## **DSA Project**

**Code:**

#include <iostream>

using namespace std;

class Node {

public:

int data;

Node\* next;

Node\* prev;

Node\* left;

Node\* right;

Node(int value) {

data = value;

next = NULL;

prev=NULL;

}

};

class LinkedList {

private:

Node\* head;

public:

LinkedList() {

head = NULL;

}

void traverse() {

if (head == NULL) {

cout << "The linked list is empty." << endl;

return;

}

Node\* current = head;

cout << "Linked list elements: ";

while (current != NULL) {

cout << current->data << " ";

current = current->next;

}

cout << endl;

}

void insertAtFirst(int value) {

Node\* newNode = new Node(value);

newNode->next = head;

head = newNode;

cout << "Value " << value << " inserted at the beginning." << endl;

}

void insertAtPosition(int value, int position) {

if (position == 0) {

insertAtFirst(value);

return;

}

Node\* newNode = new Node(value);

Node\* current = head;

int index = 0;

while (current != NULL && index < position - 1) {

current = current->next;

index++;

}

if (current == NULL) {

cout << "Error: Position " << position << " is out of bounds." << endl;

delete newNode;

} else {

newNode->next = current->next;

current->next = newNode;

cout << "Value " << value << " inserted at position " << position << "." << endl;

}

}

void insert(int value) {

Node\* newNode = new Node(value);

if (head == NULL) {

head = newNode;

} else {

Node\* current = head;

while (current->next != NULL) {

current = current->next;

}

current->next = newNode;

}

cout << "Value " << value << " inserted at the end." << endl;

}

void deleteFirst() {

if (head == NULL) {

cout << "The linked list is empty. Nothing to delete." << endl;

return;

}

Node\* temp = head;

head = head->next;

delete temp;

cout << "First node deleted." << endl;

}

void deleteAtPosition(int position) {

if (head == NULL) {

cout << "The linked list is empty. Nothing to delete." << endl;

return;

}

if (position == 0) {

deleteFirst();

return;

}

Node\* current = head;

int index = 0;

while (current->next != NULL && index < position - 1) {

current = current->next;

index++;

}

if (current->next == NULL) {

cout << "Error: Position " << position << " is out of bounds." << endl;

} else {

Node\* temp = current->next;

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

delete temp;

cout << "Node at position " << position << " deleted." << endl;

}

}

void deleteLast() {

if (head == NULL) {

cout << "The linked list is empty. Nothing to delete." << endl;

return;

}

if (head->next == NULL) {

deleteFirst();

return;

}

Node\* current = head;

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

current = current->next;

}

delete current->next;

current->next = NULL;

cout << "Last node deleted." << endl;

}

void search(int value) {

Node\* current = head;

int position = 0;

while (current != NULL) {

if (current->data == value) {

cout << "Value " << value << " found at position " << position << "." << endl;

return;

}

current = current->next;

position++;

}

cout << "Value " << value << " not found in the linked list." << endl;

}

void update(int oldValue, int newValue) {

Node\* current = head;

while (current != NULL) {

if (current->data == oldValue) {

current->data = newValue;

cout << "Value " << oldValue << " updated to " << newValue << "." << endl;

return;

}

current = current->next;

}

cout << "Value " << oldValue << " not found in the linked list." << endl;

}

void bubbleSort() {

if (head == NULL || head->next == NULL) {

cout << "No sorting needed for an empty or single-element linked list." << endl;

return;

}

bool swapped;

do {

swapped = false;

Node\* current = head;

while (current->next != NULL) {

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

swap(current->data, current->next->data);

swapped = true;

}

current = current->next;

}

} while (swapped);

cout << "Linked list sorted." << endl;

}

void initializeList(LinkedList& list) {

int defaultValues[10] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};

for (int i=0;i<10;i++) {

list.insert(defaultValues[i]);

}

}

void displaySingleLinkedlist(){

int choice, value, position, oldValue, newValue;

do {

cout << "1. Traverse\n";

cout << "2. Insert at First\n";

cout << "3. Insert at Position\n";

cout << "4. Insert at Last\n";

cout << "5. Delete First\n";

cout << "6. Delete at Position\n";

cout << "7. Delete Last\n";

cout << "8. Search\n";

cout << "9. Update\n";

cout << "10. Sort\n";

cout << "11. Exit\n";

