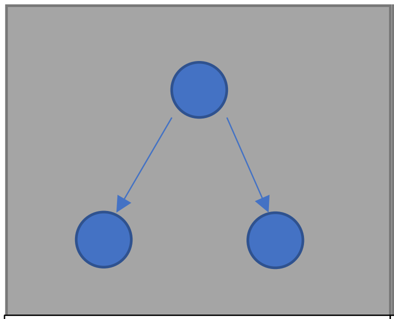
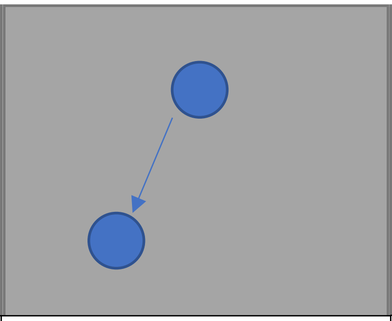
Explanation of Test Cases

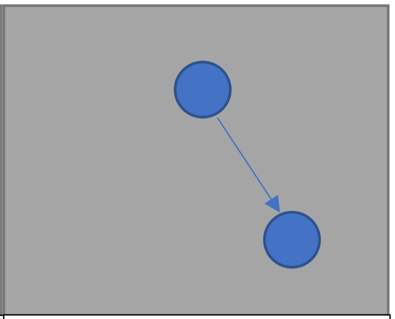
A prose writeup explaining the purpose and efficacy of your test cases

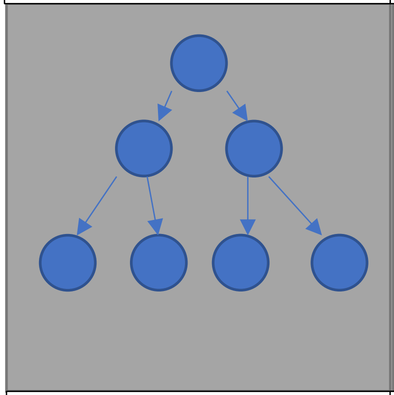
In my testing file, I separated my testing into different sections, one for each method or group of methods. For example, my height section includes both the public get_height() and my private __set_height(). I started with a preliminary test case that checked to make sure an empty tree returned '[ ]' from its string function and '0' as its height. For each public method, I also tested using strings instead of integers, but instead of duplicating all of my tests, I only included one for each, just to make sure it works. Since none of the code is different for strings, there is no need to check other cases, as it would be the same for all data types. For insert_element(), I did include three different cases, just to make sure that insertions work alphabetically, as I expected them to.

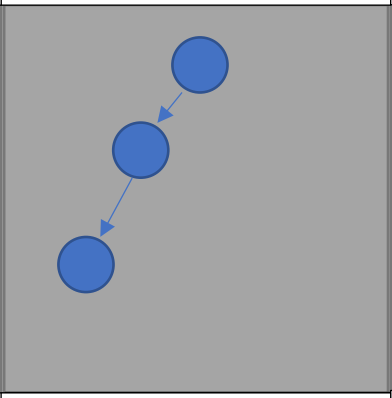
I will also include some diagrams to help illustrate what I mean when describing various trees.

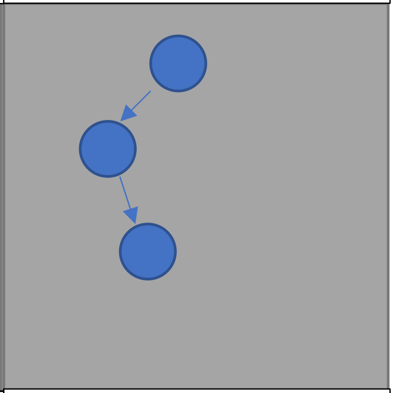

Three element full tree


Two element left leaning


Two element right leaning


Seven element full tree


Three element left leaning


Three element left leaning

Get and Set Height

First, I performed testing on my height methods. Since I had already tested get_height() for an empty tree in my preliminary test, I did not include a test for that here. I started by testing the height of a one element tree to make sure that it's height attribute initialized correctly. The rest of my tests relied on __set_height(), as it is used in the recursive cases. These tests included trees with two elements (both a left leaning and right leaning tree), trees with three elements (including a full tree and all four possible left and right leaning trees), and a full seven element tree. I likely did not need to include the testing on the seven element tree, however I wanted to include this just in case, as it is more complex than the other cases.

I also performed testing on the height after removals, as the previous test required insertions. All of this would verify that my public and recursive insert and remove methods correctly implemented the __set_height() method. I first tested to make sure that the height of a tree that is empty as a result of a removal is zero again. I then tested the height after the removal of an element with no children, an element with one child, and then an element with two children. This is because I wanted to make sure that the height decreased if there were less than two children and remained the same if there were two children.

After my code passed all of these tests, I was confident that all of my methods correctly modified height whenever needed and performed no modifications to height otherwise.

Insert

Next, I wanted to verify that the insert method also modified the tree correctly. Since I had already verified insert's height incrementing, I did not include anything about height in this section.

