1.

#include<iostream>

#include<queue>

using namespace std;

struct Node

{

string key;

string value;

Node\* left;

Node\* right;

};

class DictionaryBST

{

public:

Node\* root;

DictionaryBST() { root = NULL; }

void insert(string k, string v)

{

Node\* newNode = new Node();

newNode->key = k;

newNode->value = v;

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

if (root == NULL)

{

root = newNode;

return;

}

Node\* current = root;

while (true)

{

if (k < current->key)

{

if (current->left == NULL)

{

current->left = newNode;

return;

}

current = current->left;

}

else

{

if (current->right == NULL)

{

current->right = newNode;

return;

}

current = current->right;

}

}

}

void inorder(Node\* node)

{

if (node)

{

inorder(node->left);

cout << endl << node->key << " : " << node->value;

inorder(node->right);

}

}

void preorder(Node\* node)

{

if (node)

{

cout << endl << node->key << " : " << node->value;

preorder(node->left);

preorder(node->right);

}

}

void postorder(Node\* node)

{

if (node)

{

postorder(node->left);

postorder(node->right);

cout << endl << node->key << " : " << node->value;

}

}

void levelOrder(Node\* root)

{

if (!root) return;

queue<Node\*> q;

q.push(root);

while (!q.empty())

{

Node\* temp = q.front();

q.pop();

cout << endl << temp->key << " : " << temp->value;

if (temp->left) q.push(temp->left);

if (temp->right) q.push(temp->right);

}

}

void search(string k)

{

Node\* current = root;

while (current)

{

if (k == current->key)

{

cout << "\nFound: " << current->key << " : " << current->value;

return;

}

current = (k < current->key) ? current->left : current->right;

}

cout << "\nNot found.";

}

int height(Node\* node)

{

if (!node) return 0;

int lheight = height(node->left);

int rheight = height(node->right);

return max(lheight, rheight) + 1;

}

Node\* minNode(Node\* node)

{

Node\* temp = node;

while (temp && temp->left)

temp = temp->left;

return temp;

}

Node\* remove(Node\* node, string k)

{

if (!node) return node;

if (k < node->key)

node->left = remove(node->left, k);

else if (k > node->key)

node->right = remove(node->right, k);

else

{

if (!node->left)

{

Node\* temp = node->right;

delete node;

return temp;

}

else if (!node->right)

{

Node\* temp = node->left;

delete node;

return temp;

}

Node\* temp = minNode(node->right);

node->key = temp->key;

node->value = temp->value;

node->right = remove(node->right, temp->key);

}

return node;

}

Node\* cloneTree(Node\* node)

{

if (!node) return NULL;

Node\* newNode = new Node{ node->key, node->value, NULL, NULL };

newNode->left = cloneTree(node->left);

newNode->right = cloneTree(node->right);

return newNode;

}

Node\* mirrorTree(Node\* node)

{

if (!node) return NULL;

Node\* mirrored = new Node{ node->key, node->value, NULL, NULL };

mirrored->left = mirrorTree(node->right);

mirrored->right = mirrorTree(node->left);

return mirrored;

}

};

int main()

{

DictionaryBST dict;

int choice;

string key, value;

Node\* clonedTree = NULL;

Node\* mirroredTree = NULL;

do

{

cout << "\n1. Insert";

cout << "\n2. Inorder";

cout << "\n3. Preorder";

cout << "\n4. Postorder";

cout << "\n5. Search";

cout << "\n6. Height";

cout << "\n7. Delete";

cout << "\n8. Clone";

cout << "\n9. Mirror";

cout << "\n10. Level Order";

cout << "\n11. Exit";

cout << "\nEnter choice: ";

cin >> choice;

switch (choice)

{

case 1:

cout << "\nEnter word: ";

cin >> key;

cout << "\nEnter meaning: ";

cin.ignore();

getline(cin, value);

dict.insert(key, value);

break;

case 2:

dict.inorder(dict.root);

break;

case 3:

dict.preorder(dict.root);

break;

case 4:

dict.postorder(dict.root);

break;

case 5:

cout << "\nEnter word to search: ";

cin >> key;

dict.search(key);

break;

case 6:

cout << "\nHeight: " << dict.height(dict.root);

break;

case 7:

cout << "\nEnter word to delete: ";

cin >> key;

dict.root = dict.remove(dict.root, key);

break;

case 8:

clonedTree = dict.cloneTree(dict.root);

cout << "\nCloned tree inorder:";

dict.inorder(clonedTree);

break;

case 9:

mirroredTree = dict.mirrorTree(dict.root);

cout << "\nMirrored tree inorder:";

dict.inorder(mirroredTree);

break;

case 10:

cout << "\nLevel Order Traversal:";

dict.levelOrder(dict.root);

break;

case 11:

break;

}

} while (choice != 11);

return 0;

}

2.

