CSCI 241 Data Structures
Project 5: AVL Trees

Be certain to retain a copy of project 4 as submitted. Make a copy of project 4 to use as the starting point for project 5.

In this project, you will extend your BST implementation in Binary_Search_Tree.py to guarantee $O(\log n)$ insertions and removals. The changes required to support this are minimal. Provide a private method called `__balance(t)` that takes a node `t` as a parameter and treats it as the root of subtree. If the subtree rooted at `t` is unbalanced, rotate as necessary to balance it and return the new root of the now balanced subtree. Because this balance operation will be invoked on the return path from recursive insertion and removal, you can be sure that everything below `t` is already balanced. This means that checking to see if the tree rooted at `t` is balanced (and rotating it if it is not) is a constant time operation.

If you followed the algorithms presented on the lecture slides and in class, all you have to do after writing the `__balance(t)` function is change the last line in your private recursive insert and remove methods from `return t` to `return self.__balance(t)` and reconsider when you compute height. Note that it is not necessary to check the balance of the new node created and returned in the base case, because it will always be balanced already. Be sure that when you return a subtree it is balanced and the height attribute of every node at or below `t` is correct (but only reevaluate the heights of nodes whose subtrees could potentially have changed—that is, every node on the insertion/removal path and every node actively involved in a rotation).

Once you have balanced insertions and removals implemented, **add another public/private method pair for recursively constructing a Python list of the values** in the tree. This method should work just like the in-order traversal methods, but it should return a python list, not a string. Your public method should be called `to_list`; you are free to name your private recursive method whatever you like.

Finally, complete the implementation of the provided Fraction class by implementing the three comparison operators. The main section of this program should create a Python list of fraction objects, then insert them one at a time into an initially empty AVL tree, then get the in-order Python list representation using the new `to_list` method of `Binary_Search_Tree`, showing that the returned list is in sorted order.

SUBMISSION EXPECTATIONS

`Binary_Search_Tree.py` This should be your implementation of an AVL tree. You are free to add additional private support methods (in fact, this is necessary), but do not change the public interface to this class other than introducing the new `to_list` method.

`BST_Test.py` Your unit tests for your implementation. No skeleton file is provided for this component. For testing, notice that the three traversals (in-order, post-order, and pre-order) uniquely identify a binary search tree. No two unequal trees share all three traversal orderings. Ensure that your traversals work correctly and use them to test the insertion and removal methods.

`Fraction.py` The provided Fraction class with the comparison methods implemented and with the main section sorting a Python list of fraction objects.

`Writeup.pdf` A prose writeup explaining the purpose and efficacy of your test cases and an analysis of the worst-case asymptotic performance of every method in your implementation. Your writeup must also include a detailed performance analysis of this sorting approach. Be certain to account for all steps in the performance analyses.