Trees

15-110 – Monday 03/15

Announcements

- Hw3 was due today
  - Except Hw3 - #7, which is due **next Monday.** Download the special starter file and submit to "Hw3 – Just #7"

- Check4 & Hw4 out
  - These tend to be difficult for students. Start early and use your resources (collaboration, office hours, piazza, small group sessions)!

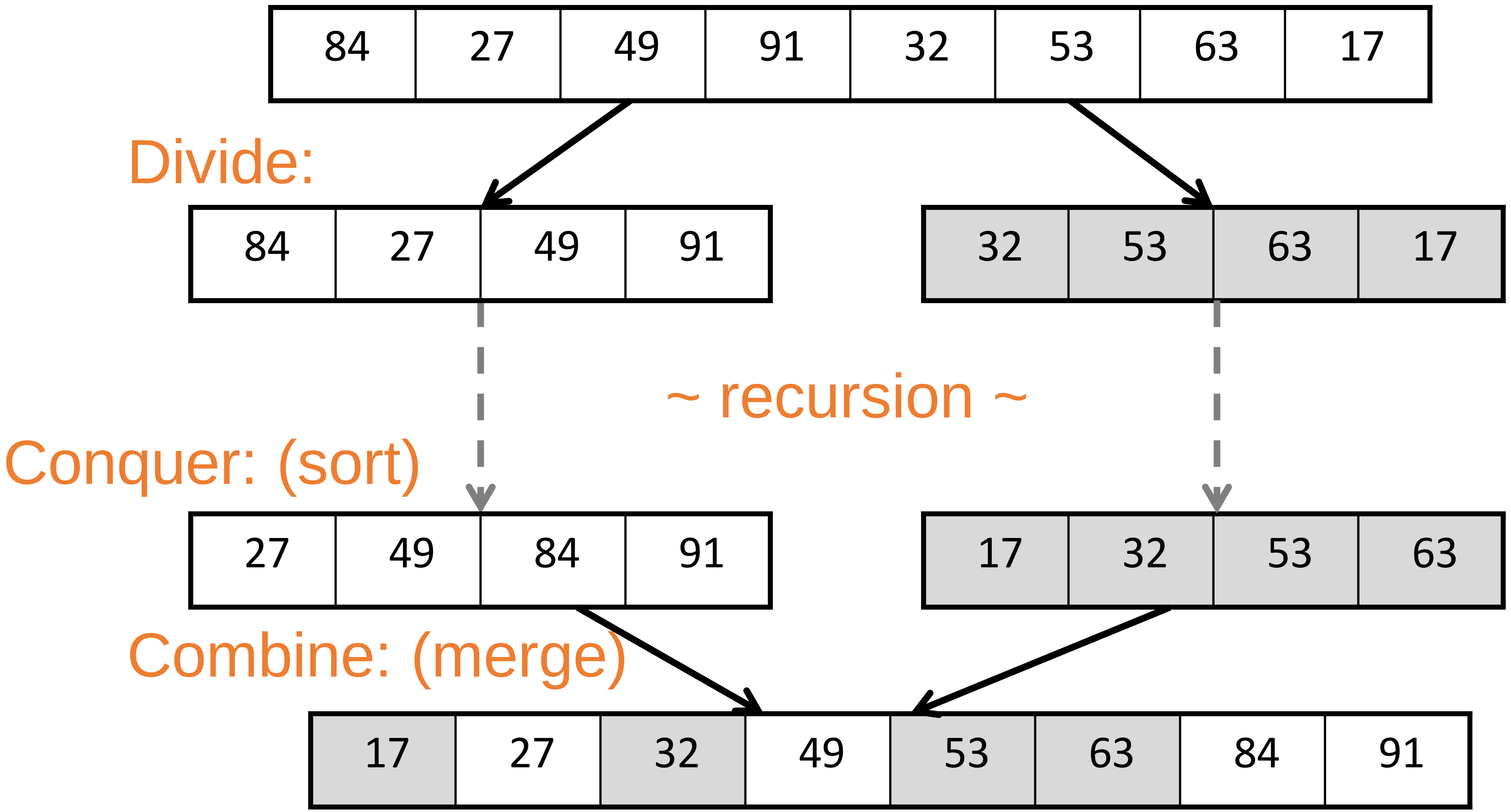
Learning Goals

- Identify core parts of **trees**, including **nodes**, **children**, the **root**, and **leaves**

