>
#include <string.h>

void removeDigitToMaximize(char* number, char digit) {
    int len = strlen(number);
    int i;
    for (i = 0; i < len - 1; i++) {
        if (number[i] == digit && number[i] < number[i + 1]) {
            for (int j = i; j < len - 1; j++) {
                number[j] = number[j + 1];
            }
            number[len - 1] = '\0';
        }
    }
    return;
}

```

```

    }
}

for (i = len - 1; i >= 0; i--) {
    if (number[i] == digit) {

        for (int j = i; j < len - 1; j++) {
            number[j] = number[j + 1];
        }
        number[len - 1] = '\0';
        return;
    }
}

int main() {
    char number[100];
    char digit;

    printf("Enter the number: ");
    fgets(number, sizeof(number), stdin);

    number[strcspn(number, "\n")] = '\0';

    printf("Enter the digit to remove: ");
    scanf(" %c", &digit);
    removeDigitToMaximize(number, digit);

    printf("Maximized number: %s\n", number);
}

```

```
return 0;}
```

output:

```
DISHA D S, 1BM23CS094
Enter the number: 123
Enter the digit to remove: 3
Maximized number: 12
```

```
DISHA D S, 1BM23CS094
Enter the number: 1231
Enter the digit to remove: 1
Maximized number: 231
```

```
DISHA D S, 1BM23CS094
Enter the number: 551
Enter the digit to remove: 5
Maximized number: 51
```

Leetcode 2:

Backspace String Compare

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

```
#include <string.h>
```

```
void processString(char* input, char* result) {
```

```
    int index = 0;
```

```
    int length = strlen(input);
```

```
    for (int i = 0; i < length; i++) {
```

```
        if (input[i] == '#') {
```

```
            if (index > 0) {
```

```
                index--;
```

```
            }
```

```

        } else {

            result[index++] = input[i];
        }
    }
    result[index] = '\0';
}

int backspaceCompare(char* s, char* t) {
    char result_s[1000];
    char result_t[1000];

    processString(s, result_s);
    processString(t, result_t);

    return strcmp(result_s, result_t) == 0;
}

int main() {
    char s[1000], t[1000];

    printf("Enter string s: ");
    fgets(s, sizeof(s), stdin);
    s[strcspn(s, "\n")] = '\0';

    printf("Enter string t: ");
    fgets(t, sizeof(t), stdin);
    t[strcspn(t, "\n")] = '\0';

```

```
if (backspaceCompare(s, t)) {  
    printf("true\n");  
} else {  
    printf("false\n");  
}  
  
return 0;  
}
```

Output:

```
Enter string s: ab#c  
Enter string t: ad#c  
true
```

```
Enter string s: ab##  
Enter string t: c#d#  
true
```

```
DISHA D S, 1BM23CS094  
Enter string s: a#c  
Enter string t: b  
false
```

LAB 5:

Write a Program 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* createLinkedList();  
void deleteFirst(struct Node** head);  
void deleteSpecified(struct Node** head, int key);  
void deleteLast(struct Node** head);  
void displayLinkedList(struct Node* head);
```

```
int main() {  
    struct Node* head = NULL;  
    int choice, key;  
  
    while (1) {  
        printf("\nMenu:\n");  
        printf("1. Create Linked List\n");  
        printf("2. Delete First Element\n");  
        printf("3. Delete Specified Element\n");  
        printf("4. Delete Last Element\n");  
        printf("5. Display Linked List\n");
```

```

printf("6. Exit\n");
printf("Enter your choice: ");
scanf("%d", &choice);

switch (choice) {
    case 1:
        head = createLinkedList();
        break;
    case 2:
        deleteFirst(&head);
        break;
    case 3:
        printf("Enter the value to delete: ");
        scanf("%d", &key);
        deleteSpecified(&head, key);
        break;
    case 4:
        deleteLast(&head);
        break;
    case 5:
        displayLinkedList(head);
        break;
    case 6:
        printf("Exiting program.\n");
        exit(0);
    default:
        printf("Invalid choice. Try again.\n");
}
}
return 0;
}

```

```

struct Node* createLinkedList() {
    struct Node* head = NULL;
    struct Node* temp = NULL;
    struct Node* newNode = NULL;
    int data;

    printf("Enter -1 to stop creating the list.\n");
    while (1) {
        printf("Enter data: ");
        scanf("%d", &data);
        if (data == -1)
            break;

        newNode = (struct Node*)malloc(sizeof(struct Node));
        newNode->data = data;
        newNode->next = NULL;

        if (head == NULL) {
            head = newNode;
        } else {
            temp->next = newNode;
        }
        temp = newNode;
    }
    return head;
}

```

```

void deleteFirst(struct Node** head) {
    if (*head == NULL) {
        printf("List is empty. Nothing to delete.\n");
        return;
    }
}

```



```

    struct Node* temp = *head;
    *head = (*head)->next;
    free(temp);
    printf("First element deleted.\n");
}

void deleteSpecified(struct Node** head, int key) {
    if (*head == NULL) {
        printf("List is empty. Nothing to delete.\n");
        return;
    }

    struct Node* temp = *head;
    struct Node* prev = NULL;

    if (temp != NULL && temp->data == key) {
        *head = temp->next;
        free(temp);
        printf("Element %d deleted.\n", key);
        return;
    }

    while (temp != NULL && temp->data != key) {
        prev = temp;
        temp = temp->next;
    }

    if (temp == NULL) {
        printf("Element %d not found.\n", key);
    }
}

```

```

        return;
    }

    prev->next = temp->next;
    free(temp);
    printf("Element %d deleted.\n", key);
}

void deleteLast(struct Node** head) {
    if (*head == NULL) {
        printf("List is empty. Nothing to delete.\n");
        return;
    }

    struct Node* temp = *head;

    if (temp->next == NULL) {
        *head = NULL;
        free(temp);
        printf("Last element deleted.\n");
        return;
    }

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

```

```

prev->next = NULL;

free(temp);

printf("Last element deleted.\n");
}

void displayLinkedList(struct Node* head) {
    if (head == NULL) {
        printf("Linked list is empty.\n");
        return;
    }

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

