National University of Computer and Emerging Sciences



Name: Muhammad Suleman

Roll #: 22F-3350

Section: BCS-4B

Lab # 11

# Problem 01

#include <iostream>

using namespace std;

template <typename T>

struct Node

{

T data;

Node<T> \*left;

Node<T> \*right;

int height;

Node(T value)

: data(value), left(nullptr), right(nullptr), height(0) {}

};

template <typename T>

class AVLTree

{

public:

AVLTree()

: root(nullptr) {}

void insert(T value)

{

root = insertNode(root, value);

}

bool search(T value)

{

return searchNode(root, value);

}

void inorderTraversal()

{

inorderTraversalNode(root);

cout << endl;

}

void preorderTraversal()

{

preorderTraversalNode(root);

cout << endl;

}

void postorderTraversal()

{

postorderTraversalNode(root);

cout << endl;

}

T findMin()

{

Node<T> \*minNode = findMinNode(root);

if (minNode)

{

return minNode->data;

}

return T();

}

T findSecondMin()

{

Node<T> \*secondMinNode = findSecondMinNode(root);

if (secondMinNode)

{

return secondMinNode->data;

}

return T();

}

bool isAVL()

{

return isAVLTree(root);

}

void deleteNode(T value)

{

root = deleteNode(root, value);

}

Node<T> \*deleteNode(Node<T> \*node, T value)

{

if (node == nullptr)

return nullptr;

if (value < node->data)

{

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

}

else if (value > node->data)

{

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

}

else

{

if (node->left == nullptr)

{

Node<T> \*temp = node->right;

delete node;

return temp;

}

else if (node->right == nullptr)

{

Node<T> \*temp = node->left;

delete node;

return temp;

}

node->data = minValueNode(node->right)->data;

node->right = deleteNode(node->right, node->data);

}

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

int balanceFactor = getBalanceFactor(node);

if (balanceFactor > 1 && value < node->left->data)

{

return rotateRight(node);

}

if (balanceFactor < -1 && value > node->right->data)

{

return rotateLeft(node);

}

if (balanceFactor > 1 && value > node->left->data)

{

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

return rotateRight(node);

}

if (balanceFactor < -1 && value < node->right->data)

{

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

return rotateLeft(node);

}

return node;

}

Node<T> \*minValueNode(Node<T> \*node)

{

if (node->left == nullptr || node == nullptr)

{

return node;

}

return minValueNode(node->left);

}

private:

Node<T> \*root;

int getHeight(Node<T> \*node)

{

if (node == nullptr)

{

return -1;

}

return node->height;

}

int getMax(int a, int b)

{

if (a > b)

{

return a;

}

return b;

}

int getBalanceFactor(Node<T> \*node)

{

if (node == nullptr)

{

return -1;

}

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

}

Node<T> \*rotateRight(Node<T> \*y)

{

Node<T> \*x = y->left;

Node<T> \*t2 = x->right;

x->right = y;

y->left = t2;

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

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

return x;

}

Node<T> \*rotateLeft(Node<T> \*x)

{

Node<T> \*y = x->right;

Node<T> \*t2 = y->left;

y->left = x;

x->right = t2;

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

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

return y;

}

Node<T> \*insertNode(Node<T> \*node, T value)

{

if (node == nullptr)

{

return new Node<T>(value);

}

if (value < node->data)

{

node->left = insertNode(node->left, value);

}

else if (value > node->data)

{

node->right = insertNode(node->right, value);

}

else

{

return node;

}

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

int balanceFactor = getBalanceFactor(node);

if (balanceFactor > 1 && value < node->left->data)

{

return rotateRight(node);

}

if (balanceFactor < -1 && value > node->right->data)

{

return rotateLeft(node);

}

if (balanceFactor > 1 && value > node->left->data)

{

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

return rotateRight(node);

}

if (balanceFactor < -1 && value < node->right->data)

{

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

return rotateLeft(node);

}

return node;

}

bool isAVLTree(Node<T> \*node)

{

if (node == nullptr)

{

return true;

}

int balanceFactor = getBalanceFactor(node);

if (balanceFactor < -1 || balanceFactor > 1)

{

return false;

}

return isAVLTree(node->left) && isAVLTree(node->right);

}

bool searchNode(Node<T> \*node, T value)

{

if (node == nullptr)

{

return false;

