

VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.



LAB REPORT On

DATA STRUCTURES (23CS3PCDST)

Submitted by

SACHIN KUMAR E

(1BM24CS419)

**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 **SACHINKUMAR E(1BM23CS419)**, 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.

Prof. Lakshmi Neelima M
Assistant 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	STACK PROGRAM	3
2	INFIX TO POSTFIX	7
3	LINEAR AND CIRCULAR QUEUE	9
4	SINGLE LINK LIST (CONCATENTION, REVERSE, SORT)	12
5	SINGLE LINK LIST (QUEUE & STACK)	16
6	DOUBLE LINK LIST	29
7	BINARY SEARCH TREE	31
8	BREADTH FIRST SEARCH	37
9	DEPTH FIRST SEARCH	41
10	HASHING	45
11	LEETCODE PROBLEMS	51

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>
```

```

#define SIZE 5

int stack[SIZE], top=-1;

void push(int st);
int pop();
void display();

int main() {
    int choice, element;

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

        switch (choice) {
            case 1:
                printf("Enter the element to push: ");
                scanf("%d", &element);
                push(element);
                break;
            case 2:
                element = pop();
                if (element != -1) {
                    printf("Popped %d from stack.\n", element);
                }
                break;
            case 3:
                display();
                break;
            case 4:
                printf("Exit");
                exit(0);
            default:
                printf("Invalid choice! Please try again.\n");
        }
    }

    return 0;
}

void push(int element) {
    if (top == SIZE - 1) {
        printf("Stack Overflow\n");
        return;
    }
}

```

```

        stack[++top] = element;
    }
    int pop() {
        if (top == -1) {
            printf("Stack Underflow\n");
            return -1; // Return -1 to indicate underflow
        }
        return stack[top--];
    }
    void display() {
        if (top == -1) {
            printf("Stack is empty\n");
            return;
        }
        printf("Stack elements are: ");
        for (int i = 0; i <= top; i++) {
            printf("%d ", stack[i]);
        }
    }
}

```

Output:

```
"C:\Users\Admin\Desktop\417\test.c\stack program.exe"
1. Push
2. Pop
3. Display
4. Exit
1
Enter your element: 1
Element 1 is pushed
1. Push
2. Pop
3. Display
4. Exit
1
Enter your element: 2
Element 2 is pushed
1. Push
2. Pop
3. Display
4. Exit
1
Enter your element: 3
Element 3 is pushed
1. Push
2. Pop
3. Display
4. Exit
1
Enter your element: 4
Element 4 is pushed
1. Push
2. Pop
3. Display
4. Exit
1
Enter your element: 5
Element 5 is pushed
1. Push
2. Pop
3. Display
4. Exit
1
Enter your element: 6
Stack overflow
1. Push
2. Pop
3. Display
4. Exit
3
5 4 3 2 1
1. Push
2. Pop
3. Display
4. Exit
2
Popped from stack: 5
1. Push
2. Pop
3. Display
4. Exit
2
Popped from stack: 4
1. Push
2. Pop
3. Display
4. Exit
2
```

```

4. Exit
1
Enter your element: 4
Element 4 is pushed
1. Push
2. Pop
3. Display
4. Exit
1
Enter your element: 5
Element 5 is pushed
1. Push
2. Pop
3. Display
4. Exit
1
Enter your element: 6
Stack overflow
1. Push
2. Pop
3. Display
4. Exit
3
5 4 3 2 1
1. Push
2. Pop
3. Display
4. Exit
2
Popped from stack: 5
1. Push
2. Pop
3. Display
4. Exit
2
Popped from stack: 4
1. Push
2. Pop
3. Display
4. Exit
2
Popped from stack: 3
1. Push
2. Pop
3. Display
4. Exit
2
Popped from stack: 2
1. Push
2. Pop
3. Display
4. Exit
2
Popped from stack: 1
1. Push
2. Pop
3. Display
4. Exit
2
Stack underflow
1. Push
2. Pop
3. Display
4. Exit
3

```

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<string.h>
#include<ctype.h>
#define size 100
int precedence(char op)
{
    if(op == '+' || op == '-')
        return 1;
    if(op == '*' || op == '/')
        return 2;
    return 0;
}
void infix_to_postfix(char*expression)
{
    char stack[size];
    int top = -1;
    char postfixexpr[size];

```

```

int k = 0;
for(int i=0;i<strlen(expression);i++)
{
    char ch=expression[i];
    if(isalnum(ch))
    {
        postfixexpr[k++]=ch;
    }
    else if(ch == '(')
    {
        stack[++top]=ch;
    }
    else if(ch == ')')
    {
        while(top!= -1 && stack[top]!='(')
        {
            postfixexpr[k++]=stack[top--];
        }
        top--;
    }
    else
    {
        while(top != -1 && precedence(stack[top])>=precedence(ch))
        {
            postfixexpr[k++]=stack[top--];
        }
        stack[++top]=ch;
    }
}

while(top != -1)
{
    postfixexpr[k++]=stack[top--];
}
postfixexpr[k]='\0';
printf("postfix expression:%s",postfixexpr);

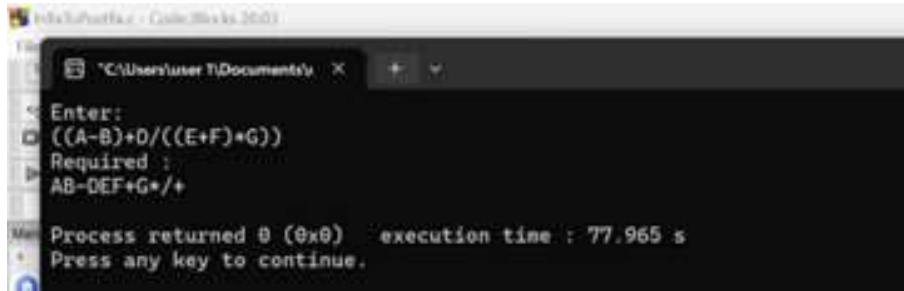
}

int main()
{
    char expression[size];
    printf("enter an infix expression:");
    scanf("%s",&expression);
    infix_to_postfix(expression);
    return 0;

}

```


Output:



```
Enter:
((A-B)+D/((E+F)*G))
Required :
AB-DEF+G*/+

Process returned 0 (0x0)   execution time : 77.965 s
Press any key to continue.
```

3) 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.

