# Report Patrick Leimer

- What is the time complexity of each method (**corresponding to a command**) in your implementation? Reflect on the worst-case time complexity represented in Big O notation.
    - **Note** that the methods here refer to the methods you call for the respective commands. You don't need to analyze the time complexities of helper methods, constructors, etc.. However, you must account for helper methods' time complexities when you analyze a command that calls those helper functions. [10 points]

### TreeNode* insertNAMEID(TreeNode* root, string name, string id);

Input Validation: The checks are done on 8-length strings so these are O(1) operations and the searchID call runs in O(log n) time in a balanced AVL tree. Since the tree is balanced, finding the insertion point is O(log n) and each rotation or height update is O(1). Making then the overall time complexity **O(log n)** for each insertion.

### TreeNode* removeID(TreeNode* r, const string& ID);

Checking if the node is null and comparing IDs are O(1) operations. The function recursively searches for the node to remove. Since the AVL tree is balanced, this traversal down the tree takes O(log n) time. Once the node is found, if it has no or one child, removal is done in constant time (O(1)). Now if it has two children, the function finds the inorder successor using findMin, which takes O(log n) time because the height of the subtree is O(log n). After deletion, the tree is rebalanced along the path from the deletion point back to the root. Each call to balance is an O(1) operation.  The overall time complexity **O(log n)** for each insertion.

### TreeNode* searchID(TreeNode* r, const string& ID);

The function recursively traverses the AVL tree to find the node with the given ID. In a balanced AVL tree, the height is O(log n), so the worst-case time complexity is O(log n).

### TreeNode* AVLTree::searchName(TreeNode* r, const string& NAME);

Since every node may be visited in the worst case, the worst-case time complexity is O(n) as there might be repeated names.

### void AVLTree::printInOrder(TreeNode* r); void AVLTree::printPreorder(TreeNode* r); void AVLTree::printPostorder(TreeNode* r);

The worst-case time complexity is O(n) as you need to visit in worst case scenario n nodes.

### void AVLTree::printLevelCount(TreeNode* r)

This is an O(1) operation, gathers height from root.

### TreeNode* AVLTree::removeInOrderN(TreeNode* r, int N)

The In-Order Traversal helper function is called to traverse the entire tree and collect all IDs. This traversal is O(n). It then calls removeID, which operates in O(log n) time on a balanced AVL. Since O(n) dominates O(log n), the worst case time complexity is O(n).

- What did you learn from this assignment, and what would you do differently if you had to start over? [2 points]

I learned so much, of course the AVL and BST concepts, but also I have practiced recursion to a whole new level. I have learned about testing with catch2 and how it improves your programming experience. If I had the chance to do something differently, I thing i wouldn't. I started out with a lot of time to leave the opportunity to change around any detail that might seem not optimal for the way i want to do this project. Overall all the concepts came together and ended up with a cool project.

