# Trees

15-110 – Wednesday 3/04

# Learning Goals

- Understand how a **hierarchical** data structure uses recursion to store data
- Define core concepts of **trees**, including **nodes**, **children**, the **root**, and **leaves**
- Identify the difference between **trees**, **binary trees**, and **binary search trees**
- Search for values in **binary search trees** using **binary search**
- Use **trees** and **binary trees** implemented with dictionaries when reading and writing code

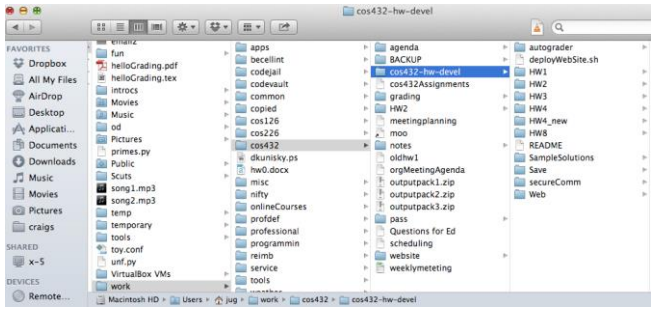
# Trees

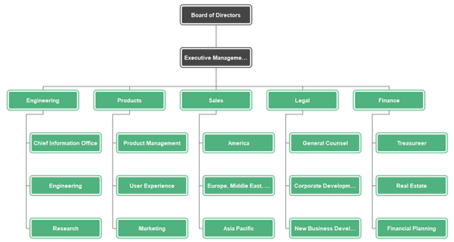
# Trees Hold Hierarchical Data

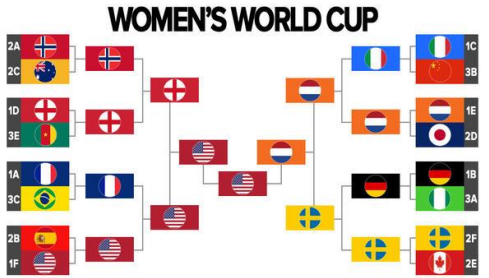
Sometimes we work with data that is **hierarchical** in nature. In this context, 'hierarchical' means that data occurs at different **levels** and is connected in some way.