The trees that I performed tests on included inserting onto an empty tree, a tree with one element, trees with two elements (resulting in all five three element tree combinations), and full trees with three elements (all four possible insertions).

Additionally, I verified that ValueErrors were correctly raised with duplicate nodes. I included three tests here, one to test a duplicate found at the root, one to test a duplicate found on a deeper node, and finally one to test a duplicate found on a leaf node. This would ensure that no matter where an element is in the tree, it can be found if a duplicate is being inserted.

Since all of these tests were passed, that means that my public insert_element() function was able to correctly call the private rrins() function and that the private recursive function handled each case correctly.

Remove

Removal was the next section that I performed tests on. I had many similar tests to those that I used with insertion. These tests included removing a leaf from either side, removing a node with two children, removing a node with one child (all four combinations of left and right), and removing a lone root to result in an empty tree. Although likely not necessary, I wanted to ensure that removal of a node with two children worked correctly, so in addition to the test above, I also had a test that removed a root with two large subtrees as its children.

I also performed two tests for my error cases. These two tests included removing from empty tree and removing a value that is not present. Both resulted in a ValueError, as expected. With all of these tests performing as intended, I can now say that both the remove_element() and __rrem() methods can correctly handle each removal case.

Traversals

Although each traversal is different, I will describe them all in one section, as they each had the same tests performed on them. The only difference in the tests is the string being used within the assertEqual line. The tests that I included for these three traversals include a tree with no elements, a tree with one element, trees with two elements (both left and right leaning), and trees with three elements (two left leaning, two right leaning, and one full).

It is also important to note that any tests for in_order() are also tests for __str__(), as the string function simply calls and returns in_order().

After these last tests, I was able to confirm that the private recursive methods were all able to correctly traverse through the tree. The public methods were also able to handle cases where the tree is empty and correctly returned the final strings resulting from the traversals.

BST_Node:

__init__(): Initializing a node is O(1), as all it does it set its value to be the given value, height to be '1', and left and right children to be 'None'. All of these can be done in constant time.

Binary_Search_Time:

__init__(): Initializing a tree is also O(1), as it only sets self.__root to be 'None'.

__set_height(): Setting the height of each node has a performance of O(1), as it can access both children in constant time. All it must do is check the height of each child and set its height to be one greater than the max of the children, which can also be done in constant time.

__rins(): Although it may seem like logarithmic time performance at first, __rins() actually has linear time performance. This is because we have unbalanced trees, so in the worst case, we could have only one node for each level, despite having room for more as the number of levels increases. For example, if we insert values in ascending order, with the root being the smallest value. Then the far right node of each level will be the only node present. In the worst case, it would require traversal through every node in the tree to reach the node of the greatest value. The cost of each individual call is constant, as it either creates a new node, raises an error, or assigns a child and the height, however the call is made n times in the worst case, which is what makes it linear time. Additionally, __rins() does depend on __set_height(), but since it is constant time, then this does not change __rins()'s runtime.

insert_element(): Since assignment can be done in constant time, insert_element() does not make __rins()'s time performance any worse. Therefore, insert_element() has linear time performance as well.

__rrem(): __rrem() has the same worst case as __rins(). It may need to traverse through every node in the tree in order to reach a value. Once again, despite seeming like it has logarithmic time performance, __rrem()'s performance is linear. The cost of each individual call is also constant, but again it must be called n times in the worst case. __rrem() also relies on __set_height(), but just like with __rins(), this does not change the runtime.

remove_element(): Just like how insert_element() has the same performance as __rins(), remove_element() has the same performance (linear time) as __rrem(), as it is only assigning self.__root to be the returned value of the recursive function.

__rin_order(): Each individual recursive call is linear time, as string concatenation has linear time performance. However, since __rin_order() must traverse through and be called on each node in the tree, this linear time performance call happens n times. Therefore, __rin_order() has O(n^2) performance.

in_order(): Since __rin_order()'s performance is quadratic, so is in_order()'s. It either returns '[ ]', which can be done in constant time, or calls the recursive call, which has quadratic time performance.

__rpre_order(): __rpre_order() has the same lines as __rin_order(), but in a different order, as that is what differs between the traversals. However, this means that they have the same time performance of O(n^2) for the exact same reason.

pre_order(): Since __rpre_order() has O(n^2), so does pre_order(), as it is the exact same situation as in_order() with the conditional and returning '[ ]' or the return from the recursive call.

__rpost_order(): Just like with __rpre_order(), __rpost_order() has the same lines as __rin_order(), but in a different order. As a result, __rpost_order() also has quadratic time performance, again for the same reasoning.

post_order(): For the same reasoning as before, post_order() has the same performance of O(n^2) as __rpost_order().

get_height(): Retrieving the height has a performance of O(1), as the tree has a reference to the root and the root has a reference to its own height. All that get_height() does is return the height of the root that it accessed in constant time.

__str__(): Since in_order() has a performance of O(n^2) and the string method returns the value from in_order(), __str__ also has quadratic time performance.