1. **Write a C/C++ program to implement addition and subtraction of rational numbers using Abstract Data Type(ADT).**

#include <stdio.h>

typedef struct {

int numerator;

int denominator;

} Rational;

// Function to find the greatest common divisor (GCD)

int gcd(int a, int b) {

if (b == 0)

return a;

return gcd(b, a % b);

}

// Function to simplify the rational number

void simplify(Rational \*num) {

int common\_divisor = gcd(num->numerator, num->denominator);

num->numerator /= common\_divisor;

num->denominator /= common\_divisor;

}

// Function to add two rational numbers

Rational add(const Rational \*num1, const Rational \*num2) {

Rational result;

result.numerator = (num1->numerator \* num2->denominator) + (num2->numerator \* num1->denominator);

result.denominator = num1->denominator \* num2->denominator;

simplify(&result);

return result;

}

// Function to subtract two rational numbers

Rational subtract(const Rational \*num1, const Rational \*num2) {

Rational result;

result.numerator = (num1->numerator \* num2->denominator) - (num2->numerator \* num1->denominator);

result.denominator = num1->denominator \* num2->denominator;

simplify(&result);

return result;

}

// Function to display the rational number

void display(const Rational \*num) {

printf("%d/%d\n", num->numerator, num->denominator);

}

int main() {

// Example usage

Rational num1 = {3, 4};

Rational num2 = {1, 2};

printf("Number 1: ");

display(&num1);

printf("Number 2: ");

display(&num2);

// Addition

printf("Addition: ");

display(&add(&num1, &num2));

// Subtraction

printf("Subtraction: ");

display(&subtract(&num1, &num2));

return 0;

}

1. **Write a C/C++ program to implement addition and subtraction of complex numbers using Abstract Data Type(ADT).**

#include <stdio.h>

typedef struct {

float real;

float imaginary;

} Complex;

Complex add(const Complex\* num1, const Complex\* num2) {

Complex result;

result.real = num1->real + num2->real;

result.imaginary = num1->imaginary + num2->imaginary;

return result;

}

Complex subtract(const Complex\* num1, const Complex\* num2) {

Complex result;

result.real = num1->real - num2->real;

result.imaginary = num1->imaginary - num2->imaginary;

return result;

}

void display(const Complex\* num) {

printf("%.2f + %.2fi\n", num->real, num->imaginary);

}

int main() {

Complex num1 = {2.5, 3.0};

Complex num2 = {1.5, 2.0};

printf("Number 1: ");

display(&num1);

printf("Number 2: ");

display(&num2);

// Addition

printf("Addition: ");

display(&add(&num1, &num2));

// Subtraction

printf("Subtraction: ");

display(&subtract(&num1, &num2));

return 0;

}

**3.Write a C/C++ program to implement multiplication and equality check of rational numbers using Abstract Data Type(ADT).**

#include <stdio.h>

typedef struct {

int numerator;

int denominator;

} Rational;

int gcd(int a, int b) {

if (b == 0)

return a;

return gcd(b, a % b);

}

void simplify(Rational\* num) {

int common\_divisor = gcd(num->numerator, num->denominator);

num->numerator /= common\_divisor;

num->denominator /= common\_divisor;

}

Rational multiply(const Rational\* num1, const Rational\* num2) {

Rational result;

result.numerator = num1->numerator \* num2->numerator;

result.denominator = num1->denominator \* num2->denominator;

simplify(&result);

return result;

}

int isEqual(const Rational\* num1, const Rational\* num2) {

return (num1->numerator == num2->numerator) && (num1->denominator == num2->denominator);

}

void display(const Rational\* num) {

printf("%d/%d\n", num->numerator, num->denominator);

}

int main() {

Rational num1 = {3, 4};

Rational num2 = {2, 3};

printf("Number 1: ");

display(&num1);

printf("Number 2: ");

display(&num2);

// Multiplication

printf("Multiplication: ");

display(&multiply(&num1, &num2));

// Equality check

if (isEqual(&num1, &num2)) {

printf("Numbers are equal.\n");

} else {

printf("Numbers are not equal.\n");

}

return 0;

}

**4. Write a C/C++ program to implement multiplication and equality check of complex numbers using Abstract Data Type(ADT).**

#include <stdio.h>

typedef struct {

float real;

float imaginary;

} Complex;

Complex multiply(const Complex\* num1, const Complex\* num2) {

Complex result;

result.real = (num1->real \* num2->real) - (num1->imaginary \* num2->imaginary);

result.imaginary = (num1->real \* num2->imaginary) + (num1->imaginary \* num2->real);

return result;

}

int isEqual(const Complex\* num1, const Complex\* num2) {

return (num1->real == num2->real) && (num1->imaginary == num2->imaginary);

}

void display(const Complex\* num) {

printf("%.2f + %.2fi\n", num->real, num->imaginary);

}

int main() {

Complex num1 = {2.5, 3.0};

Complex num2 = {1.5, 2.0};

printf("Number 1: ");

display(&num1);

printf("Number 2: ");

display(&num2);

// Multiplication

printf("Multiplication: ");

display(&multiply(&num1, &num2));

// Equality check

if (isEqual(&num1, &num2)) {

printf("Numbers are equal.\n");

} else {

printf("Numbers are not equal.\n");

}

return 0;

}

**5. Write a C/C++ program to insert and delete a number from a given location in an array.**

#include <stdio.h>

void displayArray(const int arr[], int size) {

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

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

}

printf("\n");

}

void insertElement(int arr[], int\* size, int element, int position) {

if (position < 0 || position > \*size) {

printf("Invalid position for insertion.\n");

return;

}

// Shift elements to make space for the new element

for (int i = \*size - 1; i >= position; --i) {

arr[i + 1] = arr[i];

}

// Insert the new element

arr[position] = element;

// Increase the size of the array

++\*size;

}

void deleteElement(int arr[], int\* size, int position) {

if (position < 0 || position >= \*size) {

printf("Invalid position for deletion.\n");

return;

}

// Shift elements to fill the gap left by the deleted element

for (int i = position; i < \*size - 1; ++i) {

arr[i] = arr[i + 1];

}

// Decrease the size of the array

--\*size;

}

int main() {

const int maxSize = 100;

int arr[maxSize];

int size = 5; // Initial size of the array

// Initialize the array

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

arr[i] = i + 1;

}

printf("Original Array: ");

displayArray(arr, size);

// Inserting an element at position 2

insertElement(arr, &size, 10, 2);

printf("After Insertion: ");

displayArray(arr, size);

// Deleting an element at position 4

deleteElement(arr, &size, 4);

printf("After Deletion: ");

displayArray(arr, size);

return 0;

}

**6. Write a C/C++ program to create a singly linked list.**

#include <stdio.h>

#include <stdlib.h>

// Define a structure for a node

struct Node {

int data;

struct Node\* next;

};

// Function to add a new node at the end of the list

void append(struct Node\*\* headRef, int value) {

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

newNode->data = value;

newNode->next = NULL;

if (\*headRef == NULL) {

\*headRef = newNode;

} else {

struct Node\* current = \*headRef;

while (current->next != NULL) {

current = current->next;

}

current->next = newNode;