```

OUTPUT:

```

PS C:\Users\DISHA D S\OneDrive\Desktop\cpro> cd "c:\Users\DISHA D S\OneDrive\Desktop\cpro\" ;
Menu:
1. Create Linked List
2. Delete First Element
3. Delete Specified Element
4. Delete Last Element
5. Display Linked List
6. Exit
Enter your choice: 1
Enter -1 to stop creating the list.
Enter data: 1
Enter data: 2
Enter data: 3
Enter data: 4
Enter data: 5
Enter data: -1

```

```

Enter data: -1

Menu:
1. Create Linked List
2. Delete First Element
3. Delete Specified Element
4. Delete Last Element
5. Display Linked List
6. Exit
Enter your choice: 2
First element deleted.

Menu:
1. Create Linked List
2. Delete First Element
3. Delete Specified Element
4. Delete Last Element
5. Display Linked List
6. Exit
Enter your choice: 4
Last element deleted.

Menu:
1. Create Linked List
2. Delete First Element
3. Delete Specified Element
4. Delete Last Element
5. Display Linked List
6. Exit
Enter your choice: 3
Enter the value to delete: 4
Element 4 deleted.

Menu:
1. Create Linked List
2. Delete First Element
3. Delete Specified Element
4. Delete Last Element
5. Display Linked List
6. Exit
Enter your choice: 5

```

```

Menu:
1. Create Linked List
2. Delete First Element
3. Delete Specified Element
4. Delete Last Element
5. Display Linked List
6. Exit
Enter your choice: 5
linked list contents: 2 -> 3 -> NULL

Menu:
1. Create Linked List
2. Delete First Element
3. Delete Specified Element
4. Delete Last Element
5. Display Linked List
6. Exit
Enter your choice: 6
Exiting program.

```

Leetcode:

1.Remove Duplicates from a sorted linked list

```
#include <stdio.h>

#include <stdlib.h>

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

struct Node* createSortedLinkedList();
void removeDuplicates(struct Node* head);
void displayLinkedList(struct Node* head);

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

    printf("Create a sorted linked list:\n");
    head = createSortedLinkedList();

    printf("\nOriginal Linked List: ");
    displayLinkedList(head);

    removeDuplicates(head);

    printf("\nLinked List after removing duplicates: ");
    displayLinkedList(head);

    return 0;
}
```

```

struct Node* createSortedLinkedList() {
    struct Node* head = NULL;
    struct Node* temp = NULL;
    struct Node* newNode = NULL;
    int data;

    printf("Enter sorted elements (-1 to stop):\n");
    while (1) {
        printf("Enter data: ");
        scanf("%d", &data);
        if (data == -1)
            break;

        newNode = (struct Node*)malloc(sizeof(struct Node));
        newNode->data = data;
        newNode->next = NULL;

        if (head == NULL) {
            head = newNode;
        } else {
            temp->next = newNode;
        }
        temp = newNode;
    }
    return head;
}

```

```

void removeDuplicates(struct Node* head) {
    struct Node* current = head;
    struct Node* nextNode;

    if (current == NULL) {

```

```

        return;
    }

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

void displayLinkedList(struct Node* head) {
    if (head == NULL) {
        printf("Linked list is empty.\n");
        return;
    }

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

```

OUTPUT

```
PS C:\Users\DISHA D S\OneDrive\Desktop\cpro> cd "c:\Users\DISHA D S\OneDrive\De
Create a sorted linked list:
Enter sorted elements (-1 to stop):
Enter data: 1
Enter data: 1
Enter data: 2
Enter data: -1

Original Linked List: 1 -> 1 -> 2 -> NULL

Create a sorted linked list:
Enter sorted elements (-1 to stop):
Enter data: 1
Enter data: 1
Enter data: 2
Enter data: 3
Enter data: 3
Enter data: -1

Original Linked List: 1 -> 1 -> 2 -> 3 -> 3 -> NULL

Linked List after removing duplicates: 1 -> 2 -> 3 -> NULL
```

Leetcode-2

Linked list Cycle

Given head, the head of a linked list, determine if the linked list has a cycle in it.

There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the next pointer. Internally, pos is used to denote the index of the node that tail's next pointer is connected to. **Note that pos is not passed as a parameter.**

Return true *if there is a cycle in the linked list*. Otherwise, return false.

```
#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

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



```

struct Node* createLinkedList();

bool hasCycle(struct Node* head);

void createCycle(struct Node* head, int pos);

void freeLinkedList(struct Node* head);


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

    int pos;

    printf("Create a linked list:\n");

    head = createLinkedList();

    printf("Enter the position (0-based) to create a cycle (-1 for no cycle): ");

    scanf("%d", &pos);

    if (pos != -1) {
        createCycle(head, pos);
    }

    if (hasCycle(head)) {
        printf("The linked list contains a cycle.\n");
    } else {
        printf("The linked list does not contain a cycle.\n");
    }

    if (!hasCycle(head)) {
        freeLinkedList(head);
    }

    return 0;
}


struct Node* createLinkedList() {
    struct Node* head = NULL;

    struct Node* temp = NULL;

    struct Node* newNode = NULL;

    int data;

```

```

printf("Enter elements (-1 to stop):\n");
while (1) {
    printf("Enter data: ");
    scanf("%d", &data);
    if (data == -1)
        break;

    newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = NULL;

    if (head == NULL) {
        head = newNode;
    } else {
        temp->next = newNode;
    }
    temp = newNode;
}
return head;
}

```

```

bool hasCycle(struct Node* head) {
    struct Node* slow = head;
    struct Node* fast = head;

    while (fast != NULL && fast->next != NULL) {
        slow = slow->next;
        fast = fast->next->next;

        if (slow == fast) {
            return true;
        }
    }
}

```

```

    }
    return false;
}

void createCycle(struct Node* head, int pos) {
    if (head == NULL || pos < 0) return;

    struct Node* temp = head;
    struct Node* cycleNode = NULL;
    int index = 0;

    while (temp->next != NULL) {
        if (index == pos) {
            cycleNode = temp;
        }
        temp = temp->next;
        index++;
    }

    if (cycleNode != NULL) {
        temp->next = cycleNode;
    }
}

void freeLinkedList(struct Node* head) {
    struct Node* temp;
    while (head != NULL) {
        temp = head;
        head = head->next;
        free(temp);
    }
}

```

OUTPUT:

```
PS C:\Users\DISHA D S\OneDrive\Desktop\cpro> cd "c:\Users\DISHA D S\OneDr
Create a linked list:
Enter elements (-1 to stop):
Enter data: 3
Enter data: 2
Enter data: 0
Enter data: 4
Enter data: -1
Enter the position (0-based) to create a cycle (-1 for no cycle): 1
The linked list contains a cycle.
```

```
Create a linked list:
Enter elements (-1 to stop):
Enter data: 1
Enter data: 2
Enter data: -1
Enter the position (0-based) to create a cycle (-1 for no cycle): 0
The linked list contains a cycle.
```

```
PS C:\Users\DISHA D S\OneDrive\Desktop\cpro> cd "c:\Users\DISHA D S\OneDr
Create a linked list:
Enter elements (-1 to stop):
Enter data: 1
Enter data: -1
Enter the position (0-based) to create a cycle (-1 for no cycle): 1
The linked list does not contain a cycle.
```

Leetcode 3

Palindrome linked list:

Given the head of a singly linked list, return true if it is a palindrome or false otherwise.

