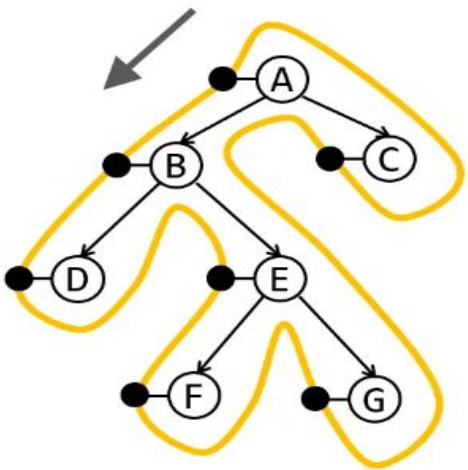
Depth First Search(DFS) on Trees

Naren Doraiswamy

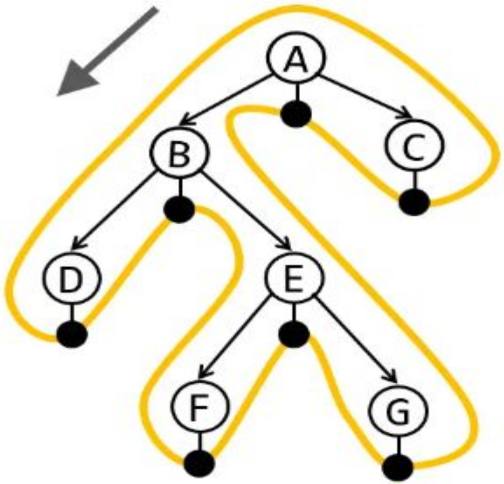
Tree Traversals

Node, Node.left, Node.right



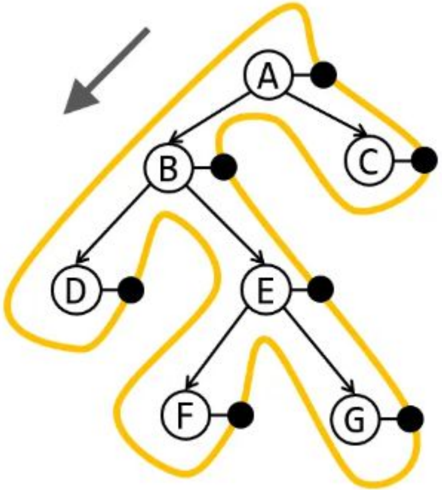
Pre-order traversal **[A, B, D, E, F, G, C]**