}

}

// Function to display the linked list

void display(struct Node\* head) {

struct Node\* current = head;

while (current != NULL) {

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

current = current->next;

}

printf("\n");

}

int main() {

struct Node\* myList = NULL;

// Appending nodes to the linked list

append(&myList, 1);

append(&myList, 2);

append(&myList, 3);

append(&myList, 4);

// Displaying the linked list

printf("Linked List: ");

display(myList);

return 0;

}

**7. Write a C/C++ program to create a circular linked list.**

#include <stdio.h>

#include <stdlib.h>

// Define a structure for a node

struct Node {

int data;

struct Node\* next;

};

// Function to add a new node at the end of the list

void append(struct Node\*\* headRef, int value) {

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

newNode->data = value;

if (\*headRef == NULL) {

\*headRef = newNode;

newNode->next = newNode; // Make it circular

} else {

struct Node\* current = \*headRef;

while (current->next != \*headRef) {

current = current->next;

}

current->next = newNode;

newNode->next = \*headRef; // Make it circular

}

}

// Function to display the circular linked list

void display(struct Node\* head) {

if (head == NULL) {

printf("Circular Linked List is empty.\n");

return;

}

struct Node\* current = head;

do {

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

current = current->next;

} while (current != head);

printf("\n");

}

int main() {

struct Node\* myCircularList = NULL;

// Appending nodes to the circular linked list

append(&myCircularList, 1);

append(&myCircularList, 2);

append(&myCircularList, 3);

append(&myCircularList, 4);

// Displaying the circular linked list

printf("Circular Linked List: ");

display(myCircularList);

return 0;

}

**8. Write a C/C++ program to create a doubly linked list.**

#include <stdio.h>

#include <stdlib.h>

// Define a structure for a node

struct Node {

int data;

struct Node\* next;

struct Node\* prev;

};

// Function to add a new node at the end of the list

void append(struct Node\*\* headRef, int value) {

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

newNode->data = value;

newNode->next = NULL;

if (\*headRef == NULL) {

\*headRef = newNode;

newNode->prev = NULL;

} else {

struct Node\* current = \*headRef;

while (current->next != NULL) {

current = current->next;

}

current->next = newNode;

newNode->prev = current;

}

}

// Function to display the doubly linked list in forward direction

void displayForward(struct Node\* head) {

struct Node\* current = head;

while (current != NULL) {

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

current = current->next;

}

printf("\n");

}

// Function to display the doubly linked list in reverse direction

void displayReverse(struct Node\* head) {

struct Node\* current = head;

while (current->next != NULL) {

current = current->next;

}

while (current != NULL) {

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

current = current->prev;

}

printf("\n");

}

int main() {

struct Node\* myDoublyList = NULL;

// Appending nodes to the doubly linked list

append(&myDoublyList, 1);

append(&myDoublyList, 2);

append(&myDoublyList, 3);

append(&myDoublyList, 4);

// Displaying the doubly linked list in forward direction

printf("Doubly Linked List (Forward): ");

displayForward(myDoublyList);

// Displaying the doubly linked list in reverse direction

printf("Doubly Linked List (Reverse): ");

displayReverse(myDoublyList);

return 0;

}

**9. Write a C/C++ program to create a circular doubly linked list.**

#include <stdio.h>

#include <stdlib.h>

// Define a structure for a node

struct Node {

int data;

struct Node\* next;

struct Node\* prev;

};

// Function to add a new node at the end of the list

void append(struct Node\*\* headRef, int value) {

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

newNode->data = value;

if (\*headRef == NULL) {

\*headRef = newNode;

newNode->prev = newNode;

newNode->next = newNode;

} else {

struct Node\* tail = (\*headRef)->prev; // Get the last node

tail->next = newNode;

newNode->prev = tail;

newNode->next = \*headRef;

(\*headRef)->prev = newNode;

}

}

// Function to display the circular doubly linked list in forward direction

void displayForward(struct Node\* head) {

if (head == NULL) {

printf("Circular Doubly Linked List is empty.\n");

return;

}

struct Node\* current = head;

do {

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

current = current->next;

} while (current != head);

printf("\n");

}

// Function to display the circular doubly linked list in reverse direction

void displayReverse(struct Node\* head) {

if (head == NULL) {

printf("Circular Doubly Linked List is empty.\n");

return;

}

struct Node\* current = head->prev; // Get the last node

do {

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

current = current->prev;

} while (current != head->prev);

printf("\n");

}

int main() {

struct Node\* myCircularDoublyList = NULL;

// Appending nodes to the circular doubly linked list

append(&myCircularDoublyList, 1);

append(&myCircularDoublyList, 2);

append(&myCircularDoublyList, 3);

append(&myCircularDoublyList, 4);

// Displaying the circular doubly linked list in forward direction

printf("Circular Doubly Linked List (Forward): ");

displayForward(myCircularDoublyList);

// Displaying the circular doubly linked list in reverse direction

printf("Circular Doubly Linked List (Reverse): ");

displayReverse(myCircularDoublyList);

return 0;

}

**10. Write a C/C++ program to count the number of nodes in the singly linked list**

#include <stdio.h>

#include <stdlib.h>

// Define a structure for a node

struct Node {

int data;

struct Node\* next;

};

// Function to add a new node at the end of the list

void append(struct Node\*\* headRef, int value) {

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

newNode->data = value;

newNode->next = NULL;

if (\*headRef == NULL) {

\*headRef = newNode;

} else {

struct Node\* current = \*headRef;

while (current->next != NULL) {

current = current->next;

}

current->next = newNode;

}

}

// Function to count the number of nodes in the linked list

int countNodes(struct Node\* head) {

int count = 0;

struct Node\* current = head;

while (current != NULL) {

++count;

current = current->next;

}

return count;

}

int main() {

struct Node\* myList = NULL;

// Appending nodes to the linked list

append(&myList, 1);

append(&myList, 2);

append(&myList, 3);

append(&myList, 4);

// Counting the number of nodes in the linked list

int nodeCount = countNodes(myList);

// Displaying the linked list and the node count

printf("Linked List: ");

struct Node\* current = myList;

while (current != NULL) {

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

current = current->next;

}

printf("\n");

printf("Number of Nodes: %d\n", nodeCount);

return 0;

}

**11. Write a C/C++ program to delete element at the begining/middle/end of singly linkedlist**

#include <stdio.h>

#include <stdlib.h>

// Node structure for the linked list

struct Node {

int data;

struct Node\* next;

};

// Function to create a new node with the given value

struct Node\* createNode(int value) {

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

newNode->data = value;

newNode->next = NULL;

return newNode;

}

// Function to insert a new node at the end of the list

void insert(struct Node\*\* head, int value) {

struct Node\* newNode = createNode(value);

if (\*head == NULL) {

\*head = newNode;

} else {

struct Node\* current = \*head;

while (current->next != NULL) {

current = current->next;

}

current->next = newNode;

}

}

// Function to display the linked list

void display(struct Node\* head) {

struct Node\* current = head;

while (current != NULL) {

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

current = current->next;

}

printf("\n");

}

// Function to delete the first node in the list

void deleteAtBeginning(struct Node\*\* head) {

if (\*head != NULL) {

struct Node\* temp = \*head;

