Blake McGahee UFID 82924917
UF – COP3530 Summer'25
6/1/2025

Project 1: Gator AVL Project Report

Time Complexity Analysis

Here's my understanding of how long each main method (command) in our AVL tree takes to run in the worst-case scenario, described using Big O notation. 'N' usually refers to the number of nodes currently in our AVL tree, and 'L' refers to the length of the command string we're parsing.

1. **insert(const std::string& name, int id)**
    - **Time Complexity:** O(logN+L)
    - **Explanation:** When we insert a new student, the tree has to figure out where it goes by going down a path from the top (root) to where it should be. Because it's an AVL tree, it keeps itself pretty balanced, so this path is usually very short, like logN steps. After inserting, it might do a couple of 'rotations' to stay balanced, but these are quick, constant-time steps. We also have to check if the name is valid (L) and if the ID already exists (which also takes logN time). So, it's basically the time to go down the tree plus the time to check the name.
2. **remove(int id)**
    - **Time Complexity:** O(logN)
    - **Explanation:** When we remove a student by their ID, the tree also has to find that student, which takes about logN steps, similar to inserting. If the student has two branches (children), we might have to find the next closest student (inorder successor), which also takes about logN steps down a branch. After removing, the tree might need to rebalance itself with a few rotations, which are quick operations. So, overall, it's efficient, taking time proportional to the tree's height.
3. **searchID(int id)**
    - **Time Complexity:** O(logN)

- **Explanation:** Searching for a student using their ID is one of the main advantages of a balanced BST like an AVL tree. We just follow the path down the tree, comparing the ID at each step. Since the tree is always balanced, the longest path is very short, taking about logN steps.

4. **searchName(const std::string& name)**
    - **Time Complexity:** O(N)
    - **Explanation:** This one is tricky! Because we need to find *all* students with a specific name, and names aren't how the tree is sorted (IDs are), we have to look at *every single student* in the tree just in case their name matches. This means visiting all N nodes in the tree in a special order (pre-order traversal). So, in the worst case, it will take time proportional to the total number of students in the tree.
5. **printInorder()**
    - **Time Complexity:** O(N)
    - **Explanation:** To print every student's name in alphabetical order by ID (which is what inorder traversal does), we have to visit every single node in the tree. This means the time taken is proportional to the total number of students, N. Building the string with all the names also takes about N steps.
6. **printPreorder()**
    - **Time Complexity:** O(N)
    - **Explanation:** Similar to printInorder(), to print every student's name in pre-order traversal (Root-Left-Right), we also have to visit every single node in the tree. So, it takes time proportional to the total number of students, N.
7. **printPostorder()**
    - **Time Complexity:** O(N)
    - **Explanation:** Again, to print every student's name in post-order traversal (Left-Right-Root), we must visit every single node in the tree. So, the time taken is proportional to the total number of students, N.
8. **printLevelCount()**

- **Time Complexity:** O(1)
- **Explanation:** This method is super fast! It just asks the root node for its height value, which is already stored and updated during inserts and removes. So, it's a constant time operation, no matter how big the tree is.

9. **removeInorder(int n)**
   - **Time Complexity:** O(N)
   - **Explanation:** To find the Nth student in the inorder sequence, the tree has to traverse through nodes in inorder until it counts up to the Nth one. In the worst case (if N is near the end of the tree), it might have to look at almost all N nodes. Once it finds and removes the student, the tree rebalances, but the traversal to find the spot is the main time-consumer.
10. **processCommand(const std::string& command)**
    - **Time Complexity:** O(L+Complexity of Called Method)
    - **Explanation:** This method first reads and understands the command, which takes time proportional to the length of the command string, L. After parsing, it calls one of the other methods we just talked about (like insert, remove, searchID, etc.). So, the total time for processCommand will be the time it takes to parse the command plus the time it takes for the specific function it calls to run. Since some commands (like searchName or printInorder) can take O(N) time, the worst-case for processCommand would be when it calls one of those, making it roughly O(L+N).

What I Learned and What I Would Do Differently

This assignment was a big one, and I learned a ton about how AVL trees actually work behind the scenes. It's not just about putting data in but also it's about constantly checking and fixing the tree's balance so searches stay fast. I definitely learned the importance of those rotations and how they keep the height low.

If I had to start over, I think the biggest thing I'd do differently is **plan out my helper functions and testing more thoroughly from the very beginning.**

Blake McGahee UFID 82924917
UF – COP3530 Summer'25
6/1/2025

1. **More Unit Tests for Helpers:** I'd write small, dedicated tests for each little helper function (like rotateLeft, rotateRight, getBalance, updateHeight) *before* I even put them into the bigger insert and remove methods. That way, if something goes wrong, I'd know exactly which small piece of code is misbehaving, instead of having to debug a whole complex insertion or removal process.
2. **Cleaner Code Reuse for Rebalancing:** While my current code has rebalancing logic in insert and remove, I noticed some repetition. If I did it again, I might try to create a super generic private helper that takes a node and just rebalances that subtree, so insert, remove, and removeInorder could all call that one function, making the code a bit tidier and easier to manage.
3. **Early Input Validation:** I ended up adding input validation for names and ID lengths later. It would have been better to design that in from the very start, even before writing the core tree logic, to handle bad user input right away.