CSCA48 Exercise 8

Due: March 17, 2017. 5:00pm

This week, we'll be playing around with trees. In particular, binary (but not necessarily binary search) trees. For all of these questions you can start with the `BTNode` class provided below (a text version will be uploaded so you don't need to copy and paste).

```
class BTNode(object):
    """A node in a binary tree."""

    def __init__(self, value, left=None, right=None):
        """(BTNode, int, BTNode, BTNode) -> NoneType
        Initialize this node to store value and have children left and right,
        as well as d value of 0.
        """
        self.value = value
        self.left = left
        self.right = right
        self.d = 0  # just a variable that we may want to set later

    def __str__(self):
        return self._str_helper("")

    def _str_helper(self,indentation = ""):
        """(BTNode, str) -> str
        Return a "sideways" representation of the subtree rooted at this node,
        with right subtrees above parents above left subtrees and each node on
        its own line, preceded by as many TAB characters as the node's depth.
        """
        ret = ""

        if(self.right != None):
            ret += self.right._str_helper(indentation + "\t") + "\n"
        ret += indentation + str(self.value) + "\n"
        if(self.left != None):
            ret += self.left._str_helper(indentation +"\t") + "\n"
        return ret

if(__name__ == "__main__"):
    #just a simple tree to practice on
    my_tree = BTNode(10,BTNode(3,BTNode(5),BTNode(2)),BTNode(7,BTNode(4,BTNode(9)),BTNode(6)))
    print(my_tree)
```

For all of these questions, you must not change the provided code, but you may add helper methods if you feel that they are necessary. Each task is do-able in about a dozen or so lines (or less). If your code is much longer, there's a good chance you're missing an obvious simpler (probably recursive) way to solve the problem. Your code must be efficient, and should not traverse the tree more than once. In particular, you can't traverse the tree once to set the depth values and then again to use those values.

We highly recommend drawing some example trees and working through the idea before you start coding, it will likely save you a great deal of time and aggravation.

set_depth

Add a method called `set_depth` which sets the `.d` parameter of all nodes in the tree rooted at a particular node, starting at the provided value. For example, a call of `my_node.set_depth(0)` would set the `.d` parameter of `my_node` to 0, all of its children's `.d` parameters to 1, etc. [1]

leaves_and_internals

Add a method called `leaves_and_internals`, which returns a tuple with two sets. The first is the set of all values stored in the leaves of the tree rooted at this node, and the second is the set of all values stored in the internal nodes of the tree rooted at this node.

Note that for our purposes, leaf nodes are all nodes which have no children, and internal nodes are all nodes that are neither the root, nor a leaf.

sum_to_deepest

Add a method which returns the sum of all values on the path from this node to the deepest leaf node (i.e., the leaf node with the longest path to the root). If there are multiple leaves at the deepest level, return the maximum sum of those paths.

What to Submit

Submit your `BTNode` class including the methods you added in a file called `ex8.py`

[1] Note that the `d` variable doesn't necessarily correlate to the actual depth in the whole tree, only in the sub-tree on which it was called. In particular, you can't assume that they're set to anything sensible for the other functions