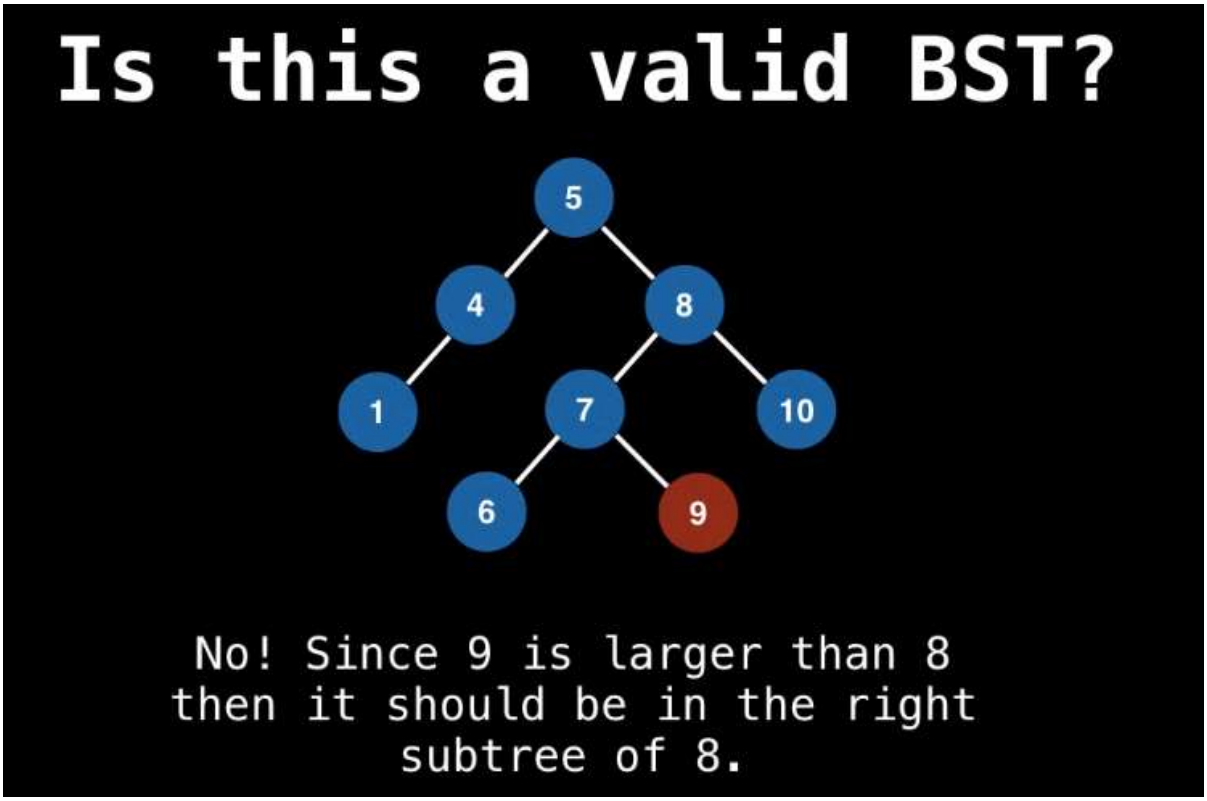
Binary Trees and Binary Search Trees (BST)

