CS 120: Intro to Computer Programming II

In-Class Activity - 06 Trees - Day 5

In this exercise, we'll be building BSTs; it will be useful if you bring in the `TreeNode` class and our pretty-print function.

1 Activity 1 - Turn in this one

Working together as a group (I don't want one person to dominate this question!), and **without** reviewing your notes or the slides or videos, see if the group can re-create the `bst_insert()` function.

Remember the following rules:

- The function is in the `x=change(x)` style. This means that it will take the old tree (which might be empty) as one of its parameters, and return the new tree. **Make sure that every return statement in the function returns something,** and also that you don't fail to have a return at the bottom of the function.
- The function must take a **value** parameter, which is the value you are adding to the tree.
- For simplicity, you may assume that no one will ever attempt to insert a duplicate value into the tree.

Once the group has come up with your solution, go look at the slide deck to check your work. Did you overlook anything? If so, discuss what the impact of your missing (or incorrect) piece would have been on how the function would run.

Solution:

```
def bst_insert(root, val):
    if root is None:
        return TreeNode(val)

    if val < root.val:
        root.left  = bst_insert(root.left,  val)
    else:
        root.right = bst_insert(root.right, val)
    return root
```

2 Activity 2 - Turn in this one

Not all people like the `x=change(x)` style. Some people prefer a different style, where the `insert()` function returns nothing. But to make this work, it means that the caller must check, every time, whether the tree is empty; if so, the **caller** (instead of the `insert()` function) must create a single `TreeNode`.

Write a new function, `bst_insert_2()`, which doesn't use the `x=change(x)` style. Instead, follow these rules:

- This function takes the same parameters as `bst_insert()`, except that the `root` parameter must **never be** `None`**.**

  Add an `assert` statement, at the start of the function, to double-check that this is always true:

  ```
  assert root is not None
  ```

- This function must **never** return anything.
- Since the **caller** is responsible for handling all empty-tree cases, you **must not** recurse into an empty sub-tree.

Solution:

```
def bst_insert2(root, val):
    assert root is not None:

    if val < root.val:
        if root.left is None:
            root.left = TreeNode(val)
        else:
            bst_insert2(root.left,  val)
    else:
        if root.right is None:
            root.right = TreeNode(val)
        else:
            bst_insert2(root.right, val)
```

(activity continues on next page)

Activity 3 - 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.

Students often have trouble understanding why the `x=change(x)` returns the "new" root node, even if that node didn't change. To help you understand this, I'm going to have you draw some "before and after pictures."

For each item below, I'll give you a set of numbers to insert into a BST. Draw the entire BST; you don't have to show all the steps. This is the **before** picture.

Then, I will give you one more value. Add it to your BST, in the proper place again. This is the **after** picture.

As you work on these examples, discuss with your group how what you are seeing, when you insert into a relatively small tree, helps you understand what it takes to insert a new value into a larger, more complex tree. Look for the **recursive** nature of a tree: each subtree is itself a small tree.

Before	**After**
`[]`	50
`[50]`	10
`[50]`	80
`[]`	75
`[80]`	75
`[50,10,80]`	75
`[75]`	77
`[80,75]`	77
`[50,10,80,75]`	77