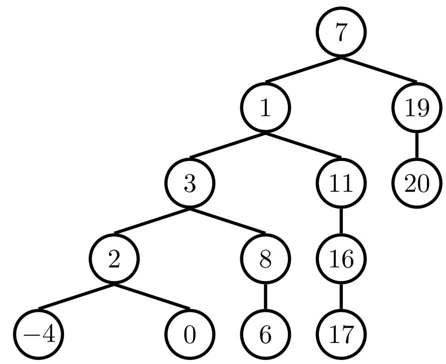
CS 61A Summer 2017 | Trees Discussion 5: July 6, 2017

1 Trees



In computer science, **trees** are recursive data structures that are widely used in various settings. The diagram to the right is an example of a simple tree.

Notice that the tree branches downward. In computer science, the **root** of a tree starts at the top, and the **leaves** are at the bottom.

Some terminology regarding trees:

- **Parent node**: A node that has branches. Parent nodes can have multiple branches.
- **Branch node**: A node that has a parent. A branch node can only belong to one parent.
- **Root**: The top node of the tree. In our example, the node that contains 7 is the root.
- **Value**: The value at a node. In our example, all of the integers are values.
- **Leaf**: A node that has no branches. In our example, the nodes that contain -4, 0, 6, 17, and 20 are leaves.
- **Branch**: Notice that each branch of a parent is itself the root of a smaller tree. In our example, the node containing 1 is the root of another tree. This is why trees are *recursive* data structures: trees have branches, which are trees themselves.
- **Depth**: How far away a node is from the root. In other words, the number of edges between the root of the tree to the node. In the diagram, the node containing 19 has depth 1; the node containing 3 has depth 2. Since there are no edges between the root of the tree and itself, the depth of the root is 0.
- **Height**: The depth of the lowest leaf. In the diagram, the nodes containing -4, 0, 6, and 17 are all the "lowest leaves," and they have depth 4. Thus, the entire tree has height 4.

In computer science, there are many different types of trees. Some vary in the number of branches each node has; others vary in the structure of the tree.

Implementation

A tree has both a value for the root node and a sequence of branches, which are also trees. In our implementation, we represent the branches as a list of trees. Since a tree is an abstract data type, our choice to use lists is simply an implementation detail.

- The arguments to the constructor `tree` are the value for the root node and a list of branches.
- The selectors for these are `root` and `branches`.

Note that `branches` returns a list of trees and not a tree directly. Although trees are represented as lists in this implementation, it is important to recognize when working with a tree or a list of trees.

We have also provided a convenience function, `is_leaf`.

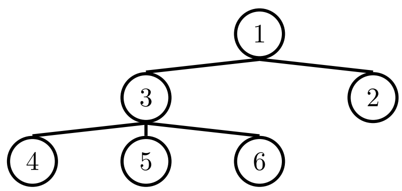
```
# Constructor
def tree(root, branches=[]):
    for branch in branches:
        assert is_tree(branch)
    return [root] + list(branches)

# Selectors
def root(tree):
    return tree[0]

def branches(tree):
    return tree[1:]

#For convenience
def is_leaf(tree):
    return not branches(tree)
```

It's simple to construct a tree. Let's try to create the tree below.




```
#Example tree construction
t = tree(1,
      [tree(3,
          [tree(4),
           tree(5),
           tree(6)]),
      tree(2)])
```

Questions

1.1 Define a function `tree_max(t)` that returns the largest number in a tree.

```
def tree_max(t):
    """Return the max of a tree."""
```

1.2 Define a function `height(t)` that returns the height of a tree. Recall that the height of a tree is the length of the longest path from the root to a leaf.

```
def height(t):
    """Return the height of a tree"""
```

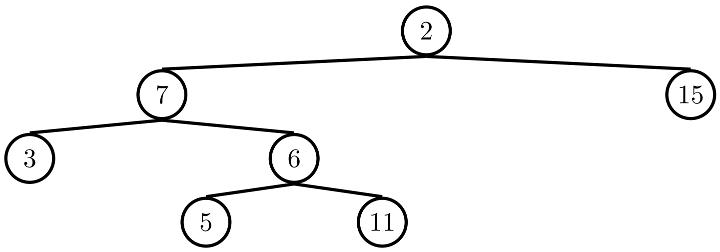
1.3 Define a function `tree_size(t)` that returns the number of nodes in a tree.

```
def tree_size(t):
    """Return the size of a tree."""
```

More Fun with Trees!

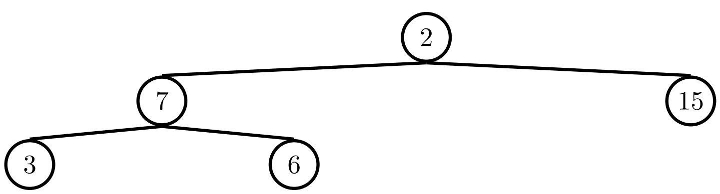
1.1 Define the procedure `find_path(tree, x)` that, given a tree `tree` and a value `x`, returns a list containing the nodes along the path required to get from the root of `tree` to a node `x`. If `x` is not present in `tree`, return `None`. Assume that the entries of `tree` are unique.

For the following tree, `find_path(t, 5)` should return `[2, 7, 6, 5]`




```
def find_path(tree, x):
    """
    >>> find_path(t, 5)
    [2, 7, 6, 5]
    >>> find_path(t, 10)  # returns None
    """
```

1.2 Implement a prune function which takes in a tree `t` and a depth `k`, and should return a new tree that is a copy of only the first `k` levels of `t`. For example, if `t` is the tree shown in the previous question, then `prune(t, 2)` should return the tree



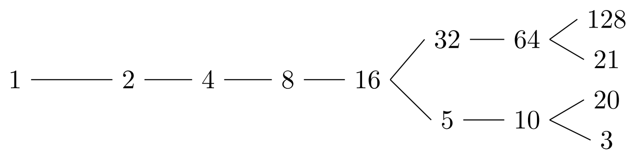

```
def prune(t, k):
```

1.3 An **expression tree** is a tree that contains a function for each non-leaf node, which can be either '+' or '*'. All leaves are numbers. Implement `eval_tree`, which evaluates an expression tree to its value. You may want to use the functions `sum` and `prod`, which take a list of numbers and compute the sum and product respectively.

```
def eval_tree(tree):
    """Evaluates an expression tree with functions as root
    >>> eval_tree(tree(1))
    1
    >>> expr = tree('*', [tree(2), tree(3)])
    >>> eval_tree(expr)
    6
    >>> eval_tree(tree('+', [expr, tree(4), tree(5)]))
    15
    """
```

1.4 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?




```python
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]]]]
    """
```