- Use **binary trees** implemented with dictionaries when reading and writing code

Merge Sort: Fast Review

Merge Sort Process: Split, then Merge



Merge Sort Code & Efficiency Computation

```
def mergeSort(lst):
    if len(lst) < 2:
        return lst
    mid = len(lst) // 2
    front = lst[:mid]
    back = lst[mid:]
    front = mergeSort(front)
    back = mergeSort(back)
    return merge(front, back)

lst = [2, 4, 1, 5, 10, 8, 3, 6, 7, 9]
sortedLst = mergeSort(lst)
print(sortedLst)
```

Copy

```
def merge(half1, half2):
    result = [ ]
    i = 0
    j = 0
    while i < len(half1) and j < len(half2):
        if half1[i] < half2[j]:
            result.append(half1[i])
            i = i + 1
        else:
            result.append(half2[j])
            j = j + 1
    result = result + half1[i:] + half2[j:]
    return result
```

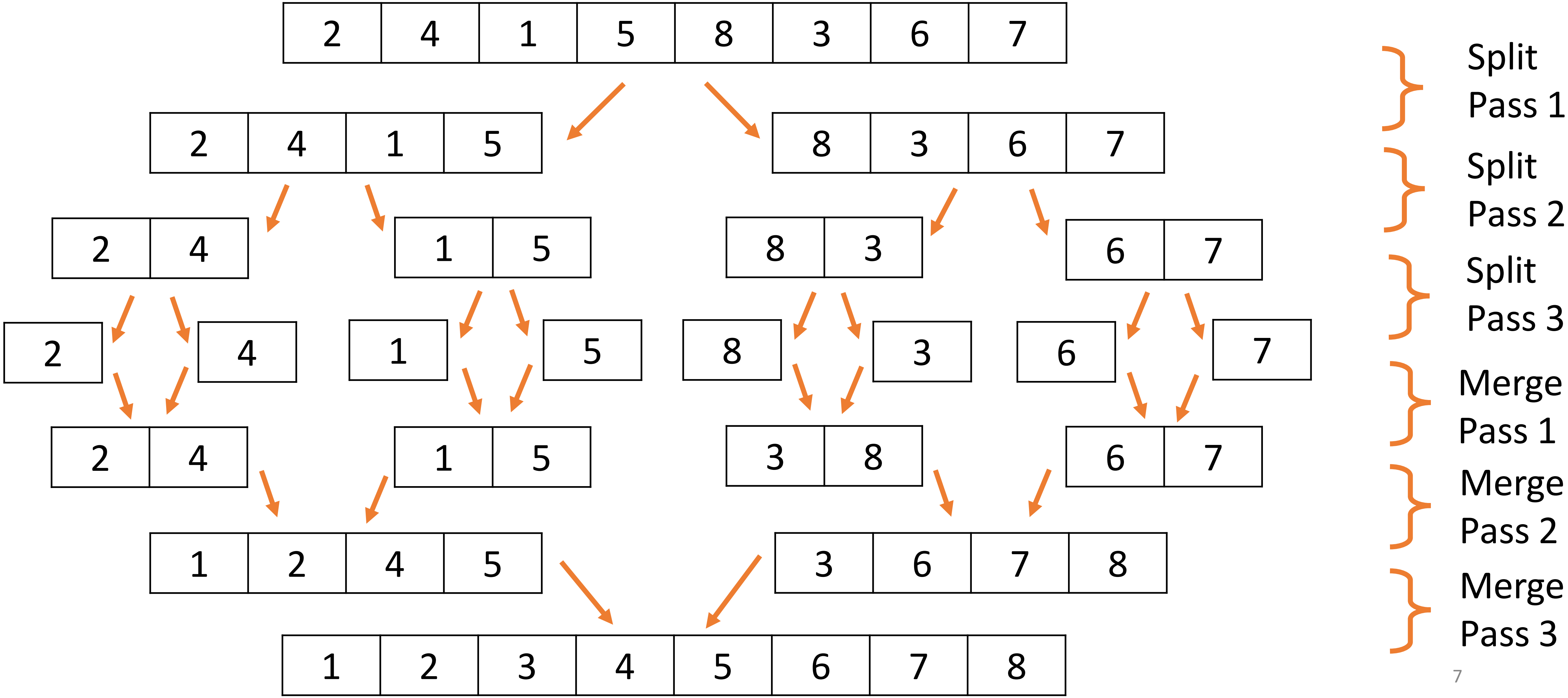
Comparison

Copy

Copy

Merge Sort Call Breakdown

n copies in each split-pass
n copies + ~n comparisons in each merge-pass



Merge Sort Efficiency

Every time a split-pass occurs, we cut the number of elements being sorted in **half**. The number of split-passes is the **number of times we can divide the list in half**, or $\log_2 n$. The number of merge-passes is the same.

