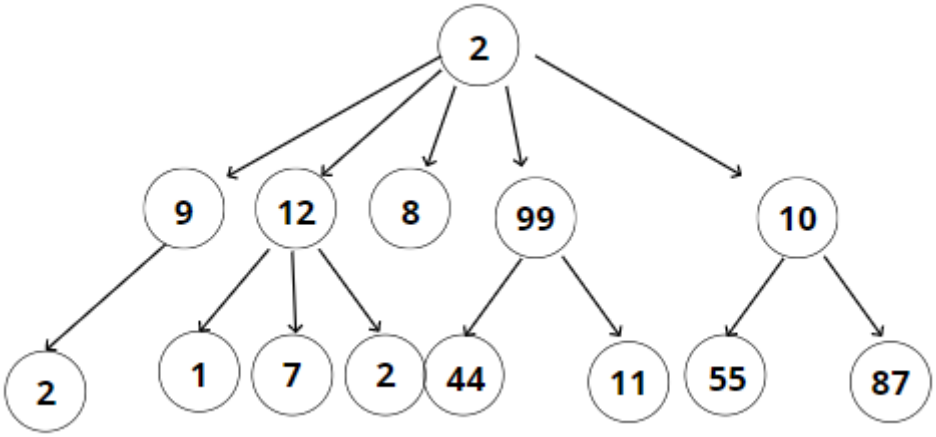
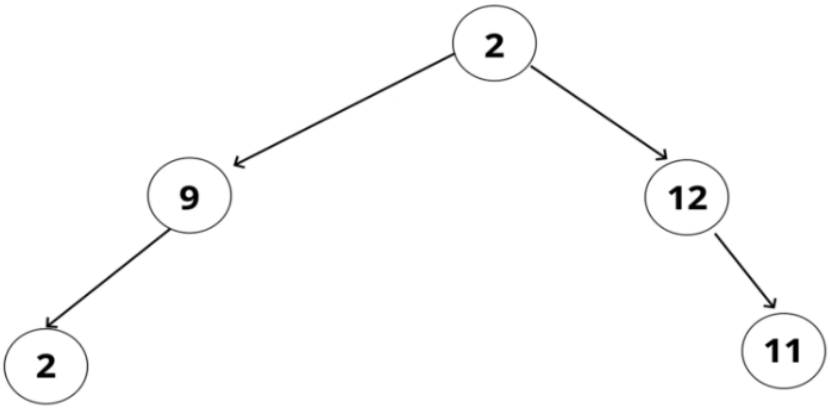
Trees

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

So from one node, we can connect many other nodes (0,1,2, … , n). We basically end up with branches.



Each circle represents a node just like linked lists. But trees are quite different from linked lists as in trees, each node can point to more than one nodes.



This is also a tree.

It doesn't have to be numbers, we can store any data we'd like.

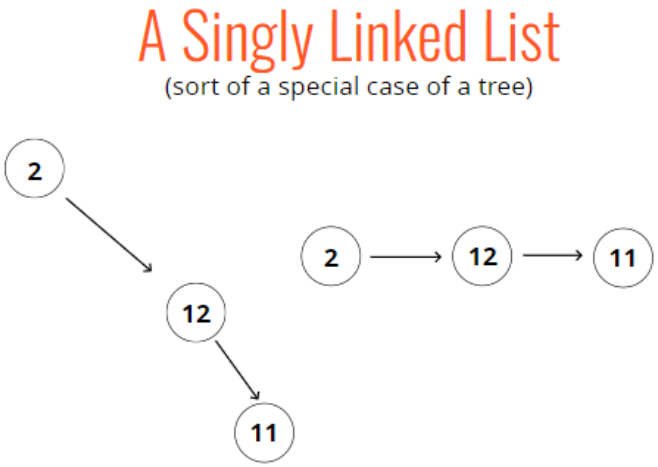
Lists - **linear**

Trees - **nonlinear**

Lists are linear. It's one thing, then the next, then the next and so on. (everything is in a line)