}

if (value < node->data)

{

return searchNode(node->left, value);

}

else if (value > node->data)

{

return searchNode(node->right, value);

}

else

{

return true;

}

}

Node<T> \*findMinNode(Node<T> \*node)

{

if (node == nullptr || node->left == nullptr)

{

return node;

}

return findMinNode(node->left);

}

Node<T> \*findSecondMinNode(Node<T> \*node)

{

if (node == nullptr || (node->left == nullptr && node->right == nullptr))

{

return nullptr;

}

Node<T> \*current = node;

Node<T> \*parent = nullptr;

while (current->left != nullptr)

{

parent = current;

current = current->left;

}

if (current->right != nullptr)

{

return findMinNode(current->right);

}

return parent;

}

void inorderTraversalNode(Node<T> \*node)

{

if (node == nullptr)

{

return;

}

inorderTraversalNode(node->left);

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

inorderTraversalNode(node->right);

}

void preorderTraversalNode(Node<T> \*node)

{

if (node == nullptr)

{

return;

}

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

preorderTraversalNode(node->left);

preorderTraversalNode(node->right);

}

void postorderTraversalNode(Node<T> \*node)

{

if (node == nullptr)

{

return;

}

postorderTraversalNode(node->left);

postorderTraversalNode(node->right);

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

}

};

int main()

{

AVLTree<int> avlTree;

for (int i = 1; i <= 15; i++)

{

avlTree.insert(i);

}

cout << "Inorder traversal before deletion: ";

avlTree.inorderTraversal();

cout << "Preorder traversal before deletion: ";

avlTree.preorderTraversal();

cout << "Postorder traversal before deletion: ";

avlTree.postorderTraversal();

cout << "Searching for 25: ";

if (avlTree.search(25))

{

cout << "Found" << endl;

}

else

{

cout << "Not found" << endl;

}

cout << "Searching for 1: ";

if (avlTree.search(1))

{

cout << "Found" << endl;

}

else

{

cout << "Not found" << endl;

}

cout << "Minimum element: " << avlTree.findMin() << endl;

cout << "Second minimum element: " << avlTree.findSecondMin() << endl;

cout << "IsAVL tree: ";

if (avlTree.isAVL())

{

cout << "AVL tree" << endl;

}

else

{

cout << "Not AVL tree" << endl;

}

// Deletion example

cout << "Deleting element 8: ";

avlTree.deleteNode(8);

cout << "Inorder traversal after deletion: ";

avlTree.inorderTraversal();

cout << "Preorder traversal after deletion: ";

avlTree.preorderTraversal();

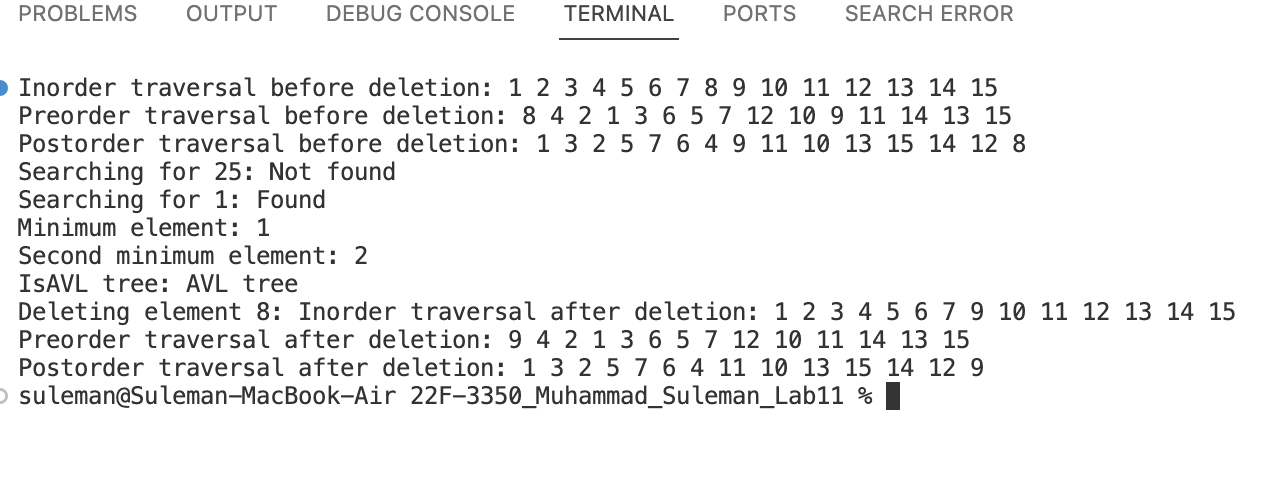
cout << "Postorder traversal after deletion: ";

avlTree.postorderTraversal();

return 0;