Overall work: n log n + 2 * (n log n) = 3 * (n log n) = **O(n log n)**

O (n log n) may not seem a lot better than $O(n^2)$, but the difference shows when you get up to large datasets!

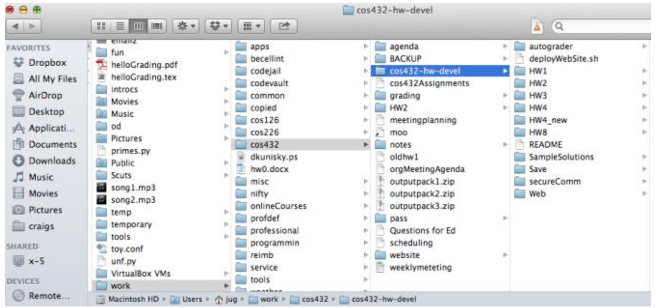
Trees

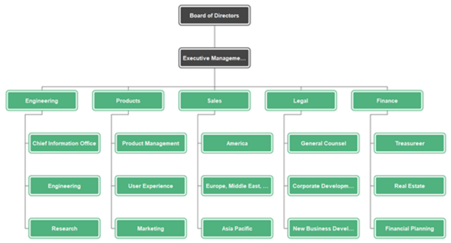
Trees Hold Hierarchical Data

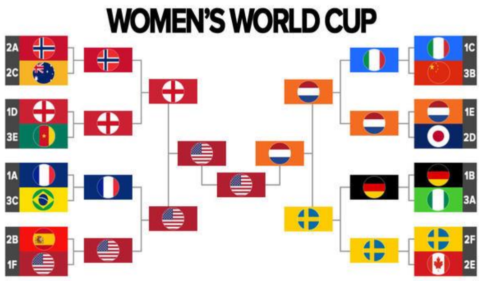
Sometimes we work with data that is **hierarchical** in nature. In this context, 'hierarchical' means that data occurs at different **levels** and is connected in some way.

Hierarchical data shows up in many different contexts.