\*head = (\*head)->next;

free(temp);

} else {

printf("List is empty. Unable to delete from the beginning.\n");

}

}

// Function to delete a node at a specific position in the list

void deleteAtPosition(struct Node\*\* head, int position) {

if (position < 0) {

printf("Invalid position. Position should be non-negative.\n");

return;

}

if (\*head == NULL) {

printf("List is empty. Unable to delete.\n");

return;

}

if (position == 0) {

deleteAtBeginning(head);

return;

}

struct Node\* current = \*head;

struct Node\* previous = NULL;

int currentPosition = 0;

while (current != NULL && currentPosition < position) {

previous = current;

current = current->next;

currentPosition++;

}

if (current == NULL) {

printf("Invalid position. Position exceeds the length of the list.\n");

return;

}

previous->next = current->next;

free(current);

}

// Function to delete the last node in the list

void deleteAtEnd(struct Node\*\* head) {

if (\*head == NULL) {

printf("List is empty. Unable to delete from the end.\n");

return;

}

if ((\*head)->next == NULL) {

deleteAtBeginning(head);

return;

}

struct Node\* current = \*head;

struct Node\* previous = NULL;

while (current->next != NULL) {

previous = current;

current = current->next;

}

previous->next = NULL;

free(current);

}

int main() {

struct Node\* myList = NULL;

insert(&myList, 1);

insert(&myList, 2);

insert(&myList, 3);

insert(&myList, 4);

printf("Original linked list: ");

display(myList);

deleteAtBeginning(&myList);

printf("Linked list after deleting the first element: ");

display(myList);

deleteAtPosition(&myList, 1);

printf("Linked list after deleting element at position 1: ");

display(myList);

deleteAtEnd(&myList);

printf("Linked list after deleting the last element: ");

display(myList);

return 0;

}

**12. Write a C/C++ program to insert element at the begining/middle/end of singly linkedlist**

#include <stdio.h>

#include <stdlib.h>

// Node structure for the linked list

struct Node {

int data;

struct Node\* next;

};

// Function to create a new node with the given value

struct Node\* createNode(int value) {

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

newNode->data = value;

newNode->next = NULL;

return newNode;

}

// Function to insert a new node at the beginning of the list

void insertAtBeginning(struct Node\*\* head, int value) {

struct Node\* newNode = createNode(value);

newNode->next = \*head;

\*head = newNode;

}

// Function to insert a new node at the end of the list

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

struct Node\* newNode = createNode(value);

if (\*head == NULL) {

\*head = newNode;

} else {

struct Node\* current = \*head;

while (current->next != NULL) {

current = current->next;

}

current->next = newNode;

}

}

// Function to insert a new node at a specific position in the list

void insertAtPosition(struct Node\*\* head, int value, int position) {

if (position < 0) {

printf("Invalid position. Position should be non-negative.\n");

return;

}

struct Node\* newNode = createNode(value);

if (position == 0) {

newNode->next = \*head;

\*head = newNode;

return;

}

struct Node\* current = \*head;

struct Node\* previous = NULL;

int currentPosition = 0;

while (current != NULL && currentPosition < position) {

previous = current;

current = current->next;

currentPosition++;

}

if (currentPosition < position) {

printf("Invalid position. Position exceeds the length of the list.\n");

free(newNode);

return;

}

previous->next = newNode;

newNode->next = current;

}

// Function to display the linked list

void display(struct Node\* head) {

struct Node\* current = head;

while (current != NULL) {

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

current = current->next;

}

printf("\n");

}

int main() {

struct Node\* myList = NULL;

insertAtEnd(&myList, 1);

insertAtEnd(&myList, 2);

insertAtEnd(&myList, 4);

printf("Original linked list: ");

display(myList);

insertAtBeginning(&myList, 0);

printf("Linked list after inserting 0 at the beginning: ");

display(myList);

insertAtPosition(&myList, 3, 2);

printf("Linked list after inserting 3 at position 2: ");

display(myList);

return 0;

}

**13. Write a C program to search a particular number in singly linked list circular linked list/circular doubly linked list/doubly linked list.**

#include <stdio.h>

#include <stdlib.h>

// Node structure for singly linked list

struct Node {

int data;

struct Node\* next;

};

// Node structure for doubly linked list

struct DoubleNode {

int data;

struct DoubleNode\* prev;

struct DoubleNode\* next;

};

// Function to create a new node for singly linked list

struct Node\* createNode(int value) {

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

newNode->data = value;

newNode->next = NULL;

return newNode;

}

// Function to create a new node for doubly linked list

struct DoubleNode\* createDoubleNode(int value) {

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

newNode->data = value;

newNode->prev = NULL;

newNode->next = NULL;

return newNode;

}

// Function to insert a new node at the end of singly linked list

void insertSingly(struct Node\*\* head, int value) {

struct Node\* newNode = createNode(value);

if (\*head == NULL) {

\*head = newNode;

} else {

struct Node\* current = \*head;

while (current->next != NULL) {

current = current->next;

}

current->next = newNode;

}

}

// Function to insert a new node at the end of doubly linked list

void insertDoubly(struct DoubleNode\*\* head, int value) {

struct DoubleNode\* newNode = createDoubleNode(value);

if (\*head == NULL) {

\*head = newNode;

} else {

struct DoubleNode\* current = \*head;

while (current->next != NULL) {

current = current->next;

}

current->next = newNode;

newNode->prev = current;

}

}

// Function to search for a number in singly linked list

int searchSingly(struct Node\* head, int key) {

struct Node\* current = head;

int position = 0;

while (current != NULL) {

if (current->data == key) {

return position;

}

current = current->next;

position++;

}

return -1; // Return -1 if the key is not found

}

// Function to search for a number in doubly linked list

int searchDoubly(struct DoubleNode\* head, int key) {

struct DoubleNode\* current = head;

int position = 0;

while (current != NULL) {

if (current->data == key) {

return position;

}

current = current->next;

position++;

}

return -1; // Return -1 if the key is not found

}

int main() {

struct Node\* singlyList = NULL;

struct DoubleNode\* doublyList = NULL;

// Insert elements into singly linked list

insertSingly(&singlyList, 1);

insertSingly(&singlyList, 2);

insertSingly(&singlyList, 3);

// Insert elements into doubly linked list

insertDoubly(&doublyList, 1);

insertDoubly(&doublyList, 2);

insertDoubly(&doublyList, 3);

int keyToSearch = 2;

// Search in singly linked list

int positionSingly = searchSingly(singlyList, keyToSearch);

if (positionSingly != -1) {

printf("%d found at position %d in singly linked list.\n", keyToSearch, positionSingly);

} else {

printf("%d not found in singly linked list.\n", keyToSearch);

}

// Search in doubly linked list

int positionDoubly = searchDoubly(doublyList, keyToSearch);

if (positionDoubly != -1) {

printf("%d found at position %d in doubly linked list.\n", keyToSearch, positionDoubly);

} else {

printf("%d not found in doubly linked list.\n", keyToSearch);

}

return 0;

}

**14. Write a C/C++ program to delete a particular node in linked list/circular linked list/circular doubly linked sort/doubly linked list.**