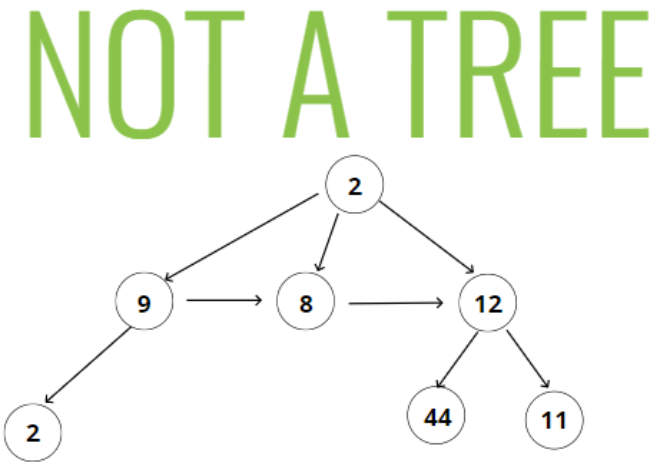
Trees are non-linear. They can branch.

You can think of singly linked list as a very special case of tree.



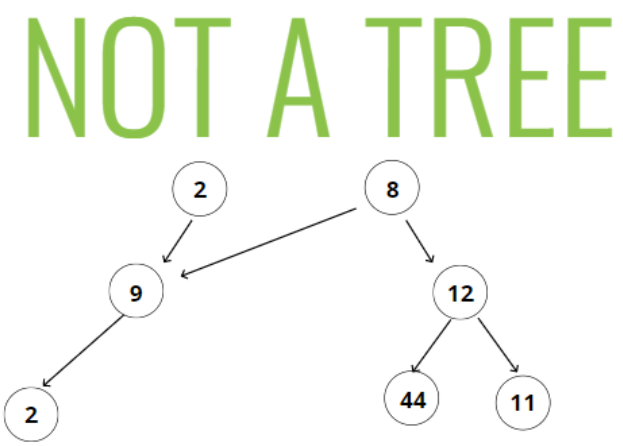
For requirement like this, we probably wouldn't use a tree, instead we'll just use linked list if this is what our data looks like. But as soon as we added a second branch, it'll become non-linear and now it's not a linked list anymore.

When we're talking about trees, there are some rules



Trees can't have nodes referencing other nodes that are not below them(not children). We can't have a child pointing to a parent or a node pointing to a sibling node.

That is actually a different data structure known as graph.



Here, there is no root node. A tree must have a single parent node at top.

Tree Terminologies:

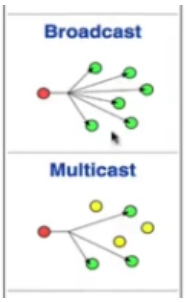
Root	-	Top Node in a tree.
Child	-	A node directly connected to another node when moving away from the root.
Parent	-	The converse notion of a child.
Siblings	-	A group of nodes with the same parent.
Leaf	-	A node with no children.
Edge	-	Connection between one node and another.

Trees has lots of different applications.

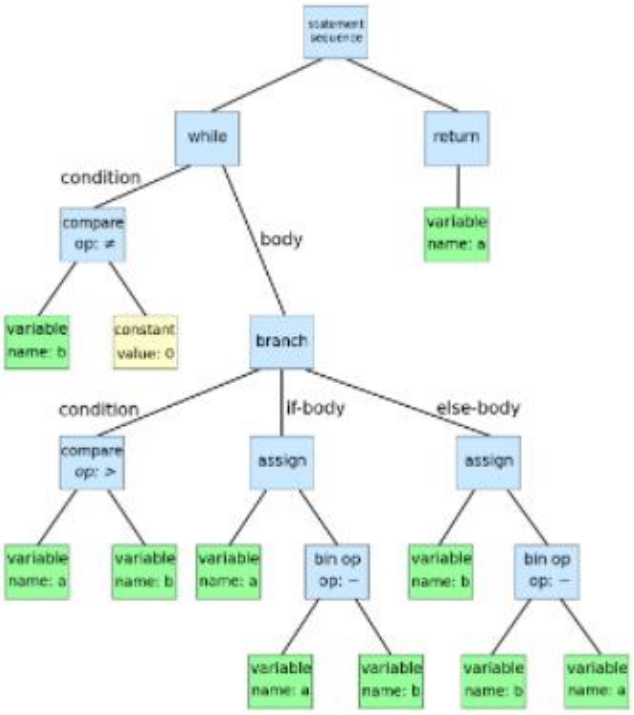
- ➔ HTML DOM
- ➔ Network Routing
- ➔ Abstract Syntax Trees
- ➔ Artificial Intelligence
- ➔ Folders in OS
- ➔ JSON response