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





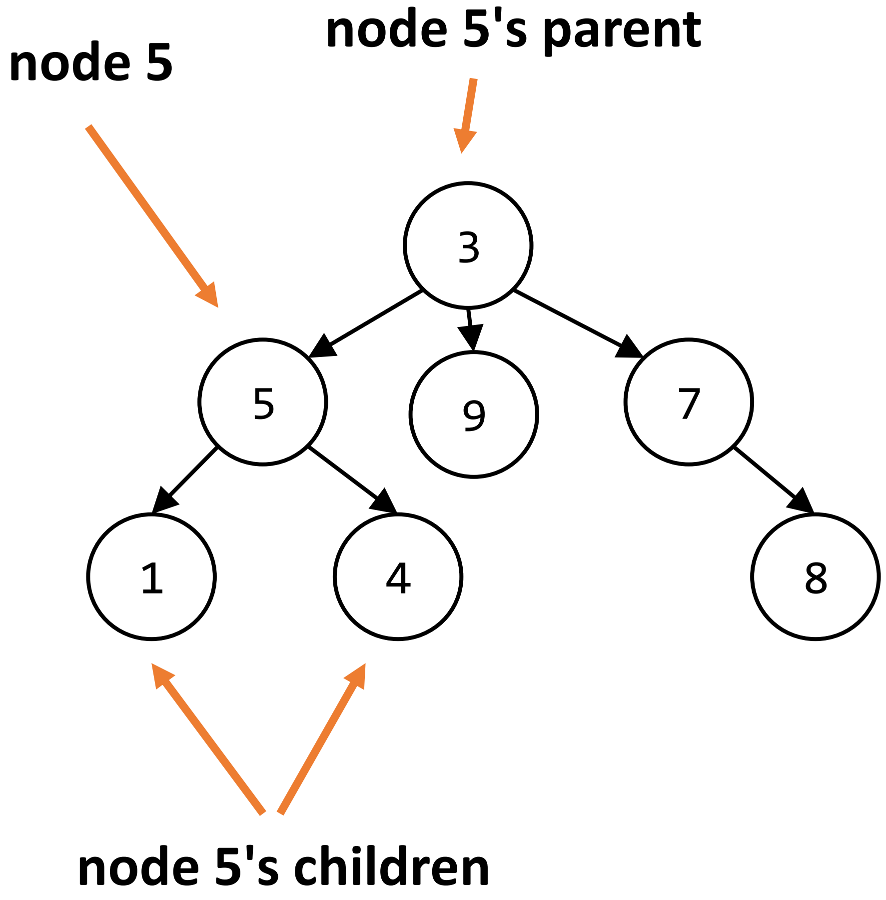


Trees are Hierarchical

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 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.



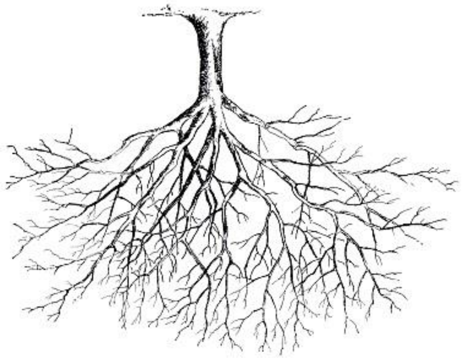
Trees are Upside-down

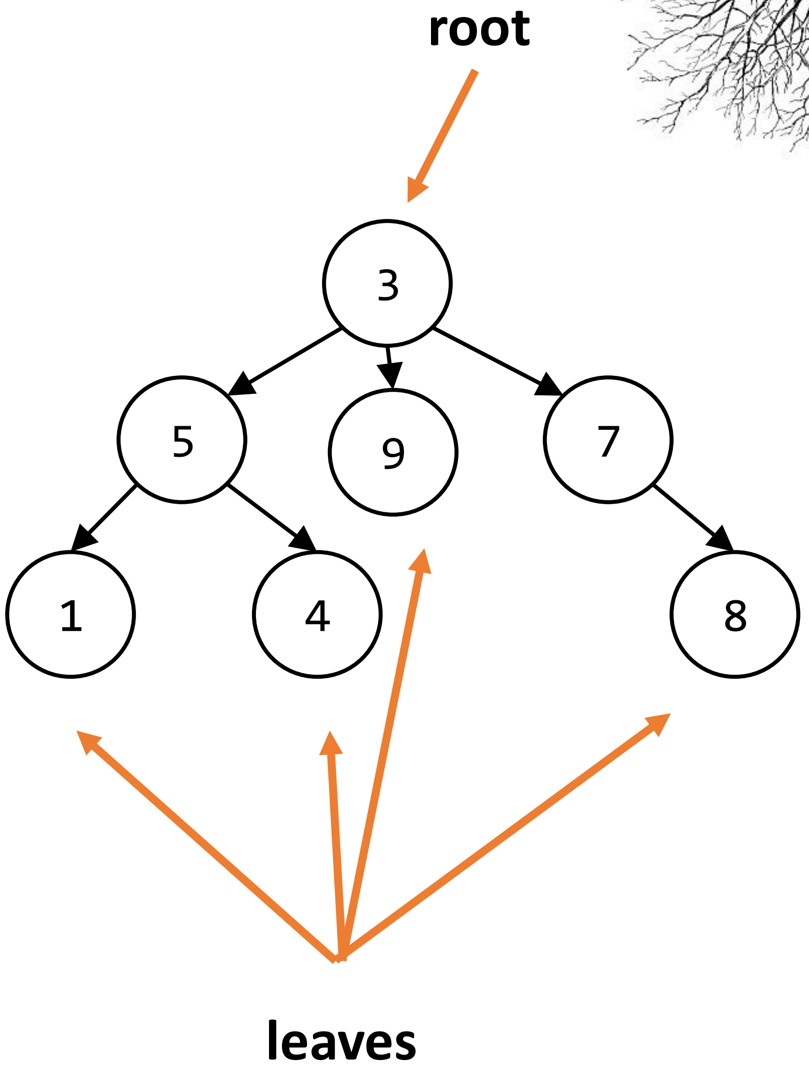
Unlike real trees, trees in computer science grow downward!

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

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**.





