Trees

15-110 – Wednesday 10/14

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

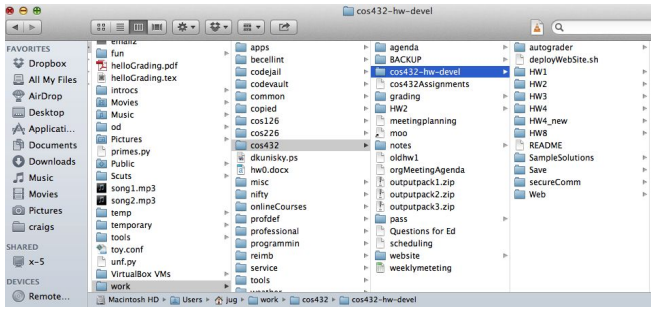
Trees

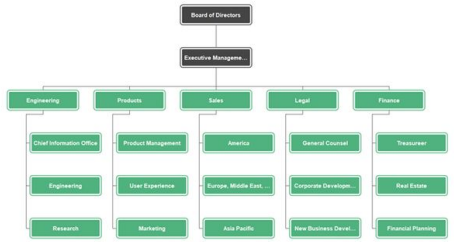
Trees Hold Hierarchical Data

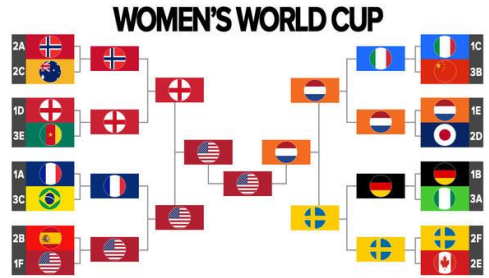
Sometimes we work with data that is **hierarchical** in nature. In this context, 'hierarchical' means that data occurs at different **levels** and is connected in some way.

Hierarchical data shows up in many different contexts.

- **File systems** in computers – each folder is a rank about 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