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





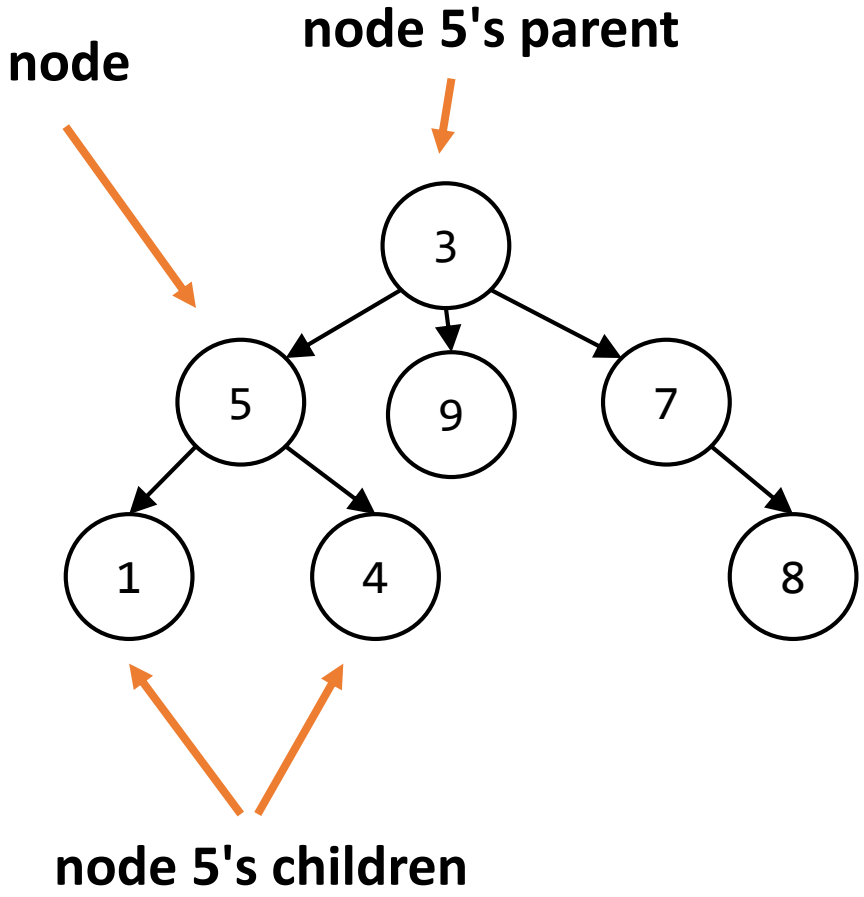


# Trees are Hierarchical

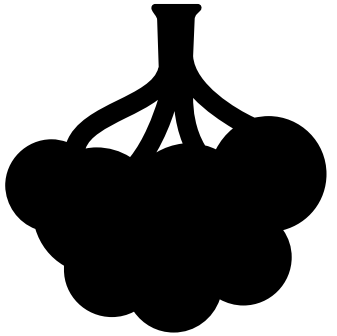
A **tree** is a hierarchical data structure composed of **nodes** (circles) in the example shown to the right.

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 can have 0 or more children.



# Trees are Upside-down

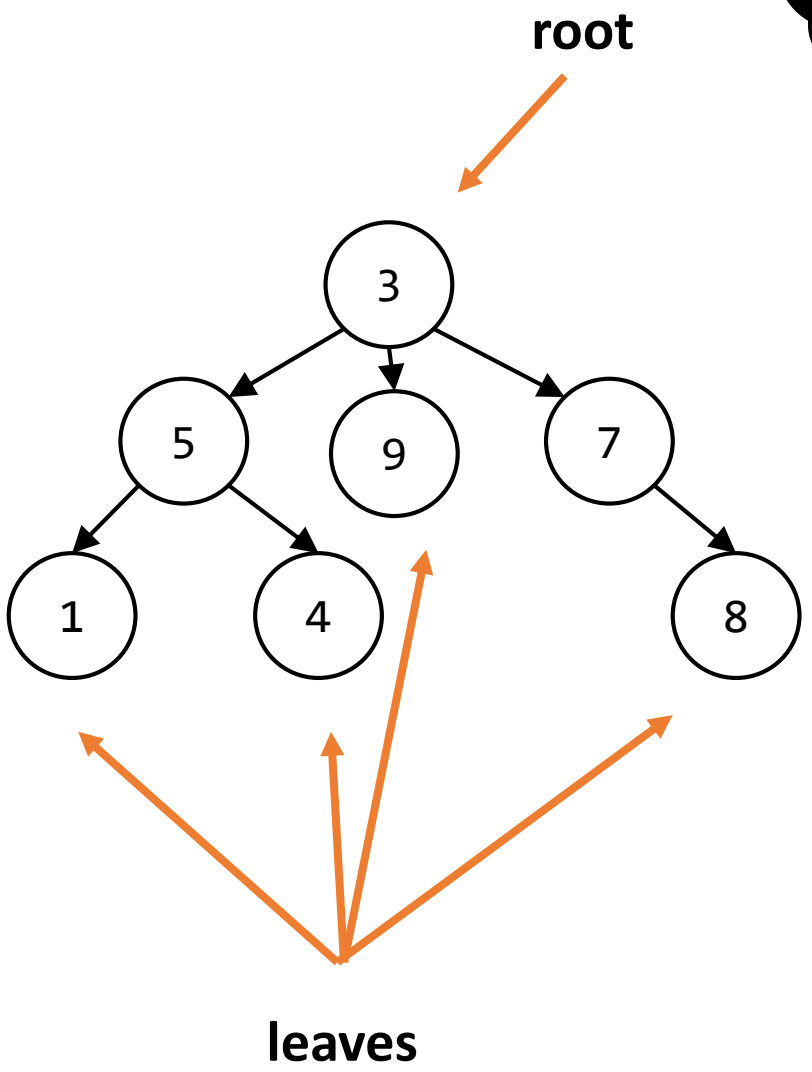


Unlike real trees, trees in computer science grow downward!

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 at the bottom with no children are called **leaves**.



