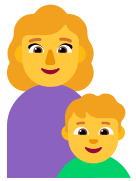
TREES

OBJECTIVES

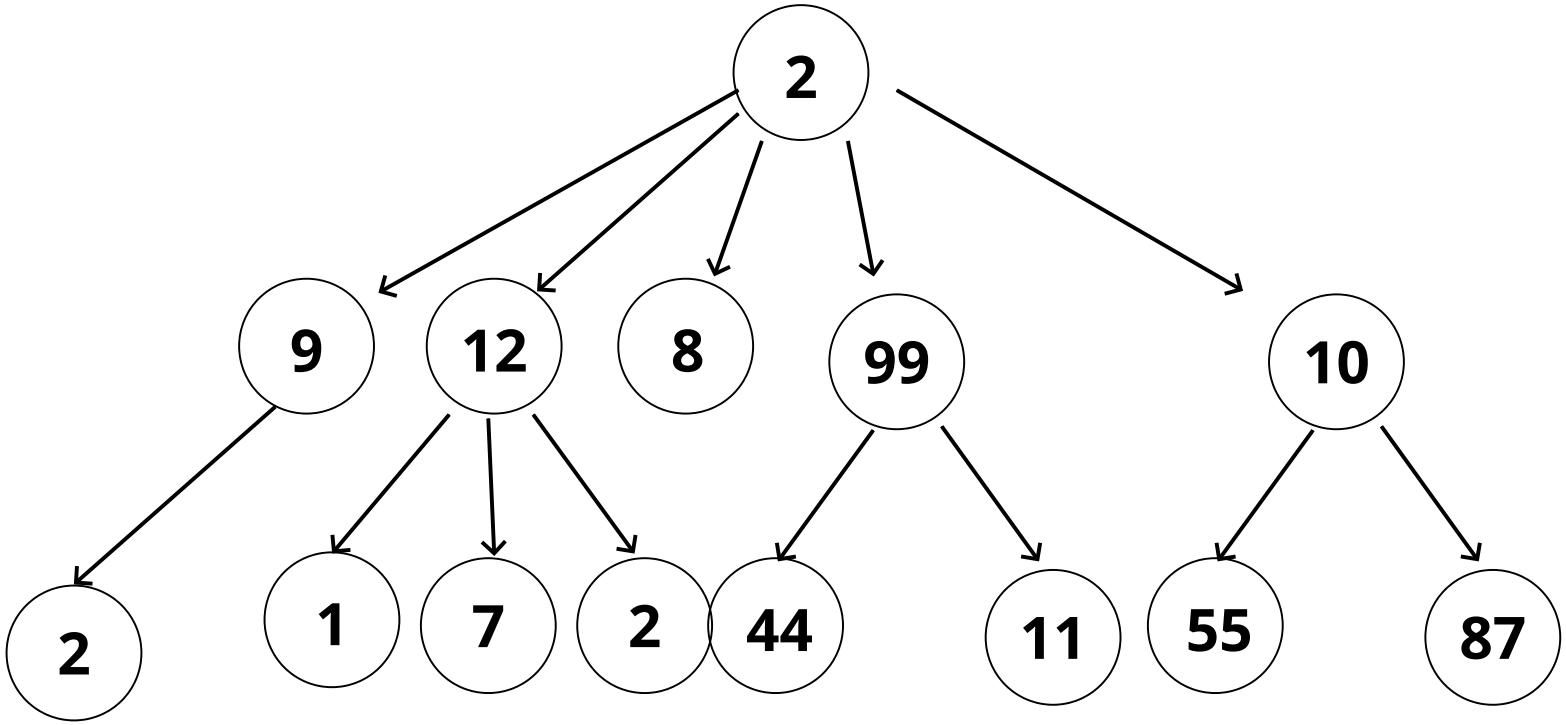
- Define what a tree is
- Compare and contrast trees and lists
- Explain the differences between trees, binary trees, and binary search trees
- Implement operations on binary search trees

WHAT IS A TREE?

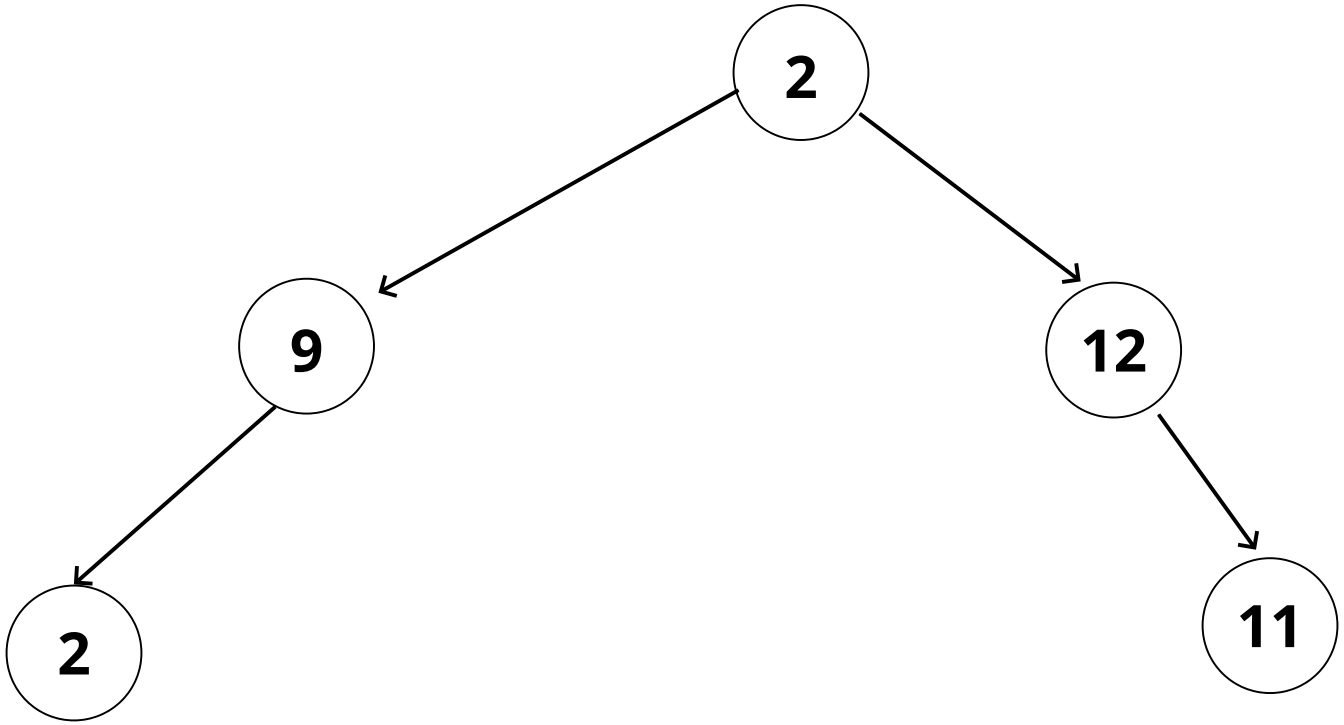
A data structure that consists of nodes in a **parent / child** relationship



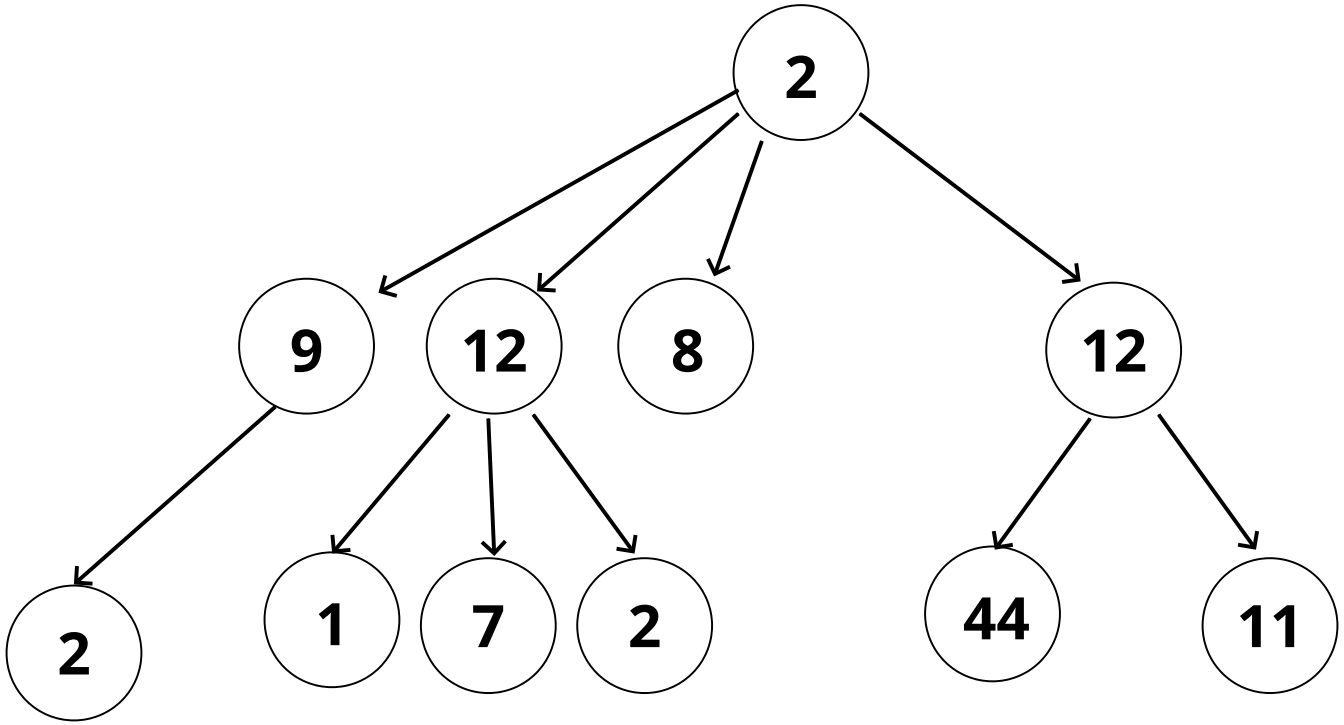
TREES



TREES



TREES

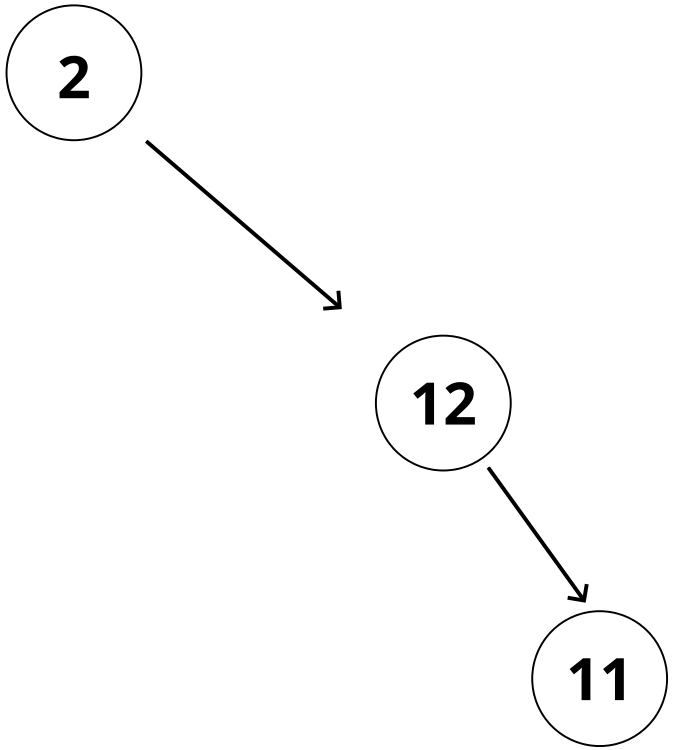


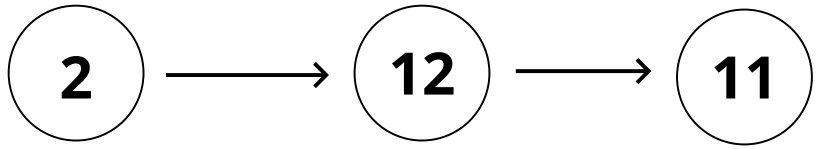
Lists - **linear**

Trees - **nonlinear**

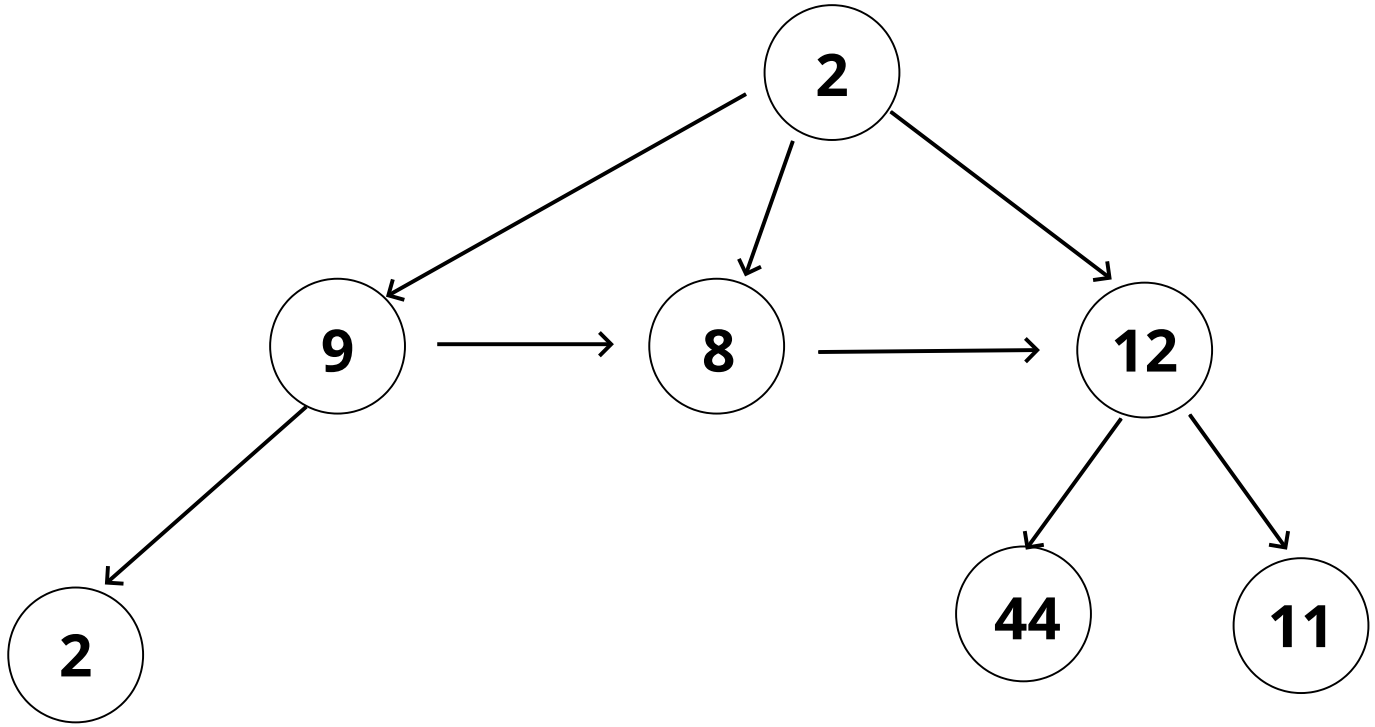
A Singly Linked List

(sort of a special case of a tree)

