VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
On

DATA STRUCTURES (23CS3PCDST)

Submitted by

RUSHILA.V (1BM22CS226)

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 RUSHILA.V **(1BM22CS226)**, who is a bonafide student of **B. M. S. College of Engineering**. It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum during the year 2023-24. The Lab report has been approved as it satisfies the academic requirements in respect of Data structures Lab - **(23CS3PCDST) work** prescribed for the said degree.

Prof. Sneha S Bagalkot
Assistant Professor
Department of CSE
BMSCE, Bengaluru

Dr. Jyothi S Nayak
Professor and Head
Department of CSE
BMSCE, Bengaluru

Index Sheet

Course outcomes:

CO1	Apply the concept of linear and nonlinear data structures.
CO2	Analyze data structure operations for a given problem
CO3	Design and develop solutions using the operations of linear and nonlinear data structure for a given specification.
CO4	Conduct practical experiments for demonstrating the operations of different data structures.

Lab program 1:

Write a program to simulate the working of stack using an array with the following:
a) Push
b) Pop
c) Display
The program should print appropriate messages for stack overflow, stack underflow.

```
#include <stdio.h>
#include<stdlib.h>
#define STACK_SIZE 5
void push(int st[],int *top)
{
        int item;
        if(*top==STACK_SIZE-1)
                printf("Stack overflow\n");
        else
        {
                printf("\nEnter an item :");
                scanf("%d",&item);
                (*top)++;
                st[*top]=item;
        }
}
void pop(int st[],int *top)
{
        if(*top==-1)
                printf("Stack underflow\n");
        else
        {
                printf("\n%d item was deleted",st[(*top)--]);
        }
}
void display(int st[],int *top)
{
        int i;
        if(*top==-1)
                printf("Stack is empty\n");
        for(i=0;i<=*top;i++)
                printf("%d\t",st[i]);
}
void main()
{
        int st[10],top=-1, c,val_del;
        while(1)
        {
                printf("\n1. Push\n2. Pop\n3. Display\n");
                printf("\nEnter your choice :");
                scanf("%d",&c);
```

```
                switch(c)
                {
                        case 1: push(st,&top);
                                break;
                        case 2: pop(st,&top);
                                break;
                        case 3: display(st,&top);
                                break;
                        default: printf("\nInvalid choice!!!");
                                exit(0);
                }
        }
}
```

Output:

```
1. Push
2. Pop
3. Display

Enter your choice :1

Enter an item :12

1. Push
2. Pop
3. Display

Enter your choice :1

Enter an item :65

1. Push
2. Pop
3. Display

Enter your choice :1

Enter an item :45

1. Push
2. Pop
3. Display

Enter your choice :1
Stack overflow
```

```
1. Push
2. Pop
3. Display

Enter your choice :2

45 item was deleted
1. Push
2. Pop
3. Display

Enter your choice :2

65 item was deleted
1. Push
2. Pop
3. Display

Enter your choice :3
12
1. Push
2. Pop
3. Display

Enter your choice :2

12 item was deleted
1. Push
2. Pop
3. Display

Enter your choice :2
Stack underflow

1. Push
2. Pop
3. Display

Enter your choice :4

Invalid choice!!!
```

Lab Program-2:

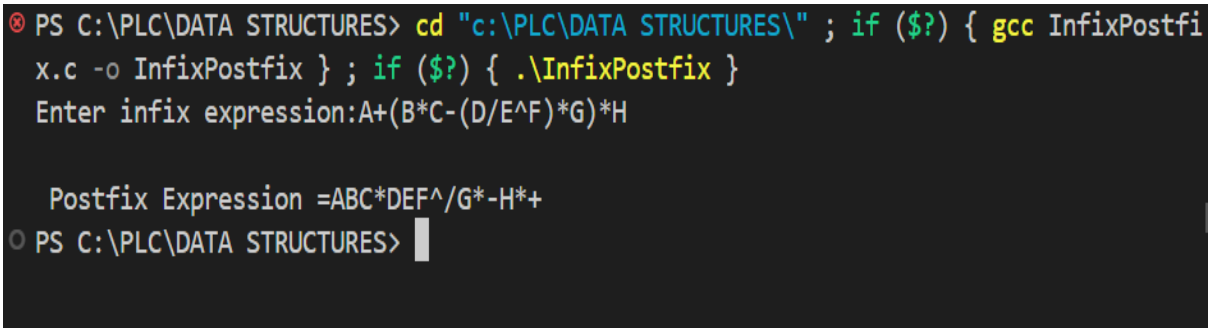
WAP to convert a given valid parenthesized infix arithmetic expression to postfix expression. The expression consists of single character operands and the binary operators + (plus), - (minus), * (multiply) and / (divide)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX 50
char stack[MAX];
char infix[MAX];
char postfix[MAX];
int top = -1;
void push(char);
char pop();
int isEmpty();
void inToPost();
void print();
int precedence(char);
int main()
{
   printf("Enter infix expression: ");
   gets(infix);
   inToPost();
   print();
   return 0;
}
void inToPost()
{
   int i, j = 0;
   char symbol, next;
   for (i = 0; i < strlen(infix); i++)
   {
      symbol = infix[i];
      switch (symbol)
      {
      case '(':
         push(symbol);
         break;
      case ')':
         while ((next = pop()) != '(')
            postfix[j++] = next;
         break;
      case '+':
      case '-':
      case '*':
      case '/':
      case '^':
         while (!isEmpty() && precedence(stack[top]) >= precedence(symbol))
```

```
                postfix[j++] = pop();
            push(symbol);
            break;
        default:
            postfix[j++] = symbol;
        }
    }

    while (!isEmpty())
        postfix[j++] = pop();
    postfix[j] = '\0';
}

int precedence(char symbol)
{
    switch (symbol)
    {
    case '^':
        return 3;
    case '/':
    case '*':
        return 2;
    case '+':
    case '-':
        return 1;
    default:
        return 0;
    }
}

void print()
{
    int i = 0;
    printf("The equivalent postfix expression is: ");
    while (postfix[i])
    {
        printf("%c ", postfix[i++]);
    }
    printf("\n");
}

void push(char c)
{
    if (top == MAX - 1)
    {
        printf("Stack overflow\n");
        exit(1);
    }
    top++;
    stack[top] = c;
```

```
}

char pop()
{
   char c;
   if (top == -1)
   {
      printf("Stack underflow\n");
      exit(1);
   }
   c = stack[top];
   top = top - 1;
   return c;
}

