MUTABLE TREES 9

COMPUTER SCIENCE 61A

July 21, 2015

1 Trees in OOP

1.1 A New Implementation

Previously, we have seen trees defined as an abstract data type using lists. Let's look at another implementation using OOP syntax. With this implementation, we can easily specify specialized tree types such as binary trees through inheritance.

```
class Tree:
    """A tree with entry as its root value."""
    def __init__(self, entry, subtrees=[]):
        self.entry = entry
        for subtree in subtrees:
            assert isinstance(subtree, Tree)
        self.subtrees = list(subtrees)

    def is_leaf(self):
        return not self.subtrees
```

Notice that with this implementation we are able to mutate the entry of a tree by reassigning `tree.entry`. This was not possible when using ADT's because the abstraction barrier prevented us from seeing how the tree was implemented.

1.2 Questions

1. Consider the following definitions and assignments and determine what Python would output for each of the calls below *if they were evaluated in order*.

```
>>> t0 = Tree(0)
>>> t0.entry


>>> t0.subtrees


>>> t1 = Tree(0, [1, 2])#Is this a valid tree?

>>> t2 = Tree(0, [Tree(1), Tree(2, [Tree(3)])])
>>> t2.subtrees[0]

>>> t2.subtrees[1].subtrees[0].entry
```

2. Define a function `square_tree(t)` that squares every entry in the non-empty tree `t`. You can assume that every entry is a number.

```
def square_tree(t):
    """Mutates a Tree t by squaring all its elements."""
```

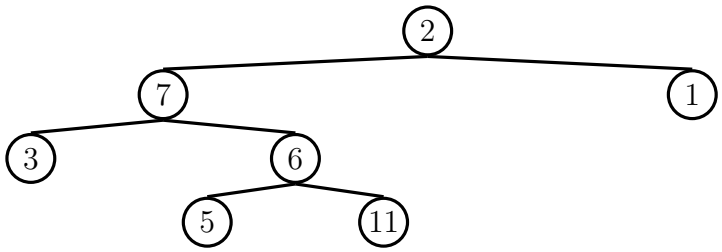
3. Assuming that every entry in `t` is a number, let's define `average(t)`, which returns the average of all the entries in `t`.

```
def average(t):
    """
    Returns the average value of all the entries in t.
    >>> t0 = Tree(0, [Tree(1), Tree(2, [Tree(3)])])
    >>> average(t0)
    1.5
    >>> t1 = Tree(8, [t0, Tree(4)])
    >>> average(t1)
    3.0
    """
```

1.3 Extra Questions

1. Define the procedure `find_path` that, given a Tree `t` and an `entry`, returns a list containing the nodes along the path required to get from the root of `t` to `entry`. If `entry` is not present in `t`, return `False`.
   Assume that the elements in `t` are unique. Find the path to an element.

   For instance, for the following tree, `find_path` should return:




```
>>> find_path(tree_ex, 5)
[2, 7, 6, 5]
def find_path(t, entry):
```

2 Binary Tree

Sometimes, it is more convenient to work with trees that have at most two subtrees per node. Instead of using a list to keep track of all the subtrees, we can use `left` and `right` to refer two the only two subtrees in a `BinaryTree`.

```
class BinaryTree:
    empty = ()

    def __init__(self, entry, left=empty, right=empty):
        self.entry = entry
        if left: assert isinstance(left, BinaryTree)
        if right: assert isinstance(right, BinaryTree)
        self.left = left
        self.right = right
```

2.1 Questions

1. Define a function `height(t)` that returns the height of a BinaryTree. The height is defined as the length of the *longest* path from the root node down to a leaf node. If a BinaryTree just consists of a root with no children, its height is 0.

```
def height(t):
    """Returns the height of the Tree t."""
```

2. Write the function `tree_max(t)` that takes in a tree and returns the max value in the tree.

```
def tree_max(t):
    """Returns the max entry in a BinaryTree t."""
```

2.2 Binary Search Tree

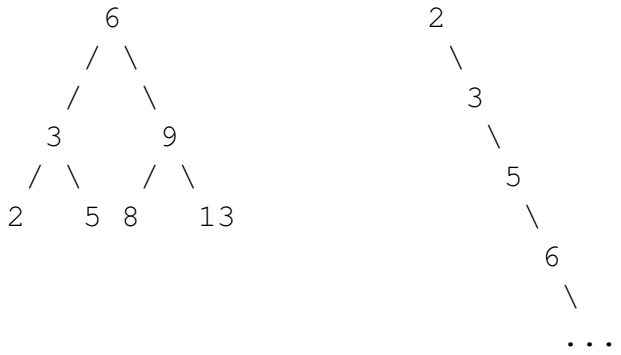
In a binary tree, each tree node has at most two subtrees: `left` and `right`. We can intruduce the concept of *binary search tree*. A binary search tree is a `BinaryTree` with the following additional constraints:

- the root entry is greater than or equal to all entries in the left side of the `BinaryTree`
- the root entry is lses than or equal to all entries in the right side of the `BinaryTree`

The purpose of binary search tree is to organize data in a way that allows for more efficient manipulation of the data.

2.3 More Questions, But Faster

1. Below are two binary search trees. The tree on the left is as shallow as possible, while the tree on the right containing the same data is as deep as possible. Suppose you have a function `tree_find(t, x)` that returns `True` when `x` is in binary search tree `t`, `False` otherwise.



   a. As a function of N, the number of entries in the tree, how long does the function `tree_find` take to run on the shallower tree? And the other tree?

   b. As a function of H, the height of the tree, how long does the function `tree_find` take to run on the shallower tree? And the other tree?

2. Given the input binary tree `t` is a search tree, write `tree_max` function which runs in $O(H)$, where H is the height of the tree.

```
def tree_max(t):
    """Returns the max entry in a binary search tree t."""
```

3. Using recursion, write `tree_find` function, that takes in a binary search tree `t` and a value `x`, and returns `True` only when `x` is in the tree, and `False` otherwise. Make sure your function run in $O(H)$, where H is the height of the tree.

```
def tree_find(t, x):
    """Return True if and only if x is an entry in binary
    search tree t.
    """
```

2.4 Extra Questions

1. Now rewrite `tree_find` with iteration. Think about whether you should use `for`, or `while` before you start.

```
def tree_find(t, x):
```