Outcome Complexity Analysis

For all functions, the strings NAME and ID are considered to have constant length, because they cannot exceed 1000 due to the testing constraints

For all functions, n denotes the size of the tree, or the amount of nodes that the tree contains

insert NAME ID

$O(log(n))$
This function visits the same nodes as a search for ID (discussed under search ID), then does the constant time operation of inserting a node. For each node passed, the constant time function fixHeights is called and rotations to balance the tree, which are also constant time, may be called.

remove ID

$O(log(n))$
This function essentially searches for the node to remove, (discussed under search ID), and once found, does constant time operations in the one child and no child cases. In the two child case, the inOrder successor must be found, but this never requires more than log n time complexity to find, because it uses a depth based search. In all cases, the height is then updated, which is also O(log n). So the worst case time complexity if $O(log(n) + log(n) + log(n)) = O(log(n))$

search ID

$O(log(n))$
Because the balance factor of each node is between -1 and 1 inclusive, and because the tree is a BST, each recursive call cuts the search space in half with perhaps one extra node on one side. As input size increases, the effect the extra node may have on the time complexity becomes negligible: $\lim_{n \to \infty} (\frac{n}{2} + 1)\frac{1}{n} = \frac{1}{2}$. Thus the function has logarithmic time complexity (base 2) with respect to n, the tree size.

search NAME

$O(n)$
Since the tree is not sorted by name, and repeats are allowed, every node's name must be checked to gather all nodes with a given name. To do this, the function traverses through the tree preOrder. Thus the function has linear time complexity with respect to n, the size of the tree.

printInorder

$O(n)$
This function is a traversal, so all nodes are accessed. Once accessed, constant time complexity operations are executed, meaning the the time complexity is linear with respect to n, the size of the tree

printPreorder

$O(n)$
This function is a traversal, so all nodes are accessed. Once accessed, constant time complexity operations are executed, meaning the the time complexity is linear with respect to n, the size of the tree

printPostorder

$O(n)$
This function is a traversal, so all nodes are accessed. Once accessed, constant time complexity operations are executed, meaning the the time complexity is linear with respect to n, the size of the tree

printLevelCount

$O(1)$
This function has constant time complexity because it only accesses data of the root node.

removeInorder N

$O(N + log(n))$
This function traverses through the tree Inorder, which takes O(log n) to reach the first node in the worst case. After reaching the first node (at the bottom of the tree), the nodes in the next N - 1 steps of the traversal are not more than constant time complexity away, because of the balanced nature of the tree. Once the desired node is reached, it is removed which is a O(log n) operation (note that this could be made constant, but as it does not change the O() time complexity of the overall algorithm I did not optimize it). So the time complexity of this method is O(N + log(n) + log(n)) = O(N + log(n))
Here n is the height of the tree (height of its root), and it is the Nth node that is desired for removal.

Reflection

This project took me over 35 hours to complete, and I spent a lot of time starting over multiple times on almost all of the functions. The biggest mistake I made was starting on the design of fundamental parts of the project, like the insert functionality, before watching module five lectures. Basically, the way I coded my own insert function did work, but it took me much longer than anticipated and the design was cumbersome and difficult to fit in with the rest of the functionality of the project. After looking at the pseudocode provided in the module, I implemented a much better insert function.
The way heights are updated after removal can probably be optimized, or at least made less disorganized, so if I revisited this project I would try to find a way to update heights within the removal function. Another thing I would do differently is think about what helper functions I need for my public functions first, to perhaps not have to make a duplicate private function for each public function. Lastly, in one case where finding the inOrder successor was necessary, I used a pointer to a pointer. There was likely a simpler way to do this.