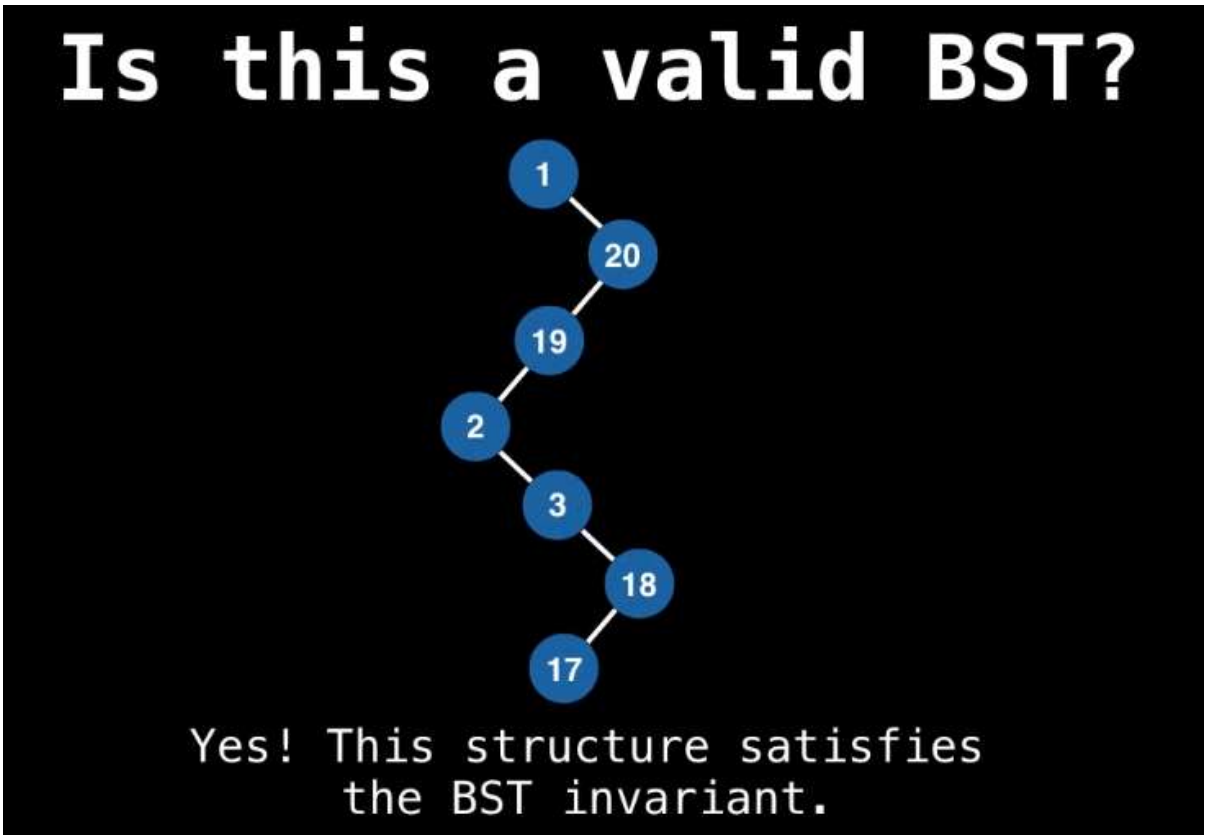
A **binary tree** is a tree for which every node has at most two child nodes.

A **binary search tree** is a binary tree that satisfies the **BST invariant**: left subtree has smaller elements and right subtree has larger elements.

BST operations allow for duplicate values, but most of the time we are only interested in having unique elements inside of our tree.

Examples:





Complexity of BSTs:

Operation	Average	Worst
Insert	O(log(n))	O(n)
Delete	O(log(n))	O(n)
Remove	O(log(n))	O(n)
Search	O(log(n))	O(n)

BST Insertion

When inserting an element in BST, we want to compare its value to the value stored in the current node we are considering. Based on that, we will decide on one of the following:

- Recurse down the left subtree (< case)
- Recurse down the right subtree (> case)
- Handle finding a duplicate value (= case)
- Create a new node (found a null leaf)

On average, the insertion time will be logarithmic. But in the worst case, this could degrade to **linear** time:

- insert(1)
- insert(2)
- insert(3)
- insert(4)
- insert(5)

