/\*

\* assign4\_test.cc

\* Assignment 4 (BST) test runner.

\*/

#include <algorithm>

#include <cassert>

#include <iostream>

#include <random>

#include <set>

#include <vector>

using std::cout;

std::vector<unsigned> make\_random\_permutation(

std::size\_t len,

int seed = 1)

{

std::default\_random\_engine generator(seed);

std::vector<unsigned> ret(len, 0);

// Initialize vector to 0...len-1

for(std::size\_t i = 0; i < len; ++i)

ret.at(i) = i;

std::shuffle(ret.begin(), ret.end(), generator);

return ret;

}

// Node structure

struct node {

int key;

node\* left;

node\* right;

node\* parent;

};

/\*

\* User-implemented functions

\*/

void rotate(node\* child, node\* parent); // Rotation

bool find(node\*& root, int value); // Search

node\* insert(node\* root, int value); // Insertion

//node\* remove(node\* root, int value); // Deletion

node\* splay(node\* t); // Splay

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Tree structure checking

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// Balance measurement, returns a balance factor between 0 (not possible) and 1

// (perfectly balanced).

float balance(node\* root) {

if(!root)

return 1.0; // Empty tree is perfectly balanced

else if(!root->left) {

// One subtree, on the right

return 0.5 \* balance(root->left);

}

else if(!root->right) {

return 0.5 \* balance(root->right);

}

else // Two subtrees

return (balance(root->right) + balance(root->left)) / 2;

}

// Safe find, that does not modify the tree structure

bool safe\_find(node\* root, int value) {

if(!root)

return false;

else if(root->key == value)

return true;

else if(value < root->key)

return safe\_find(root->left, value);

else // value < root->key

return safe\_find(root->right, value);

}

int count\_nodes(node\* root) {

if(!root)

return 0;

else

return 1 + count\_nodes(root->left) + count\_nodes(root->right);

}

int tree\_height(node\* root) {

if(!root)

return 0;

else

return 1 + std::max(tree\_height(root->left), tree\_height(root->right));

}

// Pretty-print a tree. This does cycle-checking at the same time, so that if

// there's a cycle in the tree we won't get stuck in a loop.

void print(node\* root, int level, int parents, bool left\_child, std::set<node\*>& nodes) {

if(level == 0)

cout << "--- Tree structure ---\n";

// Print indent for node

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

if(parents & (1 << i))

cout << " â”‚ ";

else

cout << " ";

if(level > 0)

cout << (left\_child ? " â”œâ”€" : " â””â”€");

if(root == nullptr) {

cout << "(null)" << std::endl;

}

else if(nodes.count(root) > 0) {

// Already printed this node somewhere else

cout << "CYCLE (" << root->key << ")" << std::endl;

}

else {

nodes.insert(root); // Visit root

// Print children

cout.width(3);

cout << root->key;

if(root->parent != nullptr)

cout << " [p = " << root->parent->key << "]";

cout << std::endl;

// Print children

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

// We only print both children if one of them is non-null.

// If both are null we don't print anything, to avoid making a huge

// mess.

// We print the children in the order right, left so that you can

// turn your head (or your screen) to the left and the tree will

// be correct.

print(root->right, level+1, parents | (1 << level), true, nodes);

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

}

}

}

void print(node\* root) {

std::set<node\*> nodes;

print(root, 0, 0, true, nodes);

}

/\* check\_for\_cycles(n)

Traverse the tree (preorder) starting at n, checking for cycles of nodes.

Note that this does not check for parent-pointer cycles, only child-pointer

cycles.