Test.cpp ⇒ I put it here because when i submit gradescope with the test, then the test cases don't work

```
#include <catch2/catch_all.hpp>
#include <iostream>
// Patrick Leimer 88717127
#include <iomanip>

#include "AVLTree.h"
```

```
using namespace std;

// the syntax for defining a test is below.
//It is important for the name to be unique, but you can group multiple tests with [tags].
// A test can have [multiple][tags] using that syntax.
TEST_CASE("Test 1: Five Incorrect Insert Commands", "[AVLTree]") {
    AVLTree* tree = new AVLTree();

    // 1) Insert a valid node first
    tree->root = tree->insertNAMEID(tree->root, "John", "00000001");
    // Should print "successful"

    // ID too short
    tree->root = tree->insertNAMEID(tree->root, "NameShort", "1234");
    // ID has non-digits
    tree->root = tree->insertNAMEID(tree->root, "Sam", "12xy3456");
    // Duplicate ID "00000001"
    tree->root = tree->insertNAMEID(tree->root, "JohnAgain", "00000001");
    // Name fails regex (contains digit)
    tree->root = tree->insertNAMEID(tree->root, "Jo3n", "00000002");
    // Empty name
    tree->root = tree->insertNAMEID(tree->root, "", "00000003");

    // Only the first valid insert ("00000001") should be in the tree
    vector<string> ids;
    gatherInOrder(tree->root, ids);

    REQUIRE(ids.size() == 1);
    REQUIRE(ids[0] == "00000001");
    delete tree;
}


TEST_CASE("Test 2: Three Edge Cases", "[AVLTree]") {
    AVLTree* tree = new AVLTree();

    // Remove from empty tree => "unsuccessful"
    tree->root = tree->removeID(tree->root, "00000001");
    REQUIRE(tree->root == nullptr);

    // Insert one valid node
    tree->root = tree->insertNAMEID(tree->root, "Alicia", "00000001");
    REQUIRE(tree->root != nullptr);

    // Remove a non-existent ID => "unsuccessful"
    tree->root = tree->removeID(tree->root, "00000099");
    REQUIRE(tree->root != nullptr);

    // Search for a non-existent ID => prints "unsuccessful" and returns nullptr
    TreeNode* search = tree->searchID(tree->root, "00000088");
    REQUIRE(search == nullptr);
}

TEST_CASE("Test 3: All Four Rotation Cases", "[AVLTree]") {

    SECTION("Left-Left Rotation: insert 30, 20, 10") {
        AVLTree* tree = new AVLTree();
        tree->root = tree->insertNAMEID(tree->root, "Thirty", "00000030");
        tree->root = tree->insertNAMEID(tree->root, "Twenty", "00000020");
        tree->root = tree->insertNAMEID(tree->root, "Ten",    "00000010");
        // LL rotation => root 20
        REQUIRE(tree->root != nullptr);
        REQUIRE(tree->root->id == "00000020");
        delete tree;

    }

    SECTION("Right-Right Rotation: insert 10, 20, 30") {
        AVLTree* tree = new AVLTree();
        tree->root = tree->insertNAMEID(tree->root, "Ten",    "00000010");
        tree->root = tree->insertNAMEID(tree->root, "Twenty", "00000020");
        tree->root = tree->insertNAMEID(tree->root, "Thirty", "00000030");
        //  RR rotation => root 20
        REQUIRE(tree->root != nullptr);
        REQUIRE(tree->root->id == "00000020");
        delete tree;

    }

    SECTION("Left-Right Rotation: insert 30, 10, 20") {
        AVLTree* tree = new AVLTree();
        tree->root = tree->insertNAMEID(tree->root, "Thirty", "00000030");
        tree->root = tree->insertNAMEID(tree->root, "Ten",    "00000010");
        tree->root = tree->insertNAMEID(tree->root, "Twenty", "00000020");
        // LR rotation => root 20
        REQUIRE(tree->root != nullptr);
        cout << tree->root->id << endl;
        REQUIRE(tree->root->id == "00000020");
        delete tree;

    }

    SECTION("Right-Left Rotation: insert 10, 30, 20") {
        AVLTree* tree = new AVLTree();
        tree->root = tree->insertNAMEID(tree->root, "Ten",    "00000010");
        tree->root = tree->insertNAMEID(tree->root, "Thirty", "00000030");
        tree->root = tree->insertNAMEID(tree->root, "Twenty", "00000020");
        // RL rotation => root 20
        REQUIRE(tree->root != nullptr);
        REQUIRE(tree->root->id == "00000020");
        delete tree;
    }
}

TEST_CASE("Test 4: All Three Deletion Cases", "[AVLTree]") {

    AVLTree* tree = new AVLTree();
    tree->root = tree->insertNAMEID(tree->root, "Ten",      "00000010");
    tree->root = tree->insertNAMEID(tree->root, "Five",     "00000005");
    tree->root = tree->insertNAMEID(tree->root, "Fifteen",  "00000015");
    tree->root = tree->insertNAMEID(tree->root, "Fourteen", "00000014");
    tree->root = tree->insertNAMEID(tree->root, "Sixteen",  "00000016");

    // Remove leaf (14)
    tree->root = tree->removeID(tree->root, "00000014");
    // Remove node with one child (15)
    tree->root = tree->removeID(tree->root, "00000015");
    // Remove node with two children => 10, left=5, right=16
    tree->root = tree->removeID(tree->root, "00000010");
    // Remaining IDs = 5, 16
    vector<string> ids;
    gatherInOrder(tree->root, ids);
    REQUIRE(ids.size() == 2);
    REQUIRE(ids[0] == "00000005");
    REQUIRE(ids[1] == "00000016");
}


TEST_CASE("Test 5: Insert 100 Nodes, Remove 10, Check 90 remain", "[AVLTree]") {
    AVLTree* tree = new AVLTree();

    // Insert 100 nodes
    for (int i = 1; i <= 100; i++) {
        string id = to_string(i);
        while (id.size() < 8) {
            id = "0" + id;
        }
        // Use a name with only letters (no digits)
        string name = "Person";
        tree->root = tree->insertNAMEID(tree->root, name, id);
    }

    // Remove 10 arbitrary IDs
    int toRemove[10] = {5, 20, 33, 47, 59, 60, 73, 80, 99, 100};
    for (int i = 0; i < 10; i++) {
        int val = toRemove[i];
        string id = to_string(val);
        while (id.size() < 8) {
            id = "0" + id;
        }
        tree->root = tree->removeID(tree->root, id);
    }

    // Check we have 90 nodes left by gathering the in-order list of IDs
    vector<string> ids;
    gatherInOrder(tree->root, ids);
    REQUIRE(ids.size() == 90);

    // Ensure that each removed ID is not present in the tree
    for (int i = 0; i < 10; i++) {
        int val = toRemove[i];
        string id = to_string(val);
        while (id.size() < 8) {
            id = "0" + id;
        }
        auto it = find(ids.begin(), ids.end(), id);
        REQUIRE(it == ids.end());
    }

    delete tree;
}
```