int isEmpty()
{
   if (top == -1)
      return 1;
   else
      return 0;
}
```

Output:

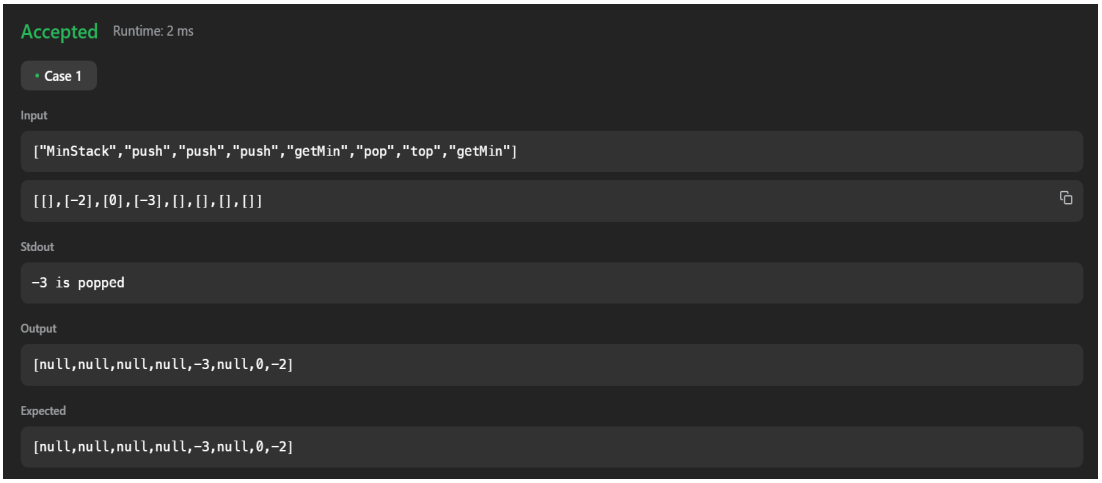


Leetcode-Minstack

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

#define STACK_SIZE 1000

typedef struct {
   int size;
   int top;
   int *s;
   int *minstack;

} MinStack;
```

```
MinStack* minStackCreate() {
    MinStack *st = (MinStack*) malloc(sizeof(MinStack));
    if(st == NULL)
    {
        printf("Memory allocation failed");
        exit(EXIT_FAILURE);
    }
    st->size = STACK_SIZE;
    st->top = -1;
    st->s = (int*) malloc (st->size * sizeof(int));
    st->minstack = (int*) malloc (st->size * sizeof(int));
    if(st->s == NULL || st->minstack == NULL)
    {
        printf("Memory allocation failed");
        free(st->s);
        free(st->minstack);
        free(st);
        exit(EXIT_FAILURE);
    }
    return st;
}

void minStackPush(MinStack* obj, int val) {
    if(obj->top == obj->size - 1)
    {
        printf("Stack is overflow\n");
        return;
    }
    obj->top++;
    obj->s[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) {
    if(obj->top == -1)
    {
        printf("Underflow\n");
        return;
    }
    int value = obj->s[obj->top];
    obj->top--;
    printf("%d is popped\n", value);
}
```

```
int minStackTop(MinStack* obj) {
    if(obj->top == -1)
    {
        printf("Underflow\n");
        exit(EXIT_FAILURE);
    }
    return obj->s[obj->top];
}

int minStackGetMin(MinStack* obj) {
    if(obj->top == -1)
    {
        printf("Underflow\n");
        exit(EXIT_FAILURE);
    }
    return obj->minstack[obj->top];
}

void minStackFree(MinStack* obj) {
    free(obj->s);
    free(obj->minstack);
    free(obj);
}

int main() {
    MinStack* obj = minStackCreate();
    minStackPush(obj, 5);
    minStackPush(obj, 3);
    minStackPush(obj, 8);
    minStackPop(obj);
    printf("Top element: %d\n", minStackTop(obj));
    printf("Min element: %d\n", minStackGetMin(obj));
    minStackFree(obj);
    return 0;
}
```



Lab Program-3(a):

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

```
#include<stdio.h>
#include<stdlib.h>
#define SIZE 5
void enqueue();
void dequeue();
void show();
struct queue{
    int arr[SIZE];
    int top;
    int rear;
};
struct queue q={{0,0,0,0,0},-1,-1};
  void enqueue(){
     int item;
     if(q.rear == SIZE-1){
        printf("OverFlow \n");
     }
     else{
        if(q.top ==-1 || q.top>=0){
           q.top=0;
           printf("Enter the element to insert:");
           scanf("%d",&item);
           printf("\n");
           q.rear+=1;
           q.arr[q.rear]=item;
        }
     }
  }
  void dequeue(){
     if(q.top==-1 || q.top>q.rear){
        printf("UnderFlow \n");
        return;
     }
     else{
        printf("Element deleted:%d \n",q.arr[q.top]);
        q.top=q.top+1;
     }
  }
  void show(){
     if(q.top == -1){
        printf("Empty Queue \n");
     }
     else{
        printf("Queue: \n");
        for(int i=q.top;i<=q.rear;i++){
           printf("%d ",q.arr[i]);
```

```
        }
        printf("\n");
    }
}
int main(){
    int ch;
    while(1){
        printf("1 for Enqueue \n");
        printf("2 for Dequeue \n");
        printf("3 for Display \n");
        printf("4 Exit \n");
        printf("Enter your choice \n");
        scanf("%d",&ch);
        printf("\n");
        switch(ch){
            case 1:
            enqueue();
            break;
            case 2:
            dequeue();
            break;
            case 3:
            show();
            break;
            case 4:
            exit(0);
            default:
            printf("Wrong Choice \n");
            break;
        }
    }
}
```

Output:

```
Queue operation
1.insertion
2.deletion
3.display
4.exit
enter the choice 1
enter the element to be inserted 12
Queue operation
1.insertion
2.deletion
3.display
4.exit
enter the choice 1
enter the element to be inserted 13
Queue operation
1.insertion
2.deletion
3.display
4.exit
enter the choice 1
enter the element to be inserted 14
Queue operation
1.insertion
2.deletion
3.display
4.exit
enter the choice 1
enter the element to be inserted 15
Queue overflow
```

```
Queue operation
1.insertion
2.deletion
3.display
4.exit
enter the choice 2
queue underflow
```

Lab Program-3(b):

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

WAP to simulate the working of a circular queue of integers using an array. Provide the following operations.
a) Insert
b) Delete
c) Display
The program should print appropriate messages for queue empty and queue overflow Conditions.

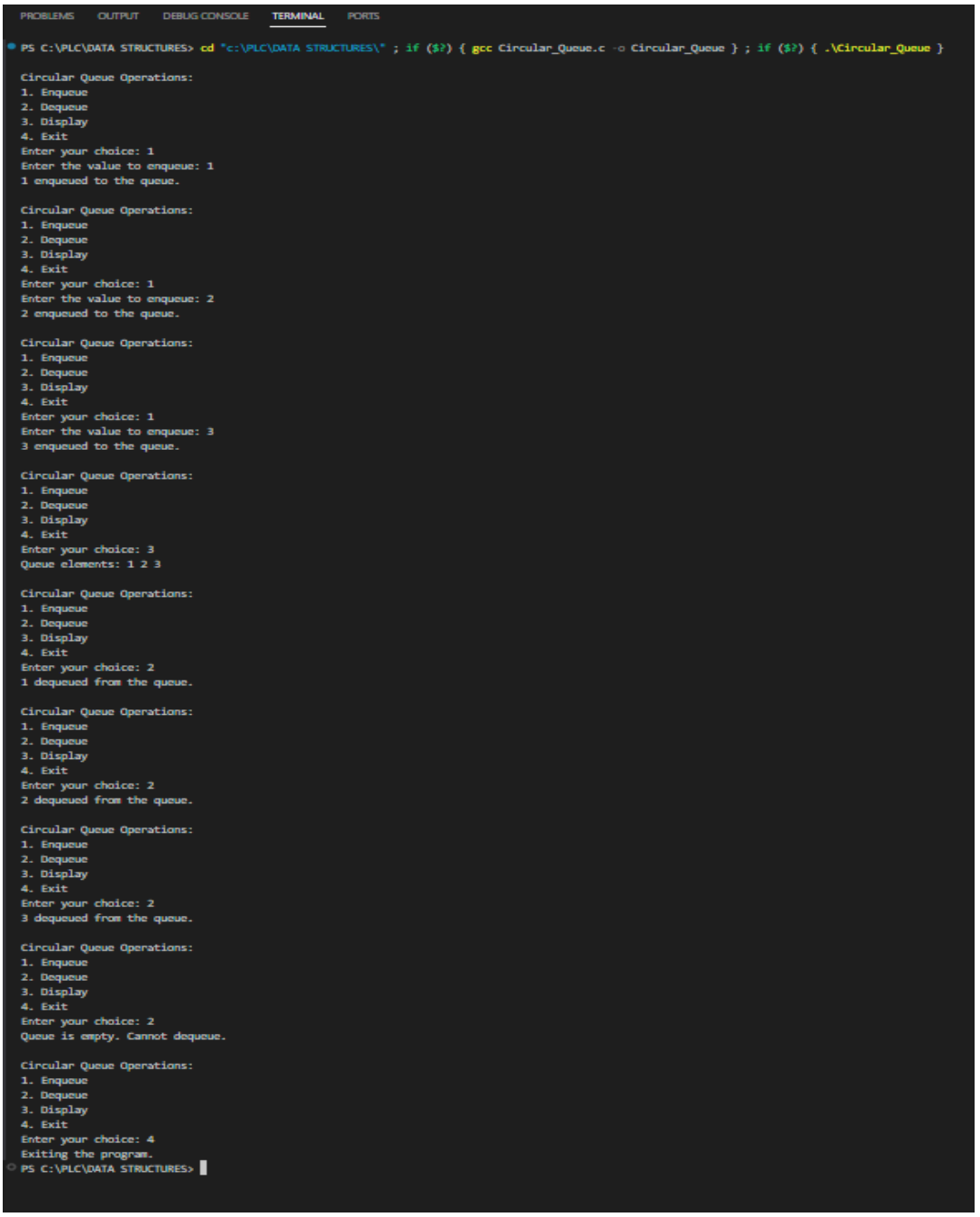
```
#include <stdio.h>
#define MAX_SIZE 5

// Circular Queue variables
int items[MAX_SIZE];
```

```
int front = -1, rear = -1;

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

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

// Function to enqueue an element into the circular queue
void enqueue(int value) {
   if (isFull()) {
      printf("Queue is full. Cannot enqueue %d.\n", value);
      return;
   }

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

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

// Function to dequeue an element from the circular queue
int dequeue() {
   int dequeuedItem;

   if (isEmpty()) {

printf("Queue is empty. Cannot dequeue.\n");
      return -1;
   }

   dequeuedItem = items[front];

   if (front == rear) {
      // If there was only one element in the queue
      front = -1;
      rear = -1;
   } else {
      front = (front + 1) % MAX_SIZE;
   }

   printf("%d dequeued from the queue.\n", dequeuedItem);
   return dequeuedItem;
```

```
}

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

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

int main() {
    int choice, value;

    do {
        printf("\nCircular Queue 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(value);
                break;

            case 2:
                dequeue();
                break;

            case 3:
                display();
                break;

            case 4:
                printf("Exiting the program.\n");
                break;

            default:
                printf("Invalid choice. Please enter a valid option.\n");
```

```
        }
    } while (choice != 4);

    return 0;
}
```

