CS 61A
Spring 2021

More Trees and OOP

Lost 5: March 5, 2021 Solutions

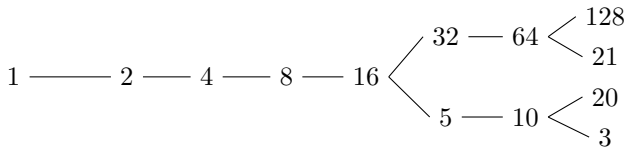
1 Learning Goals

- Learn to approach complex tree questions
- Touch on iterators and generators in the context of other topics
- Learn the basics of object-oriented programming

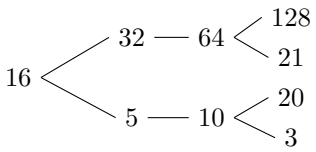
2 Trees (Exam-Level)

2.1 We can represent the hailstone sequence as a tree in the figure below, showing the route different numbers take to reach 1. Remember that a hailstone sequence starts with a number n, continuing to $n/2$ if n is even or $3n + 1$ if n is odd, ending with 1. Write a function `hailstone_tree(n, h)` which generates a tree of height `h`, containing hailstone numbers that will reach `n`.

Hint: A node of a hailstone tree will always have at least one, and at most two branches (which are also hailstone trees). Under what conditions do you add the second branch?



`hailstone_tree(1, 7)`



`hailstone_tree(16, 3)`

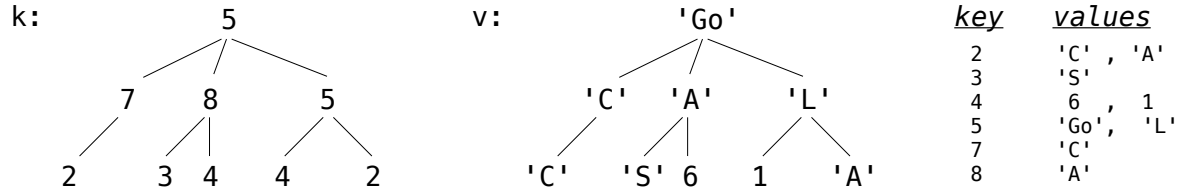
```
def hailstone_tree(n, h):
    """Generates a tree of hailstone numbers that will
       reach N, with height H.
    >>> hailstone_tree(1, 0)
    [1]
    >>> hailstone_tree(1, 4)
    [1, [2, [4, [8, [16]]]]]
    >>> hailstone_tree(8, 3)
    [8, [16, [32, [64]], [5, [10]]]]
    """


    if h == 0:
        return tree(n)
    branches = [hailstone_tree(n * 2, h - 1)]
    if (n - 1) % 3 == 0 and ((n - 1) // 3) % 2 == 1 and (n - 1) // 3 > 1:
        branches += [hailstone_tree((n - 1) // 3, h - 1)]
    return tree(n, branches)
```

2.2 **Fall 2018 Midterm 2, Question 7: Trictionary or Treat** A trictionary is a pair of *Tree* instances k and v that have identical structure: each node in k has a corresponding node in v. The labels in k are called keys. Each key may be the label for multiple nodes in k, and the values for that key are the labels of all the corresponding nodes in v.

A lookup function returns one of the values for a key. Specifically, a lookup function for a node in k is a function that takes v as an argument and returns the label for the corresponding node in v.

Implement the generator function *lookups*, which takes a *Tree* instance k and some *key*. It yields all lookup functions for nodes in k that have key as their label.




```
k = Tree(5, [Tree(7, [Tree(2)]), Tree(8, [Tree(3), Tree(4)]), Tree(5, [Tree(4), Tree(2)])])
v = Tree('Go', [Tree('C', [Tree('C')]), Tree('A', [Tree('S'), Tree(6)]), Tree('L', [Tree(1), Tree('A'
    )])])

def lookups(k, key):
    """Yield one lookup function for each node of k that has the label key.
    >>> [f(v) for f in lookups(k, 2)]
    ['C', 'A']
    >>> [f(v) for f in lookups(k, 3)]
    ['S']
    >>> [f(v) for f in lookups(k, 6)]
    []
    """

    if ____________________________________________________________________________________:

        yield lambda v: ____________________________________________________________________

    for i in range(len(k.branches)):

        ____________________________________________________________________________________:

            yield new_lookup(i, lookup)

def new_lookup(i, f):
```

```
    def g(v):
        return ____________________________________________________________________________
    return
k = Tree(5, [Tree(7, [Tree(2)]), Tree(8, [Tree(3), Tree(4)]), Tree(5, [Tree(4), Tree(2)])])
v = Tree('Go', [Tree('C', [Tree('C')]), Tree('A', [Tree('S'), Tree(6)]), Tree('L', [Tree(1), Tree('A'
    )])])
def lookups(k, key):
    """Yield one lookup function for each node of k that has the label key.
    >>> [f(v) for f in lookups(k, 2)]
    ['C', 'A']
    >>> [f(v) for f in lookups(k, 3)]
    ['S']
    >>> [f(v) for f in lookups(k, 6)]
    []
    """
    if k.label == key:
        yield lambda v: v.label
    for i in range(len(k.branches)):
        for lookup in lookups(k.branches[i], key):
            yield new_lookup(i, lookup)
def new_lookup(i, f):
    def g(v):
        return f(v.branches[i])
```

```
return g
```

3 Object-Oriented Programming

3.1 What is the relationship between a class and an ADT?

In general, we can think of an abstract data type as defined by some collection of selectors and constructors, together with some behavior conditions. As long as the behavior conditions are met (such as the division property above), these functions constitute a valid representation of the data type.

