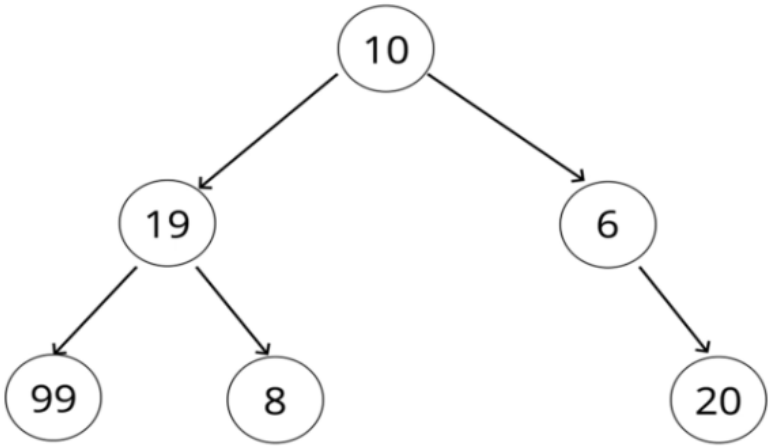
Tree Traversal

Any tree (BST, plain binary tree, ternary tree or any kind of tree) having any no of nodes. How do we visit every node one time?

It's very different from how we did it in SLL or DLL.



It's not a Binary Search Tree, it's just a Binary tree. If we have to find a specific node we can't use our shortcut like we did in BST as there's no order. We need to visit every single node. How do we do that?

There's so many ways of doing that.

Two main approaches of traversing a tree.

1- Breadth-First Search
2- Depth-First Search
    - InOrder
    - PreOrder
    - PostOrder

BFS → [10, 19, 6, 99, 8, 20]

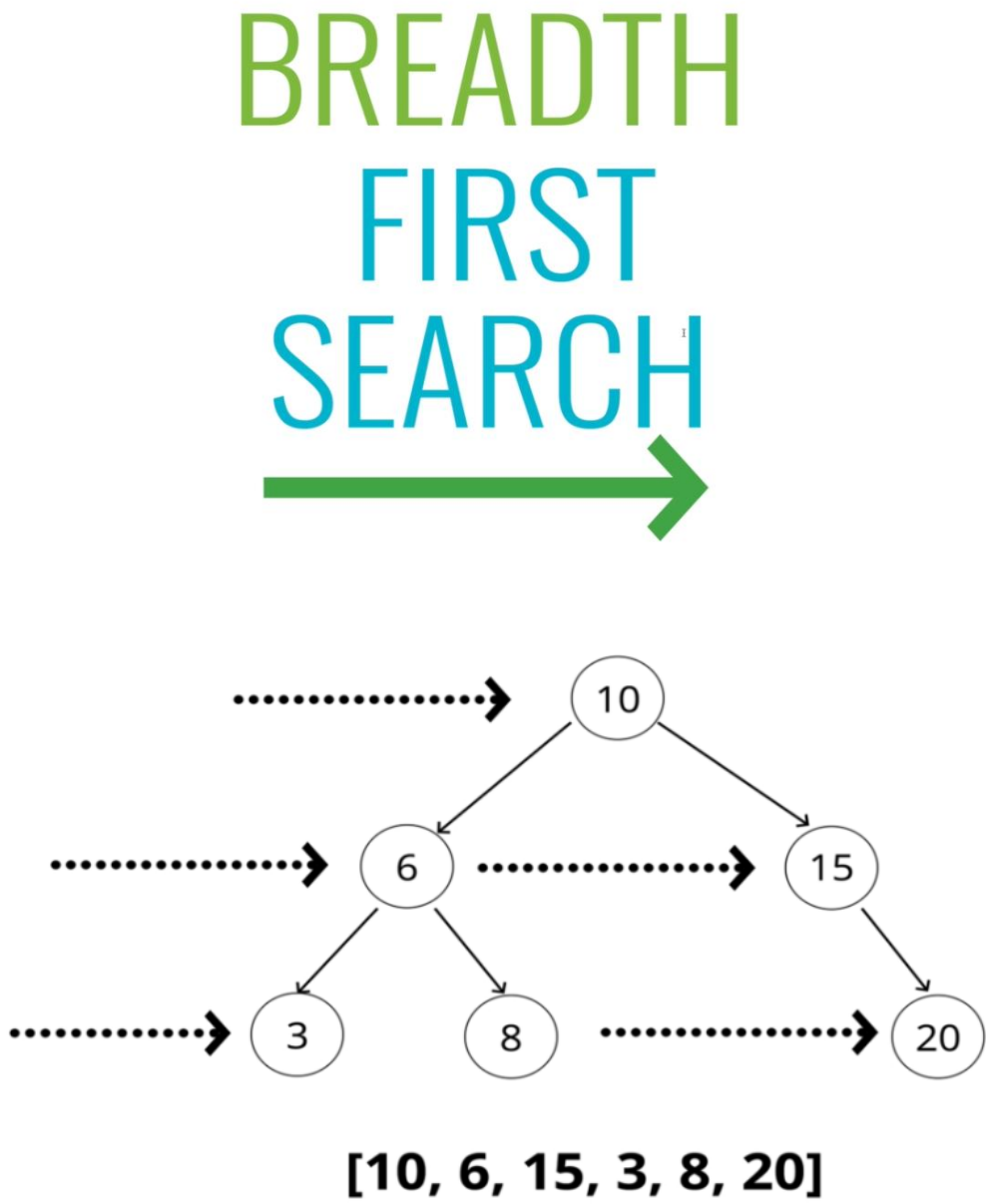
DFS PreOrder → [10, 19, 99, 8, 6, 20]