Output:



Lab Program-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.

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

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

// Function to create a new node
struct Node* createNode(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation failed\n");
        exit(1);
    }
    newNode->data = value;
    newNode->next = NULL;
    return newNode;
}

// Function to insert a node at the beginning of the list
struct Node* insertAtBeginning(struct Node* head, int value) {
    struct Node* newNode = createNode(value);
    newNode->next = head;
    return newNode;
}

// Function to insert a node at any given position in the list
struct Node* insertAtAnyPos(struct Node* head, int value, int pos) {
    struct Node* newNode = createNode(value);
    struct Node* temp = head;
    if (pos == 1) {
        newNode->next = head;
        return newNode;
    }
    for (int i = 1; i < pos - 1; i++) {
        temp = temp->next;
    }
    newNode->next = temp->next;
    temp->next = newNode;
    return head;
}
```

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

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

int main() {
    struct Node* head = NULL;
    int choice, value, pos;
    while (1) {
        printf("1. Insert at end\n");
        printf("2. Insert at beginning\n");
        printf("3. Insert at any position\n");
        printf("4. Display List\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Enter the value to be inserted: ");
                scanf("%d", &value);
                head = insertAtEnd(head, value);
                break;
            case 2:
                printf("Enter the value to be inserted: ");
                scanf("%d", &value);
                head = insertAtBeginning(head, value);
                break;
            case 3:
                printf("Enter the value to be inserted: ");
                scanf("%d", &value);
                printf("Enter the position at which to insert: ");
                scanf("%d", &pos);
```

```
            head = insertAtAnyPos(head, value, pos);
            break;
        case 4:
            displayList(head);
            break;
        case 5:
            exit(0);
        default:
            printf("Invalid choice. Please try again.\n");
      }
    }
    return 0;
}
```

```
1. Insert at end
2. Insert at beginning
3. Insert at any position
4. Display List
5. Exit
Enter your choice: 1
Enter the value to be inserted: 23
1. Insert at end
2. Insert at beginning
3. Insert at any position
4. Display List
5. Exit
Enter your choice: 4
14 -> 13 -> 12 -> 23 -> NULL
1. Insert at end
2. Insert at beginning
3. Insert at any position
4. Display List
5. Exit
Enter your choice: 3
Enter the value to be inserted: 12
Enter the position at which to insert: 2
1. Insert at end
2. Insert at beginning
3. Insert at any position
4. Display List
5. Exit
Enter your choice: 4
14 -> 12 -> 13 -> 12 -> 23 -> NULL
1. Insert at end
2. Insert at beginning
3. Insert at any position
4. Display List
5. Exit
```

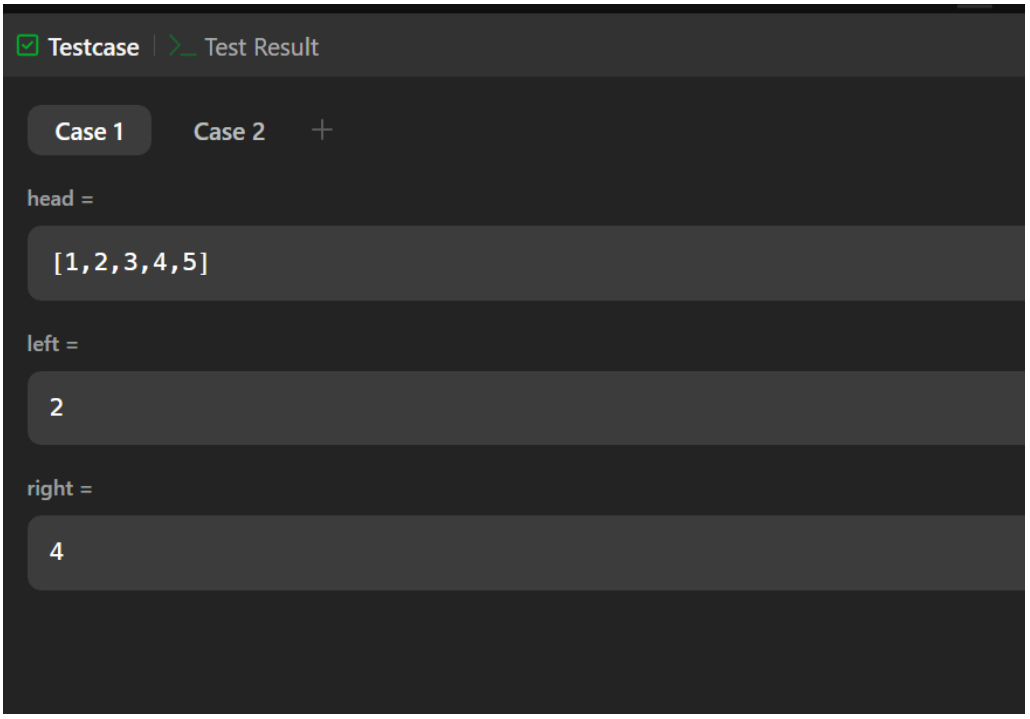
```
1. Insert at end
2. Insert at beginning
3. Insert at any position
4. Display List
5. Exit
Enter your choice: 2
Enter the value to be inserted: 12
1. Insert at end
2. Insert at beginning
3. Insert at any position
4. Display List
5. Exit
Enter your choice: 2
Enter the value to be inserted: 13
1. Insert at end
2. Insert at beginning
3. Insert at any position
4. Display List
5. Exit
Enter your choice: 2
Enter the value to be inserted: 14
1. Insert at end
2. Insert at beginning
3. Insert at any position
4. Display List
5. Exit
Enter your choice: 4
14 -> 13 -> 12 -> NULL
1. Insert at end
2. Insert at beginning
3. Insert at any position
4. Display List
5. Exit
```