HTML DOM

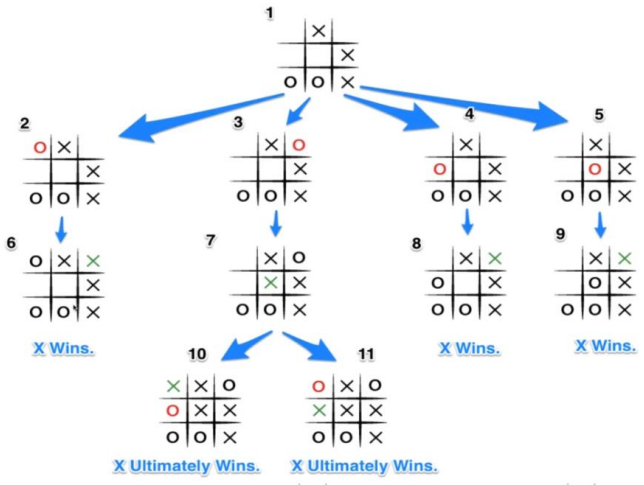
Network Routing (some logic goes into finding the shortest path)



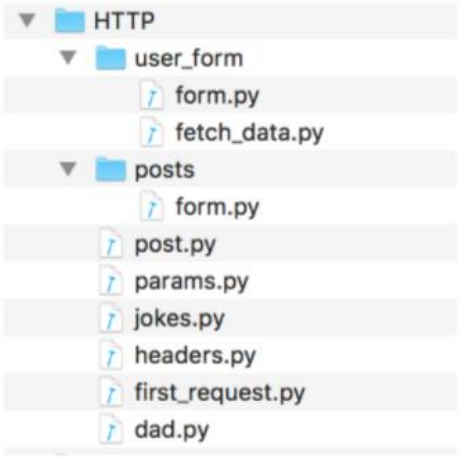
Abstract Syntax Trees (way of describing a syntax of a programming language using a tree structure)



Artificial Intelligence (decision tree)



Folders in OS



Binary Trees

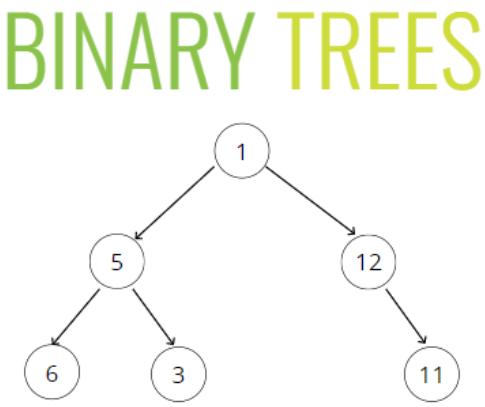
There are many varieties of trees in data structure.

Some will be faster at searching or insertion or they excel in storing data in a particular format etc etc. So Binary Trees are one of the groups of tree data structure.

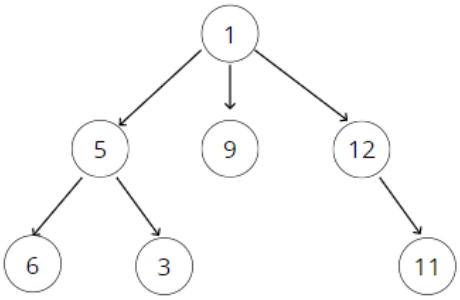
We've seen generic trees where each node could have any number of child nodes. There's no constraint on how many?