DFS PostOrder → [99, 8, 19, 20, 6, 10]

DFS InOrder → [99, 19, 8, 10, 6, 20]

These different strategies actually have an impact depending on what the tree looks like, how the data is distributed.

Breadth-First Search



We’re going from Left to Right here but it doesn’t matter. Whats more important is that we’re working horizontally.

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

```
      Breadth First Search

              10
          6       15
        3   8       20

visited []
queue [10]

visited [10]
queue [6, 15]

visited [10, 6]
queue [15, 3, 8]

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

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

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

visited [10, 6, 15, 3, 8, 20]
queue []
```

```
BFS() {
    let queue = [];
    let result = [];
    let node;

    if (queue.length) queue.push(this.root);

    while (queue.length) {
        node = queue.shift();

        result.push(node.val);
        if (node.left) queue.push(node.left);
        if (node.right) queue.push(node.right);
    }
    return result;
}
```

Depth First Search

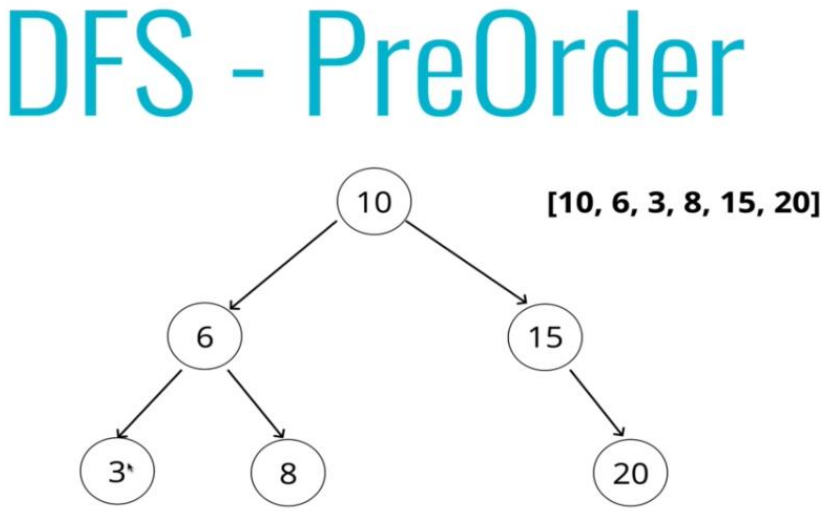
There are three main orders for DFS.