#include<iostream>

using namespace std;

struct Node{

int player\_id;

float score;

int height;

Node\* left;

Node\* right;

Node(int pid, float sc){

player\_id = pid;

score = sc;

height = 1;

left = right = NULL;

}

};

class Player{

public:

Node\* root;

Player(){

root = NULL;

}

int getHeight(Node\* node){

if(!node) return 0;

int lheight = getHeight(node->left);

int rheight = getHeight(node->right);

return max(lheight, rheight) + 1;

}

int getBalancedFactor(Node\* node){

if (root == NULL) return 0;

return (getHeight(root->left) - getHeight(root->right));

}

Node\* rightRotate(Node\* y){

Node\* x = y->left;

Node\* T2 = x->right;

x->right = y;

y->left = T2;

y->height = max(getHeight(y->left), getHeight(y->right)) + 1;

x->height = max(getHeight(x->left), getHeight(x->right)) + 1;

return x;

}

Node\* leftRotate(Node\* x){

Node\* y = x->right;

Node\* T2 = y->left;

y->left = x;

x->right = T2;

x->height = max(getHeight(x->left), getHeight(x->right)) + 1;

y->height = max(getHeight(y->left), getHeight(y->right)) + 1;

return y;

}

Node\* insertNode(Node\* node, int pid, float sc){

if (!node)

return new Node(pid, sc);

if (sc < node->score)

node->left = insertNode(node->left, pid, sc);

else if (sc > node->score)

node->right = insertNode(node->right, pid, sc);

else

return node; // Duplicate scores are not allowed

node->height = 1 + max(getHeight(node->left), getHeight(node->right));

int balance = getBalancedFactor(node);

if (balance > 1 && sc < node->left->score)

return rightRotate(node);

if (balance < -1 && sc > node->right->score)

return leftRotate(node);

if (balance > 1 && sc > node->left->score)

{

node->left = leftRotate(node->left);

return rightRotate(node);

}

if (balance < -1 && sc < node->right->score)

{

node->right = rightRotate(node->right);

return leftRotate(node);

}

return node;

}

void display(Node\* node) {

if (node) {

display(node->right);

cout << "\nPlayer ID: " << node->player\_id << " | Score: " << node->score;

display(node->left);

}

}

Node\* minValueNode(Node\* node) {

Node\* current = node;

while (current->left)

current = current->left;

return current;

}

Node\* deleteNode(Node\* root, float score) {

if (!root) return root;

if (score < root->score)

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

else if (score > root->score)

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

else {

// Node with one or no child

if (!root->left || !root->right) {

Node\* temp = root->left ? root->left : root->right;

if (!temp) {

temp = root;

root = NULL;

} else

\*root = \*temp; // Copy contents of the non-empty child

delete temp;

} else {

// Node with two children

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

root->score = temp->score;

root->player\_id = temp->player\_id;

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

}

}

if (!root) return root;

// Update height

root->height = 1 + max(getHeight(root->left), getHeight(root->right));

// Check balance

int balance = getBalancedFactor(root);

// Left Left Case

if (balance > 1 && getBalancedFactor(root->left) >= 0)

return rightRotate(root);

// Left Right Case

if (balance > 1 && getBalancedFactor(root->left) < 0) {

root->left = leftRotate(root->left);

return rightRotate(root);

}

// Right Right Case

if (balance < -1 && getBalancedFactor(root->right) <= 0)

return leftRotate(root);

// Right Left Case

if (balance < -1 && getBalancedFactor(root->right) > 0) {

root->right = rightRotate(root->right);

return leftRotate(root);

}

return root;

}

};

int main()

{

Player p;

int pi, ch;

float s, r;

do

{

cout << "\n1. Register player";

cout << "\n2. Display Player";

cout << "\n3. Remove Player";

cout << "\n4. Exit";

cout << "\nEnter your choice: ";

cin >> ch;

switch (ch)

{

case 1:

cout << "\nEnter player id: ";

cin >> pi;

cout << "\nEnter player score: ";

cin >> s;

p.root = p.insertNode(p.root, pi, s);

break;

case 2:

p.display(p.root);

break;

case 3:

cout << "\nEnter score of player which is to be removed";

cin >> r;

p.root = p.deleteNode(p.root, r);

cout<< "\nPlayer removed if score existed";

break;

case 4:

break;

}

} while (ch != 4);

return 0;

}

3.

#include <iostream>

#include <queue>

using namespace std;

// DFS & BFS

int a[7][7] = {

{0, 1, 1, 1, 0, 0, 0},

{1, 0, 0, 1, 0, 0, 0},

{1, 0, 0, 1, 1, 0, 0},

{1, 1, 1, 0, 1, 0, 0},

{0, 0, 1, 1, 0, 1, 1},

{0, 0, 0, 0, 1, 0, 0},

{0, 0, 0, 0, 1, 0, 0}};

//DFS

void DFS(int start, int end, int visited[])

{

cout << start << " ";

visited[start] = 1;

if (start == end)

{

cout << "End Reached (DFS)" << endl;

return;

}

for (int j = 0; j < 7; j++)

{

if (a[start][j] == 1 && visited[j] == 0)

{

DFS(j, end, visited);

return;

}

}

}

//BFS

void BFS(int start, int end, int visit[])

{

queue<int> q;

visit[start] = 1;

q.push(start);

while (!q.empty())

{

int d = q.front();

q.pop();

cout << d << " ";

if (d == end)

{

cout << "End Reached (BFS)" << endl;

return;

}

for (int j = 0; j < 7; j++)

{

if (a[d][j] == 1 && visit[j] == 0)

{

visit[j] = 1;

q.push(j);

}

}

}

}

int main()

{

int BFSvisit[7] = {0};

int DFSvisit[7] = {0};

cout << "DFS Traversal: ";

DFS(0, 5, DFSvisit);

cout << "BFS Traversal: ";

BFS(0, 5, BFSvisit);

return 0;

}

4.

