**Performance Analysis of AVL Tree methods**

**Class ­­\_\_BST\_Node:**

*\_\_init\_\_():*

The constructer of my *\_\_BST\_Node* class that creates a node with a given value has constant time worst-case performance, *O(1)* for the same reason as explained in the last writeup. Nothing changed about the node object in the new structure.

*update\_height():*

The *update\_height()* method in the node class performs in constant time *O(1)* in its worst-case, also for the same reason as explained previously

**Class Binary\_Search\_Tree:**

*\_\_init\_\_():*

The constructer of the *Binary\_Search\_tree* class performs in constant time in the worst case. The work is just to initialize a reference *self.\_\_root* to point to None, which is done in constant steps, making the whole constructer constant time *O(1).*

*\_\_rotate\_left(node):*

The private helper function *\_\_rotate\_left(node)* in my AVL Tree implementation runs in constant *O(1)* time in the worst case. This function returns the subtree rooted at *node* after rotating left. The work is fixed, involving the following steps: make a reference to *node’s* right child, reference the right child’s left child as *floater,* set *node* to be the right child’s new left child, set floater to be the old *node’s* new right child, and update *node’s* and the new sub-root’s (previous right child of *node*) height. Since all of the above work are either direct reference manipulations or constant *update\_height()* functions, the overall performance of the function is constant time *O(1).*

*\_\_rotate\_left(node):*

The private helper function *\_\_rotate\_right(node)* in my AVL Tree implementation runs in constant *O(1)* time in the worst case. The justification is the same as that for the *\_\_rotate\_left(node)* function as they’re actually the same operations along different directions.

*\_\_compute\_balance(node):*

My private helper function *\_\_compute\_balance(node)* returns one of “-2, -1, 0, 1, 2”, denoting the height of *node’s* right subtree (0 if None) minus the height of its left subtree (0 if None). This function runs in constant *O(1)* time in the worst case because all work is constant time given the *height* attribute already kept in my node class.

*\_\_balance(node):*

The *\_\_balance()* function returns the subtree rooted at *node* after balancing it. This function is a composite of *\_\_compute\_balance()* and my two *rotate* functions. Since all of them are worst-case constant time performance, my *\_\_balance()* function runs in constant time in the worst case.

*\_\_recursive\_insert(value, node):*

As have been explained in the previous writeup, the time of recursive calls in my *\_\_recursive\_insert()* function is proportional to the height of my tree, *O(h)*. Further, since in this structure the tree is guaranteed to be balanced, then *h* is a logarithmic function of the total number of nodes *n,* the number of recursive calls in my function is proportional to *log(n).*

In addition, inside each recursive call the work is all constant. In the previous writeup I have demonstrated that the previous work is all constant, involving creating a node or setting the current sub-root’s left or right child and update height; in the current structure, only one more step, which is *\_\_balance(),* is added, and this function is shown to be constant time. Therefore, the performance of work inside each recursive call is the same as that of the previous project, which is constant time.

Therefore, the total performance is a function of *log(n)* times a constant function, which is *O(log(n)).*

*insert\_element(value):*

The worst-case performance of this public insertion function is strictly the same as the recursive one, for the same reason as explained in the previous project. The worst-case performance is logarithmic time *O(log(n)).*

*\_\_recursive\_remove(value, node):*

For the same reason as in the *\_\_recursive\_insert()* function, the performance feature of recursive removal function is the same as the previous project because there’s only one constant function added to each recursive call, which has no impact on the overall level of performance. Further, since this additional function ensures that the tree would always be balanced, the total performance is guaranteed to be logarithmic time *O(log(n))*

*remove\_element(value):*

The worst-case performance of this public removal function is strictly the same as the recursive one, for the same reason as explained in the previous project. The worst-case performance is logarithmic time *O(log(n)).*

*in\_order; \_\_recursive\_inorder; pre\_order; \_\_recursive\_preorder; post\_order; \_\_recursive\_postorder; \_\_str\_\_():*

These seven functions are not changed, and the worst-case performance is also linear time *O(n)* as has been explained in the previous project.

*get\_height():*

This function is also unchanged, only involving returning an attribute already kept. The worst-case performance is constant time *O(1).*

*\_\_recursive\_to\_list():*

The worst-case performance of my *\_\_recursive\_to\_list()* function that returns a list representation of the in-order traversal of my AVL tree is linear time *O(n).* It adopts basically the same algorithm as the *\_\_recursive\_inorder,* resulting in *n* recursive calls and constant work of list concatenation inside each call. Therefore, the total performance is a linear function times a constant function, which is linear time *O(n).*

*to\_list():*

The performance of this public interface function, which just calls and returns the value returned by the recursive function, is exactly identical to the private recursive one, which is worst-case linear time *O(n).*

**Purpose and Efficacy of Test Cases.**

My test cases for this project is built on the assumption that my Binary\_Search\_Tree structure, without the balance function, has passed the 136 test cases designed for project 4 and proved to be reliable. Therefore, I wouldn’t need to test the functionality of insertion and removal from scratch. Rather, in this project I focused on some complex cases involving several levels of rotation to test that my balance function and insertion/removal combines well to give the right format. Further, the dimensions from which I test the result are modified to include the new *to\_list* function. Therefore, my general format for one test case involves five tests: in-order, pre-order, post-order, list representation, and height.