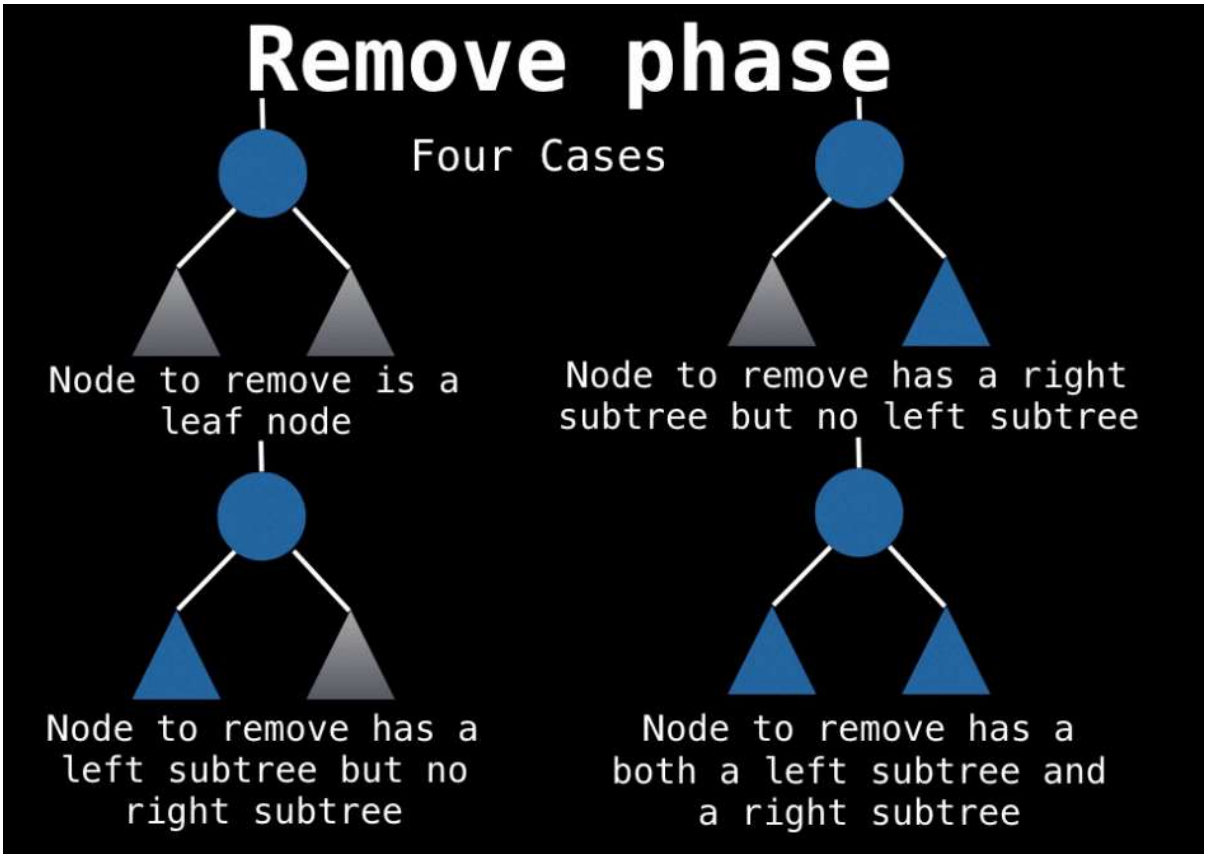
- insert(6)

This is the reason why **Balanced Binary Search Trees** were invented.

BST Removal

Removing from a BST can be seen as two step process:

1. Find the element we want to remove (if it exists) One of the following things can happen here:
    - We hit a **null node** at which point we know that this value does not exist within our BST.
    - Comparator value **equal to 0** (we have found it)
    - Comparator value **less than 0** (the value, if it exists, is in the left subtree)
    - Comparator value **greater than 0** (the value, if it exists, is in the right subtree)
2. Replace the node we want to remove with its successor (if any) to maintain the **BST invariant**



Case 1: Leaf Node

No side effects

Case II and III: either the left/right child node is a subtree

The successor of the node we are trying to remove in these cases will be the **root node of the left/right subtree**.

It may be the case that you are removing the root node of the BST in which case its immediate child becomes the new root as you would expect.