#include <iostream>

#include <queue>

using namespace std;

//Prim's Algorithm

#define MAX 100 // Maximum number of departments

#define INF 1000000000 // Large value to represent infinity

class Graph {

int V;

int adjMatrix[MAX][MAX];

public:

Graph(int V) {

this->V = V;

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

for (int j = 0; j < V; j++) {

adjMatrix[i][j] = (i == j) ? 0 : INF;

}

}

}

void addEdge(int u, int v, int weight) {

adjMatrix[u][v] = weight;

adjMatrix[v][u] = weight;

}

void primMST() {

int key[MAX];

int parent[MAX];

bool inMST[MAX] = {false};

for (int i = 0; i < V; i++) key[i] = INF, parent[i] = -1;

key[0] = 0;

for (int count = 0; count < V - 1; count++) {

int u = -1;

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

if (!inMST[i] && (u == -1 || key[i] < key[u])) {

u = i;

}

}

inMST[u] = true;

for (int v = 0; v < V; v++) {

if (adjMatrix[u][v] && !inMST[v] && adjMatrix[u][v] < key[v]) {

key[v] = adjMatrix[u][v];

parent[v] = u;

}

}

}

cout << "Minimum Spanning Tree (MST):\n";

for (int i = 1; i < V; i++) {

cout << "Department " << parent[i] << " - " << i << " (Distance: " << key[i] << ")\n";

}

}

};

int main() {

int V, E;

cout << "Enter number of departments: ";

cin >> V;

Graph campus(V);

cout << "Enter number of connections: ";

cin >> E;

cout << "Enter connections (department1 department2 distance):\n";

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

int u, v, weight;

cin >> u >> v >> weight;

campus.addEdge(u, v, weight);

}

campus.primMST();

return 0;

}

5.

#include <iostream>

using namespace std;

#define MAX 100

#define INF 1000000000 // Large value to represent infinity

class SupplyChain {

int locations;

int graph[MAX][MAX];

public:

SupplyChain(int loc) {

locations = loc;

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

for (int j = 0; j < loc; j++) {

graph[i][j] = (i == j) ? 0 : INF;

}

}

}

void addRoute(int u, int v, int distance) {

graph[u][v] = distance;

graph[v][u] = distance; // Undirected graph

}

int findMinDistance(int dist[], bool visited[]) {

int min = INF, minIndex = -1;

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

if (!visited[i] && dist[i] < min) {

min = dist[i], minIndex = i;

}

}

return minIndex;

}

void dijkstra(int src) {

int dist[MAX];

bool visited[MAX] = {false};

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

dist[i] = INF;

}

dist[src] = 0;

for (int count = 0; count < locations - 1; count++) {

int u = findMinDistance(dist, visited);

if (u == -1) break;

visited[u] = true;

for (int v = 0; v < locations; v++) {

if (!visited[v] && graph[u][v] != INF && dist[u] != INF && dist[u] + graph[u][v] < dist[v]) {

dist[v] = dist[u] + graph[u][v];

}

}

}

cout << "Shortest distances from source:\n";

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

cout << "To location " << i << " -> Distance: " << (dist[i] == INF ? -1 : dist[i]) << "\n";

}

}

};

int main() {

int locations, routes;

cout << "Enter number of locations: ";

cin >> locations;

SupplyChain sc(locations);

cout << "Enter number of routes: ";

cin >> routes;

cout << "Enter routes (start end distance):\n";

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

int u, v, d;

cin >> u >> v >> d;

sc.addRoute(u, v, d);

}

int source;

cout << "Enter source location: ";

cin >> source;

sc.dijkstra(source);

return 0;

}

6.

#include <iostream>

#include <climits>

#include <string>

using namespace std;

enum Color { RED, BLACK };

struct Node {

int data; // directory name or ID (you can use string instead of int)

Color color;

Node \*left, \*right, \*parent;

Node(int data) : data(data) {

parent = left = right = nullptr;

color = RED;

}

};

class RedBlackTree {

private:

Node\* root;

// Left rotate x

void leftRotate(Node\* x) {

Node\* y = x->right;

x->right = y->left;

if (y->left != nullptr)

y->left->parent = x;

y->parent = x->parent;

if (x->parent == nullptr)

root = y;

else if (x == x->parent->left)

x->parent->left = y;

else

x->parent->right = y;

y->left = x;

x->parent = y;

}

// Right rotate x

void rightRotate(Node\* x) {

Node\* y = x->left;

x->left = y->right;

if (y->right != nullptr)

y->right->parent = x;

y->parent = x->parent;

if (x->parent == nullptr)

root = y;

else if (x == x->parent->right)

x->parent->right = y;

else

x->parent->left = y;

y->right = x;

x->parent = y;

}

// Fix violations after insertion

void fixInsert(Node\* k) {

Node\* parent = nullptr;

Node\* grandparent = nullptr;

while (k != root && k->color == RED && k->parent->color == RED) {

parent = k->parent;

grandparent = parent->parent;

// Parent is left child of grandparent

if (parent == grandparent->left) {

Node\* uncle = grandparent->right;

// Case 1: Uncle is red

if (uncle != nullptr && uncle->color == RED) {

grandparent->color = RED;

parent->color = BLACK;

uncle->color = BLACK;

k = grandparent;

}

else {

// Case 2: k is right child

if (k == parent->right) {

leftRotate(parent);

k = parent;

parent = k->parent;

}

// Case 3: k is left child

rightRotate(grandparent);

swap(parent->color, grandparent->color);

k = parent;

}

}

else {

Node\* uncle = grandparent->left;

// Case 1

if (uncle != nullptr && uncle->color == RED) {

grandparent->color = RED;

parent->color = BLACK;

uncle->color = BLACK;

k = grandparent;