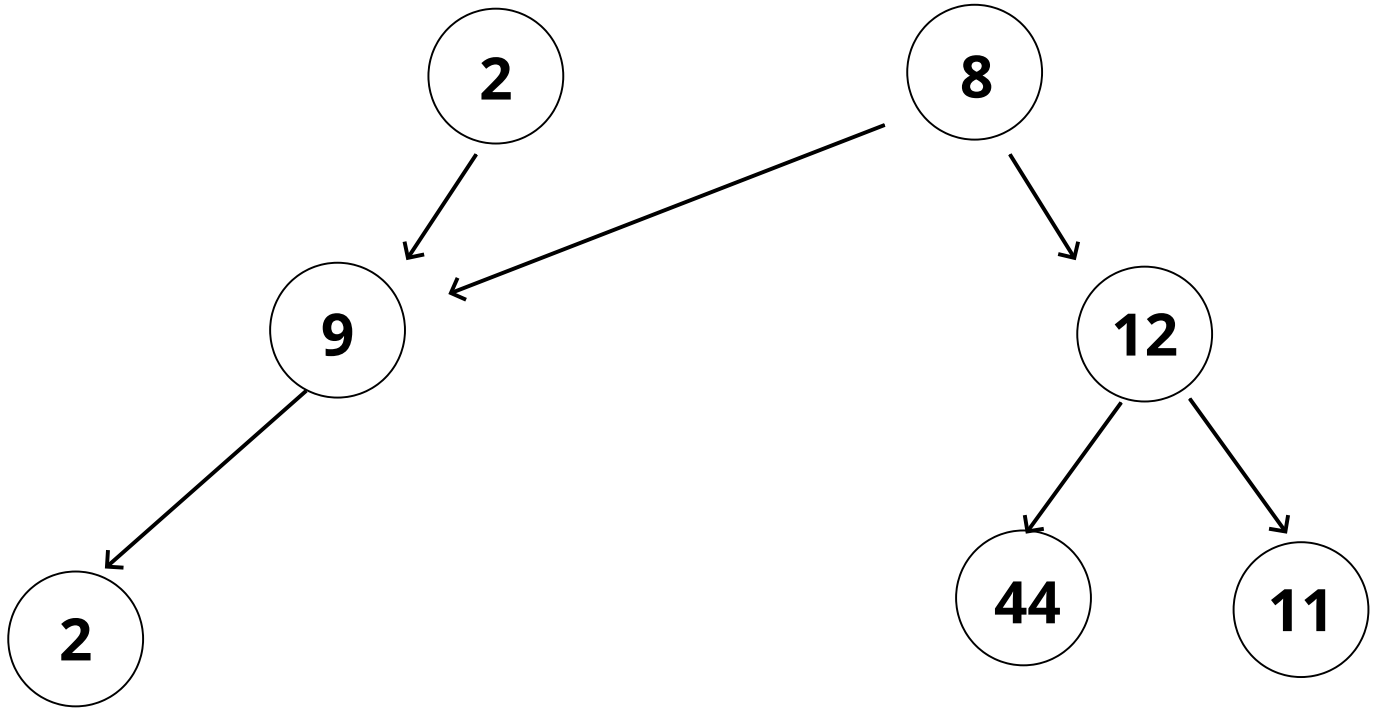




NOT A TREE



NOT A TREE



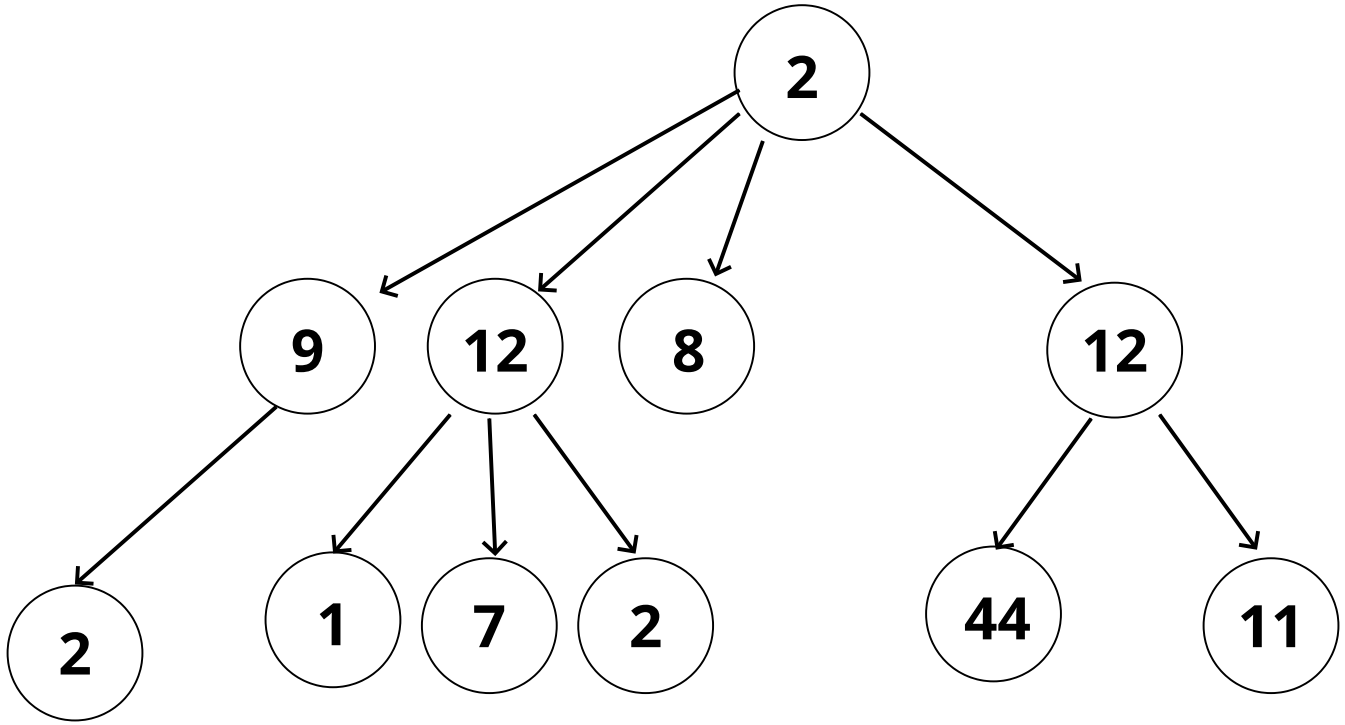
TREE TERMINOLOGY

- **Root -** The top node in a tree.
- **Child** -A node directly connected to another node when moving away from the Root.
- **Parent** - The converse notion of a child.
- **Siblings** -A group of nodes with the same parent.
- **Leaf** - A node with no children.
- **Edge** - The connection between one node and another.

KINDS OF TREES

- Trees
- Binary Trees
- Binary Search Trees

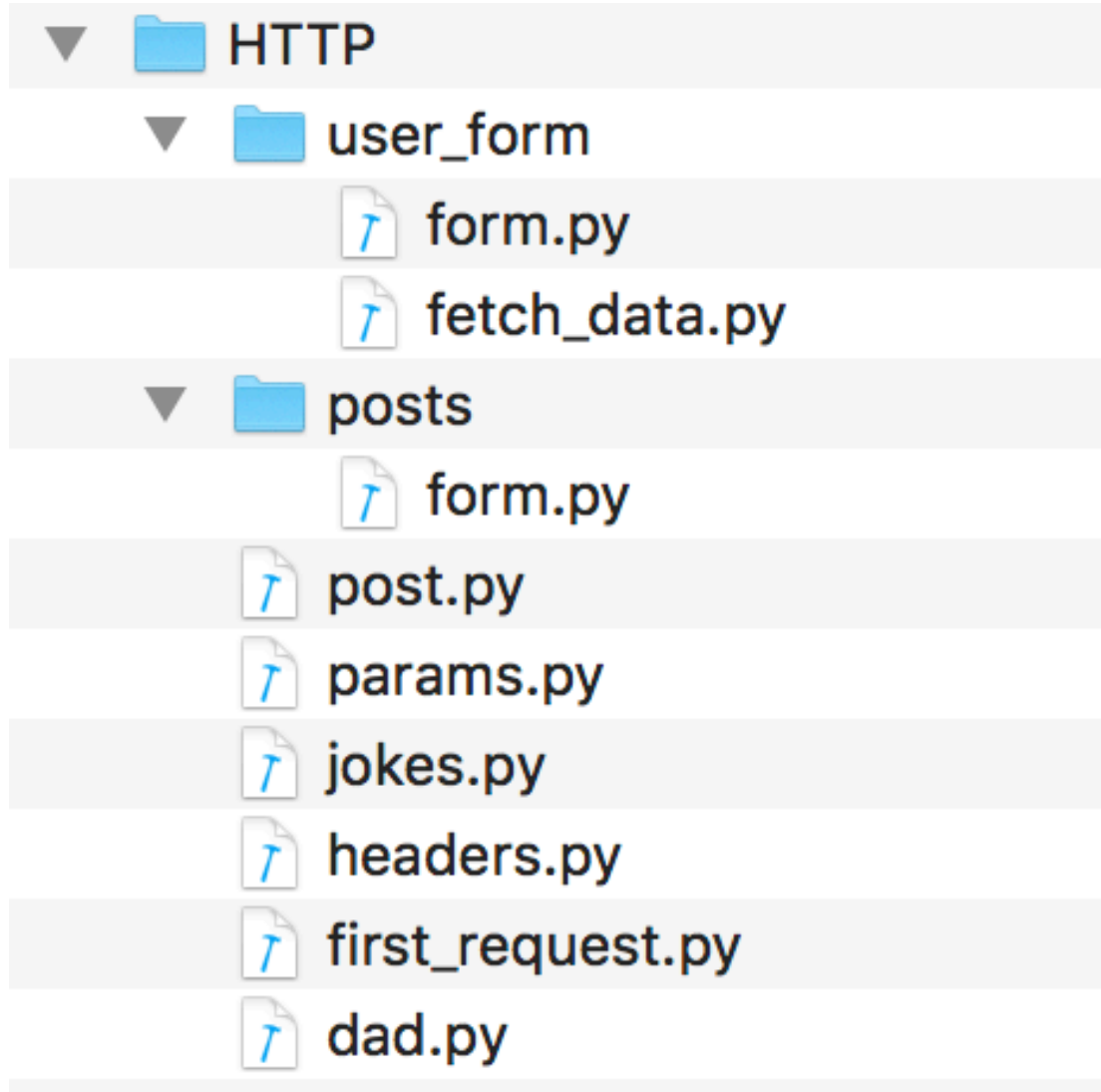
TREES



TREES

Lots of different applications!

- HTML DOM
- Network Routing
- Abstract Syntax Tree
- Artificial Intelligence
- Folders in Operating Systems
- Computer File Systems

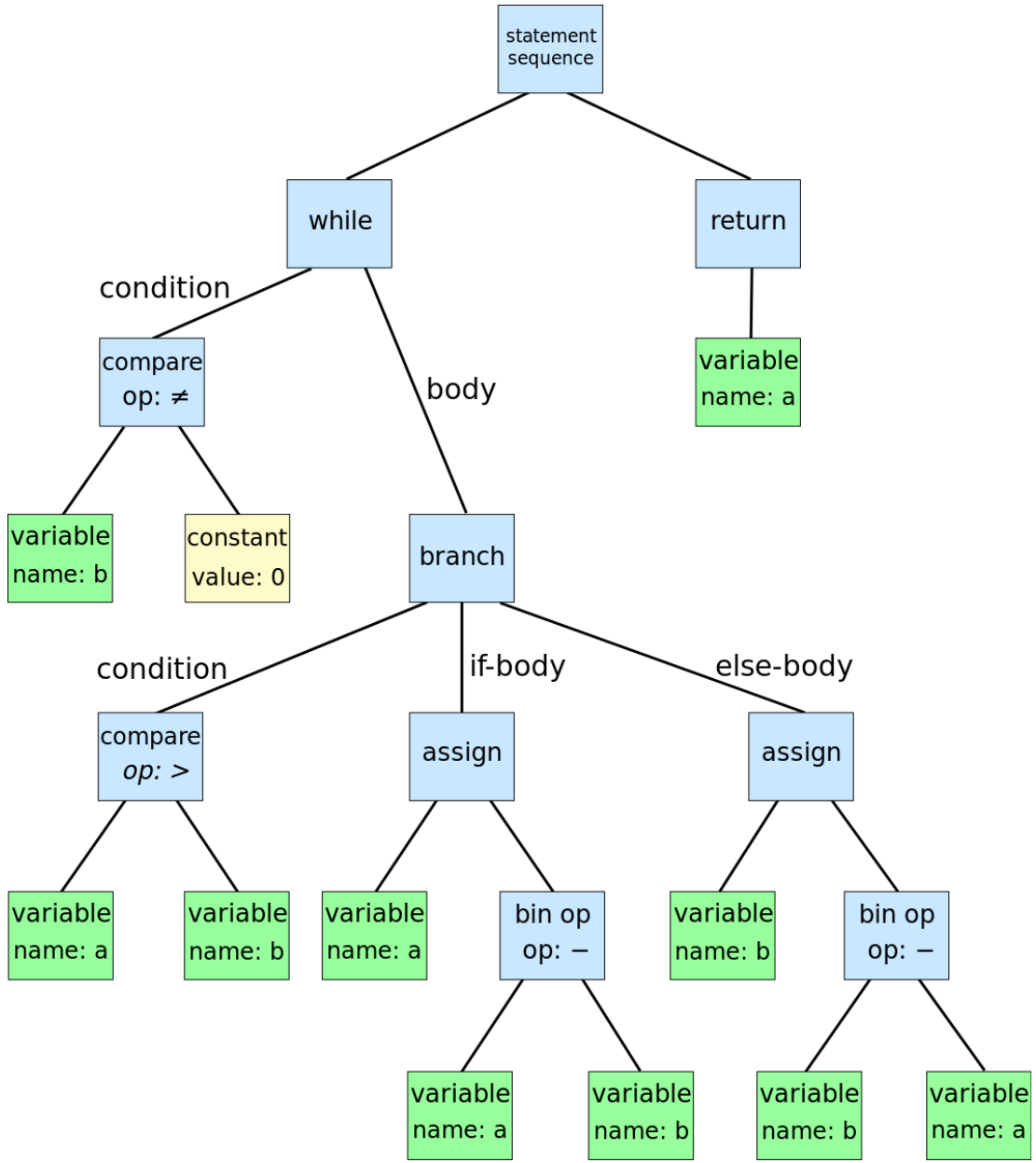

HTTP
user_form
form.py
fetch_data.py
posts
form.py
post.py
params.py
jokes.py
headers.py
first_request.py
dad.py