}

## Output



# Problem 02

## Using Binary Search Tree (BST)

#include <iostream>

#include <ctime>

#include <chrono>

#include <cstdlib>

#include <string>

using namespace std;

using namespace std::chrono;

struct Node

{

string word;

string definition;

Node \*left;

Node \*right;

Node()

: word("\0"), definition("\0"), left(nullptr), right(nullptr) {}

Node(string word, string definition)

: word(word), definition(definition), left(nullptr), right(nullptr) {}

};

class BinarySearchTree

{

public:

Node \*root;

private:

void destroyTree(Node \*node);

Node \*insert(Node \*root, string word, string definition);

Node \*find(Node \*root, string word);

void inOrder(Node \*root);

void preorderorderPrint(Node \*root);

void postorderPrint(Node \*root);

Node \*minValueNode(Node \*root);

Node \*deleteNode(Node \*root, string word);

public:

BinarySearchTree()

: root(nullptr) {}

~BinarySearchTree()

{

destroyTree(root);

}

void insert(string word, string definition);

Node \*find(string word);

void inOrder();

void preorderorderPrint();

void postorderPrint();

void deleteNode(string word);

};

int main()

{

BinarySearchTree BST;

auto start = chrono::high\_resolution\_clock::now();

BST.insert("algorithm", "a step by step procedure for solving a problem");

BST.insert("ADT", "user defined classes");

BST.insert("recursion", "function calling itself in its definition");

BST.insert("binary search", "a searching algorithm");

BST.insert("abc", "a");

BST.insert("bcd", "b");

BST.insert("cde", "c");

BST.insert("fdfd", "a");

BST.insert("dsds", "b");

BST.insert("cdsdde", "c");

BST.insert("abcsds", "a");

BST.insert("bcdsd", "b");

BST.insert("cdedew", "c");

auto end = chrono::high\_resolution\_clock::now();

auto duration = duration\_cast<nanoseconds>(end - start);

cout << "Execution time of insertion in BST: " << duration.count() << " nanoseconds." << endl;

cout << "Searching for \"algorithm\" (BST Tree)" << endl;

Node \*result = BST.find("algorithm");

if (result)

{

cout << "Word \"" << result->word << "\" Found" << endl;

cout << "Definition of \"algorithm\": " << result->definition << endl;

}

else

{

cout << "Term not found in the dictionary" << endl;

}

cout << endl;

cout << "Searching for stack:" << endl;

result = BST.find("stack");

if (result)

{

cout << "Word \"" << result->word << "\" Found" << endl;

cout << "Definition of \"stack\": " << result->definition << endl;

}

else

{

cout << "Term not found in the dictionary" << endl;

}

cout << endl;

cout << "Inorder traversal" << endl;

BST.inOrder();

cout << endl;

cout << "Deleting \"variable\" from the dictionary" << endl;

BST.deleteNode("variable");

cout << endl;

// cout << "Inorder traversal after deletion:" << endl;

// BST.inOrder();

// cout << endl;

return 0;

}

void BinarySearchTree::destroyTree(Node \*node)

{

if (node != nullptr)

{

destroyTree(node->left);

destroyTree(node->right);

delete node;

}

}

Node \*BinarySearchTree::insert(Node \*root, string word, string definition)

{

if (root == nullptr)

{

Node \*newNode = new Node(word, definition);

return newNode;

}

if (word < root->word)

{

root->left = insert(root->left, word, definition);

}

else if (word > root->word)

{

root->right = insert(root->right, word, definition);

}

return root;

}

Node \*BinarySearchTree::find(Node \*root, string word)

{

if (root == nullptr || root->word == word)

{

return root;

}

if (word < root->word)

{

return find(root->left, word);

}

else

{

return find(root->right, word);

}

}

void BinarySearchTree::inOrder(Node \*root)

{

if (root == nullptr)

{

return;

