Homework 09 : Trees

This week, we've been introduced to our second recursively-defined data structure, trees! (As a reminder, the first one was linked lists.) Recall from lecture that a tree is either:

- the empty tree, represented by `None` in Python, or
- a node that points to other trees. Nodes usually contain some data as well.

Important Note

As trees have a natural recursive definition and structure, functions that operate on tree often have simple and elegant recursive definitions. For this homework, all of your function implementations must be recursive and **you cannot use any iteration (i.e. for/while loops)**.

Part 1: Simple Tree Functions

In the starter code file, `simple_tree_functions.py`, implement the following functions.

- `height(t)`
  - Returns the height of the tree `t`.
  - Returns -1 for the empty tree.
- `size(t)`
  - Returns the number of nodes in the tree `t`.
- `find_min(t)`
  - Returns the minimum element contained within a node in the tree `t`.
  - Returns `float("inf")` for the empty tree.
- `find_max(t)`
  - Returns the maximum element contained within a node in the tree `t`.
  - Returns `float("-inf")` for the empty tree.
- `contains(t, k)`
  - Returns `True` if and only if `k` is contained within one of the nodes of `t` (returns `False` otherwise).
- `in_order(t)`
  - Returns a list containing contents of each node in the tree in the order of an in-order traversal of the tree.
- `pre_order(t)`
  - Returns a list containing contents of each node in the tree in the order of an pre-order traversal of the tree.
- `post_order(t)`
  - Returns a list containing contents of each node in the tree in the order of an post-order traversal of the tree.

Examples

```
"""
     0
    / \
  1     4
 /     / \
2     5   3
"""
>>> n1 = TreeNode(1, left=TreeNode(2))
>>> n4 = TreeNode(4, left=TreeNode(5), right=TreeNode(3))
>>> tree = TreeNode(0, left=n1, right=n4)
>>> height(tree)
2
>>> size(tree)
6
>>> find_min(tree)
0
>>> find_max(tree)
5
>>> contains(tree, 3)
True
>>> contains(tree, 6)
False
>>> in_order(tree)
[2, 1, 0, 5, 4, 3]
>>> pre_order(tree)
[0, 1, 2, 4, 5, 3]
>>> post_order(tree)
[2, 1, 5, 3, 4, 0]
```

Part 2: Higher-Order Tree Functions

A *higher-order function* is a function that does at least one of the following:

- takes one or more functions as arguments, or
- returns a function as its result.

In this part of the homework, we will explore some functions that take other functions as input and also operate on trees.

Often times the functions passed to our higher-order functions will take the form of a **lambda expression**, (e.g. `lambda x: x < 4`). These expressions provide an elegant way for us to create small anonymous functions. For a little more info on lambda expressions, check out:

- https://docs.python.org/3/reference/expressions.html#lambda
- https://www.w3schools.com/python/python_lambda.asp

In the starter code file, `higher_order_tree_functions.py`, implement the following functions.

- `count_failures(pred, t)`

  - Takes a predicate function (a function that returns either `True` or `False`) `pred`, and returns the number of nodes in the tree `t` whose data fails the predicate.
    * If `t` is a non-empty tree node, then its data fails the predicate if `pred(t.data)` returns `False`.

- `tree_map(f, t)`
  - Takes a function `f`, and returns a new tree consisting of the nodes from `t` such that `f` has been applied to the data within each of the nodes in `t`.
  - This function does NOT mutate the tree `t`, it returns a new tree.

- `tree_apply(f, t)`
  - Takes a function `f`, and *applies* it to the data held with each node of the tree `t`.
  - This function mutates the tree `t`.

Examples

```
"""
    0
   / \
  1   4
 /   / \
2   5   3
"""
>>> n1 = TreeNode(1, left=TreeNode(2))
>>> n4 = TreeNode(4, left=TreeNode(5), right=TreeNode(3))
>>> tree = TreeNode(0, left=n1, right=n4)

# Count Failures
>>> count_failures(lambda x: x % 2 == 0, tree)
3
# ^ 3 nodes contain integers that are not even
>>> count_failures(lambda x: x < 4, tree)
2
# ^ 2 nodes contain integers that are not less than 4
>>> count_failures(lambda x: x < 0, tree)
6
# ^ 6 nodes contain integers that are not less than 0

# Tree Map
>>> another_tree = tree_map(lambda x: x**2, tree)
>>> in_order(another_tree)
[4, 1, 0, 25, 16, 9]
>>> in_order(tree)
[2, 1, 0, 5, 4, 3]
# ^ original tree was not mutated by tree_map

# Tree Apply
>>> tree_apply(lambda x: x**3, tree)
```

```
>>> in_order(tree)
[8, 1, 0, 125, 64, 27]
# ^ original tree was mutated by tree_apply
```

Testing

Note that we are not requiring unit tests in your submission. At this point in the semester, we expect that you are developing your own test suites for these questions as we do not provide any unit tests on Gradescope. Therefore, your tests will not be graded, but developing a thorough suite of tests is the only way that you will be able to ensure your function implementations exhibit the correct behavior.

Imports

No imports allowed on this assignment, with the following exceptions:

- Any modules you have written yourself.
- `typing` - this is not required, but some students have requested it.

Submission

Submit the following files to Gradescope:

- `simple_tree_functions.py`
- `higher_order_tree_functions.py`

Please note this assignment is 100% manually graded. Students must submit individually by the due date (Tuesday, April 9th at 11:59 pm EST) to receive credit.