Purpose and Efficacy of Test Cases

For my testing file, I copied the testing file for project 4 and then altered or added tests when I felt necessary. So that I do not repeat myself from the previous writeup, I will only explain these changes and additions that I made, but I will still make note of all of the tests for completeness.

First, I kept the preliminary empty tree test that included testing the string and get_height() methods, except I also included a to_list() test as well. This would make sure that all three methods handle the empty tree case properly.

Get and Set Height

In this section, I included all of the same tests with the addition of one more test aimed towards verifying __balance(). This test was performed on a tree with seven elements where each of the elements that is inserted is lower than the first value. This way, __balance() must keep calling __rotation() to eventually reach a tree of height 3.

Additionally, the heights of some of the trees in this section needed to be changed, as __balance() caused them to perform one or more rotations. For example, there were left and right leaning trees with three elements and a height of 3 that changed to a height of 2 after rotating. By including these tests, I am making sure that the height is always updated whenever necessary. This also verifies the height aspect of other methods, including __rins(), insert_element(), __balance(), and __rotate().

Insert

With my insert tests, I did not need to add any tests. In my previous tests, there were already checks for both single and double rotations on the left and right side, so there was no need to repeat anything. However, since this is only checking insertions, I only verified that the insertions themselves worked, and must wait until the traversal section to verify that the rotations did in fact happen.

In terms of changes, when checking situations that raise ValueErrors, since I check the height afterwards, I also had to change some of the heights.

Remove

Unlike my insert section, I did have to add six tests to my removal section. This is because I had no removal tests that resulted in single or double rotations, so I made two for each, one for the left side and one for the right side. Additionally, two of the tests that I added had floaters that were not None to make it easier to visualize any mistake that may happen with the floaters. Again, this only checks to make sure that removal works in a situation where a rotation happens and does not actually check if the rotations happened, so that will be done in the traversal section as well. I did not need to change any heights for the ValueError subsection.

Traversals

This section is where I made the most changes. In addition to the three traversals, I also added the to_list traversal with all of the same tests as the others. The only difference these tests had was the assertEqual statement.

With the other traversals, I did decide to add several tests, however, as I felt that maybe my testing was incomplete from project 4. For each traversal, I added eight tests. These all focused on removal and included removing a leaf from a tree with three nodes, removing a leaf from a tree with four nodes to cause a single rotation, removing a leaf from a tree with four nodes to cause a double rotation, and removing a leaf from a tree with four nodes to cause a single rotation with a non-None floater. Each of these tests had a left variant and a right variant as well.

The removal from a tree with three leaves was the test I felt could have been included in project 4, as it would uniquely identify a tree after a removal, which was not included previously. The remainder of the tests were to ensure that the rotations did happen as they were expected to, and the floaters relocated to the correct place. Since the combination of the traversals can uniquely identify a specific tree, I used this to confirm that that was the case. Of course, I did include these tests for to_list as well, verifying my testing for all four of the traversals.

Analysis of the Worst-Case Asymptotic Performance

__BST_Node:

__init__(): This method is unchanged from project 4.

Binary_Search_Tree:

__init__(): This method is unchanged from project 4.

__set_height(): This method is unchanged from project 4.

__get_node_height(): This method checks if the node is None and then returns either '0' or the node's height, all of which can be done in constant time.

__rotation(): This method relies on accessing the children and grandchildren of t. Since we store the children as attributes, we can access them in constant time. Additionally, this method relies on __set_height(), however since this is also a constant time operation, so is the entire __rotation() method.

__balance(): This method involves using __get_node_height() with conditionals to then determine the necessary rotations. Since __get_node_height() and __rotation() are both constant time operations, then so is __balance().

__rins(): This method is almost the exact same as project 4, except it returns self.__balance(node) instead of node. However, since __balance() ensures that the tree stays within a height difference of 1 between children, it's performance changes from linear to logarithmic time. Additionally, __balance() can be done in constant time, so it does not make the performance worse.

insert_element(): Just like how __rins() changed from linear to logarithmic time, insert_element() does too, as it is only dependent on __rins() and constant time assignment.

__rrem(): Since __rrem() still has the same worst case as __rins(), __rrem() also changes to logarithmic time because of __balance().

remove_element(): Again, like how insert_element() relies on __rins(), remove_element() relies on rrem(), so it also performs in logarithmic time.

__rin_order(): This method is unchanged from project 4.

in_order(): This method is unchanged from project 4.

__rto_list(): __rto_list() is implemented slightly differently from the other traversals. Instead of using string concatenation, it appends each node's value to the given list. Appending is a constant time operation as opposed to a linear time operation, so __rto_list() has linear time performance. This is because it must still traverse through the tree and visit each node, but since children are stored as attributes, visiting each node is only linear time.

to_list(): Since to_list() creates an empty list and then calls __rto_list(), it has the same performance as __rto_list(), which is linear time.

__rpre_order(): This method is unchanged from project 4.

pre_order(): This method is unchanged from project 4.

__rpost_order(): This method is unchanged from project 4.

post_order(): This method is unchanged from project 4.

get_height(): This method is unchanged from project 4.

__str__(): This method is unchanged from project 4.

Fraction:

For my fraction main section, in my first test, I created twenty-five Fraction objects to illustrate my sorting. In order to sort them, I had to first insert each fraction into the tree and then call to_list. Since insert_element() has logarithmic performance and I must insert each fraction one at a time, this step has a performance of O(nlog(n)).

Then, I called to_list() and assigned the return value to a variable, in_order_rep. Since to_list() has linear time performance and it was the only thing I did in this step, then as a whole this step was also done in linear time. Of course, I then printed out my results, showing first the original fraction order and then their sorted order.

In order to demonstrate __eq__, I had a second test that had two fraction objects: $\frac{2}{3}$ and $\frac{4}{6}$. Since these two fractions are different, yet equivalent, I felt they would be good for demonstration. I then had a try statement where I used the same code as my first test with the except statement catching a ValueError. This test correctly printed that the ValueError happened, which meant that my private __lt__, __gt__, and __eq__ methods all functioned properly.