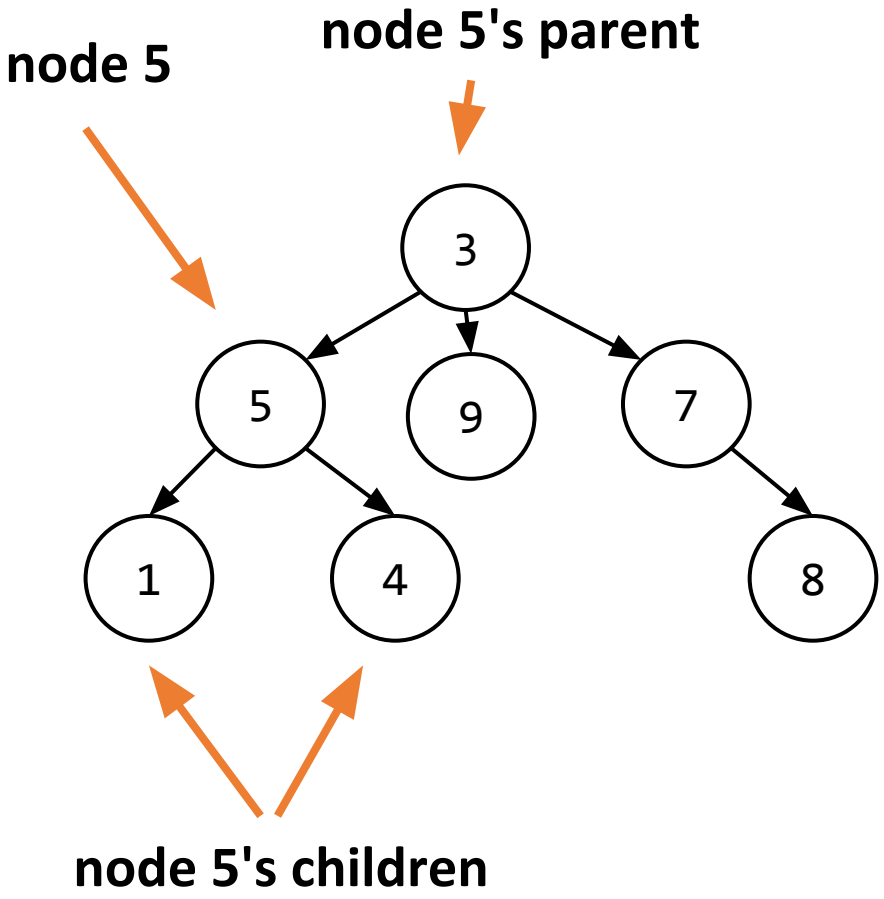


Trees are Hierarchical

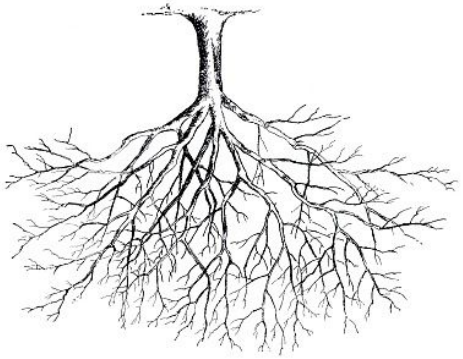
A **tree** is a hierarchical data structure composed of **nodes** (circles in the example shown to the right).

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

The node 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, but it can have any number of children.



Trees are Upside-down

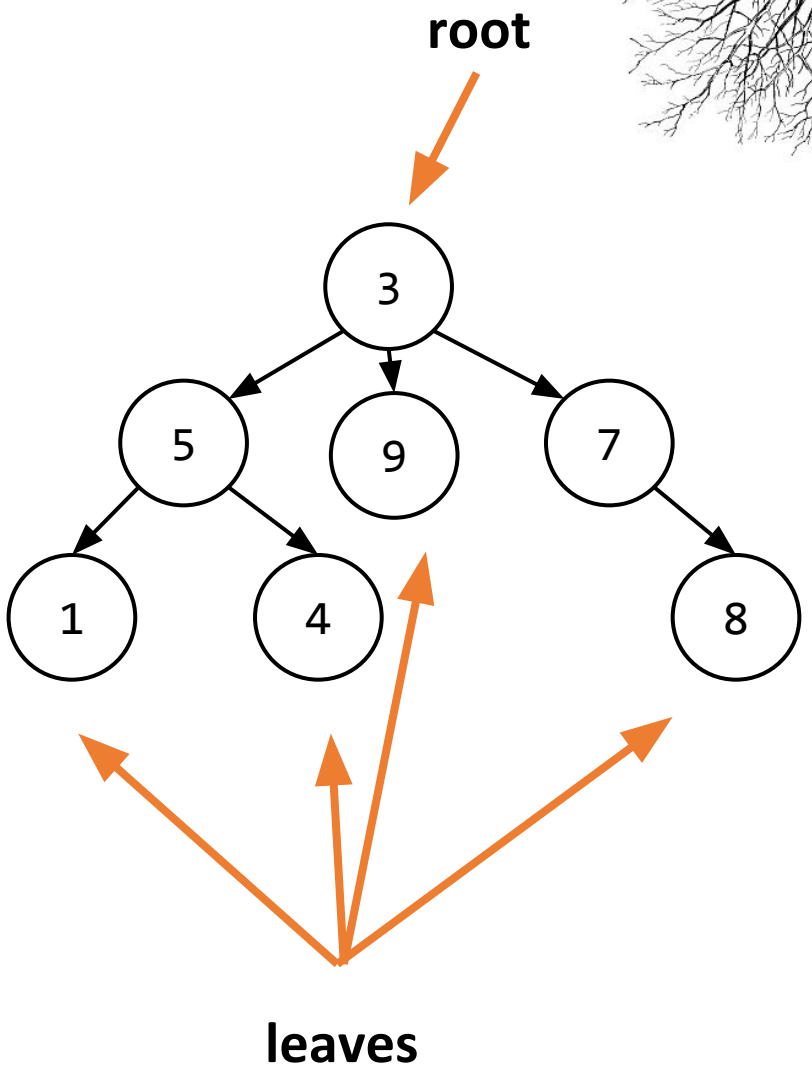


Unlike real trees, trees in computer science grow downward!

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

On the other hand, 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**.