Node.left, Node, Node.right

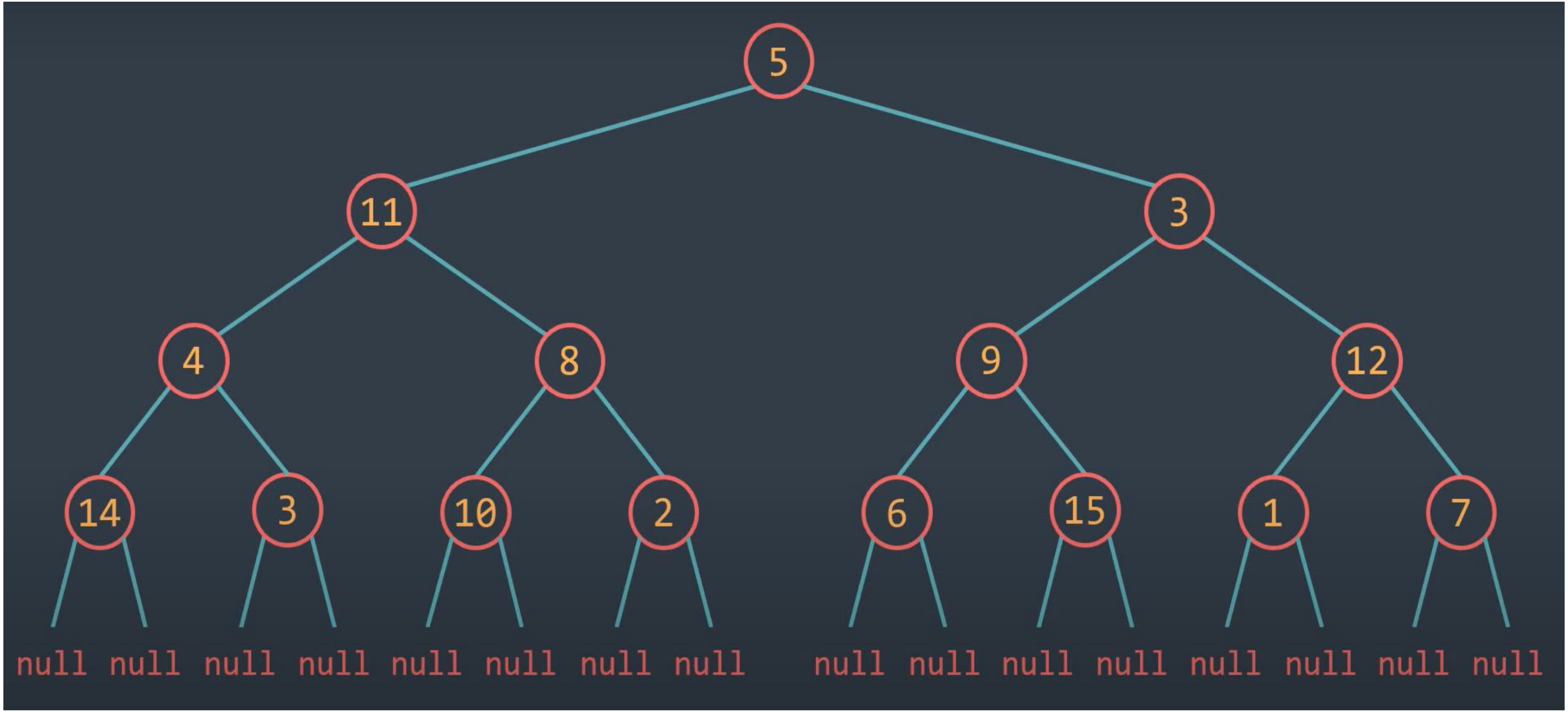


