# Tom Nguyen

**Time Complexity Analysis of Each Function**

1. Insert() – O(m log(n)) where n is the number of nodes in the tree. My insert function first checks if the id already exists by traversing the tree towards the target from root to leaves, so since the height of the tree is log(n), this part is log(n). Regex comparisons are O(m) since regex_match has time complexity of O(m) for name. My insert helper function calls multiple functions inside, with the worse time complexity component being O($(\log(n))^2$). Finally, rotate has a complexity of O(log(n)). Combining all the terms, my insert function has O(m log(n)) time complexity.
2. Remove() – O(log(n)) where n is the number of nodes in the tree. The remover helper recursively traverses a path from root to where deletion occurs, travelling down the log(n) height of the tree. The rotation function and balance factor updating function also have time complexities of O(log(n)) for the same reason. So, the remove function has time complexity O(log(n)).
3. SearchID() – O(log(n)) where n is the number of nodes in the tree as it recursively travels down the height of the tree to find the node with the right ID.
4. SearchName() – O(log(n)) where n is the number of nodes in the tree as it recursively travels down the height of the tree to find the node with the right Name.
5. printInorder() – O(n) where n is the number of nodes in the tree as it performs the O(1) operation of printing out the value of each node it touches for every node in the tree.
6. printPreorder() – O(n) where n is the number of nodes in the tree as it prints the values of every node in the tree one by one.
7. printPostorder() – O(n) where n is the number of nodes in the tree as it prints the values of each node in the tree.
8. printLevelCount() – O(log(n)) where n is the number of nodes in the tree as it calls a function that counts the levels by travelling down the height of the tree. Since balanced AVL trees have a height of log(n), the time complexity of this function is O(log(n)).
9. removeInorder() – O(n) where n is the number of nodes in the tree as in the worst case that the user asks to remove the last element of the inorder traversal, the function has to travel through every node to reach the last element before performing the constant time operation of deletion.

**What I learned and what I would do Differently**

I got a thorough understanding of writing recursive functions and got to understand the intricacies of the AVL Tree through implementing it. I also learned a lot about debugging and troubleshoot throughout the process in order to figure out and resolve issues that came up while I programmed the project. The unit test section of making my own test cases taught me how to use the Catch2 framework to test using Git Bash.

If I were to do this project again, I would be more careful in implementing the balance factor checks as I currently have it implemented to travel through the tree after inserting or deleting elements. I would also implement my searches in the RemoveHelper to not have an extra travelling pointer as it takes more time than if I implemented it without. This would mean I would think of a recursive solution for that part.