A binary tree on the other hand has a special condition.

Each node can have at most two children. So it can have 0, 1 or 2 children.



NOT A BINARY TREE



Alhough it's a valid tree in general. But it is not a binary tree as as a node can't have three children.

There may have a ternary tree where each node could have a max of 3 children or quadinary (with max of 4) and so on, but they're not commonly talked about because binary trees have some special properties that make them easier to navigate.

BINARY TREES

Lots of different applications as well!

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

There are many types of binary trees, one of which is the binary search tree.

Binary Search Trees

Special type of a binary tree which is a special type of tree and they excel at searching. Used to store data that can be compared.

Additional to having binary trees properties in it, binary search trees are sorted in a particular way.

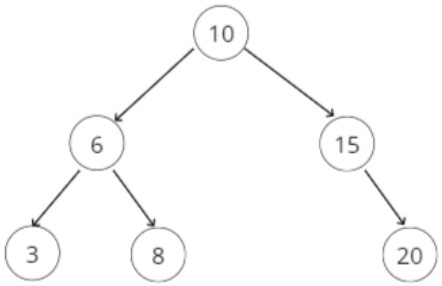
If we take any node on the tree, every item that is less than this node is located to the left of that node and every item that is greater than that node is located to the right.

And it repeat that on each child node

So, all the items at right are the ones that are greater than that node and all the items at left are the ones less than that node.

And this is what makes a binary search tree a binary search tree rather than just a binary tree. Data is kept in a particular order that every child to the right is greater the parent node and every child to the left is less than the parent node.

BINARY SEARCH TREES

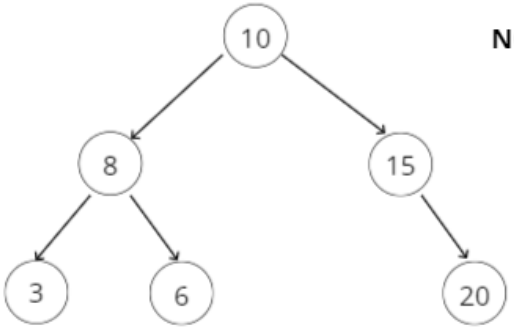


Here, in case of 15, we only have one node but as 20 is greater than 15 so it's located to the right of it.

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

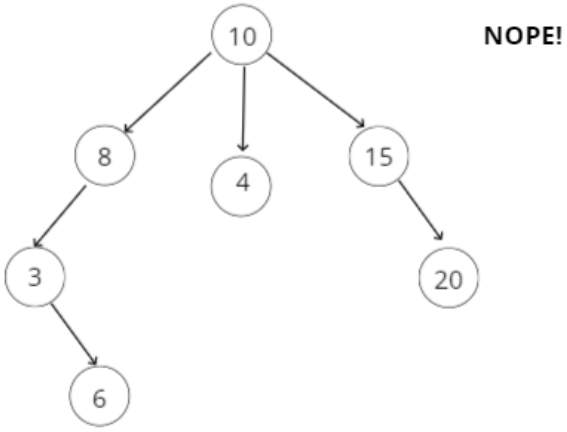
Is this a valid BST?



NOPE

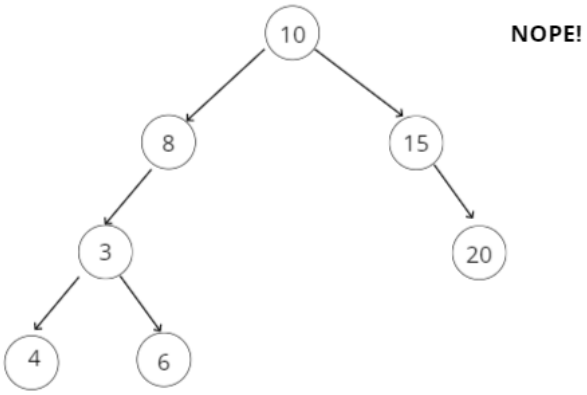
WHY NOT ?