}

else {

// Case 2

if (k == parent->left) {

rightRotate(parent);

k = parent;

parent = k->parent;

}

// Case 3

leftRotate(grandparent);

swap(parent->color, grandparent->color);

k = parent;

}

}

}

root->color = BLACK;

}

void inorderTraversal(Node\* root) {

if (root != nullptr) {

inorderTraversal(root->left);

cout << root->data << " (" << (root->color == RED ? "R" : "B") << ") ";

inorderTraversal(root->right);

}

}

public:

RedBlackTree() : root(nullptr) {}

void insert(int data) {

Node\* node = new Node(data);

Node\* y = nullptr;

Node\* x = root;

while (x != nullptr) {

y = x;

if (node->data < x->data)

x = x->left;

else

x = x->right;

}

node->parent = y;

if (y == nullptr)

root = node;

else if (node->data < y->data)

y->left = node;

else

y->right = node;

// Fix Red-Black Tree properties

fixInsert(node);

}

void display() {

inorderTraversal(root);

cout << endl;

}

};

int main() {

RedBlackTree R1;

int choice, value;

do {

cout << "\n===== Directory Red-Black Tree Menu =====\n";

cout << "1. Insert Directory\n";

cout << "2. Display Inorder\n";

cout << "3. Exit\n";

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case 1:

cout << "Enter directory ID (int): ";

cin >> value;

R1.insert(value);

cout << "Inserted successfully.\n";

break;

case 2:

cout << "Directory Structure (Inorder Traversal):\n";

R1.display();

break;

case 3:

cout << "Exiting the file system. Goodbye!\n";

break;

default:

cout << "Invalid choice. Please try again.\n";

}

} while (choice != 3);

return 0;

}

Kruskal

#include <iostream>

using namespace std;

#define MAX 100

#define INF 1000000000

struct Edge {

int u, v, weight;

};

class Graph {

int V;

int adj[MAX][MAX];

string dept[MAX];

Edge edges[MAX \* MAX];

int edgeCount = 0;

public:

Graph(int v) {

V = v;

for (int i = 0; i < V; i++)

for (int j = 0; j < V; j++)

adj[i][j] = (i == j) ? 0 : INF;

}

void addDepartment(int index, string name) {

dept[index] = name;

}

int getIndex(string name) {

for (int i = 0; i < V; i++)

if (dept[i] == name) return i;

return -1;

}

void addEdge(string d1, string d2, int w) {

int u = getIndex(d1);

int v = getIndex(d2);

if (u != -1 && v != -1) {

adj[u][v] = w;

adj[v][u] = w;

edges[edgeCount++] = {u, v, w};

}

}

int find(int parent[], int i) {

if (parent[i] != i)

parent[i] = find(parent, parent[i]);

return parent[i];

}

void unionSets(int parent[], int x, int y) {

parent[x] = y;

}

void sortEdges() {

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

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

if (edges[j].weight > edges[j + 1].weight) {

Edge temp = edges[j];

edges[j] = edges[j + 1];

edges[j + 1] = temp;

}

}

}

}

void kruskalMST() {

int parent[MAX];

for (int i = 0; i < V; i++)

parent[i] = i;

sortEdges();

cout << "Minimum Spanning Tree (Kruskal):\n";

int total = 0;

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

int u = edges[i].u;

int v = edges[i].v;

int pu = find(parent, u);

int pv = find(parent, v);

if (pu != pv) {

cout << dept[u] << " - " << dept[v] << " (Distance: " << edges[i].weight << ")\n";

total += edges[i].weight;

unionSets(parent, pu, pv);

}

}

cout << "Total distance of MST: " << total << "\n";

}

};

int main() {

int V, E;

cout << "Enter number of departments: ";

cin >> V;

Graph g(V);

cout << "Enter department names:\n";

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

string name;

cin >> name;

g.addDepartment(i, name);

}

cout << "Enter number of connections: ";

cin >> E;

cout << "Enter connections (dept1 dept2 distance):\n";

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

string d1, d2;

int w;

cin >> d1 >> d2 >> w;

g.addEdge(d1, d2, w);

}

g.kruskalMST();

return 0;

}

Implement a Binary Search Tree (BST) Class of integers using Array Representation having

following operations:

A. BST(.. ) // constructor to populate BST by generating n random numbers in the range 1 to

100

B. In-order() // inorder traversal of the BST.

C. FindClosest()// function to return element closest to the given element if its not found else

return the same element. [ closest means mod of min difference]

If tree has elements - 13 15 28 19

Element to be searched - 23

Returned element = 19

#include <iostream>

#include <cstdlib>

#include <ctime>

#include <cmath>

#include <climits>

using namespace std;

class BST {

private:

int tree[100]; // Fixed array to store BST

int size = 100;

public:

// Constructor: generates and inserts n unique random integers between 1 and 100

BST(int n) {

for (int i = 0; i < size; i++) tree[i] = -1;

srand(time(0));

int count = 0;

cout << "Inserted elements: ";

while (count < n) {

int val = rand() % 100 + 1;

if (!isPresent(val)) {

insert(0, val);

cout << val << " ";

count++;

}

}

cout << endl;

}

// Check for duplicates

bool isPresent(int value) {

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

if (tree[i] == value) return true;

}

return false;

}

// Insert value into BST (recursive)

void insert(int index, int value) {

if (index >= size) return;

if (tree[index] == -1) {

tree[index] = value;