- ➔ PreOrder
- ➔ PostOrder
- ➔ InOrder

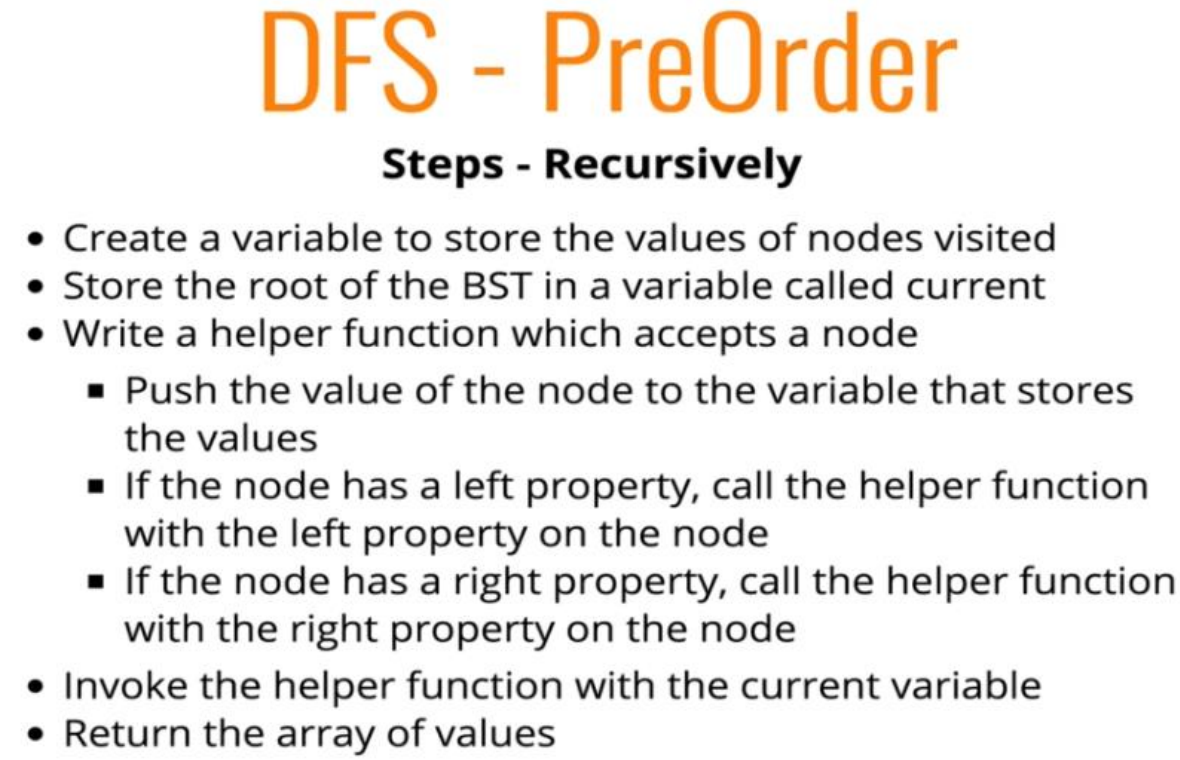
Persue all nodes they visit vertically down to the end of the tree before visiting sibling nodes.

We're going to traverse down until we hit the end of the tree at some point and then from there there's a couple of options as far as the order that we do things.

PreOrder



A high level overview of way this actually works is that we visit a node, then we traverse the entire left side, and then traverse the entire right side.



Basically means that we're going to visit the node first, then we traverse the entire left side, then we traverse the right and that's true for any node.