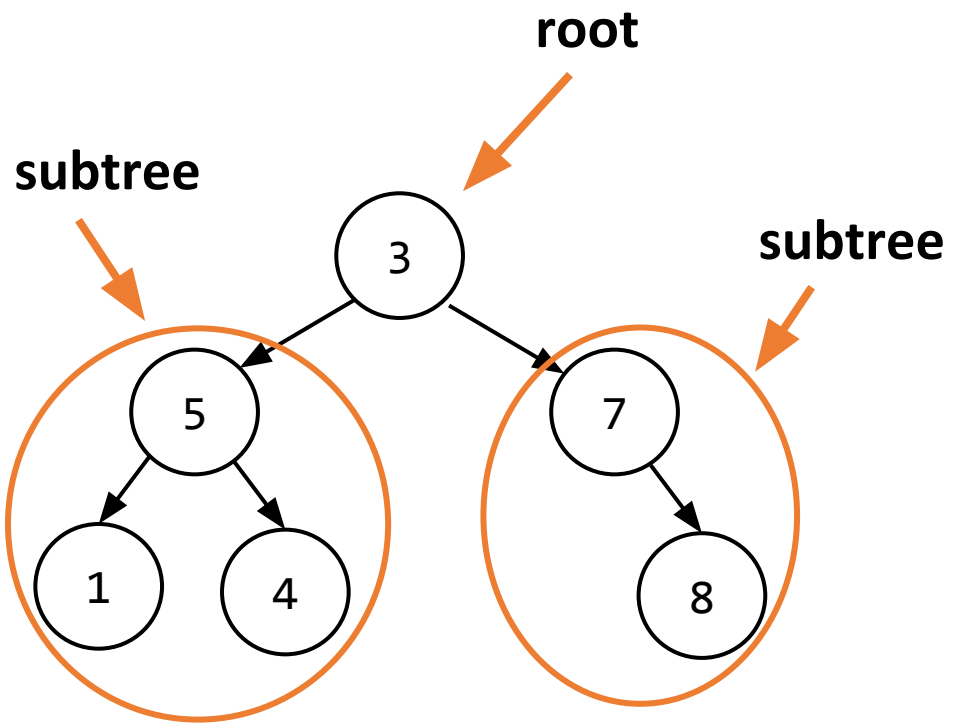
Trees are Recursive

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 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 subtrees as children.

Our **base case** can be a leaf (or even an empty tree).

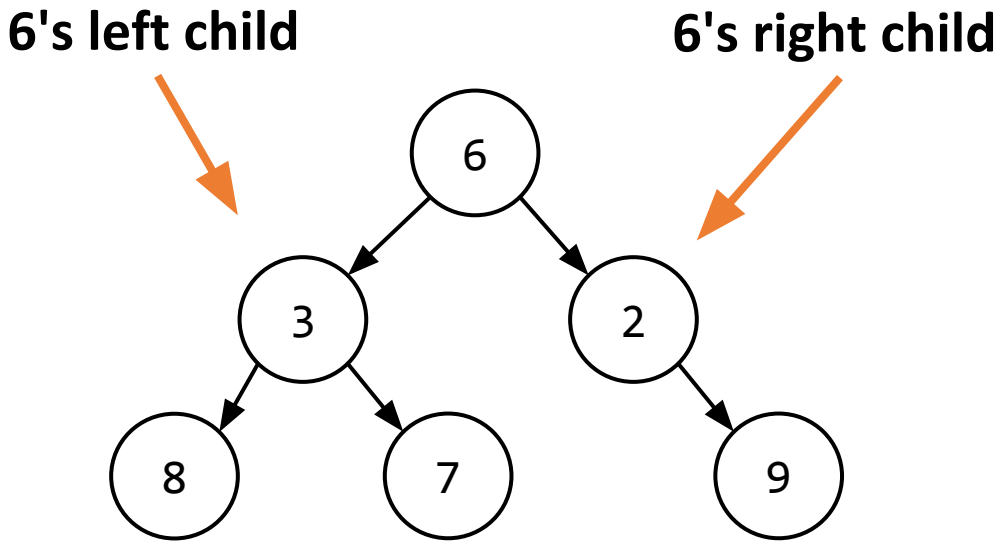
The **recursive case** is a node and its children, which are also trees.



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**.

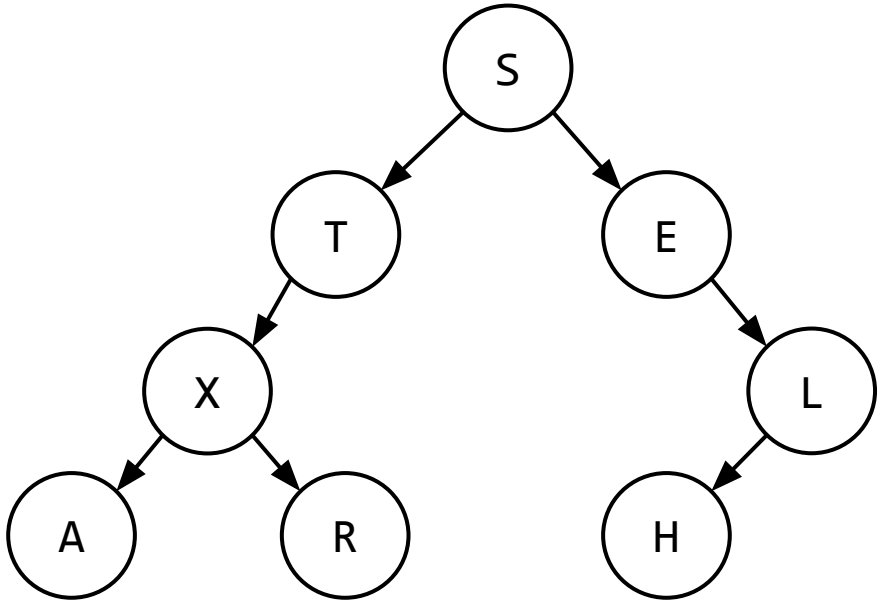
A binary tree is a tree that can have at most **2 children per node**. We assign these children names- **left** and **right**, based on their position.