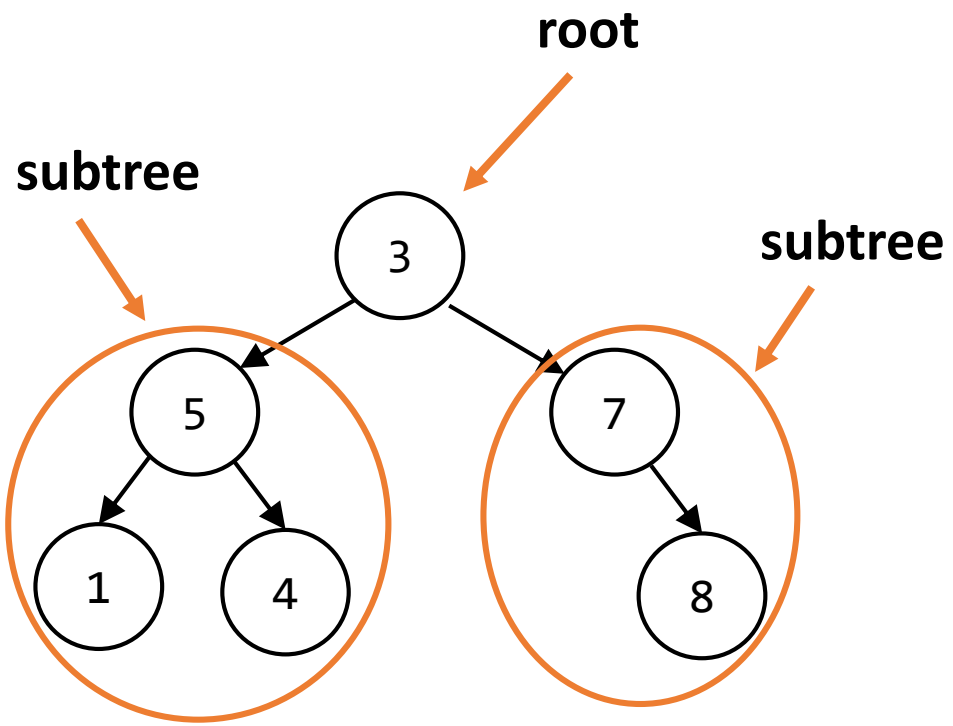
# Trees are Recursive

A tree is a naturally recursive data structure. Each node's children are **subtrees**.

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).

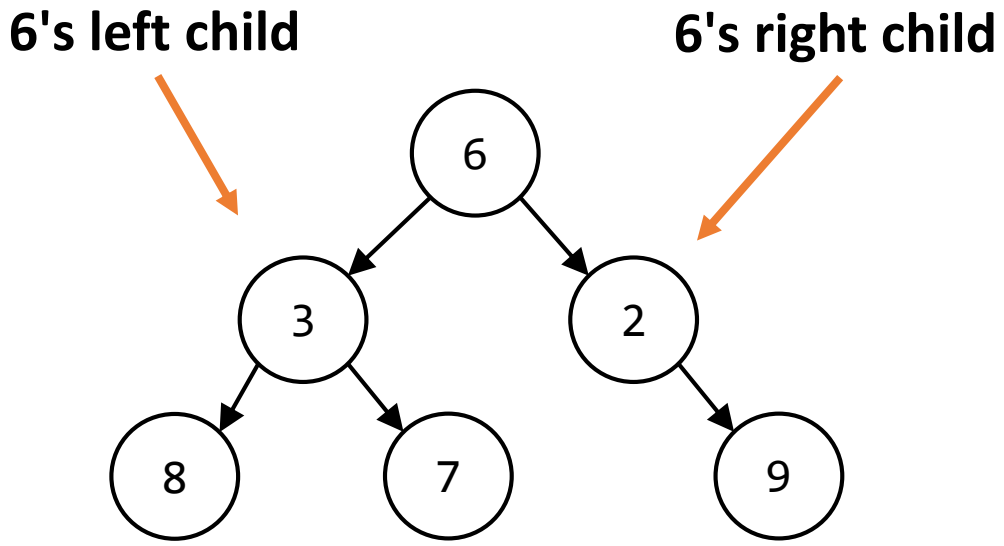
The **recursive case** is a node and its children which are also trees.



