CS 120: Intro to Computer Programming II

In-Class Activity - 06 Trees - Day 6

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

1 Activity 1 - Turn in this one

For the first part of this exercise, let's update the pretty-print function from our previous activity. Modify it so that it will print an in-order traversal, instead of a pre-order traversal. But leave the indentation intact; the root should still be the furthest left, and for each node, its children should be two spaces futher right.

Ideally, I'd like your function to have the 'size' feature that we added - but if that would take a bunch of time to implement, it's not strictly required for today's activities.

Solution:

```
def pretty_print_in_order(root, indent=""):
    if root is None:
        return
    pretty_print_in_order(root.left,  indent+"  ")
    print(f"{indent}{val}    size: {tree_size(root)}")
    pretty_print_in_order(root.right, indent+"  ")

def tree_size(root):
    if root is None:
        return 0
    return 1 + tree_size(root.left) + tree_size(root.right)
```

2 Activity 2 - Turn in this one

Now, let's build a function which will build a BST from a bunch of integers. This first version will simply iterate through an array, and insert each integer from the array into the BST using a standard BST `insert()` function.

Before you test this, the group should discuss what you expect to see: if the input data is random, should you see a balanced tree or an imbalanced one? How badly imbalanced do you expect? How common will extremely "unlucky" trees (that is, random trees which have **lots** of imbalance) be? **Include a description of what your group predicted as part of what you turn in today.**

Now, write a little code which will generate 30 random integers, and put them into an array. Then pass the array to your function, which will build a BST out of them. Then pretty-print the result.

Run this code several times. Were the group's predictions correct? Don't worry about writing down the actual trees that got generated - instead, turn in a **discussion** of what happened (along with the code your group wrote, of course).

Solution:

```
def build_bst_from_vals(data);
    root = None
    for v in vals:
        root = bst_insert(root,v)
    return root

def gen_30_random():
    retval = []
    for _ in range(30):
        retval.append( random.randint(-50,100) )
    return retval

vals = gen_30_random()
tree = build_bst_from_vals(vals)
pretty_print_in_order(tree)
```

3 Activity 3 - Turn in this one

Can you force your function to build a terrible tree? Make a tiny modification to the function that builds the tree - the one that iterates through the array - so that it will always build a tree with a linked-list shape.

Solution:

```
def build_terrible_bst_from_vals(data):
    root = None
    for v in sorted(vals):
        root = bst_insert(root,v)
    return root
```

(activity continues on next page)

4 Activity 4 - Turn in this one

While it's important to know what you can do wrong, it's more important to know how to do the right thing! This time, update the tree-builder function to always build a **perfectly balanced** tree.

To do this, rewrite the function so that it's recursive. It will still take an array as the parameter, and it will return a tree - but instead of using a loop, it will build the tree recursively.

To **build** a tree recursively, your function must create a new node for the root, and then use recursion to build the left and right trees. In order to make tree as close as possible to balanced, we want the left and right subtrees to be as close as we can to the exact same size.

So here's the question for the group: if we want the left and right subtrees to be the same size, what value should we choose for our root node? Talk about it as a group (make space for the people who might be more confused!!!), and when the group comes up with an answer, turn it into code.

Solution:

```
def build_perfect_bst_from_vals(vals):
    if len(vals) == 0:
        return None
    vals = sorted(vals)

    mid_indx = len(vals)//2

    root = TreeNode(vals[mid_indx])
    root.left  = build_perfect_bst_from_vals(vals[:mid_indx])
    root.right = build_perfect_bst_from_vals(vals[ mid_indx+1])

    return root
```

Challenge Activity - Do not turn in this one

In this activity, we've had two different ways of inserting values into a BST. One of them simply read the array in order, and inserted the values as they came; it simply **hoped** that the result would be close to balanced (and normally, that works well). The second one **sorts** the data so that it can be sure that it will always build a great tree.

Can you combine these?

Write a function which will, given a set of values in an array, return a new array with the same values - but **rearranged** so that, if we add them into a BST, one at a time, it will always be balanced. Once you've written this, try it out; your test code will:

- First, generate a random sequence of integers
- Then, re-arrange the integers into this special order
- Then, insert them into the BST, one at a time
- Then, pretty-print it all, so that you can double-check that it works

Remember: The first value in the re-arranged array will become the root node. So choose it carefully!

Solution:

```
def rearrange_vals_for_perfection(vals):
    if len(vals) == 0:
        return []
    vals = sorted(vals)

    mid_indx = len(vals)//2

    retval  = [ vals[mid_indx] ]
    retval += rearrange_vals_for_perfection(vals[:mid_indx   ])
    retval += rearrange_vals_for_perfection(vals[ mid_indx+1:])

    return retval
```