**Writeup – Project 5: AVL Search Tree**

In Binary\_Search\_Tree.py, a balanced 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, \_\_balance, and \_get\_balance.

**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 must 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.

**Balance Methods**

The balance method recursively balances the tree after each insertion and removal of a node. When the difference between the right and left side of the parent was +/- 2 it called one of the rotating functions which rotated it to balance the tree. The rotate right function is called when it has a value of -2 and the rotate left function is called when it has a value of +2. In the double rotations, it ensured that the relationship between both sides of the tree are between +2 and -2. By ensuring this, the program was able to call rotation when necessary. The rotation methods operate in constant time the insertion and removal methods operate in O(log2(n)) time. Another method that was added was \_to\_list, which is like the in-order traversal methods, however, rather than returning a string, it returns a list recursively.

**Testing**

BST\_Test tested the insert, remove, in-order, post-order, pre-order, and breadth first methods. To test the insert method, the test code ensured that the inserted node was placed correctly in the tree and whether a ValueError was raised when a node of the same value is already in the tree. By testing the insert method, it also tested the traversals methods that of in-order to see if the nodes are properly placed in increasing order. Additionally, by testing insert, it also made sure that the height was incremented accordingly. To test the removal method, the code made sure that the node with one child is removed and decreases the height if necessary (when the other side has a lower height). The test code also made sure that when the root node or a parent node with two children is removed, it correctly replaced the root or parent node with the minimum value on the right side of the binary search tree and decremented the height accordingly. The test also made sure that the nodes were traversed properly in the post-order and pre-order traversals. It also tested the breadth first method to make sure that it correctly imported our queue, enqueue, and dequed values correctly in and out of the queue in the form of a string.

**Fraction**

In the program fraction, it implemented a function of greater than, less than, or equal to, to sort the fractions in order. Since they are in terms of inserting order, it operates in O(n) time.

**Conclusion**

The binary search tree in its best-case scenario is logarithmic when the tree is full. The worse-case scenario for the unbalanced tree is when it is unbalanced O(n), resulting in a data structure like that of a linked list.

Writeup – AVL Trees

In the program AVL\_Binary\_Search\_Tree, it extended our BST implementation of the Binary\_Search\_Tree.py. This guaranteed time performances of O(log n) for the insertion and removal methods of our balanced tree program.

The code includes private methods to support the recursive algorithms such as: insert\_element, remove\_element, and recursive traversals such as in\_order, post\_order, pre\_order, and breadth\_first. However, one addition to this program is that of the \_\_balance method and rotation methods to balance the AVL tree. Instead of returning t, we returned self.\_\_balance(t).

In the insert method, the code consists of a method that inserts a new node as a lead node in the binary search tree. When the tree is empty, we set the new value as the root, othersie, it checks to see if the values inserted are greater or less than the root. It increments the size of the eight as a new level in the tree is reached. This method operates in O(log(n)) in respect to height and n nodes. It then returns self.\_\_balance of the node(t).

In the removal method, the code consists of a method where a specified node in the tree is removed. If it is removed from the parent with one child, with itself being linked to that same parent, then the node is removed. However, if the node is removed at the root or a parent in a subtree with two children, it calls the find\_right\_min fuction, which locates the minimum node value on the right and replaces it with the removed value. This method operates in O(log(n)) in respect to height and n nodes. It then returns self.\_\_balance of the node (t).

The find\_right\_min method was implemented to support the remove\_element method in the case when the node removed is a parent is removed at the root or a parent with two children in a given subtree. Since the method traverses the node on a path from the root to the lead, this function operates in O(n) time.

In the two recursive methods: ins\_rec\_elem and rem\_rec\_elem, the worse case scenario operates in O(n) time. As the binary search tree becomes more unbalanced, the performance tasks began to get worse and worse, with its limit reaching in O(n) functionality where n is the number of the nodes in the binary search tree. The worse case scenario can be depicted as a data structure almost like a linked list which has to traverse from the root to the lowest level leaf node in the tree.

For the three depth first recursive methods: pre-order, post-order, and in-order, it is shown that the code traverses through every node. Given that there are a certain amount of nodes – n nodes, the maximum number of edges in the tree is (n-1) and hence, it operates in linear time.

For the breadth\_first method, the method contains a queue to store the nodes from the root to the left child to the right child. For each level of the tree that is created, the queue removed accordingly, and the queue became empty. This operates in linear time where n is the number of nodes, and n traverses all of the nodes found in the tree.