ICS 46 Summer 2023

Project #4: AVL Trees

Due Monday, July 31, 2023 at 12:00 p.m.

Introduction

In the previous project we implemented the Dictionary ADT as a skip list. The skip list gave $O(\log n)$ insertion and removal in expectation. In this project you implement a data structure which implements the Dictionary ADT in $O(\log n)$ time *guaranteed*. That data structure is the AVL tree as seen in class.

Related Materials

I encourage you to read Goodrich/Tamassia's second edition (see class syllabus for a link to the book), section 10.1 covers Binary Search Trees and 10.2 talks about AVL Trees. Furthermore, you should look at your notes from the associated lectures to develop an intuition for the rotation mechanics.

Requirements

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

```
MyAVLTree()
```

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

```
~MyAVLTree()
```

> 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 `MyAVLTree` 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 AVL 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, an `ElementNotFoundException` should be thrown.

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

Same as the above 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 `MyAVLTree` 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:

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

treeRef.find(1);
```

Warning: this function will not be *compiled* until you *explicitly* call it on a constant MyAVLTree 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 AVL tree, the tree should be ***rebalanced*** if this insertion results in an unbalanced tree. See your notes from the lecture on the rebalancing process, or the section on AVL trees in the textbook.

```
unsigned height(const Key &k) const
```

Returns the current height of key `k` in the AVL tree. This function should throw `ElementNotFoundException` if the key doesn't exist in the tree.

~~`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.

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 `MyAVLTree` 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.

Timing

- You should be able to insert 200,000 (integer, integer) pairs into an AVL tree in 6 seconds on OpenLab. If you implement an AVL tree insert with an $O(\log n)$ run-time bound you should easily meet this requirement.
- You should be able to find an element in a tree with 200,000 keys in under 500 milliseconds.

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`.

This project will use the normal 40% public, 40% private, 20% memory leak distribution.