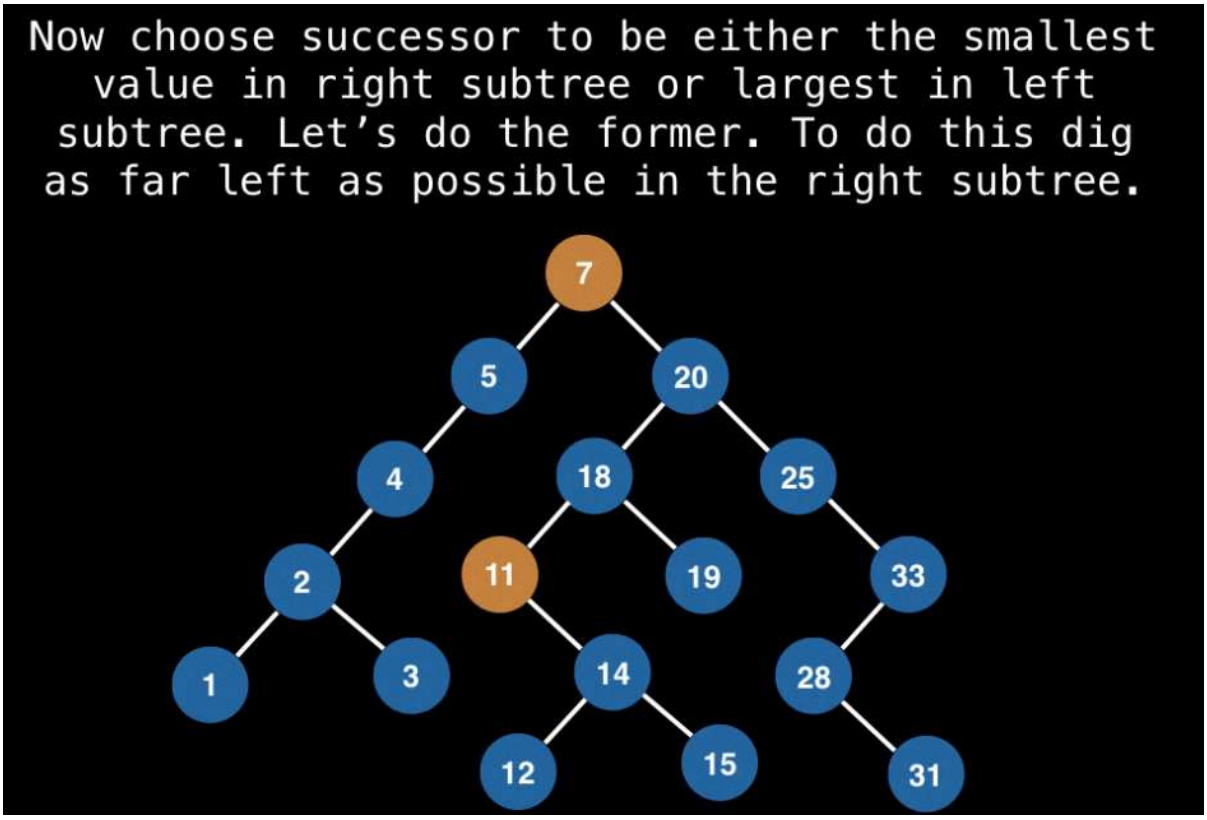
Case IV: node to remove has both left and right subtrees

