Computer Science 112 - Fundamentals of Programming II
Lab Project 7
Due on Github 11:59pm Monday 4 November

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: Heap Heap Hooray!

Create a file **heap.py** defining a **`Heap`** class. As with our other container classes, you'll want an **`add`** method and a **`__contains__`** method as the minimal API for this class, plus a **`__str__`** method to aid in debugging. Feel free to copy/paste from the lecture slides (slide deck 28) to get your heap working. To keep things simple, your **`__str__`** method can just return a string representation of the heap's array. (I tried having **`__str__`** return a string showing the hierarchy like a BT, but I didn't get too far and gave up: maybe an extra-credit opportunity!)

Part IV: Priority Queue Redux

As we discussed, the efficient way to implement a priority queue is with a heap. To get started on your new priority queue, add a **pop** method to your **Heap** class, using the code from the lecture notes.

1Technically, 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.

Next we'll want a new **`PriorityQueue`** class to compare with the one we built in Lab 5 a few weeks ago. What I did was to copy the **priorityqueue.py** file from that lab into this lab and rename it to **priorityqueue2.py**, and replace the **`LinkedList`** code with code from my new **`Heap`** class. Doing that should involve simply replacing your complicated **`PriorityQueue`** code with one-line calls to the **`Heap`** methods; for example:

```
    def push(self, datum):

        self.heap.add(datum)
```

After running some simple tests as usual, you're ready to have your new and old priority queue classes go head-to-head in an efficiency contest. (Don't forget to copy your **priorityqueue.py** and its supporting **linkedlist.py** to this lab folder to enable the comparison.) Create a new file **pqtest.py** to run this comparison. By making good use of the import statement, you can refer to your two queue implementations elegantly in your main code:

```
from priorityqueue import PriorityQueue as PQ1
from priorityqueue2 import PriorityQueue as PQ2
```

As you did last week, code up a little timing test to show the difference in cost associated with inserting an element into each of these queue implementations, making sure to put a helpful **`print`** statement before any code that you know is going to take a nontrivial amount of time to run.