CS61A Discussion 5

OOP + some Tree review

OOP

What are Objects (& OOP)

- You have been using objects this whole time! (Everything in Python is an object)
- OOP allows us to approach problems by creating our own objects
- This means we can control
  - How the object behaves
  - What the object can do
  - What the object stores
- Best part about objects: instances! (instance variables: self.____)

Object functions

- def __init__(self, args…)
    - Special!
    - What happens when you create a new object (ex. For the Tree class Tree(10,[]))
- Dot function notation for other functions
    - All functions that you will call with dot notation require self as the first argument
    - Ex. def hello(self):
    - Python uses the object before the dot as the self argument

Inheritance

- Default for a class is `Class Tree(object)`
- However, we can have an object inherit from another user-defined object
    - “Is-a” relationship
- To call functions from the parent
    - [Parent Name].__init__(self, ….)
    - [Parent Name].hello(self, …)
    - The reason for this is a lot of the behavior may be the same, but then we can add new behavior for this class

Worksheet (1.1, 2.1, 2.2, 2.3)

Start Trees at 1:30

Approaching Tree Problems

- Look for hints!!!
  - Like with any coding problem the skeleton guides your code
  - For example, knowing types are consistent with hint at what to call/format
-

Fall 15 MT 2

3. (24 points) Return of the Digits

(a) **(4 pt)** Implement `complete`, which takes a `Tree` instance `t` and two positive integers `d` and `k`. It returns whether `t` is *d-k-complete*. A tree is *d-k-complete* if every node at a depth less than `d` has exactly `k` branches and every node at depth `d` is a leaf. *Notes:* The depth of a node is the number of steps from the root; the root node has depth 0. The built-in `all` function takes a sequence and returns whether all elements are true values: `all([1, 2])` is `True` but `all([0, 1])` is `False`. `Tree` appears on the Midterm 2 Study Guide.

```
def complete(t, d, k):
    """Return whether t is d-k-complete.

    >>> complete(Tree(1), 0, 5)
    True
    >>> u = Tree(1, [Tree(1), Tree(1), Tree(1)])
    >>> [ complete(u, 1, 3)   ,  complete(u, 1, 2)   ,  complete(u, 2, 3) ]
    [True, False, False]
    >>> complete(Tree(1, [u, u, u]), 2, 3)
    True
    """
    if not t.branches:

        return ________________________________________________________________

    bs = [____________________________________________________________________]

    return ________________________________________________________ and all(bs)
```

3. (24 points) Return of the Digits

(a) (4 pt) Implement `complete`, which takes a `Tree` instance `t` and two positive integers `d` and `k`. It returns whether `t` is *d-k-complete*. A tree is *d-k-complete* if every node at a depth less than `d` has exactly `k` branches and every node at depth `d` is a leaf. *Notes:* The depth of a node is the number of steps from the root; the root node has depth 0. The built-in `all` function takes a sequence and returns whether all elements are true values: `all([1, 2])` is `True` but `all([0, 1])` is `False`. `Tree` appears on the Midterm 2 Study Guide.

```
def complete(t, d, k):
    """Return whether t is d-k-complete.

    >>> complete(Tree(1), 0, 5)
    True
    >>> u = Tree(1, [Tree(1), Tree(1), Tree(1)])
    >>> [ complete(u, 1, 3)  ,  complete(u, 1, 2)  ,  complete(u, 2, 3) ]
    [True, False, False]
    >>> complete(Tree(1, [u, u, u]), 2, 3)
    True
    """
    if not t.branches:

        return d == 0

    bs = [complete(b, d-1, k) for b in t.branches]

    return len(t.branches) == k and all(bs)
```

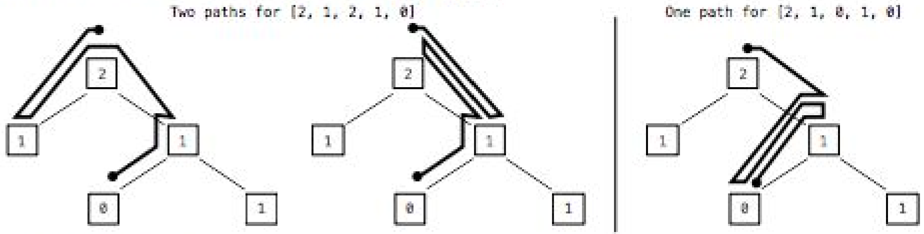
Fall 14 MT 2

Note: BinaryTree has three new operations: t.right, t.left, t.parent (If they don't exist then it returns BinaryTree.empty)

(b) **(4 pt)** Fill in the blanks of the implementation of `paths`, a function that takes two arguments: a `GrootTree` instance `g` and a list `s`. It returns the number of paths through `g` whose entries are the elements of `s`. A path through a `GrootTree` can extend either to a branch or its `parent`.

You may assume that the `GrootTree` class is implemented correctly and that the list `s` is non-empty.

The two paths that have entries `[2, 1, 2, 1, 0]` in `fib_groot(3)` are shown below (left). The one path that has entries `[2, 1, 0, 1, 0]` is shown below (right).




```
def paths(g, s):
    """The number of paths through g with entries s.

    >>> t = fib_groot(3)
    >>> paths(t, [1])
    0
    >>> paths(t, [2])
    1
    >>> paths(t, [2, 1, 2, 1, 0])
    2
    >>> paths(t, [2, 1, 0, 1, 0])
    1
    >>> paths(t, [2, 1, 2, 1, 2, 1])
    8
    """
    if g is BinaryTree.empty ________________________________:

        return 0

    elif ____________________________________________________:

        return 1

    else:

        xs = [___________________________________________________]

        return sum([ ____________________________________ for x in xs])
```

```
if g is BinaryTree.empty or s == [] or g.entry != s[0]:
    return 0
elif len(s) == 1 and g.entry == s[0]:
    return 1
else:
    extensions = [g.left, g.right, g.parent]
    return sum(paths(x, s[1:]) for x in extensions)
```

Return should have brackets around list comprehension