COP3530 Data Structures and Algorithms
Project 1: Gator AVL Project
Geoffrey Clark

Insert NAME ID

- Both average and worst-case time complexities are O(log n), where n is the number of nodes in the tree. This is due to recursively traversing the AVL tree in what is essentially a binary search to find an appropriate location. Although there are rotation, these performed in constant time and don't significantly change the O(log n) complexity.

Remove ID

- The time complexity is O(logn), where n is the number of nodes in the tree. The balance maintained in the AVL tree allows a binary search-like method to find the specific node, based on whether the specified ID's value is higher or lower than the current node's ID.

Search ID

- The time complexity is O(log n), with n being the number of nodes. Similar to "Remove ID", the AVL tree's balanced nature allows a binary-style search for a specified ID.

Search NAME

- The complexity is O(n), where n represents the number of nodes. My 'Search NAME' function uses a preorder traversal to check the name value of every node. This traversal of all nodes is necessary since the AVL tree sorts nodes by their ID, not their names, disallowing the binary search technique for names.

printInorder

- The complexity is O(n) with n being the number of nodes. This function uses an inorder traversal to add each node to a vector which is subsequently iterated over to print names. The combined operations yield a worst-case scenario of O(2n), which simplifies to O(n). To improve the function, I could skip the vector and print directly from the node.

printPreorder

- O(n) complexity, where n is the node count, this function is similar to 'printInorder', and iterates over each node to place them into a vector to then be printed. This yields a worst-case of O(2n) which simplifies to O(n).

printPostorder

- The complexity here is O(n), with n being the number of nodes. This function follows a pattern similar to the aforementioned print functions, ultimately resulting in a worst-case complexity of about O(2n), simplified to O(n).

printLevelCount

- O(n) complexity where n is the node count, this function recursively calls the height() function to iterate through all nodes and determine the lengthiest path. Despite the AVL tree's balancing threshold of 1, any given subtree may differ from others by one level. This is what causes the requirement of iteration through all nodes.

removeInorder N

- The complexity is O(n) where n is the node count. In the worst-case, an inorder traversal is executed, taking O(n) time to identify the Nth node within a vector. I then use the remove function with O(logn) complexity. Note that the combined complexity is O(n) + O(log n), not O(n) * O(log n). Therefore we simplify to O(n).

Learning points and what I would do differently if I started over:

I learned much more about AVL trees and the rotations required to keep them balanced. I also gained experience in seeing how helpful they can be for storing information in certain contexts. They not only allow a specific node to be accessed efficiently in logarithmic time complexity, but also allow for new nodes to be added efficiently as well. I did notice that this data structure only allows the binary sorting to be done with one characteristic though, and additional secondary characteristics (such as name as opposed to ID) are not able to determine the organization. Thus, secondary characteristics of nodes will require the less efficient linear time complexity to access. I am interested in learning about the applications of using different data structures with differing efficiency tradeoffs, e.g. sacrificing efficiency in accessing one type of data (e.g. ID) in return for greater efficiency in accessing multiple secondary types of data (e.g. name, phone number, age, etc).

There are numerous things I would do if I started over. I would make optimizations such as perhaps adding a height attribute to each node that gets updated during rotations and gets compared to a max/min height. Then printLevelCount could be an O(1) check instead of O(n). I would also break down certain large complex functions into smaller functions to allow for better modularity and error checking. I could also think more about the pros and cons of applying more iterative approaches rather than recursive ones. Another example of something I would do differently is to create a more comprehensive list of tests that include more edge cases.