I&C SCI 46 Winter 2023 Project 4: Finding Balance in Nature

Due February 24 at 7:30 AM. You may use late submissions as usual. Note that this is *two weeks* after the project is released, although an exam occurs in between the two. I recommend that you get the "Required" portions completed before the exam, which can be done without the level-balanced portions, and then finish the project in the week following exam two.

Reviewing related material

I encourage you to read your textbook; the section for AVL Trees is clearly marked in the Zybook, and for the Goodrich/Tamassia book's second edition, section 10.1 covers Binary Search Trees and 10.2 talks about AVL Trees. Both books are good at getting to the point, so this should not be a long read. Furthermore, you should look at your notes from the associated lectures.

However, be aware that we are NOT implementing an AVL Tree this quarter; those are *height balanced* trees. Understanding them is an important preliminary step to this.

Requirements

In this project you will be implementing the Level-balanced tree data structure as a class named `LevelBalancedTree`. The class consists of the following functions which you are responsible for implementing and have been started for you in `LevelBalancedTree.hpp`:

`LevelBalancedTree()`

This is the constructor for the class. You should initialize any variables you add to the class here.

`~LevelBalancedTree()`

This is the class destructor. You are responsible for freeing any allocated memory here. You will most likely be allocating memory to store the nodes within the tree. Since these allocations need to be dynamic, as we don't know how large the tree will be, they should be freed here in the destructor. It's your job to come up with a traversal algorithm to accomplish this. Note, if you elect to use shared pointers or unique pointers the compiler will generate code to deallocate the memory for you if certain conditions are met. You should only use these features of the standard library if you already understand them or are willing to put in extra effort. In *most* industry settings features like these will be used as opposed to explicitly implemented destructors.

However, be advised that course staff *are not expected to know these* and might not be able to help you debug problems with them. If you are unfamiliar with shared or unique pointers, use traditional (raw) pointers if you are expecting help with debugging.

`size_t` **`size()`** `const noexcept`

This function returns the number of keys stored in the tree. It returns the count as a `size_t.` It is marked const (also known as a constant member function) because it should not *modify* any member variables that you've added to the class or call any function functions that are not marked const as well. The advantage of marking this function as `const` is that it can be called on constant `LevelBalancedTree` instances. It also allows the compiler to make additional optimizations since it can assume the object this function is called on is not changed. This is a fairly good StackOverflow answer that goes into additional detail.

`bool` **`isEmpty()`** `const noexcept`

This function simply returns whether or not the tree is empty, or in other words, if the tree contains zero keys. Marked const because it should not change any member data. Marked noexcept because it should not throw any exceptions.

`bool` **`contains(const Key & k)`** `const noexcept`

Simply checks to see if the key `k` is stored in the tree. True if so, false if not. Once again, this function does *not* modify any member data, so the function is marked const. Since this is a balanced tree, this function should run in *O(log N)* time where *N* is the number of keys in the tree. This is accomplished through the on-demand balancing property of level-balanced trees and a consequence of the height of the tree never exceeding *O(log N)*. **IMPORTANT:** when comparing keys, you can ***only assume*** that the `<` and `==` operator has been defined. This means you should ***not*** use any other comparison operators for comparing keys.

`Value &` **`find(const Key & k)`**

Like `contains()`, this function searches for key `k` in the tree. However, this function returns a *reference* to the value stored at this particular key. Since this function is not marked const, and it does not return a const reference, this value is modifiable through this interface. This function should also run in *O(log N)* time since it is bound by the height of the tree. If the key `k` is not in the tree, a `ElementNotFoundException` should be thrown.

`const Value & **find(const Key & k)** const`

Same as the constant version of `find`, but returns a *constant reference* to the stored value, which prevents modification. This function is marked const to present the `find` (or "lookup") interface to *instances* of `LevelBalancedTree` which are marked const themselves. This means that member data should *not* be modified in this function. For example, the following code would call the version of `find()` marked constant:

```
LevelBalancedTree<int, int> tree;
const LevelBalancedTree<int, int> & treeRef= tree;

treeRef.find(1);
```

Warning: this function will not be *compiled* until you *explicitly* call it on a constant LevelBalancedTree as in the example above.

`void **insert(const Key & k, const Value & v)**`

Adds a (key, value) pair to the tree. If the key already exists in the tree, you may do as you please (no test cases in the grading script will deal with this situation). The key `k` should be used to identify the location where the pair should be stored, as in a normal binary search tree insertion. Since this is an level-balanced tree, the tree should be ***rebalanced*** if this insertion results in an unbalanced tree. See your notes from the lecture on February 8, 10, and 13 for more information.

Note: this is by far the most difficult part of this project.

`void **remove**(const Key & k)`

Removes the given key from the tree, fixing the balance if needed.

`std::vector<Key> **inOrder()** const`

Returns a vector consisting of the keys in the order they would be explored during an *in-order traversal* as mentioned in class. Since the traversal is "in-order", the keys should be in ascending order.

`std::vector<Key> **preOrder()** const`

Returns a *pre-ordering* of the tree as described in Section 7.2.2 of the textbook and in class. For the purpose of this assignment, the left subtree should be explored before the right subtree.

`std::vector<Key>` **`postOrder()`** `const`

Returns a *post-ordering* of the tree as described in Section 7.2.3 of the textbook and in class. For the purpose of this assignment, the left subtree should be explored before the right subtree.

Additional Notes

- Your implementation must be templated as provided.
  - Be sure yours works for non-numeric types! **char** is a numeric type.
  - Review the warnings in the lab manual, the grading policies, and in particular the warning about templated code in the "Grading Environment" section.
- You do not need to write a copy constructor or an assignment operator on this project, but knowing how to do so is generally a good thing.
- As stated in the `contains()` function: for comparing keys, use the "natural" comparison offered by <. You should assume that < and == are defined for any object used for Key. Any test cases provided will have something for the key that has this defined.
- The project will not build by default because a reference to a local variable is returned in the `find()` functions. You will need to write an implementation that doesn't do this.

Restrictions

Your implementation must be implemented **via linked nodes** in the tree format from the lecture. That is, you may ***not*** have a "vector-based tree." This means you will probably need to create a new structure inside of your LevelBalancedTree class which will represent the nodes.

You **may not** use parts of the C++ standard template library in this assignment *except* for `std::vector`. Furthermore, `std::vector` may only be used when implementing the three traversals (in-order, pre-order, post-order). For what it's worth, you won't miss it for this assignment. As always, if there's an exception that you think is within the spirit of this assignment, please let me know.

Your implementation does not have to be the most efficient thing ever, but it cannot be "too slow." In general, any test case that takes over a minute on the grader's computer may be deemed a wrong answer, even if it will later return a correct one.

Additional Grading Note

This is an additional warning that the public tests are not comprehensive. Remember that the compiler *does not* compile functions which are not used. Thus, at the bare minimum you should add additional unit tests which get all of your code to compile. Using different template types will help to make sure you don't accidentally bake in assumptions about the type of the `Key` or `Value`. Always commit your unit tests with your code.