Program:

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

#define SIZE 5

int queue[SIZE];

int front = -1, rear = -1;

void enqueue(int value);
```

```

int dequeue();
void display();
int main(){
int value, option;
while(1){
printf("Enter 1 to INSERT, 2 to DELETE, 3 to DISPLAY, 4 to EXIT: \n");
scanf("%d",&option);
switch(option){
case 1:
printf("Enter value to be inserted: \n");
scanf("%d",&value);
enqueue(value);
break;
case 2:
value = dequeue();
printf("Value deleted is: %d\n",value);
break;
case 3:
display();
break;
case 4:
exit(1);
break;
default:
printf("Invalid Input.\n");
}
}
}

void enqueue(int value){

```

```

if(rear == (SIZE-1)){
printf("Queue OVERFLOW\n");
} else if (front== -1 && rear== -1){
front++;
rear++;
queue[rear] = value;
} else {
rear++;
queue[rear] = value;
}
}

int dequeue(){
int value;

if((front== -1 && rear== -1)|| front>rear){
printf("Queue UNDERFLOW\n");
} else{
value = queue[front];
front++;
return value;
}
}

void display(){
if(front== -1){
printf("Queue is EMPTY.\n");
} else{
for(int i = front; i<rear; i++){
printf("Queue elements are: \n");
printf("%d",queue[i]);
printf("\n");
}
}
}

```

}

}

}

Output :

```
Enter 1 to INSERT, 2 to DELETE, 3 to DISPLAY, 4 to EXIT:
1
Enter value to be inserted:
12
Enter 1 to INSERT, 2 to DELETE, 3 to DISPLAY, 4 to EXIT:
2
Value deleted is: 12
Enter 1 to INSERT, 2 to DELETE, 3 to DISPLAY, 4 to EXIT:
2
Queue UNDERFLOW
Value deleted is: 0
Enter 1 to INSERT, 2 to DELETE, 3 to DISPLAY, 4 to EXIT:
1
Enter value to be inserted:
12
Enter 1 to INSERT, 2 to DELETE, 3 to DISPLAY, 4 to EXIT:
1
Enter value to be inserted:
2
Enter 1 to INSERT, 2 to DELETE, 3 to DISPLAY, 4 to EXIT:
1
Enter value to be inserted:
34
Enter 1 to INSERT, 2 to DELETE, 3 to DISPLAY, 4 to EXIT:
1
Enter value to be inserted:
45
```

```

Enter 1 to INSERT, 2 to DELETE, 3 to DISPLAY, 4 to EXIT:
1
Enter value to be inserted:
56
Queue OVERFLOW
Enter 1 to INSERT, 2 to DELETE, 3 to DISPLAY, 4 to EXIT:
3
Queue elements are:
12
2
34
Enter 1 to INSERT, 2 to DELETE, 3 to DISPLAY, 4 to EXIT:
4

Process returned 1 (0x1)   execution time : 48.149 s
Press any key to continue.

```

4) 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.

Program:

```

#include<stdio.h>
#include<string.h>
#include<conio.h>
#define SIZE 5
int queue[SIZE];
int front = -1, rear = -1;
void enqueue(int value);
int dequeue();
void display();
int main(){
int value, option;
while(1){
printf("Enter 1 to INSERT, 2 to DELETE, 3 to DISPLAY, 4 to EXIT: \n");

```

```

scanf("%d",&option);
switch(option){
case 1:
printf("Enter value to be inserted: \n");
scanf("%d",&value);
enqueue(value);
break;
case 2:
value = dequeue();
printf("Value deleted is: %d\n",value);
break;
case 3:
display();
break;
case 4:
exit(1);
break;
default:
printf("Invalid Input.\n");
}
}
}

void enqueue(int value){
if(front==(rear+1)%SIZE){
printf("Queue OVERFLOW\n");
} else if (front== -1 && rear== -1){
front++;
rear++;
queue[rear] = value;

```

```

    }else {
rear = (rear+1)%SIZE;
queue[rear] = value;
    }
}

int dequeue(){
int value;
if((front== -1 && rear== -1)|| front>rear){
printf("Queue UNDERFLOW\n");
} else if(front==rear){
value = queue[front];
front=rear=-1;
} else{
value = queue[front];
front = (front+1)%SIZE;
}
return value;
}

void display(){
int i;
if(front== -1){
printf("Queue is EMPTY.\n");
} else{
printf("Queue elements are: \n");
for(i=front;i!=rear;i=((i+1)%SIZE))
{
printf("%d\n", queue[i]);
}
printf("%d\n",queue[i]);
}

```

```
}  
  
}
```

OUTPUT:

```
Enter 1 to INSERT, 2 to DELETE, 3 to DISPLAY, 4 to EXIT:  
1  
Enter value to be inserted:  
12  
Enter 1 to INSERT, 2 to DELETE, 3 to DISPLAY, 4 to EXIT:  
1  
Enter value to be inserted:  
23  
Enter 1 to INSERT, 2 to DELETE, 3 to DISPLAY, 4 to EXIT:  
3  
Queue elements are:  
12  
23  
Enter 1 to INSERT, 2 to DELETE, 3 to DISPLAY, 4 to EXIT:  
2  
Value deleted is: 12  
Enter 1 to INSERT, 2 to DELETE, 3 to DISPLAY, 4 to EXIT:  
2  
Value deleted is: 23  
Enter 1 to INSERT, 2 to DELETE, 3 to DISPLAY, 4 to EXIT:  
2  
Queue UNDERFLOW  
Value deleted is: 4214942  
Enter 1 to INSERT, 2 to DELETE, 3 to DISPLAY, 4 to EXIT:  
1  
Enter value to be inserted:  
12
```

```
Enter 1 to INSERT, 2 to DELETE, 3 to DISPLAY, 4 to EXIT:  
1  
Enter value to be inserted:  
23  
Enter 1 to INSERT, 2 to DELETE, 3 to DISPLAY, 4 to EXIT:  
1  
Enter value to be inserted:  
34  
Enter 1 to INSERT, 2 to DELETE, 3 to DISPLAY, 4 to EXIT:  
1  
Enter value to be inserted:  
45  
Enter 1 to INSERT, 2 to DELETE, 3 to DISPLAY, 4 to EXIT:  
1  
Enter value to be inserted:  
56  
Enter 1 to INSERT, 2 to DELETE, 3 to DISPLAY, 4 to EXIT:  
1  
Enter value to be inserted:  
67  
Queue OVERFLOW  
Enter 1 to INSERT, 2 to DELETE, 3 to DISPLAY, 4 to EXIT:  
4  
  
