Binary Search Tree

get_height(), O(1): In order to keep this public method constant, the height is stored under the __BST_Node attribute so that every time the **insert_element()** or **remove_element()** method is called, a private **_height()** method is invoked so that the height can be recursively calculated under the **insert_element()** and **remove_element()** methods. Although this will impact the performance of the mentioned two methods, the work-around guarantees the **get_height()** method to be constant time since it will only return an already stored-value under the attribute self.height. For the performance of **_get_height()**, the recursion forces the interpreter to visit every single node within the tree, therefore giving the worst case performance of O(n).

insert_element(), O($log_2(n)$), amortized to O(n) if include the _get_height() invocation and worst case scenario(all nodes on the left/right): The **insert_element()** invokes the main **_insert_recursion()** method. In this method, the recursion traverses and compares values either on the left or right side of the root. Since this is a binary search tree, the time complexity for this private method is O($log_2(n)$). The base case where the root node is empty is covered under the public method.

remove_element(), O($log_2(n)$), amortized to O(n) if include the _get_height() and worst case scenario(all nodes on the left/right): The method takes on a similar nature as to **insert_element()**, but there are now more cases to considered, like when the removed node has left/right child or two children. The method also invoke the private **_remove_recursion()** method to consider the cases. The time performance is accounted for which side the value is located, giving the recursion method O($log_2(n)$).

Each depth-first traversals (**in_order(), pre_order(),** and **post_order()**) are organized so that it takes two private methods to invoke the recursion and produce the desired string formatting. For example, for the **in_order()** method, the content invokes the **_in_order()** private method, where the base case is considered and begins the string concatenations and formatting. To get the desired the ordering of the node values, another private method, **_in_order_rec**(), is invoked within the **_in_order()** method, leading to the recursion of the string format. Essentially, the order of the recursion is extremely important, but it all begins at the root node. Depending the type of depth-first, the recursion can concatenate the node value depending on the node.left or node.right value, it can enter the recursion again, but this time, the node attributes (left or right) becomes the "root node" in this traversal. This gives the worst-case time complexity of **O(n):**

"For a Graph, the complexity of a depth-first traversal is O(n + m), where n is the number of nodes, and m is the number of edges. Since a binary search tree is also a Graph, the same applies here. The complexity of each of these depth-first traversals is O(n+m). Since the number of edges that can originate from a node is limited to 2 in the case of a Binary Tree, the maximum number of total edges in a Binary Tree is n-1, where n is the total number of nodes. Thus, the complexity then becomes O(n + n-1), which is O(n)".[i]

breadth_first(), **O(n)**: This method is not a recursion, but rather uses ADT Queue from the previous project to implement horizontal traversal on the binary search tree. The method first

enqueue the self.__root and then considers its subtree from left to right. If there is a value, add the value to the string. The time complexity is O(n) since the method visits every node once.

---

[i] https://stackoverflow.com/questions/4547012/complexities-of-binary-tree-traversals