Project 1: AVL Tree Report

By: Michael Gerber

Time Complexity:

In this project there were nine main commands that needed to be implemented in the AVL tree and tested. The time complexity of these commands in Big-O notation, and particularly the worst-case time complexity of these commands is described in the text below.

Note: These descriptions are in terms of n, where n is the number of nodes in the AVL tree.

insert (string name, string ufid) – The worst-case time complexity of insert is **O(log n)**. The insert function works by recursively traversing the tree until a leaf is reached at which it inserts a new node based on the value of the passed in ufid compared to the node. It then calls a function to get balance factor which calls another function that uses recursion to get the height of a given subtree. These functions have a worst case time complexity of O(log n) when they have to travel the longest path from root to leaf. The function also performs rotations as necessary to balance the tree, however these rotations take only O(1) time. In total the worst case time complexity for insert is O(log n) when it travels along the entire height of the AVL tree which is at most log n.

remove (string ufid) - The worst-case time complexity of remove is **O(log n) .** This implementation of deletion is similarly structured to that of deletion from a BST. In the worst case, deletion from a BST is O(n) where the height of the tree can behave like a linked list and have height of n. However in an AVL tree the height is at most log n, therefore deletion takes O(log n) time. Also in this implementation, for the case when deleting a node with 2 children, the inorder successor was found by using a function minNode which traverses down the longest path of the left subtree to return the smallest node. This function doesn't alter the worst-case time complexity of remove as it also contributes O(log n) time.

searchId(ufid) - The worst-case time complexity of searchId is **O(log n).** This function uses the value of the passed in ufid to determine which direction (left or right) to search on the tree until the id is found or not found. In the worst case, this function will search down the longest path of either its left or right subtree which takes at most O(log n) time for an AVL tree.

searchName(name) - The worst-case time complexity of searchName is **O(n).** This function uses a preorder traversal to search for the names in order to account for cases when there are duplicate names but different ufid's. Since this is a traversal, it visits every node, and therefore has a time complexity of O(n).

printInorder() - The worst-case time complexity of printInorder is **O(n).** This function uses an inorder traversal to print the name of every node in the tree and therefore will have a time complexity of O(n).

printPreorder() - The worst-case time complexity of printPreorder is **O(n).** This function uses a preorder traversal to print the names of every node in the tree and therefore will also have a time complexity of O(n).

printPostorder() - The worst-case time complexity of printPostorder is **O(n).** This function uses a postorder traversal to print the names of every node in the tree and will also have a time complexity of O(n).

printLevelCount() - The worst-case time complexity of printLevelCount is **O(log n).** This function uses the height helper function to determine the level count. This function therefore has a worst case time complexity of O(log n) when height of root node is log n.

removeInorder(n) - The worst-case time complexity of removeInorder is **O(n)**. This function calls to an inorder traversal helper function which has a worst-case time complexity of O(n) since it visits every node. The next main call in this function is to the same remove helper function that was used in the main remove function, which as described above takes O(log n) time in the worst case. In total, the time complexity of removeInorder equates to O(n).

Project Takeaways:

Overall, I enjoyed this assignment. I found it very helpful that a lot of the pseudocode and theory behind the methods described above could be found in the lectures. I felt the lectures did a great job of providing enough information needed to complete the project but leaving it up to the student to put all the pieces together. While it was time consuming trying to get the implementation to work precisely the way it needed to for the test cases, I feel I learned a lot during the creation and debugging processes.

One thing I learned from this project that I wasn't too familiar with before was input parsing. I started the project by working on the main function. During this process, I ran into some problems dealing with double quotes and differentiating between if the function was passing a name or an id. However, after this process, I now feel more comfortable with using commands like erase(), substr(), and find() to parse through input to zone in on exactly what it is that needs to be passed in.

If I had to start this project over I definitely would make sure to utilize vectors more from the beginning. I ran into some issues about halfway through the project with getting the print traversal functions to properly print. I initially kept getting a list where the final name listed would be followed by a comma when it shouldn't have had anything following it. This was because my print helper functions were directly printing with "cout" and it made it difficult to properly print a comma separated list because the function wasn't aware of when the last item was coming. Due to this, I found it best to go back and add vectors to all my traversal helper functions and then print the traversals by simply iterating through each item in the vector.