Time complexity

1. Insert:
    - Inserting a node in an AVL tree is O(logn) due to the property of AVL trees being balanced. The rotation of the tree is O(1) operations. Therefore, the overall time complexity is O(log n).
2. RemoveId:
    - The deletion operation in an AVL tree, including the AVL tree balancing, has a time complexity of O(log n).
3. minValueNode:
    - This method has a time complexity of O(log n) in the worst case because it traverses down the left side of the AVL tree.
4. searchID:
    - The search operation for a binary search tree is O(log n)
5. searchForID:
    - It calls for searchID method, therefore is o(log n).
6. searchName:
    - In the worst case, the method would need to traverse the entire tree. This results in a time complexity of O(n).
7. rotateLeft & rotateRight:
    - These are constant time operations because they involve a fixed number of pointer changes. The time complexity is O(1).
8. printPreOrder, printInOrder, printPostOrder:
    - These tree traversal methods will touch every node exactly once, resulting in a time complexity of O(n).
9. printLevelCount:

- This method traverses each level of the AVL tree once, so its time complexity is O(n).

10. removeInOrderN:

    - In the worst case, this function could potentially traverse the entire tree, giving it a time complexity of O(n)

What did I learn from this assignment and what would I do differently if I had to start over?

The critical takeaway was seeing how data structures can be optimized for specific operations.

If I had to start over, I might consider adding more comments to each functions to better explain the logic behind each operation. I would also add more helper methods to make the main methods more concise and readable.