Kyle Lund

AVL Tree Documentation

Function time complexity:

Note: For all methods, n represents the number of nodes currently in the tree and all time complexities are worst case.

Insert: Insertion in a balanced binary search tree runs in O(log(n)) time. At any node, we can do three things, if the value of the insertion is less than that at the node we're looking at, we look to the left. If it's greater, we look to the right. And if it's equal, in this implementation we do nothing. At each decision point we are ruling out half of the available options, resulting in a log(n) runtime. Insertion may require rebalancing the tree trigger a series of rotations, but each individual rotation runs in constant time, and the number of rotations is log(n) in the worse case as we simply move up the tree, reversing the course we took during the initial portion of the insert.

Search: A search method reduces to an insert and runs in the same O(log(n)) time. Search uses the same traversal that an insert uses to find the proper place of the desired tree but stops short of inserting that value and avoids the need to perform any tree rebalancing.

Remove: Remove also reduces down to an insert and runs in O(log(n)) time. Just like search, we first traverse down the tree to find the proper node. Once found, just like with an insertion, we may perform a log(n) number of additional constant time rebalancing operations in the worst case.

In/Pre/Post order traversals: These all run in O(n) time. Each node in the tree is visited exactly once, the only difference between these three is the order in which they are visited.

Remove in order: This method reduces to an in-order traversal and therefore runs in O(n). An AVL tree does not store order information in the tree, so this information must be derived. This is done with an in-order traversal. Once the specific node is found, a standard removal is performed which runs in O(log(n)) in the worst case. This results in a complexity of O(n + log(n)), which per the rules of big O, reduces to O(n).

Level count: In an AVL tree which stores height information in each node, this method runs in constant time. The number of levels in a balanced tree is simply the height of the root node (when height is 1 indexed). The height value is updated any time an insert, removal, or rotation takes place, so no further computation is required.

Lessons learned:

I had two main takeaways from this project. The first lesson learned for me was to remember to keep testing in mind when designing the architecture and functions, and to test continuously during development. My mistake was building almost the entirety of the AVL tree class and an interface class to get user input before running tests. The result of this was that I had to make significant modifications to the structure and responsibilities of both classes in order to facilitate testing.

The second takeaway was that I needed to spend more time up front designing the overall structure of the program. One of the main things that I dislike about C++ is the pain involved in compiling multiple files together, and my first implementation had everything in one giant file. This worked, but as it got bigger it started requiring more and more effort to maintain. After the code was functionally complete, but before getting deep into testing, I choose to refactor and break out the two primary classes into their own .h and .cpp files, with a simple main file to run the program. This refactor, and the extra testing required to make sure it worked, cost me several hours of development time that could have been

prevented with ~15 minutes of planning, and a little extra willpower to force myself to deal with C++ compiling.

The main things I'll need to do different next time is use a proper TDD cycle throughout development, spend more time up front planning the development, and as a bonus, figure out how to use smart pointers in projects like this without it devolving into one giant circular dependency.