CS417 Lab #18

Skills

This lab builds two skills:

- Recursive functions
- Binary search trees

Getting Started

Begin the lab by creating a folder for your files. Then, download the following file into that folder:

- `tree.py`

Starter Code

In this lab, you will work with a binary tree. There is a partial implementation in the file `tree.py`. You should edit this file.

The file `tree.py` defines a class `Tree`, and has these methods already implemented:

- `__init__` : constructor
- `insert(value)` : insert value into tree, using BST insertion.
- `__str__()` : string version of tree.
- `__repr__()` : programmer-friendly string version. USE IT! Invoked as `repr(tree)`, or `tree.__repr__()` .

Debugging Aids

The file includes a class `Node`, which has a very useful `__repr__()` method. If you are in the middle of a method, and `node` refers to some tree node, just `print(repr(node))` and you'll see some useful info, including the node's `id`.

Also, the `Node` class has a convenience `is_leaf()` method, which you'll need in the last question.

Also, the tree's `__repr__` method prints the tree "sideways", listing each node indented according to its depth. If you mentally rotate this result clockwise, you can picture the relative positions of the nodes.

Exercises

1. Implement the `is_empty()` method, which returns `True`/`False` if the root is/isn't `None` .

---

2. Implement the `subtree_size(node)` method, which visits every node recursively, and counts the nodes. Here are the cases you should handle:

   - Base case (`node == None`): return zero size.

   - Recursive case:

   a. call `self.subtree_size(node.left)`, and save the result.
   b. call `self.subtree_size(node.right)`, and save that result too.
   c. compute 1 + size of left subtree + size of right subtree. Return that result.

---

3. Implement the `subtree_height(node)` method, which returns the height of the subtree whose root is `node`. You **must** use recursion. Here are the cases you should handle:

   - Base case (`node == None`): height is zero.

   - Recursive case: this is very similar to the previous question:

   a. call `self.subtree_height(node.left)`, and save the result.
   b. call `self.subtree_height(node.right)`, and save that result too.
   c. compute 1 + max(size of left subtree, size of right subtree). Return the result.

---

4. Implement the `righmost_data(node)` method, which returns the data in the node which is on right-most in the tree (if tree is empty, return `None`).

   You don't need recursion; just start at the root, and loop: repeat `node = node.right`, until you can't go right. It's just like walking down a linked list, to reach the tail node.

---

5. Implement `subtree_sum(node)`, which sums all the values in the subtree whose root is the given `node`. You must do this recursively. Here are the cases you should handle:

   - Base case (`node == None`): return 0

   - Recursive case: `node`'s value, plus sums of both subtrees. Follow the model for questions 2 and 3.

---

6. Implement the `subtree_contains(value, node)` method, which returns `True`/`False` if the value is/isn't in the subtree rooted at `node`.

Since the tree is not sorted in any way, you have to search every node. Use recursion:

a. Base case (`node == None`): return `False`

b. Recursive case:

- if `value == node.data`, return `True`.
- else, if `node.left.contains(value)`, it's in the left subtree: return `True`.
- else, check the right subtree, in a similar way.
- else, you now know that it's not in the `node`, not in the `left` subtree, and not in the `right` subtree. Return `False`.

---

7. (10 % bonus) Implement `subtree_deepest(node, depth)`, which finds the value at the lowest leaf in the subtree rooted at `node`.

   Note the second argument, `depth`. Each time you call yourself, pass in `depth+1` to track the subtree's depth.

   Return value: the method should return a two-element list. Its first value is the data in the deepest leaf, and the second is the depth of the deepest leaf.

   Here are the cases you should handle:

   Base case (`node` is a leaf: check by calling `node.is_leaf()`): return a two-element list, with [`node`'s data, and `depth`].

   Recursive case: get deepest pairs in both subtrees (call yourself, passing `depth+1`), and return the deeper pair.

Submitting your work

At the end of the lab session, turn in any work you have completed. You can re-submit your work until midnight of the lab day, with no late penalty. Go to `mycourses.unh.edu`, find CS417, and the lab. Then click the "Submit" button and upload `tree.py` .