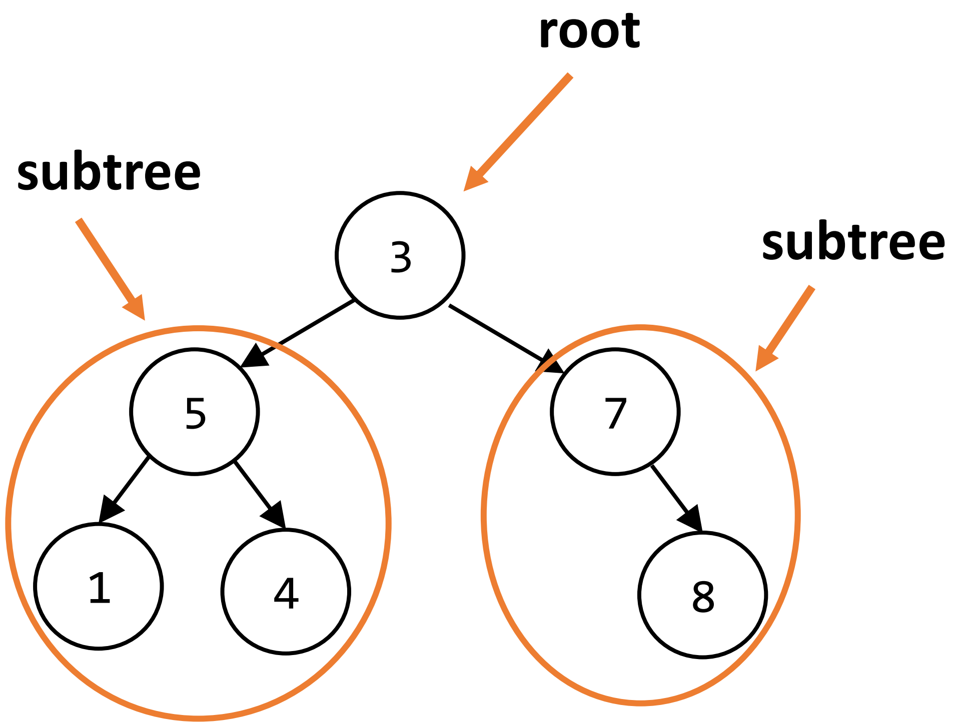
Trees are Recursive

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).

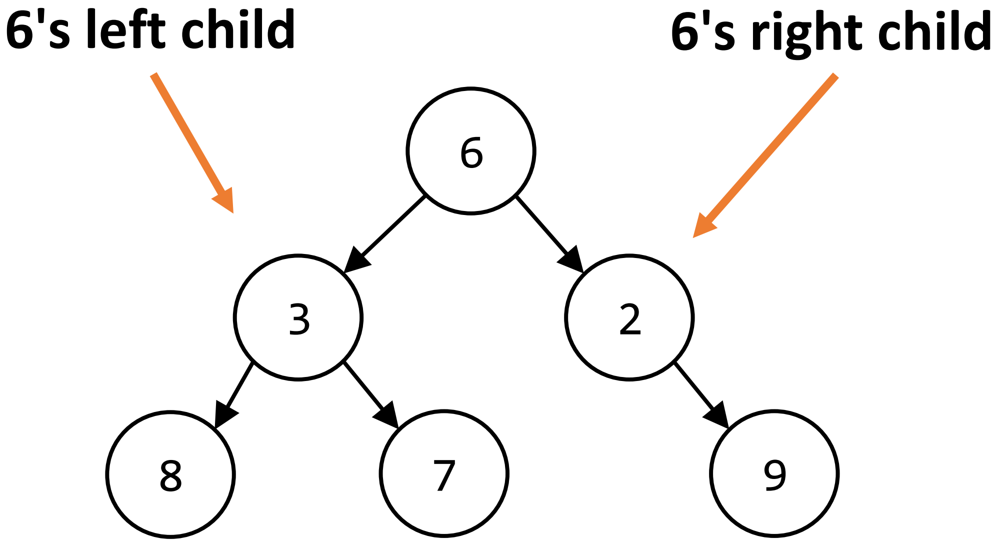
The **recursive case** makes the problem smaller by repeating on the children, which are also trees.



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**.