#include <stdio.h>

#include <stdlib.h>

// Node structure for singly linked list

struct Node {

int data;

struct Node\* next;

};

// Node structure for doubly linked list

struct DoubleNode {

int data;

struct DoubleNode\* prev;

struct DoubleNode\* next;

};

// Function to create a new node for singly linked list

struct Node\* createNode(int value) {

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

newNode->data = value;

newNode->next = NULL;

return newNode;

}

// Function to create a new node for doubly linked list

struct DoubleNode\* createDoubleNode(int value) {

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

newNode->data = value;

newNode->prev = NULL;

newNode->next = NULL;

return newNode;

}

// Function to insert a new node at the end of singly linked list

void insertSingly(struct Node\*\* head, int value) {

struct Node\* newNode = createNode(value);

if (\*head == NULL) {

\*head = newNode;

} else {

struct Node\* current = \*head;

while (current->next != NULL) {

current = current->next;

}

current->next = newNode;

}

}

// Function to insert a new node at the end of doubly linked list

void insertDoubly(struct DoubleNode\*\* head, int value) {

struct DoubleNode\* newNode = createDoubleNode(value);

if (\*head == NULL) {

\*head = newNode;

} else {

struct DoubleNode\* current = \*head;

while (current->next != NULL) {

current = current->next;

}

current->next = newNode;

newNode->prev = current;

}

}

// Function to delete a particular node in singly linked list

void deleteNodeSingly(struct Node\*\* head, int key) {

struct Node\* current = \*head;

struct Node\* previous = NULL;

// Handle the case where the key is in the first node

if (current != NULL && current->data == key) {

\*head = current->next;

free(current);

return;

}

// Search for the key in the rest of the list

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

previous = current;

current = current->next;

}

// If the key is not present in the list

if (current == NULL) {

printf("Node with key %d not found in singly linked list.\n", key);

return;

}

// Unlink the node from the list

previous->next = current->next;

free(current);

}

// Function to delete a particular node in doubly linked list

void deleteNodeDoubly(struct DoubleNode\*\* head, int key) {

struct DoubleNode\* current = \*head;

// Handle the case where the key is in the first node

if (current != NULL && current->data == key) {

\*head = current->next;

if (\*head != NULL) {

(\*head)->prev = NULL;

}

free(current);

return;

}

// Search for the key in the rest of the list

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

current = current->next;

}

// If the key is not present in the list

if (current == NULL) {

printf("Node with key %d not found in doubly linked list.\n", key);

return;

}

// Unlink the node from the list

if (current->prev != NULL) {

current->prev->next = current->next;

}

if (current->next != NULL) {

current->next->prev = current->prev;

}

free(current);

}

// Function to display the linked list

void displaySingly(struct Node\* head) {

struct Node\* current = head;

while (current != NULL) {

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

current = current->next;

}

printf("\n");

}

// Function to display the doubly linked list

void displayDoubly(struct DoubleNode\* head) {

struct DoubleNode\* current = head;

while (current != NULL) {

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

current = current->next;

}

printf("\n");

}

int main() {

struct Node\* singlyList = NULL;

struct DoubleNode\* doublyList = NULL;

// Insert elements into singly linked list

insertSingly(&singlyList, 1);

insertSingly(&singlyList, 2);

insertSingly(&singlyList, 3);

// Insert elements into doubly linked list

insertDoubly(&doublyList, 1);

insertDoubly(&doublyList, 2);

insertDoubly(&doublyList, 3);

int keyToDelete = 2;

// Delete node in singly linked list

printf("Original singly linked list: ");

displaySingly(singlyList);

deleteNodeSingly(&singlyList, keyToDelete);

printf("Singly linked list after deleting node with key %d: ", keyToDelete);

displaySingly(singlyList);

// Delete node in doubly linked list

printf("Original doubly linked list: ");

displayDoubly(doublyList);

deleteNodeDoubly(&doublyList, keyToDelete);

printf("Doubly linked list after deleting node with key %d: ", keyToDelete);

displayDoubly(doublyList);

return 0;

}

**15. Write a C/C++ program to count the number of nodes in a circular linked list/circular doubly linked list/doubly linked list.**

#include <stdio.h>

#include <stdlib.h>

// Node structure for singly linked list

struct Node {

int data;

struct Node\* next;

};

// Node structure for doubly linked list

struct DoubleNode {

int data;

struct DoubleNode\* prev;

struct DoubleNode\* next;

};

// Function to create a new node for singly linked list

struct Node\* createNode(int value) {

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

newNode->data = value;

newNode->next = NULL;

return newNode;

}

// Function to create a new node for doubly linked list

struct DoubleNode\* createDoubleNode(int value) {

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

newNode->data = value;

newNode->prev = NULL;

newNode->next = NULL;

return newNode;

}

// Function to insert a new node at the end of singly linked list

void insertSingly(struct Node\*\* head, int value) {

struct Node\* newNode = createNode(value);

if (\*head == NULL) {

\*head = newNode;

} else {

struct Node\* current = \*head;

while (current->next != NULL) {

current = current->next;

}

current->next = newNode;

}

}

// Function to insert a new node at the end of doubly linked list

void insertDoubly(struct DoubleNode\*\* head, int value) {

struct DoubleNode\* newNode = createDoubleNode(value);

if (\*head == NULL) {

\*head = newNode;

} else {

struct DoubleNode\* current = \*head;

while (current->next != NULL) {

current = current->next;

}

current->next = newNode;

newNode->prev = current;

}

}

// Function to count the number of nodes in singly linked list

int countNodesSingly(struct Node\* head) {

if (head == NULL) {

return 0;

}

int count = 0;

struct Node\* current = head;

do {

count++;

current = current->next;

} while (current != head);

return count;

}

// Function to count the number of nodes in doubly linked list

int countNodesDoubly(struct DoubleNode\* head) {

int count = 0;

struct DoubleNode\* current = head;

while (current != NULL) {

count++;

current = current->next;

}

return count;

}

// Function to display the linked list

void displaySingly(struct Node\* head) {

if (head == NULL) {

printf("Empty list\n");

return;

}

struct Node\* current = head;

do {

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

current = current->next;

} while (current != head);

printf("\n");

}

// Function to display the doubly linked list

void displayDoubly(struct DoubleNode\* head) {

if (head == NULL) {

printf("Empty list\n");

return;

}

struct DoubleNode\* current = head;

while (current != NULL) {

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

current = current->next;

}

printf("\n");

}

int main() {

struct Node\* circularList = NULL;

struct DoubleNode\* circularDoublyList = NULL;

struct DoubleNode\* doublyList = NULL;

// Insert elements into circular linked list

insertSingly(&circularList, 1);

insertSingly(&circularList, 2);

insertSingly(&circularList, 3);

// Insert elements into circular doubly linked list

insertDoubly(&circularDoublyList, 1);

insertDoubly(&circularDoublyList, 2);

insertDoubly(&circularDoublyList, 3);

// Insert elements into doubly linked list

insertDoubly(&doublyList, 1);

insertDoubly(&doublyList, 2);

insertDoubly(&doublyList, 3);

// Count nodes in circular linked list

printf("Number of nodes in circular linked list: %d\n", countNodesSingly(circularList));

