TEST CASES:
I did the simple insertions that were the same for a regular BST. Just inserting and removing one element, this was to check to see that the function balanced did not impact the base cases of the tree and still worked as expected. Then I used a few test cases to check simple three node left left, left right, and vice versa rotations. This was to see that the None node floating variables were properly taken care of and no issues arose. Then instead of having variables of node none, I used actual nodes to see that the Balanced function did a good job with the rotations and was not impacted by either a none node or a regular node. I then began to insert just multiple values and checked up to see that balance was correctly fixing the tree and also the height. At first the height was not updating properly, which was impacting the way balance was working since the calc balance function depended on the height of the nodes being updated as rotations occurred. I then created a multi-level tree that had a few rotations in order to be built. The insertion aspect worked properly. Then I began by removing one node and checking to see that the height and balance worked. I continued to do so for three until we were back to a simple tree. By the multiple insertions and rotations needed. I am confident that the code has worked. The rotation of the tree kept working properly and the height also was updated whenever there was a rotation that may have lowered the height a total of one or two levels.

Performance Analysis:
Inside of the inner Node Class:
Constructor: I had references to the value of node, setting the new nodes right and left equal to None and by default whenever the node is created it has a height of one since nodes are always inserted at leaf positions.
Get_node_height: The two functions work in constant time. They do not enter any loops and just do basic arithmetic to replace the previous height of the node with a proper value.

The BST class:
Insert_element and __rins: this includes the private function, ***__rins which also calls on the balance function,*** which is the recursive function that helps find the correct position to insert the given element. In the worst case, the function will run in log(n) time. Since BST are balanced, in order to locate a specific location, the insertion is dependent on the height of the tree at that given point. Each time hacking away half of the tree. Inside, the recursive function, I call upon the get_node_height function in order to update the height of each node as the recursive call goes through, this has no impact on the time since the function, get node height, works in constant time to update the value of height for the nodes. The **__balanced** function also runs in constant time. It first calls on the calc_balance function to do a constant time computation. This then just uses if statements to see what type of imbalance has occurred and then calls on the constant function again. Each time the tree is balanced, I again call on get node height which is a constant time function to update the heights of all the nodes that have been moved or rotated.

Remove_element and __rrem: this includes the private function, **__rrem which also calls on the balance function**, which is the recursive call that helps locate the specified element and remove and update the tree properly. In the worst case, that is removing from the root with an balanced tree, that has height h. The function would run in log(n) time. The function would enter a loop, using, ***__smallest_value,*** which would locate the leftmost node on the tree rooted to the right of the root. Each time the function goes to a new level, half of the tree is taken away. The balanced function is called as well which runs in constant time. In the balanced another constant time function, calculates the. Imbalance. This has no impact on the time it takes the function to run. The get_node_height is also called during the function, but this runs in constant time in order to update the node height during the recursive call and has no impact on the overall time. The get node height is also called in balance to update the nodes impacted by rotation during the call.

In_order, pre_order, post_order, to_list: All four traversals work in the same performance time. The worst case would be a balanced tree with height n. The function would run in linear time since it would continue to execute based off of the number of nodes in the tree since it has to visit every single one of them. Then it is a matter in how the string/array is created. As a node is added the time it takes for the traversals to function is also added.

Get_height: This function runs in constant time, since the height of the root position is updated every time an insertion or removal recursive call is used. Thus, all that is needed is to return the attribute n_height of the root which is the height of tree.

Fraction: This program first must insert all of the Fraction variables into a balanced BST. This will run in O(log(n)) time. After it must print out the sorted array. This will call on the function to_list() which runs in O(n) time. So for the program to run entirely, it would run in n + log(n) performance time.