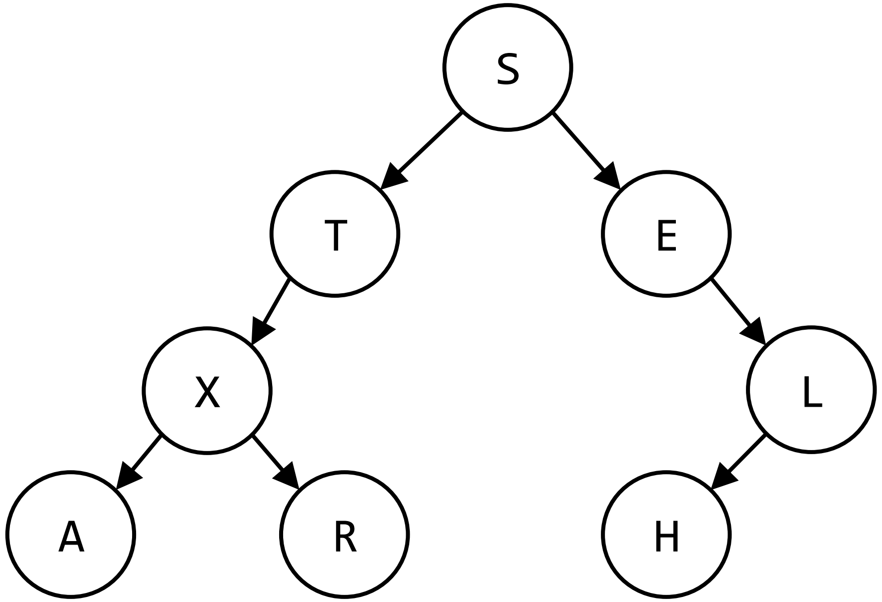
A binary tree is a tree that can have at most **2 children per node**. We assign these children names- **left** and **right**, based on their position.



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

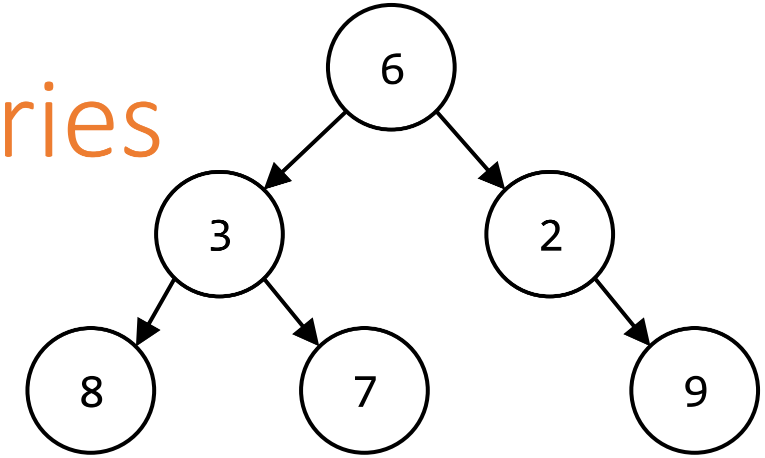
Implementing New Data Structures

Computer science uses a large number of classical data structures. Some of these (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 nodes. 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 `"contents"`, 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.

```
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 } } }
```

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

Simple Example: `getChildren(t)`

Given a tree, how can we get 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):
  result = []
  if t["left"] != None:
    leftT = t["left"]
    result.append(leftT["contents"])
  if t["right"] != None:
    rightT = t["right"]
    result.append(rightT["contents"])
  return result
```

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 current node is a leaf and we need to do something with its value.

In the **recursive case**, we'll call the function recursively on the left and then call again on the right child, if both 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. This can be more confusing to think about but is often simpler to program.

Example: `countNodes`

Let's write a program that takes a tree of values and counts the number of nodes in the tree.

The **base case**: return 1 (a single node).

The **recursive case**: add the counts of the left and right subtrees together if they exist, then add 1 more for the current node.

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

Example: countNodes – 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 0 nodes; a non-empty tree has a number of nodes based on its two subtrees, plus the current node.

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

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

Example: `sumNodes(t)`

What if we instead wanted to add all the nodes in the tree? (Let's assume it's a tree of integers). Now we'll need to use the nodes' **values**.

Base case: directly return the value of the only node (the leaf).

Recursive case: combine the sums of the two subtrees (if they exist) with the current node's value.

Our code structure is very similar to `countNodes`, but now we're using `t["contents"]`.