```
#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

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

struct Node* createLinkedList();
bool isPalindrome(struct Node* head);
struct Node* reverseList(struct Node* head);
void displayLinkedList(struct Node* head);
void freeLinkedList(struct Node* head);

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

    printf("Create a linked list:\n");
    head = createLinkedList();

    printf("\nOriginal Linked List: ");
    displayLinkedList(head);

    if (isPalindrome(head)) {
        printf("\nThe linked list is a palindrome.\n");
    } else {
        printf("\nThe linked list is not a palindrome.\n");
    }
}
```

```

    freeLinkedList(head);
    return 0;
}

struct Node* createLinkedList() {
    struct Node* head = NULL;
    struct Node* temp = NULL;
    struct Node* newNode = NULL;
    int data;

    printf("Enter elements (-1 to stop):\n");
    while (1) {
        printf("Enter data: ");
        scanf("%d", &data);
        if (data == -1)
            break;

        newNode = (struct Node*)malloc(sizeof(struct Node));
        newNode->data = data;
        newNode->next = NULL;

        if (head == NULL) {
            head = newNode;
        } else {
            temp->next = newNode;
        }
        temp = newNode;
    }
    return head;
}

bool isPalindrome(struct Node* head) {

```

```

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

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

/

struct Node* secondHalf = reverseList(slow);

struct Node* firstHalf = head;
struct Node* temp = secondHalf;
while (temp != NULL) {
    if (firstHalf->data != temp->data) {
        reverseList(secondHalf);
        return false;
    }
    firstHalf = firstHalf->next;
    temp = temp->next;
}

reverseList(secondHalf);

return true;
}

```

```

struct Node* reverseList(struct Node* head) {
    struct Node* prev = NULL;
    struct Node* current = head;
    struct Node* next = NULL;

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

    return prev;
}

```

```

void displayLinkedList(struct Node* head) {
    if (head == NULL) {
        printf("Linked list is empty.\n");
        return;
    }

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

```

```

void freeLinkedList(struct Node* head) {
    struct Node* temp;

```



```
while (head != NULL) {  
    temp = head;  
    head = head->next;  
    free(temp);  
}  
}
```

OUTPUT:

```
PS C:\Users\DISHA D S\OneDrive\Desktop\cpro> cd "c:\Users\DISHA D S\  
Create a linked list:  
Enter elements (-1 to stop):  
Enter data: 1  
Enter data: 2  
Enter data: 2  
Enter data: 1  
Enter data: -1  
  
The linked list is a palindrome.  
PS C:\Users\DISHA D S\OneDrive\Desktop\cpro> cd "c:\Users\DISHA D S\  
Create a linked list:  
Enter elements (-1 to stop):  
Enter data: 1  
Enter data: 2  
Enter data: -1  
  
Original Linked List: 1 -> 2 -> NULL  
  
The linked list is not a palindrome.
```

LAB – 6:

6 A) 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>
```

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

```
struct Node* createLinkedList();
```

```
void sortLinkedList(struct Node** head);
```

```
void reverseLinkedList(struct Node** head);
```

```
struct Node* concatenateLists(struct Node* head1, struct Node* head2);
```

```
void displayLinkedList(struct Node* head);
```

```
int main() {
```

```
    struct Node* head1 = NULL;
```

```
    struct Node* head2 = NULL;
```

```
    struct Node* concatenated = NULL;
```

```
    int choice;
```

```
    printf("Create First Linked List:\n");
```

```
    head1 = createLinkedList();
```

```
    printf("\nCreate Second Linked List:\n");
```

```
    head2 = createLinkedList();
```

```
    printf("\nFirst Linked List: ");
```

```
    displayLinkedList(head1);
```

```
    printf("\nSecond Linked List: ");
```

```

displayLinkedList(head2);

while (1) {
    printf("\nMenu:\n");
    printf("1. Reverse First Linked List\n");
    printf("2. Reverse Second Linked List\n");
    printf("3. Sort First Linked List\n");
    printf("4. Sort Second Linked List\n");
    printf("5. Concatenate Two Linked Lists\n");
    printf("6. Exit\n");
    printf("Enter your choice: ");
    scanf("%d", &choice);

    switch (choice) {
        case 1:
            reverseLinkedList(&head1);
            printf("\nFirst Linked List after reversing: ");
            displayLinkedList(head1);
            break;

        case 2:
            reverseLinkedList(&head2);
            printf("\nSecond Linked List after reversing: ");
            displayLinkedList(head2);
            break;

        case 3:
            sortLinkedList(&head1);
            printf("\nFirst Linked List after sorting: ");
            displayLinkedList(head1);
            break;
    }
}

```

```

case 4:

    sortLinkedList(&head2);

    printf("\nSecond Linked List after sorting: ");

    displayLinkedList(head2);

    break;

case 5:

    concatenated = concatenateLists(head1, head2);

    printf("\nConcatenated Linked List: ");

    displayLinkedList(concatenated);

    break;

case 6:

    printf("Exiting program.\n");

    exit(0);

default:

    printf("Invalid choice. Try again.\n");

}

}

return 0;

}

```

```

struct Node* createLinkedList() {

    struct Node* head = NULL;

    struct Node* temp = NULL;

    struct Node* newNode = NULL;

    int data;

    printf("Enter elements (-1 to stop):\n");

    while (1) {

        printf("Enter data: ");