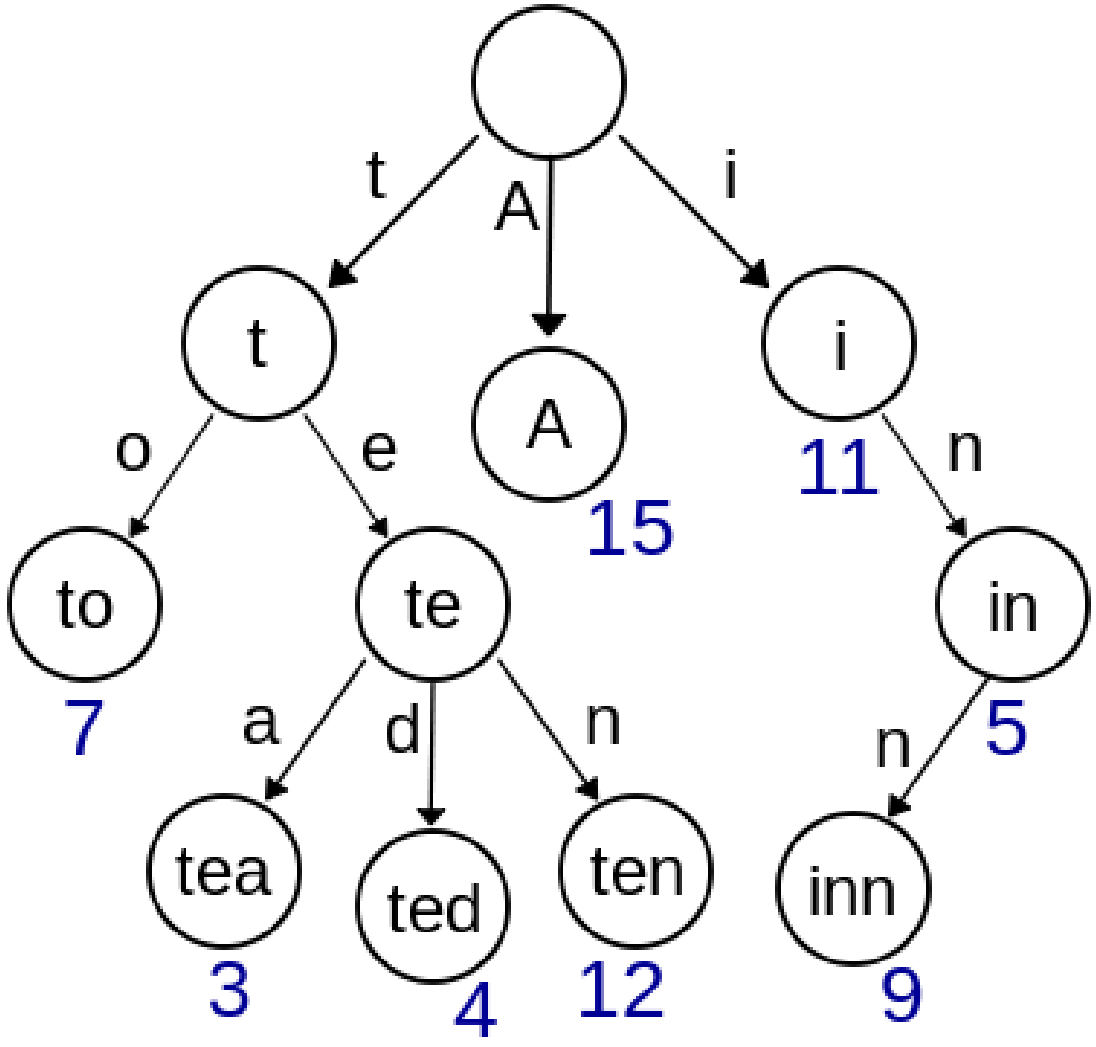

...<body class="default-theme des-mat" style="background: rgb(255, 255, 255);">

<div class id="mngb">

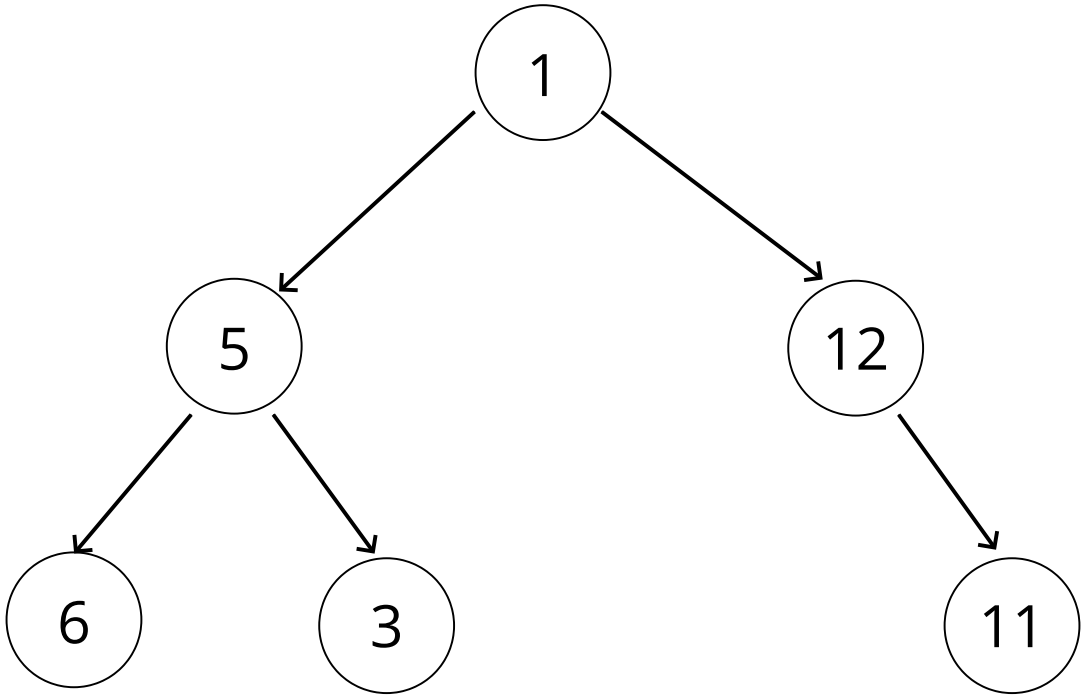
<div class="gb_nb gb_Pg gb_R gb_Og gb_Sg gb_T" style="min-width: 241px;
">…

…
<textarea name="csi" id="csi" style="display:none"></textarea>
<script nonce="MHwVu6oUuFKDgG6HDEGpzQ==">…</script>
…
…
<script src="/xjs/_/js/k=xjs.ntp.en_US.4JEq6NIPOCY.O/m=spch/am=gAgSMw/rt=j/
d=1/exm=sx,jsa,ntp,d,csi/ed=1/rs=ACT90oGmcAW5XA4yhGQRyZR_wRqm9_fi3w?xjs=s1"

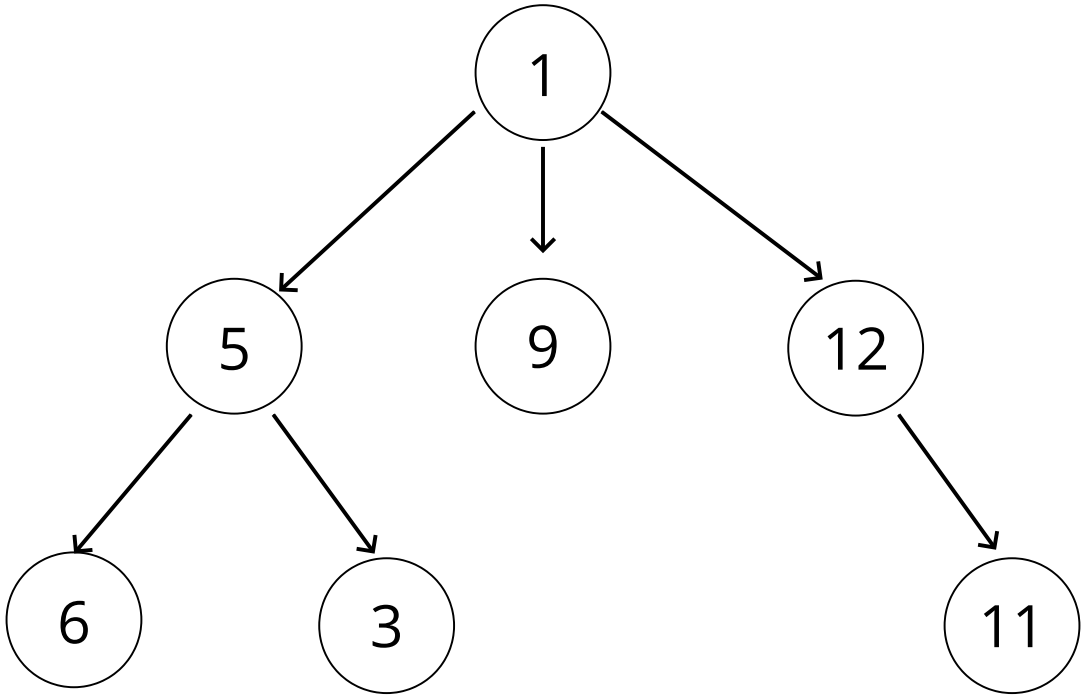

statement sequence
while
return
condition
body
compare op: ≠
variable name: a
variable name: b
constant value: 0
branch
condition
if-body
else-body
compare op: >
assign
assign
variable name: a
variable name: b
variable name: a
bin op op: −
variable name: b
bin op op: −
variable name: a
variable name: b
variable name: b
variable name: a


t
A
i
t
A
i
o
e
11
n
15
to
te
in
7
a
d
n
n
5
tea
ted
ten
inn
3
4
12
9

BINARY TREES



NOT A BINARY TREE

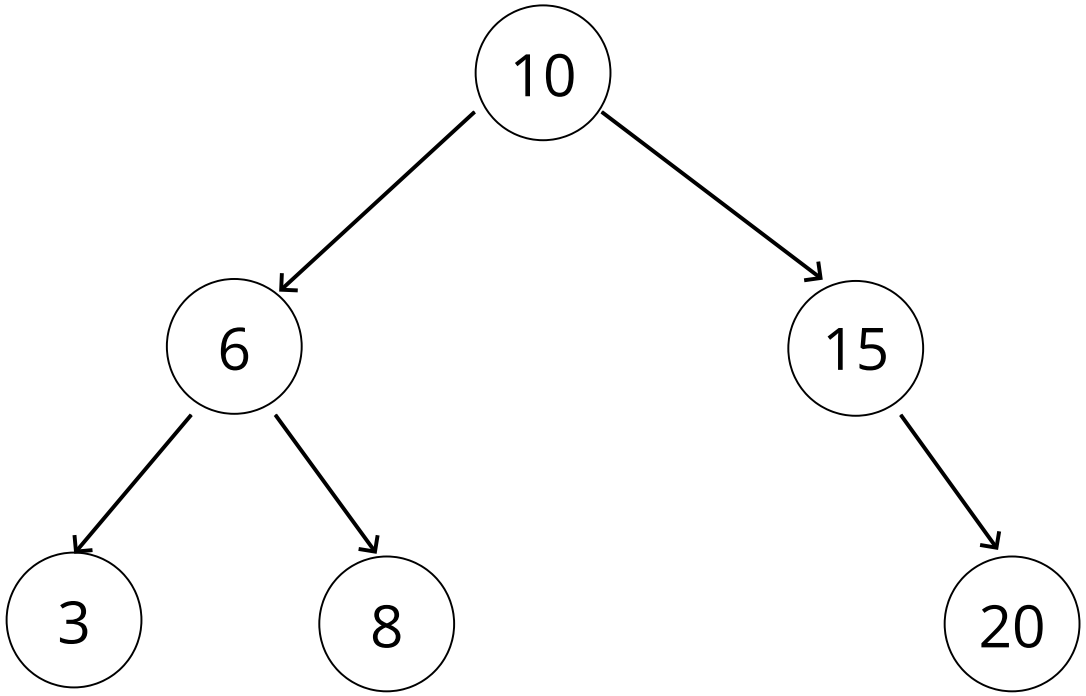


BINARY TREES

Lots of different applications as well!