```
def sumNodes(t):
    if t["left"] == None and \
       t["right"] == None:
        return t["contents"]
    else:
        result = 0
        if t["left"] != None:
            result += sumNodes(t["left"])
        if t["right"] != None:
            result += sumNodes(t["right"])
        return result + t["contents"]
```

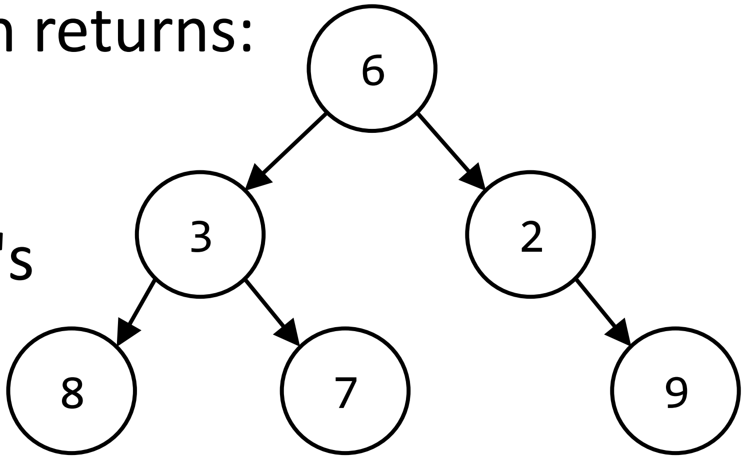
Activity: `listValues`

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: this is *almost* the same structure as `sumNodes`, but you need to consider the **type** of the values you'll return.

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

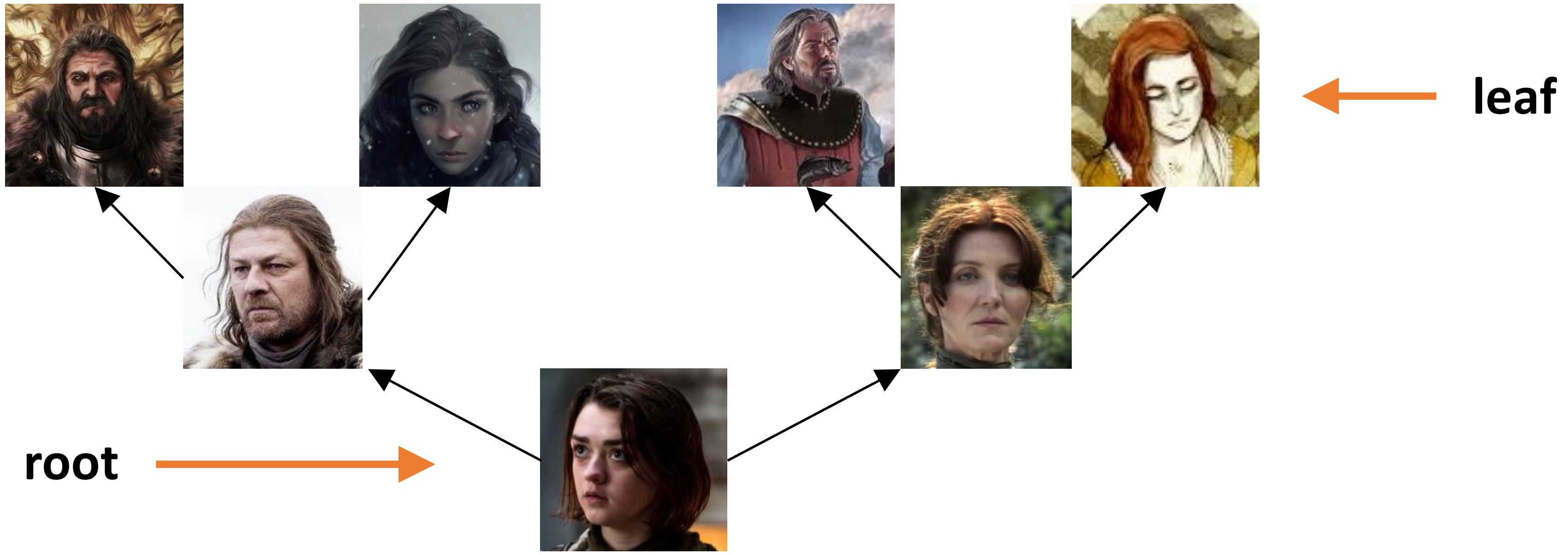
You can test your code by copying the example tree's implementation on Slide 12.



Advanced Example: Family Trees (if time)

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["contents"] ]
    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: http://bit.ly/110-s21-feedback**