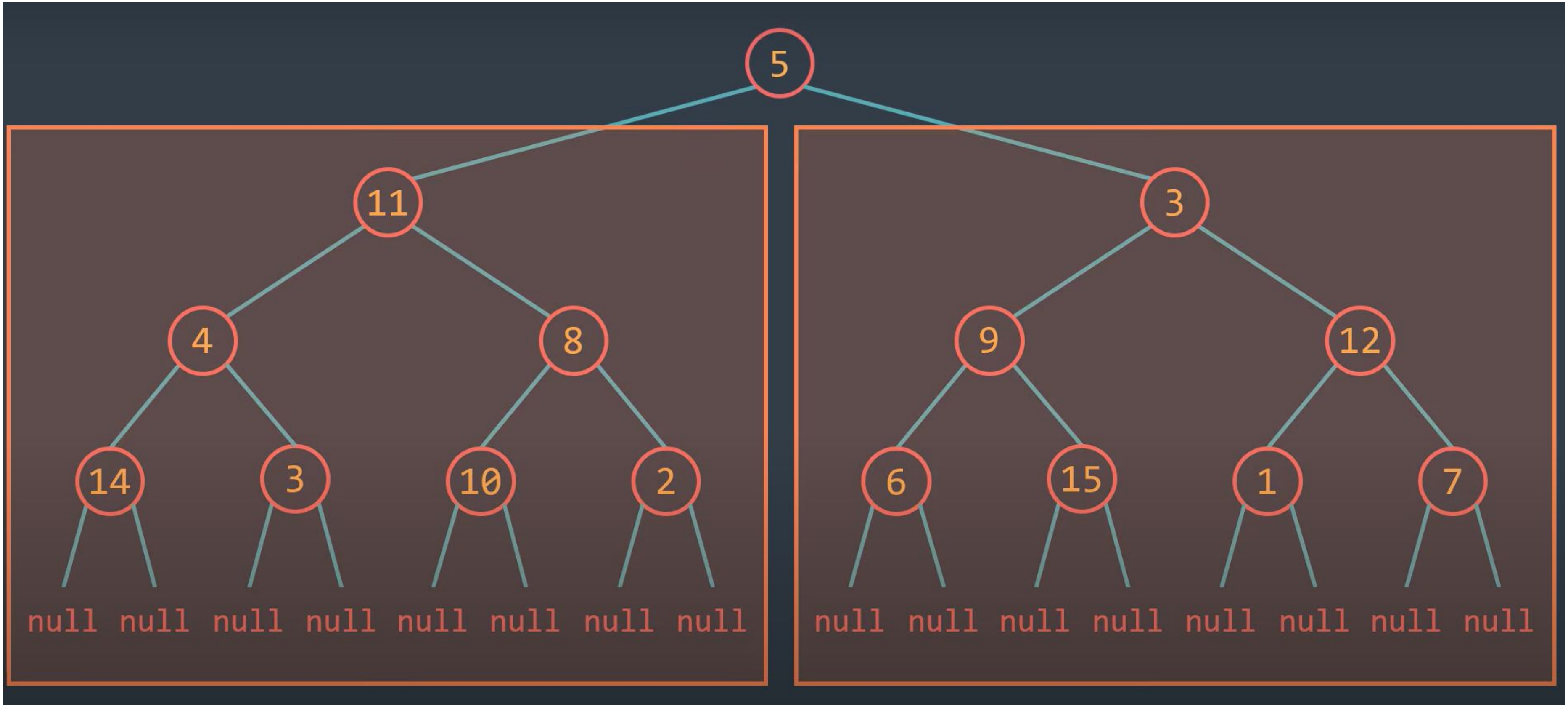
In-Order Traversal **[D, B, F, E, G, A, C]**