# 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.



# Coding with Trees

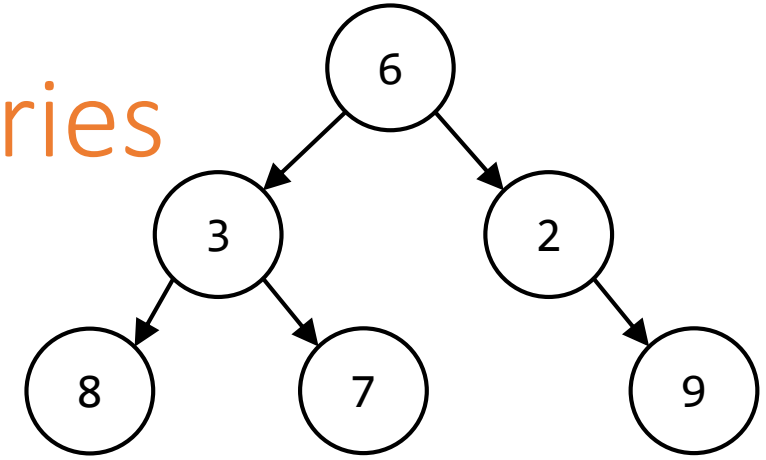
# Implementing New Data Structures

Computer science uses a large number of classical data structures. Some of these (like lists and dictionaries) are implemented directly by Python. Others (like hashtables) 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 find a solution.

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 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). 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**: return the leaf's value directly.

The **recursive case**: add the sums of the left and right children to the value.

```
def sumTree(t):
  if t["left"] == None and \
     t["right"] == None:
    return t["value"]
  else:
    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.

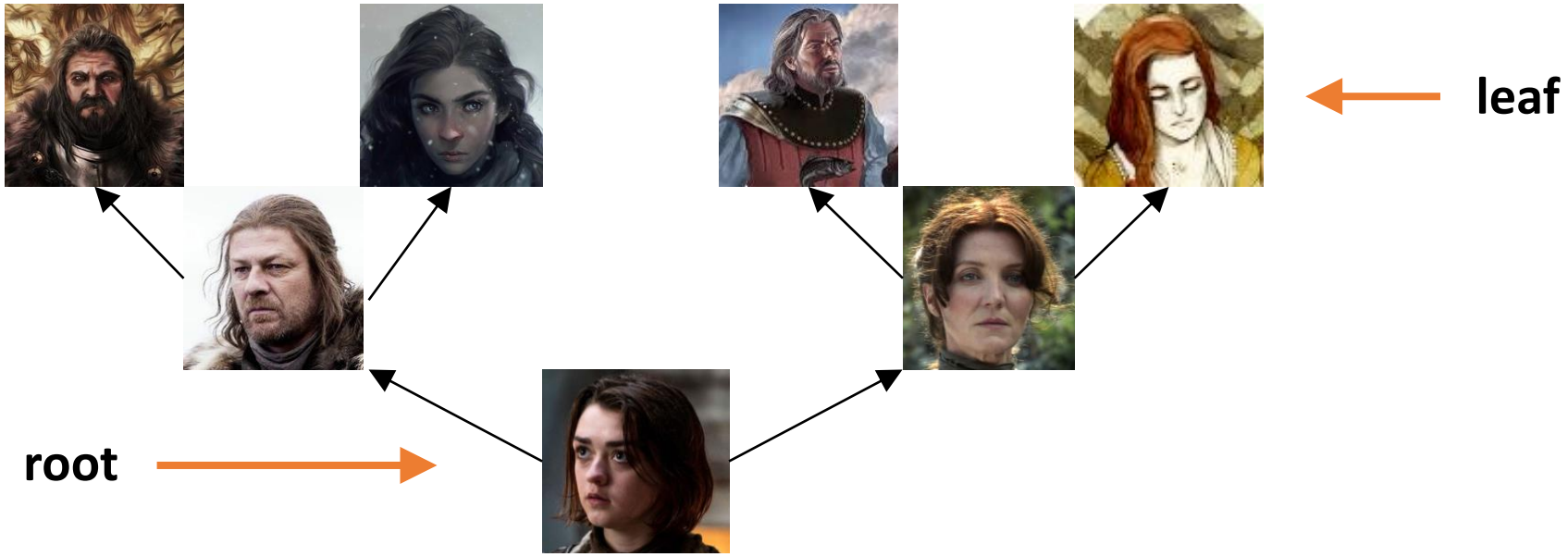
The difference here is that there are recursive calls to both children, even if they might be None.

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

# Advanced Example: Family Trees

Now let's write a function that takes a 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
```

