CSC148 Summer 2018: Lab 6

Introduction

The goals of this lab are:

- To get you familiar with how Trees work
- To give you practice traversing Trees

Don't hesitate to make use of other resources for this lab, including the course notes, your TAs, instructor, or other students.

General Lab Notes

1. Make sure you have lab_pyta.txt downloaded and placed in the directory (or directories) where you'll be working.
2. To use PythonTA, include the following code (if you already have a main block, just add the body to the end of it):

```
if __name__ == '__main__':
    import python_ta
    python_ta.check_all(config="lab_pyta.txt")
```

Your lab_pyta.txt should be in the same folder as the .py files you're running. PythonTA will raise errors regarding style, specifying the lines you need to fix. You should get familiar with what the errors mean, and how to fix them: this will be important for your exercises and assignments.

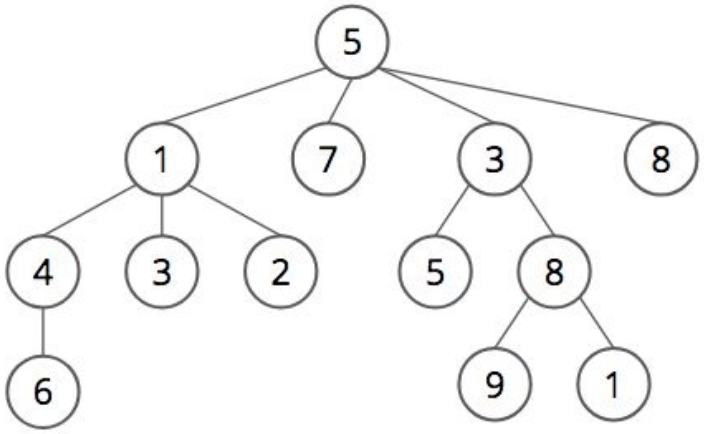
Getting Started

This lab will have you using Trees which only have very basic functionality. Download lab_trees.py or write your own Tree classes. This class contains only the ___init___ and a ___str___ method for Trees.

Creating a Tree

Within the `if __name__ == '__main__'` block, the Tree t was created. Make sure you understand how it was created, what it looks like, and what the subtrees in it are.

Create a new tree called `large_tree` that looks like the following:



We'll use `large_tree` as the example for all of the methods to be implemented in this lab. You'll want to make additional trees to test these methods on.

Traversals

If we printed the nodes of `large_tree` using a pre-order traversal, what's the order of values to be printed?

What if we used a post-order traversal? Level-order?

Counting all occurrences of a value in a Tree (count_occurrences)

Write a method in the Tree class called count_occurences that counts the number of times a value appears in the Tree. This should be done recursively.

For example, calling `large_tree.count_occurrences(3)` should return `2`.

Getting all of the values of the internal nodes (get_internal_values)

Recall that an internal node is any node that's not a leaf (i.e. it has at least 1 child).

Write a method in the Tree class called get_internal_values that returns the values of all internal nodes of the Tree in pre-order.

For example, calling `large_tree.get_internal_values()` should return `[5, 1, 4, 3, 8]`.

If you wanted this to be in post-order, what would you have to change? What would the post-order version be?

Finding the depth of a value (get_depth_of)

Write a method in the Tree class called get_depth_of that takes in a single value and returns the depth of the node with that value. This should return the depth of the first node found with that value during a pre-order traversal.

For example, calling `large_tree.get_depth_of(3)` should return `2` and `large_tree.get_depth_of(5)` should return `0`.

What would the post-order version be? If we called `large_tree.get_depth_of(5)` and used a post-order traversal, what would be returned instead of `0`?

Finding all values at a certain depth (get_values_at_depth)

Write a method in the Tree class called get_values_at_depth which, given a depth, return all of the values in the Tree at that depth.

For example, calling `large_tree.get_values_at_depth(2)` should return `[4, 3, 2, 5, 8]`. If no such nodes are found, then return an empty list.

Hint: You'll want to adjust depth with each recursive call you make. If we're looking for a value that's has a depth of 3 away from the root, then how far should it be from the subtrees?

Getting the maximum branching factor (get_max_branching_factor)

Write a method in the Tree class called get_max_branching_factor that returns the maximum branching factor of the Tree.

For example, calling `large_tree.get_max_branching_factor()` should return `4`.

Create a copy (copy)

Write a method in the Tree class that returns a copy of the Tree. All changes to the copy shouldn't affect the original, and vice versa.

For example, if you did:

```
large_tree_copy = large_tree.copy()
large_tree_copy.value = 10
large_tree_copy.children[0].value = 3
print(large_tree.value)  # This should print 5
print(large_tree_copy.value) # This should print 10
print(large_tree.children[0].value) # This should print 1
print(large_tree_copy.children[0].value) # This should print 3
```

Then large_tree itself shouldn't be changed at all when you print it, but large_tree_copy should have its root value as 10, and its first child's value being 3.

Hint: You'll want to create and return new tree using Tree(self.value). The children of this Tree should be copies of each subtree.

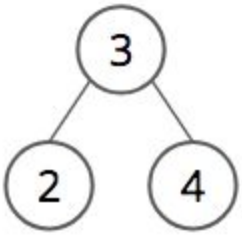
Printing the values of a Tree in Level-order (print_level_order)

In Week 3, we talked about printing values of a nested list in level-order by using a Queue. Do the same for a Tree. To do this:

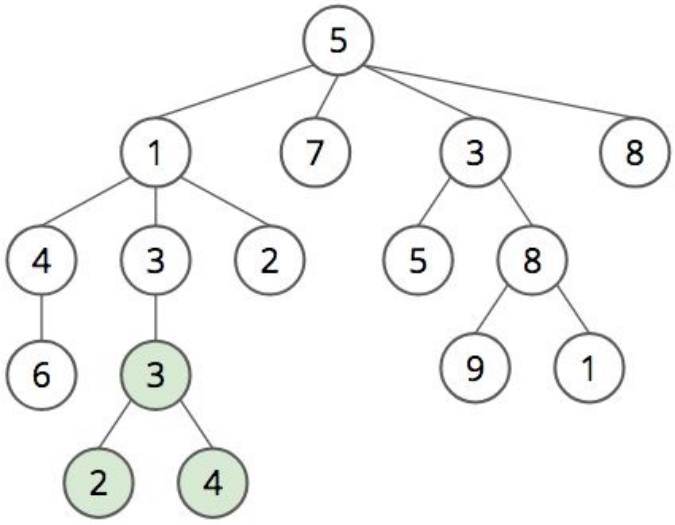
1. Make a Queue
2. Add our Tree to it
3. Remove the Tree at the front of the Queue
4. Print its value
5. Add all of its children to the end of the Queue
6. Repeat steps 3-5 until the Queue is empty.

Adding a Tree as the subtree of another Tree with a given value (add_subtree_to)

Write a method in the Tree class called add_subtree_to that takes in 2 parameters: a Tree, and a value. Add that Tree as a subtree of the Tree with the given value; this should only be added to the first Tree found with that value in a pre-order traversal. For example, if we had the following Tree `tree_to_add`:



And we called `large_tree.add_subtree_to(tree_to_add, 3)` then `large_tree` should look like:



Where the nodes in green belong to the subtree (`tree_to_add`) that was added.