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case 1:

traverse();

break;

case 2:

cout << "Enter value to insert at the beginning: ";

cin >> value;

insertAtFirst(value);

break;

case 3:

cout << "Enter value to insert: ";

cin >> value;

cout << "Enter position to insert at: ";

cin >> position;

insertAtPosition(value, position);

break;

case 4:

cout << "Enter value to insert at the end: ";

cin >> value;

insert(value);

break;

case 5:

deleteFirst();

break;

case 6:

cout << "Enter position to delete: ";

cin >> position;

deleteAtPosition(position);

break;

case 7:

deleteLast();

break;

case 8:

cout << "Enter value to search: ";

cin >> value;

search(value);

break;

case 9:

cout << "Enter old value to update: ";

cin >> oldValue;

cout << "Enter new value: ";

cin >> newValue;

update(oldValue, newValue);

break;

case 10:

bubbleSort();

break;

case 11:

cout << "Exiting the program. Goodbye!" << endl;

break;

default:

cout << "Invalid choice! Please try again." << endl;

}

} while (choice != 11);

}

};

class Array {

private:

int arr[10];

int size;

public:

Array() {

size = 10;

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

arr[i] =i+1 ;

}

}

void traverse() {

cout << "Current elements in the array: ";

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

cout << arr[i] << " ";

}

cout << endl;

}

void search() {

int element;

cout << "Enter the element to search: ";

cin >> element;

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

if (arr[i] == element) {

cout << "Element " << element << " found at position " << i << "." << endl;

return;

}

}

cout << "Element " << element << " not found in the array." << endl;

}

void insert() {

int element, position;

cout << "Enter the element to insert: ";

cin >> element;

cout << "Enter the position (1 to " << size << "): ";

cin >> position;

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

arr[i] = arr[i - 1];

}

arr[position] = element;

size++;

cout << "Element inserted successfully." << endl;

}

void update() {

int position, newValue;

cout << "Enter the position of the element to update (1 to " << size << "): ";

cin >> position;

cout << "Enter the new value: ";

cin >> newValue;

arr[position] = newValue;

cout << "Element updated successfully." << endl;

}

void remove() {

int position;

cout << "Enter the position of the element to delete (1 to " << size << "): ";

cin >> position;

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

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

}

size--;

cout << "Element deleted successfully." << endl;

}

void sort() {

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

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

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

int temp = arr[j];

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

arr[j + 1] = temp;

}

}

}

cout << "Array sorted successfully." << endl;

}

void displayArray(){

int choice;

do {

cout << "\nOperations:\n";

cout << "1. Traverse\n";

cout << "2. Search\n";

cout << "3. Insert\n";

cout << "4. Update\n";

cout << "5. Delete\n";

cout << "6. Sort\n";

cout << "7. Exit\n";

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case 1:

traverse();

break;

case 2:

search();

break;

case 3:

insert();

break;

case 4:

update();

break;

case 5:

remove();

break;

case 6:

sort();

break;

case 7:

cout << "Exiting program." << endl;

break;

default:

cout << "Invalid choice! Try again." << endl;

}

} while (choice != 7);

}

};

class DoublyLinkedList {

private:

Node\* head;

public:

DoublyLinkedList() {

head = NULL;

}

void traverse() {

if (head == NULL) {

cout << "The linked list is empty." << endl;

return;

}

Node\* current = head;

while (current != NULL) {

cout << current->data << " <-> ";

current = current->next;

}

cout << "NULL" << endl;

}

void insertAtFirst(int value) {

Node\* newNode = new Node(value);

if (head != NULL) {

newNode->next = head;

head->prev = newNode;

}

head = newNode;

cout << "Value " << value << " inserted at the beginning of the linked list." << endl;

}

void insertAtPosition(int value, int position) {

if (position == 0) {

insertAtFirst(value);

return;

}

Node\* newNode = new Node(value);

Node\* current = head;

int index = 0;

while (current != NULL && index < position - 1) {

current = current->next;

index++;

}

if (current == NULL) {

cout << "Position " << position << " is out of bounds." << endl;

delete newNode;

} else {

newNode->next = current->next;

newNode->prev = current;

if (current->next != NULL) {

current->next->prev = newNode;

}

current->next = newNode;

cout << "Value " << value << " inserted at position " << position << "." << endl;

}

}

void insert(int value) {

Node\* newNode = new Node(value);

if (head == NULL) {

head = newNode;

} else {

Node\* current = head;

while (current->next != NULL) {

current = current->next;

}

current->next = newNode;

newNode->prev = current;

}

cout << "Value " << value << " inserted at the end of the linked list." << endl;