\*/

bool check\_for\_cycles(node\* n, std::set<node\*>& nodes) {

if(nodes.count(n) > 0)

return false;

else {

nodes.insert(n); // Mark n as seen

// Explore left and right subtrees

bool ret = true;

if(n->left)

ret = ret && check\_for\_cycles(n->left, nodes);

if(n->right)

ret = ret && check\_for\_cycles(n->right, nodes);

return ret;

}

}

bool check\_for\_cycles(node\* n) {

std::set<node\*> nodes;

if(!check\_for\_cycles(n, nodes)) {

cout << "FAILED: tree structure contains a cycle.\n";

return false;

}

else

return true;

}

// Check the pointer structure of the tree (parent/child) to make sure it is

// correct.

bool check\_tree\_pointers(node\* root, bool is\_root = true) {

if(!root)

return true;

else {

if(is\_root && root->parent != nullptr) {

cout << "FAILED: root->parent should always be null.\n";

return false;

}

// Child child nodes (if they exist) to make sure their parents

// point back to root.

if(root->left) {

if(root->left->parent != root) {

cout << "FAILED: found node " << root->left->key

<< " with incorrect parent pointer.\n";

return false;

}

if(root->left->key >= root->key) {

cout << "FAILED: found node " << root->left->key

<< " which is on the wrong side of parent.\n";

return false;

}

}

if(root->right) {

if(root->right->parent != root) {

cout << "FAILED: found node " << root->right->key

<< " with incorrect parent pointer.\n";

return false;

}

if(root->right->key <= root->key) {

cout << "FAILED: found node " << root->right->key

<< " which is on the wrong side of parent.\n";

return false;

}

}

if(root->right && root->left) {

// Both children, if they exist, have valid parent pointers.

// So now we check both subtrees recursively.

return check\_tree\_pointers(root->left, false) &&

check\_tree\_pointers(root->right, false);

}

return true;

}

}

bool check\_tree\_values(node\* root,

int low = std::numeric\_limits<int>::min(),

int high = std::numeric\_limits<int>::max()) {

if(!root)

return true;

else if(root->key <= low) {

cout << "FAILED: found node " << root->key << " improperly placed.\n";

return false;

}

else if(root->key >= high) {

cout << "FAILED: found node " << root->key << " improperly placed.\n";

return false;

}

else { // root->key is in the correct range

return check\_tree\_values(root->left, low, root->key) &&

check\_tree\_values(root->right, root->key, high);

}

}

bool check\_tree(node\* root) {

if(root->parent != nullptr) {

cout << "FAILED: Root of tree must have null parent pointer";

cout << " (root->parent->key = " << root->parent->key << ")\n";

return false;

}

return check\_for\_cycles(root) &&

check\_tree\_pointers(root) &&

check\_tree\_values(root);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Tree testing

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

template<typename Func>

struct scope\_exit {

scope\_exit(Func f) : exit(f)

{}

~scope\_exit() {

exit();

}

Func exit;

};

template<typename Func>

scope\_exit<Func> make\_scope\_exit(Func f) {

return scope\_exit<Func>(f);

}

// To test the tree functions, we generate a random permutation of the integers

// from -20 to 20 and insert them into the tree. Then, we generate another

// permutation and find them in that order. Finally, we generate another

// permutation and remove them in that order. After every operation, we perform

// a full check of the tree structure. The test stops if the tree structure is

// not valid at any point.

bool test\_rotate() {

// This is a huge mess. I need to come up with a better way to test

// left/right rotations. Maybe use member-pointers to abstract over

// the orientation?

// Root of the pseudo-tree

node\* root = new node{10000, nullptr, nullptr, nullptr};

/\* Left-rotation tree:

p

/ \

c Z

/ \

X Y

\*/

node\* X = new node{-10, nullptr, nullptr, nullptr};

node\* Y = new node{-20, nullptr, nullptr, nullptr};

node\* Z = new node{-30, nullptr, nullptr, nullptr};

node\* child = new node{2, X, Y, nullptr};

node\* parent = new node{1, child, Z, root};

// This is to avoid memory leaks: the function will be called when this

// function returns.

auto exiter = make\_scope\_exit([&]() {

delete X;

delete Y;

delete Z;

delete child;

delete parent;

});

child->parent = parent;

X->parent = Y->parent = child;

Z->parent = parent;

rotate(child, parent);

/\* New structure should be

c

/ \

X p

/ \

Y Z

\*/

if(child->parent != root) {

cout << "FAILED: parent's parent is not preserved.\n";

return false;

}

if(child->right != parent) {

cout << "FAILED: rotate did not make parent into child.\n";

return false;