- Decision Trees (true / false)
- Database Indicies
- Sorting Algorithms

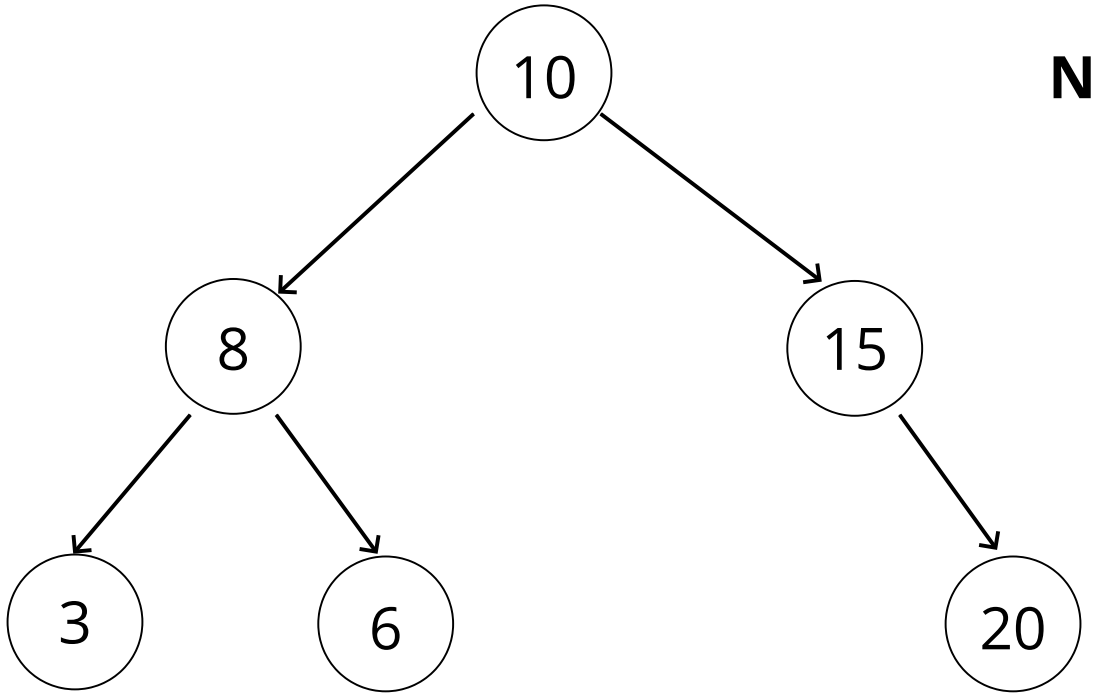
BINARY SEARCH TREES



HOW BSTS WORK

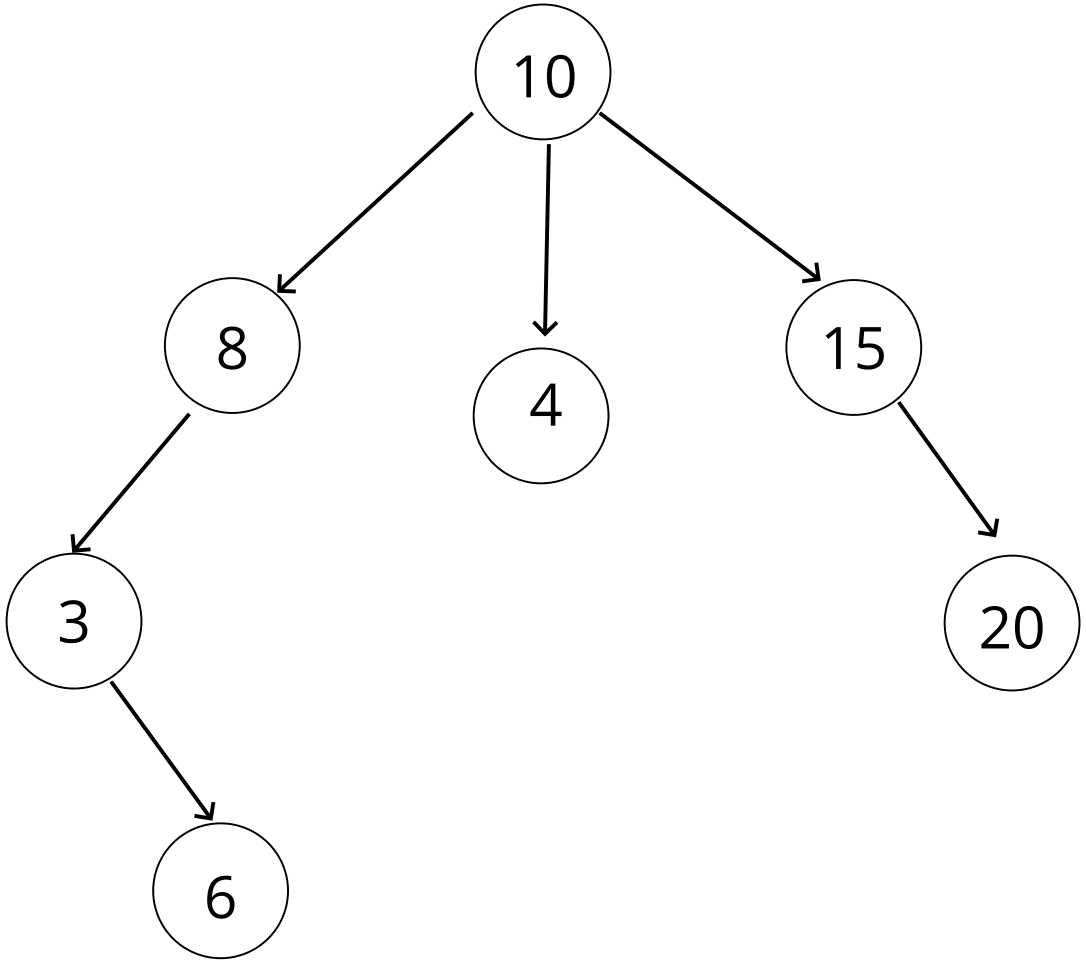
- Every parent node has at most **two** children
- Every node to the left of a parent node is **always less** than the parent
- Every node to the right of a parent node is **always greater** than the parent

Is this a valid BST?



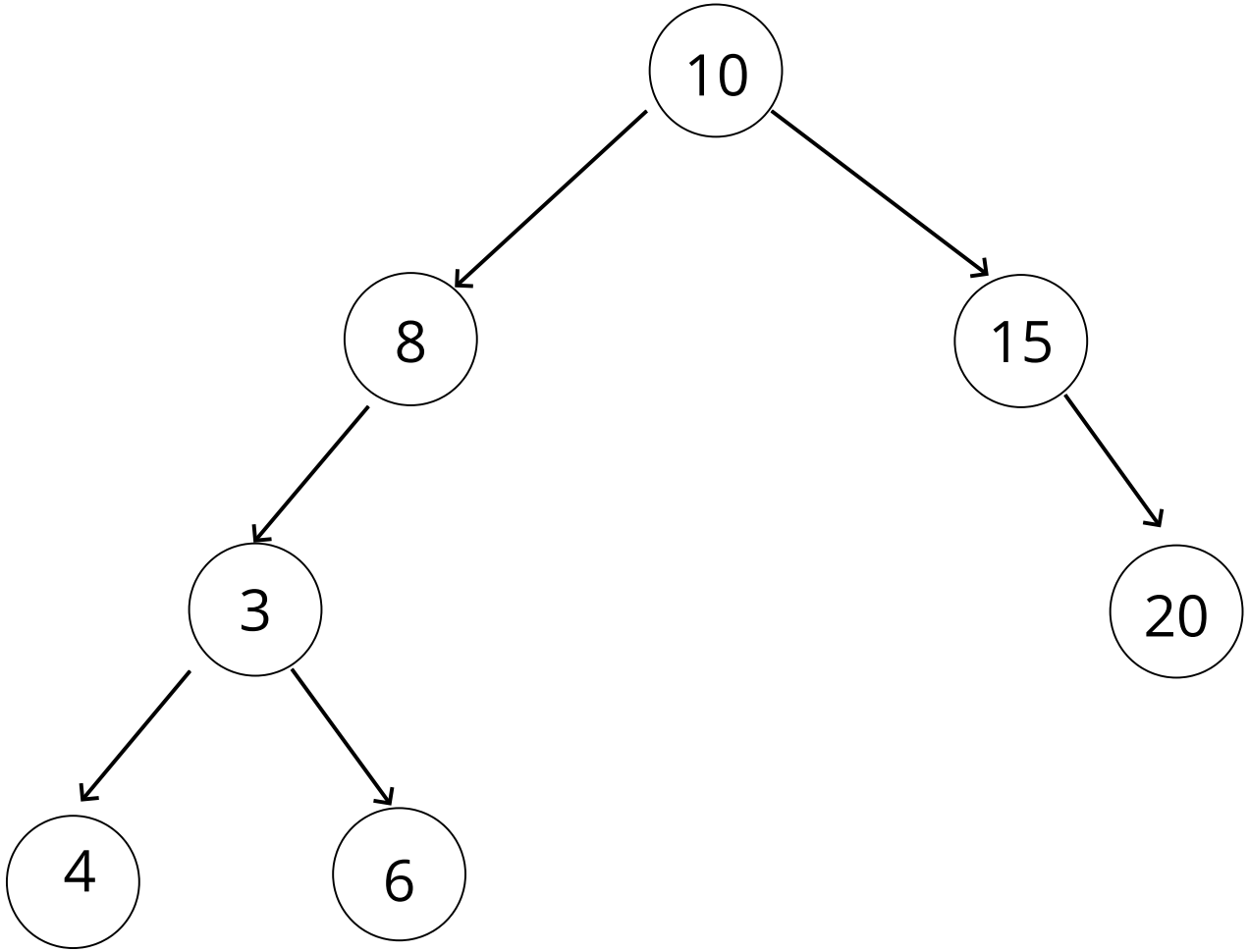
NOPE

Is this a valid BST?



NOPE!

Is this a valid BST?



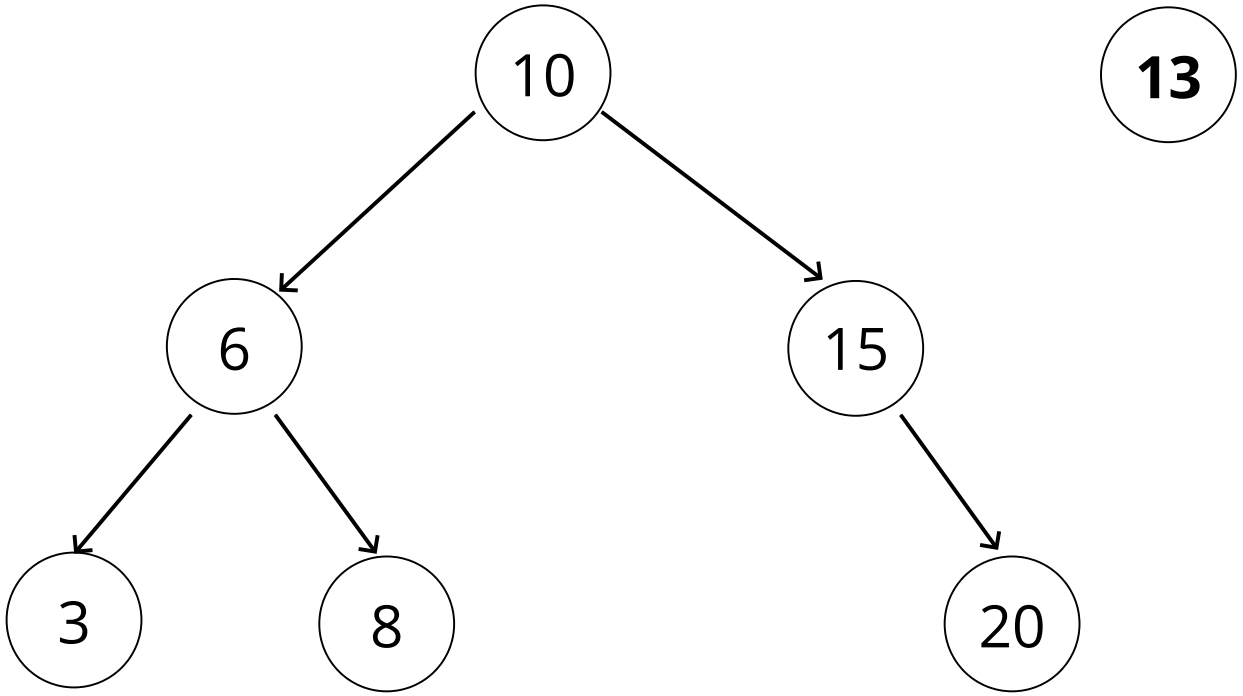
NOPE!

The BinarySearchTree Class

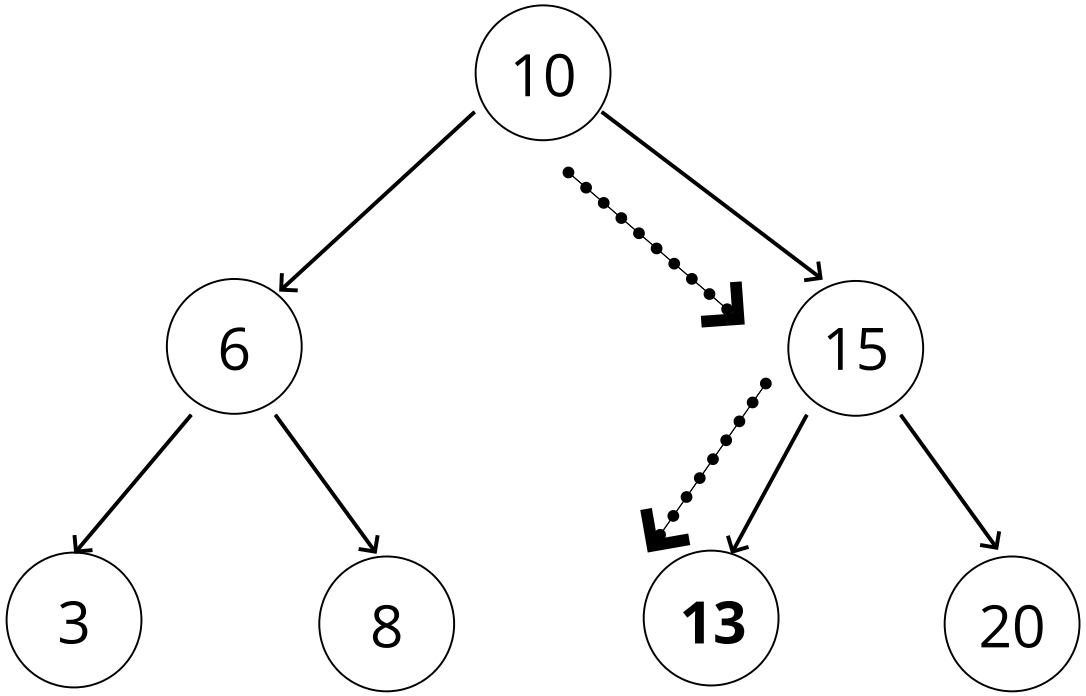
```
class BinarySearchTree {
    constructor(){
        this.root = null;
    }
}
```

```
class Node {
    constructor(value){
        this.value = value;
        this.left = null;
        this.right = null;
    }
}
```

INSERTING



INSERTING



INSERTING A NODE

Steps - Iteratively or Recursively

- Create a new node
- Starting at the root
  - Check if there is a root, if not - the root now becomes that new node!
  - If there is a root, check if the value of the new node is greater than or less than the value of the root
  - If it is greater
    - Check to see if there is a node to the right
      - If there is, move to that node and repeat these steps
      - If there is not, add that node as the right property
  - If it is less
    - Check to see if there is a node to the left
      - If there is, move to that node and repeat these steps
      - If there is not, add that node as the left property

YOUR TURN

Finding a Node in a BST

Steps - Iteratively or Recursively

- Starting at the root
  - Check if there is a root, if not - we're done searching!
  - If there is a root, check if the value of the new node is the value we are looking for. If we found it, we're done!
  - If not, check to see if the value is greater than or less than the value of the root
  - If it is greater
    - Check to see if there is a node to the right
      - If there is, move to that node and repeat these steps
      - If there is not, we're done searching!
  - If it is less
    - Check to see if there is a node to the left
      - If there is, move to that node and repeat these steps
      - If there is not, we're done searching!