```

```

scanf("%d", &data);
if (data == -1)
    break;

newNode = (struct Node*)malloc(sizeof(struct Node));
newNode->data = data;
newNode->next = NULL;

if (head == NULL) {
    head = newNode;
} else {
    temp->next = newNode;
}
temp = newNode;
}
return head;
}

```

```

void sortLinkedList(struct Node** head) {
    if (*head == NULL) return;

    struct Node* current = *head;
    struct Node* index = NULL;
    int temp;

    while (current != NULL) {
        index = current->next;
        while (index != NULL) {
            if (current->data > index->data) {
                temp = current->data;
                current->data = index->data;
                index->data = temp;
            }
            index = index->next;
        }
        current = current->next;
    }
}

```

```

    }
    index = index->next;
}
current = current->next;
}
}

```

```

void 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;
}

```

```

struct Node* concatenateLists(struct Node* head1, struct Node* head2) {

```

```

    if (head1 == NULL) return head2;
    if (head2 == NULL) return head1;

```

```

    struct Node* temp = head1;
    while (temp->next != NULL) {
        temp = temp->next;
    }

```

```

    temp->next = head2;

```

```

    return head1;
}

void displayLinkedList(struct Node* head) {
    if (head == NULL) {
        printf("Linked list is empty.\n");
        return;
    }

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

```

OUTPUT:

```

Create First Linked List:
Enter elements (-1 to stop):
Enter data: 1
Enter data: 23
Enter data: 43
Enter data: 54
Enter data: 12
Enter data: -1

Create Second Linked List:
Enter elements (-1 to stop):
Enter data: 21
Enter data: 3
Enter data: 43
Enter data: 65
Enter data: 23
Enter data: -1

First Linked List: 1 -> 23 -> 43 -> 54 -> 12 -> NULL
Second Linked List: 21 -> 3 -> 43 -> 65 -> 23 -> NULL

```

Menu:

1. Reverse First Linked List
2. Reverse Second Linked List
3. Sort First Linked List
4. Sort Second Linked List
5. Concatenate Two Linked Lists
6. Exit

Enter your choice: 1

First Linked List after reversing: 12 -> 54 -> 43 -> 23 -> 1 -> NULL

Menu:

1. Reverse First Linked List
2. Reverse Second Linked List
3. Sort First Linked List
4. Sort Second Linked List
5. Concatenate Two Linked Lists
6. Exit

Enter your choice: 2

Second Linked List after reversing: 23 -> 65 -> 43 -> 3 -> 21 -> NULL

Menu:

1. Reverse First Linked List
2. Reverse Second Linked List
3. Sort First Linked List
4. Sort Second Linked List
5. Concatenate Two Linked Lists
6. Exit

Enter your choice: 3

First Linked List after sorting: 1 -> 12 -> 23 -> 43 -> 54 -> NULL

Menu:

1. Reverse First Linked List
2. Reverse Second Linked List
3. Sort First Linked List
4. Sort Second Linked List
5. Concatenate Two Linked Lists
6. Exit

Enter your choice: 4

Second Linked List after sorting: 3 -> 21 -> 23 -> 43 -> 65 -> NULL

Menu:

1. Reverse First Linked List
2. Reverse Second Linked List
3. Sort First Linked List
4. Sort Second Linked List
5. Concatenate Two Linked Lists
6. Exit

Enter your choice: 5

Concatenated Linked List: 1 -> 12 -> 23 -> 43 -> 54 -> 3 -> 21 -> 23 -> 43 -> 65 -> NULL

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

```
#include <stdio.h>

#include <stdlib.h>

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

void push(struct Node** top, int value);
int pop(struct Node** top);
void enqueue(struct Node** front, struct Node** rear, int value);
int dequeue(struct Node** front, struct Node** rear);
void display(struct Node* head);

int main() {
    struct Node* stack = NULL;
    struct Node* queueRear = NULL;
    int choice, value;

    while (1) {
        printf("\nMenu:\n");
        printf("1. Push (Stack)\n");
        printf("2. Pop (Stack)\n");
        printf("3. Enqueue (Queue)\n");
        printf("4. Dequeue (Queue)\n");
        printf("5. Display Stack\n");
        printf("6. Display Queue\n");
        printf("7. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
```

```

switch (choice) {
    case 1:
        printf("Enter value to push onto stack: ");
        scanf("%d", &value);
        push(&stack, value);
        break;
    case 2:
        value = pop(&stack);
        if (value != -1) {
            printf("Popped value from stack: %d\n", value);
        }
        break;
    case 3:
        printf("Enter value to enqueue into queue: ");
        scanf("%d", &value);
        enqueue(&queueFront, &queueRear, value);
        break;
    case 4:
        value = dequeue(&queueFront, &queueRear);
        if (value != -1) {
            printf("Dequeued value from queue: %d\n", value);
        }
        break;
    case 5:
        printf("Stack: ");
        display(stack);
        break;
    case 6:
        printf("Queue: ");
        display(queueFront);
        break;
    case 7:

```

```

        printf("Exiting program.\n");
        exit(0);
    default:
        printf("Invalid choice. Please try again.\n");
    }
}

return 0;
}

void push(struct Node** top, int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = *top;
    *top = newNode;
    printf("Pushed %d onto the stack.\n", value);
}

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

void enqueue(struct Node** front, struct Node** rear, int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));

```

```

newNode->data = value;
newNode->next = NULL;
if (*rear == NULL) {
    *front = *rear = newNode;
} else {
    (*rear)->next = newNode;
    *rear = newNode;
}
printf("Enqueued %d into the queue.\n", value);
}

int dequeue(struct Node** front, struct Node** rear) {
    if (*front == NULL) {
        printf("Queue is empty.\n");
        return -1;
    }
    int dequeuedValue = (*front)->data;
    struct Node* temp = *front;
    *front = (*front)->next;
    if (*front == NULL) {
        *rear = NULL;
    }
    free(temp);
    return dequeuedValue;
}

void display(struct Node* head) {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }
    struct Node* temp = head;

```

```
while (temp != NULL) {  
    printf("%d -> ", temp->data);  
    temp = temp->next;  
}  
printf("NULL\n");  
}
```