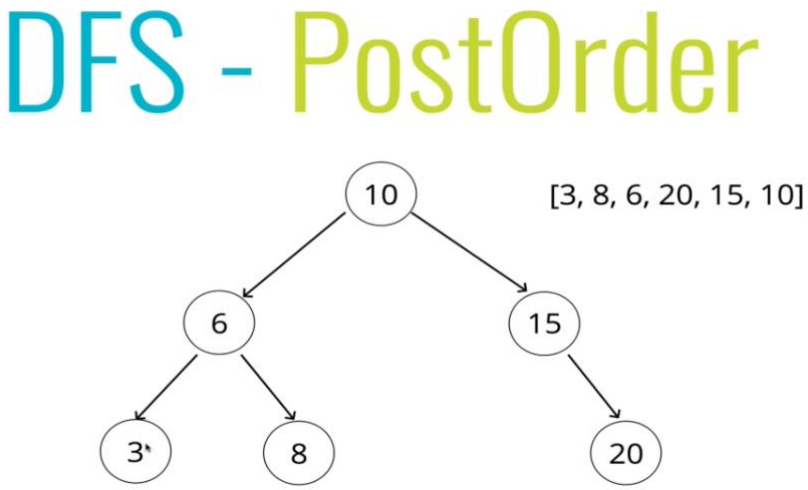
```
DFS_pre_order() {
    let result = [];
    let current = this.root;

    function traverse(node){
        result.push(node.val);
        if(node.left) traverse(node.left)
        if(node.right) traverse(node.right)
    }
    traverse(current)

    return result;
}
```

The order of the lines in our traverse function here is very important as it changes our output.

PostOrder

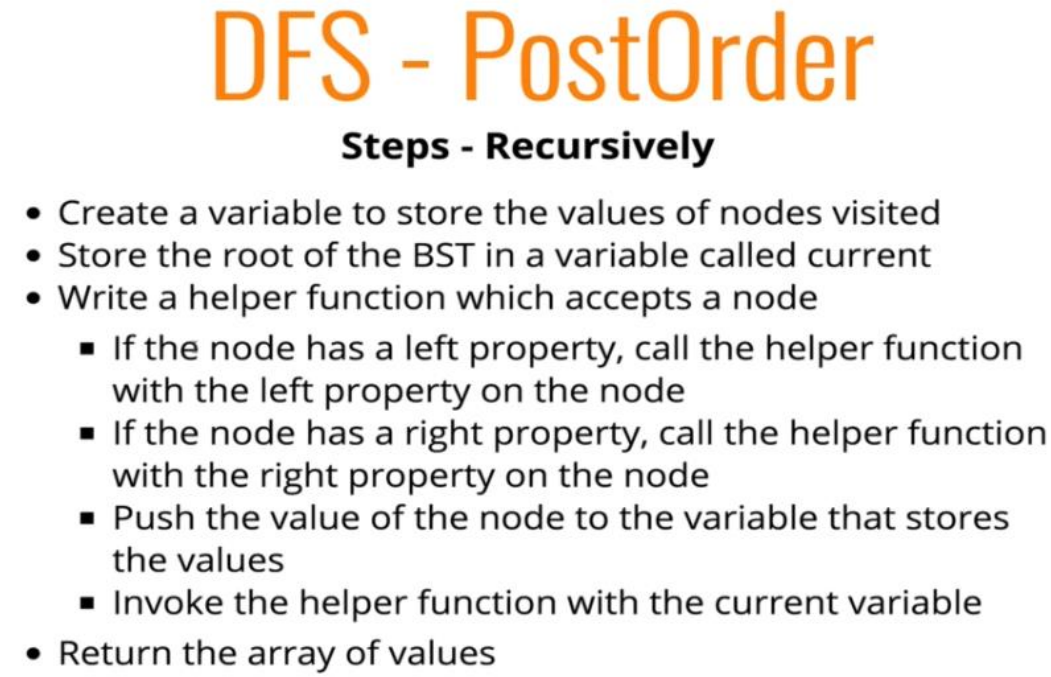


In preOrder, we visit the node first, then look at the left (and when the entire left is done) then we look at the right.

In postOrder, we visit the node after we have looked at the left and the right. So we explore everything first, we traverse the entire tree or the entire branch from a given node (the left and the right), then we visit the node.

So, we started at 10 but we're not gonna visit it, we're gonna look at the left, we get 6, then we're gonna look at the left, there's nothing on left so we check at right, theres nothing there too. Then we're gonna add 3 and then check to the right of 6, we find 8 (check it has no left or right) so we add 8 then maving up add 6, then we don't visit 10 yet as we've only done the left side of 10. Now we do the right side, we get 15, there's no left to 15 so we check right, find 20 there, there's no left or right to 20 so we add 20 then 15 and then afer completing the right side we add 10.