There are two different layers to the abstract data type:

1) The program layer, which uses the data, and
2) The concrete data representation that is independent of the programs that use the data. The only communication between the two layers is through selectors and constructors that implement the abstract data in terms of the concrete representation.

Classes are a way to implement an Abstract Data Type. But, ADTs can also be created using a collection of functions, as shown by the rational number example. (See Composing Programs 2.2)

3.2 What is the definition of a Class? What is the definition of an Instance?

Class: a template for all objects whose type is that class that defines attributes and methods that an object of this type has.
Instance: A specific object created from a class. Each instance shares class attributes and stores the same methods and attributes. But the values of the attributes are independent of other instances of the class. For example, all humans have two eyes but the color of their eyes may vary from person to person.

3.3 What is a Class Attribute? What is an Instance Attribute?

Class Attribute: A static value that can be accessed by any instance of the class and is shared among all instances of the class.
Instance Attribute: A field or property value associated with that specific instance of the object.

3.4 What Would Python Display?

```
class Foo():
    x = 'bam'
    def __init__(self, x):
        self.x = x
    def baz(self):
        return self.x
```

```
class Bar(Foo):
        x = 'boom'
        def __init__(self, x):
                Foo.__init__(self, 'er' + x)
        def baz(self):
                return Bar.x + Foo.baz(self)

foo = Foo('boo')

Foo.x

'bam'

foo.x

'boo'

foo.baz()

'boo'

Foo.baz()

Error

Foo.baz(foo)

'boo'

bar = Bar('ang')
Bar.x

'boom'

bar.x

'erang'

bar.baz()

'boomerang'
```

3.5 What Would Python Display?

```
class Student:
        def __init__(self, subjects):
                self.current_units = 16
                self.subjects_to_take = subjects
                self.subjects_learned = {}
```

```
                self.partner = None

        def learn(self, subject, units):
                print('I just learned about ' + subject)
                self.subjects_learned[subject] = units
                self.current_units -= units

        def make_friends(self):
                if len(self.subjects_to_take) > 3:
                        print('Whoa! I need more help!')
                        self.partner = Student(self.subjects_to_take[1:])
                else:
                        print("I'm on my own now!")
                        self.partner = None

        def take_course(self):
                course = self.subjects_to_take.pop()
                self.learn(course, 4)
                if self.partner:
                        print('I need to switch this up!')
                        self.partner = self.partner.partner
                        if not self.partner:
                                print('I have failed to make a friend :(')

tim = Student(['Chem1A', 'Bio1B', 'CS61A', 'CS70', 'CogSci1'])
tim.make_friends()
```

```
Whoa! I need more help!
```

```
print(tim.subjects_to_take)
```

```
['Chem1A', 'Bio1B', 'CS61A', 'CS70', 'CogSci1']
```

```
tim.partner.make_friends()
```

```
Whoa! I need more help!
```

```
tim.take_course()
```

```
I just learned about CogSci1
I need to switch this up!
```

```
tim.partner.take_course()
```

```
I just learned about CogSci1
```

```
tim.take_course()
```

```
I just learned about CS70
I need to switch this up!
I have failed to make a friend :(
tim.make_friends()

I'm on my own now!
```

3.6 Fill in the implementation for the Cat and Kitten classes. When a cat meows, it should say "Meow, (name) is hungry" if it is hungry, and "Meow, my name is (name)" if not. Kittens do the same thing as cats, except they say "i'm baby" instead of "meow", and they say "I want mama (parent's name)" after every call to meow().

```
>>>cat = Cat('Tuna')
>>>kitten = kitten('Fish', cat)
>>>cat.meow()
meow, Tuna is hungry
>>>kitten.meow()
i'm baby, Fish is hungry
I want mama Tuna
>>>cat.eat()
meow
>>>cat.meow()
meow, my name is Tuna
>>>kitten.eat()
i'm baby
>>>kitten.meow()
meow, my name is Fish
I want mama Tuna
```

```
class Cat():
    noise = 'meow'
    def __init__(self, name):

        self.name = name
        self.hungry = True
    def meow(self):

        if self.hungry:
            print(self.noise + ', ' + self.name ' + is hungry!')
        else:
            print(self.noise + ', my name is ' + self.name)
    def eat(self):
        print(self.noise)
        self.hungry = False
```

```
class Kitten(Cat):

    noise = "i'm baby"
    def __init__(self, name, parent):
        Cat.__init__(self, name)
        self.parent = parent
    def meow(self):
        Cat.meow(self)
        print('I want mama' + parent.name)
```