}

if(child->left != X) {

cout << "FAILED: left child of child should be unchanged\n";

return false;

} else if(parent->left != Y) {

cout << "FAILED: child's right child should become right-child of parent.\n";

return false;

} else if(parent->right != Z) {

cout << "FAILED: right child of parent should be unchanged.\n";

return false;

}

else if(parent->parent != child) {

cout << "FAILED: parent->parent is not original child.\n";

return false;

}

else if(!check\_for\_cycles(child)) {

cout << "FAILED: rotation created a cycle\n";

print(child);

return false;

}

// Right-rotation

delete child; delete parent;

child = new node{2, Y, Z, nullptr};

parent = new node{1, X, child, root};

child->parent = parent;

X->parent = parent;

Y->parent = Z->parent = child;

rotate(child, parent);

if(child->parent != root) {

cout << "FAILED: parent's parent is not preserved.\n";

return false;

}

if(child->left != parent) {

cout << "FAILED: rotate did not make parent into child.\n";

return false;

}

if(parent->left != X) {

cout << "FAILED: left child of parent should be unchanged.\n";

return false;

}

else if(child->right != Z) {

cout << "FAILED: right child of child should be unchanged.\n";

return false;

}

else if(parent->right != Y) {

cout << "FAILED: left child of child should become right child of parent\n";

return false;

}

else if(parent->parent != child) {

cout << "FAILED: parent->parent is not original child.\n";

return false;

}

else if(!check\_for\_cycles(child)) {

cout << "FAILED: rotation created a cycle\n";

print(child);

return false;

}

// Do a quick test with null children and null root

// If the user made a mistake here, this will most likely segfault.

delete child;

delete parent;

child = new node{1, nullptr, nullptr, nullptr};

parent = new node{0, child, nullptr, nullptr};

child->parent = parent;

rotate(child, parent);

if(parent->parent != child) {

cout << "FAILED: parent did not become the child\n";

return false;

}

else if(child->right != parent) {

cout << "FAILED: parent did not become right child\n";

return false;

}

return true;

}

bool test\_tree() {

node\* t = nullptr; // Empty tree

// Generate test data

std::vector<unsigned> test = make\_random\_permutation(41, 12);

// Insert a random permutation

cout << "Testing tree insertion...";

for(unsigned u : test) {

const int i = static\_cast<int>(u);

cout << u << " ";

t = insert(t, i);

if(!check\_tree(t)) {

print(t);

return false; // Stop if the check fails.

}

if(t->key != i) {

cout << "FAILED: After inserting " << i << " it should be splayed to the root\n";

print(t);

return false;

}

}

cout << std::endl;

int cn = count\_nodes(t);

if(cn != 41) {

cout << "FAILED: tree does not have the correct number of nodes. ";

cout << "(expected 41, found " << cn << ")\n";

print(t);

return false;

}

else {

cout << "OK so far...\n";

print(t);

}

// Find a random permutation

cout << "Testing tree find()...";

for(unsigned u : test) {

const int i = static\_cast<int>(u);

cout << i << " ";

if(!find(t, i)) {

cout << "FAILED: find() couldn't find " << i << "\n";

return false;

}

if(t->key != i) {

cout << "FAILED: find() did not splay target to the root.\n";

print(t);

return false;

}

if(!check\_tree(t)) {

print(t);

return false;

}

}

cout << std::endl;

print(t);

// We no longer test removal, because students are not required to implement

// remove(). It's too fiddly to get right, too many edge cases.

/\*

// Remove a random permutation

cout << "Testing tree removal...\n";

for(unsigned u : test) {

const int i = static\_cast<int>(u);

t = remove(t, i);

if(!check\_tree(t)) {

print(t);

return false;

}

if(safe\_find(t,i)) {

cout << "FAILED: removed element " << i << " is still present in the tree\n";

print(t);

return false;

}

}

if(t != nullptr) {

cout << "FAILED: Tree not empty after removing all elements.\n";

print(t);

return false;

}

\*/

return true;

}

int main() {

cout << "---- Beginning tree tests ----\n";

if(test\_rotate() && test\_tree())

cout << "---- All tests successful ----\n";

return 0;

}