CS 120: Intro to Computer Programming II

In-Class Activity - 06 Trees - Day 3

Activity 1 - Turn in this one

As always, keep everybody in the group engaged! What if each person only writes one line of the function below?

A recursive function over a tree generally does something to the node itself, and also recurses into each of the node's children. Write a recursive function which prints the value stored in every node; for now, we won't care about what order they are printed.

Your function should take only a single parameter, which is the root of the tree (or subtree), and it should handle empty trees (by printing nothing at all).

Solution:

```
def print_tree(root):
    if root is None:
        return

    print_tree(root.left)
    print(root.val)
    print_tree(root.right)
```

Activity 2 - Turn in this one

Not all trees are BSTs! Below, I've given you the code to build two different binary trees - one of them is a BST, and one of them is not. But they have **exactly the same shape!**

Talk amongst the group - without drawing anything at first - can you tell which one is a BST, and which one is not? How might you tell? (If you don't figure it out from the code, it's OK to draw it later, but let's try to do it without drawing, first.)

Then, run both of these trees through the print-all function you wrote in the previous activity. Double-check that the function is working the way you expect.

```
tree1 = TreeNode(74)
tree1.left  = TreeNode(4)
tree1.right = TreeNode(80)
tree1.left .left  = TreeNode(0)
tree1.left .right = TreeNode(17)
tree1.right.left  = TreeNode(77)
tree1.right.right = TreeNode(96)

tree2 = TreeNode(69)
tree2.left  = TreeNode(38)
tree2.right = TreeNode(63)
tree2.left .left  = TreeNode(53)
tree2.left .right = TreeNode(68)
tree2.right.left  = TreeNode(88)
tree2.right.right = TreeNode(46)
```

Solution: The second one is **not** a BST, it has lots of search property violations!

- The right child of 69 is 63
- The left child of 38 is 53
- The left child of 68 is 88. (This is also in the left subtree of 69, which is also a violation.)
- The right child of 68 is 46

(Any one of these is proof that it's not a BST, even without all the others.)

(activity continues on next page)

Activity 3 - Turn in this one

Write a recursive function which calculates the total height of the tree. To do this, you will need to call into both children, and get their height; then, calculate the height of the current subtree as a little more than either of those. For example, suppose that one of your child subtrees has a height of 3, and the other has a height of 7; the height that you should return is 8.

Remember, in my class (other teachers may be different), we define the "height of a tree as the number of **links,** going from the root to the deepest.

Special Requirement:
For this function, I'm going to give you a hint/requirement about how to write this function. Your code must check for each of the following cases; some of them will require recursion, and others will be base cases:

- An empty tree
- A leaf node
- A node that has a left child, but not a right
- A node that has a right child, but not a left
- A node that has two children

(If you look carefully, you will realize that if you write it this way, you will **never** need to hit the empty-tree case, unless the user gives you a tree which is truly empty, right from the start. This will seem like a good thing - but in the next Activity, I'll show you something better!)

Solution:

```
def tree_height1(root):
    if root is None:
        return 0         # NOTE: we'll change this in the next Activity
    if root.left is None and root.right is None:
        return 0
    if root.left is None:
        return 1 + tree_height1(root.right)
    if root.right is None:
        return 1 + tree_height1(root.left)

    return 1 + max(tree_height1(root.left), tree_height1(root.right))
```

Activity 4 - Optional

OPTIONAL. Complete this if you have time, and turn it in. If you don't have time, you may report to your TA that you ran out of time.

Now, re-write the previous function, but this time, with a very strange requirement: now, the **only** special case you can check for is an empty tree. In **all** other cases, you are required to recurse into both children.

Now, consider the case of a leaf node. You know that your function must return 0, if called on a leaf node. But you also know that, in this weird, limited version of the depth function, you are also required to recurse when you hit a leaf node - you can't check for it! Using this information, **figure out what value you ought to return, in the empty-tree case.**

Solution:

```
def tree_height2(root):
    if root is None:
        return -1        # this is strange!  But it makes the leaf case work well.
    return 1 + max(tree_height2(root.left), tree_height2(root.right))
```

Challenge Activity - Do not turn in this one

In the video that you watched before this class - and in the slides - we've already given you the `bst_insert()` function. Go get a copy, and copy it into your editor.

Now, write a loop which will insert 1000 randomly-chosen values into a BST. Remember, we can generate a random number by importing the `random` library, and then running something like this:

```
rand_value = random.randint(-5*1000,20*1000)
```

Remember that your function uses the `x = change(x)` form; how do you insert one value into a tree?

Run your code, and see how long it takes to insert 1000 values (it shouldn't take longer than a couple of seconds; if it does, do some debugging).

Finally, run your tree-height function on the tree that you've built; how high is it? Are you surprised by your answer? Get out a calculator, and figure out (approximately) what $\lg 1000$ is. (Remember, $\lg n == \log_2 n$.)

Solution:

```
root = None
for i in range(1000):
    root = bst_insert(root, random.randint(-5*1000, 20*1000))
```

Since $\lg 1000 \approx 10$, we would guess that the actual height of the tree will be around 12-15, rarely more than 20.