Let's visualize this!

YOUR TURN

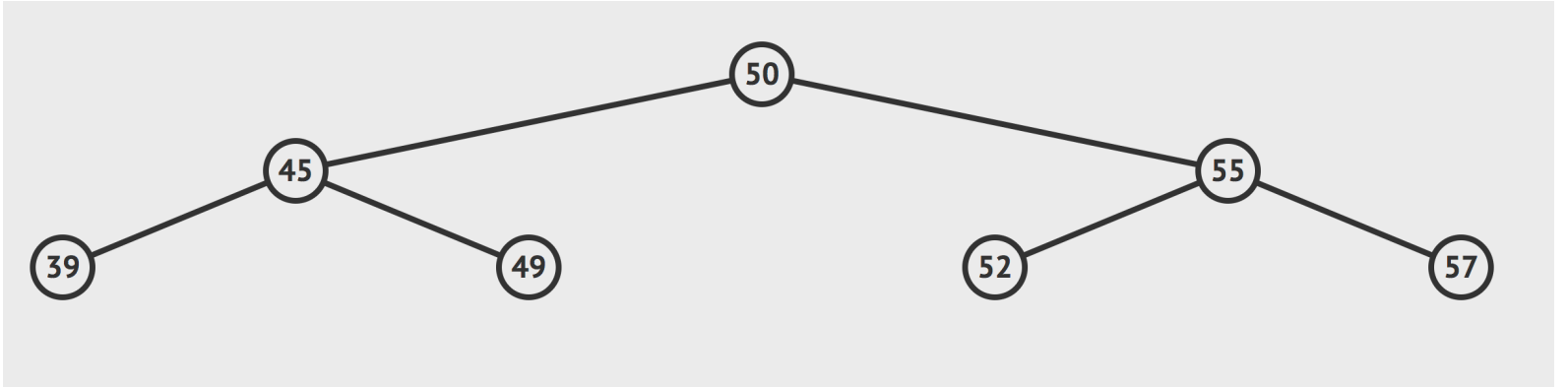
Big O of BST

Insertion - **O(log n)**

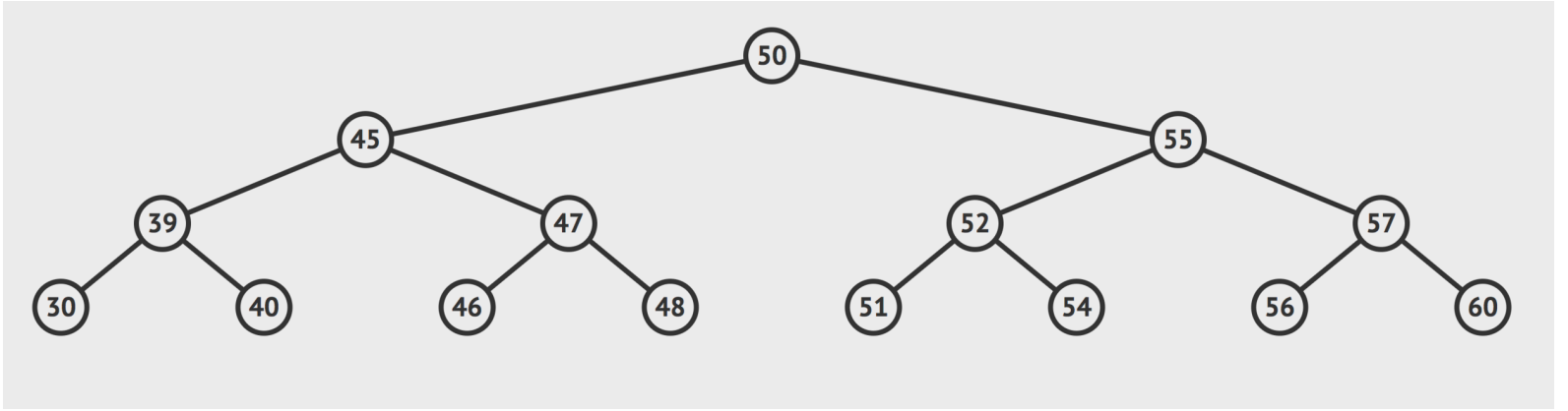
Searching - **O(log n)**

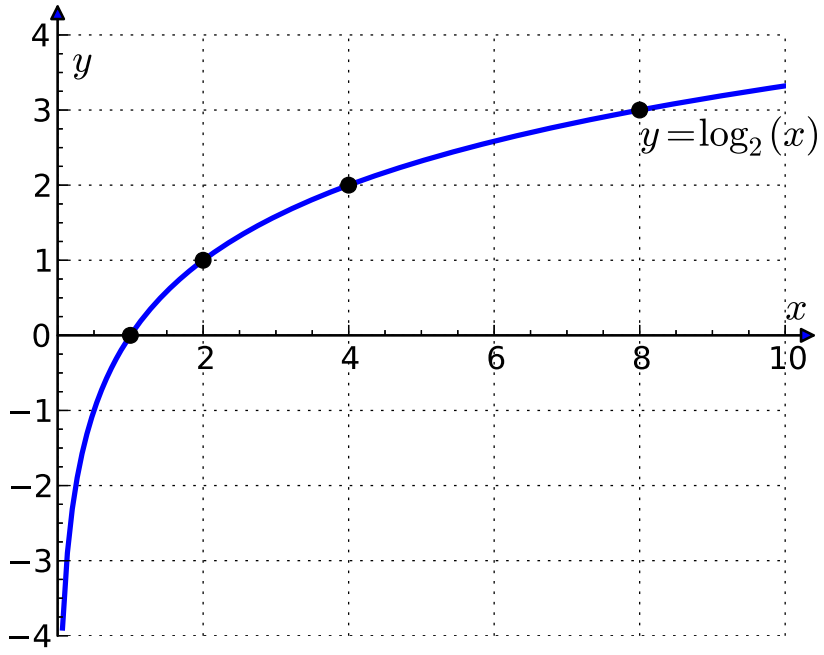
NOT guaranteed!

Double the number of nodes...