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

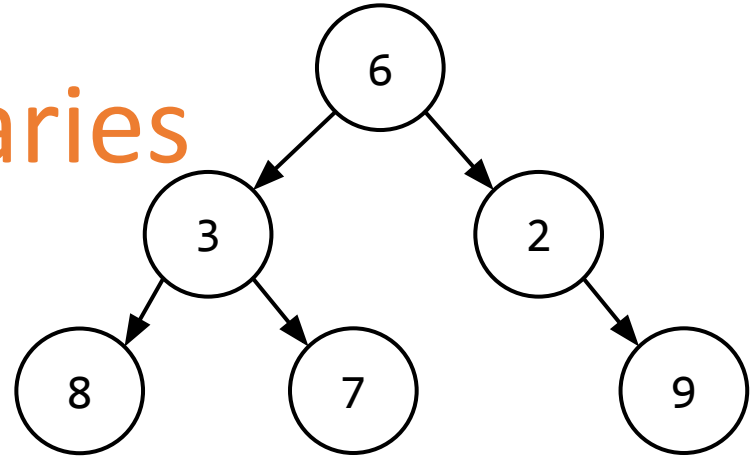
Implementing New Data Structures

Computer science uses a large number of classical data structures. Some of these (like lists and dictionaries) are implemented directly by Python. Others are not implemented directly; we need to design an implementation ourselves.

Python does not implement trees directly. We'll implement trees using **recursively nested dictionaries**.

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

Python Syntax – Trees as Dictionaries



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

- The first key is the string `"value"`, which maps to the value in the node.
- The second key is the string `"left"`, which either maps to a node (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 node (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 = { "value" : 6,
      "left"  : { "value" : 3,
                  "left"  : { "value" : 8,
                              "left"  : None,
                              "right" : None },
                  "right" : { "value" : 7,
                              "left"  : None,
                              "right" : None } },
    "right" : { "value" : 2,
                "left"  : None,
                "right" : { "value" : 9,
                            "left"  : None,
                            "right" : None } } }
```

Use Recursion When Coding with Trees

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

The **base case** is when the node is a leaf and we need to do something with its value.

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

Alternative approach: Make the base case when the tree is None (an empty tree), and always recurse on both left and right children in the recursive case. It can be more confusing to think about, but is often simpler to program.

Example: sumTree

Let's write a program that takes a tree of integers and sums all of the integers together.

The **base case**: don't make recursive calls on left/right branches (there are none); just return the leaf's value.

The **recursive case**: compute the sums of the left and right children (if they exist) and add them to this node's value.

```
def sumTree(t):
  result = t["value"]
  if t["left"] != None:
    result += sumTree(t["left"])
  if t["right"] != None:
    result += sumTree(t["right"])
  return result
```

Example: sumTree – Different Base Case

Alternatively, we could solve this by checking a different base case: whether the node is an empty tree (if the current node is `None`).

An empty tree has a sum of `0`; a non-empty tree has a sum based on its node and the sums of its left and right subtrees.

The difference here is that there are always recursive calls to both children, even if they might be `None`.

```
def sumTree(t):
    if t == None:
        return 0
    result = t["value"]
    result += sumTree(t["left"])
    result += sumTree(t["right"])
    return result
```

Example: `listNodes(t)`

What if we instead wanted to generate a **list** of all the nodes in the tree?

Base case: a list with a single node: return the leaf's value, **in a list**. Types matter!

Recursive case: combine the lists produced by the left and right children, if they exist, with a list containing the node's value.

Note that our code is mostly the same, except that the result value changes. **The type of value that you return matters a lot!**