}

inOrder(root->left);

cout << "Word: " << root->word << endl;

cout << "Defination: " << root->definition << endl;

inOrder(root->right);

}

void BinarySearchTree::preorderorderPrint(Node \*root)

{

if (root == nullptr)

{

return;

}

cout << "Word: " << root->word << endl;

cout << "Defination: " << root->definition << endl;

preorderorderPrint(root->left);

preorderorderPrint(root->right);

}

void BinarySearchTree::postorderPrint(Node \*root)

{

if (root == nullptr)

{

return;

}

postorderPrint(root->left);

postorderPrint(root->right);

cout << "Word: " << root->word << endl;

cout << "Defination: " << root->definition << endl;

}

Node \*BinarySearchTree::minValueNode(Node \*root)

{

while (root != nullptr && root->left != nullptr)

{

root = root->left;

}

return root;

}

Node \*BinarySearchTree::deleteNode(Node \*root, string word)

{

if (root == nullptr)

{

return root;

}

else if (word < root->word)

{

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

}

else if (word > root->word)

{

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

}

else

{

if (root->left == nullptr)

{

Node \*temp = root->right;

delete root;

return temp;

}

else if (root->right == nullptr)

{

Node \*temp = root->left;

delete root;

return temp;

}

else

{

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

root->word = temp->word;

root->definition = temp->definition;

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

}

}

return root;

}

void BinarySearchTree::insert(string word, string definition)

{

root = insert(root, word, definition);

}

Node \*BinarySearchTree::find(string word)

{

return find(root, word);

}

void BinarySearchTree::inOrder()

{

inOrder(root);

}

void BinarySearchTree::preorderorderPrint()

{

preorderorderPrint(root);

}

void BinarySearchTree::postorderPrint()

{

postorderPrint(root);

}

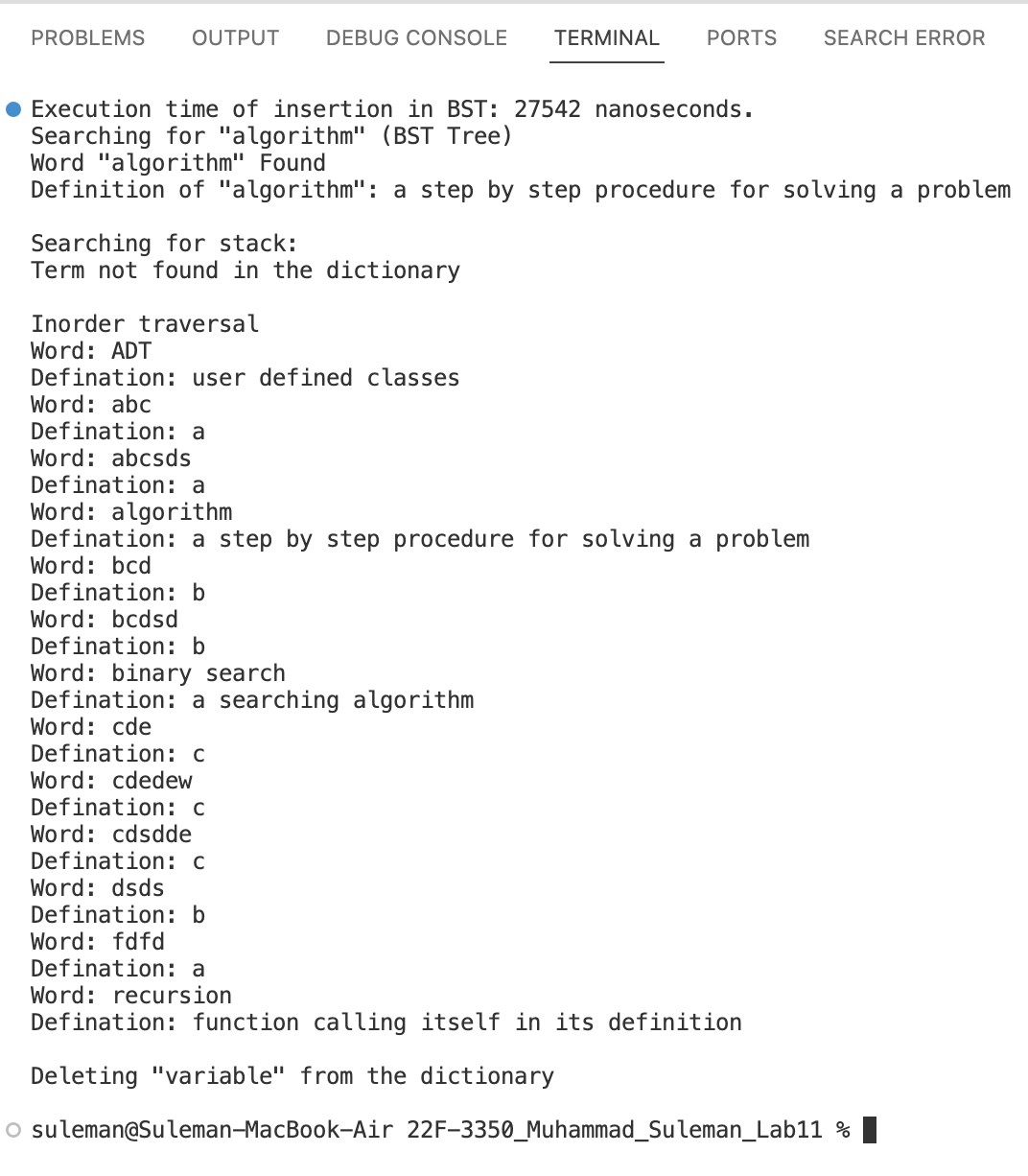
void BinarySearchTree::deleteNode(string word)