You only increase the number of steps to insert/find by 1





2x number of nodes: 1 extra step

4x number of nodes: 2 extra steps

8x number of nodes: 3 extra steps

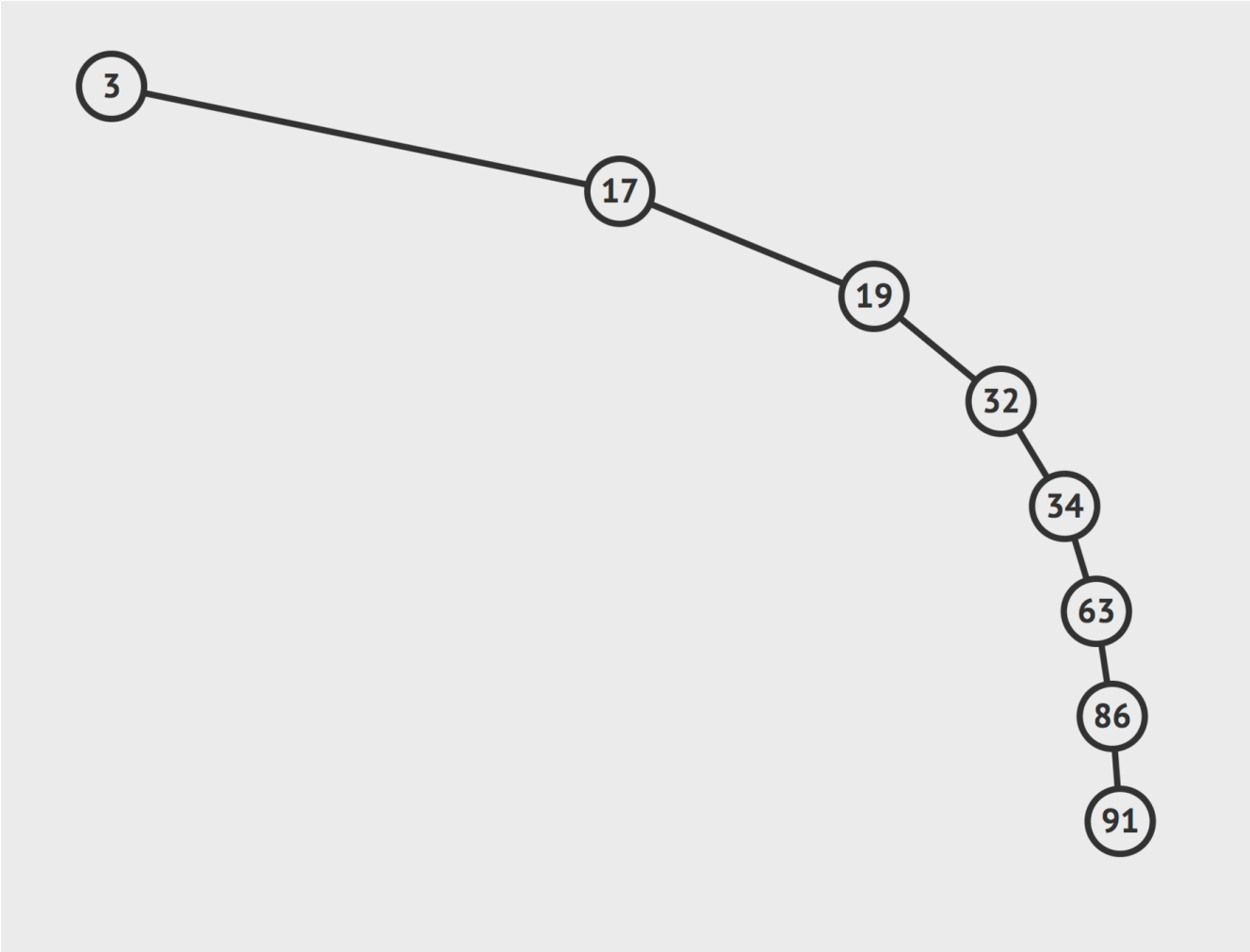
Big O of BST

Insertion - **O(log n)**

Searching - **O(log n)**

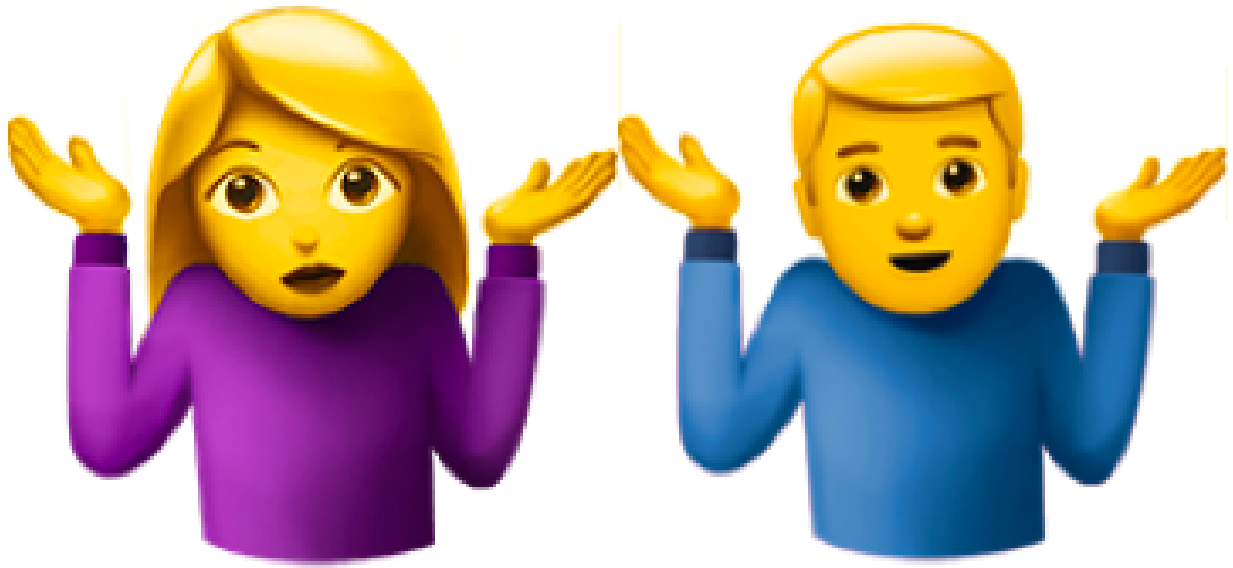
😬 **NOT guaranteed!** 😬

THIS IS A VALID BINARY SEARCH TREE



TREE TRAVERSAL

VISIT EVERY NODE ONCE



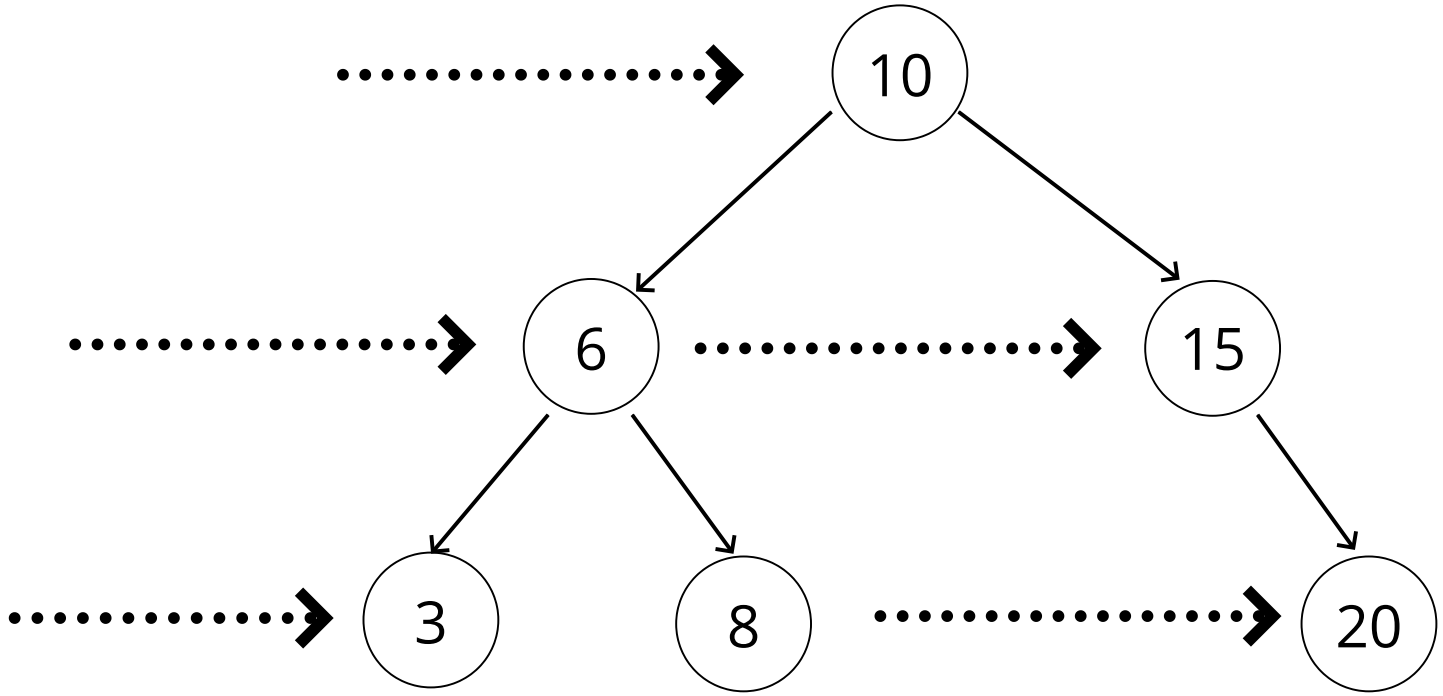
TRAVERSING A TREE

Two ways:

- Breadth-first Search
- Depth-first Search

BREADTH FIRST SEARCH

BFS



[10, 6, 15, 3, 8, 20]

BFS

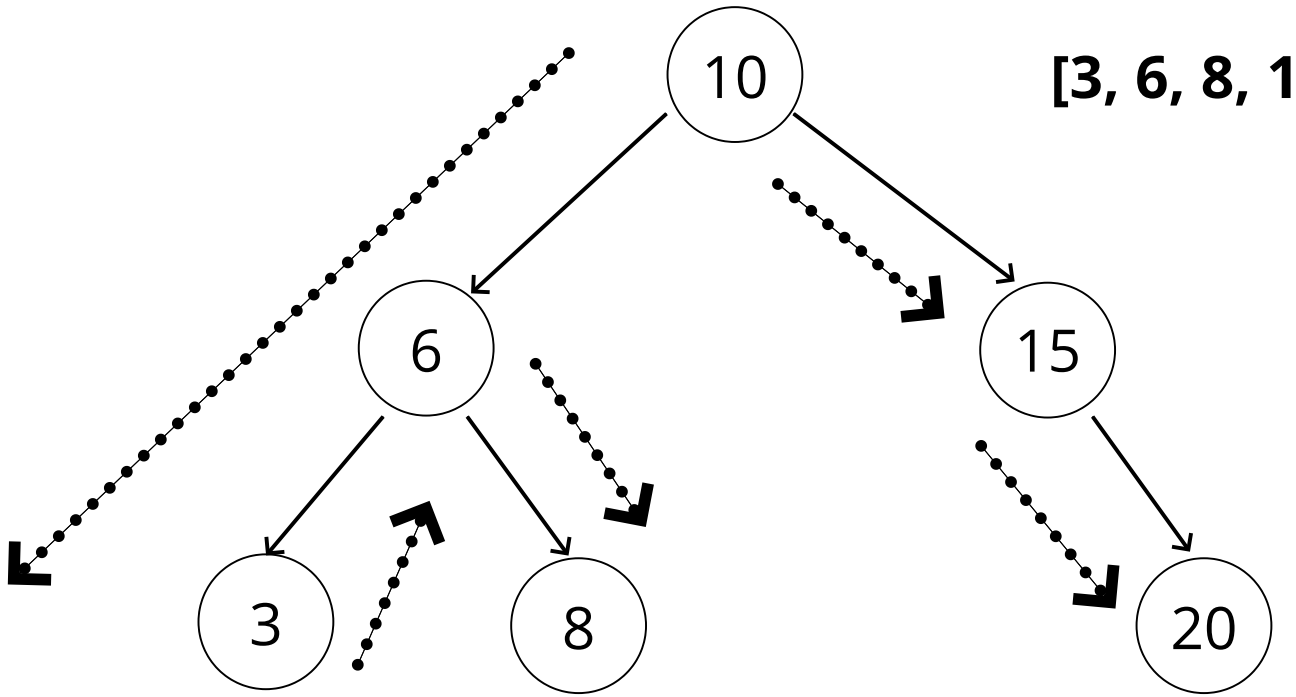
Steps - Iteratively

- Create a queue (this can be an array) and a variable to store the values of nodes visited
- Place the root node in the queue
- Loop as long as there is anything in the queue
  - Dequeue a node from the queue and push the value of the node into the variable that stores the nodes
  - If there is a left property on the node dequeued - add it to the queue
  - If there is a right property on the node dequeued - add it to the queue
- Return the variable that stores the values

YOUR TURN

DEPTH FIRST SEARCH

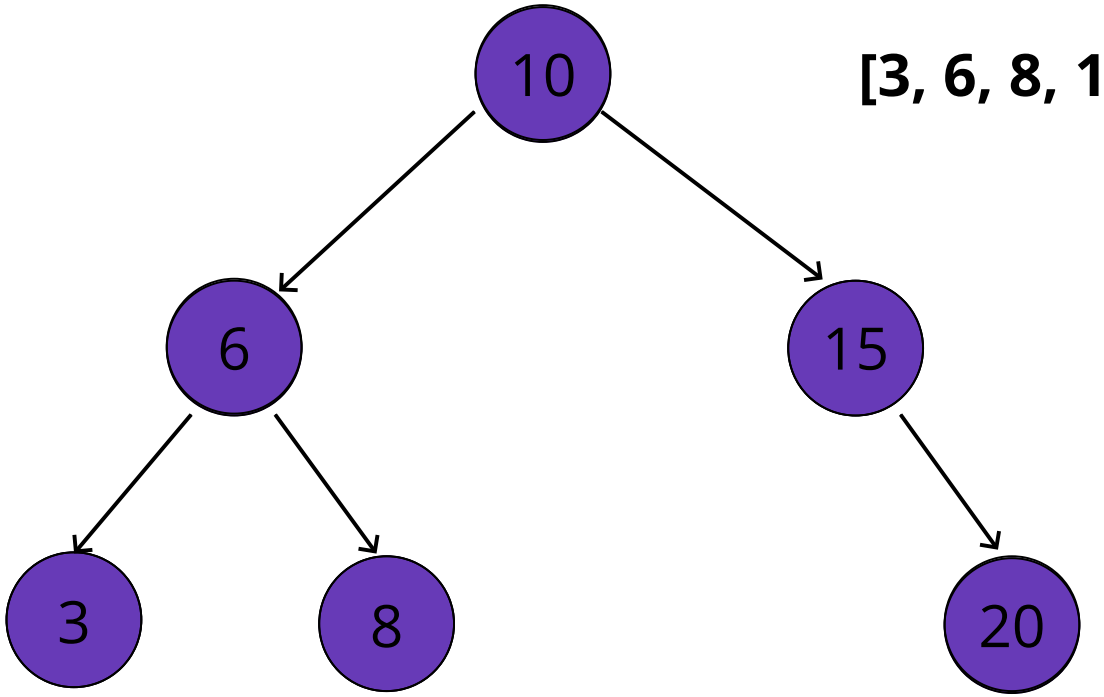
DFS - InOrder



[3, 6, 8, 10, 15, 20]

Let's visualize this!

DFS - InOrder



[3, 6, 8, 10, 15, 20]

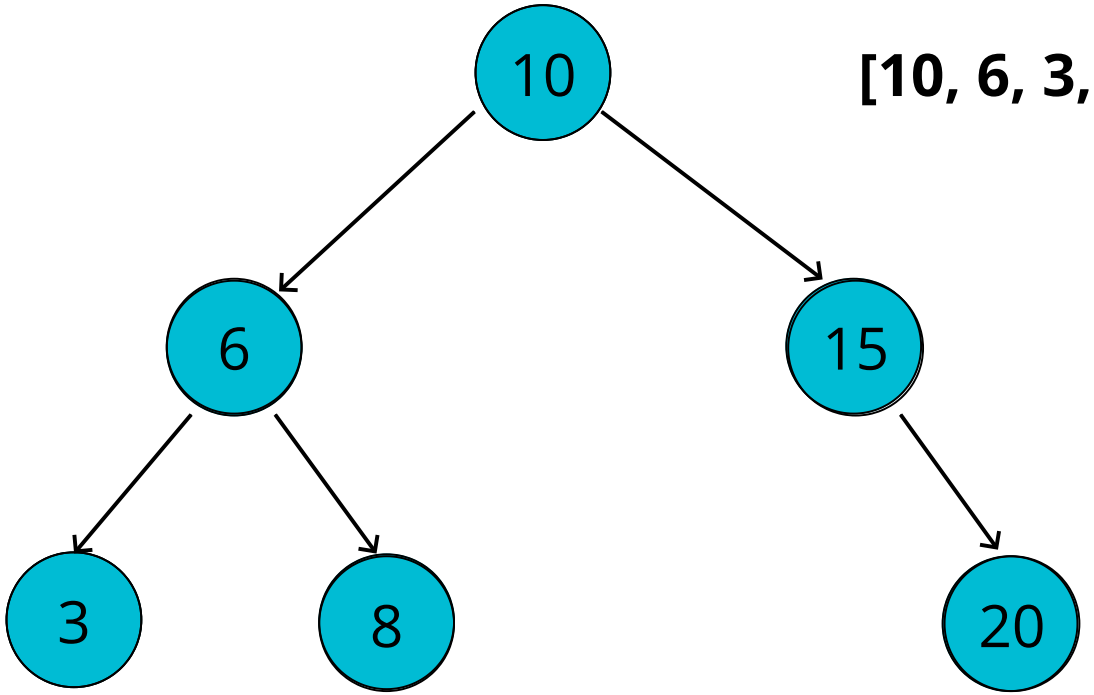
Let's visualize this!

DFS - InOrder

Steps - Recursively

- Create a variable to store the values of nodes visited
- Store the root of the BST in a variable called current
- Write a helper function which accepts a node
  - If the node has a left property, call the helper function with the left property on the node
  - Push the value of the node to the variable that stores the values
  - If the node has a right property, call the helper function with the right property on the node
- Invoke the helper function with the current variable
- Return the array of values

DFS - PreOrder



[10, 6, 3, 8, 15, 20]

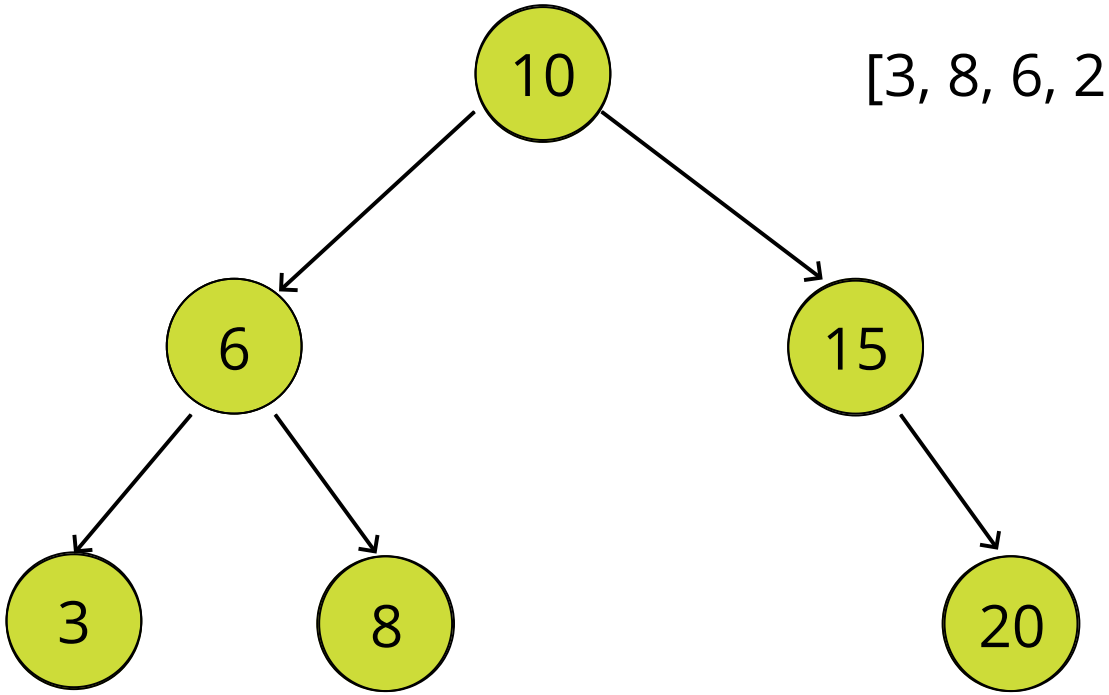
Let's visualize this!

DFS - PreOrder

Steps - Recursively

- Create a variable to store the values of nodes visited
- Store the root of the BST in a variable called current
- Write a helper function which accepts a node
  - Push the value of the node to the variable that stores the values
  - If the node has a left property, call the helper function with the left property on the node
  - If the node has a right property, call the helper function with the right property on the node
- Invoke the helper function with the current variable
- Return the array of values

DFS - PostOrder



[3, 8, 6, 20, 15, 10]

Let's visualize this!