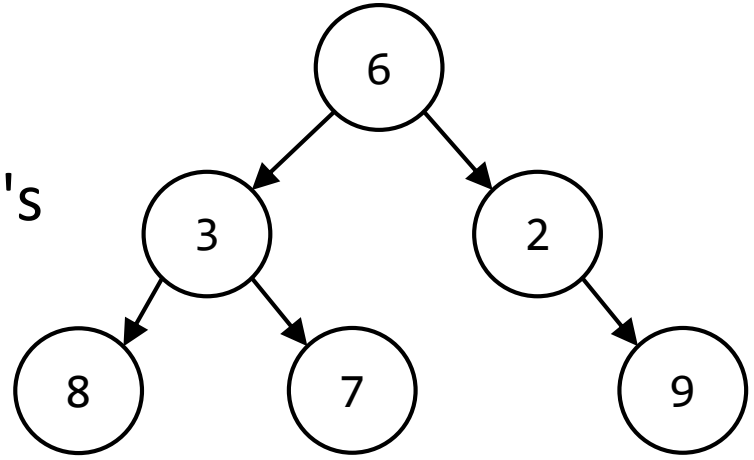
# Activity: listValues

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

Hint: 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 should return `[8, 3, 7, 6, 2, 9]`.

You can test your code by copying the example tree's implementation on Slide 10.



# Another Example: Search

How could we search a tree to see if it contains a given value? A value could be in any part of a tree, which means we need to check every single node to determine if the value exists.

We have two base cases: one for when we reach an empty tree, and one for when we find the item. In the recursive case, check if either subtree contains the item.

Efficiency: if there are n nodes in the tree, in the worst case we visit each node once; this is O(n). That's no better than linear search- can we do better?

```
def search(t, target):
  if t == None:
    return False
  elif t["value"] == target:
    return True
  else:
    return search(t["left"], target) or \
           search(t["right"], target)
```

# Binary Search Trees

# Specialized Data Structures

We can improve the performance of search on a tree, but to do so, we'll need to restrict how data is organized in the structure.

We've done this before! We're able to search for values in a hashtable in O(1) time, **but** we must use immutable values and have to store them according to a hash function. We're able to search a list in O(log n) time, **but** we must make sure the list is sorted first.

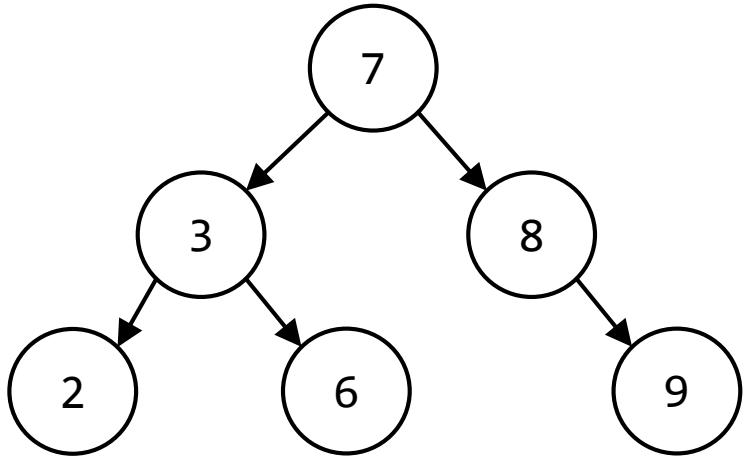
For trees, we'll use a similar restriction to lists. We'll 'sort' the tree by restricting where nodes can be located.