// Count nodes in circular doubly linked list

printf("Number of nodes in circular doubly linked list: %d\n", countNodesDoubly(circularDoublyList));

// Count nodes in doubly linked list

printf("Number of nodes in doubly linked list: %d\n", countNodesDoubly(doublyList));

return 0;

}

**16. Write a C/C++ program to implement stack operations (PUSH) using array.**

#include <stdio.h>

#include <stdlib.h>

// Stack structure

struct Stack {

int\* array;

int capacity;

int top;

};

// Function to initialize the stack

struct Stack\* initializeStack(int capacity) {

struct Stack\* stack = (struct Stack\*)malloc(sizeof(struct Stack));

if (!stack) {

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

exit(EXIT\_FAILURE);

}

stack->capacity = capacity;

stack->top = -1;

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

if (!stack->array) {

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

exit(EXIT\_FAILURE);

}

return stack;

}

// Function to check if the stack is full

int isFull(struct Stack\* stack) {

return stack->top == stack->capacity - 1;

}

// Function to resize the stack by doubling its capacity

void resizeStack(struct Stack\* stack) {

stack->capacity \*= 2;

stack->array = (int\*)realloc(stack->array, stack->capacity \* sizeof(int));

if (!stack->array) {

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

exit(EXIT\_FAILURE);

}

}

// Function to push an element onto the stack

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

if (isFull(stack)) {

resizeStack(stack);

}

stack->array[++stack->top] = value;

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

}

// Function to check if the stack is empty

int isEmpty(struct Stack\* stack) {

return stack->top == -1;

}

// Function to pop an element from the stack

int pop(struct Stack\* stack) {

if (isEmpty(stack)) {

printf("Stack is empty. Cannot pop.\n");

exit(EXIT\_FAILURE);

}

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

}

// Function to display the elements in the stack

void display(struct Stack\* stack) {

if (isEmpty(stack)) {

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

return;

}

printf("Stack elements: ");

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

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

}

printf("\n");

}

// Function to deallocate memory used by the stack

void freeStack(struct Stack\* stack) {

free(stack->array);

free(stack);

}

int main() {

struct Stack\* stack = initializeStack(5);

push(stack, 1);

push(stack, 2);

push(stack, 3);

push(stack, 4);

push(stack, 5);

display(stack);

push(stack, 6);

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

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

display(stack);

freeStack(stack);

return 0;

}

**17. Write a C/C++ program to implement stack operations (PUSH and POP) using array.**

#include <stdio.h>

#include <stdlib.h>

#define MAX\_SIZE 5

// Stack structure

struct Stack {

int array[MAX\_SIZE];

int top;

};

// Function to initialize the stack

void initializeStack(struct Stack\* stack) {

stack->top = -1;

}

// Function to check if the stack is full

int isFull(struct Stack\* stack) {

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

}

// Function to check if the stack is empty

int isEmpty(struct Stack\* stack) {

return stack->top == -1;

}

// Function to push an element onto the stack

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

if (isFull(stack)) {

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

return;

}

stack->array[++stack->top] = value;

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

}

// Function to pop an element from the stack

int pop(struct Stack\* stack) {

if (isEmpty(stack)) {

printf("Stack is empty. Cannot pop.\n");

exit(EXIT\_FAILURE);

}

int value = stack->array[stack->top--];

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

return value;

}

// Function to display the elements in the stack

void display(struct Stack\* stack) {

if (isEmpty(stack)) {

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

return;

}

printf("Stack elements: ");

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

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

}

printf("\n");

}

int main() {

struct Stack stack;

initializeStack(&stack);

push(&stack, 1);

push(&stack, 2);

push(&stack, 3);

push(&stack, 4);

push(&stack, 5);

display(&stack);

push(&stack, 6);

pop(&stack);

pop(&stack);

display(&stack);

return 0;

}

**18. Write a C program to implement stack operation (PUSH) using linked list**.

#include <stdio.h>

#include <stdlib.h>

// Node structure for the stack

struct Node {

int data;

struct Node\* next;

};

// Stack structure

struct Stack {

struct Node\* top;

};

// Function to initialize the stack

void initializeStack(struct Stack\* stack) {

stack->top = NULL;

}

// Function to check if the stack is empty

int isEmpty(struct Stack\* stack) {

return stack->top == NULL;

}

// Function to push an element onto the stack

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

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

if (!newNode) {

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

exit(EXIT\_FAILURE);

}

newNode->data = value;

newNode->next = stack->top;

stack->top = newNode;

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

}

// Function to display the elements in the stack

void display(struct Stack\* stack) {

if (isEmpty(stack)) {

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

return;

}

printf("Stack elements: ");

struct Node\* current = stack->top;

while (current != NULL) {

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

current = current->next;

}

printf("\n");

}

int main() {

struct Stack stack;

initializeStack(&stack);

push(&stack, 1);

push(&stack, 2);

push(&stack, 3);

push(&stack, 4);

push(&stack, 5);

display(&stack);

return 0;

}

**19. Write a C/C++ program to evaluate postfix expressions using stack .**

#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

// Node structure for the stack

struct Node {

int data;

struct Node\* next;

};

// Stack structure

struct Stack {

struct Node\* top;

};

// Function to initialize the stack

void initializeStack(struct Stack\* stack) {

stack->top = NULL;

}

// Function to check if the stack is empty

int isEmpty(struct Stack\* stack) {

return stack->top == NULL;

}

// Function to push an element onto the stack

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

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

if (!newNode) {

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

exit(EXIT\_FAILURE);

}

newNode->data = value;

newNode->next = stack->top;

stack->top = newNode;

}

// Function to pop an element from the stack

int pop(struct Stack\* stack) {

if (isEmpty(stack)) {

printf("Error: Stack underflow.\n");

exit(EXIT\_FAILURE);

}

int value = stack->top->data;

struct Node\* temp = stack->top;

stack->top = stack->top->next;

free(temp);

return value;

}

// Function to evaluate a postfix expression

int evaluatePostfix(char postfix[]) {

struct Stack stack;

initializeStack(&stack);

int i, operand1, operand2, result;

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

if (isdigit(postfix[i])) {

push(&stack, postfix[i] - '0'); // Convert character to integer

} else {

operand2 = pop(&stack);

operand1 = pop(&stack);

switch (postfix[i]) {

case '+':

result = operand1 + operand2;

break;

case '-':

result = operand1 - operand2;

break;

case '\*':

result = operand1 \* operand2;

break;

case '/':

result = operand1 / operand2;

break;

default:

printf("Invalid operator: %c\n", postfix[i]);

exit(EXIT\_FAILURE);

}

push(&stack, result);

}

}

if (!isEmpty(&stack) && stack.top->next == NULL) {

result = pop(&stack);

return result;

} else {

printf("Error: Invalid postfix expression.\n");

exit(EXIT\_FAILURE);

}

}

int main() {

char postfixExpression[50];

printf("Enter postfix expression: ");

scanf("%s", postfixExpression);

int result = evaluatePostfix(postfixExpression);

printf("Result of the postfix expression: %d\n", result);

return 0;

}

**20. Write a C/C++ program to reverse a list of given numbers using stack.**

#include <stdio.h>

#include <stdlib.h>

// Node structure for the stack

struct Node {

int data;

struct Node\* next;