Root is the last thing that is visited. For any node, we explore all of its children before we actually visit the node.

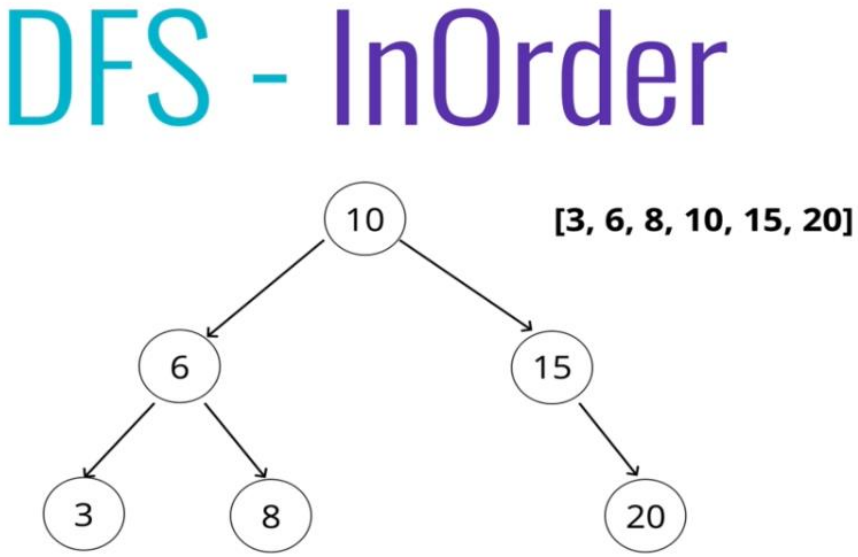


Same psuedo code except the order is different.Rather than pushing first then doing the left and the right, we're going to explore the left, explore the right, and then push the value in.

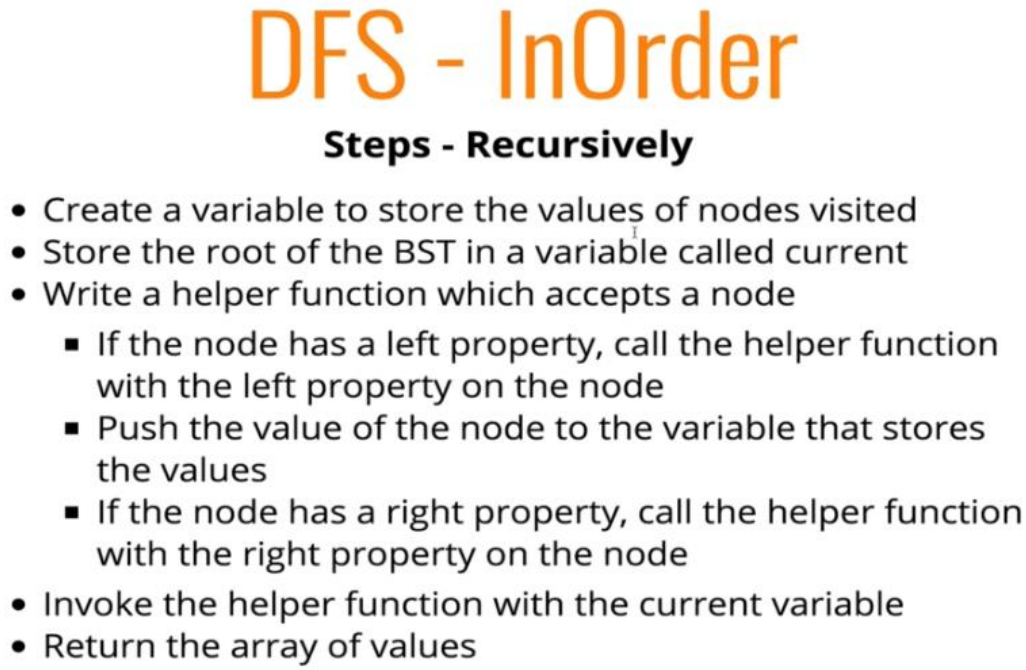
```
DFS_post_order() {
    let result = [];
    let current= this.root;
    function traverse(node){
        if(node.left) traverse(node.left)
        if(node.right) traverse(node.right)
        result.push(node.val);
    }
    traverse(current);
    console.log(result);
    return result;
}
```

InOrder

In inOrder, we we're first going to traverse the entire left side, then visit the node, and then traverse the entire right side.