Process returned 1 (0x1)   execution time : 76.081 s  
Press any key to continue.
```


5) WAP to Implement Singly Linked List with following operations

- a) Create a linked list.
- b) Deletion of first element, specified element and last element in the list.
- c) Display the contents of the linked list.

Program:

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

struct node{
    int data;
    struct node *next;
};
struct node *head = NULL;
struct node* createnode(int data) {
    struct node* newnode = (struct node*)malloc(sizeof(struct node));
    newnode->data = data;
    newnode->next = NULL;
    return newnode;
}
void insert_at_beg(int data) {
    struct node* newnode = createnode(data);
    newnode->next = head;
    head = newnode;
}

void insert_at_end(int data) {
    struct node* newnode = createnode(data);
    if (head == NULL) {
        head = newnode;
        return;
    }
    struct node* temp = head;
    while (temp->next != NULL) {
        temp = temp->next;
    }
    temp->next = newnode;
}

void insert_at_pos(int data, int pos) {
    struct node* newnode = createnode(data);
    if (pos == 1) {
        newnode->next = head;
        head = newnode;
        return;
    }
    struct node* temp = head;
```

```

        for (int i = 0; i < pos - 2 && temp != NULL; i++) {
            temp = temp->next;
        }
        if (temp == NULL) {
            printf("Invalid position\n");
            free(newnode);
            return;
        }
        newnode->next = temp->next;
        temp->next = newnode;
    }

```

```

void delete_at_beg() {
    if (head == NULL) {
        printf("List is empty\n");
        return;
    }
    struct node* temp = head;
    head = head->next;
    free(temp);
}

```

```

void delete_at_end() {
    if (head == NULL) {
        printf("List is empty\n");
        return;
    }
    struct node* temp = head;
    if (temp->next == NULL) {
        head = NULL;
        free(temp);
        return;
    }
    while (temp->next->next != NULL) {
        temp = temp->next;
    }
    free(temp->next);
    temp->next = NULL;
}

```

```

void delete_at_pos(int pos) {
    if (head == NULL) {
        printf("List is empty\n");
        return;
    }
    if (pos == 1) {
        struct node* temp = head;
        head = head->next;
        free(temp);
        return;
    }

```

```

    }
    struct node* temp = head;
    for (int i = 0; i < pos - 2 && temp != NULL; i++) {
        temp = temp->next;
    }
    if (temp == NULL || temp->next == NULL) {
        printf("Position not valid\n");
        return;
    }
    struct node* deletenode = temp->next;
    temp->next = deletenode->next;
    free(deletenode);
}

void display() {
    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");
}

int main() {
    int option, data, position;

    while (1) {
        printf("\n1. Insert at beginning\n2. Insert at end\n3. Insert at position\n4. Delete at
beginning\n5. Delete at end\n6. Delete at position\n7. Display\n8. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &option);

        switch (option) {
            case 1:
                printf("Enter data to insert at beginning: ");
                scanf("%d", &data);
                insert_at_beg(data);
                break;
            case 2:
                printf("Enter data to insert at end: ");
                scanf("%d", &data);
                insert_at_end(data);
                break;
            case 3:
                printf("Enter data and position to insert: ");
                scanf("%d %d", &data, &position);

```

```

        insert_at_pos(data, position);
        break;

    case 4:
        delete_at_beg();
        break;

    case 5:
        delete_at_end();
        break;

    case 6:
        printf("Enter position to delete: ");
        scanf("%d", &position);
        delete_at_pos(position);
        break;

    case 7:
        display();
        break;

    case 8:
        return 0; // Exit gracefully

    default:
        printf("Invalid choice\n");
    }
}

return 0;
}

```

OUTPUT:

```

1. Insert at beginning
2. Insert at end
3. Insert at position
4. Delete at beginning
5. Delete at end
6. Delete at position
7. Display
8. Exit
Enter your choice: 1
Enter data to insert at beginning: 1

1. Insert at beginning
2. Insert at end
3. Insert at position
4. Delete at beginning
5. Delete at end
6. Delete at position
7. Display
8. Exit
Enter your choice: 2
Enter data to insert at end: 2

1. Insert at beginning
2. Insert at end
3. Insert at position
4. Delete at beginning
5. Delete at end
6. Delete at position
7. Display
8. Exit
Enter your choice: 3
Enter data and position to insert: 3
2

```

```
1. Insert at beginning
2. Insert at end
3. Insert at position
4. Delete at beginning
5. Delete at end
6. Delete at position
7. Display
8. Exit
```

Enter your choice: 7

1 -> 3 -> 2 -> NULL

```
1. Insert at beginning
2. Insert at end
3. Insert at position
4. Delete at beginning
5. Delete at end
6. Delete at position
7. Display
8. Exit
```

Enter your choice: 4

```
1. Insert at beginning
2. Insert at end
3. Insert at position
4. Delete at beginning
5. Delete at end
6. Delete at position
7. Display
8. Exit
```

Enter your choice: 7

3 -> 2 -> NULL

```
1. Insert at beginning
2. Insert at end
3. Insert at position
4. Delete at beginning
5. Delete at end
6. Delete at position
7. Display
8. Exit
```

Enter your choice: 5

```
1. Insert at beginning
2. Insert at end
3. Insert at position
4. Delete at beginning
5. Delete at end
6. Delete at position
7. Display
8. Exit
```

Enter your choice: 7

3 -> NULL

```
1. Insert at beginning
2. Insert at end
3. Insert at position
4. Delete at beginning
5. Delete at end
6. Delete at position
7. Display
8. Exit
```

Enter your choice: 1

Enter data to insert at beginning: 1

```
1. Insert at beginning
2. Insert at end
3. Insert at position
4. Delete at beginning
5. Delete at end
6. Delete at position
7. Display
8. Exit
Enter your choice: 6
Enter position to delete: 1
```

```
1. Insert at beginning
2. Insert at end
3. Insert at position
4. Delete at beginning
5. Delete at end
6. Delete at position
7. Display
8. Exit
Enter your choice: 7
3 -> NULL
```

```
1. Insert at beginning
2. Insert at end
3. Insert at position
4. Delete at beginning
5. Delete at end
6. Delete at position
7. Display
8. Exit
Enter your choice: 4
```

```
1. Insert at beginning
2. Insert at end
3. Insert at position
4. Delete at beginning
5. Delete at end
6. Delete at position
7. Display
8. Exit
Enter your choice: 7
List is empty
```

```
1. Insert at beginning
2. Insert at end
3. Insert at position
4. Delete at beginning
5. Delete at end
6. Delete at position
7. Display
8. Exit
Enter your choice: 4
List is empty
```

6a) WAP to Implement Single Link List with following operations: Sort the linked list, Reverse the linked list, Concatenation of two linked list.
6b) WAP to Implement Single Link List to simulate Stack & Queue Operations.

