Balanced Binary Tree and Generators

It is possible over time for a binary tree to become unbalanced as nodes are inserted and deleted from the tree. Several algorithms exist for balancing trees, including such classics as AVL-tree, red-black tree and others. Many algorithms immediately balance a tree during insertion and deletion, or rebalance if one subtree differs in height by more than 1 from its sibling subtree. Some more modern algorithms amortize the cost of balancing a tree by checking only periodically and rebalancing if needed, instead of after every insert.

You must add the following function to your BinarySearchTree class of the previous project:

- inorder : performs an in-order traversal of the tree. Returns either a list or a generator. Can be recursive or iterative (hint: recursive is easier).
- rebalance_tree : use the method described below to re-balance the current tree
- So that this re-balance method always words, you will have to modify the add() method so that it **WILL NOT** add duplicate elements to the tree.

If you need extra "helper" methods to properly implement the two extra required methods, that is just fine.

No "driver" (i.e. main.py) is required for this project. However, you are encouraged to build one to help you test your new functionality.

re-balance

For this project, you will modify the BST you created in a previous project to include periodic rebalancing or rebalance when some imbalance condition is true: such as one side is x amount higher than the other side. The rebalancing algorithm you will implement is as follows:

1. do an inorder traversal of the tree and write the node values out to a list. If you wish you can use a generator to easily create this list.
2. take the middle value as root
3. split the list in left and right halves, excluding the middle value
4. recursively rebuild the tree, using steps 2 and 3 until done.

When finished, the tree will be balanced.

Test Cases

Test inorder

create an empty tree

add 125 random integers

call the inorder() method

assert that the result of inorder() is in ascending order

Test Tree Height

create an empty tree

assert that the height of the empty tree is -1

add 510 integers, all increasing by one (range(511)) (all nodes should be “right-chidren” of their parent)

assert that the height of the tree is 510

rebalance the tree

assert that the height is 8

Test Tree Rebalance

create an empty tree

add 126 random, but sorted integers

rebalance

assert the tree has a correct height

Test Code Quality

PyLint will be run on binarysearchtree.py. It must have at least an 8.5 PyLint score.

Grading (100 points)

• Passes Inorder Test	20
• Passed Test Tree Height	30
• Passed Tree Rebalance	30
• passed Coding Standards	20

Turn in to Canvas:

- binarysearchtree.py