Writeup – Project 4: It's Just a Jump to the Left and a Step to the Right

In Binary_Search_Tree.py, an unbalanced binary search tree was implemented with different methods within the class. In this program it was outlined with public methods and private methods were added to make the recursive algorithm possible, that of : ins_rec_elem, rem_rec_elem, rec_in_order, rec_post_order, rec_pre_order.

Insert

For the insert method, it inserts a new node as a leaf node in the binary search tree. However, when the tree is empty, the value is set as the root. If the insertion causes the tree to add another level, the size of the height was incremented. This method performs in O(log(n)) in respect to height and n nodes.

Removal

For the remove method, it removes a specified node within the tree. If it was removed from a parent with only one child, the node is simply removed. However, if the node that is being removed is the root or a parent node that possesses two children, the find_right_min_node method is called. find_right_min_node finds the smallest value node on the right and replaces it with the removed value. This method performs in O(log(n)) in respect to height and a node. The find_right_min_node method was constructed to support the remove_element method, in the scenario when the node is removed at the root or a parent with two children. As this method traverses the node from the root to the leaf like a linked list it performs in linear time.

Height

For the get_height method, the method obtains the correct number of levels in the binary search tree. This method initializes by giving the height a value of 1, since the height of the subtree rooted at a newly created node is always 1. The get_height method operates in constant time.

Recursive Methods

From the recursive methods: rec_ins_elem, rec_rem_elem, I was able to find out that the worst case scenario operates in linear time. When the binary tree becomes more unbalanced, the insert and removal performance became worse and eventually began to operate in a linear fashion, O(n) where n is the number of tree nodes. When the tree becomes more unbalanced it begins to look like a linked list, where it has to traverse from the root to the lowest level of the node in the tree. For the three recursive depth first traversal methods: pre_order, post_order, and in_order, each node has to be traversed. Since we have n nodes, the maximum number of edges is (n-1) and thus, it operates in linear fashion – O(n). In the breadth_first method a queue was implemented to store the nodes starting with the root to the left child to the right child and so forth. For additional level of the tree that is inserted to the queue, it removed as well, and the queue eventually became empty. Like the depth first traversals, the breadth_first method operates in O(n) with n being the number of nodes.

Test

The test code tested the insert and remove methods, and the traversal methods: in_order, post_order, pre_order, and breadth_first. The purpose of testing the insert method is to see if the inserted alue is placed correctly in the tree and whether or not the algorithm correctly raises a ValueError when the inserted node is already within the tree. Through testing the insert method, the traversal methods were also tested to see if in the method in_order performed properly by checking if the nodes are in placed in increasing order. Additionally, the height was also tested to see if it incremented, decremented, or stayed the same when a node is inserted. The test code also tested to see if the nodes are correctly traversed for pre_order and post_order traversals. For the depth first traversals, it tested the breadth_first method to see if it imported queue, enqueue, and dequeued values in and out of the queue properly. In the removal method, it tests to see if nodes at the end were removed and decremented height when height on the other side of the tree is lower. The test code also goes over the scenario when the root or a parent node that has two children is removed. As wanted, the test case shows that it replaces the root or the parent node with the minimum value on the right side of the tree.

Conclusion

In conclusion, we can see that a balanced binary search tree operates logarithmically when the tree is full. The worse-case scenario for the balanced tree is when it is unbalanced as it results in a data structure similar to that of a linked list that operates in linear time.