};

// Stack structure

struct Stack {

struct Node\* top;

};

// Function to initialize the stack

void initializeStack(struct Stack\* stack) {

stack->top = NULL;

}

// Function to check if the stack is empty

int isEmpty(struct Stack\* stack) {

return stack->top == NULL;

}

// Function to push an element onto the stack

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

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

if (!newNode) {

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

exit(EXIT\_FAILURE);

}

newNode->data = value;

newNode->next = stack->top;

stack->top = newNode;

}

// Function to pop an element from the stack

int pop(struct Stack\* stack) {

if (isEmpty(stack)) {

printf("Error: Stack underflow.\n");

exit(EXIT\_FAILURE);

}

int value = stack->top->data;

struct Node\* temp = stack->top;

stack->top = stack->top->next;

free(temp);

return value;

}

// Function to reverse a list of numbers using a stack

void reverseList(struct Stack\* stack, int numbers[], int size) {

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

push(stack, numbers[i]);

}

printf("Original list: ");

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

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

}

printf("\n");

printf("Reversed list: ");

while (!isEmpty(stack)) {

printf("%d ", pop(stack));

}

printf("\n");

}

int main() {

struct Stack stack;

initializeStack(&stack);

int numbers[] = {1, 2, 3, 4, 5};

int size = sizeof(numbers) / sizeof(numbers[0]);

reverseList(&stack, numbers, size);

return 0;

}

**21. Write a C/C++ program to implement Circular/ Priority/Double Ended queue & operation (insert and delete) using array.**

#include <stdio.h>

#include <stdlib.h>

// Maximum size of the queue

#define MAX\_SIZE 5

// Node structure for double-ended queue (deque)

struct Node {

int data;

struct Node\* next;

};

// Queue structure

struct Queue {

int front, rear;

int array[MAX\_SIZE];

};

// Priority Queue structure

struct PriorityQueue {

int array[MAX\_SIZE];

int rear;

};

// Double-ended queue (Deque) structure

struct Deque {

struct Node\* front;

struct Node\* rear;

};

// Function to initialize the circular queue

void initializeQueue(struct Queue\* queue) {

queue->front = -1;

queue->rear = -1;

}

// Function to check if the circular queue is full

int isFull(struct Queue\* queue) {

return (queue->front == 0 && queue->rear == MAX\_SIZE - 1) || (queue->front == queue->rear + 1);

}

// Function to check if the circular queue is empty

int isEmpty(struct Queue\* queue) {

return queue->front == -1;

}

// Function to insert an element at the rear of the circular queue

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

if (isFull(queue)) {

printf("Error: Circular queue is full. Cannot insert %d.\n", value);

return;

}

if (isEmpty(queue)) {

queue->front = 0;

}

queue->rear = (queue->rear + 1) % MAX\_SIZE;

queue->array[queue->rear] = value;

printf("Inserted %d at the rear of the circular queue.\n", value);

}

// Function to delete an element from the front of the circular queue

int deleteCircular(struct Queue\* queue) {

if (isEmpty(queue)) {

printf("Error: Circular queue is empty. Cannot delete.\n");

exit(EXIT\_FAILURE);

}

int value = queue->array[queue->front];

if (queue->front == queue->rear) {

// Last element in the circular queue

queue->front = -1;

queue->rear = -1;

} else {

queue->front = (queue->front + 1) % MAX\_SIZE;

}

return value;

}

// Function to initialize the priority queue

void initializePriorityQueue(struct PriorityQueue\* pq) {

pq->rear = -1;

}

// Function to check if the priority queue is full

int isFullPriority(struct PriorityQueue\* pq) {

return pq->rear == MAX\_SIZE - 1;

}

// Function to insert an element into the priority queue

void insertPriority(struct PriorityQueue\* pq, int value) {

if (isFullPriority(pq)) {

printf("Error: Priority queue is full. Cannot insert %d.\n", value);

return;

}

pq->rear++;

int i = pq->rear;

// Shift elements greater than the current value to the right

while (i > 0 && value > pq->array[i - 1]) {

pq->array[i] = pq->array[i - 1];

i--;

}

pq->array[i] = value;

printf("Inserted %d into the priority queue.\n", value);

}

// Function to delete an element from the priority queue

int deletePriority(struct PriorityQueue\* pq) {

if (pq->rear == -1) {

printf("Error: Priority queue is empty. Cannot delete.\n");

exit(EXIT\_FAILURE);

}

int value = pq->array[pq->rear];

pq->rear--;

return value;

}

// Function to initialize the double-ended queue (deque)

void initializeDeque(struct Deque\* deque) {

deque->front = NULL;

deque->rear = NULL;

}

// Function to check if the double-ended queue (deque) is empty

int isEmptyDeque(struct Deque\* deque) {

return deque->front == NULL;

}

// Function to insert an element at the front of the double-ended queue (deque)

void insertFrontDeque(struct Deque\* deque, int value) {

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

if (!newNode) {

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

exit(EXIT\_FAILURE);

}

newNode->data = value;

newNode->next = NULL;

if (isEmptyDeque(deque)) {

deque->front = newNode;

deque->rear = newNode;

} else {

newNode->next = deque->front;

deque->front = newNode;

}

printf("Inserted %d at the front of the deque.\n", value);

}

// Function to insert an element at the rear of the double-ended queue (deque)

void insertRearDeque(struct Deque\* deque, int value) {

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

if (!newNode) {

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

exit(EXIT\_FAILURE);

}

newNode->data = value;

newNode->next = NULL;

if (isEmptyDeque(deque)) {

deque->front = newNode;

deque->rear = newNode;

} else {

deque->rear->next = newNode;

deque->rear = newNode;

}

printf("Inserted %d at the rear of the deque.\n", value);

}

// Function to delete an element from the front of the double-ended queue (deque)

int deleteFrontDeque(struct Deque\* deque) {

if (isEmptyDeque(deque)) {

printf("Error: Deque is empty. Cannot delete from front.\n");

exit(EXIT\_FAILURE);

}

int value = deque->front->data;

struct Node\* temp = deque->front;

if (deque->front == deque->rear) {

// Last element in the deque

deque->front = NULL;

deque->rear = NULL;

} else {

deque->front = deque->front->next;

}

free(temp);

return value;

}

// Function to delete an element from the rear of the double-ended queue (deque)

int deleteRearDeque(struct Deque\* deque) {

if (isEmptyDeque(deque)) {

printf("Error: Deque is empty. Cannot delete from rear.\n");

exit(EXIT\_FAILURE);

}

int value = deque->rear->data;

struct Node\* temp = deque->rear;

if (deque->front == deque->rear) {

// Last element in the deque

deque->front = NULL;

deque->rear = NULL;

} else {

struct Node\* current = deque->front;

while (current->next != deque->rear) {

current = current->next;

}

current->next = NULL;

deque->rear = current;

}

free(temp);

return value;

}

// Function to display the circular queue

void displayCircularQueue(struct Queue\* queue) {

if (isEmpty(queue)) {

printf("Circular queue is empty.\n");

return;

}

printf("Circular queue elements: ");

int i = queue->front;

do {

printf("%d ", queue->array[i]);

i = (i + 1) % MAX\_SIZE;

