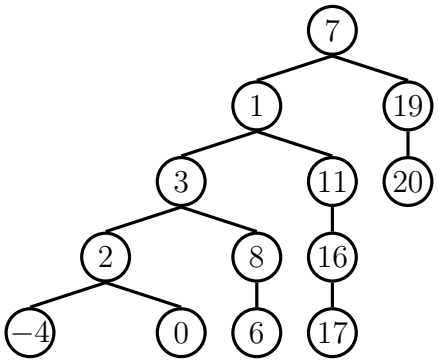
TREES 5

COMPUTER SCIENCE 61A

July 7, 2016

1 Trees

In computer science, **trees** are recursive data structures that are widely used in various settings. This is a diagram 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 children. Parent nodes can have multiple children.
- **Child node**: A node that has a parent. A child 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.
- **Leaf**: A node that has no children. In our example, the nodes that contain -4, 0, 6, 17, and 20 are leaves.

- **Subtree**: Notice that each child 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 are made up of subtrees, 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 children each node has; others vary in the structure of the tree.

A tree has both an entry and a sequence of children, which are also trees. In our implementation, we represent the children as lists of subtrees. Since a tree is an abstract data type, our choice to use lists is simply an implementation detail.

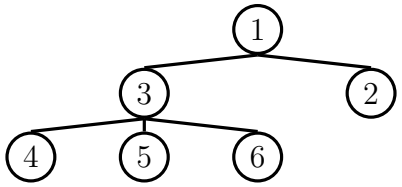
- The arguments to the constructor, `tree`, as a value for the entry and a list of children.
- The selectors are `entry` and `children`.

```
# Constructor
def tree(entry, children=[]):
    return [entry] + list(children)

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

def children(tree):
    return tree[1:]
```

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

```
def is_leaf(tree):
    return not children(tree)
```

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



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

1.1 Question

1. Define a function `square_tree(t)` that squares every item in the tree `t`. It should return a new tree. You can assume that every item is a number.

```
def square_tree(t):
    """Return a tree with the square of every element in t"""
```

Solution:

```
    sq_children = [square_tree(child) for child in children
       (t)]
    return tree(entry(t)**2, sq_children)
```

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"""
```

Solution:

```
    if is_leaf(t):
        return 0
    return 1 + max([height(subtree) for subtree in children
       (t)])
```

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."""
```

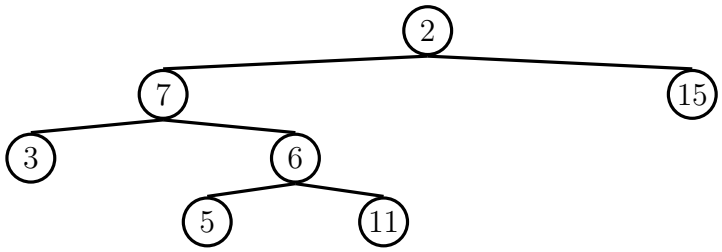
Solution:

```
    return 1 + sum([tree_size(child) for child in children(
        t)])
```

1.2 More fun with Trees

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]`



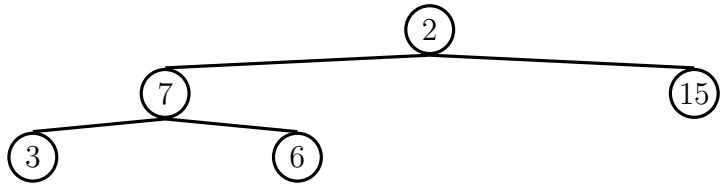

```
def find_path(tree, x):
    """
    >>> find_path(t, 5)
    [2, 7, 6, 5]
    >>> find_path(t, 10)  # returns None
    """
```

Solution:

```
    if entry(tree) == x:
        return [entry(tree)]
    node, trees = entry(tree), children(tree)
    for path in [find_path(t, x) for t in trees]:
        if path:
            return [node] + path
```

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



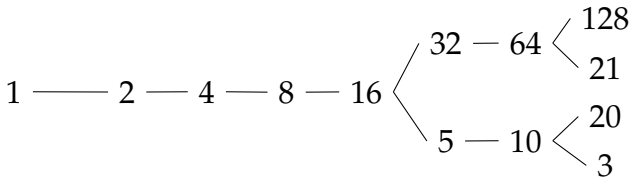

```
def prune(t, k):
```

Solution:

```
    if k == 0:
        return tree(entry(t), [])
    else:
        return tree(entry(t), [prune(child, k - 1) for
           child in children(t)])
```

3. 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 children (which are also hailstone trees). Under what conditions do you add the second branch?




```
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]]]]
    """
```

Solution:

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