Trees

15-110 – Wednesday 10/08

Quizlet4

Learning Goals

- Identify core parts of **trees**, including **nodes**, **children**, the **root**, and **leaves**
- Use **binary trees** implemented with dictionaries when reading and writing code

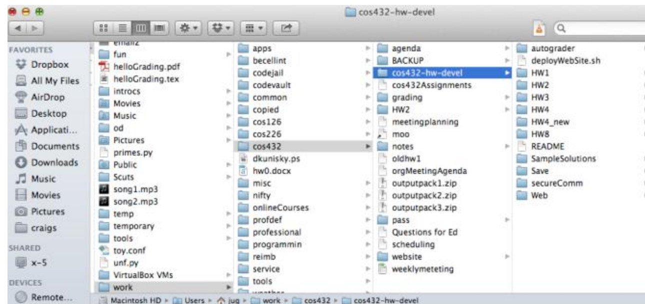
Trees

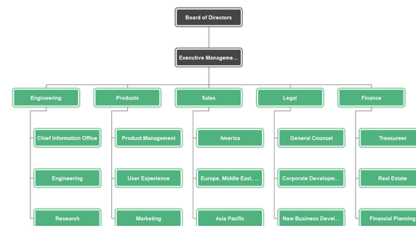
Trees Hold Hierarchical Data

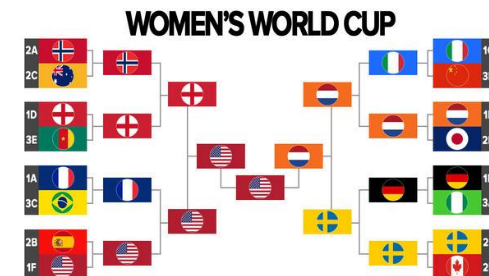
Sometimes we work with data that is **hierarchical** in nature. In this context, 'hierarchical' means that data occurs at different **levels** that are connected in some way.

Hierarchical data shows up in many different contexts.

- **File systems** in computers – each folder is a rank above the files it contains
- **Company organization schemas** – the CEO at the top, interns at the bottom
- **Sports tournament brackets** – the overall winner is ranked highest