OUTPUT:

```
Menu:  
1. Push (Stack)  
2. Pop (Stack)  
3. Enqueue (Queue)  
4. Dequeue (Queue)  
5. Display Stack  
6. Display Queue  
7. Exit  
Enter your choice: 1  
Enter value to push onto stack: 1  
Pushed 1 onto the stack.  
  
Menu:  
1. Push (Stack)  
2. Pop (Stack)  
3. Enqueue (Queue)  
4. Dequeue (Queue)  
5. Display Stack  
6. Display Queue  
7. Exit  
Enter your choice: 1  
Enter value to push onto stack: 2  
Pushed 2 onto the stack.  
  
Menu:  
1. Push (Stack)  
2. Pop (Stack)  
3. Enqueue (Queue)  
4. Dequeue (Queue)  
5. Display Stack  
6. Display Queue  
7. Exit  
Enter your choice: 1  
Enter value to push onto stack: 3  
Pushed 3 onto the stack.
```

```
Menu:
1. Push (Stack)
2. Pop (Stack)
3. Enqueue (Queue)
4. Dequeue (Queue)
5. Display Stack
6. Display Queue
7. Exit
Enter your choice: 2
Popped value from stack: 3
```

```
Menu:
1. Push (Stack)
2. Pop (Stack)
3. Enqueue (Queue)
4. Dequeue (Queue)
5. Display Stack
6. Display Queue
7. Exit
Enter your choice: 5
Stack: 2 -> 1 -> NULL
```

```
Menu:
1. Push (Stack)
2. Pop (Stack)
3. Enqueue (Queue)
4. Dequeue (Queue)
5. Display Stack
6. Display Queue
7. Exit
Enter your choice: 3
Enter value to enqueue into queue: 1
Enqueued 1 into the queue.
```

```
Menu:
1. Push (Stack)
2. Pop (Stack)
3. Enqueue (Queue)
4. Dequeue (Queue)
5. Display Stack
6. Display Queue
7. Exit
Enter your choice: 3
Enter value to enqueue into queue: 2
Enqueued 2 into the queue.
```

```
Menu:
1. Push (Stack)
2. Pop (Stack)
3. Enqueue (Queue)
4. Dequeue (Queue)
5. Display Stack
6. Display Queue
7. Exit
Enter your choice: 3
Enter value to enqueue into queue: 4
Enqueued 4 into the queue.
```

```
Menu:
1. Push (Stack)
2. Pop (Stack)
3. Enqueue (Queue)
4. Dequeue (Queue)
5. Display Stack
6. Display Queue
7. Exit
Enter your choice: 4
Dequeued value from queue: 1
```

```
Menu:
1. Push (Stack)
2. Pop (Stack)
3. Enqueue (Queue)
4. Dequeue (Queue)
5. Display Stack
6. Display Queue
7. Exit
Enter your choice: 6
Queue: 2 -> 4 -> 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**
- d) Display the contents of the list**

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

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

```
struct Node {
```

```
    int data;
```

```
    struct Node* next;
```

```
    struct Node* prev;
```

```
};
```

```
struct Node* createDoublyLinkedList();
```

```
void insertLeft(struct Node* head, int key, int newValue);
```

```
void deleteNode(struct Node** head, int value);
```

```
void displayList(struct Node* head);
```

```
int main() {
```

```
    struct Node* head = NULL;
```

```
    int choice, value, key;
```

```
    head = createDoublyLinkedList();
```

```
    while (1) {
```

```
        printf("\nMenu:\n");
```

```
        printf("1. Insert node to the left of a specific node\n");
```

```
        printf("2. Delete a node based on a specific value\n");
```

```

printf("3. Display the contents of the list\n");
printf("4. Exit\n");
printf("Enter your choice: ");
scanf("%d", &choice);

switch (choice) {
    case 1:
        printf("Enter the value of the node to insert left of: ");
        scanf("%d", &key);
        printf("Enter the new value to insert: ");
        scanf("%d", &value);
        insertLeft(head, key, value);
        break;
    case 2:
        printf("Enter the value of the node to delete: ");
        scanf("%d", &value);
        deleteNode(&head, value);
        break;
    case 3:
        displayList(head);
        break;
    case 4:
        printf("Exiting program.\n");
        exit(0);
    default:
        printf("Invalid choice! Please try again.\n");
}
}

return 0;
}

```



```

struct Node* createDoublyLinkedList() {
    struct Node* head = NULL;
    struct Node* temp = NULL;
    struct Node* newNode = NULL;
    int data;

    printf("Enter elements for the doubly linked list (-1 to stop):\n");
    while (1) {
        printf("Enter data: ");
        scanf("%d", &data);
        if (data == -1)
            break;

        newNode = (struct Node*)malloc(sizeof(struct Node));
        newNode->data = data;
        newNode->next = NULL;
        newNode->prev = NULL;

        if (head == NULL) {
            head = newNode;
        } else {
            temp = head;
            while (temp->next != NULL) {
                temp = temp->next;
            }
            temp->next = newNode;
            newNode->prev = temp;
        }
    }

    return head;
}

```

```

void insertLeft(struct Node* head, int key, int newValue) {
    struct Node* temp = head;

    while (temp != NULL && temp->data != key) {
        temp = temp->next;
    }

    if (temp != NULL) {

        struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
        newNode->data = newValue;
        newNode->prev = temp->prev;
        newNode->next = temp;

        if (temp->prev != NULL) {
            temp->prev->next = newNode;
        } else {
            head = newNode;
        }
        temp->prev = newNode;

        printf("Inserted %d to the left of %d.\n", newValue, key);
    } else {
        printf("Node with value %d not found!\n", key);
    }
}

void deleteNode(struct Node** head, int value) {
    struct Node* temp = *head;

```

```

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

if (temp == NULL) {
    printf("Node with value %d not found!\n", value);
    return;
}

// If the node is the first node
if (temp == *head) {
    *head = temp->next;
}

// If the node is not the last node
if (temp->next != NULL) {
    temp->next->prev = temp->prev;
}

// If the node is not the first node
if (temp->prev != NULL) {
    temp->prev->next = temp->next;
}

free(temp);
printf("Node with value %d deleted.\n", value);
}

void displayList(struct Node* head) {
    if (head == NULL) {

```

```

        printf("The list is empty.\n");
        return;
    }

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

```

OUTPUT:

```

PS C:\Users\DISHA D S\OneDrive\Desktop\cpro> cd "c:\Users\DISHA D S\OneDrive\De
Enter elements for the doubly linked list (-1 to stop):
Enter data: 1
Enter data: 3
Enter data: 4
Enter data: -1

Menu:
1. Insert node to the left of a specific node
2. Delete a node based on a specific value
3. Display the contents of the list
4. Exit
Enter your choice: 1
Enter the value of the node to insert left of: 2
Enter the new value to insert: 32
Inserted 32 to the left of 2.