{

root = deleteNode(root, word);

}

## Output



## Using AVL Tree

#include <iostream>

#include <ctime>

#include <chrono>

#include <cstdlib>

#include <string>

using namespace std;

using namespace std::chrono;

struct Node

{

string word;

string definition;

Node \*left;

Node \*right;

int height;

Node() : word("\0"), definition("\0"), left(nullptr), right(nullptr), height(0) {}

Node(string word, string definition)

: word(word), definition(definition), left(nullptr), right(nullptr), height(0) {}

};

class AVLTree

{

public:

Node \*root;

private:

void destroyTree(Node \*node);

Node \*insert(Node \*root, string word, string definition);

Node \*find(Node \*root, string word);

void inOrder(Node \*root);

void preorderorderPrint(Node \*root);

void postorderPrint(Node \*root);

Node \*minValueNode(Node \*root);

Node \*deleteNode(Node \*root, string word);

Node \*rotateRight(Node \*y);

Node \*rotateLeft(Node \*x);

int getHeight(Node \*node);

int getBalanceFactor(Node \*node);

public:

AVLTree() : root(nullptr) {}

~AVLTree()

{

destroyTree(root);

}

void insert(string word, string definition);

Node \*find(string word);

void inOrder();

void preorderorderPrint();

void postorderPrint();

void deleteNode(string word);

};

int main()

{

AVLTree avlTree;

auto start = chrono::high\_resolution\_clock::now();

avlTree.insert("algorithm", "a step by step procedure for solving a problem");

avlTree.insert("ADT", "user defined classes");

avlTree.insert("recursion", "function calling itself in its definition");

avlTree.insert("binary search", "a searching algorithm");

avlTree.insert("abc", "a");

avlTree.insert("bcd", "b");

avlTree.insert("cde", "c");

avlTree.insert("fdfd", "a");

avlTree.insert("dsds", "b");

avlTree.insert("cdsdde", "c");

avlTree.insert("abcsds", "a");

avlTree.insert("bcdsd", "b");

avlTree.insert("cdedew", "c");

auto end = chrono::high\_resolution\_clock::now();

auto duration = duration\_cast<nanoseconds>(end - start);

cout << "Execution time of insertion in AVL: " << duration.count() << " nanoseconds." << endl;

cout << "Searching for \"algorithm\" (AVL Tree)" << endl;

Node \*result = avlTree.find("algorithm");

if (result)

{

cout << "Word \"" << result->word << "\" Found" << endl;

cout << "Definition of \"algorithm\": " << result->definition << endl;

}

else

{

cout << "Term not found in the dictionary" << endl;

}

cout << endl;

cout << "Searching for stack:" << endl;

result = avlTree.find("stack");

if (result)

{

cout << "Word \"" << result->word << "\" Found" << endl;

cout << "Definition of \"stack\": " << result->definition << endl;

}

else

{

cout << "Term not found in the dictionary" << endl;