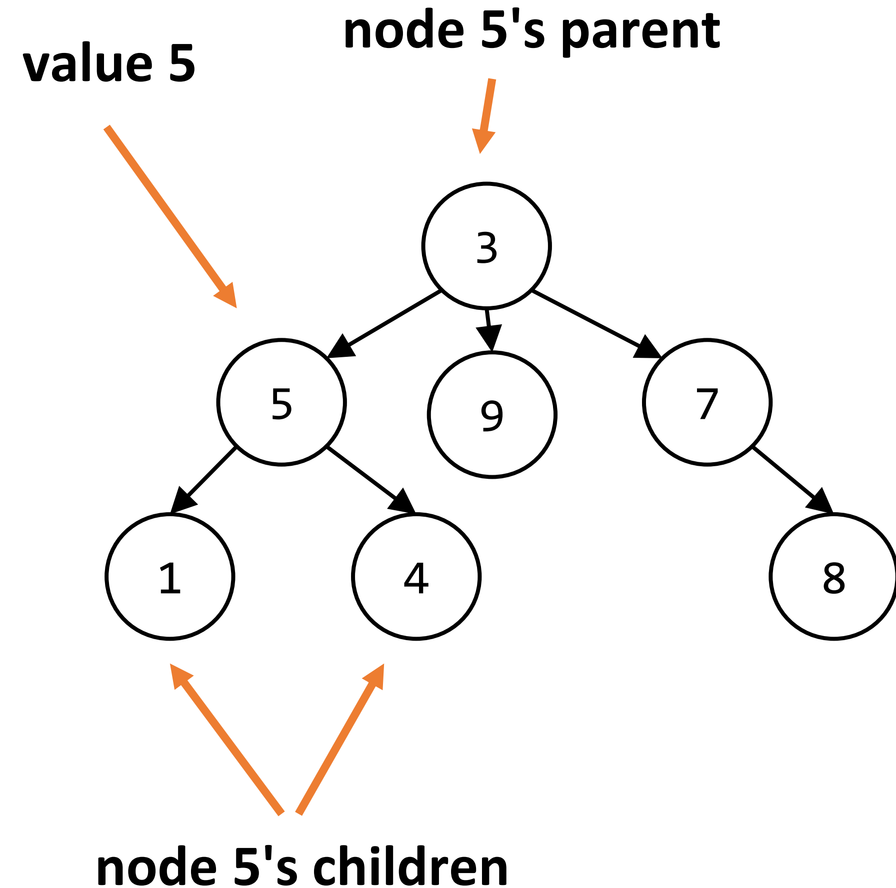


Trees are Hierarchical

A **tree** is a hierarchical data structure composed of **nodes** (circles in the example shown to the right).

Each node can hold a **value** (its data).

The node connected the level above a node is called its **parent**, and nodes connected on the level below are called its **children**. In general, a node has exactly one parent and can have any number of children.