Program:

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
struct node {
    int data;
    struct node *next;
};

struct node *head = NULL;

struct node* concatenate(struct node *t1, struct node *t2);
struct node* reverse(struct node *s);
void sort(struct node *s);
void pop();
void push();
void enqueue();
void dequeue();
struct node* createnode();
void display(struct node *head);

struct node* concatenate(struct node *t1, struct node *t2) {
    if (t1 == NULL) return t2;
    if (t2 == NULL) return t1;

    struct node *ptr = t1;
    while (ptr->next != NULL) {
        ptr = ptr->next;
    }
    ptr->next = t2;
    return t1;
}

struct node* reverse(struct node *s) {
    struct node *prev = NULL, *current = s, *next = NULL;
    while (current != NULL) {
        next = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }
    return prev;
}

void sort(struct node *s) {
    if (s == NULL) {
        printf("List is empty, nothing to sort.\n");
    }
}
```



```

        return;
    }

    struct node *i, *j;
    int temp;
    for (i = s; i != NULL; i = i->next) {
        for (j = i->next; j != NULL; j = j->next) {
            if (i->data > j->data) {
                temp = i->data;
                i->data = j->data;
                j->data = temp;
            }
        }
    }
}

void push() {
    struct node *ptr = createnode();
    ptr->next = head;
    head = ptr;
}

void pop() {
    if (head == NULL) {
        printf("Stack is empty\n");
        return;
    }

    struct node *ptr = head;
    head = head->next;
    printf("Element %d removed\n", ptr->data);
    free(ptr);
}

void enqueue() {
    struct node *ptr = createnode();
    if (head == NULL) {
        head = ptr;
    } else {
        struct node *temp = head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = ptr;
    }
}

void dequeue() {
    if (head == NULL) {
        printf("Queue is empty\n");
        return;
    }
}

```

```

        struct node *ptr = head;
        head = head->next;
        printf("Deleted element: %d\n", ptr->data);
        free(ptr);
    }
    struct node* createnode() {
        struct node *newnode = (struct node *)malloc(sizeof(struct node));
        if (newnode == NULL) {
            printf("Memory allocation failed\n");
            exit(1);
        }

        printf("Enter value to be inserted:\n");
        scanf("%d", &newnode->data);
        newnode->next = NULL;
        return newnode;
    }
    void display(struct node *head) {
        if (head == NULL) {
            printf("List is empty\n");
            return;
        }

        struct node *ptr = head;
        while (ptr != NULL) {
            printf("%d -> ", ptr->data);
            ptr = ptr->next;
        }
        printf("NULL\n");
    }
    int main() {
        int choice;
        struct node *ptr1 = NULL, *ptr2 = NULL;

        while (1) {
            printf("\nEnter your choice:\n");
            printf("1. Sort\n2. Concatenate\n3. Reverse\n4. Push into stack\n5. Pop out of stack\n6. Enqueue\n7. Dequeue\n8. Display\n9. Exit\n");
            scanf("%d", &choice);

            switch (choice) {
                case 1:
                    if (head == NULL) {
                        printf("Create a list first.\n");
                        head = createnode();
                    }
                    sort(head);
                    printf("Sorted list: ");
                    display(head);

```

```

        break;

case 2:
    printf("Create first list:\n");
    ptr1 = createnode();
    printf("Create second list:\n");
    ptr2 = createnode();
    head = concatenate(ptr1, ptr2);
    printf("Concatenated list: ");
    display(head);
    break;

case 3:
    head = reverse(head);
    printf("Reversed list: ");
    display(head);
    break;

case 4:
    push();
    break;

case 5:
    pop();
    break;

case 6:
    enqueue();
    break;

case 7:
    dequeue();
    break;

case 8:
    display(head);
    break;

case 9:
    exit(0);

default:
    printf("Invalid choice\n");
}
}

return 0;
}
OUTPUT:

```

```
"C:\Users\user 1\Documents\k  X + v
Enter your choice:
1. Sort
2. Concatenate
3. Reverse
4. Push (Stack)
5. Pop (Stack)
6. Enqueue (Queue)
7. Dequeue (Queue)
8. Display
9. Exit
1
Create a list first.
Enter value to be inserted:
2
Sorted list: 2 -> NULL
```

```
Enter your choice:
1. Sort
2. Concatenate
3. Reverse
4. Push (Stack)
5. Pop (Stack)
6. Enqueue (Queue)
7. Dequeue (Queue)
8. Display
9. Exit
2
Create first list:
Enter value to be inserted:
1
Create second list:
Enter value to be inserted:
2
Concatenated list: 1 -> 2 -> NULL

Enter your choice:
1. Sort
2. Concatenate
3. Reverse
4. Push (Stack)
5. Pop (Stack)
6. Enqueue (Queue)
7. Dequeue (Queue)
8. Display
9. Exit
8
```

```
1 -> 2 -> NULL

Enter your choice:
1. Sort
2. Concatenate
3. Reverse
4. Push (Stack)
5. Pop (Stack)
6. Enqueue (Queue)
7. Dequeue (Queue)
8. Display
9. Exit
4
Enter value to be inserted:
4

Enter your choice:
1. Sort
2. Concatenate
3. Reverse
4. Push (Stack)
5. Pop (Stack)
6. Enqueue (Queue)
7. Dequeue (Queue)
8. Display
9. Exit
4
Enter value to be inserted:
34
```

```
Enter your choice:
1. Sort
2. Concatenate
3. Reverse
4. Push (Stack)
5. Pop (Stack)
6. Enqueue (Queue)
7. Dequeue (Queue)
8. Display
9. Exit
8
34 -> 4 -> 1 -> 2 -> NULL

Enter your choice:
1. Sort
2. Concatenate
3. Reverse
4. Push (Stack)
5. Pop (Stack)
6. Enqueue (Queue)
7. Dequeue (Queue)
8. Display
9. Exit
6
Enter value to be inserted:
12
```

```
Enter your choice:
1. Sort
2. Concatenate
3. Reverse
4. Push (Stack)
5. Pop (Stack)
6. Enqueue (Queue)
7. Dequeue (Queue)
8. Display
9. Exit
8
34 -> 4 -> 1 -> 2 -> 12 -> NULL
```

```
Enter your choice:
1. Sort
2. Concatenate
3. Reverse
4. Push (Stack)
5. Pop (Stack)
6. Enqueue (Queue)
7. Dequeue (Queue)
8. Display
9. Exit
7
Deleted element: 34
```