}

void deleteFirst() {

if (head == NULL) {

cout << "The linked list is empty." << endl;

return;

}

Node\* temp = head;

head = head->next;

if (head != NULL) {

head->prev = NULL;

}

delete temp;

cout << "First node deleted from the linked list." << endl;

}

void deleteAtPosition(int position) {

if (head == NULL) {

cout << "The linked list is empty." << endl;

return;

}

if (position == 0) {

deleteFirst();

return;

}

Node\* current = head;

int index = 0;

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

current = current->next;

index++;

}

if (current == NULL) {

cout << "Position " << position << " is out of bounds." << endl;

} else {

if (current->prev != NULL) {

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

}

if (current->next != NULL) {

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

}

delete current;

cout << "Node at position " << position << " deleted from the linked list." << endl;

}

}

void deleteLast() {

if (head == NULL) {

cout << "The linked list is empty." << endl;

return;

}

if (head->next == NULL) {

deleteFirst();

return;

}

Node\* current = head;

while (current->next != NULL) {

current = current->next;

}

current->prev->next = NULL;

delete current;

cout << "Last node deleted from the linked list." << endl;

}

void search(int value) {

Node\* current = head;

int position = 0;

while (current != NULL) {

if (current->data == value) {

cout << "Value " << value << " found at position " << position << "." << endl;

return;

}

current = current->next;

position++;

}

cout << "Value " << value << " not found in the linked list." << endl;

}

void update(int oldValue, int newValue) {

Node\* current = head;

while (current != NULL) {

if (current->data == oldValue) {

current->data = newValue;

cout << "Value " << oldValue << " updated to " << newValue << "." << endl;

return;

}

current = current->next;

}

cout << "Value " << oldValue << " not found in the linked list." << endl;

}

void displayDoublyLinkedlist(){

DoublyLinkedList list;

int choice, value, position, oldValue, newValue;

do {

cout << "\nOperations:\n";

cout << "1. Traverse\n";

cout << "2. Insert at First\n";

cout << "3. Insert at Position\n";

cout << "4. Insert at Last\n";

cout << "5. Delete First\n";

cout << "6. Delete at Position\n";

cout << "7. Delete Last\n";

cout << "8. Search\n";

cout << "9. Update\n";

cout << "10. Exit\n";

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case 1:

list.traverse();

break;

case 2:

cout << "Enter value to insert at the beginning: ";

cin >> value;

list.insertAtFirst(value);

break;

case 3:

cout << "Enter position to insert: ";

cin >> position;

cout << "Enter value to insert: ";

cin >> value;

list.insertAtPosition(value, position);

break;

case 4:

cout << "Enter value to insert at the end: ";

cin >> value;

list.insert(value);

break;

case 5:

list.deleteFirst();

break;

case 6:

cout << "Enter position to delete: ";

cin >> position;

list.deleteAtPosition(position);

break;

case 7:

list.deleteLast();

break;

case 8:

cout << "Enter value to search: ";

cin >> value;

list.search(value);

break;

case 9:

cout << "Enter old value to update: ";

cin >> oldValue;

cout << "Enter new value: ";

cin >> newValue;

list.update(oldValue, newValue);

break;

case 10:

cout << "Exiting program." << endl;

break;

default:

cout << "Invalid choice! Try again." << endl;

}

} while (choice != 10);

}

};

class CircularLinkedList {

private:

Node\* head;

public:

CircularLinkedList() {

head = NULL;

}

void traverse() {

if (head == NULL) {

cout << "The linked list is empty." << endl;

return;

}

Node\* current = head;

do {

cout << current->data << " -> ";

current = current->next;

} while (current != head);

cout << "HEAD" << endl;

}

void insertAtFirst(int value) {

Node\* newNode = new Node(value);

if (head == NULL) {

head = newNode;

head->next = head;

} else {

Node\* temp = head;

while (temp->next != head) {

temp = temp->next;

}

newNode->next = head;

temp->next = newNode;

head = newNode;

}

cout << "Value " << value << " inserted at the beginning of the linked list." << endl;

}

void insertAtPosition(int value, int position) {

if (position == 0) {

insertAtFirst(value);

return;

}

Node\* newNode = new Node(value);

Node\* current = head;

int index = 0;

do {

if (index == position - 1) {

newNode->next = current->next;

current->next = newNode;

cout << "Value " << value << " inserted at position " << position << "." << endl;

return;

}

current = current->next;

index++;

} while (current != head);

cout << "Position " << position << " is out of bounds." << endl;

delete newNode;