# Binary Search Trees (BST) are "sorted"

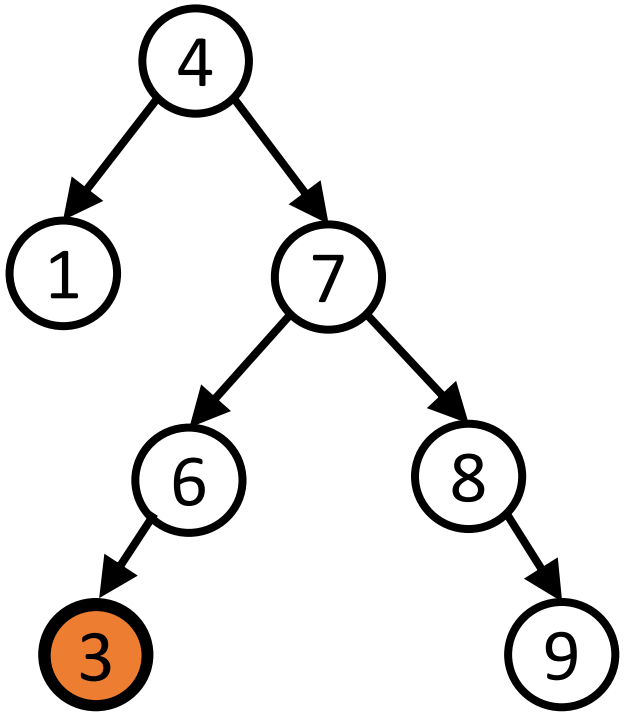
For every node n in a tree which has a value v:

- Each **left** child (and all its children, etc.) must be strictly **less** than v
- Each **right** child (and all its children, etc.) must be strictly **greater** than v

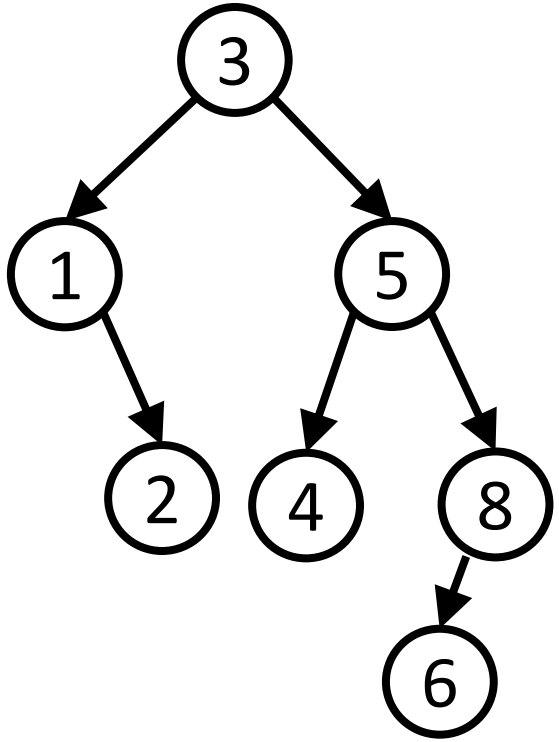
Note: the left and right subtrees are BSTs!  BSTs are recursive!



# Example: Is this a BST?



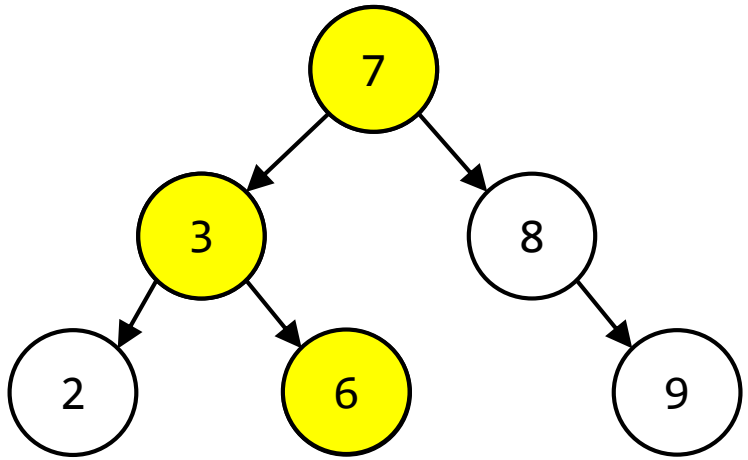

no




yes

# Binary Search Trees Can Use Binary Search

When we want to search for the value 5 in the tree to the left, we start at the root node, 7. Because all nodes less than 7 must be in the left child tree, and 5 is less than 7, **we only need to search the left child tree.**




Then, when we compare 5 to 3, we know that all values greater than 3 (but less than 7) must be in the right child of 3, and 5 is greater than 3. So we **only need to search the right child**.