```
Enter your choice:
1. Sort
2. Concatenate
3. Reverse
4. Push (Stack)
5. Pop (Stack)
6. Enqueue (Queue)
7. Dequeue (Queue)
8. Display
9. Exit
8
4 -> 1 -> 2 -> 12 -> NULL
```

```
Enter your choice:
1. Sort
2. Concatenate
3. Reverse
4. Push (Stack)
5. Pop (Stack)
6. Enqueue (Queue)
7. Dequeue (Queue)
8. Display
9. Exit
7
Deleted element: 4
```

```
Enter your choice:
1. Sort
2. Concatenate
3. Reverse
4. Push (Stack)
5. Pop (Stack)
6. Enqueue (Queue)
7. Dequeue (Queue)
8. Display
9. Exit
7
Deleted element: 1
```

```
Enter your choice:
1. Sort
2. Concatenate
3. Reverse
4. Push (Stack)
5. Pop (Stack)
6. Enqueue (Queue)
7. Dequeue (Queue)
8. Display
9. Exit
5
Element 2 removed
```

```
Enter your choice:
1. Sort
2. Concatenate
3. Reverse
4. Push (Stack)
5. Pop (Stack)
6. Enqueue (Queue)
7. Dequeue (Queue)
8. Display
9. Exit
8
12 -> NULL
```

```
Enter your choice:
1. Sort
2. Concatenate
3. Reverse
4. Push (Stack)
5. Pop (Stack)
6. Enqueue (Queue)
7. Dequeue (Queue)
8. Display
9. Exit
5
Element 12 removed
```

7) WAP to Implement doubly link list with primitive operations

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

d) Display the contents of the list

Program:

```
#include<stdio.h>

#include<stdlib.h>

struct node{

    struct node *prev;

    int data;

    struct node *next;

};

struct node *head = NULL;

void display();

struct node *createnode(){

    struct node *newnode = (struct node*)malloc(sizeof(struct node));

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

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

    newnode->prev = NULL;

    newnode->next = NULL;

    return newnode;

}

int main(){

    int option;

    while(1){

        printf("ENTER \n1. INSERT \n2. DELETE \n3. DISPLAY \n4. EXIT\n");

        scanf("%d",&option);

        switch(option){

            case 1:
```



```

        insert();

        break;

case 2:

        delete();

        break;

case 3:

        display();

        break;

case 4:

        exit(0);

        break;

default:

        printf("INVALID ENTRY");

        }

        }

}

void insert(){

    struct node *newnode = createnode(), *ptr = head;

    int val;

    printf("Enter the value before which insertion has to be done: ");

    scanf("%d",&val);

    if(head == NULL){

        newnode->next = NULL;

        newnode->prev = NULL;

        head = newnode;

    }else if(head->data == val){

        newnode->next = head;

```

```

newnode->prev = NULL;

head->prev = newnode;

head = newnode;

}else {

while(ptr->data!=val)

ptr= ptr->next;

if (ptr == NULL) {

printf("Value not found for insertion.\n");

free(newnode);

}

newnode->next = ptr;

newnode->prev = ptr->prev;

ptr->prev->next = newnode;

ptr->prev = newnode;

}

}

int delete() {

int value;

printf("Enter value to delete: ");

scanf("%d", &value);

struct node *ptr = head;

if (head == NULL) {

printf("List is empty\n");

return;

}

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

ptr = ptr->next;

```

```

    }

    if (ptr == NULL) {
printf("Node with value %d not found\n", value);

return;

    }

    if (ptr == head) {
head = ptr->next;

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

    } else if (ptr->next == NULL) {

ptr->prev->next = NULL;

    } else {

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

    }

free(ptr);

return 0;

}

void display(){

    struct node *ptr = head;

    if(head==NULL){

        printf("LIST EMPTY\n");

    }else{

        while(ptr!=NULL){

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

            ptr = ptr->next;

        }

    }

}

```

```

    }

}

}

```

OUTPUT:

```

C:\Users\user1\Documents\>
ENTER
1. INSERT
2. DELETE
3. DISPLAY
4. EXIT
2
Enter value to delete: 2
List is empty
ENTER
1. INSERT
2. DELETE
3. DISPLAY
4. EXIT
3
LIST EMPTY
ENTER
1. INSERT
2. DELETE
3. DISPLAY
4. EXIT
1
Enter value to be inserted: 12
Enter the value before which insertion has to be done: 1
ENTER
1. INSERT
2. DELETE
3. DISPLAY
4. EXIT
3
12 - ENTER

```

```

1. INSERT
2. DELETE
3. DISPLAY
4. EXIT
1
Enter value to be inserted: 23
Enter the value before which insertion has to be done: 12
ENTER
1. INSERT
2. DELETE
3. DISPLAY
4. EXIT
1
Enter value to be inserted: 45
Enter the value before which insertion has to be done: 12
ENTER
1. INSERT
2. DELETE
3. DISPLAY
4. EXIT
3
23 - 45 - 12 - ENTER
1. INSERT
2. DELETE
3. DISPLAY
4. EXIT
2
Enter value to delete: 45

```

```

ENTER
1. INSERT
2. DELETE
3. DISPLAY
4. EXIT
3
23 - 12 - ENTER
1. INSERT
2. DELETE
3. DISPLAY
4. EXIT
2
Enter value to delete: 23
ENTER
1. INSERT
2. DELETE
3. DISPLAY
4. EXIT
3
12 - ENTER
1. INSERT
2. DELETE
3. DISPLAY
4. EXIT
2
Enter value to delete: 12

```

```

ENTER
1. INSERT
2. DELETE
3. DISPLAY
4. EXIT
3
LIST EMPTY
ENTER
1. INSERT
2. DELETE
3. DISPLAY
4. EXIT
2
Enter value to delete: 1
List is empty
ENTER
1. INSERT
2. DELETE
3. DISPLAY
4. EXIT
4

