Computer Science 112 - Fundamentals of Programming II
Lab Project 7
Due on Github 11:59pm Monday 8 November

In this project, you will add some features to your Binary Search Tree (**`BST`**) class, and then implement a new class for general binary trees. To get started I just copied my **bst.py** from last week and added some test cases and methods.

Part I: Tree traversals

Implement and test the preorder, inorder, and postorder traversal methods we discussed in class. Your **`preorder`**, **`inorder`**, and **`postorder`** methods should return a list of the data (node contents) produced by visiting the tree in that order. For example:

```
bst = BST()

bst.add(2)
bst.add(3)
bst.add(1)

print(bst.inorder())

[1, 2, 3]
```

Part II: Rebalance[1] your tree

Your next task is to write a **`balanced()`** method for your **`BST`** class. This method will take a tree (balanced or unbalanced) and return a balanced version of it. As we discussed, it's easy to create an unbalanced tree by adding nodes in descending order (3, 2, 1). Slides #33 and 34 of slide deck 25 show the algorithm / pseudocode for rebalancing. Since we're not mutating the original tree as this algorithm does, we don't have to clear the tree; we just create a new tree and add to it instead.

Part III: Breadth-First Search

As we've seen, preorder, inorder, and postorder traversals are all *depth-first* strategies, because they visit a node's descendants before visiting its sibling(s). For this exercise you will write a method **`bfs`** that will implement *breadth-first* search, in which you visit the siblings first. The algorithm is described in my supplementary lecture slides on Tree Search.

Now, it makes no sense to implement **`bfs`** as method of your BST class, since BSTs are already designed for efficient searching. So to get started on this exercise, copy/paste your **bst.py** file into a new file **bt.py** and modify **bt.py** to provide a class **`BT`** for general binary trees. This will require re-

[1]Technically, this should be called *balancing* the tree, not *rebalancing*, since we have to reason to assume the tree was balanced already. That's why I chose the name **`balanced`** for the method you'll write. For a similar non-logical use of word prefixes in ordinary language, see *re-entry vehicle, irregardless,* etc.

thinking your **`add`** method somewhat, so that nodes are added simply in the order you enter them, resulting in a balanced tree. Keeping the tree balanced is only tricky when a node has both left and right children. In that case, I just used **`random.random()`** to "flip a coin" to decide whether to put the new node in the left or right branch. Here's some test results:

```
bt = BT()

bt.add(1)
bt.add(2)
bt.add(3)
bt.add(4)
bt.add(5)
bt.add(6)
bt.add(7)

print(bt)
print(bt.height())
```

```
(1 (2 (4 (7 _ _) _) (6 _ _)) (3 (5 _ _) _))
3
```

Unlike your previous methods returning a list, this **`BT.bfs`** method should return simply True or False, depending on whether the target datum was found in the BST. To aide in debugging, use a verbosity flag that defaults to False and if True reports (prints) the datum of each node as it is visited:

```
def bfs(self, target, verbose=False):
```

You will probably find it useful to use (and perhaps add to) your **`Queue`** class from a previous lab, in which case, please also submit the files necessary to support that class (**queue.py**, **linkedlist.py**).

Part IV: Depth-Limited BFS

As discussed in the Tree Search slides, BFS suffers from the potential for *exponential blow-up*, because the number of nodes is $O(2^n)$for a recursion depth of n. So to finish up this lab, add a depth limit to your **`bfs`** method. If this limit is **`None`**, your **`bfs`** method should search the entire tree; otherwise, **`bfs`** should stop once the current depth goes beyond the limit.

Because I had trouble figuring out an efficient way to track the depth of the tree during the actual search, I went back and modified my **`Node`** class and its **`add`** method to encode each node's depth as it was added.

To get a better sense of the cost of $O(2^n)$ algorithms, we'll finish this lab by doing some timing tests on our **`bfs`** method. At the top of your bt.py code, add a line **`from time import time`**. Now you can use the **`time`** function to time various pieces of code:

```
start = time()
# do something here
print(int(time() - start), ' seconds')
```

Before running this timing code, you can use a **for k in range(n)** loop to enter a large sequence of numbers **k** into a binary tree **bt**. Then the ***# do something here*** part can simply be a call to **bt**. **bfs(n)**, thereby guaranteeing you search the entire tree and don't find the number.

Here's my entire output for this part, showing timings for various searches. I found that 500,000 items in the tree took long enough to search to be interesting, but not so long that I ran out of patience:

```
Part IV ------------------------------
Building tree with 500000 nodes; could take a few seconds ...
Tree has height 21

Full BFS for target 100000
True
0  second(s)

Full BFS for missing datum...
False
3  second(s)

BFS for missing datum, depth limited to 15
Maxed out at depth 15; quitting
False
0  second(s)
```