A Binary Search tree is a particular kind of binary tree. Binary search trees are similar to binary trees but have some important properties that make it different from binary trees. The values in the left subtree are less than its parent's value, and the values in the right subtree are greater than its parent's value. Moreover, the tree cannot contain duplicate nodes. I am going to talk about a binary search tree's methods and their worst-case performances. In addition to this, I am going to show some test cases to prove that these functions work.

1. Purpose and efficacy of test cases

1.1) Raising errors test cases

I first checked if my `__rec_insert_element` and `__rec_remove_element` functions raised value errors when a value tried to get inserted that already existed in the tree or when a nonexistent value tried to get removed from a tree. I expected that the string representation of the tree remained the same after raising a value error.

1.2) Insertion test cases (inorder/preorder/postorder)

I then tested inserting a varying number of values to produce a tree of size zero, one, three, and seven. I included my testing of the different tree traversal in addition to test cases for these various sizes. I tested these specific tree sizes to see if the function recurred down the right path starting at the tree's node and returned the newly inserted node at the correct position in the tree. Moreover, I tested each type of tree traversal at these various sizes to check if they recurred through the entire tree, visiting each node in the correct order. I excepted to see a string representation in the right order depending on

the tree traversal used and containing the same number of values in the string as the number of values that I inserted into the tree.

1.3) Get height test cases

Next, I tested the height of a tree after inserting no values, one value, three values, and five values. My test cases checked the tree's height after removing a node with no children, a single left child, a single right child, or two children. I expected to change or remain the same based on the type of binary search tree.

1.4) Remove test cases (inorder/preorder/postorder)

Lastly, I tested removing the root node and non-root node when it had no children, a single child on its left, a single child on its left, or two children. Just as in my insertion test cases, my removal functions had different test cases for each kind of tree traversal. I expected the value removed to not appear in the string representation of the tree and to be in the correct order based on the tree traversal I was using.

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

2.1) Recursive height method

`update_height` has a worst-case performance of O(n). It recurs through every node in the binary search tree once that results in a runtime of O(n) where n is the number of nodes in the binary search tree. `get_height` always performs at O(1). This function returns a height of 0 when there are no values in the tree and has O(1) performance.

When there is at least one value in the tree, this function returns the height of the root and has O(1) performance as well. This performance occurs because `self.height` is an attribute initialized to each node object in the tree.

2.2) Recursive insert / remove methods & minVal method

`__rec_insert_element` and `__rec_remove_element` both have a worst-case performance of $O(n^2)$. These functions both always start at the tree's root and recurs down the leaf node. This makes the number of steps these functions take equal to the height of the tree. In the worst case, this makes the height of the tree is equal to the number of nodes in the tree making the performance of these functions O(n). This can be visualized as a tree where values are already inserted in decreasing or increasing order. However, these functions must also call `update_height` to recursively update the height takes O(n) time as well making these functions run in $O(n^2)$ time. `insert_element and remove_element` both have a worst-case performance of $O(n^2)$ as well. They both call their respective recursive functions that both run in $O(n^2)$ time. The `minVal` function has a worst-case performance of O(n). This is because this function traverses down the left of the given node's right subtree until it reaches the leaf node.

2.3) Recursive inorder / preorder / postorder methods

`__rec_in_order, __rec__pre_order,` and `__rec_post_order` have a worst-case performance of $O(n^2)$. All of these functions have to traverse through each node once which has O(n) performance. Each of these traversal functions must also concentrate a string that takes O(n) time making these tree traversal functions operate in

$O(n^2)$ time. The order in which inorder, preorder, and postorder traverse through each node in the tree is the only difference between the three functions. `In_order`, `pre_order`, and `post_order` have a worst-case performance of $O(n^2)$ as well since they call their respective recursive functions that have $O(n^2)$ performance.

2.4) String method

`__str__` has a worst-case performance of $O(n^2)$. This is because this function calls one of the three tree traversal public functions that all run in $O(n^2)$ time.