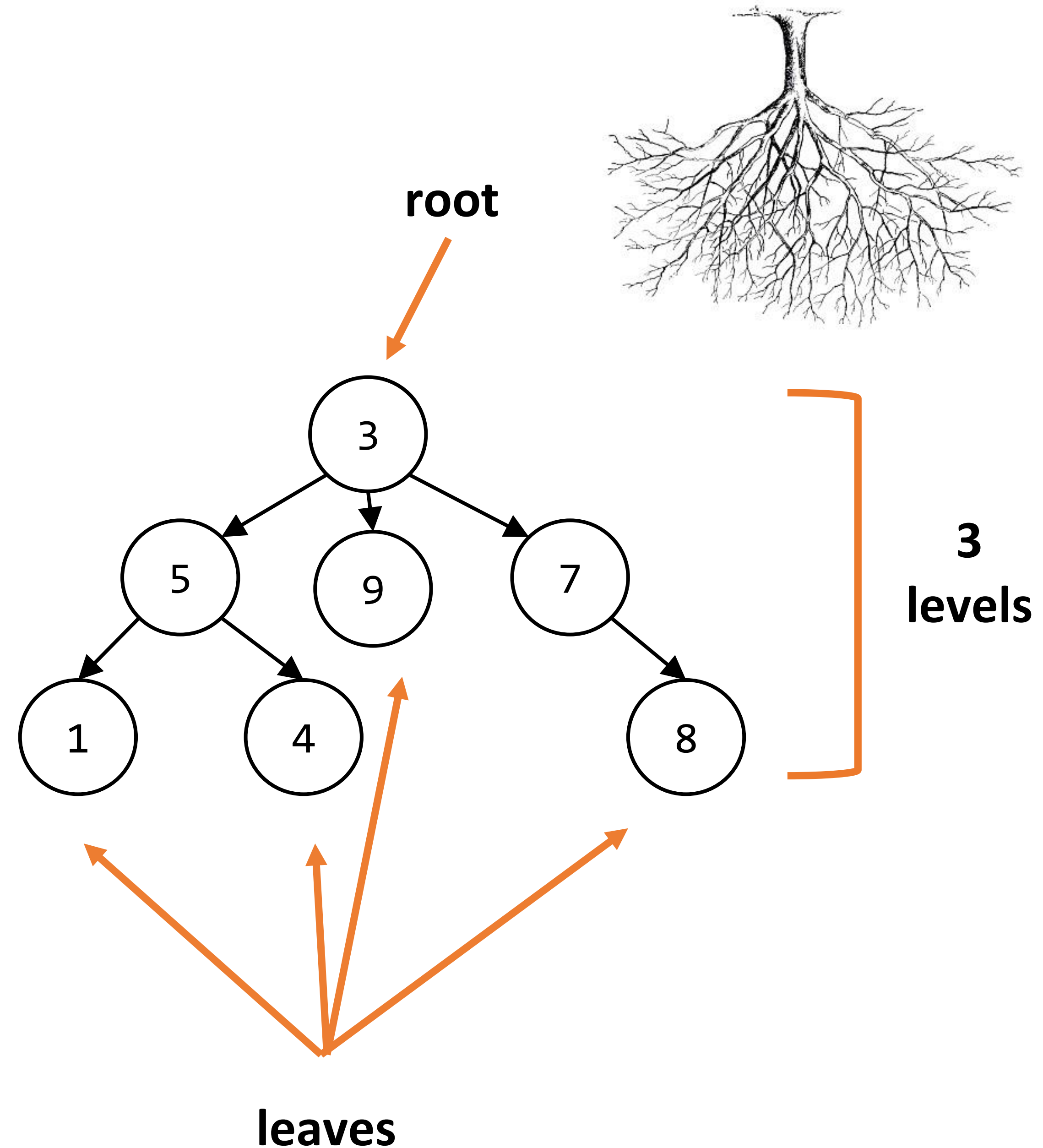
Trees are Upside-down

Unlike real trees, trees in computer science grow downward!

The top-most node is called the **root**. Every non-empty tree has a root. The root has no parent.

A node can have other nodes as children, and those nodes can have children as well. The number of **levels** a tree can have is unlimited.

Nodes that have no children are called **leaves**.



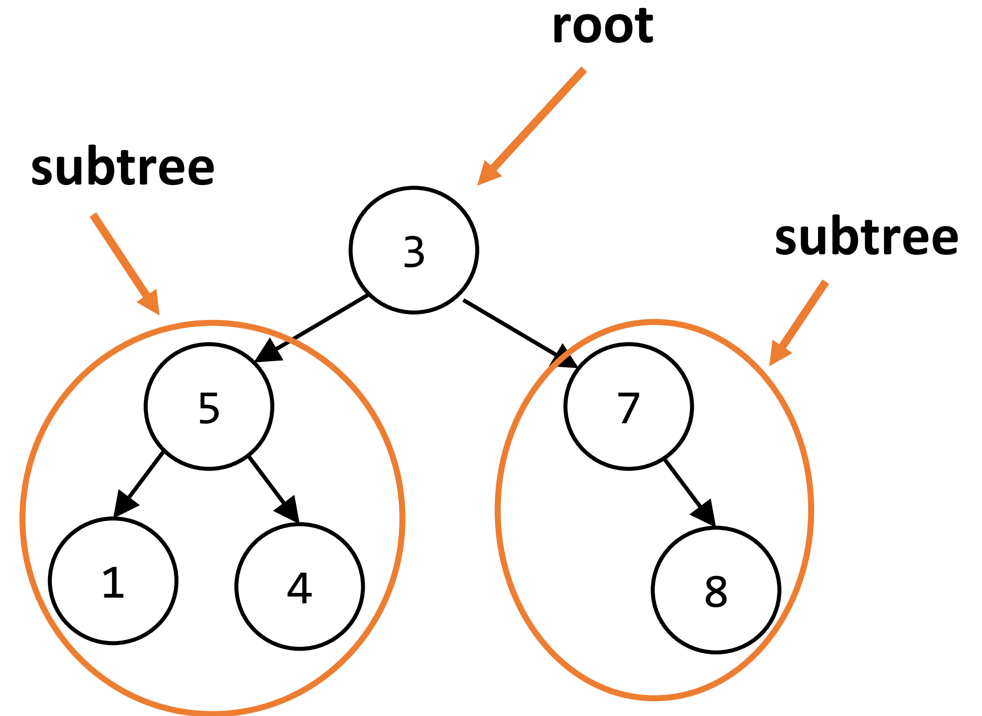
Trees are Recursive

A tree is a naturally recursive data structure. Each node's children are **subtrees**, which are just trees again.

For example, the root node 3 has two children that are subtrees. The subtree on the left has a root node 5. The subtree on the right has a root node 7. Each of these root nodes have their own subtrees as children.

Our **base case** can usually be an empty tree.