Menu:
1. Insert node to the left of a specific node
2. Delete a node based on a specific value
3. Display the contents of the list
4. Exit
Enter your choice: 2
Enter the value of the node to delete: 4
Node with value 4 deleted.

Menu:
1. Insert node to the left of a specific node
2. Delete a node based on a specific value
3. Display the contents of the list
4. Exit
Enter your choice: 3
Doubly Linked List: 1 <-> 32 <-> 2 <-> 3 <-> NULL

Menu:
1. Insert node to the left of a specific node
2. Delete a node based on a specific value
3. Display the contents of the list
4. Exit
Enter your choice: 4
Exiting program.

```

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>
```

```
// Define the structure for a node in the binary search tree
```

```
struct Node {
```

```
    int data;
```

```
    struct Node* left;
```

```
    struct Node* right;
```

```
};
```

```
// Function prototypes
```

```
struct Node* createNode(int value);
```

```
struct Node* insert(struct Node* root, int value);
```

```
void inorderTraversal(struct Node* root);
```

```
void preorderTraversal(struct Node* root);
```

```
void postorderTraversal(struct Node* root);
```

```
void display(struct Node* root);
```

```
int main() {
```

```
    struct Node* root = NULL;
```

```
    int choice, value;
```

```
    while (1) {
```

```
        printf("\nMenu:\n");
```

```
        printf("1. Insert value into Binary Search Tree\n");
```

```
        printf("2. In-order Traversal\n");
```

```
        printf("3. Pre-order Traversal\n");
```

```

printf("4. Post-order Traversal\n");
printf("5. Display Tree\n");
printf("6. Exit\n");
printf("Enter your choice: ");
scanf("%d", &choice);

switch (choice) {
    case 1:
        printf("Enter value to insert into BST: ");
        scanf("%d", &value);
        root = insert(root, value);
        break;
    case 2:
        printf("In-order Traversal: ");
        inorderTraversal(root);
        break;
    case 3:
        printf("Pre-order Traversal: ");
        preorderTraversal(root);
        break;
    case 4:
        printf("Post-order Traversal: ");
        postorderTraversal(root);
        break;
    case 5:
        printf("Tree elements: ");
        display(root);
        break;
    case 6:
        printf("Exiting program.\n");
        exit(0);
    default:

```

```

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

return 0;
}

// Function to create a new node
struct Node* createNode(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->left = NULL;
    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) {
        // If the tree is empty, create a new node
        return createNode(value);
    }

    if (value < root->data) {
        // If the value is smaller, insert it in the left subtree
        root->left = insert(root->left, value);
    } else if (value > root->data) {
        // If the value is larger, insert it in the right subtree
        root->right = insert(root->right, value);
    }

    return root;
}

```

```

}

// In-order traversal (Left, Root, Right)
void inorderTraversal(struct Node* root) {
    if (root != NULL) {
        inorderTraversal(root->left);
        printf("%d ", root->data);
        inorderTraversal(root->right);
    }
}

// Pre-order traversal (Root, Left, Right)
void preorderTraversal(struct Node* root) {
    if (root != NULL) {
        printf("%d ", root->data);
        preorderTraversal(root->left);
        preorderTraversal(root->right);
    }
}

// Post-order traversal (Left, Right, Root)
void postorderTraversal(struct Node* root) {
    if (root != NULL) {
        postorderTraversal(root->left);
        postorderTraversal(root->right);
        printf("%d ", root->data);
    }
}

// Function to display the tree elements (In-order traversal)
void display(struct Node* root) {
    if (root == NULL) {

```



```

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

        return;

    }

    inorderTraversal(root);

    printf("\n");
}

```

OUTPUT

```

PS C:\Users\DISHA D S\OneDrive\Desktop\cpro> cd "c:\Users\DISHA D S\OneDrive\Desktop\cpro\" ; .
Menu:
1. Insert value into Binary Search Tree
2. In-order Traversal
3. Pre-order Traversal
4. Post-order Traversal
5. Display Tree
6. Exit
Enter your choice: 1
Enter value to insert into BST: 1

Menu:
1. Insert value into Binary Search Tree
2. In-order Traversal
3. Pre-order Traversal
4. Post-order Traversal
5. Display Tree
6. Exit
Enter your choice: 1
Enter value to insert into BST: 34

Menu:
1. Insert value into Binary Search Tree
2. In-order Traversal
3. Pre-order Traversal
4. Post-order Traversal
5. Display Tree
6. Exit
Enter your choice: 1
Enter value to insert into BST: 23

Menu:
1. Insert value into Binary Search Tree
2. In-order Traversal
3. Pre-order Traversal
4. Post-order Traversal
5. Display Tree
6. Exit
Enter your choice: 1
Enter value to insert into BST: 56

```

```
Menu:
1. Insert value into Binary Search Tree
2. In-order Traversal
3. Pre-order Traversal
4. Post-order Traversal
5. Display Tree
6. Exit
Enter your choice: 1
Enter value to insert into BST: 12
```

```
Menu:
1. Insert value into Binary Search Tree
2. In-order Traversal
3. Pre-order Traversal
4. Post-order Traversal
5. Display Tree
6. Exit
Enter your choice: 2
In-order Traversal: 1 12 23 34 56
```

```
Menu:
1. Insert value into Binary Search Tree
2. In-order Traversal
3. Pre-order Traversal
4. Post-order Traversal
5. Display Tree
6. Exit
Enter your choice: 3
Pre-order Traversal: 1 34 23 12 56
```

```
Menu:
1. Insert value into Binary Search Tree
2. In-order Traversal
3. Pre-order Traversal
4. Post-order Traversal
5. Display Tree
6. Exit
Enter your choice: 4
Post-order Traversal: 12 23 56 34 1
```

```
Menu:
1. Insert value into Binary Search Tree
2. In-order Traversal
3. Pre-order Traversal
```

```
Menu:
1. Insert value into Binary Search Tree
2. In-order Traversal
3. Pre-order Traversal
4. Post-order Traversal
5. Display Tree
6. Exit
Enter your choice: 4
Post-order Traversal: 12 23 56 34 1
Menu:
1. Insert value into Binary Search Tree
2. In-order Traversal
3. Pre-order Traversal
4. Post-order Traversal
5. Display Tree
6. Exit
Enter your choice: 5
Tree elements: 1 12 23 34 56

Menu:
1. Insert value into Binary Search Tree
2. In-order Traversal
3. Pre-order Traversal
4. Post-order Traversal
5. Display Tree
6. Exit
Enter your choice: 6
Exiting program.
```

