My test cases for this project were broken up into four different sections to test each method in the **Binary_Search_Tree** class. These test cases focus mainly on small trees because the methods work recursively. Any method that would work with a small tree size would also work with a larger tree size with the only difference being how many times the recursive functions are called.

The first section of my tests cases focuses on the insertion method. The first case tests the most basic insertion when the root is **None** and the tree is completely empty. This is a special case that will only occur when the tree is empty. The second and third cases are testing the method's ability to insert a node at the left and right child of the tree by creating a tree with a height of two. Since the method is recursive, insertion for a child at any node will be the exact same thing as inserting at the root node. Finally, the last test case tries to insert a value that is already inside of the tree. The **insert_element** method should raise a **ValueError** which is caught by the test case. The method should also keep the tree intact and exactly the same as before the method call.

The next section of my test cases focuses on the print outputs and traversal of the tree recursively. I created an identical tree in all three tests and then called the **in_order, post_order, and pre_order** methods to compare the output. I made this tree slightly larger than the other test cases because more complexity would reveal any problems in the methods. Any issues would be magnified and very easy to spot, and the method had to use many calls to **__traversal** in order to construct the string. In addition to these tests, the **in_order** method was called for almost every test case where I used the **__str__** method.

The third section of testing aimed to check the heights of various trees after some insertions and removals. The first case checks the height of a perfect tree with both of the

children being the same height. The next two cases are similar but check the height of the tree when the children have different heights. Whether the left child's height or the right child's height was larger, the test checked that the **insert_element** method was correctly updating the tree's height using the larger height. The next tests checked the simple case of a new tree being empty and having a height of 0. It also tested to ensure that when all of the elements of a tree were removed, the height returned to 0. The final series of cases tested the height of a tree after several removals. This included removing nodes with one child, two children, or no children and checking the height afterwards.

The final series of cases tested the **remove_element** method with nodes that had a varying number of children. The first test was a special case where the method attempted to remove a value from a completely empty tree. This should raise a **ValueError** since nothing can be removed from an already empty tree. The second case tested the simplest removal where the initial root is the value that needed to be removed. No recursion was necessary for this removal. The third case was similar to the second and tested the removal of a value at a leaf node. This removal required recursion which means the method will also would for larger trees of any size. The next two tests checked the removal method for a node with a single child both on the right and left sides. Since the method is recursive this method should work no matter how far down the tree the node is. After, there are three cases meant to test removal of a node with two children. To be extra thorough, I also tested removal of trees that were perfect and trees that were unbalanced to the left and right sides to be sure that the entire tree would stay intact after removal. Finally, the last test case checked the removal method when the provided value wasn't in the tree. This should return a **ValueError** and keep the tree exactly the same as it was before.

The performance of many of the methods I created have a similar worst case when the tree is very unbalanced, and all of the values are on one side of the tree. In this case the recursive methods have to be called once for every value in the tree, or **n** times.

For my **insert_element** method the worst-case scenario is an insertion at the end of a tree that is very unbalanced. For example, if every node in the tree only has a left child, the tree height will be the same as the number of nodes. This is a worst case for insertion because the method uses the **__ins** private method for recursion. The main method begins by going through the if statements to determine which direction to recur in. Once this happens, the private method is called and begins the recursion process. Since the tree height used for the recursion will get smaller by one every time the method is called, the performance is O(n). After the recursion occurs **n** number of times, the main method continues with inserting the node with the value the user specified in constant time. Afterwards, the height for each node is updated in constant time as well. This is in constant time because the height is updated for every node on the methods way back up the tree after all of the recursive calls.

For the **remove_element** method, the worst case is also when the tree has every node on one side of the tree. The performance is similar to insertion and calls the private recursive function **__remove n** number of times, for a total performance of O(n). The removal method starts by checking the initial noe to see if it is the correct value for removal. If not, it uses recursion **n** number of times in the worst case to find the value it is looking for. After finding the value, the main method replaces the root with the correct node in constant time. This is in constant time regardless of if the removed root has zero, one, or two children. After the replacement, the height of the root is updated which occurs in constant time.

Every method that uses string traversal is in linear time are all very similar. This includes the **in_order, post_order, and pre_order** methods. These methods are in linear time because the method must iterate over the entire tree every single time no matter how the values are stacked. If the tree is unbalanced to one side in the worst case or a perfect tree in the best case, the method will still iterate through the entire thing eventually. The only difference being the order the root nodes are printed in relative to their children. All of the methods include a private method called **__traversal** which recurs through the entire tree. This iteration is set up to visit every value in the tree, so the performance is always O(n). The methods by themselves would be in constant time, but because of their use of the recursive method **__traversal**, they are in linear time.

The height method occurs in constant time regardless of the tree size or orientation because it only checks the height of the root node. This node will always be one step away at all times. Since the height value is recursively updated every time a removal or insertion occurs, there is no need for recursion or iteration through the tree.

Finally, the string method occurs in linear time because it makes us of another method in the **Binary_Search_Tree** class. The **in_order** method is called whenever the **__str__** method is called, therefore every value in the tree needs to be visited in linear time. This method has O(n) because of its call to the **in_order** method, which needs to create a new string in linear time every time it is called.