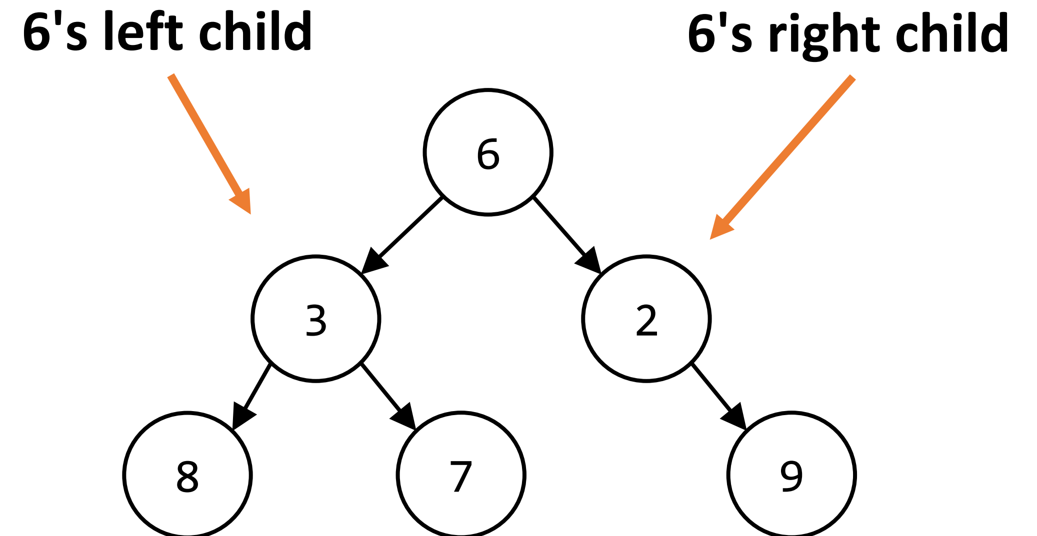
The **recursive case** makes the problem smaller by removing the root and repeating on each of the children, which are also trees.



Binary Trees

It's possible to write algorithms for trees that have an arbitrary number of children, but in this class we'll focus on **binary trees**.

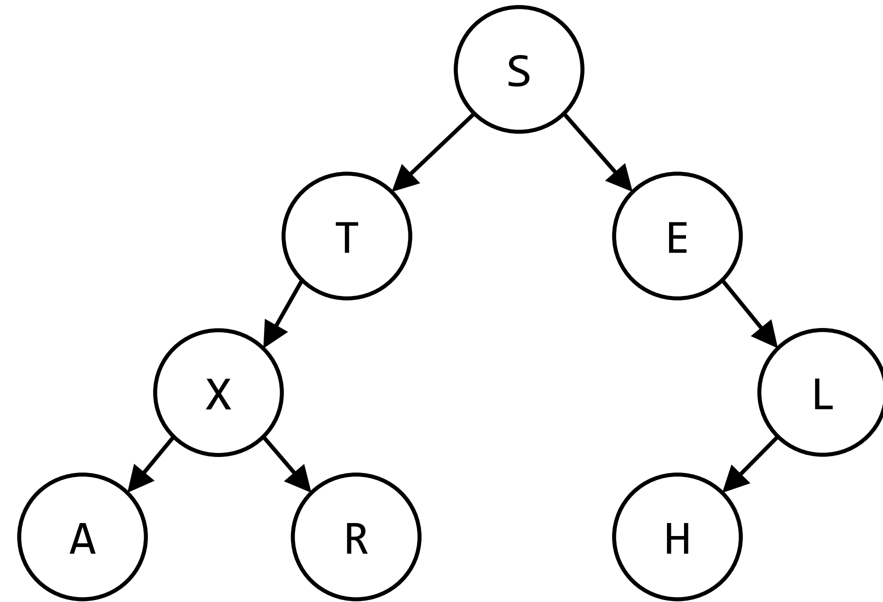
A binary tree is a tree that can have at most **2 children per node**. We assign these children names- **left** and **right**, based on their position.



Activity: Find the Tree Parts

Given the tree shown to the right:

- What is the **root**?
- What are the **children** of node X?
- What is the node X's **parent**?
- What are the **leaves**?