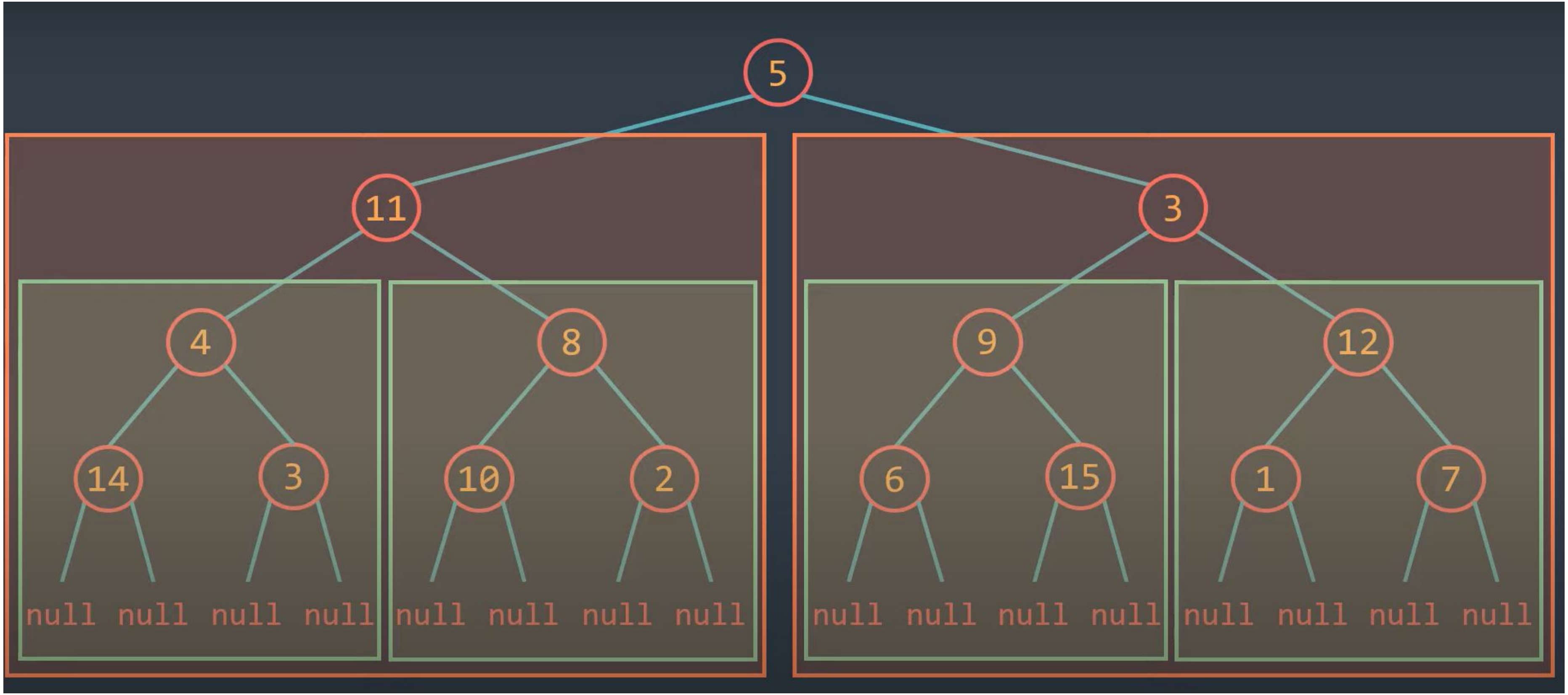
Node.left, Node.right, Node

