In project 4, we looked at binary search trees and their methods, testing their performance and functionality. In this project, we looked at AVL trees, which were just binary search trees that could now balance themselves if unbalanced. This was a neat feature for binary search trees to have because of the time it saved when it called its methods, in particular when inserting or removing an element from a tree.

## 1. Purpose and efficacy of test cases

### 1.1) General approach to making test cases for `__balance` and `to_list` methods

When testing the `__balance` method, I created my test cases so that the last insertion or deletion call in my test cases triggered an imbalance. Moreover, I made my test cases so that a variety of imbalances occurred that required different numbers and kinds of rotations to balance it. For each of my test cases, I asserted each of the three traversals in addition to the new to list traversal to make sure that the structure of the tree looked correct after it was balanced. I also checked the heights of the trees as another way to make checked that the tree was balanced.

### 1.2) Insertion balance (preorder, inorder, postorder, to list) test cases

For my insertion test cases, only one rotation operation was involved. Even though the left-right and right-left cases involved two rotations, I counted these as single rotation operations. I asserted all 4 traversals (preorder, inorder, postorder, and to list) and checked the balanced tree’s height to make sure that the structure of the tree was correct.

**1.3) Removal balance (preorder, inorder, postorder, to list) test cases**

My removal test cases were similar to that of my insertion test cases except that I also checked for the fact that two rotation operations could now occur. I had to create additional test cases to check all of the different case combinations that could occur with two rotation operations.

**1.4) Remaining methods**

The large majority of my test cases from project 4 already had balanced trees, so there were little to no changes required for the test cases I created previously in project 4.

2. **Analysis of the worst-case performance of every method**

**2.1) Insertion methods performance**

The worst-case performance for `__rec_insert_element` was O(log(n)) because the worst case for traversing through the tree was O(log(n)) since the number of nodes traversed was equal to the height of the tree. The worst-case for a binary search tree could no longer be staggered (increasing or decreasing order) because the tree has to be balanced. The `insert_element` method had the same worst-case performance as `__rec_insert_element` because it called this method.

**2.2) Removal methods performance**

The worst case performance for `__rec_remove_element` was also O(log(n)) for the same reasons as `__rec_insert_element`. The `remove_element` method also had the same worst case performance as `__rec_remove_element` for the same reason as `insert_element`.

**2.3) Rotate left and right performance**

The worst-case performance for `rotate_left` and `rotate_right` were both constant time.

**2.4) Balance performance**

The worst-case performance for `__balance` was constant time as well because it called `rotate_right` and `rotate_left`, both of which were also constant time.

**2.5) to_list performance**

The worst-case performance for `__rec_to_list` was O(n) because it had to visit each node in the tree. The `__rec_to_list` method also appended all of the values of a tree into a list which was constant time making the worst-case performance for `__rec_to_list` O(n). The `to_list` method also had a worst-case performance of O(n) because it called the `__rec_to_list`.

**2.6) Remaining methods performance**

The remaining methods all had the same worst-case performance as in project 4 which I had already explained in project 4.

**3. Performance analysis for this sorting approach**

The worst-case performance for this sorting approach was O(nlog(n)). First of all, fraction objects were stored in an array in constant time. After that, an empty binary search tree was created in constant time. Then, a for loop went through each fraction objects in the array and inserted them into an empty binary search tree. A for loop had to go through every fraction object in the array taking O(n) time. Inserting a fraction into the tree took O(log(n)) (explained in 2.1). Since every fraction needed to be inserted into the tree, the worst-case performance for this sorting approach was O(nlog(n)). Printing out the original and in-order traversal arrays do not affect the performance of this sorting approach since they were just there to show that the fractions had been sorted.