Trees

15-110 – Wednesday 10/09

Quizlet 4

Announcement

- Exam1 Solution session scheduled for **Friday 10/11 5pm** on Zoom

Learning Goals

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

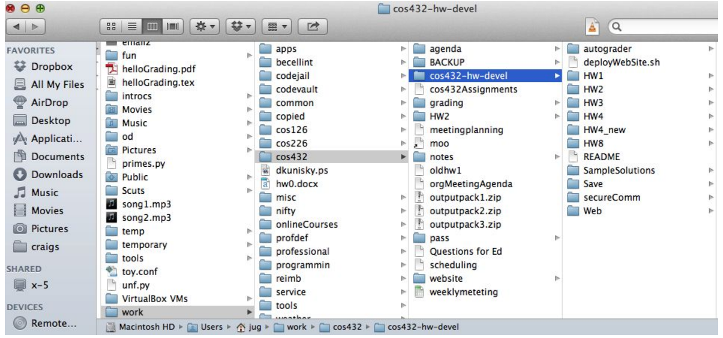
- Use **binary trees** implemented with dictionaries when reading and writing code

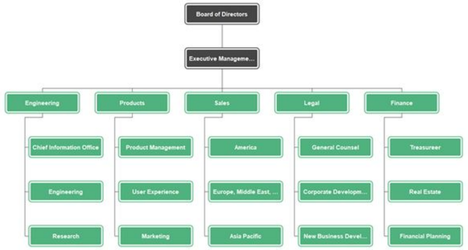
Trees

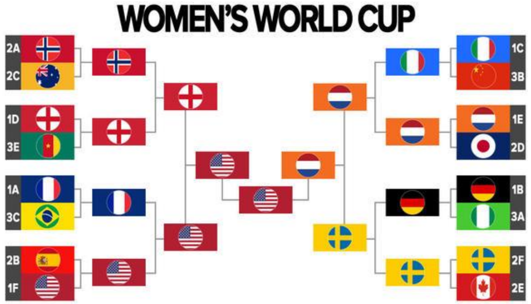
Sometimes we work with data that is **hierarchical** in nature: data that occurs at **different levels that are connected** in some way.