Starting from 10, we visit left, found 6, then visit left of 6, found 3, there's no left to 3 so we visit 3, then we look at the right of 3 there's nothing to it's right so we move up and visit 6, then we go to 8, there's no left to 8 so we visit 8, then we look at the right of 8 there's nothing to it's right so we move up and up again(as we've already visit 6), visit 10 then look what's to the right of it, we found 15, we check its left there's no left of 15 so we visit 15, then look for right, found 20, as there's no left to 20 so we visit 20, then check for right, there's no right to 20 so we move up and up again, we're now at our root so we finish.



Same psuedo code except the order is different. Rather than pushing first then doing the left and the right, OR exploring the left, explore the right, and then push the value in,

We explore the left side, visit the node and then explore the right.

```
DFS_in_order() {
    let result = [];
    let current= this.root;
    function traverse(node){
        if(node.left) traverse(node.left)
        result.push(node.val);
        if(node.right) traverse(node.right)
    }
    traverse(current);
    console.log(result);
    return result;
}
```

COMPARISON

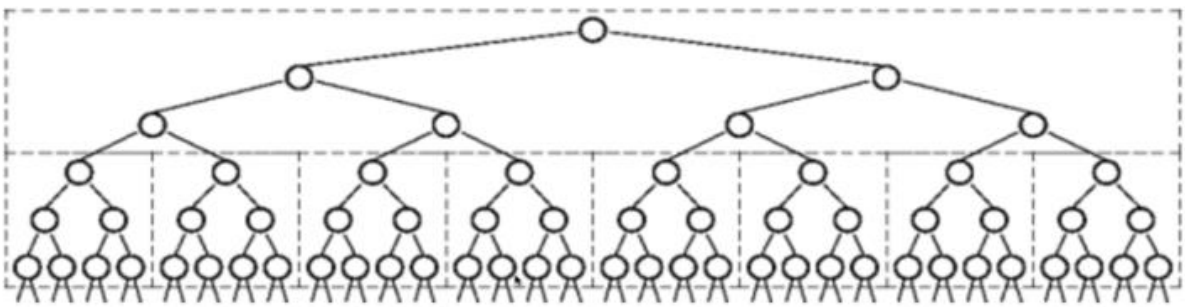
So which one is better? Well, it depends on your situation!

Time Complexity doesn't matter as we're visiting each node once so time complexity of both BFS and DFS will be same.

Whether BFS or DFS is better in your case, depends on the structure of the tree.

WIDE tree, DFS is better.

If you have a tree that is fully fleshed out, like this and we use BFS,



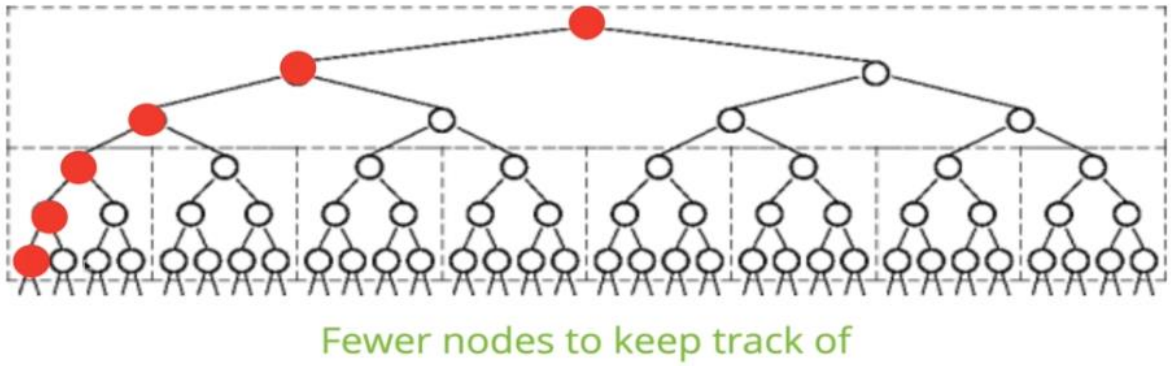
Using BFS here(wide trees) means, lots of nodes to keep track of!