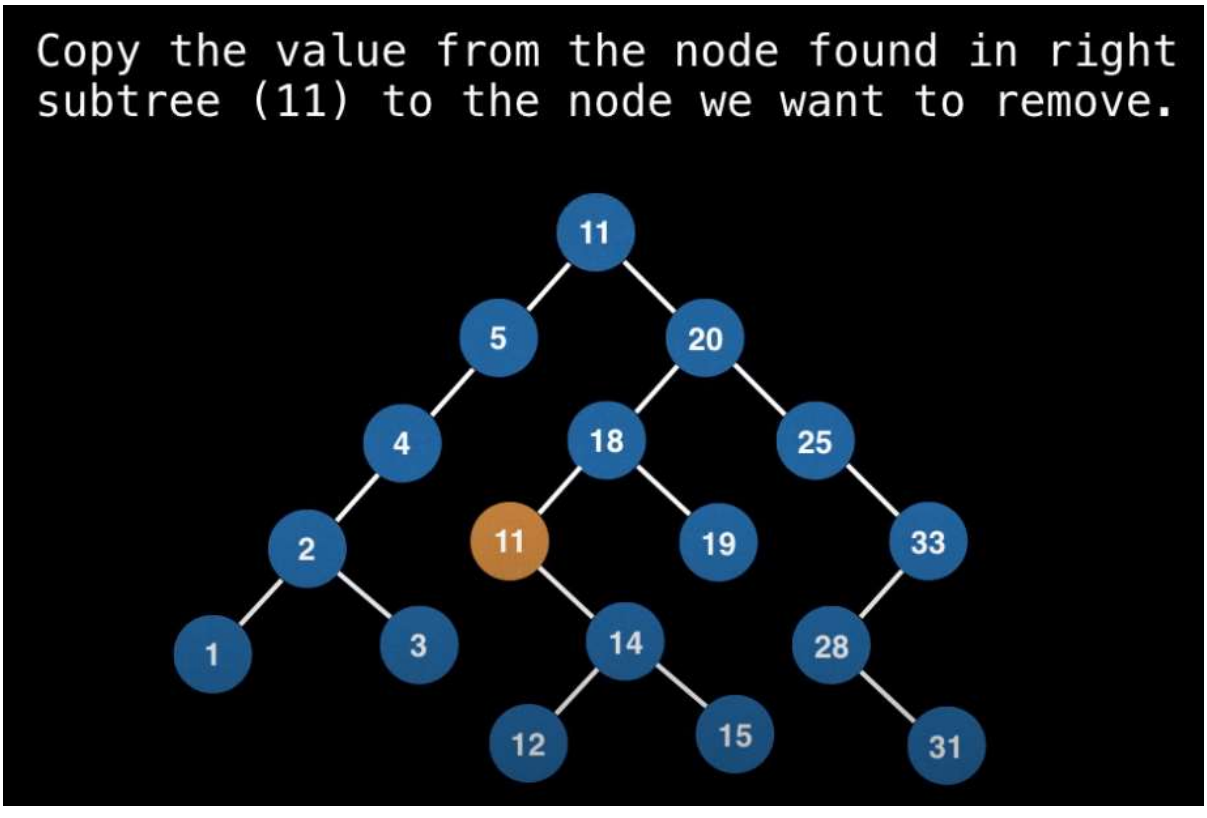
Q: In which subtree the successor of the node we are trying to remove be?

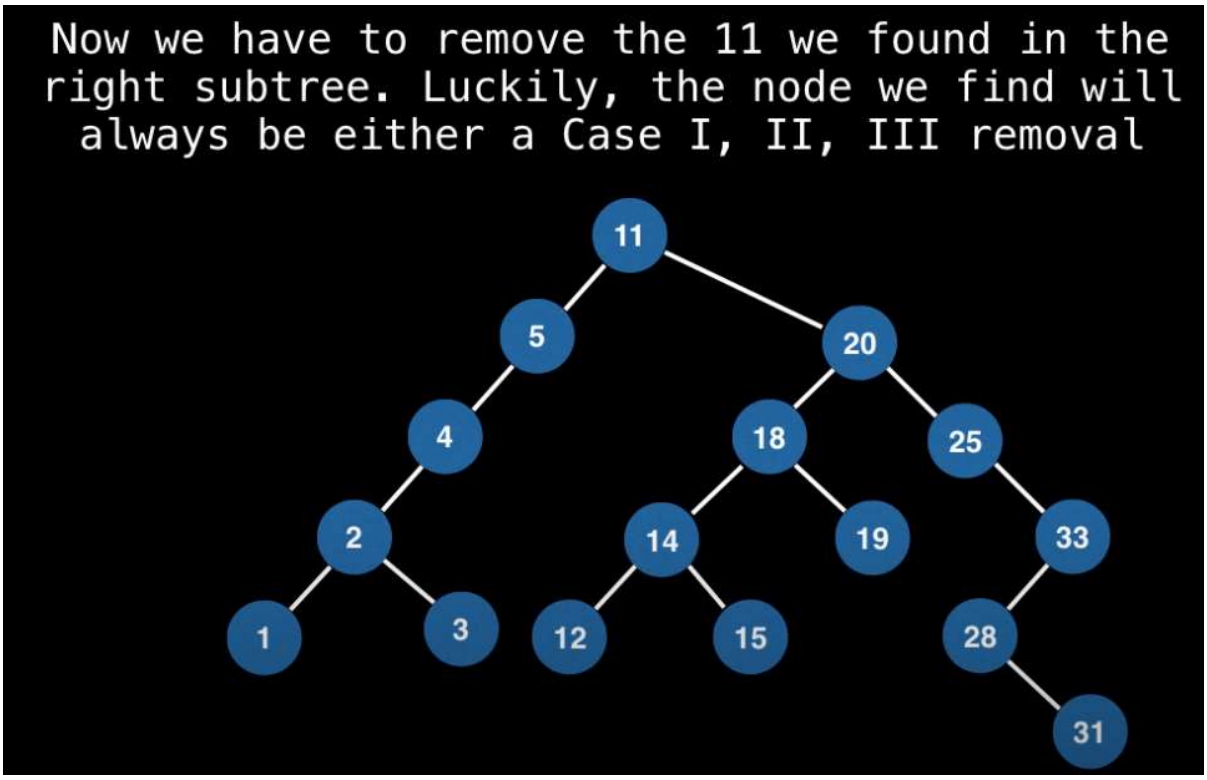
A: The answer is **both**. The successor can either be the **largest value in the left subtree** OR the **smallest value in the right subtree** (SO IT COULD BE BOTH).

