Describing worst case Big O complexity of each method

insertNameId()

Variable under consideration and definition: h, the height of the AVL tree, which corresponds to the number of levels from the root to the deepest node.

Justification: Since an AVL tree stays balanced, every insertion, deletion, and searching elements is O (log n). The height of an AVL tree with n nodes is O(log n)

removeAccount()

Variable under consideration and definition: h, the height of the AVL tree, which corresponds to the number of levels from the root to the deepest node.

Justification: The time complexity of the removeAccount operation is determined by the traversal from the root to the we want to delete. Given that this is an AVL tree, this traversal takes O(log n) time, where n is the number of nodes in the AVL tree.

searchGatorId()

Variable under consideration and definition: h, the height of the AVL tree, which corresponds to the number of levels from the root to the deepest node

Justification: In this method, IDs are unique. Searching IDs will have a time complexity of O(log n), where n is the number of nodes. This method searches for a student Id in the AVL tree. In the worst case, the traversal is O(log n) since the height of the tree is always bound by O(log n) in an AVL tree that stays balanced after every operation.

searchNameId()

Variable under consideration and definition: h, the height of the AVL tree, which corresponds to the number of levels from the root to the deepest node.

Justification: In this case, IDs are unique, but names may have duplicates. The searchNameId method searches student names. Since names may not be unique, the function may have to check for a specific name on both left and right subtrees if it is not found in one branch, which means we would have to traverse additional nodes. Therefore, in the worst case, it could visit all nodes in the AVL tree, corresponding to a linear complexity or O(n), where n is the total number of nodes.

printInorder()

Variable under consideration and definition: n, the total number of nodes in the AVL tree.
Justification: This method performs an in-order traversal, where each node is visited only once, and the time complexity is proportional to the number of nodes in the tree or

O(n). While visiting just one node would be a constant-time operation or O(1), the operation will be linear **O(n)** since there are n nodes to traverse to print the In-order traversal.

printPreorder()

Variable under consideration and definition: n, the total number of nodes in the AVL tree.

Justification: This method performs a pre-order traversal, where each node is visited only once, and, therefore, the time complexity is proportional to the number of nodes in the tree or O(n).

printPostorder()

Variable under consideration and definition: n, the total number of nodes in the AVL tree.

Justification: The method performs a post-order traversal by recursion. Here, every node is visited only once, and since the traversal must print all nodes or n nodes, the worst-case time complexity is linear or O(n), where n is the number of nodes in the tree.

printLevelCount()

Variable under consideration and definition: h, the height of the AVL tree (height of the root node).

Justification: This method prints the stored height of the root node, and the operation does not require tree traversal or rotations that would involve O(n). Thus, the time complexity for the printLevelCount method is O(1).

removeInorder()

Variable under consideration and definition: n, the total number of nodes in the AVL tree.

Justification: This method first performs an in-order traversal until the desired position is reached. As we traverse the AVL tree, the operation would require visiting all nodes in the worst case, which is O(n). Once the position is found, the removeAccount() method is called, which time complexity is O(log n) as mentioned earlier. However, the time complexity of the function removeInorder() as a whole is O(n) since O(n) grows faster than O(log n).

Reflection on what I learned and what I would do differently

In this project, I put into practice my knowledge gained during the first modules of the course. I could see how a balanced tree, or a tree that is as bushy as possible, plays a fundamental role in ensuring the efficiency of certain operations. I better understood the time complexity behind each operation behavior and the relationship between the height of the tree and the number of nodes in an AVL tree. I would have liked more time to perform more extensive test cases and make my code more efficient, but I could not achieve that due to other projects and duties. Thus, although I am satisfied with the result, if I had to start over again, I would commit more time to the design stage before going deeper into the implementation. I would improve my time management to allow more time and flexibility and ensure my code is robust and efficient. With this assignment, I have learned the importance of careful planning and a clearer idea of what I want to achieve in my code so that the process flows better and I do not get stuck for too long on some parts of the code implementation.