```
def listNodes(t):
  result = [ t["value"] ]
  if t["left"] != None:
    result += listNodes(t["left"])
  if t["right"] != None:
    result += listNodes(t["right"])
  return result
```

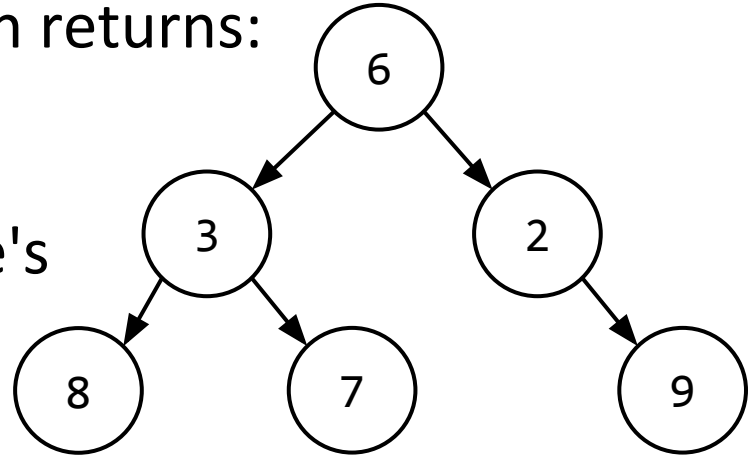
Activity: `listValuesInOrder`

You do: write the function `listValuesInOrder(t)`, which takes a tree and returns a list of all the values in the tree, **in left-to-right order**.

Hint: this is *almost* the same as `listValues`, but you need to combine the results in a different way. Get the list of values for the left-subtree, then the current value, then the list for the right-subtree.

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

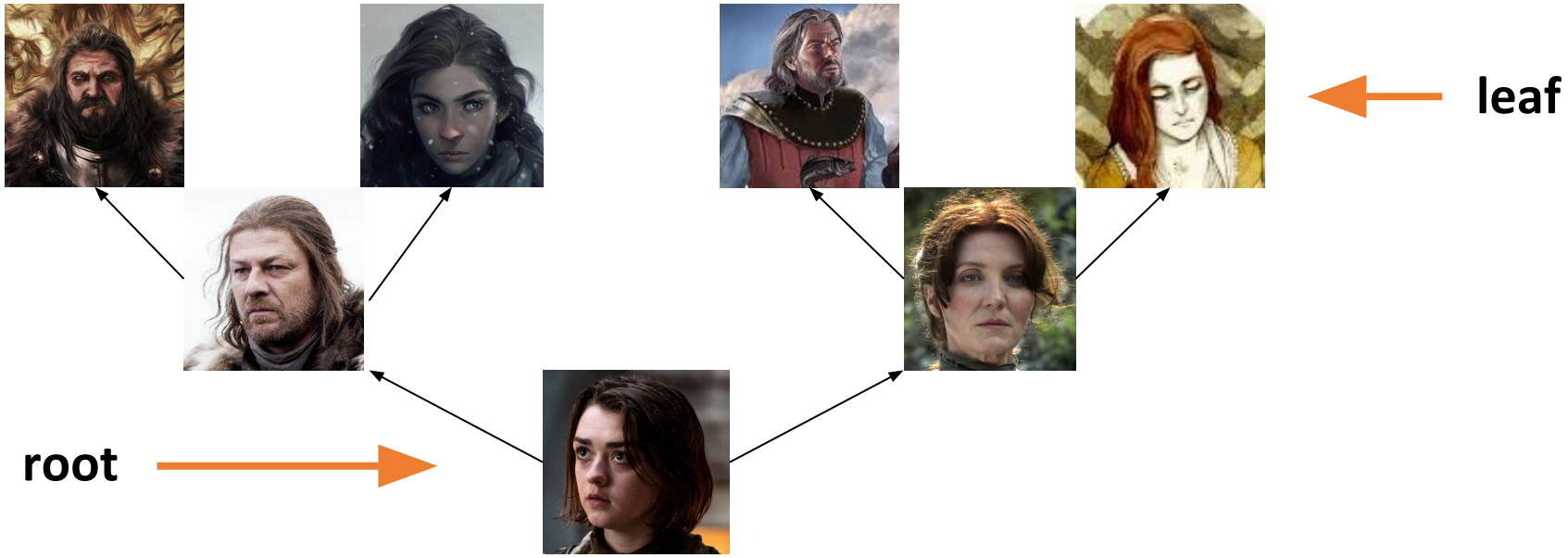
You can test your code by copying the example tree's implementation on Slide 12.



Advanced Example: Genealogical Trees

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

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



Advanced Example: getPastGen

Let's write a function that finds 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, our base case is not a leaf- it's when we reach the generation we're looking for.

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

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

- **Feedback:** https://bit.ly/110-feedback