Insert (NAME, ID):

- **Time Complexity**: O(log n)
  - In an AVL tree, the height is maintained as O(log n) due to balancing operations. Therefore, inserting a new node takes O(log n) time in the worst case.

Remove (ID):

- **Time Complexity**: O(log n)
  - Removal in an AVL tree involves finding the node to remove and rebalancing the tree if necessary, which takes O(log n) time in the worst case due to the tree's balanced height.

Search (ID):

- **Time Complexity**: O(log n)
  - Searching for an ID in an AVL tree follows the binary search pattern, taking O(log n) time in the worst case.

Search (NAME):

- **Time Complexity**: O(n)
  - Since names are not ordered lexicographically in the AVL tree, searching by name involves traversing the entire tree, which takes O(n) time in the worst case.

Print Inorder:

- **Time Complexity**: O(n)
  - Inorder traversal visits each node exactly once, so the time complexity is O(n).

Print Preorder:

- **Time Complexity**: O(n)
  - Preorder traversal also visits each node once, leading to a time complexity of O(n).

Print Postorder:

- **Time Complexity**: O(n)
  - Postorder traversal similarly visits each node once, resulting in O(n) time complexity.

Print Level Count:

- **Time Complexity**: O(n)
  - Level-order traversal visits each node in the tree once, making the time complexity O(n).

Remove Inorder (N):

- **Time Complexity**: O(n)
  - Removing the nth node in inorder involves first collecting the nodes in an inorder traversal, which takes O(n) time. The actual removal process is O(log n), but the traversal dominates the time complexity.

What would you do differently ?

The search by name currently has a time complexity of O(n) due to the need to traverse both left and right subtrees. I would consider alternative ways to store names or implement an auxiliary data structure (like a hashmap) to enable faster lookups by name.