Practice with a `Circle` Class

Imagine you were crudely modeling urban sprawl using circles of varying radii. You might be interested in creating this abstraction because you're interested in both the area and diameter of certain types of sprawl. This is an excellent opportunity to create a new `Circle` type with a Python class.

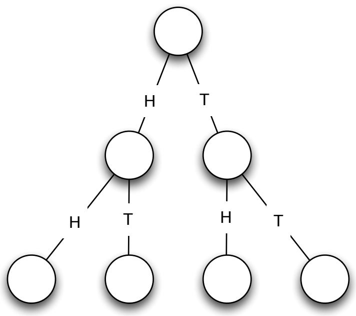
- A natural way of describing a circle is with a radius. This means the internal state of our circle should include radius as a *member* (or *instance*) variable.
- Thus, the *constructor* or *initializer* method should have one parameter, `radius`, besides the `self` parameter.
- The `Circle` class needs to store this value as an instance variable.

```python
import math

class Circle:

    def __init__(self, radius):
        self.radius = radius

    def diameter(self):
        return self.radius*2

    def area(self):
        return math.pi * self.radius**2

    def __repr__(self):
        return "Circle({})".format(self.radius)
```

```
>>> import circle
>>> c = circle.Circle(1.0)
>>> c
Circle(1.0)
>>> c.radius
1.0
>>> c.diameter()
2.0
>>> c.area()
3.141592653589793
>>>
```

1 Binary Trees

Imagine trying to represent the process of tossing a two-sided coin 2 times in a row, or more generally, n times in a row.



The natural structure that emerges is what computer scientists call a *binary tree*. In its most basic form, we can think of a binary tree as being composed of a *root node*, and two other, simpler binary trees, which are the *left child* and the *right child*. The simplest binary tree is a leaf, which is just a root node with no left or right children. This definition is inductive: we start with a base case—the leaf—and then define binary trees in terms of simpler binary trees. This type of induction is called *structural induction* and most modern programming languages, including Python, support it. This means we're free to write a Binary tree class as follows:

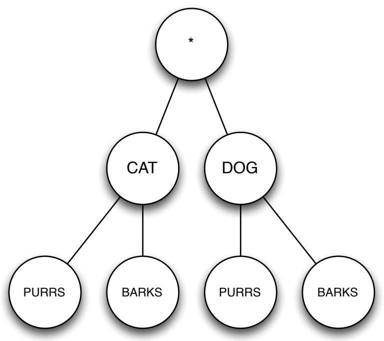
```
class BinaryTree:

  def __init__(self, root, left=None, right=None):
    self.root = root
    self.left = left
    self.right = right

  def has_left(self):
    return self.left is not None

  def has_right(self):
    return self.right is not None

  def leaf(self):
    """ returns True if and only if this tree is a leaf """
  return not (self.has_left() or self.has_right())
```

Computer scientists often call this a *recursive* data structure because it is defined *recursively*. Let's look at this code more closely.

- The `__init__` method takes three parameters (we usually don't count the `self` parameter of a method because when you call the method, the `self` is implicit passed) that correspond to the root node and the left and right child trees respectively.
- The `left` and `right` parameters are called *keyword parameters* and they have default values of `None` so that any binary tree created only with a root node is a leaf.
- The `leaf` method returns true if and only if the node is a leaf. This is a convenience method so that any node can tell whether it is a leaf or not.

1.1 Constructing all Paths

Imagine that you're given a binary tree where each node is a string and you wish to construct all possible sentences flowing from top-to-bottom—from the root to the leaves. This corresponds to considering all possible paths from the root to the leaves.



A natural way of constructing these paths is to use the recursive definition of the binary tree to our advantage. Here's how we'll think about it.

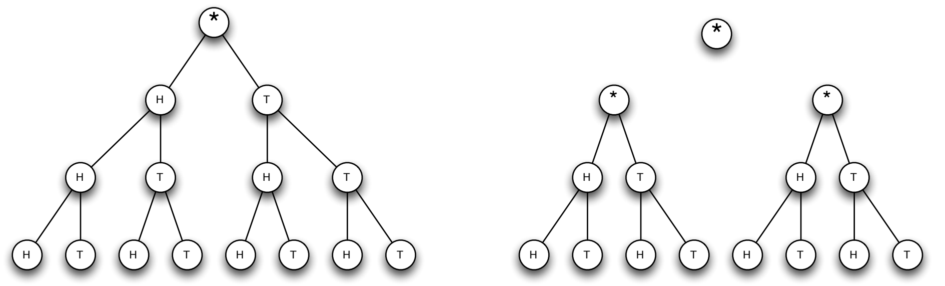
- First, our `all_paths` method should return a list of paths, where each path is a list of node values.
- If the tree is a leaf, then there is only one path so we'd return a list containing a single path, which is a list corresponding to the root of the tree. In other words, `all_paths` should return `[[self.root]]`
- If the tree is not a leaf, then we can (1) recursively find all the left paths if they exist, insert our root node value to the front of all of them and (2) recursively find all the right paths if the exist and do the same. We can then return the concatenation of these two lists.

Writing this code in Python is almost like transcribing our above sketch.

```
if self.leaf():
    return [[self.root]]
else:
    paths = []

    if self.has_left():
        left_paths = self.left.all_paths()
        for path in left_paths:
            path.insert(0, self.root)
        paths.extend(left_paths)

    if self.has_right():
        right_paths = self.right.all_paths()
        for path in right_paths:
            path.insert(0, self.root)
        paths.extend(right_paths)

    return paths
```

1.2 Generating Coin Flips

Let's reconsider our representation for generating all outcomes of tossing a two-sided coin n times. The root node represents, in some sense, not tossing the coin. The two children represent the possibilities after tossing the coin once. Their two children, respectively, represent tossing it again. Given some value n, how would we create this tree? One thing to notice is that the tree representing three tosses, is just a combination of the two trees representing two tosses, with the root node value now replaced with the respective choice. What is zero tosses? Just a leaf with a star value!



This leads to the following algorithm recursive construction algorithm.

```
from tree import BinaryTree
import sys

def build_tree(n):
   """
   Create a binary tree corresponding to all possible
   outcomes of 'n' tosses of a two−sided coin
   """
   if n == 0:
      return BinaryTree('*')
   else:
      left = build_tree(n−1)
   right = build_tree(n−1)
      left.root = 'H'
      right.root = 'T'
      return BinaryTree('*', left, right)
```

Now we can use our `all_paths` method to generate all possible coin flips. These paths will all start with a '*' so we should ignore the first node in each path. We can print them out by joining them together with a '-'. Viola!

```

def generate_tosses(n):
   return [path[1:] for path in build_tree(n).all_paths()]

if __name__ == '__main__':
   for path in generate_tosses(int(sys.argv[1])):
   print("−".join(path))
```