Lab 9:

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

```
#include <stdio.h>

#include <stdlib.h>

#define MAX_VERTICES 10

struct Queue {
    int items[MAX_VERTICES];
    int front, rear;
};

struct Graph {
    int vertices;
    int adj[MAX_VERTICES][MAX_VERTICES];
};

void initQueue(struct Queue* q);
int isQueueEmpty(struct Queue* q);
void enqueue(struct Queue* q, int value);
int dequeue(struct Queue* q);

void initGraph(struct Graph* g, int vertices);
void addEdge(struct Graph* g, int u, int v);
void BFS(struct Graph* g, int startVertex);

int main() {
    struct Graph g;
    int vertices, edges, u, v, startVertex, i;

    // Initialize the graph
    printf("Enter the number of vertices: ");
```

```

scanf("%d", &vertices);
initGraph(&g, vertices);

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

printf("Enter the edges (u v) one by one:\n");
for (i = 0; i < edges; i++) {
    scanf("%d %d", &u, &v);
    addEdge(&g, u, v); // Add edge to the graph
}

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

printf("BFS traversal starting from vertex %d: ", startVertex);
BFS(&g, startVertex);

return 0;
}

// Initialize the queue
void initQueue(struct Queue* q) {
    q->front = -1;
    q->rear = -1;
}

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

```

```

// Enqueue an element
void enqueue(struct Queue* q, int value) {
    if (q->rear == MAX_VERTICES - 1) {
        printf("Queue is full\n");
    } else {
        if (q->front == -1) {
            q->front = 0;
        }
        q->rear++;
        q->items[q->rear] = value;
    }
}

```

```

// Dequeue an element
int dequeue(struct Queue* q) {
    int item;
    if (isEmpty(q)) {
        printf("Queue is empty\n");
        return -1;
    } else {
        item = q->items[q->front];
        q->front++;
        if (q->front > q->rear) {
            q->front = q->rear = -1;
        }
        return item;
    }
}

```

```

// Initialize the graph with given number of vertices
void initGraph(struct Graph* g, int vertices) {
    g->vertices = vertices;
}

```

```

// Initialize adjacency matrix with 0
for (int i = 0; i < vertices; i++) {
    for (int j = 0; j < vertices; j++) {
        g->adj[i][j] = 0;
    }
}

// Add an edge to the graph
void addEdge(struct Graph* g, int u, int v) {
    g->adj[u][v] = 1; // For an undirected graph, also set g->adj[v][u] = 1
    g->adj[v][u] = 1;
}

// Perform BFS traversal starting from a given vertex
void BFS(struct Graph* g, int startVertex) {
    int visited[MAX_VERTICES] = {0}; // Keep track of visited nodes
    struct Queue q;
    initQueue(&q);

    // Mark the start vertex as visited and enqueue it
    visited[startVertex] = 1;
    enqueue(&q, startVertex);

    while (!isQueueEmpty(&q)) {
        int currentVertex = dequeue(&q);
        printf("%d ", currentVertex); // Print the current node

        // Enqueue all unvisited neighbors
        for (int i = 0; i < g->vertices; i++) {
            if (g->adj[currentVertex][i] == 1 && !visited[i]) {
                visited[i] = 1;
            }
        }
    }
}

```

```

        enqueue(&q, i);
    }
}
}
printf("\n");
}

```

OUTPUT:

```

PS C:\Users\DISHA D S\OneDrive\Desktop\cpro> cd "c:\Users\DISHA D S\OneDrive\Desktop\c
Enter the number of vertices: 5
Enter the number of edges: 4
Enter the edges (u v) one by one:
0 1
0 2
1 3
1 4
Enter the starting vertex for BFS: 0
BFS traversal starting from vertex 0: 0 1 2 3 4

```

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

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

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

```
#define MAX_VERTICES 10
```

```
// Graph structure (adjacency matrix representation)
```

```
struct Graph {
```

```
    int vertices;
```

```
    int adj[MAX_VERTICES][MAX_VERTICES];
```

```
};
```

```
// Function prototypes
```

```
void initGraph(struct Graph* g, int vertices);
```

```
void addEdge(struct Graph* g, int u, int v);
```

```
void DFS(struct Graph* g, int vertex, int visited[]);
```

```
int isConnected(struct Graph* g);
```



```

int main() {
    struct Graph g;
    int vertices, edges, u, v, i;

    // Initialize the graph
    printf("Enter the number of vertices: ");
    scanf("%d", &vertices);
    initGraph(&g, vertices);

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

    printf("Enter the edges (u v) one by one:\n");
    for (i = 0; i < edges; i++) {
        scanf("%d %d", &u, &v);
        addEdge(&g, u, v); // Add edge to the graph
    }

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

    return 0;
}

// Initialize the graph with given number of vertices
void initGraph(struct Graph* g, int vertices) {
    g->vertices = vertices;
    // Initialize adjacency matrix with 0