6 is not greater than 8, still it's to the right of 8.



WHY NOT ?

In BST, a node can't have 3 child nodes.



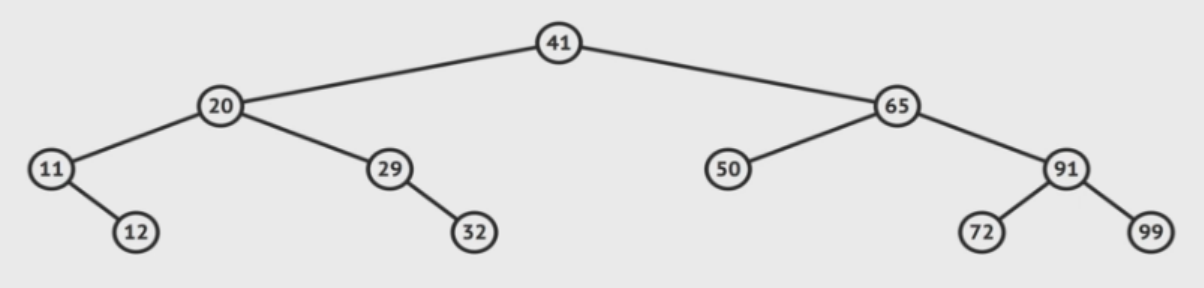
WHY NOT ?

6 is not greater than 3, still it's to the right of 3.

Searching a Binary Search Tree

The way we order a BST makes it very fast to look things up. Also it makes it very easy to insert things to find a place for a node. So the search part becomes very fast compared to an unsorted tree.

Suppose that we have to find 72.



We check Is 72 greater or less than 41(root node)?

It’s greater than root so we look at the right child and forgot about the left side.

Then we check again and it’s again greater than 65, so we again look at the right child.

Then we check again and now it’s less than 91, so we look at the left child and there it is.

Are duplicates allowed in binary tree?

For a node x, with key k, every key in x's left subtree is less than ***or equal to*** k, and every key in x's right subtree is greater than or equal to k. Note that the definition permits duplicate keys. **Some BSTs don't permit duplicate keys**. Whether to permit duplicate keys depends upon the application that uses the BST.

If we wanna keep track of the duplicate node, you could add a frequency/count to the node and you could increment that.

```
      10(2)
   5      13
 2   7  11  16(5)
```

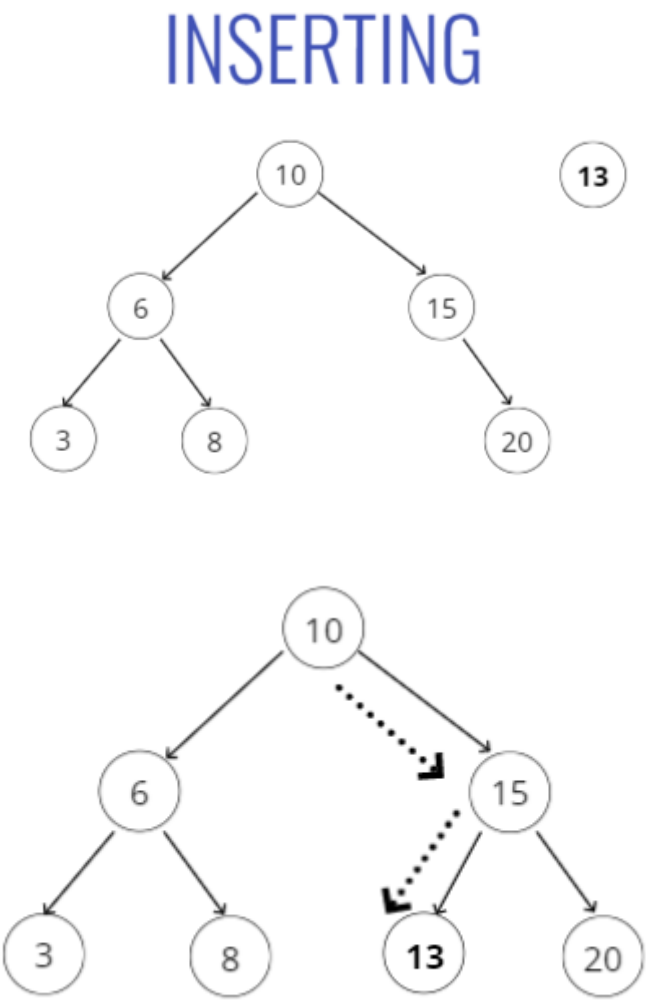
```
class Node{
    constructor(value) {
        this.val = value;
        this.left = null;
        this.right = null;
    }
}
```