} else if (value < tree[index]) {

insert(2 \* index + 1, value); // Left child

} else {

insert(2 \* index + 2, value); // Right child

}

}

// In-order traversal

void inorder(int index) {

if (index >= size || tree[index] == -1) return;

inorder(2 \* index + 1);

cout << tree[index] << " ";

inorder(2 \* index + 2);

}

// Find the closest element to a given key

int findClosest(int key) {

int index = 0;

int closest = -1;

int minDiff = INT\_MAX;

while (index < size && tree[index] != -1) {

int diff = abs(tree[index] - key);

if (diff < minDiff) {

minDiff = diff;

closest = tree[index];

}

if (key == tree[index]) {

return tree[index]; // Exact match found

} else if (key < tree[index]) {

index = 2 \* index + 1; // Move left

} else {

index = 2 \* index + 2; // Move right

}

}

return closest;

}

};

// -------------------- MAIN --------------------

int main() {

int n, key;

cout << "Enter number of elements to insert: ";

cin >> n;

BST bst(n);

cout << "\nIn-order Traversal: ";

bst.inorder(0);

cout << endl;

cout << "\nEnter element to find closest match: ";

cin >> key;

int closest = bst.findClosest(key);

cout << "Element closest to " << key << " is: " << closest << endl;

return 0;

}

Implement a Binary Search Tree (BST) Class of integers using an Array having following

operations:

A. BST(.. ) // constructor to populate BST with the integers from a text file “input.txt”.

B. In-order() // Non-Recursive inorder traversal of the BST.

C. Search(..) // Recursive function to search for a key

D. Write a driver code to invoke and test above functions

#include <iostream>

#include <fstream>

#include <stack>

using namespace std;

class BST {

private:

int tree[100];

int size = 100;

public:

// Constructor: Read from file and insert into BST

BST() {

for (int i = 0; i < size; i++) tree[i] = -1;

ifstream file("input.txt");

if (!file) {

cout << "Error: Cannot open input.txt\n";

exit(1);

}

int num;

cout << "Inserting values from file: ";

while (file >> num) {

cout << num << " ";

insert(0, num);

}

cout << endl;

file.close();

}

// Insert into array-based BST

void insert(int index, int value) {

if (index >= size) return;

if (tree[index] == -1) {

tree[index] = value;

} else if (value < tree[index]) {

insert(2 \* index + 1, value);

} else {

insert(2 \* index + 2, value);

}

}

// Non-recursive in-order traversal

void inorder() {

stack<int> s;

int index = 0;

while (!s.empty() || (index < size && tree[index] != -1)) {

while (index < size && tree[index] != -1) {

s.push(index);

index = 2 \* index + 1; // Move to left child

}

if (!s.empty()) {

index = s.top();

s.pop();

cout << tree[index] << " ";

index = 2 \* index + 2; // Move to right child

}

}

}

// Recursive search

bool search(int index, int key) {

if (index >= size || tree[index] == -1)

return false;

if (tree[index] == key)

return true;

else if (key < tree[index])

return search(2 \* index + 1, key);

else

return search(2 \* index + 2, key);

}

};

// ---------------- DRIVER CODE ----------------

int main() {

BST bst;

cout << "\nIn-order traversal (non-recursive): ";

bst.inorder();

cout << endl;

int key;

cout << "\nEnter element to search: ";

cin >> key;

bool found = bst.search(0, key);

if (found)

cout << key << " is found in the BST.\n";

else

cout << key << " is NOT found in the BST.\n";

return 0;

}

Implement a Binary Search Tree (BST) Class of integers using Linked Representation having

following operations:

A. BST(.. ) // constructor to populate BST with the integers from a text file “input.txt”.

B. In-order() // inorder traversal of the BST.

C. EmptyIt() // function to delete all nodes of a tree

#include <iostream>

#include <fstream>

using namespace std;

// Define Node structure

struct Node {

int data;

Node\* left;

Node\* right;

Node(int val) {

data = val;

left = right = NULL;

}

};

// BST Class

class BST {

private:

Node\* root;

// Helper for insertion

Node\* insert(Node\* node, int val) {

if (node == NULL)

return new Node(val);

if (val < node->data)

node->left = insert(node->left, val);

else

node->right = insert(node->right, val);

return node;

}

// Helper for in-order traversal

void inorder(Node\* node) {

if (node) {

inorder(node->left);

cout << node->data << " ";

inorder(node->right);

}

}

// Helper for deleting all nodes

void deleteAll(Node\* node) {

if (node) {

deleteAll(node->left);

deleteAll(node->right);

delete node;

}

}

public:

// Constructor to read from file

BST() {

root = NULL;

ifstream file("input.txt");

if (!file) {

cout << "Error: Cannot open input.txt\n";

exit(1);

}

int val;

while (file >> val) {

root = insert(root, val);

}

file.close();

}

// In-order traversal

void inorder() {

if (!root)

cout << "Tree is empty.\n";

else

inorder(root);

cout << endl;

}

// Delete all nodes

void emptyIt() {

deleteAll(root);

root = NULL;

cout << "Tree has been emptied.\n";

}

};

// ----------- DRIVER CODE ------------

int main() {

BST tree;

cout << "In-order Traversal of BST:\n";

tree.inorder();

cout << "\nEmptying the BST...\n";

tree.emptyIt();

cout << "Trying to traverse after emptying:\n";

tree.inorder();

return 0;

}

Implement a Binary Search Tree (BST) Class of integers using Linked Representation having

following operations:

A. BST(.. ) // constructor to populate BST by generating n random numbers in the range 1 to

100

B. In-order() // inorder traversal of the BST.

C. EmptyIt() // function to delete all nodes of a tree

#include <iostream>

#include <fstream>

#include <cstdlib>

#include <ctime>

using namespace std;

struct Node {

int data;

Node\* left;

Node\* right;

Node(int val) : data(val), left(NULL), right(NULL) {}

};

class BST {

private:

Node\* root;

// Helper function for inserting nodes in BST

Node\* insert(Node\* node, int val) {

if (node == NULL) {

return new Node(val);

}

if (val < node->data) {

node->left = insert(node->left, val);

} else {

node->right = insert(node->right, val);

}

return node;

}

// In-order traversal (recursive)

void inorder(Node\* node) {

if (node) {

inorder(node->left);

cout << node->data << " ";

inorder(node->right);

}

}

// Helper function to delete all nodes

void deleteAll(Node\* node) {

if (node) {

deleteAll(node->left);

deleteAll(node->right);

delete node;

}

}

public:

// Constructor to generate n random numbers and populate the BST

BST(int n) {

root = NULL;

srand(time(0));

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

int randomValue = rand() % 100 + 1; // Random numbers between 1 and 100

root = insert(root, randomValue);

}

}

// In-order traversal

void inorder() {

if (!root) {

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

return;

}

inorder(root);

cout << endl;

}

// Delete all nodes of the tree

void emptyIt() {

deleteAll(root);

root = NULL;

cout << "Tree has been emptied." << endl;

}

};

int main() {

int n;

cout << "Enter number of elements to insert in the BST: ";

cin >> n;

BST bst(n);

cout << "\nIn-order traversal of BST: ";

bst.inorder();

cout << "\nEmptying the BST...\n";

bst.emptyIt();

return 0;

}

Implement a Binary Search Tree (BST) Class of integers using Linked Representation having

following operations:

A. BST(.. ) // constructor to populate BST by generating n random numbers in the range 1 to

100

B. Post-order() // post-order traversal of the BST.

C. SumLeaves() // function to return the sum of leaf nodes.

#include <iostream>

#include <cstdlib>

#include <ctime>

using namespace std;

struct Node {

int data;

Node\* left;

Node\* right;

Node(int val) : data(val), left(NULL), right(NULL) {}

};

class BST {

private:

Node\* root;

// Helper function for inserting nodes in BST

Node\* insert(Node\* node, int val) {

if (node == NULL) {

return new Node(val);

}

if (val < node->data) {

node->left = insert(node->left, val);

} else {

node->right = insert(node->right, val);

}

return node;

}

// Post-order traversal (recursive)

void postorder(Node\* node) {

if (node) {

postorder(node->left);

postorder(node->right);

cout << node->data << " ";

}

}

// Calculate sum of leaf nodes

int sumLeaves(Node\* node) {

if (node == NULL) return 0;

if (node->left == NULL && node->right == NULL) return node->data;

return sumLeaves(node->left) + sumLeaves(node->right);

}

public:

// Constructor to generate n random numbers and populate the BST

BST(int n) {

root = NULL;

srand(time(0));

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

int randomValue = rand() % 100 + 1; // Random numbers between 1 and 100

root = insert(root, randomValue);

}

}

// Post-order traversal

void postorder() {

if (!root) {

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

return;

}

postorder(root);

cout << endl;

}

// Sum of leaf nodes

int sumLeaves() {

return sumLeaves(root);

}

};

int main() {

int n;

cout << "Enter number of elements to insert in the BST: ";

cin >> n;

BST bst(n);

cout << "\nPost-order traversal of BST: ";

bst.postorder();

cout << "\nSum of leaf nodes: " << bst.sumLeaves() << endl;

return 0;

}

Max heap

#include <iostream>

#include <climits>

using namespace std;

class MaxHeap {

private:

int\* heap;

int capacity;

int size;

// Helper function to maintain heap property

void heapify(int index) {

int left = 2 \* index + 1;

int right = 2 \* index + 2;

int largest = index;

if (left < size && heap[left] > heap[largest]) {

largest = left;

}

if (right < size && heap[right] > heap[largest]) {

largest = right;

}

if (largest != index) {

swap(heap[index], heap[largest]);

heapify(largest);

}

}

public:

MaxHeap(int cap) {

capacity = cap;

size = 0;

heap = new int[capacity];

}

// Insert operation

void insert(int value) {

if (size == capacity) {

cout << "Heap is full!" << endl;

return;

}

size++;

int index = size - 1;

heap[index] = value;

while (index > 0 && heap[(index - 1) / 2] < heap[index]) {

swap(heap[index], heap[(index - 1) / 2]);

index = (index - 1) / 2;

}

}

// Delete operation (remove root)

void remove() {

if (size == 0) {

cout << "Heap is empty!" << endl;

return;

}

heap[0] = heap[size - 1];

size--;

heapify(0);

}

// Display the heap

void display() {

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

cout << heap[i] << " ";

}

cout << endl;

}

~MaxHeap() {

delete[] heap;

}

};

int main() {

MaxHeap heap(10);

heap.insert(40);

heap.insert(20);

heap.insert(60);

heap.insert(10);

heap.insert(30);

cout << "Max Heap: ";

heap.display();

cout << "Removing root...\n";

heap.remove();

cout << "Max Heap after removal: ";

heap.display();

return 0;

}

Min heap

#include <iostream>

#include <climits>

using namespace std;

class MinHeap {

private:

int\* heap;

int capacity;

int size;