}

void insert(int value) {

Node\* newNode = new Node(value);

if (head == NULL) {

head = newNode;

head->next = head;

} else {

Node\* current = head;

while (current->next != head) {

current = current->next;

}

current->next = newNode;

newNode->next = head;

}

cout << "Value " << value << " inserted at the end of the linked list." << endl;

}

void deleteFirst() {

if (head == NULL) {

cout << "The linked list is empty." << endl;

return;

}

if (head->next == head) {

delete head;

head = NULL;

} else {

Node\* temp = head;

Node\* current = head;

while (current->next != head) {

current = current->next;

}

head = head->next;

current->next = head;

delete temp;

}

cout << "First node deleted from the linked list." << endl;

}

void deleteAtPosition(int position) {

if (head == NULL) {

cout << "The linked list is empty." << endl;

return;

}

if (position == 0) {

deleteFirst();

return;

}

Node\* current = head;

Node\* previous = NULL;

int index = 0;

do {

if (index == position) {

previous->next = current->next;

delete current;

cout << "Node at position " << position << " deleted from the linked list." << endl;

return;

}

previous = current;

current = current->next;

index++;

} while (current != head);

cout << "Position " << position << " is out of bounds." << endl;

}

void deleteLast() {

if (head == NULL) {

cout << "The linked list is empty." << endl;

return;

}

if (head->next == head) {

delete head;

head = NULL;

cout << "Last node deleted from the linked list." << endl;

return;

}

Node\* current = head;

Node\* previous = NULL;

while (current->next != head) {

previous = current;

current = current->next;

}

previous->next = head;

delete current;

cout << "Last node deleted from the linked list." << endl;

}

void search(int value) {

if (head == NULL) {

cout << "The linked list is empty." << endl;

return;

}

Node\* current = head;

int position = 0;

do {

if (current->data == value) {

cout << "Value " << value << " found at position " << position << "." << endl;

return;

}

current = current->next;

position++;

} while (current != head);

cout << "Value " << value << " not found in the linked list." << endl;

}

void update(int oldValue, int newValue) {

if (head == NULL) {

cout << "The linked list is empty." << endl;

return;

}

Node\* current = head;

do {

if (current->data == oldValue) {

current->data = newValue;

cout << "Value " << oldValue << " updated to " << newValue << "." << endl;

return;

}

current = current->next;

} while (current != head);

cout << "Value " << oldValue << " not found in the linked list." << endl;

}

void displayCircularLinkedlist(){

CircularLinkedList list;

int choice, value, position, oldValue, newValue;

do {

cout << "\nOperations:\n";

cout << "1. Traverse\n";

cout << "2. Insert at First\n";

cout << "3. Insert at Position\n";

cout << "4. Insert at Last\n";

cout << "5. Delete First\n";

cout << "6. Delete at Position\n";

cout << "7. Delete Last\n";

cout << "8. Search\n";

cout << "9. Update\n";

cout << "10. Exit\n";

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case 1:

list.traverse();

break;

case 2:

cout << "Enter value to insert at the beginning: ";

cin >> value;

list.insertAtFirst(value);

break;

case 3:

cout << "Enter position to insert: ";

cin >> position;

cout << "Enter value to insert: ";

cin >> value;

list.insertAtPosition(value, position);

break;

case 4:

cout << "Enter value to insert at the end: ";

cin >> value;

list.insert(value);

break;

case 5:

list.deleteFirst();

break;

case 6:

cout << "Enter position to delete: ";

cin >> position;

list.deleteAtPosition(position);

break;

case 7:

list.deleteLast();

break;

case 8:

cout << "Enter value to search: ";

cin >> value;

list.search(value);

break;

case 9:

cout << "Enter old value to update: ";

cin >> oldValue;

cout << "Enter new value: ";

cin >> newValue;

list.update(oldValue, newValue);

break;

case 10:

cout << "Exiting program." << endl;

break;

default:

cout << "Invalid choice! Try again." << endl;

}

} while (choice != 10);

}

};

class Stack{

private:

int arr[100];

int top;

int capacity;

public:

Stack(){

capacity=100;

top=-1;

}

Stack(int size) {

if (size > 100) {

cout << "Maximum stack size is 100. Setting capacity to 100." << endl;

size = 100;

}

capacity = size;

top = -1;

}

void push(int value) {

if(top==capacity-1){

cout<<"Stack overflow.Cannot push "<<value<<"."<<endl;

return;

}

arr[++top]=value;

cout<<"value "<<value<<" pushed in the stack."<<endl;