DFS - PostOrder

Steps - Recursively

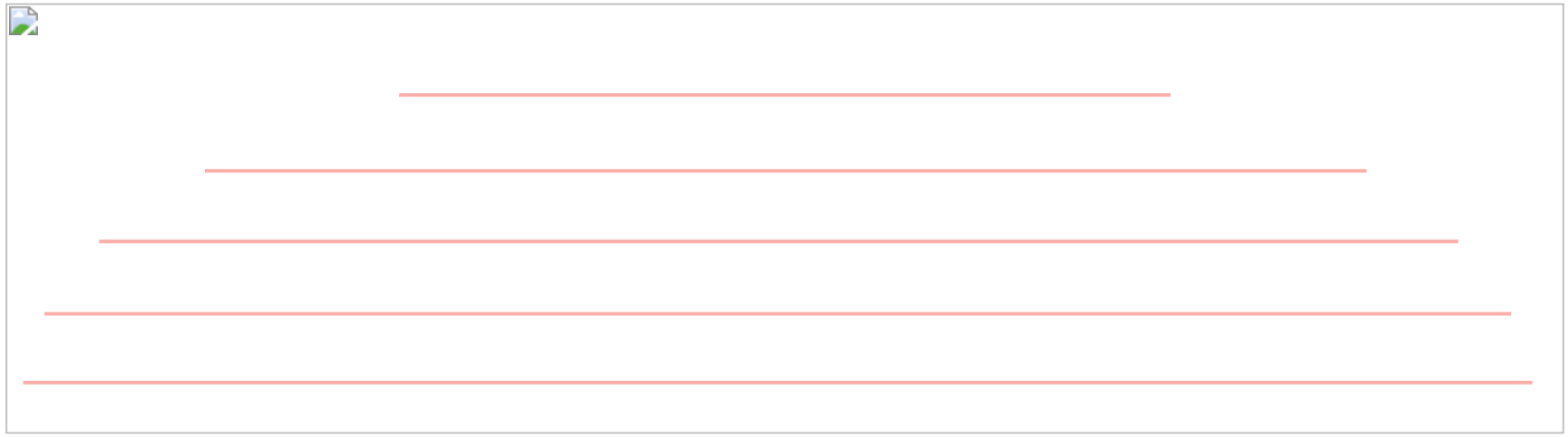
- Create a variable to store the values of nodes visited
- Store the root of the BST in a variable called current
- Write a helper function which accepts a node
  - If the node has a left property, call the helper function with the left property on the node
  - If the node has a right property, call the helper function with the right property on the node
  - Push the value of the node to the variable that stores the values
  - Invoke the helper function with the current variable
- Return the array of values

YOUR TURN

BFS?
DFS?

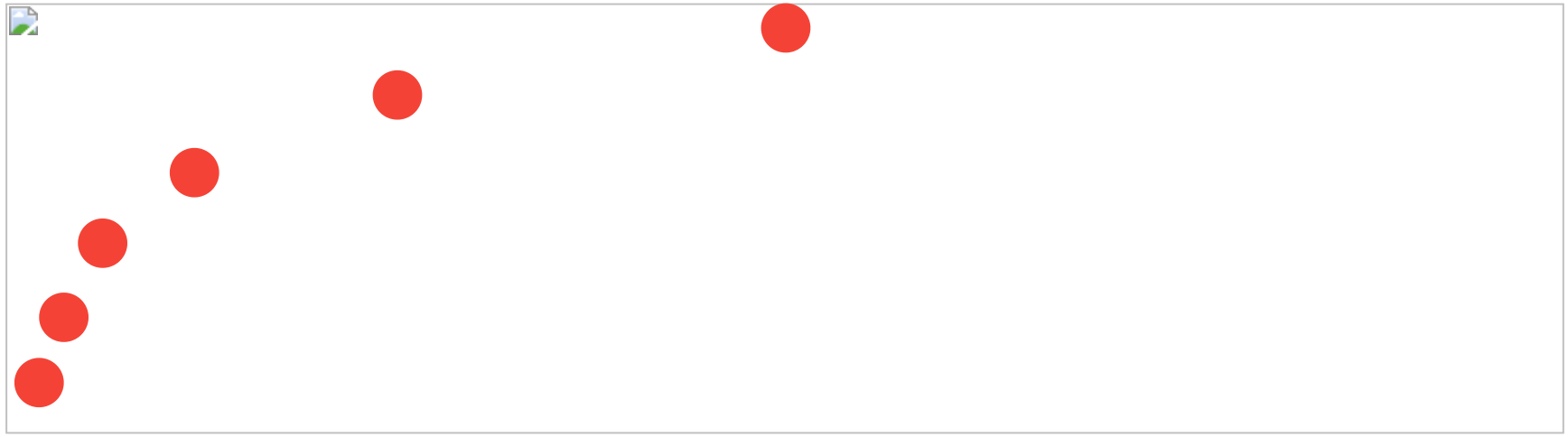
Which is better?

BREADTH FIRST



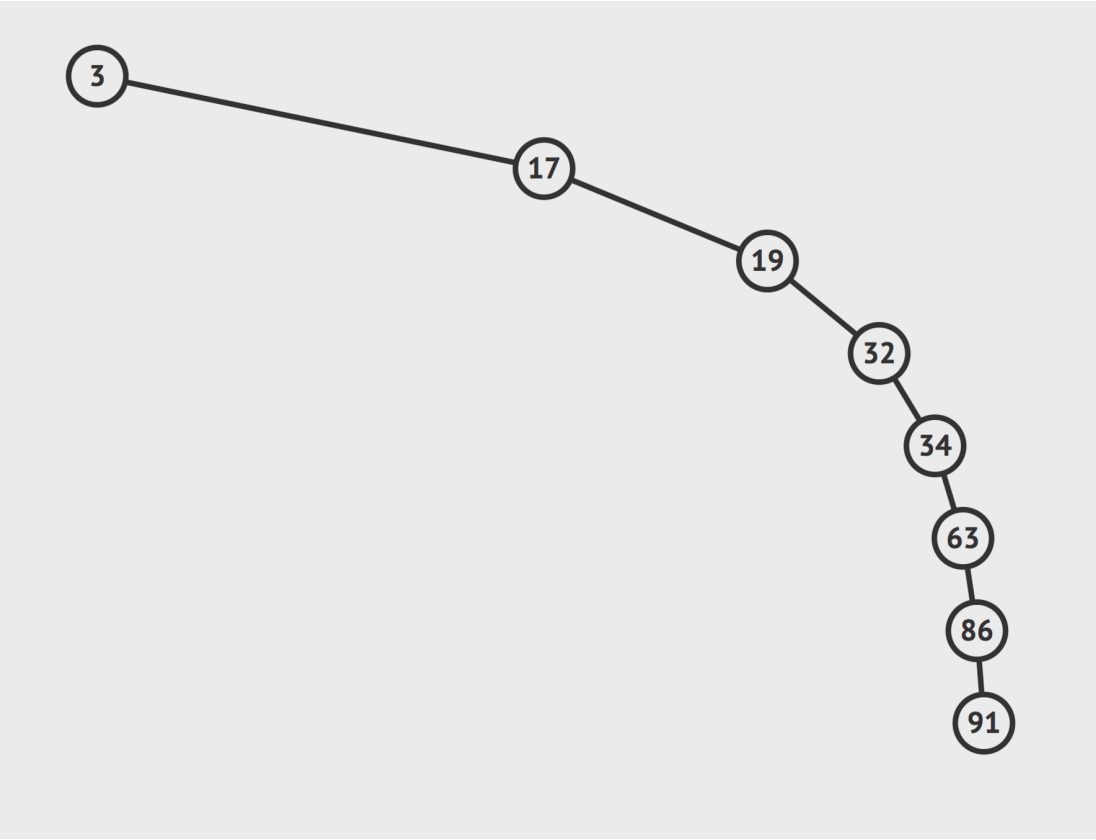
Lots of nodes to keep track of!

DEPTH FIRST



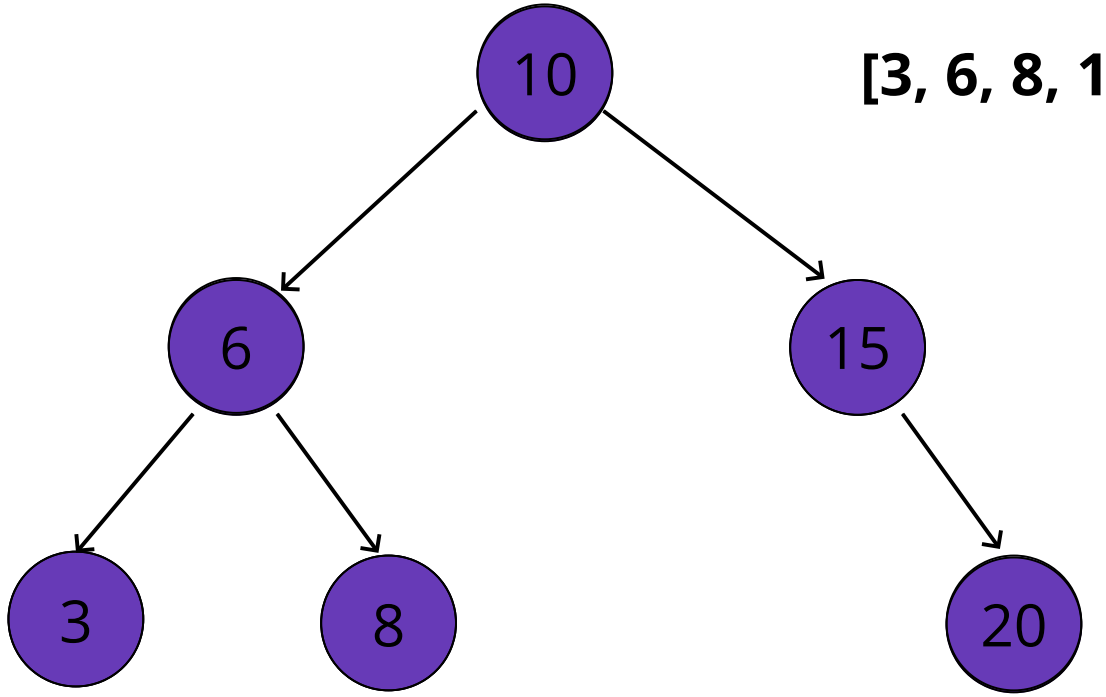
Fewer nodes to keep track of

BREADTH FIRST



Fewer nodes to keep track of

DFS - InOrder

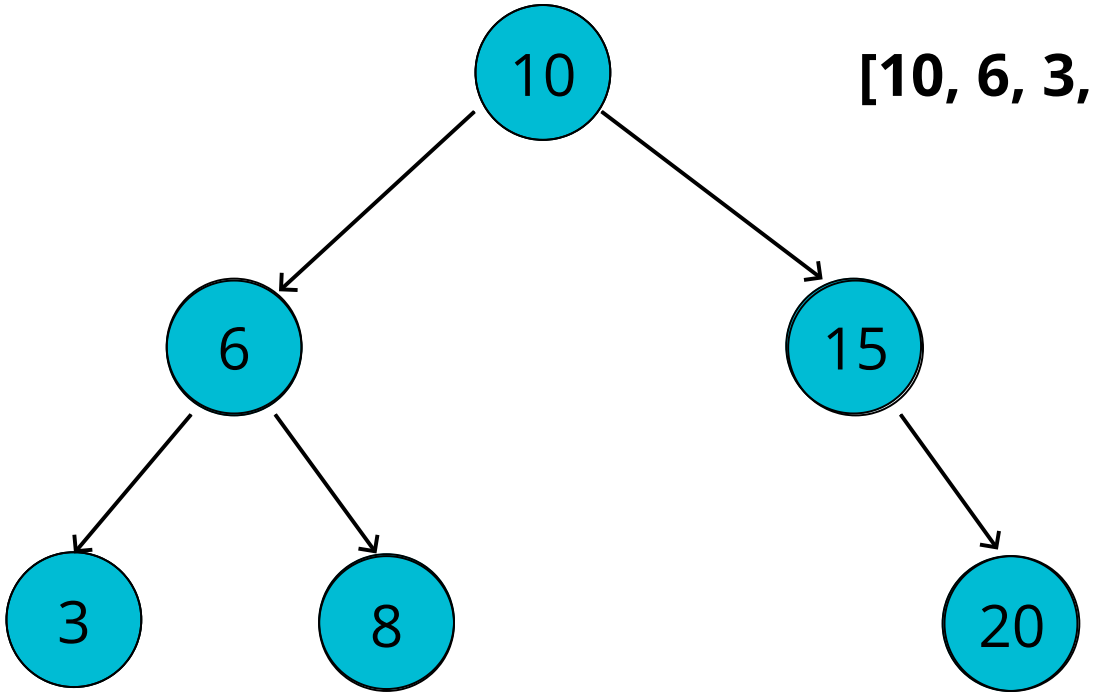


[3, 6, 8, 10, 15, 20]

Used commonly with BST's

Notice we get all nodes in the tree in their underlying order

DFS - PreOrder



[10, 6, 3, 8, 15, 20]

Can be used to "export" a tree structure so that it is easily reconstructed or copied.