} while (i != (queue->rear + 1) % MAX\_SIZE);

printf("\n");

}

// Function to display the priority queue

void displayPriorityQueue(struct PriorityQueue\* pq) {

if (pq->rear == -1) {

printf("Priority queue is empty.\n");

return;

}

printf("Priority queue elements: ");

for (int i = 0; i <= pq->rear; ++i) {

printf("%d ", pq->array[i]);

}

printf("\n");

}

// Function to display the double-ended queue (deque)

void displayDeque(struct Deque\* deque) {

if (isEmptyDeque(deque)) {

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

return;

}

printf("Deque elements: ");

struct Node\* current = deque->front;

while (current != NULL) {

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

current = current->next;

}

printf("\n");

}

int main() {

// Circular Queue

struct Queue circularQueue;

initializeQueue(&circularQueue);

insertCircular(&circularQueue, 1);

insertCircular(&circularQueue, 2);

insertCircular(&circularQueue, 3);

insertCircular(&circularQueue, 4);

insertCircular(&circularQueue, 5);

displayCircularQueue(&circularQueue);

printf("Deleted from circular queue: %d\n", deleteCircular(&circularQueue));

printf("Deleted from circular queue: %d\n", deleteCircular(&circularQueue));

displayCircularQueue(&circularQueue);

// Priority Queue

struct PriorityQueue priorityQueue;

initializePriorityQueue(&priorityQueue);

insertPriority(&priorityQueue, 10);

insertPriority(&priorityQueue, 5);

insertPriority(&priorityQueue, 20);

insertPriority(&priorityQueue, 15);

displayPriorityQueue(&priorityQueue);

printf("Deleted from priority queue: %d\n", deletePriority(&priorityQueue));

printf("Deleted from priority queue: %d\n", deletePriority(&priorityQueue));

displayPriorityQueue(&priorityQueue);

// Double-Ended Queue (Deque)

struct Deque deque;

initializeDeque(&deque);

insertFrontDeque(&deque, 1);

insertFrontDeque(&deque, 2);

insertRearDeque(&deque, 3);

insertRearDeque(&deque, 4);

displayDeque(&deque);

printf("Deleted from front of deque: %d\n", deleteFrontDeque(&deque));

printf("Deleted from rear of deque: %d\n", deleteRearDeque(&deque));

displayDeque(&deque);

return 0;

}

**22. Write a C/C++ program to implement Circular/ Priority/Double Ended queue & operation (insert and delete) using linked list.**

#include <stdio.h>

#include <stdlib.h>

// Node structure for the linked list

struct Node {

int data;

struct Node\* next;

};

// Circular Queue structure

struct CircularQueue {

struct Node \*front, \*rear;

};

struct Node\* createNode(int data) {

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

newNode->data = data;

newNode->next = NULL;

return newNode;

}

struct CircularQueue\* initializeQueue() {

struct CircularQueue\* cq = (struct CircularQueue\*)malloc(sizeof(struct CircularQueue));

cq->front = cq->rear = NULL;

return cq;

}

int isEmpty(struct CircularQueue\* cq) {

return cq->front == NULL;

}

void enqueue(struct CircularQueue\* cq, int data) {

struct Node\* newNode = createNode(data);

if (isEmpty(cq)) {

cq->front = cq->rear = newNode;

} else {

cq->rear->next = newNode;

cq->rear = newNode;

}

cq->rear->next = cq->front; // Make the queue circular

}

int dequeue(struct CircularQueue\* cq) {

if (isEmpty(cq)) {

printf("Queue is empty, cannot dequeue\n");

return -1;

}

int data;

struct Node\* temp = cq->front;

if (cq->front == cq->rear) {

data = temp->data;

cq->front = cq->rear = NULL;

} else {

data = temp->data;

cq->front = cq->front->next;

cq->rear->next = cq->front; // Maintain circularity

}

free(temp);

return data;

}

void display(struct CircularQueue\* cq) {

if (isEmpty(cq)) {

printf("Queue is empty\n");

return;

}

struct Node\* temp = cq->front;

do {

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

temp = temp->next;

} while (temp != cq->front);

printf("\n");

}

int main() {

struct CircularQueue\* cq = initializeQueue();

enqueue(cq, 1);

enqueue(cq, 2);

enqueue(cq, 3);

enqueue(cq, 4);

printf("Circular Queue elements: ");

display(cq);

int deletedElement = dequeue(cq);

printf("Deleted element from Circular Queue: %d\n", deletedElement);

printf("Circular Queue elements after deletion: ");

display(cq);

return 0;

}

23. Write a C/C++ program to implement BST and find

largest node in the tree

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* left;

struct Node\* right;

};

struct Node\* createNode(int data) {

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

newNode->data = data;

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

return newNode;

}

struct Node\* insert(struct Node\* root, int data) {

if (root == NULL) {

return 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;

}

struct Node\* findLargestNode(struct Node\* root) {

if (root == NULL || root->right == NULL) {

return root;

}

return findLargestNode(root->right);

}

int main() {

// Example usage of the BST

struct Node\* root = NULL;

root = insert(root, 50);

insert(root, 30);

insert(root, 70);

insert(root, 20);

insert(root, 40);

insert(root, 60);

insert(root, 80);

struct Node\* largestNode = findLargestNode(root);

if (largestNode != NULL) {

printf("The largest node in the BST is: %d\n", largestNode->data);

} else {

printf("The BST is empty.\n");

}

return 0;

}

24. Write a C/C++ program to implement BST Traversal - preorder, postorder and inorder

#include <stdio.h>

#include <stdlib.h>

// Node structure for the Binary Search Tree

struct Node {

int data;

struct Node\* left;

struct Node\* right;

};

struct Node\* createNode(int data) {

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

newNode->data = data;

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

return newNode;

}

struct Node\* insert(struct Node\* root, int data) {

if (root == NULL) {

return 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 inorderTraversal(struct Node\* root) {

if (root != NULL) {

inorderTraversal(root->left);

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

inorderTraversal(root->right);

}

}

void preorderTraversal(struct Node\* root) {

if (root != NULL) {

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

preorderTraversal(root->left);

preorderTraversal(root->right);

}

}

void postorderTraversal(struct Node\* root) {

if (root != NULL) {

postorderTraversal(root->left);

postorderTraversal(root->right);

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

}

}

int main() {

// Example usage of the BST

struct Node\* root = NULL;

root = insert(root, 50);

insert(root, 30);

insert(root, 70);

insert(root, 20);

insert(root, 40);

insert(root, 60);

insert(root, 80);

printf("Inorder Traversal: ");

inorderTraversal(root);

printf("\n");

printf("Preorder Traversal: ");

preorderTraversal(root);

printf("\n");

printf("Postorder Traversal: ");

postorderTraversal(root);

printf("\n");

return 0;

}

25 Write a C/C++ program to count the number of nodes in binary search tree.

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* left;

struct Node\* right;

};

struct Node\* createNode(int data) {

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

newNode->data = data;

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

return newNode;

}

struct Node\* insert(struct Node\* root, int data) {

if (root == NULL) {

return 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;

}

int countNodes(struct Node\* root) {

if (root == NULL) {

return 0;