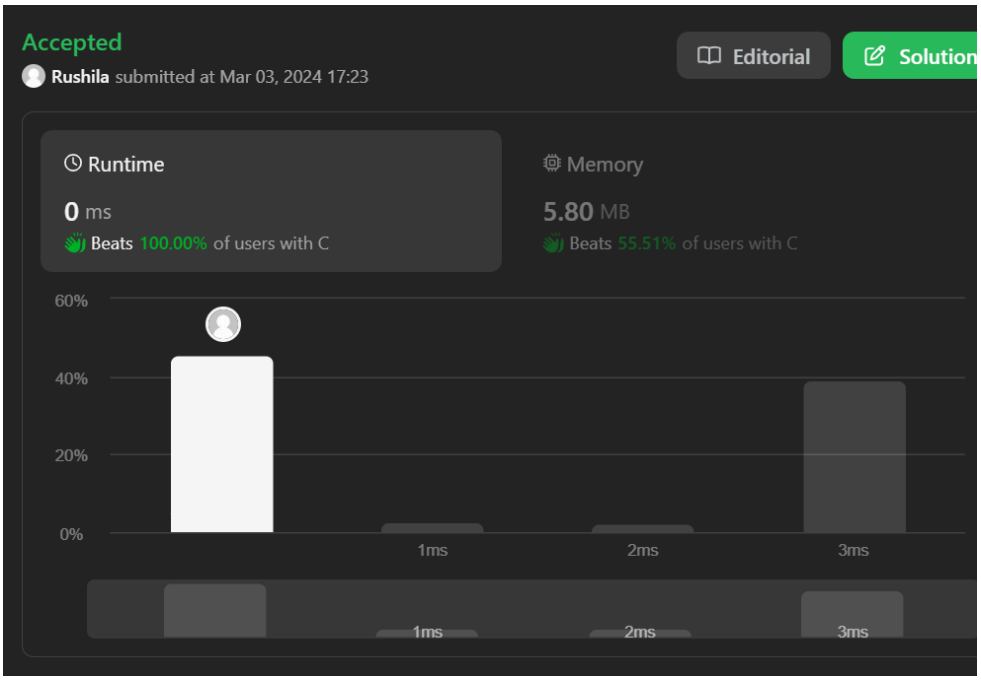
Leetcode-Reverse Linked List

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

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

struct ListNode* reverseBetween(struct ListNode* head, int left, int right) {
    if (head == NULL || left == right) {
        return head;
    }

    struct ListNode* dummy = (struct ListNode*)malloc(sizeof(struct ListNode));
    dummy->next = head;
    struct ListNode* prev = dummy;

    // Move to the node just before the reversal starts
    for (int i = 1; i < left; ++i) {
        prev = prev->next;
    }

    // Reverse the nodes from left to right
    struct ListNode* current = prev->next;
    struct ListNode* next = NULL;
```

```
    struct ListNode* tail = current;

    for (int i = left; i <= right; ++i) {
        struct ListNode* temp = current->next;
        current->next = next;
        next = current;
        current = temp;
    }

    // Connect the reversed portion with the rest of the list
    prev->next = next;
    tail->next = current;

    // Return the modified list
    struct ListNode* result = dummy->next;
    free(dummy); // Free the dummy node
    return result;
}

int main() {
    // Sample usage
    // Constructing a sample linked list
    struct ListNode* head = (struct ListNode*)malloc(sizeof(struct ListNode));
    struct ListNode* node1 = (struct ListNode*)malloc(sizeof(struct ListNode));
    struct ListNode* node2 = (struct ListNode*)malloc(sizeof(struct ListNode));
    struct ListNode* node3 = (struct ListNode*)malloc(sizeof(struct ListNode));

    head->val = 1;
    head->next = node1;
    node1->val = 2;
    node1->next = node2;
    node2->val = 3;
    node2->next = node3;
    node3->val = 4;
    node3->next = NULL;

    // Printing the original list
    printf("Original List: ");
    struct ListNode* current = head;
    while (current != NULL) {
        printf("%d -> ", current->val);
        current = current->next;
    }
    printf("NULL\n");

    // Reversing a portion of the list (e.g., from position 2 to 3)
    int left = 2, right = 3;
    head = reverseBetween(head, left, right);

    // Printing the reversed list
    printf("Reversed List: ");
    current = head;
```

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

    // Freeing memory
    while (head != NULL) {
        struct ListNode* temp = head;
        head = head->next;
        free(temp);
    }

    return 0;
}
```





Lab Program-5:

WAP to Implement Circular Singly Linked List with following operations
a) Create a linked list.
b) Insertion of a node at first position, and at end of list and at any position
c) Delete a node at front and at the end of the list and at any position
d) Display the contents of the linked list.

```
#include<stdio.h>
#include<stdlib.h>
struct Node{
    int data;
    struct Node *next;
};
//Create Circular Linked List
struct Node* createCircularLL(struct Node* head){
    int data;
    struct Node*p;
    printf("Enter -1 to stop.\n");
    printf("Enter Number:");
    scanf("%d",&data);
    while(data!=-1){

        struct Node *newNode=(struct Node*)malloc(sizeof(struct Node));
        newNode->data=data;
        if(head==NULL){
            newNode->next=newNode;
            head=newNode;
        }
        else{
            p=head;
            while(p->next!=head){
                p=p->next;
            }
```

```
            p->next=newNode;
            newNode->next=head;

        }
    printf("Enter Number:");
    scanf("%d",&data);
    }
    return head;
}
//Display Circular Linked List
struct Node* displayCircularLL(struct Node* head){
    struct Node* p;
    p=head;
    printf("Circular List Elements:");
    while(p->next !=head){
        printf("%d ",p->data);
        p=p->next;
    }

printf("%d ",p->data);
    printf("\n");
    return head;
}
//Insert At Beginning
struct Node* insertFirst(struct Node* head){
    int num;
    struct Node *p;
    p=head;
    printf("Enter Number:");
    scanf("%d",&num);
    struct Node *newNode=(struct Node*)malloc(sizeof(struct Node));
    newNode->data=num;
    while(p->next!=head){
        p=p->next;
    }
    p->next=newNode;
    newNode->next=head;
    head=newNode;
    return head;
}
//Insert At End
struct Node* insertEnd(struct Node* head){
    int num;
    struct Node *p;
    p=head;
    printf("Enter Number:");
    scanf("%d",&num);
    struct Node *newNode=(struct Node*)malloc(sizeof(struct Node));
    newNode->data=num;
    while(p->next!=head){
        p=p->next;
    }
```

```
    p->next=newNode;
    newNode->next=head;
    return head;
}
//Insert At Any Position
struct Node* insertPosition(struct Node* head, int pos){
    int num,i=0;
    struct Node *p;
    p=head;
    printf("Enter Number:");
    scanf("%d",&num);
    struct Node *newNode=(struct Node*)malloc(sizeof(struct Node));
    newNode->data=num;
    if(pos==0){
        head=insertFirst(head);
        return head;
    }
    else{

 while(i!=pos-1){
            p=p->next;
            i++;
        }
    newNode->next=p->next;
    p->next=newNode;
    return head;
    }

}
//Delete From Front
struct Node* DelFromFront(struct Node* head){
    struct Node *p;
    p=head;
    while(p->next!=head){
        p=p->next;
    }
    p->next=head->next;
    free(head);
    head=p->next;
}
//Delete From End
struct Node* DelFromEnd(struct Node* head){
    struct Node *p,*preNode;
    p=head;
    while(p->next!=head){
        preNode=p;
        p=p->next;
    }
    preNode->next=p->next;
    free(p);
    return head;
}
```

```
//Delete From Any Position
struct Node* DelFromPos(struct Node* head,int pos){
   int i=0;
   struct Node* p,*preNode;
   p=head;
   if(pos==0){
      head=DelFromFront(head);
      return head;
   }
   else{
      while(i!=pos){
         preNode=p;
         p=p->next;
         i++;
      }
   preNode->next=p->next;
   free(p);
   return head;

}
}
int main(){
   struct Node* head=NULL;
   head=createCircularLL(head);
   printf("Linked List Created. \n");
   head=displayCircularLL(head);

   head=insertFirst(head);
   printf("Linked List after insertion at begining.\n");
   head=displayCircularLL(head);

   head=insertEnd(head);
   printf("Linked List after insertion at end.\n");
   head=displayCircularLL(head);

   head=insertPosition(head, 3);
   printf("Linked List after insertion at pos.\n");
   head=displayCircularLL(head);

   head=DelFromFront(head);
   printf("Linked List after deletion from front\n");
   head=displayCircularLL(head);

   head=DelFromEnd(head);
   printf("Linked List after Deletion form end.\n");
   head=displayCircularLL(head);

   head=DelFromPos(head,2);
   printf("Linked List after Deletion form pos.\n");
   head=displayCircularLL(head);
}
```