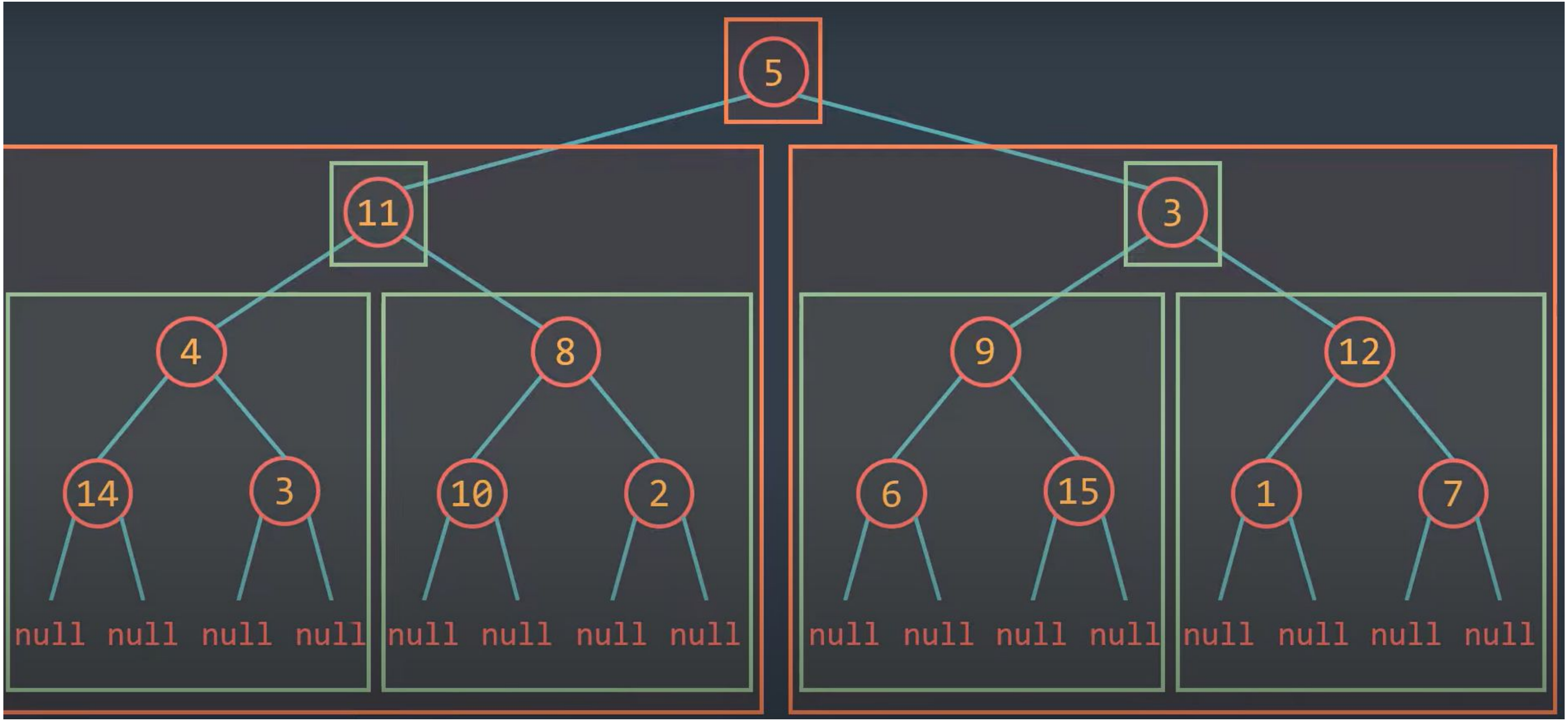


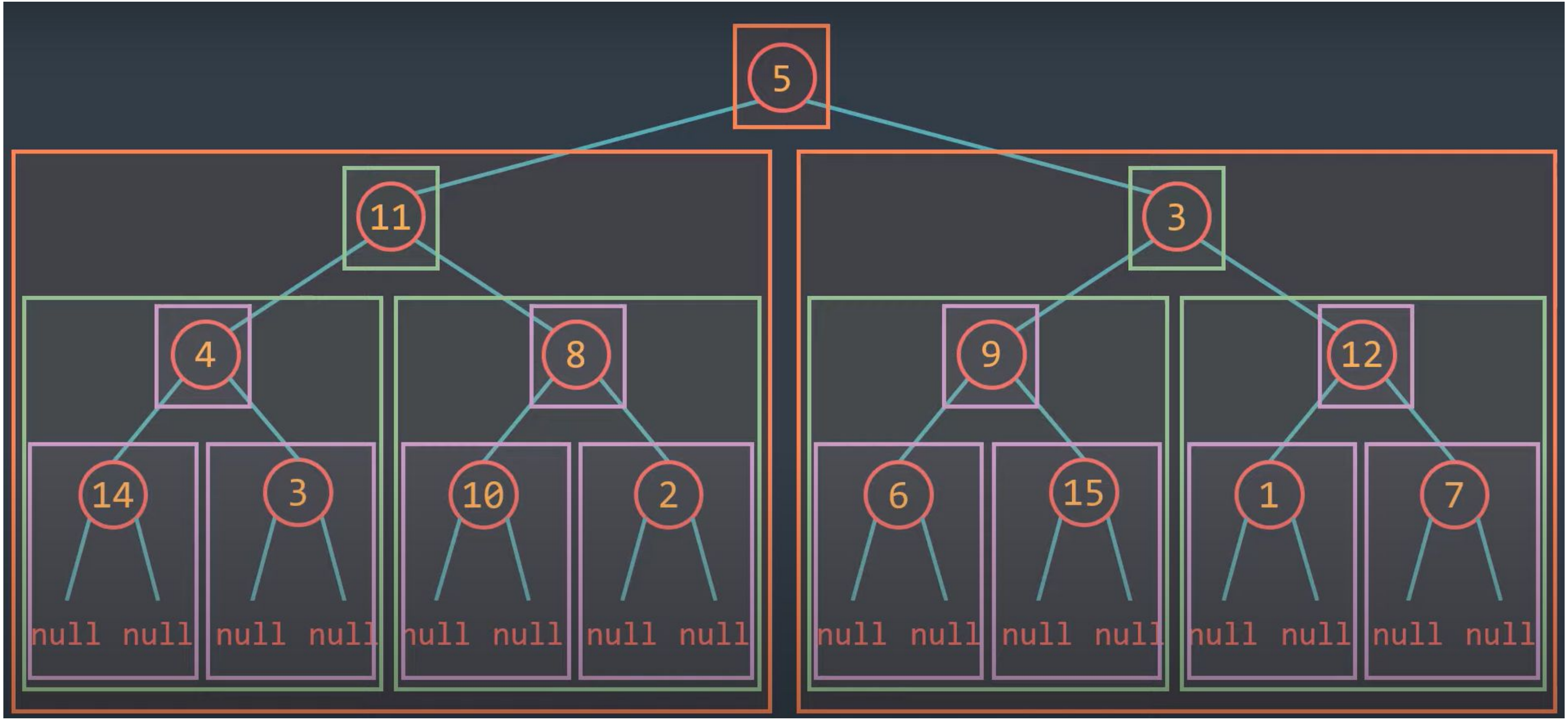
Post-Order Traversal **[D, F, G, E, B, C, A]**

