For my first three test cases, a tree is built with ten nodes in varying orders. The first case checks the rotation in the tree when there is a right-right imbalance almost every time there is an insertion. The second case does the opposite and checks for left-left imbalance rotation. This is a single rotation and is a simple problem for the method to solve. Lastly, the third insertion test checks the tree when there is a random set of values entered into it. Within this test, a right-left and left-right imbalance occurs and requires the use of a double rotation. For each test case, the structure of the tree was checked with **in_order, pre_order, and post_order** traversals. Since there is only one specific tree that can have the same output for all three of the traversals, the test ensures the correct values were shifted to the right places and the general ordering of the tree was maintained.

For my second set of test cases, I checked the ability of the removal method. This method should have been able to remove any values contained within the tree in any order. The tree should also have been able to adjust to remove any imbalances. Because the method is recursive, imbalances are checked for on the way back up the tree after a removal. These test cases only use one **assertEqual** check because the full structure of the tree wasn't being tested. Even after a few insertions and removals, the tree still operates the same as the insertion function in terms of the balancing operations.

In the final set of cases, the height of the tree is tested after various insertions and removals. This is to check that the heights of the nodes are being updated correctly after the rotations. For example, if the height was the same as the number of insertions, the rotation operation clearly failed and did not update the tree to become more balanced.

The worst case for many of the methods in this project goes from linear time to logarithmic time. This happens because the tree balance method is able to adjust the tree so not every value is ever on one side of the tree. Without this balancing, the tree could have linear performance and would have to traverse through each value in the tree to get to an insertion or removal point.

The worst-case performance of the **__balance** method is constant time. This function only operates recursively in the insertion and removal methods, so it only works with constant operations at any root. The balance method starts by identifying the heights of its children to check for an imbalance in constant time. Depending on the balance value, the method decides which constant time rotation is going to be performed. This rotation includes only redefining the children of each node. There can be an infinite amount of values underneath the rotation with the same performanc, so it occurs in constant time. The **__balance** method also includes the private **__update_height** method, which is mostly used as a space saver and also runs in constant time to calculate the heights of a tree after the rotation can occur.

The worst case for the insertion and removal methods is logarithmic time. If there is ever a case where the tree becomes unbalanced, the **__balance** method rearranges the tree in constant time at the source of the imbalance. This means that the largest possible number of nodes the recursive function will have to iterate through is logarithmic. The rest of the method including creating or removing a node also happens in constant time because it is recursive. The height to the tree is adjusted upon every return of a node in constant time as well.

The worst-case performance for the traversals is the same as the traversals in the unbalanced tree project. Each traversal by nature must visit every value in the tree no matter what, so it will always be operating in its worst case, which is linear time. The only difference

between these traversals is the order in which they add values to the string. Even if the tree is extremely unbalanced or not, there is no change to performance. This means the **__str__** method also has linear time performance because it calls upon the **in_order** traversal to create a string.

The worst-case performance of the **to_list** method is the same as the string traversal methods. This method works by iterating through the entire tree and appending each value on to the end of the list. Because append works in constant time, the overall performance is linear time. No matter the size or shape of the tree, the method must eventually visit every value to add it to the list representation which will eventually be returned at the end of the traversal.