}

void pop() {

if(top==-1){

cout<<"Stack underflow.Cannot pop."<<endl;

return;

}

cout<<"value "<<arr[top--]<<" popped from the stack."<<endl;

}

int peek() {

if(top==-1){

cout<<"Stack is empty."<<endl;

return 0;

}

return arr[top];

}

void traverse() {

if(top==-1){

cout<<"Stack is empty."<<endl;

return;

}

cout<<"stack elements: ";

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

cout<<arr[i]<<" ";

}

cout<<endl;

}

void search(int value) {

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

if (arr[i] == value) {

cout << "Value " << value << " found at index " << i << "." << endl;

return;

}

}

cout << "Value " << value << " not found in the stack." << endl;

}

void update(int oldValue, int newValue) {

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

if (arr[i] == oldValue) {

arr[i] = newValue;

cout << "Value " << oldValue << " updated to " << newValue << "." << endl;

return;

}

}

cout << "Value " << oldValue << " not found in the stack." << endl;

}

void displayStack(){

int size;

cout << "Enter the size of the stack (max 100): ";

cin >> size;

int choice, value, oldValue, newValue;

Stack stack(size);

do {

cout << "\nMenu:\n";

cout << "1. Traverse\n";

cout << "2. Search\n";

cout << "3. Push (Insert)\n";

cout << "4. Pop (Delete)\n";

cout << "5. Update\n";

cout << "6. Exit\n";

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case 1:

traverse();

break;

case 2:

cout << "Enter value to search: ";

cin >> value;

search(value);

break;

case 3:

cout << "Enter value to push: ";

cin >> value;

push(value);

break;

case 4:

pop();

break;

case 5:

cout << "Enter value to update: ";

cin >> oldValue;

cout << "Enter new value: ";

cin >> newValue;

update(oldValue, newValue);

break;

case 6:

cout << "Exiting program." << endl;

break;

default:

cout << "Invalid choice. Try again." << endl;

}

} while (choice != 6);

}

};

class Queue {

private:

int arr[100];

int front;

int rear;

int capacity;

public:

Queue(){

int size;

capacity = size;

front = -1;

rear = -1;

}

Queue(int size) {

if (size > 100) {

cout << "Maximum queue size is 100. Setting capacity to 100." << endl;

size = 100;

}

capacity = size;

front = -1;

rear = -1;

}

void enqueue(int value) {

if((rear+1)%capacity==front){

cout<<"Queue overflow."<<endl;

return ;

}

if(front==-1){

front=0;

}

rear=(rear+1)%capacity;

arr[rear]=value;

cout<<"value enqueued into the queue."<<endl;

}

void dequeue() {

if(front==-1){

cout<<"Queue underflow."<<endl;

return ;

}

cout<<"value"<<arr[front]<< " dequeued." <<endl;

if(front==rear){

front=rear=-1;

}

else{

front=(front+1)%capacity;

}

}

void traverse() {

if(front==-1){

cout<<"Queue is empty."<<endl;

return ;

}

cout<<"queue elements:";

int i=front;

while(true){

cout<<arr[i]<<" ";

if(i==rear)break;

i=(i+1)%capacity;

}

cout<<endl;

}

void search(int value) {

if (front == -1) {

cout << "Queue is empty." << endl;

return;

}

int i = front;

int pos=0;

while (true) {

if (arr[i] == value) {

cout << "Value " << value << " found at position " << pos << "from the front." << endl;

return;

}

if (i == rear) break;

i = (i + 1) % capacity;

pos++;

}

cout << "Value " << value << " not found in the queue." << endl;

}

void update(int oldValue, int newValue) {

if (front == -1) {

cout << "Queue is empty." << endl;

return;

}

int i = front;

while (true) {

if (arr[i] == oldValue) {

arr[i] = newValue;

cout << "Value " << oldValue << " updated to " << newValue << "." << endl;

return;

}

if (i == rear) break;

i = (i + 1) % capacity;

}

cout << "Value " << oldValue << " not found in the queue." << endl;

}

void displayQueue(){

int size;

cout << "Enter the size of the queue (max 100): ";

cin >> size;

Queue queue(size);

int choice, value, oldValue, newValue;

do {

cout << "\nMenu:\n";

cout << "1. Traverse\n";

cout << "2. Search\n";

cout << "3. Enqueue (Insert)\n";

cout << "4. Dequeue (Delete)\n";

cout << "5. Update\n";

cout << "6. Exit\n";