Process returned 0 (0x0)   execution time : 58.061 s
Press any key to continue.

```

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.

Program:

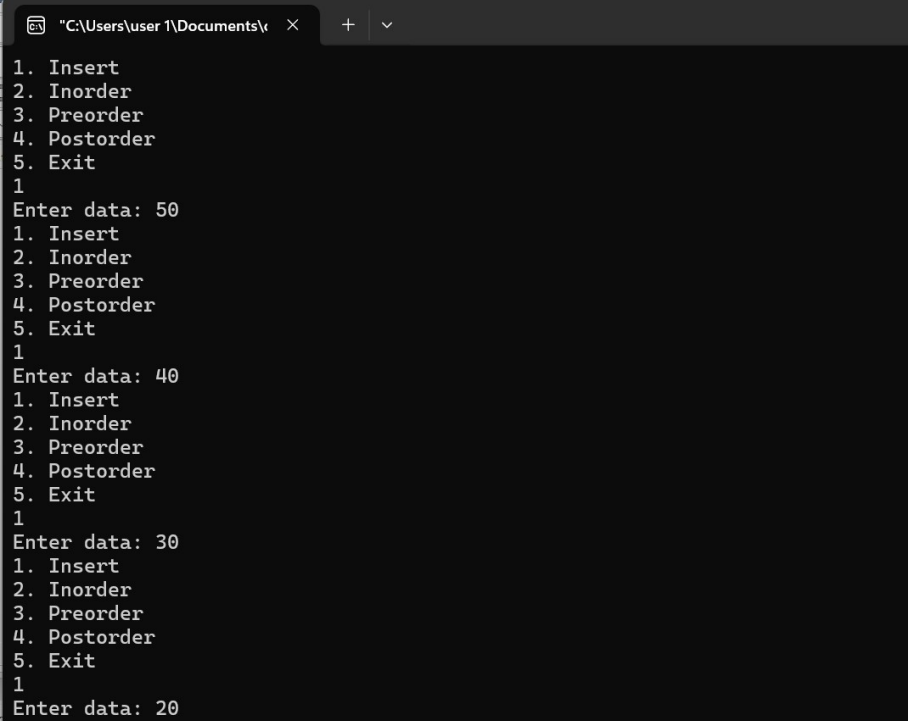
```
#include<stdio.h>
#include<stdlib.h>
struct tree{
    int data;
    struct tree *left;
    struct tree *right;
};
struct tree *createnode(int data){
    struct tree *newnode=(struct tree*)malloc(sizeof(struct tree));
    newnode->data=data;
    newnode->left = NULL;
    newnode->right = NULL;
    return newnode;
}
struct tree *insert(struct tree *root, int data){
    if(root == NULL){
        root = createnode(data);
    }
    if(data< root->data){
        root->left = insert(root->left, data);
    }else if(data> root->data){
        root->right = insert(root->right, data);
    }
    return root;
}
void inorder(struct tree *root){
    if(root != NULL){
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right);
    }
}
void preorder(struct tree *root){
    if(root != NULL){
        printf("%d ", root->data);
        preorder(root->left);
        preorder(root->right);
    }
}
void postorder(struct tree *root){
    if(root != NULL){
        postorder(root->left);
        postorder(root->right);
        printf("%d ", root->data);
    }
}
int main() {
    int option, data;
    struct tree *root = NULL;
    while(1){
        printf("1. Insert\n2. Inorder\n3. Preorder\n4. Postorder\n5. Exit \n");
        scanf("%d", &option);
        switch(option){
```

```

        case 1:
            printf("Enter data: ");
            scanf("%d", &data);
            root = insert(root, data);
            break;
        case 2:
            printf("Inorder traversal : ");
            inorder(root);
            printf("/n");
            break;
        case 3:
            printf("Preorder traversal : ");
            preorder(root);
            printf("/n");
            break;
        case 4:
            printf("Postorder traversal : ");
            postorder(root);
            printf("/n");
            break;
        case 5:
            exit(0);
            break;
        default:
            printf("Invalid option\n");
    }
}
}
}

```

OUTPUT:



```

C:\Users\user 1\Documents\
1. Insert
2. Inorder
3. Preorder
4. Postorder
5. Exit
1
Enter data: 50
1. Insert
2. Inorder
3. Preorder
4. Postorder
5. Exit
1
Enter data: 40
1. Insert
2. Inorder
3. Preorder
4. Postorder
5. Exit
1
Enter data: 30
1. Insert
2. Inorder
3. Preorder
4. Postorder
5. Exit
1
Enter data: 20

```

```
"C:\Users\user 1\Documents\  × + v
1. Insert
2. Inorder
3. Preorder
4. Postorder
5. Exit
1
Enter data: 50
1. Insert
2. Inorder
3. Preorder
4. Postorder
5. Exit
1
Enter data: 40
1. Insert
2. Inorder
3. Preorder
4. Postorder
5. Exit
1
Enter data: 30
1. Insert
2. Inorder
3. Preorder
4. Postorder
5. Exit
1
Enter data: 20
```

```
1. Insert
2. Inorder
3. Preorder
4. Postorder
5. Exit
1
Enter data: 70
1. Insert
2. Inorder
3. Preorder
4. Postorder
5. Exit
1
Enter data: 60
1. Insert
2. Inorder
3. Preorder
4. Postorder
5. Exit
1
Enter data: 75
1. Insert
2. Inorder
3. Preorder
4. Postorder
5. Exit
2
Inorder traversal : 20 30 40 50 60 70 75 /n1. Insert
```



```

2. Inorder
3. Preorder
4. Postorder
5. Exit
3
Preorder traversal : 50 40 30 20 70 60 75 /n1. Insert
2. Inorder
3. Preorder
4. Postorder
5. Exit
4
Postorder traversal : 20 30 40 60 75 70 50 /n1. Insert
2. Inorder
3. Preorder
4. Postorder
5. Exit
5

Process returned 0 (0x0)   execution time : 58.401 s
Press any key to continue.

```

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

Program:

```

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

#define MAX 100

struct Queue {
    int visited[MAX];
    int front, rear;
};

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

int isEmpty(struct Queue* q) {
    return q->front == -1;
}

void enqueue(struct Queue* q, int value) {
    if (q->rear == MAX - 1) {
        printf("Queue is full\n");
    } else {
        if (q->front == -1) {

```

```

        q->front = 0;
    }
    q->visited[++q->rear] = value;
}

}

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

void bfs(int graph[MAX][MAX], int startVertex, int n) {
    int visited[MAX] = {0};
    struct Queue* q = createQueue();

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

    printf("BFS Traversal: ");

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

        for (int i = 1; i <= n; i++) {
            if (graph[currentVertex][i] == 1 && !visited[i]) {
                visited[i] = 1;
                enqueue(q, i);
            }
        }
    }

    printf("\n");
}

int main() {
    int n, startVertex;
    int graph[MAX][MAX];

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

    printf("Enter the adjacency matrix:\n");
    for (int i = 1; i <= n; i++) {

```

```

        for (int j = 1; j <= n; j++) {
            printf("Vertex %d, %d : ", i, j);
            scanf("%d", &graph[i][j]);
        }
    }

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

    bfs(graph, startVertex, n);

    return 0;
}

```

OUTPUT:

```

"C:\Users\user 1\Documents\"
Enter the number of vertices : 4
Enter the adjacency matrix:
Vertex 1, 1 : 0
Vertex 1, 2 : 1
Vertex 1, 3 : 1
Vertex 1, 4 : 0
Vertex 2, 1 : 0
Vertex 2, 2 : 0
Vertex 2, 3 : 1
Vertex 2, 4 : 1
Vertex 3, 1 : 0
Vertex 3, 2 : 0
Vertex 3, 3 : 0
Vertex 3, 4 : 0
Vertex 4, 1 : 0
Vertex 4, 2 : 1
Vertex 4, 3 : 0
Vertex 4, 4 : 0
Enter the starting vertex: 4
BFS Traversal: 4 2 3

```

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

Program:

```

#include <stdio.h>

#define MAX 10

int adj[MAX][MAX]; // Adjacency matrix
int visited[MAX];  // Visited array

```

```

void dfs(int start, int n) {
    printf("%d ", start);
    visited[start] = 1;

    for (int i = 0; i < n; i++) {
        if (adj[start][i] && !visited[i]) {
            dfs(i, n);
        }
    }
}

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

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

    for (int i = 0; i < edges; i++) {
        printf("Enter edge (u v): ");
        scanf("%d %d", &u, &v);
        adj[u][v] = adj[v][u] = 1; // Undirected graph
    }

    printf("Enter starting vertex: ");
    scanf("%d", &start);

    printf("DFS Traversal: ");

```

```

    dfs(start, n);

    printf("\n");

    return 0;
}

```

OUTPUT:

```

Enter number of vertices: 4
Enter number of edges: 4
Enter edge (u v): 0 1
Enter edge (u v): 0 2
Enter edge (u v): 1 3
Enter edge (u v): 3 2
Enter starting vertex: 2
DFS Traversal: 2 0 1 3

=== Code Execution Successful ===

```

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.

Program:

```

#include <stdio.h>

#include <stdlib.h>

```

```

#define TABLE_SIZE 10

```

```

#define EMPTY -1

```

```

struct HashMap {

```

```

    int key;

    int value;
};

void initTable(struct HashMap table[]) {
    for (int i = 0; i < TABLE_SIZE; i++) {
        table[i].key = EMPTY; // Mark all slots as empty
        table[i].value = 0;
    }
}

int hash(int key) {
    return key % TABLE_SIZE;
}

void insert(struct HashMap table[], int key, int value) {

    int index = hash(key);

    int originalIndex = index;

    while (table[index].key != EMPTY) {
        index = (index + 1) % TABLE_SIZE;
        if (index == originalIndex) { // Table is full
            printf("Hash table is full. Cannot insert key %d.\n", key);
            return;
        }
    }

    table[index].key = key;
    table[index].value = value;

    printf("Inserted (key: %d, value: %d) at index %d\n", key, value, index);
}

```

```
}
```

```
void search(struct HashMap table[], int key) {
```

```
    int index = hash(key);
```

```
    int originalIndex = index;
```

```
    while (table[index].key != EMPTY) {
```

```
        if (table[index].key == key) {
```

```
            printf("Key %d found with value %d at index %d\n", key, table[index].value, index);
```

```
            return;
```

```
        }
```

```
        index = (index + 1) % TABLE_SIZE;
```

```
        if (index == originalIndex) { // Full cycle completed
```

```
            break;
```

```
        }
```

```
    }
```

```
    printf("Key %d not found in the table.\n", key);
```

```
}
```

```
void display(struct HashMap table[]) {
```

```
    printf("\nHash Table:\n");
```

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

```
        if (table[i].key != EMPTY)
```

```
            printf("Index %d -> Key: %d, Value: %d\n", i, table[i].key, table[i].value);
```

```
        else
```

```
            printf("Index %d -> Empty\n", i);
```

```
    }
```

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

```
}
```

```

int main() {

    struct HashMap table[TABLE_SIZE];

    int choice, key, value;

    initTable(table);

    while (1) {

        printf("1. Insert\n2. Search\n3. Display\n4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {

            case 1:

                printf("Enter key and value to insert: ");
                scanf("%d %d", &key, &value);
                insert(table, key, value);
                break;

            case 2:

                printf("Enter key to search: ");
                scanf("%d", &key);
                search(table, key);
                break;

            case 3:

                display(table);
                break;

            case 4:

                printf("Exiting...\n");
                return 0;
        }
    }
}

```



```

        default:

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

        }

    }

    return 0;
}

```

OUTPUT:

```

1. Insert
2. Search
3. Display
4. Exit
Enter your choice: 1
Enter key and value to insert: 1
2
Inserted (key: 1, value: 2) at index 1
1. Insert
2. Search
3. Display
4. Exit
Enter your choice: 1
Enter key and value to insert: 2
3
Inserted (key: 2, value: 3) at index 2
1. Insert
2. Search
3. Display
4. Exit
Enter your choice: 1
Enter key and value to insert: 2
5

```

```
Inserted (key: 2, value: 5) at index 3
```

1. Insert
2. Search
3. Display
4. Exit

Enter your choice: 1

Enter key and value to insert: 3

8

```
Inserted (key: 3, value: 8) at index 4
```

1. Insert
2. Search
3. Display
4. Exit

Enter your choice: 1

Enter key and value to insert: 4

6

```
Inserted (key: 4, value: 6) at index 5
```

1. Insert
2. Search
3. Display
4. Exit

Enter your choice: 2

Enter key to search: 6

Enter key to search: 6

Key 6 not found in the table.

1. Insert
2. Search
3. Display
4. Exit

Enter your choice: 2

Enter key to search: 4

Key 4 found with value 6 at index 5

1. Insert
2. Search
3. Display
4. Exit

Enter your choice: 3

Hash Table:

Index 0 -> Empty

Index 1 -> Key: 1, Value: 2

Index 2 -> Key: 2, Value: 3

Index 3 -> Key: 2, Value: 5

Index 4 -> Key: 3, Value: 8

Index 5 -> Key: 4, Value: 6

Index 6 -> Empty

Index 7 -> Empty

Index 8 -> Empty

Index 9 -> Empty

LEETCODE

Majority Element

Given an array nums of size n, return the majority element.

The majority element is the element that appears more than $\lfloor n / 2 \rfloor$ times. You may assume that the majority element always exists in the array.

