Overview:

An AVL-Tree is a self-balancing tree which keeps data stored in a most efficient and effective way. Based off of the binary search tree, an AVL tree keeps the tree as full as possible, by making sure that the left/right side of the tree does not have a height equal to two in comparison to the other's side. In order to accomplish this, each level of the tree is kept in check by a height variable. This allows for the tree's height to be updated in constant time, allowing for the checking the balance to be easier. If the left side has a height of 5, but the right side has a height of 3, the tree will initiate rotations of the sub-tree's left side, to see which part has the greatest height. It then updates this side, and rotates to the right, so that a balance of the tree is maintained. Before going into the performance analysis, I will next talk about the unit testing.

Test Cases:

BST_Tree.py

def setUp(self): This initializes the variable equal to the **Binary_Search_Tree().**

def test_Empty_Cases(self): This tests for empty cases, in all of the traversal methods.

def test_Remove_Method(self): This method tests to remove an element inside of an empty tree, raising a value error as a result.

def test_Height_Insert_Remove(self): This method inserts a value into the tree, tests to see if the height of the tree is working, and then removes the value and tests it again.

def test_Left_Tree_String_Output(self): This method tests the left side of the tree, making sure all of the values are inputted correctly and the traversals are outputted correctly as well.

def test_Right_Tree_String_Output(self): This method tests the right side of the tree, making sure all of the values are inputted correctly, and outputted correctly for every traversal.

def test_Remove_Node_With_One_Child(self): This method tests to remove a node with one child, making sure the removal process happens properly, and the output is okay after the removal is done, for every traversal.

def test_Height_Of_Tree_And_String(self): This tests to make sure the height of the tree is working, and the strings are outputted properly for each traversal.

def test_Height_Of_Tree(self): This tests to make sure the height of the tree is properly working.

def test_Insert_Value_Already_Stored_In_Tree(self): This method tests that an error is raised, if an individual tries to insert an element already stored in the tree.

def test_Value_Error(self): This method tests to make sure that removing a value from an empty tree raises an error.

def test_Removing_Leaf_Node(self): This method tests removing a single leaf node, and makes sure the string outputs are okay for the traversals.

def test_Removing_Node_With_One_Child(self): This method removes a node with only one child, an makes sure the removal process happens properly, in addition to the string outputs being okay for each traversal.

def test_Removing_Leaf_Nodes_And_Testing_Height(self): This method tests to removing leaf nodes, and also makes sure the heights are being updated properly.

def test_Removing_Value_Not_In_Tree(self): This method tests the removing of a value which is not in the tree, and makes sure the value error is raised as a result.

def test_Inserting_Value_Already_Inside_Tree(self): This method tests to make sure a value error is raised, when the user attempts to insert a value which is already inside of the tree.

def test_Removing_Nodes_Two_Children_And__Getting_Height(self): This method tests to make sure that you remove a node with two children, and can update the height as a result. This helps make sure the removal process works, and the height process also is working.

Next I will be going through the performance analysis of the AVL Tree.

Performance Analysis:

Fraction.py:

class Fraction: This class imports from **Binary_Search_Tree**, the **Binary_Search_Tree** library, and from **random**, the **randint** library. The **randint** is used so the fractions can have randomly assigned values as both the denominator and numerator. The values range from 1 to 1000 and were chosen to not be duplicated. The fractions are initially placed into an unsorted list named **fractionList**, printing an initial string so that the user can see the initial values that were chosen. There is a secondary variable, which calls the **Binary_Search_Tree()** and has every element inside of the unordered list, placed into the tree. Then the elements are sorted and outputted using the **to_list()** method, and finally displayed for the user's viewing. The performance of this method is **O(n)**, which will discussed further in the AVL-Tree portion of the analysis. The sorting of the elements is **O(n*log(n))** as the insertion is done in **O(log(n))** time and you do it **O(n)** times.

Binary_Search_Tree.py:

def __init__(self, value): Initializes the variables used in the Node class, value for the value, right child, equal to "None", left child equal to "None", the height "equal to "1". These will be used to initialize the creation of a node.

def _retrieve_balance(self) [O(1)]: This method is used to check the balance of the children, first going seeing if we have both children, right child, or left child. It then returns the height of them to the root. The performance of this method is **O(1)**.

def _every_height(self) [O(1)]: This method checks the height of the tree, and returns the value 1 (for the root) as well as if there are any children, and their heights as well. The performance of this method is **O(1)**.

def __balance(self, current_node) [O(n)]: This method checks the balance of the tree, passing in the node as a parameter. If the tree has a balance of a range between +2 and - 2, then it just returns the node value. However, if the balance is equal to 2 or negative, then it has to do rotations of the children (depending upon the value +2 for right, -2 for left), and then return the new values to the root node. The performance initially is **O(n)**, however the rotations (left or right) are happening in constant time, it depends on the amount of nodes being utlized.

def _rotate_right(self, node) [O(1)]: This method rotates the nodes to the right, and has a parameter passed in, which is the node we are going to rotate in order to become the new left node. In addition, it has another variable which is equal to the right child of the passed node, and that will become the new parent node. The method after, after completing the rotation, updates the height of the nodes, and returns the new parent node to the root. Performance of this method is in constant time **O(1)**.

