insert NAME ID

This function has a complexity of **O(log n)** where n is the number of nodes in the tree. Since the tree is a balanced binary search tree, and the maximum height would be log n, the worst case time complexity would be O(log n). The program compares to see if the ID that wishes to be inserted is higher or lower than the ID that it is looking at, and goes to the next node accordingly. Since it is balanced the tree won't be very long on one side, and since the tree is sorted it can find where to go.

remove ID

This function has a complexity of **O(n)** where n is the number of nodes in the tree. Similar to the insert function, the maximum height of the tree is the maximum traversal to find the node to remove. On the other hand, once the node is found the tree must perform an inorder search on the tree to find the node's inorder successor. An inorder search has a time complexity of O(n). So the full time complexity is O(n + log n). However since in big O notation with addition you remove the smaller order number, the worst case time complexity would be O(n).

search ID

This function has a complexity of **O(log n)** where n is the number of nodes in the tree. Since there is only one of every ID, the function merely performs a binary search to find the proper ID. With the same logic as insert, the maximum height of the tree is log n, so the farthest the program will need to traverse is log n.

search NAME

This function has a complexity of **O(n)** where n is the number of nodes in the tree. Since the tree is not sorted by names, and there can be multiple names, the program will need to look at every single node. Therefore it has a complexity of O(n).

printInorder

This function has a complexity of **O(n)** where n is the number of nodes in the tree. This function prints every node in the tree, so it must look at every node in the tree.

printPreorder

This function has a complexity of **O(n)** where n is the number of nodes in the tree. This function prints every node in the tree, so it must look at every node in the tree.

printPostorder

This function has a complexity of **O(n)** where n is the number of nodes in the tree. This function prints every node in the tree, so it must look at every node in the tree.

printLevelCount

This function has a complexity of **O(n)** where n is the number of nodes in the tree. This function finds the longest path down. Since the program does not know precisely which

branch will lead to the longest path, it simply compares all of the branches together to find which branch is the longest. Therefore it looks at every node in the tree.

removeInorder

This function has a complexity of **O(n)** where n is the number of nodes in the tree. This function must traverse the given amount of nodes, so in the worst case scenario it would have to traverse *n* nodes. After finding the proper node, I take the UFID at the found node and call the RemoveID function. This means that the complexity is O(n) + O(n) + O(log n). Since with addition you remove the lower order numbers, the time complexity would still be O(n).

If I had to redo this assignment I would definitely change a few implementations. The biggest example being removeInorder. Since removeInorder already traverses the tree in inorder order, the inorder successor could be found in a much more efficient way. I unfortunately couldn't get this solution to work. I would also change removeInorder to be recursive rather than iterative, because I feel it is a more elegant solution. Besides the implementation of my functions, I would also have changed how I handled white spaces. I handle it differently every time the issue of white spaces comes up, so if I had to redo the project I would handle it in the same manner every single time.

With this assignment I learned about using test cases and especially about ways to take input. The input took a while for me to implement especially since users needed to be able to input a name with a space. I also learned a lot about balancing trees like how to tell when a tree needs to be balanced, how to tell how a tree needs to be balanced, and performing the balancing operations. I had already known about trees in high school but self balancing trees are much more difficult.