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

```
let tree = new BinarySearchTree();
tree.root = new Node(10);
tree.root.right = new Node(15);
tree.root.left = new Node(5);
tree.root.left.right = new Node(7);     // 5-10
tree.root.left.left = new Node(3);      // <5
```

```
//          10
//      5       15
//    3   7
```

Its a huge pain to insert things like this, so we're going to add a method called insert and it'll figure out where something goes based off of comparisons.



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

```
insert(value) {
    let newNode = new Node(value);
    if (!this.root) {
        this.root = newNode;
        return this;
    }

    let currentNode = this.root;

    while (true) {
        if (value == currentNode.val) return undefined;
        if (value < currentNode.val) {
            // go to left
            if (currentNode.left == null) {
                currentNode.left = newNode;
                return this;
            }
            currentNode = currentNode.left;
        } else {
            // go to right
            if (currentNode.right == null) {
                currentNode.right = newNode;
                return this;
            }
            currentNode = currentNode.right;
        }
    }
}
```

Finding a Node in a BST

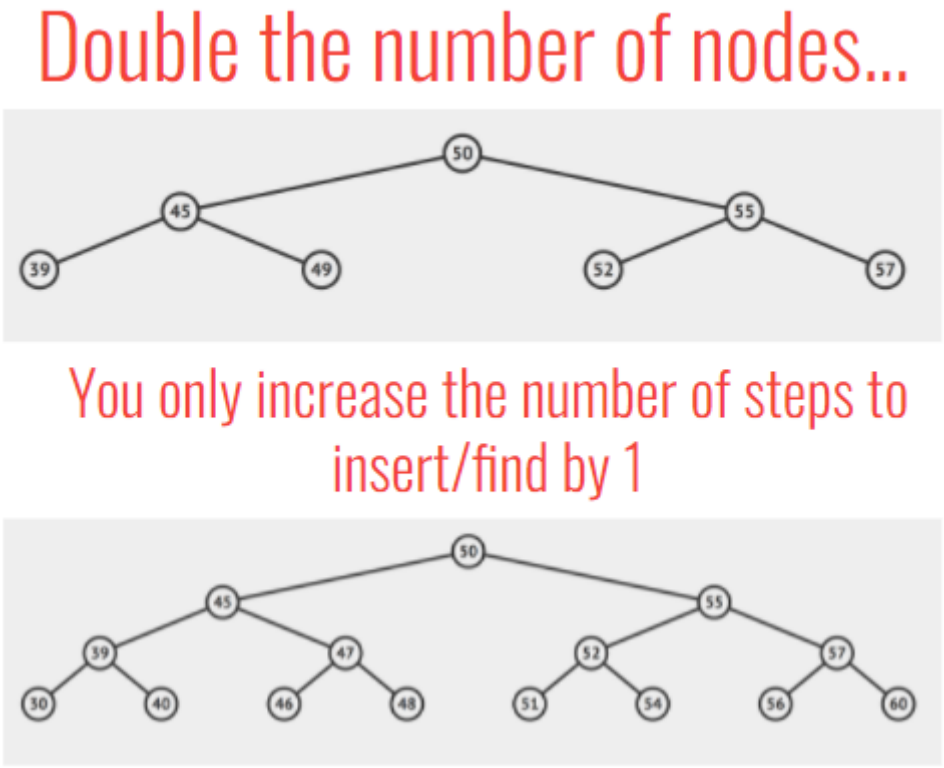
Steps - Iteratively or Recursively

- 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!