Output:

```
PS C:\PLC\DATA STRUCTURES> cd "c:\PLC\DATA STRUCTURES\" ; if ($?) { gcc Circ
ular_LL.c -o Circular_LL } ; if ($?) { .\Circular_LL }
Enter -1 to stop.
Enter Number:1
Enter Number:2
Enter Number:3
Enter Number:-1
Linked List Created.
Circular List Elements:1 2 3
Enter Number:10
Linked List after insertion at begining.
Circular List Elements:10 1 2 3
Enter Number:7
Linked List after insertion at end.
Circular List Elements:10 1 2 3 7
Enter Number:5
Linked List after insertion at pos.
Circular List Elements:10 1 2 5 3 7
Linked List after deletion from front
Circular List Elements:1 2 5 3 7
Linked List after Deletion form end.
Circular List Elements:1 2 5 3
Linked List after Deletion form pos.
Circular List Elements:1 2 3
PS C:\PLC\DATA STRUCTURES>
```

Leetcode : Split Linked list

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

// Define a structure for a ListNode
typedef struct ListNode lnode;

// Function to get the length of the linked list
int get_len(lnode *head) {
    int n = 0;
    // Traverse the linked list and count the number of nodes
    while (head) {
        n++;
        head = head->next;
    }
    return n;
}

// Function to split the linked list into k parts
struct ListNode** splitListToParts(struct ListNode* head, int k, int* returnSize) {
    // Get the length of the linked list
    int n = get_len(head);
```

```
    // Set the returnSize pointer to the number of parts
    *returnSize = k;
    // Allocate memory for an array of ListNode pointers
    lnode **list = (lnode**)calloc(k, sizeof(lnode*));
    // Temporary pointer to traverse the linked list
    lnode *t = head;

    // If the length of the linked list is greater than k
    if (n > k) {
        for (int i = 0; i < k; i++) {
            // Calculate the number of elements in the current part
            int elems = i < n % k ? n / k + 1 : n / k;
            // Traverse the linked list to the end of the current part
            for (int j = 0; j < elems; j++) {
                t = head;
                head = head->next;
            }
            // Disconnect the current part from the rest of the list
            t->next = NULL;
            // Store the head of the current part in the array
            list[i] = head;
        }
    } else { // If the length of the linked list is less than or equal to k
        for (int i = 0; i < n; i++) {
            // Store each node as a separate part in the array
            list[i] = head;
            head = head->next;
            // Disconnect each node from the rest of the list
            list[i]->next = NULL;
        }
    }
    // Return the array of ListNode pointers
    return list;
}
```

Lab Program-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>

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

// Function to append a new node to the end of the list
   void append(struct Node** head_ref, int new_data) {
   struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
   struct Node* last = *head_ref;

   new_node->data = new_data;
   new_node->next = NULL;

   if (*head_ref == NULL) {
      *head_ref = new_node;
      return;
   }

   while (last->next != NULL)
      last = last->next;

   last->next = new_node;
```

```
}

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

// Function to sort the linked list
void sort(struct Node** head_ref) {
    struct Node *current, *index;
    int temp;

    if (*head_ref == NULL)
        return;

    current = *head_ref;
    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;
    }
}

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

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

// Function to concatenate two linked lists
void concatenate(struct Node** head1_ref, struct Node* head2) {
    struct Node* current = *head1_ref;
```

```
    if (*head1_ref == NULL) {
        *head1_ref = head2;
        return;
    }

    while (current->next != NULL)
        current = current->next;

    current->next = head2;
}

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

    while (1) {
        printf("\n1. Append to list 1\n");
        printf("2. Append to list 2\n");
        printf("3. Sort list 1\n");
        printf("4. Reverse list 1\n");
        printf("5. Concatenate lists\n");
        printf("6. Display list 1\n");
        printf("7. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter data to append to list 1: ");
                scanf("%d", &data);
                append(&list1, data);
                break;
            case 2:
                printf("Enter data to append to list 2: ");
                scanf("%d", &data);
                append(&list2, data);
                break;
            case 3:
                sort(&list1);
                printf("List 1 sorted.\n");
                break;
            case 4:
                reverse(&list1);
                printf("List 1 reversed.\n");
                break;
            case 5:
                concatenate(&list1, list2);
                printf("Lists concatenated.\n");
                break;
            case 6:
```

```
            printf("List 1: ");
            display(list1);
            break;
        case 7:
            printf("Exiting program.\n");
            exit(0);
        default:
            printf("Invalid choice! Please try again.\n");
    }
  }

  return 0;
}
```

Output:

```
PS C:\PLC\DATA STRUCTURES> cd "c:\PLC\DATA STRUCTURES\" ; if ($?) { gcc Sort
_Reverse_Concatenate_LL.c -o Sort_Reverse_Concatenate_LL } ; if ($?) { .\Sor
t_Reverse_Concatenate_LL }
Enter -1 to stop.
Enter Number:4
Enter Number:3
Enter Number:5
Enter Number:-1
Linked List Elements:4 3 5
After Sorting
Linked List Elements:3 4 5
After Reversal
Linked List Elements:5 4 3
Enter 1st Linked List :
Enter -1 to stop.
Enter Number:1
Enter Number:2
Enter Number:3
Enter Number:-1
Enter 2nd Linked List :
Enter -1 to stop.
Enter Number:4
Enter Number:5
Enter Number:6
Enter Number:-1
After Concatenation
Linked List Elements:1 2 3 4 5 6
PS C:\PLC\DATA STRUCTURES>
```

Lab Program-6(b):

WAP to Implement Single Link List to simulate Stack Operations.

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

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

// Function to create a new node
struct Node* createNode(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation failed\n");
        exit(1);
    }
    newNode->data = value;
    newNode->next = NULL;
    return newNode;
}

// Function to push a value onto the stack
void push(struct Node** topRef, int value) {
    struct Node* newNode = createNode(value);
    newNode->next = *topRef;
    *topRef = newNode;
}

// Function to pop a value from the stack
int pop(struct Node** topRef) {
    if (*topRef == NULL) {
        printf("Stack underflow\n");
        exit(1);
    }
    struct Node* temp = *topRef;
    int poppedValue = temp->data;
    *topRef = (*topRef)->next;
    free(temp);
    return poppedValue;
}

// Function to peek at the top element of the stack
int peek(struct Node* top) {
    if (top == NULL) {
        printf("Stack is empty\n");
        exit(1);
    }
    return top->data;
```

```
}

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

// Function to display the stack
void display(struct Node* top) {
    if (isEmpty(top)) {
        printf("Stack is empty\n");
        return;
    }
    struct Node* temp = top;
    printf("Stack: ");
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

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

    while (1) {
        printf("\n1. Push\n");
        printf("2. Pop\n");
        printf("3. Peek\n");
        printf("4. Display\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter value to push: ");
                scanf("%d", &value);
                push(&top, value);
                printf("%d pushed onto the stack.\n", value);
                break;
            case 2:
                if (!isEmpty(top)) {
                    printf("Popped value: %d\n", pop(&top));
                }
                break;
            case 3:
                if (!isEmpty(top)) {
                    printf("Top element: %d\n", peek(top));
                }
                break;
```

```
            case 4:
                display(top);
                break;
            case 5:
                // Free memory before exiting
                while (top != NULL) {
                    struct Node* temp = top;
                    top = top->next;
                    free(temp);
                }
                exit(0);
            default:
                printf("Invalid choice. Please try again.\n");
        }
    }

    return 0;
}
```

Output:

```
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 1
Enter value to push: 12
12 pushed onto the stack.