**Test insertion**

To test the functionality of balanced insertion, I have the following cases:

1. Inserting four elements in breath-first order, resulting in an already balanced tree: in this case the balanced insertion should result in exactly the same AVL tree as the tree produced by the structure in project 4.
2. Inserting four elements in decreasing orders, which requires a single right rotation to balance the tree: in this case the resulting AVL tree should be produced rightly as reflecting the rotation; I can test this by the pre-order and post-order traversal; and I expect to see the height is 3 instead of 4
3. Inserting four elements in increasing orders, which requires a single left rotation: I expect to see the right AVL tree produced, which can be tested by the pre-order and post-order traversal; also, the height should be 3 instead of 4.
4. Inserting four elements 5, 7, 6, 8, which requires a double rotation (first right, then left) to balance (5, 7, 6), and the height should be 3.
5. Inserting four elements, 5, 3, 4, 2, which requires a double rotation (first left, then right) to balance, and the height should be 3.
6. Inserting five elements in breath-first order, resulting in a balanced tree, and the structure should produce a tree exactly as expected.
7. Inserting five elements in increasing order, which requires two single left rotations to balance, and the height should be 3 instead of 5.
8. Inserting five elements in decreasing order, which requires two single right rotations to balance the tree and preserve the height to 3.
9. Inserting five elements 5, 8, 6, 10, 9, which requires two double rotations (first right, then left) to balance it and preserve the height to 3.
10. Inserting five elements 5, 3, 4, 1, 2, which requires two double rotations (first left, then right) to balance it.
11. Inserting five elements 6, 1, 2, 4, 5, which requires first a left-right double rotation and then a right-left double rotation to balance it and preserve the height.

Through the above cases, I have tested the functionality of my balance function in nearly all possible cases of imbalance, and if all tests are passed, I have very good reasons to believe my structure works well.

**Test removal**

For removal I designed 5 complex cases, two involving one level of imbalance after removal, two involving two levels of imbalance, and the last one involving tree levels of imbalance after removal.

1. For the one-level imbalance case, I constructed an AVL tree whose right and left sub-trees are two minimum AVL trees of height 3 and remove the side node which would result in one of that minimum AVL tree being imbalanced. I replicate this case in mirror directions to produce two cases involving different rotations.
2. For the two-level imbalance case, I constructed a minimum AVL tree of height 5, whose left sub-tree is a minimum AVL tree of height 4 and right sub-tree is a minimum AVL tree of height 3. Then I remove the side node of the height-3 AVL tree, which would result in two nodes being imbalanced. I replicate this case in the mirror direction to produce two test cases involving different rotations.
3. For the three-level imbalance case, I constructed a minimum AVL tree of height 7, whose left sub-tree is a minimum height-5 AVL tree and right sub-tree is a minimum height-6 AVL tree. Then I remove the node positioned at the same place in the height-5 AVL tree as the removed node in my previous cases, resulting in tree levels of imbalance. For this case, I only produced one test to make sure that my function works well in this extremely complex case.

My test logic of the removal method resembles that of mathematical induction: the first case is about height-4 AVL tree, which is a sub-tree in my next case of height-5 AVL tree, which is further a sub-tree in my third height-7 AVL tree. If all my tests are passed, then I would have reasons to believes my structure would work well from here to infinity.

**Procedure and Performance of AVL tree Sorting**

The procedure of using an AVL tree structure to sort an array of *n* elements has only one step: inserting the *n* elements into an AVL tree. Then the elements are already sorted because of the nature of a binary search tree.

The performance of this sorting approach is *O(nlog(n)).* The reasons are as follows:

To insert *n* element into an AVL tree, according to the performance feature of my insertion method, the total running time is approximately given by (ignoring the constant-time step involved in insertion):

log(1)(insert the first) + log(2)(insert the second) + … + log(n)(insert the nth)

Since: , whose dominant parts are xlog(x) – x,

the result of this expression can be approximated by nlog(n) – n, which, ignoring the smaller part, is nlog(n).

Therefore, the total performance of inserting *n* elements into an AVL tree is worst-case linear times logarithmic time *O(nlog(n)),* which is also the performance of the sorting algorithm using an AVL tree.