Overview:

A binary search tree is a data structure that stores data based using a “tree” format, where an initial value is stored into the “root” node (top of the tree) and then additional values are stored based upon whether the value is greater or lesser than. If it is greater, it will be stored in the right side of the root node, or if it is lesser than, then it will be stored in the left side of the root. Additionally, if a value is greater than the root, but lesser than it’s right child it is stored in the left leaf node, in a similar fashion if a value is lesser than the root, but greater than it’s left child, it is stored in the right leaf node. Because of the structure and how it stores it’s data, the performance for the most part is **$O(\log_2(n))$**, but before going in-depth to the performance analysis, I will next discuss the test cases.

Test Cases:

def test_Empty_Cases(self): This test sees if the tree is empty, if it is, then it returns an empty string (as there are no elements).

def test_Remove_Method(self): This tests to see if the removal function works properly. More importantly, it looks to see if the error works, when trying to remove an element, when the tree is completely empty.

def test_Height_Insert_Remove(self): This tests to see if the height method works properly, in addition to the removal method. We first get the height, add an element, get the height again, remove the element and check the height one last time.

def test_Left_Tree_String_Output(self): This tests to make sure the left side of the tree inserts properly, and the output is displayed in a correct manner as well.

def test_Right_Tree_String_Output(self): This tests to make sure the right side of the tree has elements inserted properly, and the output is displayed in a correct manners as well.

def test_Remove_Node_With_One_Child(self): This tests to make sure the elements are properly inserted, in addition to removing a node with a singular child, making sure the child takes it’s place.

def test_Height_Of_Tree_And_String(self): This tests the string output for the tree, in addition to making sure the height works properly, with regard to the method being implemented without any issues.

def test_Height_Of_Tree(self): This test’s purely to make sure the height of the tree is working, as well as the elements are inserted into the tree.

def test_Insert_Value_Already_Stored_In_Tree(self): This test’s to make sure that a value which has already been stored into the tree is not stored again. It should raise a ValueError if the methods are working properly.

Next I will be discussing the performance analysis of the data structure.

Performance Analysis:

Binary_Search_Tree.py:

class __BST_Node: This class holds attributes of the node, including height, which will be used to create the binary tree.

def insert_element(self, value) [O(log$_2$(n))]: This method grants the user the ability to insert an element at their choosing, whether it is a value to create the right, left or root node for creating the tree. Additionally, if the root node exists, this method will then insert the element as a right or left child to the root, or recursively continue until the base case (when None is found). The Big-O performance for this method is **O(log$_2$(n))**, though if the method for getting height is invoked, then the performance will be amortized to **O(n)**.

def _insert(self, value, current_node) [O(log$_2$(n))]: This is a private recursive method to insert an element into the tree. The Big-O performance for this method is **O(log$_2$(n))**, due to it needing to check both the left-children and right-children in order find the node needing to be inserted.

def remove_element(self, value) [O(log$_2$(n))]: This method first checks if the root node is empty. If so, then it returns a Value Error, but if not, then the method goes through the entire tree looking for the element. The overall Big-O performance for this method is **O(log$_2$(n))**, as it traverses through every leaf searching for the element to be erased. Additionally, it also will be amortized to **O(n)**, with invoking the method for getting height.

def _remove(self, current_node, value) [O(log$_2$(n))]: This is a private recursive method which is used to remove an element. First it checks if the root node is empty, otherwise it will raise a value error. It then traverses through every single node (on both sides) before finding the value and removing it. For this reason the Big-O performance is **O(log$_2$(n))**, but will be amortized to **O(n)** if invoking the method of receiving the height.

def in_order(self): This method returns the private method for which the actual processing occurs for the in-order string.

def _in_order(self): This is a private method which deals with stringing the values stored in the tree, and returning them in the appropriate style. If the tree is empty, it will return the appropriate brackets with a single space between them. If not it then calls the private recursive method to find all elements in the tree, and then places them all together in a single string variable, slices them, and returns the string variable for public viewing.

def _print_in_order(self, current_node, string_in_order) [O(n)]: This recursive method is used for going through the tree and processing the data. It sees whether or not the tree has more than one variable (root), if not, then returns 1, but if it does, then it checks the root to see if the tree has a left child, right child or both. This then keeps on recurring until base case ("None"). Once complete, it strings the values along the way into one singular variable, and then returns them into the appropriate format for completion. The Big-O performance for this is **O(n)** as it traverses through every node, finding all elements and stringing them together.

def pre_order(self): This method simply returns the private method which is used to doing all of the processing of the pre-order stringing.

def _pre_order(self): This method is similar to the in-order version, where it calls a private recursive function to process all of the data. It additionally checks to see if the root node is equal to None, if it is, then it returns a value error, otherwise it creates a string variable, as it set to an open bracket, and then goes through the private recursion to process data and return the string variable.

def _print_pre_order(self, current_node, string_pre_order) [O(n)]: This private recursive method processes all of the data in the tree, and returns them in pre-order. It checks to see if the root node has a right child, left child, or both children. Additionally it goes through the tree until the base case (None) occurs. Upon hitting base case it goes through the other side of the tree and does the same thing, which then is returned to the previous method, as it is strung together. The Big-O performance is **O(n)** as it traverses through each of the leaves processing each element, stringing and returning them for display.

def get_height(self) [O(1)]: The height of the tree is stored as an attribute in the BST class, and therefore is constant time of **O(1)**.

def _height(self, current_node) [O(n)]: This is a private recursive method for retrieving height of the tree. It sees if the tree has only right children, left children, or both children. If it has either of the sides, it will return the max of the side plus one (for the root). If it has both, it returns the max of both of the sides and then adds one (for the root). Just as the equation which we were given, in similar form. If the tree does not have any, it will simply return 1. The Big-O performance for this method is **O(n)** as it has to go down all of either side, or both sides in retrieving the height of the tree.

def post_order(self): This method simply returns the private method which processes the data for the user to see.

def _post_order(self): This is a private method for the post-order, it first checks to see if the tree is empty. If it is, then it returns the brackets with a singular space between the two of them. If not, then it creates a string variable, calls the private recursive method and returns the completed string-variable with a closing brace.

def _print_post_order(self, current_node, string_post_order) [O(n)]: This is a private recursive method for the post-ordering of the binary search tree. It checks to see if it has either a left child, a right child, or both children. Then it goes through the correct function and processes the data, stringing everything along the way. Once done, it returns the string to the private method which calls it. The Big-O performance is **O(n)** as it has to traverse through each and every leaf of the tree, processing the data.

Conclusion:

The binary search tree is a unique structure as it halves the data stored every time, since the values are based off of the root node. Comparing this data structure to the previous structures which we have looked at so far in the course, the binary search tree is good when it comes to storing data, as it allows for accessing data to be pretty fast and efficient. However, given that the array deque still has constant time for most of its methods, it appears that the binary search tree is still secondary when comparing the two.