Coding with Trees

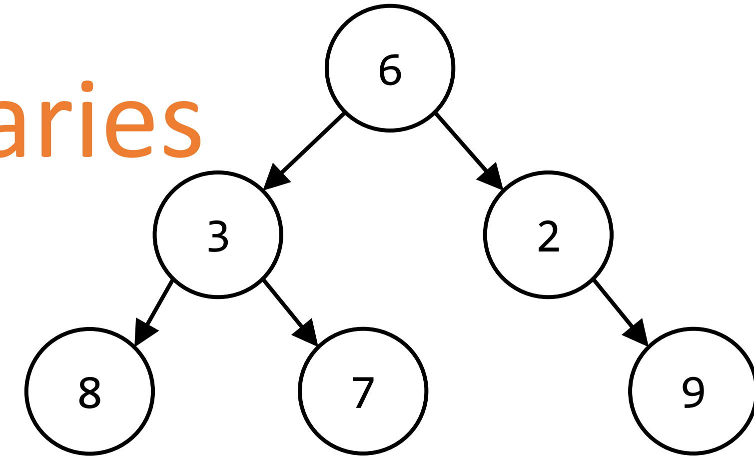
Implementing New Data Structures

Computer science uses a large number of classical data structures. Some of these are implemented directly by Python. Others are not implemented directly; we need to design an implementation ourselves. The way we design the implementation will affect the data structure's efficiency!

Python implements lists and dictionaries, but not trees. We'll design our own trees using **recursively nested dictionaries**.

Sidebar: these trees will be **mutable**; we can change the values in them and add/remove nodes. That's beyond the scope of this class, though.

Python Syntax – Trees as Dictionaries



An **empty tree** is represented as None.

Each **node** of the tree will be a dictionary that has three keys.

- The first key is the string `"contents"`, which maps to the value in the node.
- The second key is the string `"left"`, which either maps to a tree (dictionary) if the node has a left child, or None if there is no left child.
- The third key is the string `"right"`, which either maps to a tree (dictionary) if the node has a right child, or None if there is no right child.

Our example tree is written as a dictionary to the right.

```
t = { "contents" : 6,
      "left"   : { "contents" : 3,
                   "left"   : { "contents" : 8,
                                "left"   : None,
                                "right" : None },
                   "right" : { "contents" : 7,
                                "left"   : None,
                                "right" : None } },
      "right" : { "contents" : 2,
                   "left"   : None,
                   "right" : { "contents" : 9,
                                "left"   : None,
                                "right" : None } } }
```

Simple Example: `getChildren(t)`

Given a tree, how can we get the values of the children of the root node?

Access the `"left"` and `"right"` subtrees directly, then access their `"contents"`, *if they exist*.

Note that we use two separate `if`s, not an `if`-`elif`, because it's possible for both to be `True`.

```
def getChildren(t):
  if t == None:
    return []

  result = []

  left = t["left"]
  if left != None:
    result.append(left["contents"])

  right = t["right"]
  if right != None:
    result.append(right["contents"])

  return result
```

Use Recursion When Coding with Trees

Because this is a recursive data structure, we'll usually need to use recursion to operate on trees.

The **base case** is when we reach an empty tree.

In the **recursive case**, we'll often call the function recursively on the left child and then call again on the right child. Usually we'll then combine those results in some way with the root node's value.

Example: countNodes

How can we count the number of nodes in a tree?

Base case: an empty tree has `0` nodes.

Recursive case: a non-empty tree has the total number of nodes in the two subtrees, plus the current node.

```
def countNodes(t):
  if t == None:
    return 0
  else:
    count = 0
    count += countNodes(t["left"])
    count += countNodes(t["right"])
    return count + 1
```

Example: `countEven(t)`

What if we instead wanted to count all the even numbers in a tree? Now we'll need to use the nodes' **values**.

Base case: an empty tree has no values; return `0`.

Recursive case: determine if the root node's value is even or not, then add in the counts of the two subtrees.

Like the previous recursive problems we solved, we have to combine the **leftover part** with the recursive calls.

```
def countEven(t):
    if t == None:
        return 0
    else:
        if t["contents"] % 2 == 0:
            result = 1
        else:
            result = 0

        result += countEven(t["left"])
        result += countEven(t["right"])
        return result
```

