#include <iostream>

#include <algorithm> // for std::max

using namespace std;

//Node class

| class Node {  public:  int data, height;  Node\* left\_node;  Node\* right\_node;  Node(int val) {  data = val;  height = 1; // Initial height for a leaf node  left\_node = right\_node = nullptr;  }  }; |
| --- |

//AVL class

| //create AVL Class  class AVLTree {  //or  /\*  // Nested Node struct within AVLTree  struct Node {  int data, height;  Node\* left\_node;  Node\* right\_node;  Node(int val) : data(val), height(1), left\_node(nullptr), right\_node(nullptr) {}  };  \*/  public:  Node\* root;  AVLTree() {  root = nullptr;  }  //get height of a node   | int get\_height(Node\* node) {  if (node == nullptr)  return 0;  return node->height;  } | | --- |   //get balance factor   | int get\_balance\_factor(Node\* node) {  if (node == nullptr)  return 0;  return get\_height(node->left\_node) - get\_height(node->right\_node);  } | | --- |   //1- Right rotation   | Node\* LL\_rotation(Node\* node) {  Node\* child = node->left\_node;  node->left\_node = child->right\_node;  child->right\_node = node;  node->height = max(get\_height(node->left\_node), get\_height(node->right\_node)) + 1;  child->height = max(get\_height(child->left\_node), get\_height(child->right\_node)) + 1;  return child;  } | | --- |     //2- left rotation   | Node\* RR\_rotation(Node\* node) {  Node\* child = node->right\_node;  node->right\_node = child->left\_node;  child->left\_node = node;  node->height = max(get\_height(node->left\_node), get\_height(node->right\_node)) + 1;  child->height = max(get\_height(child->left\_node), get\_height(child->right\_node)) + 1;  return child;  } | | --- |   //add a new Node   | Node\* insert(Node\* node, int val) {  if (node == nullptr)  return new Node(val);  if (val < node->data) {  node->left\_node = insert(node->left\_node, val);  } else if (val > node->data) {  node->right\_node = insert(node->right\_node, val);  } else {  return node; // Duplicate values not allowed, return the same node  }  node->height = max(get\_height(node->left\_node), get\_height(node->right\_node)) + 1;  int balance = get\_balance\_factor(node);  // Left-Left (LL) case  if (balance > 1 && val < node->left\_node->data)  return LL\_rotation(node);  // Right-Right (RR) case  if (balance < -1 && val > node->right\_node->data)  return RR\_rotation(node);  // Left-Right (LR) case  if (balance > 1 && val > node->left\_node->data) {  node->left\_node = RR\_rotation(node->left\_node);  return LL\_rotation(node);  }  // Right-Left (RL) case  if (balance < -1 && val < node->right\_node->data) {  node->right\_node = LL\_rotation(node->right\_node);  return RR\_rotation(node);  }  return node;  } | | --- |   //in order traversal   | void in\_order(Node\* node) {  if (node != nullptr) {  in\_order(node->left\_node); // Traverse the left subtree  cout << node->data << " "; // Visit the root (current node)  in\_order(node->right\_node); // Traverse the right subtree  }  } | | --- |   //inorder successor (smallest value from left subtree)   | Node\* min\_value\_node(Node\* node) {  Node\* current = node;  while (current->left\_node != nullptr)  current = current->left\_node;  return current;  } | | --- |   //delete a node   | Node\* delete\_node(Node\* root, int key) {  if (root == nullptr)  return root;  if (key < root->data) {  root->left\_node = delete\_node(root->left\_node, key);  } else if (key > root->data) {  root->right\_node = delete\_node(root->right\_node, key);  } else {  if ((root->left\_node == nullptr) || (root->right\_node == nullptr)) {  Node\* temp = root->left\_node ? root->left\_node : root->right\_node;  if (temp == nullptr) {  temp = root;  root = nullptr;  } else {  \*root = \*temp;  }  delete temp;  } else {  Node\* temp = min\_value\_node(root->right\_node);  root->data = temp->data;  root->right\_node = delete\_node(root->right\_node, temp->data);  }  }  if (root == nullptr)  return root;  root->height = max(get\_height(root->left\_node), get\_height(root->right\_node)) + 1;  int balance = get\_balance\_factor(root);  if (balance > 1 && get\_balance\_factor(root->left\_node) >= 0)  return LL\_rotation(root);  if (balance > 1 && get\_balance\_factor(root->left\_node) < 0) {  root->left\_node = RR\_rotation(root->left\_node);  return LL\_rotation(root);  }  if (balance < -1 && get\_balance\_factor(root->right\_node) <= 0)  return RR\_rotation(root);  if (balance < -1 && get\_balance\_factor(root->right\_node) > 0) {  root->right\_node = LL\_rotation(root->right\_node);  return RR\_rotation(root);  }  return root;  } | | --- |   }; |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |

| int main() {  AVLTree tree;  tree.root = tree.insert(tree.root, 17);  tree.root = tree.insert(tree.root, 18);  tree.root = tree.insert(tree.root, 19);  tree.root = tree.insert(tree.root, 11);  tree.root = tree.insert(tree.root, 10);  tree.root = tree.insert(tree.root, 16);  cout << "in\_order Traversal of the AVL Tree before deletion: ";  tree.in\_order(tree.root);  cout << endl;  tree.root = tree.delete\_node(tree.root, 18);  cout << "in\_order Traversal of the AVL Tree after deletion: ";  tree.in\_order(tree.root);  cout << endl;  cout << "In-order Traversal of the AVL Tree after deletion: ";  tree.in\_order(tree.root);  cout << endl;  return 0;  } |
| --- |