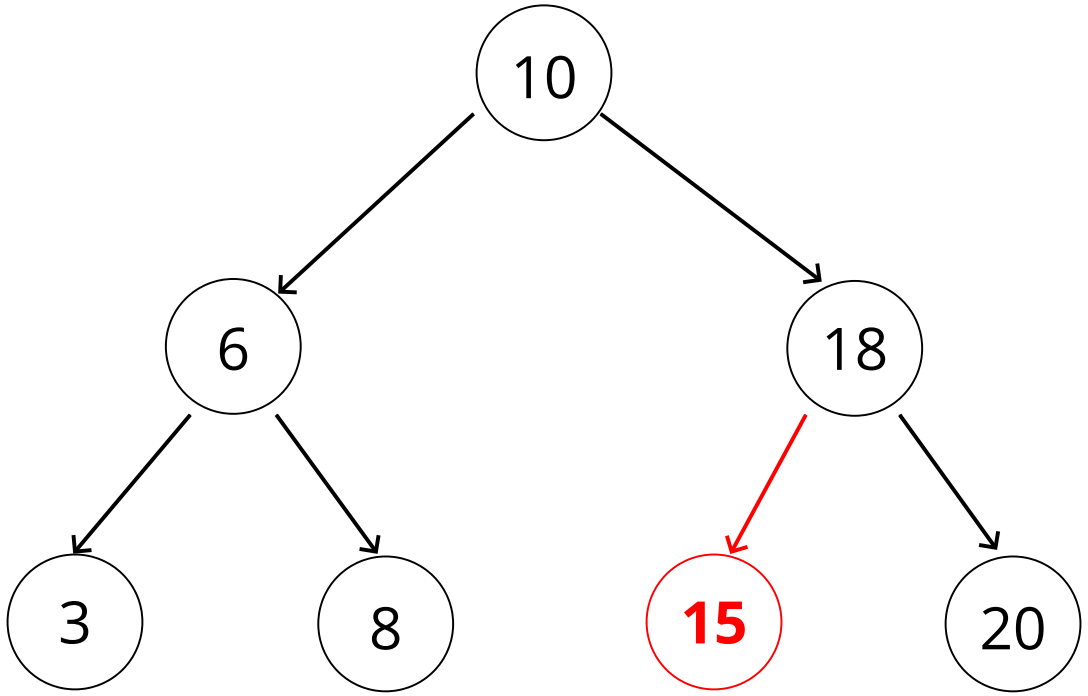
RECAP

- Trees are non-linear data structures that contain a root and child nodes
- Binary Trees can have values of any type, but at most two children for each parent
- Binary Search Trees are a more specific version of binary trees where every node to the left of a parent is less than it's value and every node to the right is greater
- We can search through Trees using BFS and DFS

Removing a Node in a BST

This one can be **tough!**

No Children, No Problem



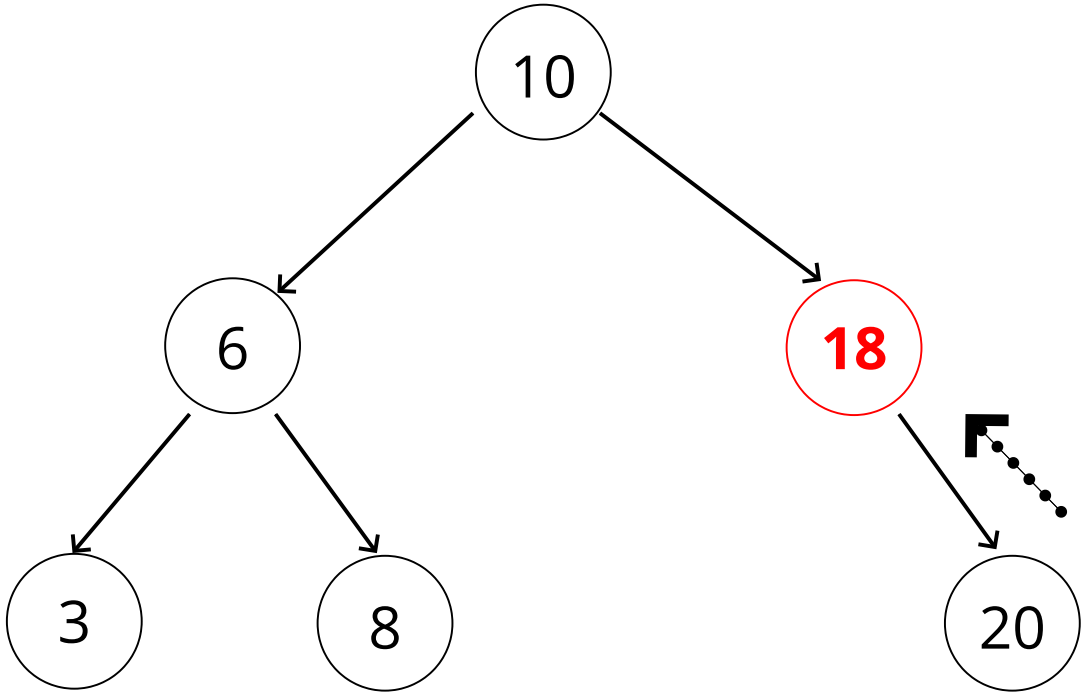
Let's visualize this!

Removing a Node - 0 children

Steps - Iteratively

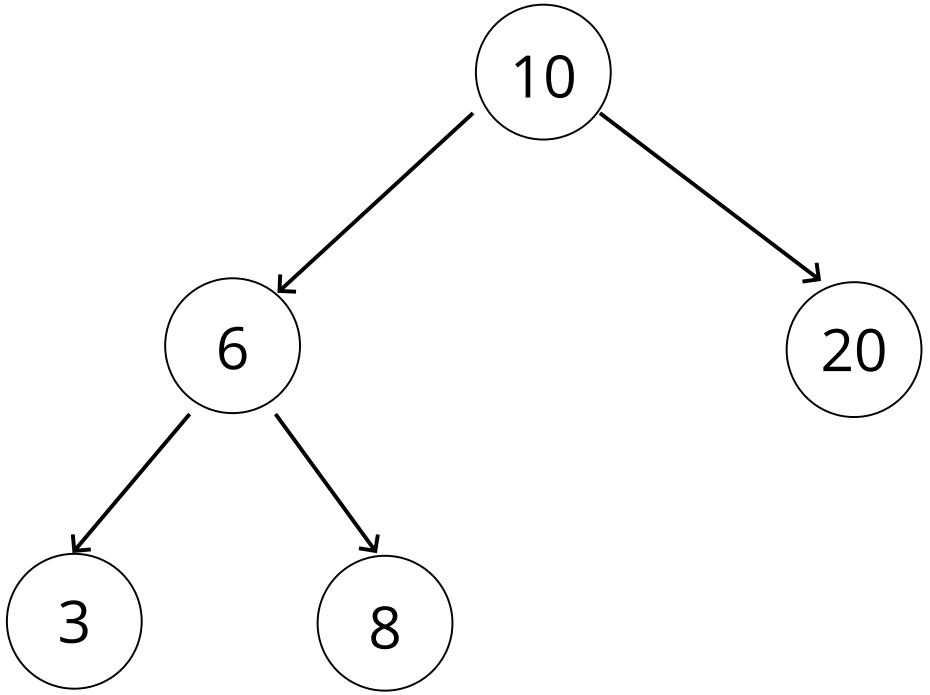
- Find the parent of the node that needs to be removed and the node that needs to be removed
- If the value we are removing is greater than the parent node
  - Set the right property of the parent to be null
- If the value we are removing is less than the parent node
  - Set the left property of the parent to be null
- Otherwise, the node we are removing has to be the root, so set the root to be null

One Child, One Problem



Let's visualize this!

One Child, One Problem



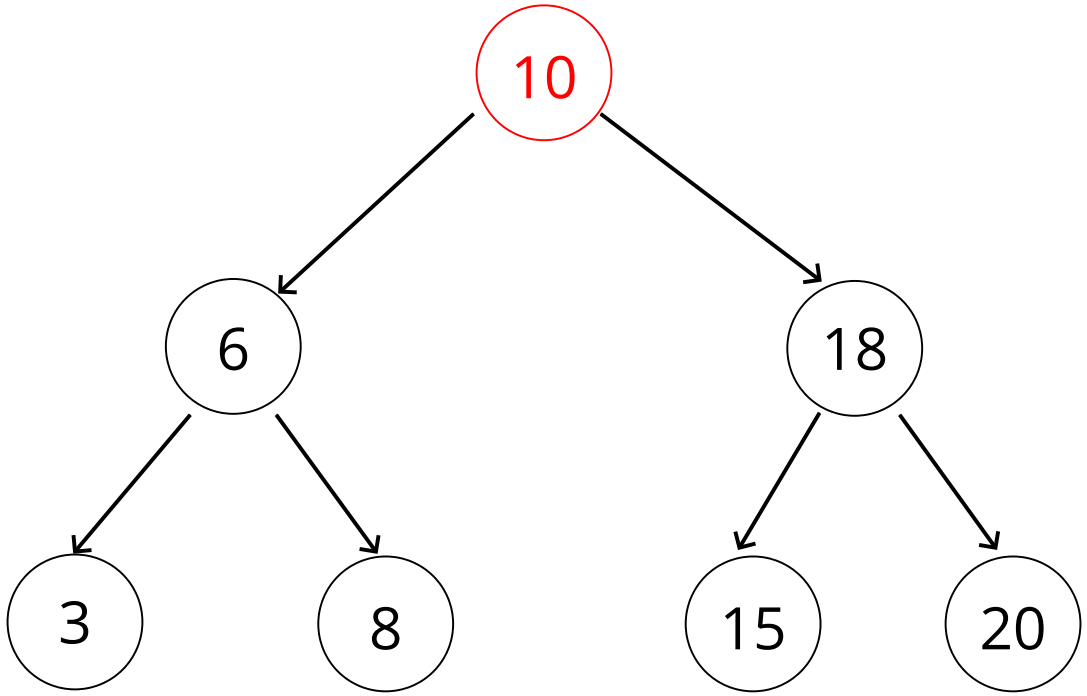
Let's visualize this!

Removing a Node - 1 child

Steps - Iteratively

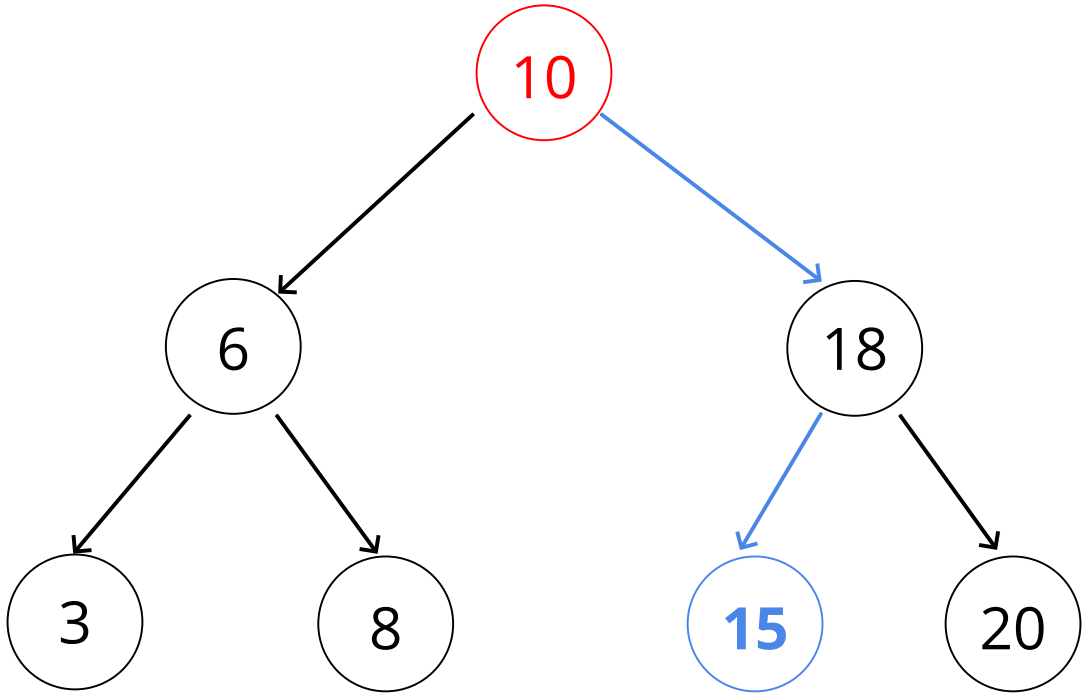
- Find the parent of the node that needs to be removed and the node that needs to be removed
- See if the child of the node to be removed is on the right side or the left side
- If the value we are removing is greater than the parent node
  - Set the right property of the parent to be the child
- If the value we are removing is less than the parent node
  - Set the left property of the parent to be the child
- Otherwise, set the root property of the tree to be the child

Two Children, More Problems

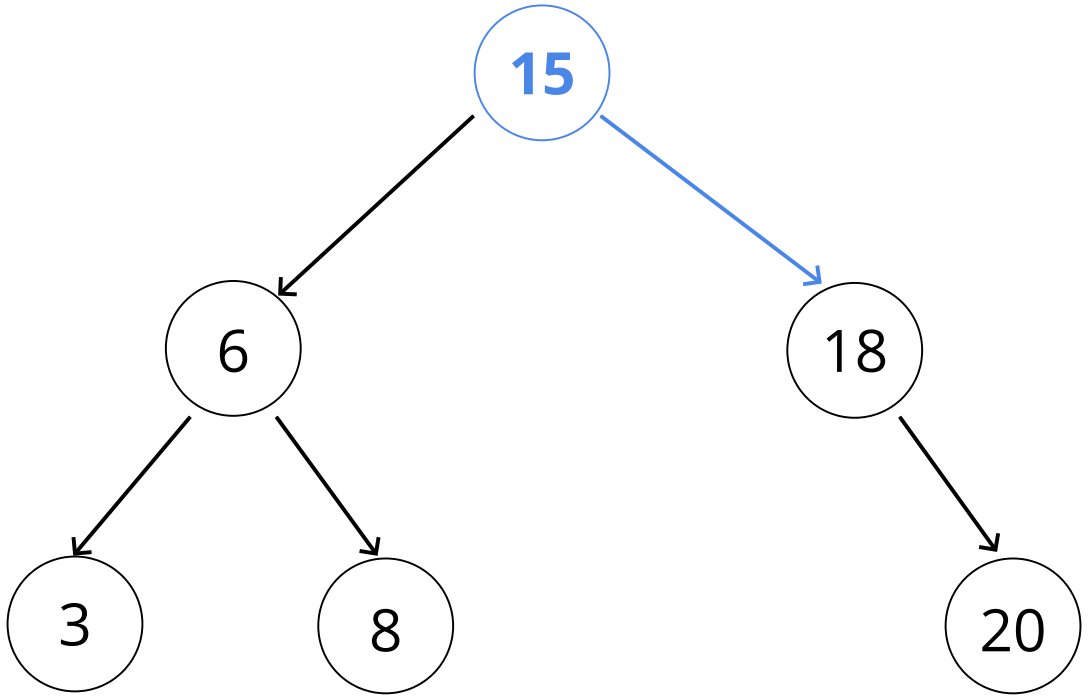


Let's visualize this!

Find the Predecessor Node!



One Right, As left as possible



Removing a Node - 2 children

Steps - Iteratively

- Find the parent of the node that needs to be removed and the node that needs to be removed
- Find the predecessor node and store that in a variable
- Set the left property of the predecessor node to be the left property of the node that is being removed
- If the value we are removing is greater than the parent node
  - Set the right property of the parent to be the right property of the node to be removed
- If the value we are removing is less than the parent node
  - Set the left property of the parent to be the right property of the node to be removed
- Otherwise, set the root of the tree to be the right property of the node to be removed

YOUR TURN