#ifndef BALANCED\_TREE\_H

#define BALANCED\_TREE\_H

#include<iostream>

#include <string>

using namespace std;

struct node {

int information\_node = 0;

node\* left;

node\* right;

node(int value) {

information\_node = value;

left = nullptr;

right = nullptr;

}

~node() {

information\_node = NULL;

delete left, right;

}

};

class Tree {

private:

node\* root;

public:

Tree(node\* root) {

this->root = root;

}

void createTree(node\* root,int n) {

int nl, nr, x;

if (n == 0) return;

nl = n / 2;

if (nl == 0) return;

root->left = new node(rand()%1000+100);

createTree(root->left,nl);

nr = n - nl - 1;

if (nr == 0) return;

root->right = new node(rand() % 1000 + 100);

createTree(root->right, nr);

}

void reverseTreeRightToLeft(node\* root) {

if (root == nullptr) return;

node\* tmp = root->left;

root->left = root->right;

root->right = tmp;

reverseTreeRightToLeft(root->left);

reverseTreeRightToLeft(root->right);

}

void print( node \* root, int level) {

if (root == nullptr) return;

print( root->right, level + 1);

for (int i = 0; i < level; i++) cout << "\t\t";

cout << " " << root->information\_node << "\n";

print(root->left, level + 1);

}

int countTreeHeight(node\* root) {

int heightLeftTree, heightRightTree, heightRootTree = 0;

if (root != nullptr) {

heightLeftTree = countTreeHeight(root->left);

heightRightTree = countTreeHeight(root->right);

heightRootTree = ((heightLeftTree > heightRightTree) ? heightLeftTree : heightRightTree) + 1;

}

return heightRootTree;

}

int getTreeLength(node\* root) {

int length = 0;

if (root->left != nullptr)

length += 1 + getTreeLength(root->left);

if (root->right != nullptr)

length += 1 + getTreeLength(root->right);

return length;

}

void add(node\* root) {

if (root == nullptr) return;

node\* placeToAdd = findNodeToAdd(root);

if (placeToAdd->left == nullptr) placeToAdd->left = new node(rand() % 1000 + 100);

else placeToAdd->right = new node(rand() % 1000 + 100);

}

node\* findNodeToAdd(node\* root) {

int lengthRightPart, lengthLeftPart;

if (root->left == nullptr || root->right == nullptr) {

return root;

}

lengthLeftPart = getTreeLength(root->left);

lengthRightPart = getTreeLength(root->right);

if (lengthLeftPart == lengthRightPart) {

node\* currentNode = root;

while (currentNode->left != nullptr) {

currentNode = currentNode->left;

}

return currentNode;

}

return (lengthLeftPart < lengthRightPart) ? findNodeToAdd(root->left) : findNodeToAdd(root->right);

}

double getAverage() {

return ((double)getValuesSum(this->root)) / (getTreeLength(this->root) + 1);

}

node\* getRoot() {

return this->root;

}

int getValuesSum(node\* root) {

return (root == nullptr) ? 0 : root->information\_node + getValuesSum(root->left) + getValuesSum(root->right);

}

~Tree() {

delete root;

}

};

#endif