This is just binary search! Hence, **Binary Search Tree** (**BST**).

# BST Search in Python

We would write binary search for a BST as follows:

```
def search(t, target):
    if target == None:
        return False
    elif t["value"] == target:
        return True
    elif target < t["value"]:
        return search(t["left"], target)
    else:
        return search(t["right"], target)
```

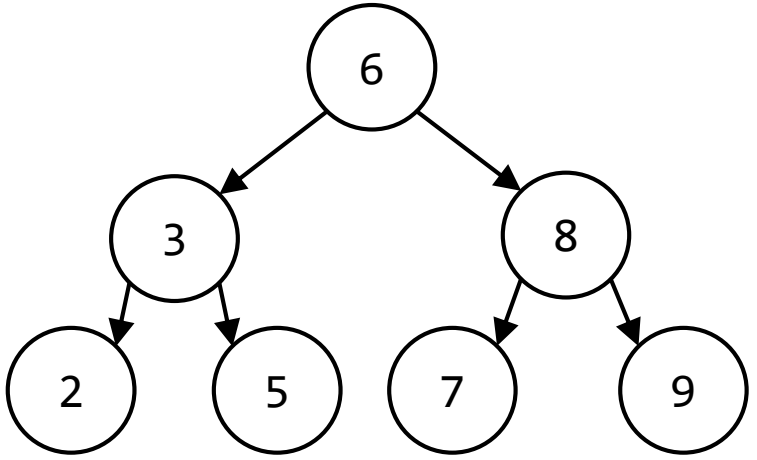
Note that we do just one recursive call, either on the left subtree or on the right subtree.

# BST Search Runtime – Balanced Trees

Let's consider the runtime of search on a BST that is balanced.

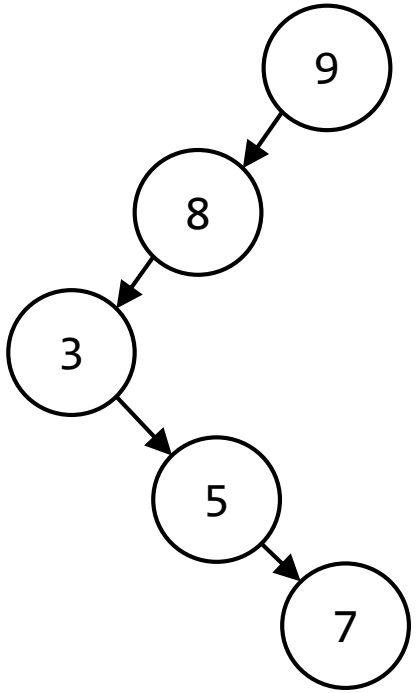
A tree is **balanced** if for every node in the tree, the node's left and right subtrees are approximately the same size. This results in a tree that minimizes the number of recursive levels.

Every time you take a search step in a balanced tree, you cut the number of nodes to be searched in half. This means that you'll take O(log n) time, like with ordinary binary search.



# BST Search Runtime – Unbalanced Trees

A tree is considered **unbalanced** if at least one node has significantly different sizes in its left and right children. For example, consider the tree on the right.




This is a valid BST, but it is still difficult to search! If you search it for a number like 6, it can still take **O(n)** time.

When we put data into BSTs, we usually strive to make them balanced, to avoid these edge cases. You can assume the average runtime will be O(log n).

# Benefits of BSTs

At first glance, BSTs may seem less useful than hashtables or dictionaries. However, they can have perks!

For example, storing data in a BST lets us quickly find data that is **close** to a specific value, in addition to searching for a value itself. This can provide contextual information, and makes certain tasks (like looking for a good-enough value) much easier.

BSTs also make it much easier to **add new data** to a dataset. In a sorted list, you would need to slide a bunch of values over; in a BST, you can just search for this new value and when you reach a leaf, add a node with the new value.

In general, try to choose a data structure that matches the task you need to solve.

# Learning Goals

- Understand how a **hierarchical** data structure uses recursion to store data
- Define core concepts of **trees**, including **nodes**, **children**, the **root**, and **leaves**
- Identify the difference between **trees**, **binary trees**, and **binary search trees**
- Search for values in **binary search trees** using **binary search**
- Use **trees** and **binary trees** implemented with dictionaries when reading and writing code