**Computer Science 112 - Fundamentals of Programming II**
**Lab Project 6**
**Due on Github 11:59pm Monday 1 November**

In this project, you will experiment with Binary Search Trees (BSTs) by coding them from scratch. You will only need to submit one file **bst.py** when you're done.

## **Part I:** Write **bst.py**

As usual, start by creating some test cases for the basic API (functionality) of a class **`BST`**. As we discussed, the basic functionality for a BST is simply the **`add()`** and **`__contains__()`** methods. As usual, you'll also want a **`__str__()`** method to help with debugging. This method will be recursive, but its output doesn't have to be as fancy as the one in the slides: you can instead follow the traditional convention of using parentheses to indicate structure; e.g., **`(3 (2 _ _) (4 _ _))`** represents a tree with 3 as the root and children 2 and 4. As with our **`LinkedList`** class from last time, you'll find it useful to have **`Node`** class to support your **`BST`** class. This **`Node`** class will be the one where the actual recursion takes place, in its own set of recursive methods.

Before you even write the bodies of these three methods, think of some good test cases. For trees I find it very useful to sketch some trees out on scrap paper (let me know if you need some!) before coding them up as test cases.

## **Part II:** Add some functionality

No good deed goes unpunished: as soon as you implement the basic functionality of a BST, your customer wants some additional features, because reasons.

One thing you can ask about any tree (BST or otherwise) is its *height*; i.e., the number of steps from the root to the deepest leaf. Write a method **`height()`** that does this for your BST. As before, the recursive part will be done by your **`Node`** class.

Next, even though a tree is not a sequential / linear structure like a list, it is possible to query its *length*, defined as the total number of nodes it contains. In other words, if **`b`** is a BST, then **`len(b)`** should return the number of nodes in it.

### **Part III:** Check your balance

As with sorting, the specific data that we enter into our BST can affect the performance of its core functionality. For BSTs, the idea is to get O(log n) performance for our **`__contains__()`** method by exploiting the branching of the tree. This appears to give us the best of both words: we get the O(log n) efficiency of a sorted array without the O(n) cost of having to resize an array if we exceed its size when adding a new item.

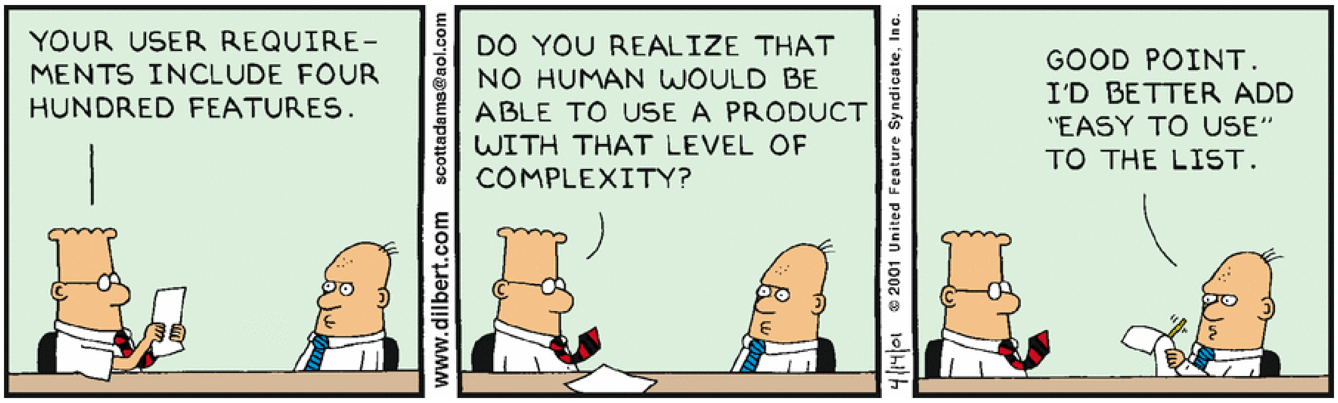
As we've seen throughout the course, however, the familiar TANSTAAFL[1] principle still applies: if our tree isn't *balanced* (branching evenly on both sides like a Christmas tree), the cost of lookup can be

[1]***T**here **A**in't **N**o **S**uch **T**hing **A**s **A** **F**ree **L**unch*

O(n). This can happen when (for whatever reason) the data are entered in strict order. We can see this with our original 3, 2, 4 example: If we enter the data in the order 3, 2, 4, we'll get a nice balanced tree with 3 at the root and 2 and 4 at the leaves. But if we enter the data in the order 2, 3, 4 we'll get 2 as the root, with 3 as its right child, and 4 as the right child of 3. Then instead of taking just one step to find 4, it will take two steps. Extend this example with many more items, and you'll see that such "skinny" trees are a kind of linked list and therefore have the dreaded O(n) lookup cost.

To finish up this week's lab, then, add a method **`isBalanced()`** to your BST class. Thankfully this method doesn't have to be recursive and will require only one line of code to implement, once you've completed the **`height()`** and **`__len__()`** methods. Specifically, we will say that a BST with height *h* and length *l* is balanced when $\boldsymbol{h < \mathrm{ceil}(\log_2(l))}$, where ceil is a rounding function that always rounds up. (For ceil and $\log_2$ you can just import **`math`** and use **`math.log2`** and **`math.ceil`**). Test your **`isBalanced()`** method on a balanced tree and a skinny tree to make sure you've coded it up correctly.

Next week we'll start by adding a mutator method **`balance()`** to our BST class, but for now, you're all done!


YOUR USER REQUIRE-MENTS INCLUDE FOUR HUNDRED FEATURES.
DO YOU REALIZE THAT NO HUMAN WOULD BE ABLE TO USE A PRODUCT WITH THAT LEVEL OF COMPLEXITY?
GOOD POINT. I'D BETTER ADD "EASY TO USE" TO THE LIST.
scottadams@aol.com www.dilbert.com
© 2001 United Feature Syndicate, Inc.
4/14/01