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case 1:

queue.traverse();

break;

case 2:

cout << "Enter value to search: ";

cin >> value;

queue.search(value);

break;

case 3:

cout << "Enter value to enqueue: ";

cin >> value;

queue.enqueue(value);

break;

case 4:

queue.dequeue();

break;

case 5:

cout << "Enter value to update: ";

cin >> oldValue;

cout << "Enter new value: ";

cin >> newValue;

queue.update(oldValue, newValue);

break;

case 6:

cout << "Exiting program." << endl;

break;

default:

cout << "Invalid choice. Try again." << endl;

}

} while (choice != 6);

}

};

class BST {

private:

Node\* root;

Node\* createNode(int value) {

Node\* newNode = new Node(value);

newNode->data = value;

newNode->left = NULL;

newNode->right = NULL;

return newNode;

}

Node\* insert(Node\* root, int value) {

if (root == NULL) {

return createNode(value);

}

if (value < root->data) {

root->left = insert(root->left, value);

} else if (value > root->data) {

root->right = insert(root->right, value);

}

return root;

}

bool search(Node\* root, int value) {

if (root == NULL) return false;

if (root->data == value) return true;

if (value < root->data) return search(root->left, value);

return search(root->right, value);

}

Node\* findMin(Node\* root) {

while (root->left != NULL) {

root = root->left;

}

return root;

}

Node\* deleteNode(Node\* root, int value) {

if (root == NULL) {

return root;

}

if (value < root->data) {

root->left = deleteNode(root->left, value);

} else if (value > root->data) {

root->right = deleteNode(root->right, value);

} else {

if (root->left == NULL) {

Node\* temp = root->right;

delete root;

return temp;

} else if (root->right == NULL) {

Node\* temp = root->left;

delete root;

return temp;

}

Node\* temp = findMin(root->right);

root->data = temp->data;

root->right = deleteNode(root->right, temp->data);

}

return root;

}

void inorder(Node\* root) {

if (root != NULL) {

inorder(root->left);

cout << root->data << " ";

inorder(root->right);

}

}

public:

BST() {

root = NULL;

}

void insert(int value) {

root = insert(root, value);

cout << "Value " << value << " inserted into the BST." << endl;

}

void search(int value) {

if (search(root, value)) {

cout << "Value " << value << " found in the BST." << endl;

} else {

cout << "Value " << value << " not found in the BST." << endl;

}

}

void deleteNode(int value) {

root = deleteNode(root, value);

cout << "Value " << value << " deleted from the BST (if it existed)." << endl;

}

void traverse() {

if (root == NULL) {

cout << "BST is empty." << endl;

} else {

cout << "Inorder traversal: ";

inorder(root);

cout << endl;

}

}

void displayBST(){

BST bst;

int choice, value;

do {

cout << "\nMenu:\n";

cout << "1. Traverse (Inorder)\n";

cout << "2. Search\n";

cout << "3. Insert\n";

cout << "4. Delete\n";

cout << "5. Exit\n";

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case 1:

bst.traverse();

break;

case 2:

cout << "Enter value to search: ";

cin >> value;

bst.search(value);

break;

case 3:

cout << "Enter value to insert: ";

cin >> value;

bst.insert(value);

break;

case 4:

cout << "Enter value to delete: ";

cin >> value;

bst.deleteNode(value);

break;

case 5:

cout << "Exiting program." << endl;

break;

default:

cout << "Invalid choice. Try again." << endl;

}

} while (choice != 5);

}

};

int main() {

Array array;

LinkedList list;

DoublyLinkedList doublelist;

CircularLinkedList circularlist;

Stack stack;

Queue queue;

BST bst;

int choice;

do {

cout << "\nOperations:\n";

cout << "1. Array\n";

cout << "2. Sinle linked list\n";

cout << "3. Doubly linked list\n";

cout << "4. Circular linked list\n";

cout << "5. Stack\n";

cout << "6. Queue\n";

cout << "7. Binary Search Tree\n";

cout << "8. Exit\n";

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case 1:

array.displayArray();

break;

case 2:

list.displaySingleLinkedlist();

break;

case 3:

doublelist.displayDoublyLinkedlist();

break;

case 4:

circularlist.displayCircularLinkedlist();

break;

case 5:

stack.displayStack();

break;

case 6:

queue.displayQueue();

break;

case 7:

bst.displayBST();

break;

case 8:

cout << "Exiting program." << endl;

break;

default:

cout << "Invalid choice! Try again." << endl;

}

} while (choice != 7);

return 0;

}