def _rotate_left(self, node) [O(1)]: This method rotates the nodes to the left, and has a parameter pass in, which is the node we are going to rotate in order to become the new right node. Additionally, it has a second variable which is set to become the new parent node, and is initially equal to the passed node's left child. After doing the rotation, the method updates the height of the nodes information and returns it to the root node. Performance of this method is in constant time **O(1)**.

def __init__(self): This method was updated from the previous project, to include the variable of the tree, which is initially set equal to 0.

def insert_element(self, value) [O($\log_2$(n))]: This method has a parameter passed into it, which is a value to be inserted into the tree. It also was updated and modified to include a new recursion insert as well as the self.__tree variable, in retaining the height of the root. The initial performance for this method is **O($\log_2$(n))** in worst case, assuming we are to not include the **_retrieve_height()** invocation. This method does invoke the **_recursive_insert()** method.

def _recursive_insert(self, current_node, value) [O($\log_2$(n))]: This method has two parameters passed, the node and the value to be inserted. The method also goes through and compares the values of the left and right children of the root. Because it is a binary search tree, the time complexity for this private method is **O($\log_2$(n))**. The base case is when the root node is either empty or no longer covered under the public method.

def remove_element(self, value) [O($\log_2$(n))]: This method has a passed parameter of the value to be removed. Additionally, it has the root equal to the private recursive method, which it invokes to find and remove the value inside of the tree. The worst case performance for this method is **O($\log_2$(n))**, not taking into account the **_retrieve_height()** method. This is similar to that of the **insert_element()** method, but has there are additional cases which are taken into consideration.

def _recursive_removal(self, current_node, value) [O($\log_2$(n))]: This method has two parameters passed in, the current_node and the value. If the current_node is empty, then a value error is raised, if not then it checks to see if the value is equal to, greater than, or less than the

node being stood upon. If it is equal, then it invokes a secondary removal method **(_remove()),** and updates the balance of the tree to the root. If the value is greater than, it descends right, or less than it descends left. The private recursion method has to go through the tree, cutting half every time, since it is a binary search tree. As a result, the performance for this method is **$O(\log_2(n))$**.

def _remove(self, current_node) [$O(\log_2(n))$]: This private remove method has a parameter passed in, that deals with node going to be removed. It checks to see if the node has any children, if it does, then the node returns either left child, right child, or None (no children). Additionally, this method creates a variable to the right child of the node, and has that value equal to the passed in node. After this, it has the right child recursively go through the removal method, and updates the root node with the new information. The performance for this method is **$O(\log_2(n))$**.

def in_order(self) [O(n)]: This method uses two private methods, to receive the desired output of the "in-order" format. It initially uses the **_in_order()** method which has a base-case being considered, that then concatenates the string as well as formats it. However, this invokes a secondary method **_print_in_order(),** which is the private recursive function that strings everything together. The performance for all three of these is **O(n)** as worst case, since it has to do a walk of the entire tree concatenating the strings together.

def pre_order(self) [O(n)]: This method, similar to **in_order()** uses two private methods, to receive the desired output of the "pre-order" format. It initially uses the **_pre_order()** method which has a base-case being considered, that then concatenates the string as well as formats it. However, this invokes a secondary method **_print_pre_order(),** which is the private recursive function that strings everything together. The performance for all three of these is **O(n)** as worst case, since it has to do a walk of the entire tree concatenating the strings together.

def post_order(self) [O(n)]: This method, similar to both **in_order()** and **pre_order()** uses two private methods, to receive the desired output of the "post-order" format. It initially uses the **_post_order()** method which has a base-case being considered, that then concatenates the string as well as formats it. However, this invokes a secondary method **_print_post_order(),** which is the private recursive function that strings everything together. The performance for all three of these is **O(n)** as worst case, since it has to do a walk of the entire tree concatenating the strings together.

def get_height(self) [O(1)]: In keeping this method constant, it is stored under the Binary_Search_Tree **__Init__** constructor as self.__tree. Every time the **insert_element()** or **remove_element()** methods are called, the private method **_height()** is invoked, so it can be calculated in a recursive manner, under the previously mentioned methods. This creates an impact of the performance for both methods, but it does help the **get_height()** method to be in constant time, as it only returns the value stored in the attribute of self.height. The performance of this method is **O(1)**.

def _height(self, current_node) [O(n)]: This method is privately recursive, and allows the interpreter of to visit all of the nodes located within the AVL tree, the worst-case performance is **O(n)**.

def to_list(self) [O(n^2)]: This method places all of the nodes in a list, and then returns the list in the order which it picked them up. It invokes the **__in_order_recursion()** method, to place them in order, recursively, but that is pretty much the extent of the method. It goes through every node, so the performance of this method is **O(n^2)**.

def __in_order_recursion(self, current_node) [O(n)]: This method was created so that the tree could be presented in-order, but as a list. Because it has to go through every node recursively, and place the children in proper order, the performance for this method is **O(n)**.

def __str__(self): This method returns the **in_order()** method.

Conclusion:

In comparing the AVL-Tree to the Binary Search tree, the performance is improved, based upon the ability of keeping the tree balanced, which reduces the search of going through the entire tree in linear time, by allowing insertions being in **O($\log_2$(n))** time. Comparing the two structures, though they both are similar, the AVL-Tree definitely is better overall with regards to making things more efficient and affordable when storing and accessing data.