```

```

for (int i = 0; i < vertices; i++) {
    for (int j = 0; j < vertices; j++) {
        g->adj[i][j] = 0;
    }
}

// Add an edge to the graph
void addEdge(struct Graph* g, int u, int v) {
    g->adj[u][v] = 1; // For an undirected graph, also set g->adj[v][u] = 1
    g->adj[v][u] = 1;
}

// DFS traversal function
void DFS(struct Graph* g, int vertex, int visited[]) {
    visited[vertex] = 1; // Mark the current vertex as visited
    // Visit all unvisited adjacent vertices
    for (int i = 0; i < g->vertices; i++) {
        if (g->adj[vertex][i] == 1 && !visited[i]) {
            DFS(g, i, visited);
        }
    }
}

// Function to check if the graph is connected
int isConnected(struct Graph* g) {
    int visited[MAX_VERTICES] = {0}; // Array to track visited vertices

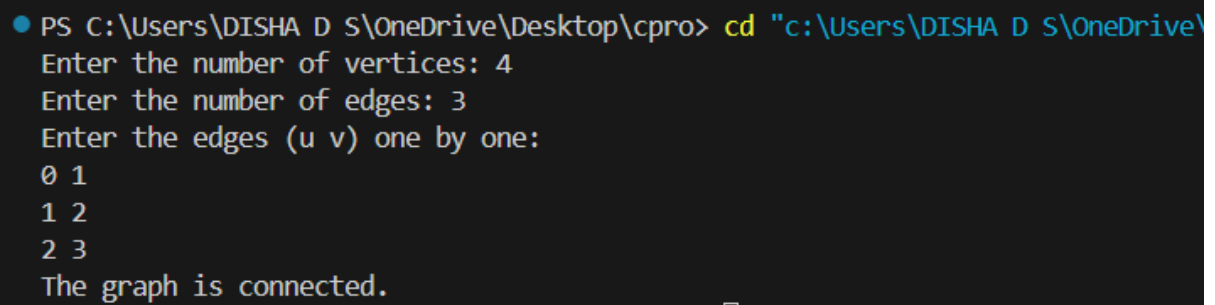
    // Perform DFS starting from vertex 0
    DFS(g, 0, visited);

    // Check if all vertices are visited

```

```
for (int i = 0; i < g->vertices; i++) {  
    if (visited[i] == 0) {  
        return 0; // If any vertex is not visited, the graph is disconnected  
    }  
}  
return 1; // If all vertices are visited, the graph is connected  
}
```

OUTPUT:



```
PS C:\Users\DISHA D S\OneDrive\Desktop\cpro> cd "c:\Users\DISHA D S\OneDrive\
Enter the number of vertices: 4
Enter the number of edges: 3
Enter the edges (u v) one by one:
0 1
1 2
2 3
The graph is connected.
```

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 $H: K \rightarrow L$ as $H(K)=K \bmod m$ (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>
```

```
#define MAX_EMPLOYEES 100 // Max number of employees (hash table size)
```

```
#define MAX_KEY_LENGTH 4 // Employee key length (4 digits)
```

```
#define MAX_ADDR_LENGTH 2 // Address length (2 digits)
```

```
// Structure for storing employee record
```

```
struct Employee {
```

```
    int key;          // Unique 4-digit key
```

```
    char name[50];    // Employee name
```

```
};
```

```
// Hash table structure
```

```
struct HashTable {
```

```
    struct Employee *table[MAX_EMPLOYEES]; // Array of employee records
```

```
    int m;                // Hash table size (m memory locations)
```

```
};
```

```
// Function to initialize the hash table
```

```
void initHashTable(struct HashTable *ht, int size) {
```

```
    ht->m = size;
```

```
    for (int i = 0; i < size; i++) {
```

```
        ht->table[i] = NULL; // Initialize all locations to NULL (empty)
```

```

    }
}

// Hash function  $H(K) = K \bmod m$ 
int hashFunction(int key, int m) {
    return key % m;
}

// Linear probing to handle collisions
int linearProbing(struct HashTable *ht, int key) {
    int index = hashFunction(key, ht->m);
    int originalIndex = index; // Keep track of the original index

    // Keep probing until an empty slot is found
    while (ht->table[index] != NULL) {
        // If the key already exists in the hash table, return the index
        if (ht->table[index]->key == key) {
            return index;
        }
        index = (index + 1) % ht->m; // Move to the next location
        if (index == originalIndex) {
            return -1; // If we looped back to the original index, table is full
        }
    }

    return index;
}

// Insert employee record into the hash table
void insertEmployee(struct HashTable *ht, int key, char name[]) {
    int index = linearProbing(ht, key);
    if (index != -1) {

```

```

        // Allocate memory for a new employee record and insert it into the hash table
        struct Employee *emp = (struct Employee *)malloc(sizeof(struct Employee));
        emp->key = key;
        snprintf(emp->name, sizeof(emp->name), "%s", name); // Copy the name
        ht->table[index] = emp;
    } else {
        printf("Error: Hash Table is full.\n");
    }
}

// Display the hash table contents
void displayHashTable(struct HashTable *ht) {
    printf("Hash Table Contents:\n");
    for (int i = 0; i < ht->m; i++) {
        if (ht->table[i] != NULL) {
            printf("Index %d -> Key: %d, Name: %s\n", i, ht->table[i]->key, ht->table[i]->name);
        } else {
            printf("Index %d -> Empty\n", i);
        }
    }
}

int main() {
    struct HashTable ht;
    int m, N; // m = size of hash table, N = number of employee records
    int key;
    char name[50];

    // Ask the user for hash table size and number of employee records
    printf("Enter the size of the hash table (m): ");
    scanf("%d", &m);
    initHashTable(&ht, m);

```

```
printf("Enter the number of employee records: ");
scanf("%d", &N);

// Input employee records
for (int i = 0; i < N; i++) {
    printf("\nEnter Employee %d details:\n", i + 1);
    printf("Enter 4-digit unique key: ");
    scanf("%d", &key);
    printf("Enter employee name: ");
    scanf(" %[^\n]", name); // Read the entire line for name
    insertEmployee(&ht, key, name);
}

// Display the hash table contents
displayHashTable(&ht);

return 0;
}
```

OUTPUT:

```
Enter the size of the hash table (m): 10
```

```
Enter the number of employee records: 3
```

```
Enter Employee 1 details:
```

```
Enter 4-digit unique key: 1234
```

```
Enter employee name: aaa
```

```
Enter Employee 2 details:
```

```
Enter 4-digit unique key: 5678
```

```
Enter employee name: bbb
```

```
Enter Employee 3 details:
```

```
Enter 4-digit unique key: 9499
```

```
Enter employee name: ccc
```

```
Hash Table Contents:
```

```
Index 0 -> Empty
```

```
Index 1 -> Empty
```

```
Index 2 -> Empty
```

```
Index 3 -> Empty
```

```
Index 4 -> Key: 1234, Name: aaa
```

```
Index 5 -> Empty
```

```
Index 6 -> Empty
```

```
Index 7 -> Empty
```

```
Index 8 -> Key: 5678, Name: bbb
```

```
Index 9 -> Key: 9499, Name: ccc
```

```
PS C:\Users\DISHA D S\OneDrive\Desktop\cpro> |
```