Thus, there are 2 possible successors.

Example: Let's remove 7







The removal of duplicate successor can be done recursively with the same function.

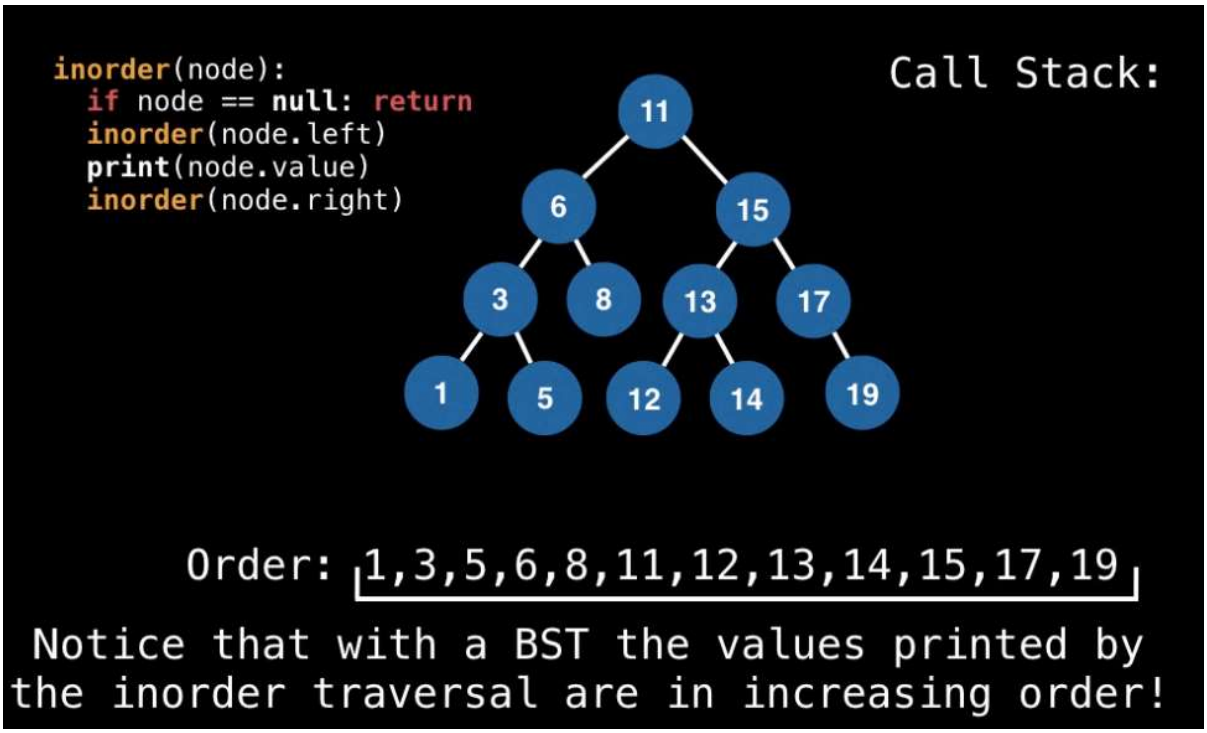
Tree Traversals

1. V: visit the node
2. L: traverse the left subtree of the node
3. R: traverse the right subtree of the node

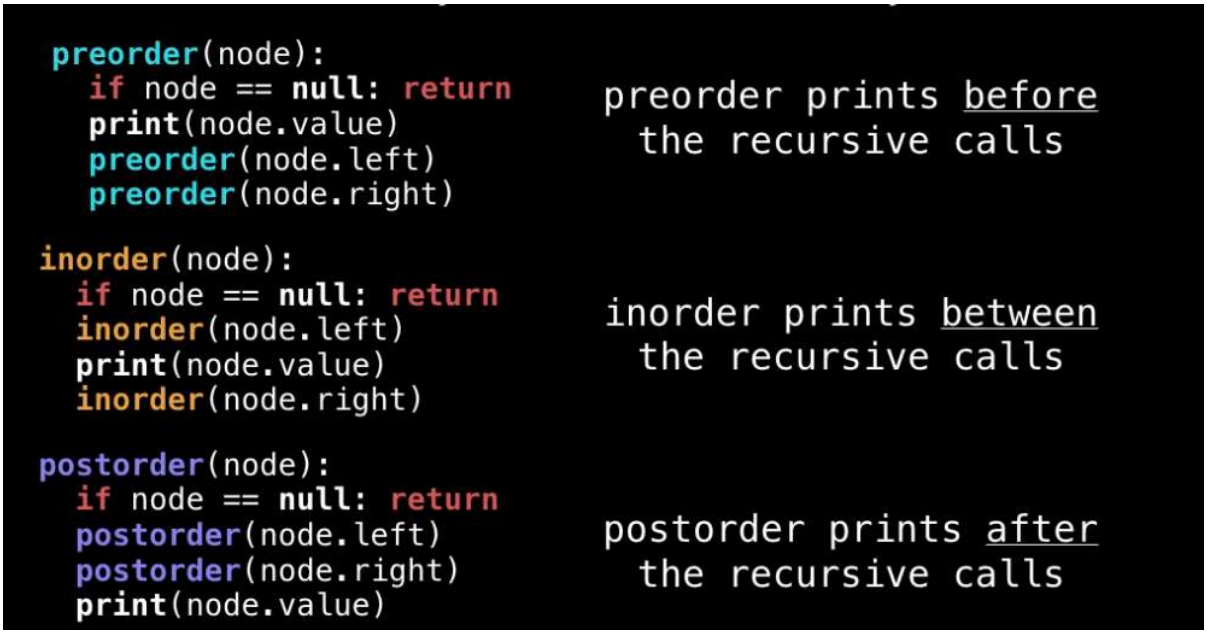
Pre-order Traversal: The pre-order traversal is a special case of the **depth-first traversal**. In the pre-order traversal, we start at the root, and then perform the above operations in the order: **VLR**. That is, every node is visited before both of its subtrees, and then we always traverse the left subtree before the right subtree.

In-order Traversal: The in-order traversal performs the traversal operations in the sequence: **LVR**, starting at the root. That is, the left subtree of a node is always traversed before the node is visited, and the right subtree of a node is always traversed after the node is visited. **In-order traversal is neither breadth-first, nor depth-first traversal**.

In-order traversal on BSTs: values are printed in the increasing order.



Post-order Traversal: The post-order traversal starts at the root and performs the traversal operations in the sequence: **LRV**. That is, both the left and right subtrees of a node t are traversed before t is visited. Again, this is **neither a breadth-first, nor depth-first traversal**.



Binary Search Tree Traversals VIDEO

BST Data Structure JAVA