}

cout << endl;

cout << "Inorder traversal" << endl;

avlTree.inOrder();

cout << endl;

cout << "Deleting \"variable\" from the dictionary" << endl;

avlTree.deleteNode("variable");

cout << endl;

// cout << "Inorder traversal after deletion:" << endl;

// avlTree.inOrder();

// cout << endl;

return 0;

}

void AVLTree::destroyTree(Node \*node)

{

if (node != nullptr)

{

destroyTree(node->left);

destroyTree(node->right);

delete node;

}

}

Node \*AVLTree::insert(Node \*root, string word, string definition)

{

if (root == nullptr)

{

Node \*newNode = new Node(word, definition);

return newNode;

}

if (word < root->word)

{

root->left = insert(root->left, word, definition);

}

else if (word > root->word)

{

root->right = insert(root->right, word, definition);

}

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

int balance = getBalanceFactor(root);

// left left case

if (balance > 1 && word < root->left->word)

return rotateRight(root);

// right right case

if (balance < -1 && word > root->right->word)

return rotateLeft(root);

// left right case

if (balance > 1 && word > root->left->word)

{

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

return rotateRight(root);

}

// right left case

if (balance < -1 && word < root->right->word)

{

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

return rotateLeft(root);

}

return root;

}

Node \*AVLTree::find(Node \*root, string word)

{

if (root == nullptr || root->word == word)

{

return root;

}

if (word < root->word)

{

return find(root->left, word);

}

else

{

return find(root->right, word);

}

}

void AVLTree::inOrder(Node \*root)

{

if (root == nullptr)

{

return;

}

inOrder(root->left);

cout << "Word: " << root->word << endl;

cout << "Definition: " << root->definition << endl;

inOrder(root->right);

}

void AVLTree::preorderorderPrint(Node \*root)

{

if (root == nullptr)

{

return;

}

cout << "Word: " << root->word << endl;

cout << "Definition: " << root->definition << endl;

preorderorderPrint(root->left);

preorderorderPrint(root->right);

}

void AVLTree::postorderPrint(Node \*root)

{

if (root == nullptr)

{

return;

}

postorderPrint(root->left);

postorderPrint(root->right);

cout << "Word: " << root->word << endl;

cout << "Definition: " << root->definition << endl;

}

Node \*AVLTree::minValueNode(Node \*root)

{

while (root != nullptr && root->left != nullptr)

{

root = root->left;

}

return root;

}

Node \*AVLTree::deleteNode(Node \*root, string word)

{

if (root == nullptr)

{

return root;

}

else if (word < root->word)

{

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

}

else if (word > root->word)

{

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

}

else

{

if (root->left == nullptr)

{

Node \*temp = root->right;

delete root;

return temp;

}

else if (root->right == nullptr)

{

Node \*temp = root->left;

delete root;

return temp;

}

else

{

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

root->word = temp->word;

root->definition = temp->definition;

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

}

}

if (root == nullptr)

{

return root;

}

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

int balance = getBalanceFactor(root);

// left left case

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

{

return rotateRight(root);

}

// left right case

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

{

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

return rotateRight(root);

}

// right right case

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

{

return rotateLeft(root);

}

// right left case

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

{

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

return rotateLeft(root);

}

return root;

}

void AVLTree::insert(string word, string definition)

{

root = insert(root, word, definition);

}

Node \*AVLTree::find(string word)

{

return find(root, word);

}

void AVLTree::inOrder()

{

inOrder(root);

}

void AVLTree::preorderorderPrint()

{

preorderorderPrint(root);

}

void AVLTree::postorderPrint()

{

postorderPrint(root);

}

void AVLTree::deleteNode(string word)

{

root = deleteNode(root, word);

}

Node \*AVLTree::rotateRight(Node \*y)

{

Node \*x = y->left;

Node \*T2 = x->right;

x->right = y;

y->left = T2;

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

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

return x;

}

Node \*AVLTree::rotateLeft(Node \*x)

{

Node \*y = x->right;

Node \*T2 = y->left;

y->left = x;

x->right = T2;

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

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

return y;

}

int AVLTree::getHeight(Node \*node)

{

if (node == nullptr)

{

return -1;

}

return node->height;

}

int AVLTree::getBalanceFactor(Node \*node)

{

if (node == nullptr)

{

return 0;

}

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

}

## Output