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





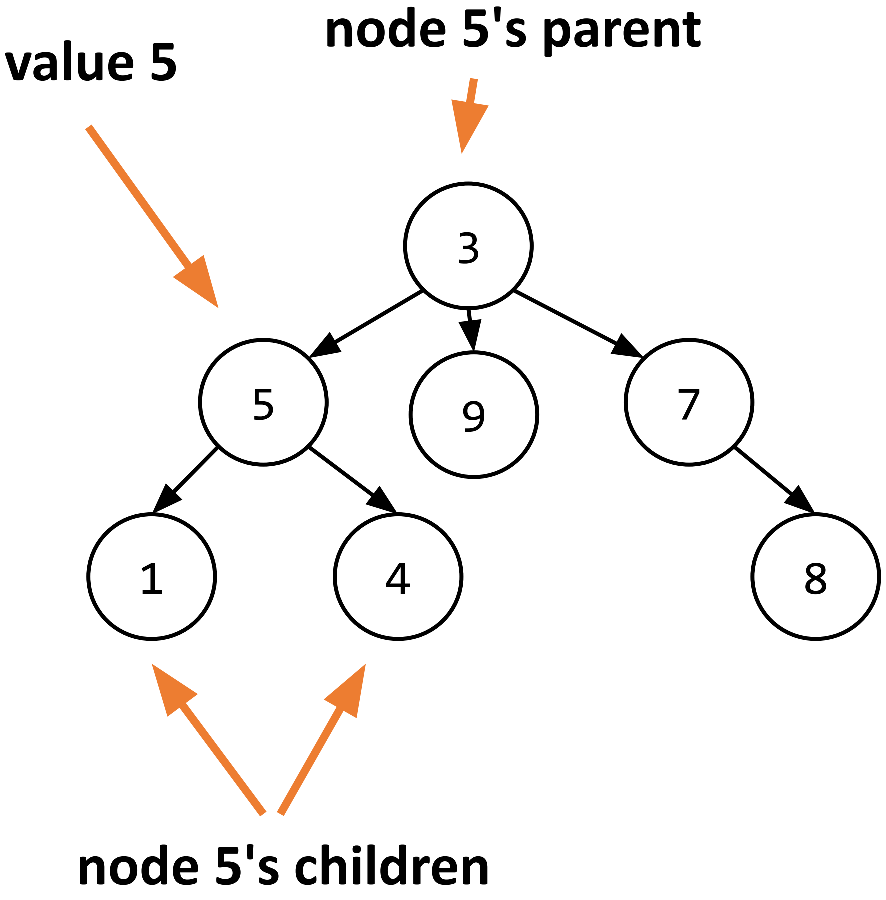


A **tree** is a **hierarchical data structure** composed of **nodes**.

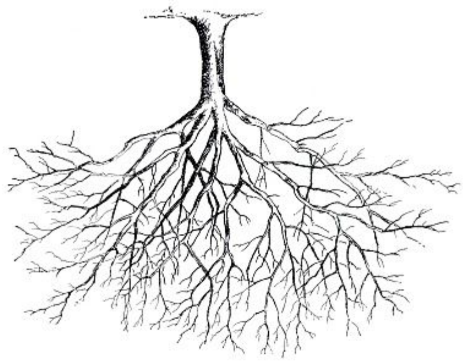
Nodes (circles) have **values** (its data).

A node's **parent** is the node one level above it.

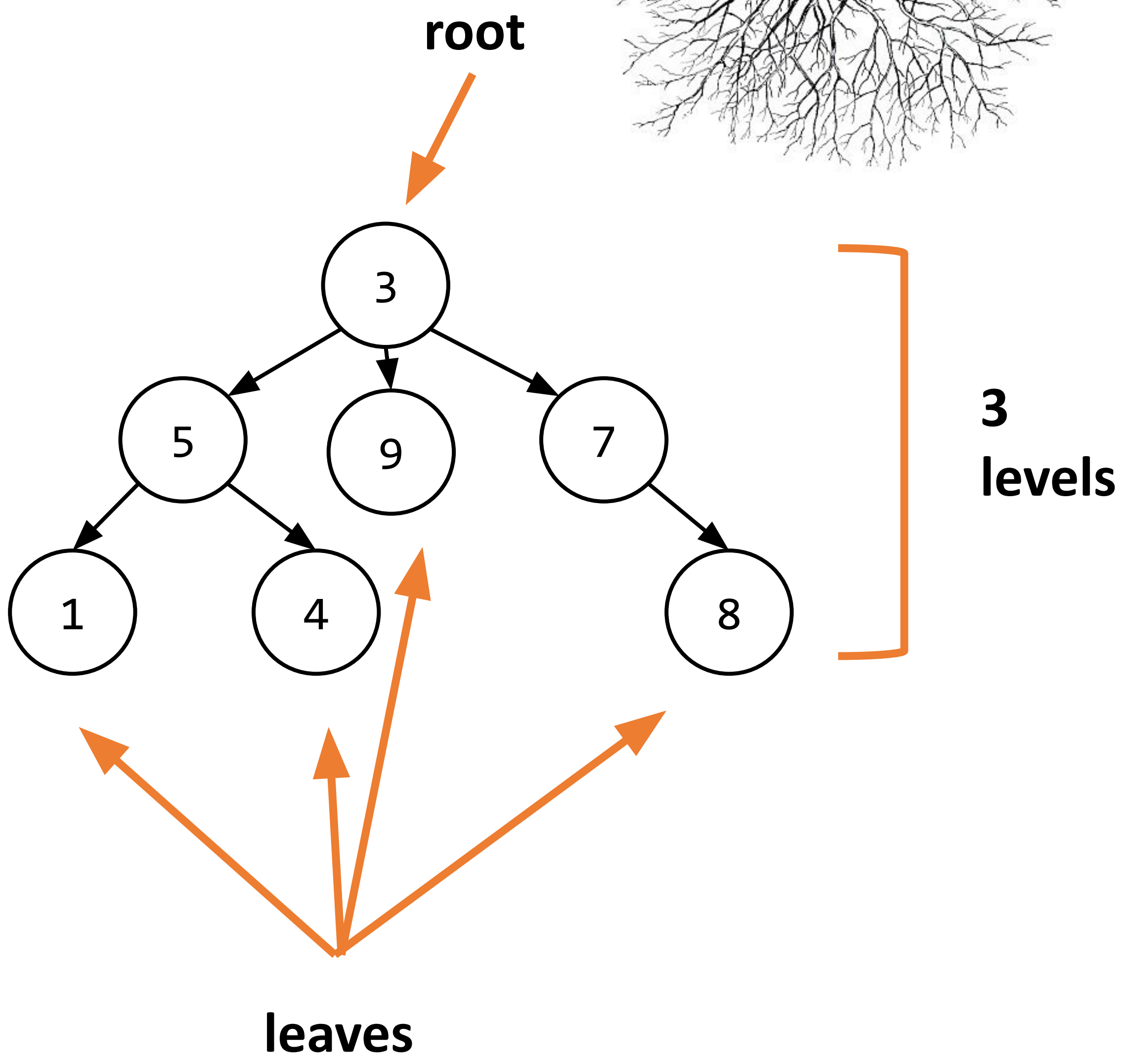
A node's **children** are the nodes one level below it.