If we're working with BFS here, we'd be storing all of the children nodes we have yet to visit in queue.

We're gonna visit one node and store left and right to the queue, and go to the other node and store their left and right and so on, our queue grows, and it stores a bunch of nodes.

As the level increases, we're going to have to store a ton of data in our queue just in memory. So space sompexity wise, BFS would be a bad option here as the space that's used up in this scenario could be a lot more.

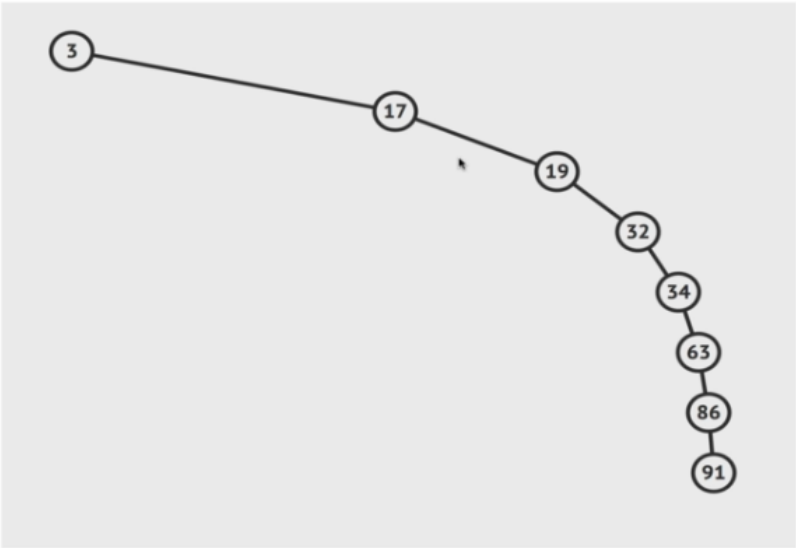
While if we use DFS here,



Using DFS here(wide trees) means, fewer nodes to keep track of!

As we're using DFS, we're not storing all of those nodes across across the tree. We only have to keep track of nodes in a given branch all the way down to the end. So it would use less space.

If we have a slim tree of huge height like this,



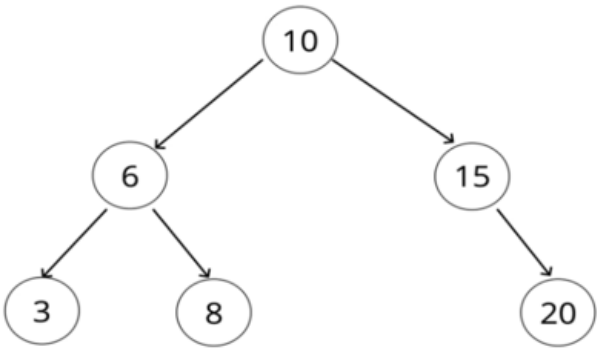
Using BFS here(slim and large height trees) means, fewer nodes to keep track of!

Here, using BFS, the space we take up in memory is basically nothing compared to the previous.

Using DFS here(slim and large height trees) means, lots of nodes to keep track of!

But if we're using DFS, that means ton of levels to go down, and keep every other level obove it in memory, so that would not be good space-wise.

Variants of Depth first search:



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

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

If you want to clone a tree, store it in a file or database where you need to recreate this structure.

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

Used commonly with binary search trees.

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

Depending on what you're doing, this could be useful.

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

Couldn't think of a particular situation.

Choosing DFS variants wouldn't be the bigger concern here, main concern is to choose between BFS and DFS.

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