}

return 1 + countNodes(root->left) + countNodes(root->right);

}

int main() {

struct Node\* root = NULL;

root = insert(root, 50);

insert(root, 30);

insert(root, 70);

insert(root, 20);

insert(root, 40);

insert(root, 60);

insert(root, 80);

int numberOfNodes = countNodes(root);

printf("Number of nodes in the BST: %d\n", numberOfNodes);

return 0;

}

26. Write a C/C++ program for sorting using Insertion Sort

#include <stdio.h>

void insertionSort(int arr[], int n) {

int i, key, j;

for (i = 1; i < n; i++) {

key = arr[i];

j = i - 1;

while (j >= 0 && arr[j] > key) {

arr[j + 1] = arr[j];

j = j - 1;

}

arr[j + 1] = key;

}

}

void printArray(int arr[], int size) {

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

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

}

printf("\n");

}

int main() {

int arr[] = {12, 11, 13, 5, 6};

int n = sizeof(arr) / sizeof(arr[0]);

printf("Original array: ");

printArray(arr, n);

insertionSort(arr, n);

printf("Sorted array: ");

printArray(arr, n);

return 0;

}

27.Write a C/C++ program to implement BST and find smallest node in the tree

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* left;

struct Node\* right;

};

struct Node\* createNode(int data) {

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

newNode->data = data;

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

return newNode;

}

struct Node\* insert(struct Node\* root, int data) {

if (root == NULL) {

return 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;

}

struct Node\* findSmallestNode(struct Node\* root) {

if (root == NULL || root->left == NULL) {

return root;

}

return findSmallestNode(root->left);

}

int main() {

// Example usage of the BST

struct Node\* root = NULL;

root = insert(root, 50);

insert(root, 30);

insert(root, 70);

insert(root, 20);

insert(root, 40);

insert(root, 60);

insert(root, 80);

struct Node\* smallestNode = findSmallestNode(root);

if (smallestNode != NULL) {

printf("The smallest node in the BST is: %d\n", smallestNode->data);

} else {

printf("The BST is empty.\n");

}

return 0;

27.Write a C/C++ program to implement BST and find smallest node in the tree

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* left;

struct Node\* right;

};

// Function to create a new node

struct Node\* createNode(int data) {

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

newNode->data = data;

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

return newNode;

}

struct Node\* insert(struct Node\* root, int data) {

if (root == NULL) {

return 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;

}

struct Node\* findSmallestNode(struct Node\* root) {

if (root == NULL || root->left == NULL) {

return root;

}

return findSmallestNode(root->left);

}

void inorderTraversal(struct Node\* root) {

if (root != NULL) {

inorderTraversal(root->left);

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

inorderTraversal(root->right);

}

}

int main() {

// Example usage of the BST

struct Node\* root = NULL;

root = insert(root, 50);

insert(root, 30);

insert(root, 70);

insert(root, 20);

insert(root, 40);

insert(root, 60);

insert(root, 80);

struct Node\* smallestNode = findSmallestNode(root);

if (smallestNode != NULL) {

printf("The smallest node in the BST is: %d\n", smallestNode->data);

} else {

printf("The BST is empty.\n");

}

// Print the inorder traversal of the BST

printf("Inorder Traversal of the BST: ");

inorderTraversal(root);

printf("\n");

return 0;

}

28. Write a C/C++ program for sorting using bubble Sort

#include <stdio.h>

// Function to perform Bubble Sort on an array

void bubbleSort(int arr[], int n) {

for (int i = 0; i < n - 1; i++) {

for (int j = 0; j < n - i - 1; j++) {

// Swap if the element found is greater

// than the next element

if (arr[j] > arr[j + 1]) {

int temp = arr[j];

arr[j] = arr[j + 1];

arr[j + 1] = temp;

}

}

}

}

// Function to print an array

void printArray(int arr[], int size) {

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

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

}

printf("\n");

}

int main() {

int arr[] = {64, 25, 12, 22, 11};

int n = sizeof(arr) / sizeof(arr[0]);

printf("Original array: ");

printArray(arr, n);

bubbleSort(arr, n);

printf("Sorted array: ");

printArray(arr, n);

return 0;

}

29 Write a C/C++ program for searching a number using linear search

#include <stdio.h>

// Function to perform linear search on an array

int linearSearch(int arr[], int n, int target) {

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

if (arr[i] == target) {

return i; // Return the index if the target is found

}

}

return -1; // Return -1 if the target is not found

}

int main() {

int arr[] = {2, 5, 8, 12, 16, 23, 38, 42, 50};

int n = sizeof(arr) / sizeof(arr[0]);

int target = 16;

int index = linearSearch(arr, n, target);

if (index != -1) {

printf("Element %d found at index %d\n", target, index);

} else {

printf("Element %d not found in the array\n", target);

}

return 0;

}

30 Write a C/C++ program for searching a number using binary search

#include <stdio.h>

// Function to perform binary search on a sorted array

int binarySearch(int arr[], int left, int right, int target) {

while (left <= right) {

int mid = left + (right - left) / 2;

// Check if the target is present at the middle

if (arr[mid] == target) {

return mid;

}

// If the target is greater, ignore the left half

if (arr[mid] < target) {

left = mid + 1;

}

// If the target is smaller, ignore the right half

else {

right = mid - 1;

}

}

// If the target is not present in the array

return -1;

}

int main() {

int arr[] = {2, 5, 8, 12, 16, 23, 38, 42, 50};

int n = sizeof(arr) / sizeof(arr[0]);

int target = 16;

int index = binarySearch(arr, 0, n - 1, target);

if (index != -1) {

printf("Element %d found at index %d\n", target, index);

} else {

printf("Element %d not found in the array\n", target);

}

return 0;

}

31 Implement a set and perform addition/ deletion of elements in it in C++ using standard template libraries (STL).

#include <iostream>

#include <set>

int main() {

// Create a set

std::set<int> mySet;

// Add elements to the set

mySet.insert(10);

mySet.insert(20);

mySet.insert(30);

mySet.insert(40);

// Display the elements in the set

std::cout << "Set elements: ";

for (const auto& element : mySet) {

std::cout << element << " ";

}

std::cout << std::endl;

// Add an element

mySet.insert(25);

// Display the updated set

std::cout << "Set elements after addition: ";

for (const auto& element : mySet) {

std::cout << element << " ";

}

std::cout << std::endl;

// Remove an element

mySet.erase(30);

// Display the set after deletion

std::cout << "Set elements after deletion: ";

for (const auto& element : mySet) {

std::cout << element << " ";

}

std::cout << std::endl;

return 0;

}

32 Implement a dictionary and display its elements with values in C++ using standard template libraries (STL).

#include <iostream>

#include <map>

int main() {

// Create a dictionary (map) with string keys and int values

std::map<std::string, int> myDictionary;

// Add elements to the dictionary

myDictionary["apple"] = 10;

myDictionary["banana"] = 5;

myDictionary["orange"] = 8;

myDictionary["grape"] = 12;

// Display the elements of the dictionary with their values

std::cout << "Dictionary elements with values:\n";

for (const auto& entry : myDictionary) {

std::cout << entry.first << ": " << entry.second << "\n";

}

return 0;

}