1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 1
Enter value to push: 23
23 pushed onto the stack.

1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 1
Enter value to push: 56
56 pushed onto the stack.
```

```
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 1
Enter value to push: 90
90 pushed onto the stack.

1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 4
Stack: 90 56 23 12

1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 2
Popped value: 90

1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 4
Stack: 56 23 12
```

Lab Program-6(b):

WAP to Implement Single Link List to simulate Queue Operations.

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

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

// Function to create a new node
struct Node* createNode(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation failed\n");
        exit(1);
    }
```

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

// Function to enqueue (insert at rear) in the queue
void enqueue(struct Node** rearRef, int value) {
    struct Node* newNode = createNode(value);
    if (*rearRef == NULL) {
        *rearRef = newNode;
        return;
    }
    (*rearRef)->next = newNode;
    *rearRef = newNode;
}

// Function to dequeue (remove from front) in the queue
int dequeue(struct Node** frontRef) {
    if (*frontRef == NULL) {
        printf("Queue underflow\n");
        exit(1);
    }
    struct Node* temp = *frontRef;
    int dequeuedValue = temp->data;
    *frontRef = (*frontRef)->next;
    free(temp);
    return dequeuedValue;
}

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

// Function to display the queue
void display(struct Node* front) {
    if (isEmpty(front)) {
        printf("Queue is empty\n");
        return;
    }
    struct Node* temp = front;
    printf("Queue: ");
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

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

```
    int choice, value;

    while (1) {
        printf("\n1. 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 value to enqueue: ");
                scanf("%d", &value);
                enqueue(&rear, value);
                if (front == NULL) {
                    front = rear;
                }
                printf("%d enqueued into the queue.\n", value);
                break;
            case 2:
                if (!isEmpty(front)) {
                    printf("Dequeued value: %d\n", dequeue(&front));
                    if (front == NULL) {
                        rear = NULL; // Reset rear when the last element is dequeued
                    }
                }
                break;
            case 3:
                display(front);
                break;
            case 4:
                // Free memory before exiting
                while (front != NULL) {
                    struct Node* temp = front;
                    front = front->next;
                    free(temp);
                }
                exit(0);
            default:
                printf("Invalid choice. Please try again.\n");
        }
    }

    return 0;
}
```

Output:

```
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 1
Enter value to enqueue: 12
12 enqueued into the queue.

1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 1
Enter value to enqueue: 14
14 enqueued into the queue.

1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 1
Enter value to enqueue: 18
18 enqueued into the queue.

1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 1
Enter value to enqueue: 10
10 enqueued into the queue.
```

```
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 3
Queue: 12 14 18 10

1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 2
Dequeued value: 12

1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 3
Queue: 14 18 10
```

Lab Program-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>

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

// Function to create a new node
struct Node* createNode(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation failed\n");
        exit(1);
    }
    newNode->data = value;
    newNode->prev = NULL;
    newNode->next = NULL;
    return newNode;
}

// Function to insert a new node at the beginning of the list
struct Node* insertAtBeginning(struct Node* head, int value) {
    struct Node* newNode = createNode(value);
    if (head == NULL) {
        return newNode;
    }
    newNode->next = head;
    head->prev = newNode;
    return newNode;
}

// Function to insert a new node to the left of a specific node value
struct Node* insertToLeft(struct Node* head, int value, int target) {
    struct Node* newNode = createNode(value);
    struct Node* current = head;
    while (current != NULL) {
        if (current->data == target) {
            newNode->prev = current->prev;
            newNode->next = current;
```

```
            if (current->prev != NULL) {
                current->prev->next = newNode;
            }
            current->prev = newNode;
            if (current == head) {
                return newNode;
            }
            return head;
        }
        current = current->next;
    }
    printf("Node with value %d not found\n", target);
    return head;
}

// Function to delete a node based on a specific value
struct Node* deleteNode(struct Node* head, int value) {
    if (head == NULL) {
        printf("List is empty\n");
        return NULL;
    }
    struct Node* current = head;
    while (current != NULL) {
        if (current->data == value) {
            if (current->prev != NULL) {
                current->prev->next = current->next;
            }
            if (current->next != NULL) {
                current->next->prev = current->prev;
            }
            if (current == head) {
                head = current->next;
            }
            free(current);
            printf("Node with value %d deleted\n", value);
            return head;
        }
        current = current->next;
    }
    printf("Node with value %d not found\n", value);
    return head;
}

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

```
}

int main() {
    struct Node* head = NULL;
    int choice, value, target;

    while (1) {
        printf("\n1. Create a doubly linked list\n");
        printf("2. Insert a new node at the beginning\n");
        printf("3. Insert a new node to the left of a specific node value\n");
        printf("4. Delete a node based on a specific value\n");
        printf("5. Display the contents of the list\n");
        printf("6. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter the value for the first node: ");
                scanf("%d", &value);
                head = createNode(value);
                break;
            case 2:
                printf("Enter the value to insert at the beginning: ");
                scanf("%d", &value);
                head = insertAtBeginning(head, value);
                break;
            case 3:
                printf("Enter the value to insert: ");
                scanf("%d", &value);
                printf("Enter the target value: ");
                scanf("%d", &target);
                head = insertToLeft(head, value, target);
                break;
            case 4:
                printf("Enter the value to delete: ");
                scanf("%d", &value);
                head = deleteNode(head, value);
                break;
            case 5:
                displayList(head);
                break;
            case 6:
                // Free memory before exiting
                while (head != NULL) {
                    struct Node* temp = head;
                    head = head->next;
                    free(temp);
                }
                exit(0);
            default:
                printf("Invalid choice. Please try again.\n");
```

```
        }
    }

    return 0;
}
```

Output:

```
1. Create a doubly linked list
2. Insert a new node at the beginning
3. Insert a new node to the left of a specific node value
4. Delete a node based on a specific value
5. Display the contents of the list
6. Exit
Enter your choice: 1
Enter the value for the first node: 12

1. Create a doubly linked list
2. Insert a new node at the beginning
3. Insert a new node to the left of a specific node value
4. Delete a node based on a specific value
5. Display the contents of the list
6. Exit
Enter your choice: 1
Enter the value for the first node: 13

1. Create a doubly linked list
2. Insert a new node at the beginning
3. Insert a new node to the left of a specific node value
4. Delete a node based on a specific value
5. Display the contents of the list
6. Exit
Enter your choice: 2
Enter the value to insert at the beginning: 34
```

```
1. Create a doubly linked list
2. Insert a new node at the beginning
3. Insert a new node to the left of a specific node value
4. Delete a node based on a specific value
5. Display the contents of the list
6. Exit
Enter your choice: 2
Enter the value to insert at the beginning: 56

1. Create a doubly linked list
2. Insert a new node at the beginning
3. Insert a new node to the left of a specific node value
4. Delete a node based on a specific value
5. Display the contents of the list
6. Exit
Enter your choice: 2
Enter the value to insert at the beginning: 30

1. Create a doubly linked list
2. Insert a new node at the beginning
3. Insert a new node to the left of a specific node value
4. Delete a node based on a specific value
5. Display the contents of the list
6. Exit
Enter your choice: 3
Enter the value to insert: 98
Enter the target value: 2
Node with value 2 not found
```

```
1. Create a doubly linked list
2. Insert a new node at the beginning
3. Insert a new node to the left of a specific node value
4. Delete a node based on a specific value
5. Display the contents of the list
6. Exit
Enter your choice: 5
List: 30 -> 56 -> 34 -> 13 -> NULL

1. Create a doubly linked list
2. Insert a new node at the beginning
3. Insert a new node to the left of a specific node value
4. Delete a node based on a specific value
5. Display the contents of the list
6. Exit
Enter your choice: 4
Enter the value to delete: 34
Node with value 34 deleted

1. Create a doubly linked list
2. Insert a new node at the beginning
3. Insert a new node to the left of a specific node value
4. Delete a node based on a specific value
5. Display the contents of the list
6. Exit
Enter your choice: 5
List: 30 -> 56 -> 13 -> NULL
```

