***TEST CASES:***

The test cases begin by testing solely the insertion of nodes into the Binary Search Tree. I began by testing an empty tree. To see that the traversals returned an empty bracket properly. Then I began by inserting just the root, then a child each. Each time, making sure that the traversals returned the proper tree. Once I did a height of 2, I began by inserting leaf nodes to those children 2 at a time on each and then inserting all 4. The three traversals would expectedly be very different. And by checking the return of each, I saw that the tree was being created properly. After doing so, I was fairly confident that insertion would work properly for any other numbers that were encountered.

The next set of tests concern solely the proper function of the function get_height. I began by inserting one node, then checking the height. Then two, and then kept repeating until I had a tree of height 5. I then concluded that the get_height function worked properly for all insertion-based height changes. I also checked to see that if the right side of the tree is of height 5 and then I insert an element to make the left side of the tree 6, that the tree would recognize and update the root height to be 6.

The next few test cases began to check to see that the removal function worked properly. I would first do a normal insertion and the removal checking the traversals and the height in each case to see that the tree was working as expected. I then would make a tree of height of 2 and then check both sorts of removal, the head and the leaf. This worked properly, and the height also updated accordingly. I then increased the size of the tree to a complete tree of height 3. I removed from each height to see that removal would take the leftmost value from the right subtree rooted at the point of removal. The height was checked to make sure that nothing had changed since at least on subtree would not be impacted. I then created a tree of height three that when I removed from the root, there would have to be two removal operations since the leftmost node on the right subtree had a child. The traversals checked to see that the rotation worked accordingly. And that the height was also updated properly.

By checking to see that it was working on trees of height 3, I assume that it would work on trees of any height by the concept of the principle of mathematical induction. I do not have a concrete proof but a relative confidence that it holds true for any size.

***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 is the recursive function that helps find the correct position to insert the given element. In the worst case, the function will run in linear time. Since most BST are not perfect nor are balanced, in order to locate a specific location, the insertion is dependent on the height of the tree at that given point. 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.

***Remove_element and __rrem:*** this includes the private function, __rrem, 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 unbalanced tree, that its right subtree has height n. The function would run in linear 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. This could be the farthest point in the tree from the root. Which would occur in the loop. 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.

***In_order, pre_order, post_order:*** All three traversals work in the same performance time. The worst case would be an unbalanced tree with height n. The function would run in linear time since it would continue to execute based off of the height of the tree. Then it is a matter in how the string is created.

***Get_height:*** This function runs in constant time, since the height of the root position is updated everytime 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.