* **Code:**

#include <iostream>

#include <algorithm>

using namespace std;

class AVLNode {

public:

int key;

int height;

AVLNode\* left;

AVLNode\* right;

AVLNode(int k) : key(k), height(1), left(NULL), right(NULL) {}

};

class AVLTree {

private:

AVLNode\* root;

int getHeight(AVLNode\* node) {

return (node == NULL) ? 0 : node->height;

}

int getBalanceFactor(AVLNode\* node) {

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

}

void updateHeight(AVLNode\* node) {

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

}

AVLNode\* rotateRight(AVLNode\* y) {

AVLNode\* x = y->left;

AVLNode\* T2 = x->right;

x->right = y;

y->left = T2;

updateHeight(y);

updateHeight(x);

return x;

}

AVLNode\* rotateLeft(AVLNode\* x) {

AVLNode\* y = x->right;

AVLNode\* T2 = y->left;

y->left = x;

x->right = T2;

updateHeight(x);

updateHeight(y);

return y;

}

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

if (node == NULL) {

return new AVLNode(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;

}

updateHeight(node);

int balance = getBalanceFactor(node);

if (balance > 1) {

if (key < node->left->key) {

return rotateRight(node);

} else {

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

return rotateRight(node);

}

}

if (balance < -1) {

if (key > node->right->key) {

return rotateLeft(node);

} else {

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

return rotateLeft(node);

}

}

return node;

}

AVLNode\* minValueNode(AVLNode\* node) {

AVLNode\* current = node;

while (current->left != NULL) {

current = current->left;

}

return current;

}

AVLNode\* deleteNode(AVLNode\* root, int key) {

if (root == NULL) {

return root;

}

if (key < root->key) {

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

} else if (key > root->key) {

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

} else {

// Node with only one child or no child

if (root->left == NULL) {

AVLNode\* temp = root->right;

delete root;

return temp;

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

AVLNode\* temp = root->left;

delete root;

return temp;

}

// Node with two children

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

root->key = temp->key;

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

}

updateHeight(root);

int balance = getBalanceFactor(root);

if (balance > 1) {

if (getBalanceFactor(root->left) >= 0) {

return rotateRight(root);

} else {

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

return rotateRight(root);

}

}

if (balance < -1) {

if (getBalanceFactor(root->right) <= 0) {

return rotateLeft(root);

} else {

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

return rotateLeft(root);

}

}

return root;

}

AVLNode\* search(AVLNode\* root, int key) {

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

return root;

}

if (key < root->key) {

return search(root->left, key);

}

return search(root->right, key);

}

void inorderTraversal(AVLNode\* root) {

if (root != NULL) {

inorderTraversal(root->left);

cout << root->key << " ";

inorderTraversal(root->right);

}

}

public:

AVLTree() : root(NULL) {}

void insert(int key) {

root = insert(root, key);

}

void remove(int key) {

root = deleteNode(root, key);

}

bool search(int key) {

return search(root, key) != NULL;

}

void printInorder() {

cout << "Inorder Traversal: ";

inorderTraversal(root);

cout << std::endl;

}

};

int main() {

AVLTree avl;

avl.insert(10);

avl.insert(20);

avl.insert(30);

avl.insert(40);

avl.insert(50);

avl.insert(60);

avl.printInorder();

avl.remove(20);

avl.printInorder();

std::cout << "Search for 10: " << (avl.search(10) ? "Found" : "Not Found") << std::endl;

std::cout << "Search for 20: " << (avl.search(20) ? "Found" : "Not Found") << std::endl;

return 0;

}

* **Output:**