Unlike real trees, trees in computer science grow upside down with the **root at the top**.



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



The number of **levels** of a tree is unlimited.

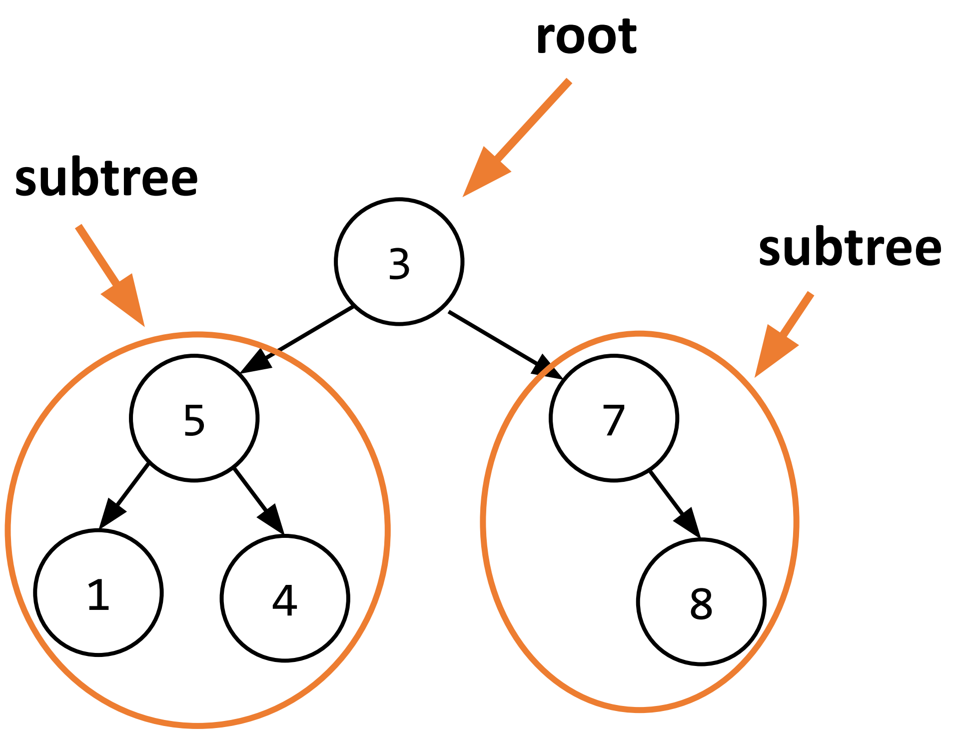
Nodes that have no children are called **leaves**.

A tree is a **naturally recursive structure**.

Trees are composed of subtrees. Each node's children are also trees.