// Helper function to maintain heap property

void heapify(int index) {

int left = 2 \* index + 1;

int right = 2 \* index + 2;

int smallest = index;

if (left < size && heap[left] < heap[smallest]) {

smallest = left;

}

if (right < size && heap[right] < heap[smallest]) {

smallest = right;

}

if (smallest != index) {

swap(heap[index], heap[smallest]);

heapify(smallest);

}

}

public:

MinHeap(int cap) {

capacity = cap;

size = 0;

heap = new int[capacity];

}

// Insert operation

void insert(int value) {

if (size == capacity) {

cout << "Heap is full!" << endl;

return;

}

size++;

int index = size - 1;

heap[index] = value;

while (index > 0 && heap[(index - 1) / 2] > heap[index]) {

swap(heap[index], heap[(index - 1) / 2]);

index = (index - 1) / 2;

}

}

// Delete operation (remove root)

void remove() {

if (size == 0) {

cout << "Heap is empty!" << endl;

return;

}

heap[0] = heap[size - 1];

size--;

heapify(0);

}

// Display the heap

void display() {

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

cout << heap[i] << " ";

}

cout << endl;

}

~MinHeap() {

delete[] heap;

}

};

int main() {

MinHeap heap(10);

heap.insert(40);

heap.insert(20);

heap.insert(60);

heap.insert(10);

heap.insert(30);

cout << "Min Heap: ";

heap.display();

cout << "Removing root...\n";

heap.remove();

cout << "Min Heap after removal: ";

heap.display();

return 0;

}

#include <iostream>

#include <algorithm>

using namespace std;

class Node {

public:

int key;

Node\* left;

Node\* right;

int height;

Node(int val) {

key = val;

left = right = nullptr;

height = 1;

}

};

class AVLTree {

private:

Node\* root;

int height(Node\* node) {

return node ? node->height : 0;

}

int getBalance(Node\* node) {

return node ? height(node->left) - height(node->right) : 0;

}

Node\* rotateRight(Node\* y) {

cout << "Performing Right Rotation (LL)\n";

Node\* x = y->left;

Node\* T2 = x->right;

x->right = y;

y->left = T2;

y->height = max(height(y->left), height(y->right)) + 1;

x->height = max(height(x->left), height(x->right)) + 1;

return x;

}

Node\* rotateLeft(Node\* x) {

cout << "Performing Left Rotation (RR)\n";

Node\* y = x->right;

Node\* T2 = y->left;

y->left = x;

x->right = T2;

x->height = max(height(x->left), height(x->right)) + 1;

y->height = max(height(y->left), height(y->right)) + 1;

return y;

}

Node\* insert(Node\* node, int key) {

if (!node)

return new Node(key);

if (key < node->key)

node->left = insert(node->left, key);

else if (key > node->key)

node->right = insert(node->right, key);

else

return node; // no duplicates

node->height = 1 + max(height(node->left), height(node->right));

int balance = getBalance(node);

// Left Left

if (balance > 1 && key < node->left->key)

return rotateRight(node);

// Right Right

if (balance < -1 && key > node->right->key)

return rotateLeft(node);

// Left Right

if (balance > 1 && key > node->left->key) {

cout << "Performing Left-Right Rotation (LR)\n";

node->left = rotateLeft(node->left);

return rotateRight(node);

}

// Right Left

if (balance < -1 && key < node->right->key) {

cout << "Performing Right-Left Rotation (RL)\n";

node->right = rotateRight(node->right);

return rotateLeft(node);

}

return node;

}

void inorder(Node\* node) {

if (node) {

inorder(node->left);

cout << "Key: " << node->key

<< ", Balance Factor: " << getBalance(node) << endl;

inorder(node->right);

}

}

public:

AVLTree() {

root = nullptr;

}

void insert(int key) {

root = insert(root, key);

}

void displayInOrderWithBalance() {

if (!root) {

cout << "Tree is empty.\n";

} else {

cout << "\nInorder Traversal with Balance Factors:\n";

inorder(root);

}

}

};

// ------------------ Driver Code ------------------

int main() {

AVLTree tree;

int choice, key;

do {

cout << "\n===== AVL Tree Menu =====\n";

cout << "1. Insert\n";

cout << "2. Display In-order with Balance Factors\n";

cout << "3. Exit\n";

cout << "Enter choice: ";

cin >> choice;

switch (choice) {

case 1:

cout << "Enter integer to insert: ";

cin >> key;

tree.insert(key);

break;

case 2:

tree.displayInOrderWithBalance();

break;

case 3:

cout << "Exiting...\n";

break;

default:

cout << "Invalid choice. Try again.\n";

}

} while (choice != 3);

return 0;

}

BTREE

#include <iostream>

#include <queue>

using namespace std;

class BTreeNode {

public:

int keys[3]; // Maximum 2 keys for order 3

BTreeNode\* children[4]; // Max children = Order (3) + 1

int numKeys;

bool isLeaf;

BTreeNode(bool leaf) {

isLeaf = leaf;

numKeys = 0;

for (int i = 0; i < 4; i++) children[i] = nullptr;

}

};

class BTree {

public:

BTreeNode\* root;

BTree() {

root = new BTreeNode(true);

}

void insert(int key) {

BTreeNode\* r = root;

if (r->numKeys == 2) {

BTreeNode\* s = new BTreeNode(false);

root = s;

s->children[0] = r;

splitChild(s, 0, r);

insertNonFull(s, key);

} else {

insertNonFull(r, key);

