CS417 Lab 19

Getting Started

This lab builds skills with binary trees.

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

- `tree2.py` : code for you to modify
- `tree_output.txt` : correct output for the program

In this lab, you will work with a binary tree. A partial implementation is provided for you, which you must edit. The file `tree.py` implements two classes:

- `Node` : a node in the tree
- `BST` : a binary search tree.

Exercises

All of the exercises here use free-standing functions, not methods of the `BST` class. Thus they will work with `Node` objects.

Each of these functions will be called by a method in the tree, and will usually be passed the root of the tree, initially.

1. Implement the function `preceding_node(node)`. It finds the node that comes just before the given node, and returns it.

   Assume that node is the root of a BST, so the following technique applies:

   a. If `node` is `None`, it has no predecessor. Return `None`

   b. Otherwise, if `node` has no left child, it has no predecessor. Return `None`.

   c. Otherwise, set `child = node.left`, and advance right, as many times as possible. When you reach a node that whose `right` is `None`, you are at the predecessor. Return it.

---

2. Implement the function `inorder_with_depth(node, depth)`. It runs inorder traversal, and prints each node's key, and its depth.

   Here is the inorder traversal algorithm in pseudocode:

```
inorder(node)
    if node isn't empty,
        inorder(node.left)
        visit node
        inorder(node.right)
```

Notice the `depth` parameter. Each time you call `inorder_with_depth`, this should increase. Hence, pass `depth+1` in your recursive calls.

To "visit" a node, just print its key, and the depth.

---

3. Implement the function `indented_print(node,depth)`.

   This function is identical to the previous one, except that the "visit" step should print `4×depth` spaces, followed by the key, all on the same line.

   For example the following tree:

```
          5
        /   \
      2      10
     / \    /  \
    1   3  9    20
```

   should produce this output:

```
                1
        2
                3
    5
                9
        10
                20
```

---

4. Implement the function `max_key(node)`, which finds the biggest key in the subtree whose root is node. Assume all keys are positive numbers.

   Don't assume the subtree is a BST. Instead, proceed recursively:

   a. Base case: if the node is empty, return 0

   b. otherwise, (recursive case):

   - find the max in the left subtree
   - find the max in the right subtree
   - return the biggest of: the node's key, the left max, and the right max

---

5. Implement the function `min_key(node)`, which finds the smallest key in the subtree whose root is node. Assume all keys are less than 1,000,000

   Don't assume the subtree is a BST. Instead, proceed recursively:

   a. Base case: if the node is empty, return 1000000

   b. otherwise, (recursive case):

   - find the min in the left subtree
   - find the min in the right subtree
   - return the smallest of the node's key, the left min, and the right min

---

6. Implement the function `is_root_of_bst(node)`. It returns `True` or `False` depending on whether the subtree is or isn't a true binary search tree.

   A subtree rooted at `node` is a BST if:

   a. `node` is empty, or
   b. it isn't empty, and

   - the biggest key on the left side is less than the node's key, and
   - the smallest key on the right side is more than the node's key, and
   - both children are BSTs

---

7. (10% Bonus) Implement the function `has_path_sum(node, total)`. It considers all possible root-to-leaf paths starting at node, down to a leaf. If there is path whose keys add up to total, return True. Otherwise, False.

   Example: this tree

```
      5
     / \
    4   8
   /   / \
  11  13  4
 /  \      \
7    2      1
```

   has a path with total 22: 5 -> 4 -> 11 -> 2 .

   Method:

   - if `node` is empty, return `True` if `total` is 0, otherwise `False`

   - otherwise, check if there is a left subtree, and whether the left subtree has a path adding up to `total - node.key`.

- otherwise, check the right subtree in the same way.

Submitting your work

Go to mycourses.unh.edu, find CS417, and the lab. Then click the “Submit” button and upload `tree2.py`.