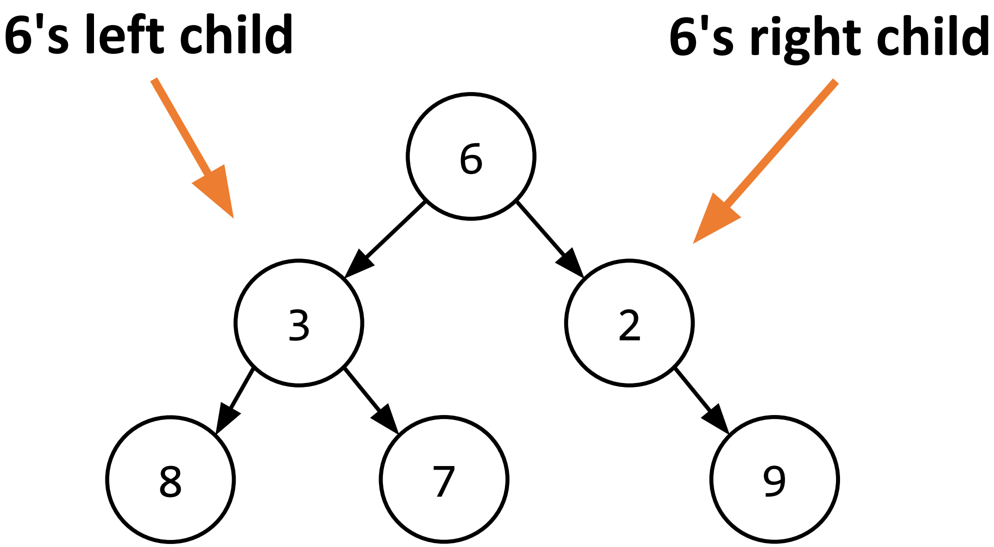
Our **base case** can be an empty tree (or sometimes a leaf).

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



There are special trees we care about in this class: **binary trees**.

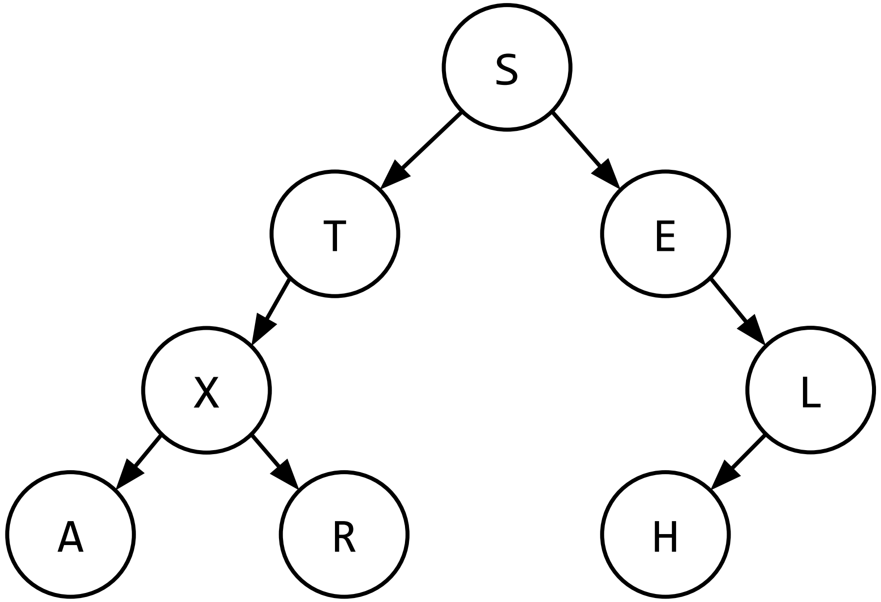
A binary tree is a tree that can have **at most 2 children per node**. We assign these children names- **left** and **right**, based on their position.



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**?



Problem Solving with Trees

We have described trees as abstract data structure, but how do we actually **implement it in Python**?

For many data structures, it is possible to implement them in different ways and that implementation **impacts efficiency of operations** on that data structure.

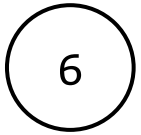
Python implements lists and dictionaries for us, but not trees.

We will implement trees using **recursively nested dictionaries**.

Note: While these trees will be mutable because dictionaries are mutable, in the context of this class we will not be mutating trees.

We implement each **node as a dictionary with 3 keys**: `"contents"`, `"left"`, and `"right"`.