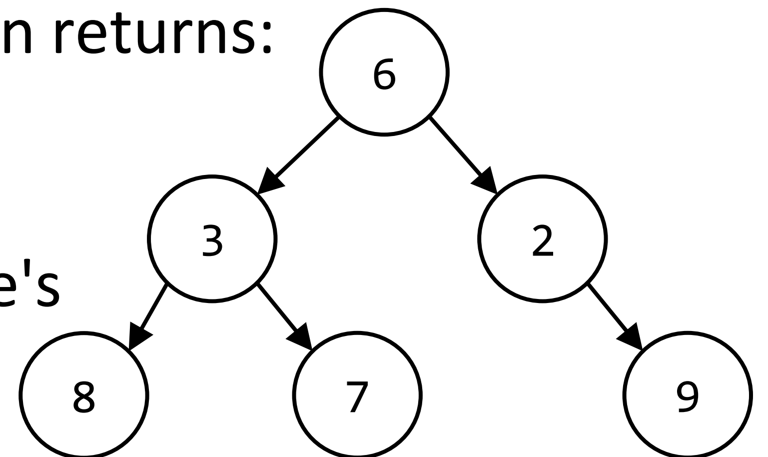
Activity: `listValues`

You do: write the function `listValues(t)`, which takes a tree and returns a list of all the values in the tree. The values can be in any order, but try to put them in left-to-right order if possible.

Hint: consider the **type** of the values you'll return.

Given our example tree (shown below), the function returns: `[8, 3, 7, 6, 2, 9]`.

You can test your code by copying the example tree's implementation on Slide 13.



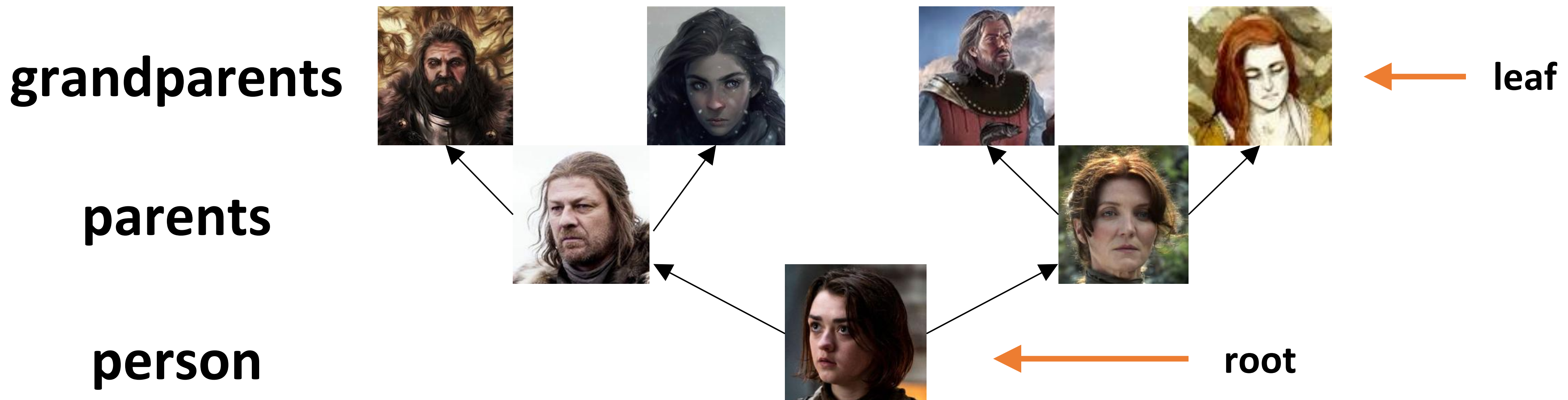
Learning Goals

- Identify core parts of **trees**, including **nodes**, **children**, the **root**, and **leaves**
- Use **binary trees** implemented with dictionaries when reading and writing code

[Optional] Advanced Example: Family Trees

Now let's write a function that takes a genealogical family tree as data.

We have to flip the tree – the person creating the tree is at the root, their parents are the node's children, etc.



[Optional] Advanced Example: getPastGen

We want to find all the child's ancestors from N generations ago. N=1 would be their parents; N=2 would be grandparents; etc.

Note that for this problem, we have an additional base case – when we reach the level we're looking for.

```
def getPastGen(t, n):
  if t == None:
    return [ ]
  elif n == 0:
    return [ t["contents"] ]
  else:
    gen = [ ]
    gen += getPastGen(t["left"], n-1)
    gen += getPastGen(t["right"], n-1)
    return gen
```