Leetcode-Rotate List

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

struct ListNode* rotateRight(struct ListNode* head, int k){
   if (head == NULL || k == 0)
      return head;

   int length = GetLength(head);

   if (length == 1)
      return head;

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

Lab Program-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 a structure for a tree node
struct TreeNode {
    int data;
    struct TreeNode* left;
    struct TreeNode* right;
};

// Function to create a new tree node
struct TreeNode* createNode(int value) {
    struct TreeNode* newNode = (struct TreeNode*)malloc(sizeof(struct TreeNode));
    if (newNode == NULL) {
        printf("Memory allocation failed\n");
        exit(1);
    }
    newNode->data = value;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}
```

```
// Function to insert a value into a binary search tree
struct TreeNode* insert(struct TreeNode* root, int value) {
    if (root == NULL) {
        return createNode(value);
    }
    if (value < root->data) {
        root->left = insert(root->left, value);
    } else if (value > root->data) {
        root->right = insert(root->right, value);
    }
    return root;
}

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

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

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

// Function to display the elements of the binary search tree
void displayTree(struct TreeNode* root) {
    if (root == NULL) {
        printf("Tree is empty\n");
        return;
    }
    printf("Elements of the tree: ");
    inorderTraversal(root);
    printf("\n");
```

```
}

int main() {
    struct TreeNode* root = NULL;
    int choice, value;

    while (1) {
        printf("\n1. Insert into binary search tree\n");
        printf("2. Traverse the tree (Inorder)\n");
        printf("3. Traverse the tree (Preorder)\n");
        printf("4. Traverse the tree (Postorder)\n");
        printf("5. Display the elements of the tree\n");
        printf("6. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter value to insert: ");
                scanf("%d", &value);
                root = insert(root, value);
                printf("%d inserted into the tree.\n", value);
                break;
            case 2:
                printf("Inorder traversal: ");
                inorderTraversal(root);
                printf("\n");
                break;
            case 3:
                printf("Preorder traversal: ");
                preorderTraversal(root);
                printf("\n");
                break;
            case 4:
                printf("Postorder traversal: ");
                postorderTraversal(root);
                printf("\n");
                break;
            case 5:
                displayTree(root);
                break;
            case 6:
                // Free memory before exiting
                exit(0);
            default:
                printf("Invalid choice. Please try again.\n");
        }
    }

    return 0;
```

}

```
1. Insert into binary search tree
2. Traverse the tree (Inorder)
3. Traverse the tree (Preorder)
4. Traverse the tree (Postorder)
5. Display the elements of the tree
6. Exit
Enter your choice: 1
Enter value to insert: 23
23 inserted into the tree.

1. Insert into binary search tree
2. Traverse the tree (Inorder)
3. Traverse the tree (Preorder)
4. Traverse the tree (Postorder)
5. Display the elements of the tree
6. Exit
Enter your choice: 1
Enter value to insert: 45
45 inserted into the tree.

1. Insert into binary search tree
2. Traverse the tree (Inorder)
3. Traverse the tree (Preorder)
4. Traverse the tree (Postorder)
5. Display the elements of the tree
6. Exit
Enter your choice: 1
Enter value to insert: 65
65 inserted into the tree.
```

```
1. Insert into binary search tree
2. Traverse the tree (Inorder)
3. Traverse the tree (Preorder)
4. Traverse the tree (Postorder)
5. Display the elements of the tree
6. Exit
Enter your choice: 1
Enter value to insert: 12
12 inserted into the tree.

1. Insert into binary search tree
2. Traverse the tree (Inorder)
3. Traverse the tree (Preorder)
4. Traverse the tree (Postorder)
5. Display the elements of the tree
6. Exit
Enter your choice: 5
Elements of the tree: 12 23 45 65

1. Insert into binary search tree
2. Traverse the tree (Inorder)
3. Traverse the tree (Preorder)
4. Traverse the tree (Postorder)
5. Display the elements of the tree
6. Exit
Enter your choice: 2
Inorder traversal: 12 23 45 65
```

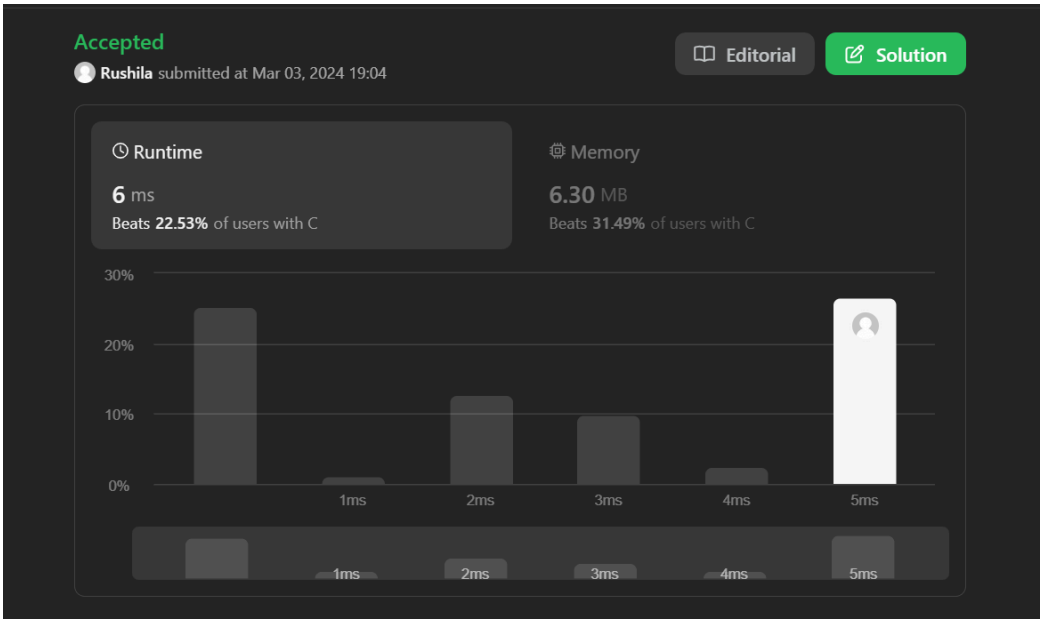
```
1. Insert into binary search tree
2. Traverse the tree (Inorder)
3. Traverse the tree (Preorder)
4. Traverse the tree (Postorder)
5. Display the elements of the tree
6. Exit
Enter your choice: 3
Preorder traversal: 23 12 45 65

1. Insert into binary search tree
2. Traverse the tree (Inorder)
3. Traverse the tree (Preorder)
4. Traverse the tree (Postorder)
5. Display the elements of the tree
6. Exit
Enter your choice: 4
Postorder traversal: 12 65 45 23
```

Lab Program-9(a):

Write a program to traverse a graph using BFS method.

```
#include <stdio.h>
#define MAX_VERTICES 10

int n, i, j, visited[MAX_VERTICES], queue[MAX_VERTICES], front = 0, rear = 0;
int adj[MAX_VERTICES][MAX_VERTICES];

void bfs(int v) {
   visited[v] = 1;
   queue[rear++] = v;

   while (front < rear) {
      int current = queue[front++];
      printf("%d\t", current);

      for (int i = 0; i < n; i++) {
         if (adj[current][i] && !visited[i]) {
            visited[i] = 1;
            queue[rear++] = i;
         }
```