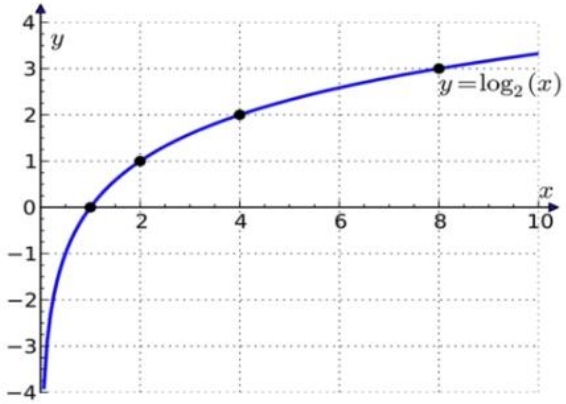
```
find(value) {
    if (!this.root) return false;
    let currentNode = this.root;
    while (true) {
        if (value < currentNode.val) {
            if (currentNode.left == null) {
                return undefined;
            }
            currentNode = currentNode.left;
        }
        else if (value > currentNode.val) {
            if (currentNode.right == null) {
                return undefined;
            }
            currentNode = currentNode.right;

        } else {
            return currentNode;
        }
    }
}
```

Time Complexity **(BST)**

Best case	O(log n)
Average case	O(log n)
Worst case	O(n)



Because the way we keep it sorted, as the tree doubles, we only increase the no of steps by 1. It's LOG base 2 relationship.



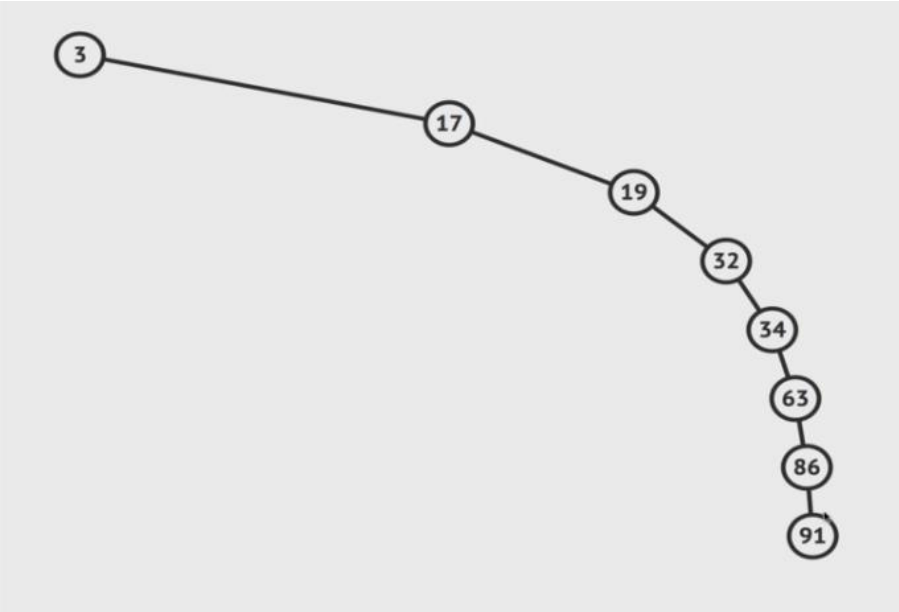
2x number of nodes: 1 extra step

4x number of nodes: 2 extra steps

8x number of nodes: 3 extra steps

Why worst case is O(n) ?

Some BST configurations are very slow compared to the one's we've been looking at.



That's a valid BST.

In this case, the no of steps it's going to take to insert/search is going to grow as the no of nodes grow.

That would be O(n) if you had a completely one sided tree like this.

Solution?

Just don't use a BST for this but if you really needed this as a tree, then you could rewrite this BST where you pick a different root, lets say 34 and then restructure the entire tree.