}

}

void levelOrderDisplay() {

if (!root) return;

queue<BTreeNode\*> q;

q.push(root);

while (!q.empty()) {

BTreeNode\* node = q.front();

q.pop();

for (int i = 0; i < node->numKeys; i++)

cout << node->keys[i] << " ";

cout << "| ";

if (!node->isLeaf) {

for (int i = 0; i <= node->numKeys; i++) {

if (node->children[i])

q.push(node->children[i]);

}

}

}

cout << endl;

}

void remove(int key) {

root = deleteKey(root, key);

}

private:

void insertNonFull(BTreeNode\* node, int key) {

int i = node->numKeys - 1;

if (node->isLeaf) {

while (i >= 0 && key < node->keys[i]) {

node->keys[i + 1] = node->keys[i];

i--;

}

node->keys[i + 1] = key;

node->numKeys++;

} else {

while (i >= 0 && key < node->keys[i]) i--;

i++;

if (node->children[i]->numKeys == 2) {

splitChild(node, i, node->children[i]);

if (key > node->keys[i]) i++;

}

insertNonFull(node->children[i], key);

}

}

void splitChild(BTreeNode\* parent, int i, BTreeNode\* child) {

BTreeNode\* newNode = new BTreeNode(child->isLeaf);

newNode->keys[0] = child->keys[2]; // Move the last key

if (!child->isLeaf) {

newNode->children[0] = child->children[2];

newNode->children[1] = child->children[3];

}

child->numKeys = 1; // Only keep the first key in original node

for (int j = parent->numKeys; j > i; j--) {

parent->children[j + 1] = parent->children[j];

parent->keys[j] = parent->keys[j - 1];

}

parent->keys[i] = child->keys[1];

parent->children[i + 1] = newNode;

parent->numKeys++;

}

BTreeNode\* deleteKey(BTreeNode\* node, int key) {

if (!node) return nullptr;

int i;

for (i = 0; i < node->numKeys && key > node->keys[i]; i++);

if (i < node->numKeys && key == node->keys[i]) {

if (node->isLeaf) {

for (int j = i; j < node->numKeys - 1; j++)

node->keys[j] = node->keys[j + 1];

node->numKeys--;

}

// Note: For simplicity, this delete only supports leaf deletion.

return node;

}

if (!node->isLeaf) {

node->children[i] = deleteKey(node->children[i], key);

}

return node;

}

};

int main() {

BTree tree;

int choice, key;

do {

cout << "\nB-Tree Operations:\n";

cout << "1. Insert\n2. Level order display\n3. Delete\n4. Exit\n";

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case 1:

cout << "Enter key to insert: ";

cin >> key;

tree.insert(key);

break;

case 2:

cout << "B-Tree Level Order: ";

tree.levelOrderDisplay();

break;

case 3:

cout << "Enter key to delete: ";

cin >> key;

tree.remove(key);

break;

case 4:

cout << "Exiting...\n";

break;

default:

cout << "Invalid choice. Try again.\n";

}

} while (choice != 4);

return 0;

}

Non recursive

Level order

#include<iostream>

#include<queue>

using namespace std;

struct Node {

int key;

string value;

Node\* left;

Node\* right;

Node(int k, string val) {

key = k;

value = val;

left = right = nullptr;

}

};

void LevelOrder(Node \*root){

if (!root) return;

queue<Node \*> q;

q.push(root);

while(!q.empty()){

Node \*temp = q.front();

q.pop();

cout<<temp->key<<temp->value;

if(temp->left)

q.push(temp->left);

if(temp->right)

q.push(temp->right);

}

}

int main(){

return 0;

}

Inorder

#include<iostream>

#include<queue>

#include<stack>

using namespace std;

struct Node {

int key;

string value;

Node\* left;

Node\* right;

Node(int k, string val) {

key = k;

value = val;

left = right = nullptr;

}

};

void inorder(Node \*root){

stack<Node \*> st;

Node \*curr = root;

while(curr!=NULL || !st.empty()){

while(curr!=NULL){

st.push(curr);

curr = curr->left;

}

curr = st.top();

st.pop();

cout<<curr->value;

curr = curr->right;

}

}

int main(){

return 0;

}

Preorder

#include<iostream>

#include<queue>

#include<stack>

using namespace std;

struct Node {

int key;

string value;

Node\* left;

Node\* right;

Node(int k, string val) {

key = k;

value = val;

left = right = nullptr;

}

};

void preorder(Node \*root){

if (root == NULL) return;

stack<Node \*> st;

st.push(root);

while(!st.empty()){

Node \*curr = st.top();

st.pop();

cout<<curr->value;

if(curr->right != NULL){

st.push(curr->right);

}

if(curr->left != NULL){

st.push(curr->left);

}

}

}

int main(){

return 0;

}

Postorder

#include<iostream>

#include<queue>

#include<stack>

using namespace std;

struct Node {

int key;

string value;

Node\* left;

Node\* right;

Node(int k, string val) {

key = k;

value = val;

left = right = nullptr;

}

};

void postorder(Node \*root){

if (root == NULL) return;

stack<Node \*> st1, st2;

st1.push(root);

while(!st1.empty()){

Node \*curr = st1.top();

st1.pop();

st2.push(curr);

if(curr->right != NULL){

st1.push(curr->right);

}

if(curr->left != NULL){

st1.push(curr->left);

}

}

while(!st2.empty()){

cout<<st2.top()->value;

st2.pop();

}

}

int main(){

return 0;

}