```
        }
    }
}

int main() {
    int v;
    printf("Enter the number of vertices: ");
    scanf("%d", &n);

    for (i = 0; i < n; i++) {
        visited[i] = 0; // Initialize all nodes as unvisited
    }

    printf("Enter graph data in matrix form:\n");
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            scanf("%d", &adj[i][j]);

    printf("Enter the starting vertex: ");
    scanf("%d", &v);
    bfs(v);

    for (i = 0; i < n; i++) {
        if (!visited[i]) {
            printf("\nBFS is not possible. Not all nodes are reachable.\n");
            return 0;
        }
    }

    return 0;
}
```



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

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

#define MAX_VERTICES 10

void dfs(int graph[MAX_VERTICES][MAX_VERTICES], int num_vertices, bool
visited[MAX_VERTICES], int vertex) {
   visited[vertex] = true;
   int i;
   for (i = 0; i < num_vertices; ++i) {
      if (graph[vertex][i] == 1 && !visited[i]) {
         dfs(graph, num_vertices, visited, i);
      }
   }
}

bool is_connected(int graph[MAX_VERTICES][MAX_VERTICES], int num_vertices) {
   bool visited[MAX_VERTICES] = {false};

   // Perform DFS starting from vertex 0
   dfs(graph, num_vertices, visited, 0);
   int i;

   // Check if all vertices were visited
   for (i = 0; i < num_vertices; ++i) {
      if (!visited[i]) {
         return false;
      }
   }
   return true;
}

int main() {
   int num_vertices;
   printf("Enter the number of vertices: ");
   scanf("%d", &num_vertices);
   int i, j;
   int graph[MAX_VERTICES][MAX_VERTICES];
   printf("Enter the adjacency matrix:\n");
   for (i = 0; i < num_vertices; ++i) {
      for (j = 0; j < num_vertices; ++j) {
         scanf("%d", &graph[i][j]);
      }
   }

   if (is_connected(graph, num_vertices)) {
```

```
        printf("The graph is connected.\n");
    } else {
        printf("The graph is not connected.\n");
    }

    return 0;
}
```

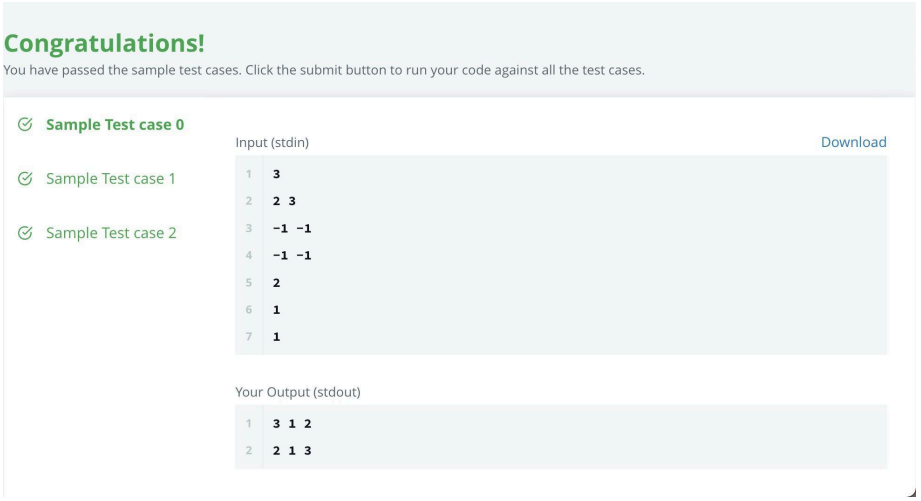
```
Enter the number of vertices: 7
Enter the adjacency matrix:
0 1 0 1 0 0 0
1 0 1 1 0 1 1
0 1 0 1 1 1 0
1 1 1 0 1 0 0
0 0 1 1 0 0 1
0 1 1 0 0 0 0
0 1 0 0 1 0 0
The graph is connected.
```

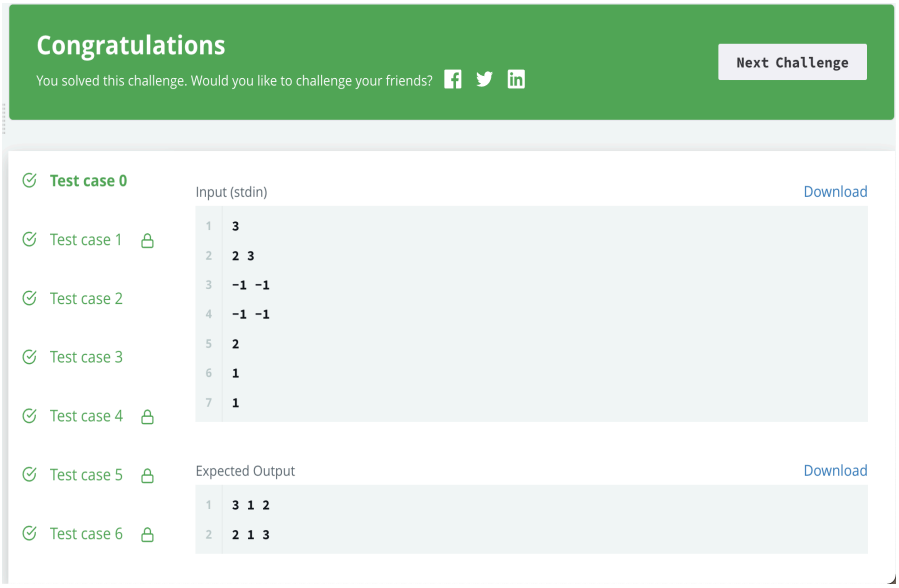
Hackerrank-Swapping of trees

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

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

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

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

```
    inorder(tree->right);
}

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

    while(i < no_of_nodes )
    {
        tree[i].id = i+1;
        scanf("%d %d", &l, &r);
        if(l == -1)
            tree[i].left = NULL;
        else
        {
            tree[i].left = &tree[l-1];
            tree[i].left->depth = tree[i].depth + 1;
            max_depth = tree[i].left->depth;
        }

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

        i++;
    }

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

```
                tree[k].right = temp;
            }
        }
        l = l + r;
    }
    inorder(tree);
    printf("\n");
}

return 0;
}
```





Lab Program-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 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 // Maximum number of employees
#define HASH_TABLE_SIZE 10 // Size of the hash table

// Structure for employee record
struct Employee {
    int key; // 4-digit key
    // Other employee details can be added here
};

// Function prototypes
int hashFunction(int key);
void insertEmployee(struct Employee employees[], int hashTable[], struct Employee emp);
void displayHashTable(int hashTable[]);

int main() {
    struct Employee employees[MAX_EMPLOYEES]; // Array to hold employee records
    int hashTable[HASH_TABLE_SIZE] = {0}; // Hash table initialized with 0
    int n, i;

    // Input the number of employees
    printf("Enter the number of employees: ");
    scanf("%d", &n);
```

```
    // Input employee records
    printf("Enter employee records:\n");
    for (i = 0; i < n; ++i) {
        printf("Employee %d:\n", i + 1);
        printf("Enter 4-digit key: ");
        scanf("%d", &employees[i].key);
        // Additional details can be input here
        insertEmployee(employees, hashTable, employees[i]);
    }

    // Display the hash table
    printf("\nHash Table:\n");
    displayHashTable(hashTable);

    return 0;
}

// Hash function: H(K) = K mod m
int hashFunction(int key) {
    return key % HASH_TABLE_SIZE;
}

// Function to insert an employee into the hash table
void insertEmployee(struct Employee employees[], int hashTable[], struct Employee emp) {
    int index = hashFunction(emp.key);

    // Linear probing to resolve collisions
    while (hashTable[index] != 0) {
        index = (index + 1) % HASH_TABLE_SIZE;
    }

    // Insert the employee key into the hash table
    hashTable[index] = emp.key;
}

// Function to display the hash table
void displayHashTable(int hashTable[]) {
    int i;

    for (i = 0; i < HASH_TABLE_SIZE; ++i) {
        printf("%d -> ", i);
        if (hashTable[i] == 0) {
            printf("Empty\n");
        } else {
            printf("%d\n", hashTable[i]);
        }
    }
}
```

Output:

```
Enter the number of employees: 5
Enter employee records:
Employee 1:
Enter 4-digit key: 1234
Employee 2:
Enter 4-digit key: 3456
Employee 3:
Enter 4-digit key: 9087
Employee 4:
Enter 4-digit key: 4567
Employee 5:
Enter 4-digit key: 1009

Hash Table:
0 -> Empty
1 -> Empty
2 -> Empty
3 -> Empty
4 -> 1234
5 -> Empty
6 -> 3456
7 -> 9087
8 -> 4567
9 -> 1009
```