```
class Solution {
public:
    int majorityElement(vector<int>& nums) {
        int count = 0;
        int var = 0;

        for (int num : nums) {
            if (count == 0) {
                var = num;
            }

            if (num == var) {
                count++;
            } else {
                count--;
            }
        }

        return var;
    }
};
```

2)Palindrome Linked List

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

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     struct ListNode *next;
 * };
 */
bool isPalindrome(struct ListNode* head) {
    // Check for empty list or single node
    if (head == NULL || head->next == NULL) {
        return 1;
    }

    struct ListNode* reverse(struct ListNode* previous, struct ListNode* node) {
```

```

    struct ListNode* temp;
    if (node->next == NULL) {
        node->next = previous;
        return node;
    }
    temp = node->next;
    node->next = previous;
    return reverse(node, temp);
}

struct ListNode* copy = head;
int size = 1;
while (copy->next != NULL) {
    size++;
    copy = copy->next;
}
size = (size / 2) + (size % 2);
copy = head;
while (size > 0) {
    size--;
    copy = copy->next;
}
if (copy->next != NULL) {
    copy = reverse(NULL, copy);
}
while (copy != NULL) {
    if (copy->val != head->val) {
        return 0;
    }
    copy = copy->next;
    head = head->next;
}
return 1;
}

```

3)GAME OF TWO STACKS(HACKER RUN)

Alexa has two stacks of non-negative integers, stack $a[n]$ and $b[m]$ stack where index denotes the top of the stack. Alexa challenges Nick to play the following game:

In each move, Nick can remove one integer from the top of either stack 'a' or 'b' stack . Nick keeps a running sum of the integers he removes from the two stacks.

Nick is disqualified from the game if, at any point, his running sum becomes greater than some integer 'maxsum' given at the beginning of the game.

Nick's final score is the total number of integers he has removed from the two stacks. Given 'a', 'b', and 'maxsum' for 'g' games, find the maximum possible score Nick can achieve.

```

#include <assert.h>
#include <ctype.h>
#include <limits.h>
#include <math.h>
#include <stdbool.h>
#include <stddef.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char* readline();
char* ltrim(char*);
char* rtrim(char*);
char** split_string(char*);

int parse_int(char*);

/*
 * Complete the 'twoStacks' function below.
 *
 * The function is expected to return an INTEGER.
 * The function accepts following parameters:
 * 1. INTEGER maxSum
 * 2. INTEGER_ARRAY a
 * 3. INTEGER_ARRAY b
 */
int twoStacks(int maxSum, int a_count, int* a, int b_count, int* b) {
    int sum = 0;
    int a_index = 0;
    int b_index = 0;
    int result = 0;
    while (a_index < a_count) {
        if (sum + a[a_index] > maxSum) {
            break;
        } else {
            sum += a[a_index];
            a_index++;
        }
    }
    result = a_index;

    while (b_index < b_count) {
        sum += b[b_index];
        b_index++;
        while (sum > maxSum && a_index > 0) {
            a_index--;
            sum -= a[a_index];
        }
        if ((a_index + b_index > result) && (sum <= maxSum)) {

```

```

        result = a_index + b_index;
    }
}
return result;
}

int main()
{
    FILE* fptr = fopen(getenv("OUTPUT_PATH"), "w");

    int g = parse_int(ltrim(rtrim(readline())));

    for (int g_itr = 0; g_itr < g; g_itr++) {
        char** first_multiple_input = split_string(rtrim(readline()));

        int n = parse_int(*(first_multiple_input + 0));

        int m = parse_int(*(first_multiple_input + 1));

        int maxSum = parse_int(*(first_multiple_input + 2));

        char** a_temp = split_string(rtrim(readline()));

        int* a = malloc(n * sizeof(int));

        for (int i = 0; i < n; i++) {
            int a_item = parse_int(*(a_temp + i));

            *(a + i) = a_item;
        }

        char** b_temp = split_string(rtrim(readline()));

        int* b = malloc(m * sizeof(int));

        for (int i = 0; i < m; i++) {
            int b_item = parse_int(*(b_temp + i));

            *(b + i) = b_item;
        }

        int result = twoStacks(maxSum, n, a, m, b);

        fprintf(fptr, "%d\n", result);
    }

    fclose(fptr);
}

```

```

    return 0;
}

char* readline() {
    size_t alloc_length = 1024;
    size_t data_length = 0;

    char* data = malloc(alloc_length);

    while (true) {
        char* cursor = data + data_length;
        char* line = fgets(cursor, alloc_length - data_length, stdin);

        if (!line) {
            break;
        }

        data_length += strlen(cursor);

        if (data_length < alloc_length - 1 || data[data_length - 1] == '\n') {
            break;
        }

        alloc_length <= 1;

        data = realloc(data, alloc_length);

        if (!data) {
            data = '\0';

            break;
        }
    }

    if (data[data_length - 1] == '\n') {
        data[data_length - 1] = '\0';

        data = realloc(data, data_length);

        if (!data) {
            data = '\0';
        }
    } else {
        data = realloc(data, data_length + 1);

        if (!data) {
            data = '\0';
        } else {
            data[data_length] = '\0';
        }
    }
}

```

```

    }
}

return data;
}

char* ltrim(char* str) {
    if (!str) {
        return '\0';
    }

    if (!*str) {
        return str;
    }

    while (*str != '\0' && isspace(*str)) {
        str++;
    }

    return str;
}

char* rtrim(char* str) {
    if (!str) {
        return '\0';
    }

    if (!*str) {
        return str;
    }

    char* end = str + strlen(str) - 1;

    while (end >= str && isspace(*end)) {
        end--;
    }

    *(end + 1) = '\0';

    return str;
}

char** split_string(char* str) {
    char** splits = NULL;
    char* token = strtok(str, " ");

    int spaces = 0;

    while (token) {
        splits = realloc(splits, sizeof(char*) * ++spaces);
    }
}

```



```

        if (!splits) {
            return splits;
        }

        splits[spaces - 1] = token;

        token = strtok(NULL, " ");
    }

    return splits;
}

int parse_int(char* str) {
    char* endptr;
    int value = strtol(str, &endptr, 10);

    if (endptr == str || *endptr != '\0') {
        exit(EXIT_FAILURE);
    }

    return value;
}

```

4) PATH SUM

Given the root of a binary tree and an integer targetSum, return true if the tree has a root-to-leaf path such that adding up all the values along the path equals targetSum.

```

/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     struct TreeNode *left;
 *     struct TreeNode *right;
 * };
 */
bool hasPathSum(struct TreeNode* root, int targetSum) {
    if (!root) return false;
    if (!(root->left || root->right)) return (targetSum == root->val);

    return (hasPathSum(root->left, targetSum - root->val) ||
            hasPathSum(root->right, targetSum - root->val));
}

```