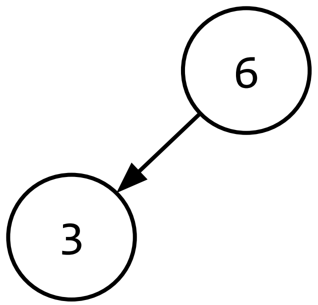
- `"contents"` maps to the **value in the node**
- `"left"` maps to a node (dictionary) if the node has a **left child**, or `None` if there is no left child.
- `"right"` maps to a node (dictionary) if the node has a **right child**, or `None` if there is no right child.




```
t = { "contents" : 6,
      "left"  : None
      "right": None
    }
```

We implement each **node as a dictionary with 3 keys**: `"contents"`, `"left"`, and `"right"`.

- `"contents"` maps to the **value in the node**
- `"left"` maps to a node (dictionary) if the node has a **left child**, or `None` if there is no left child.
- `"right"` maps to a node (dictionary) if the node has a **right child**, or `None` if there is no right child.

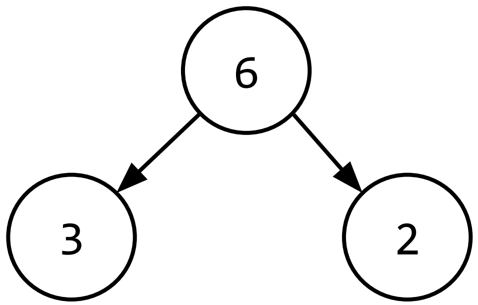



```
t = { "contents" : 6,
      "left" : { "contents" : 3,
                 "left" : None ,
                 "right": None },
      "right": None
    }
```

We implement each **node as a dictionary with 3 keys**: `"contents"`, `"left"`, and `"right"`.

- `"contents"` maps to the **value in the node**
- `"left"` maps to a node (dictionary) if the node has a **left child**, or `None` if there is no left child.
- `"right"` maps to a node (dictionary) if the node has a **right child**, or `None` if there is no right child.

Note: The empty tree is `None`.




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

Activity: Draw the tree

You do: What is the tree that corresponds to this dictionary?

```
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 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
```

Because this is a recursive data structure, we **will usually need to use recursion** to do operations on trees.

Base case: When the tree is empty. (`t == None`).

Recursive case: Call the function recursively on the left child and then call again on the right child. Combine those results in some way with the node's value.

Alternative approach: Base case is when the node is a leaf. Call the function recursively on the left child and the right child, if they are not `None`, but this code is generally more complex.

Example: `countNodes`

Algorithm: Count the number of nodes in the tree.

Recursive Case: Count is 1 for the root node. Add the number of nodes in the left subtree. Add the number of nodes in the right subtree.

Base Case: Empty tree has 0 nodes.

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

Example: `sumNodes(t)`

Algorithm: Get the sum of all the node values.

Recursive Case: Add the value of the root node. Add the sum of the nodes in the left subtree. Add the sum of the nodes in the right subtree.

Base Case: Empty tree has a sum of 0.

```
def sumNodes(t):
  if t == None:
    return 0
  else:
    # add contents of root node
    total = t["contents"]
    # left and right subtrees
    total += sumNodes(t["left"])
    total += sumNodes(t["right"])
    return total
```

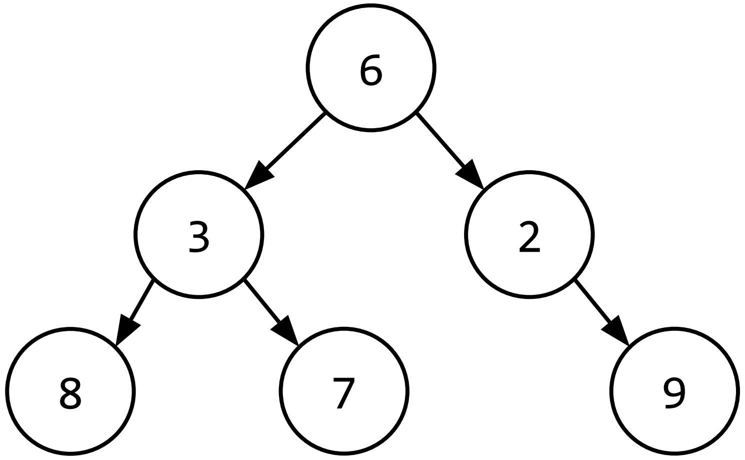
Activity: `listValues`

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 18.



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