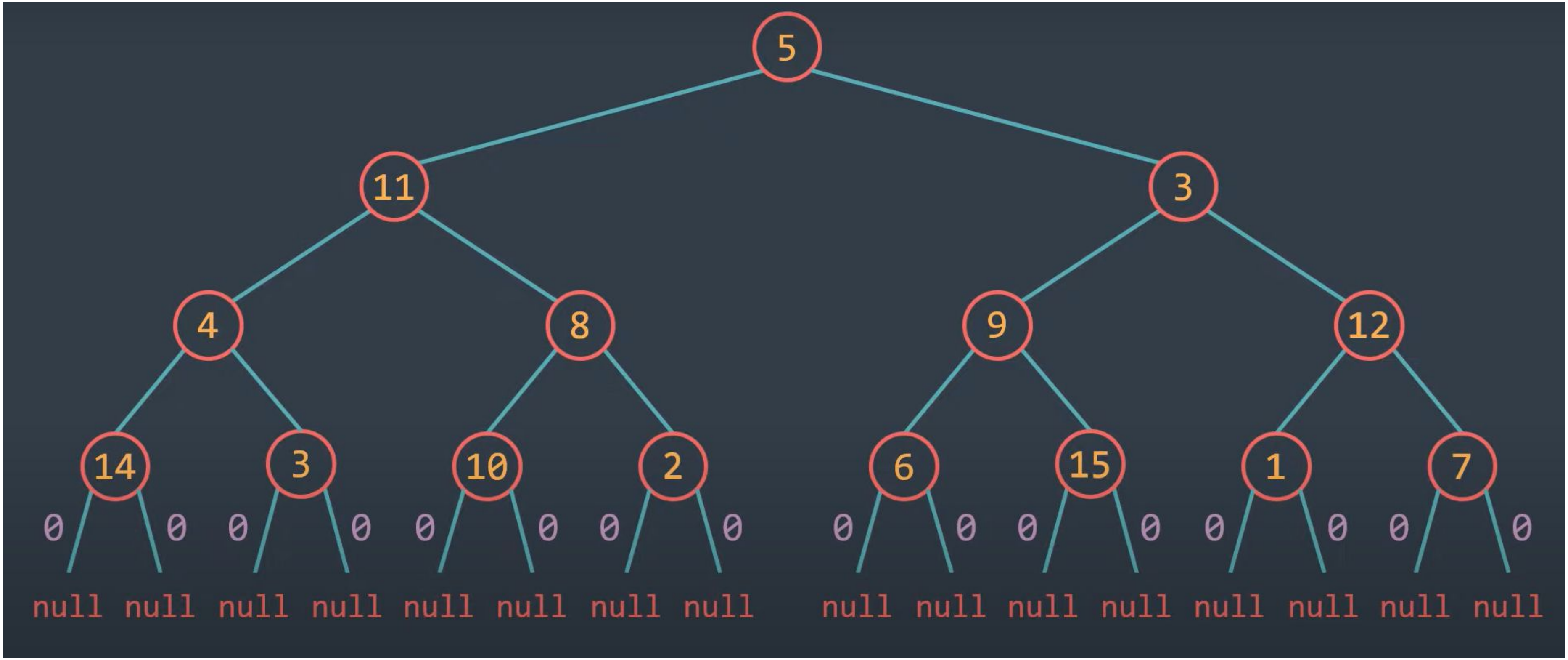

5
11
3
4
8
9
12
14
3
10
2
6
15
1
7
null null null null null null null null null null null null null null null null

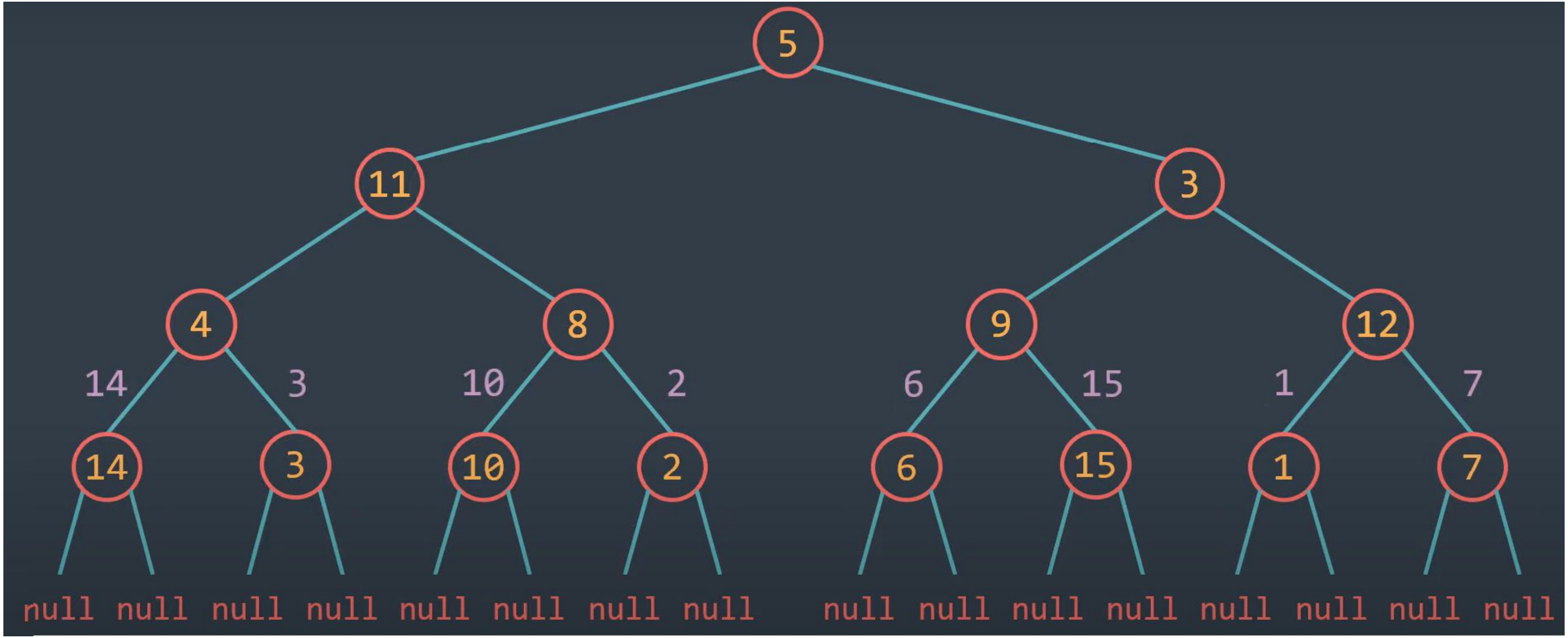

5
11
3
4
8
9
12
14
3
10
2
6
15
1
7
null null null null null null null null
null null null null null null null null

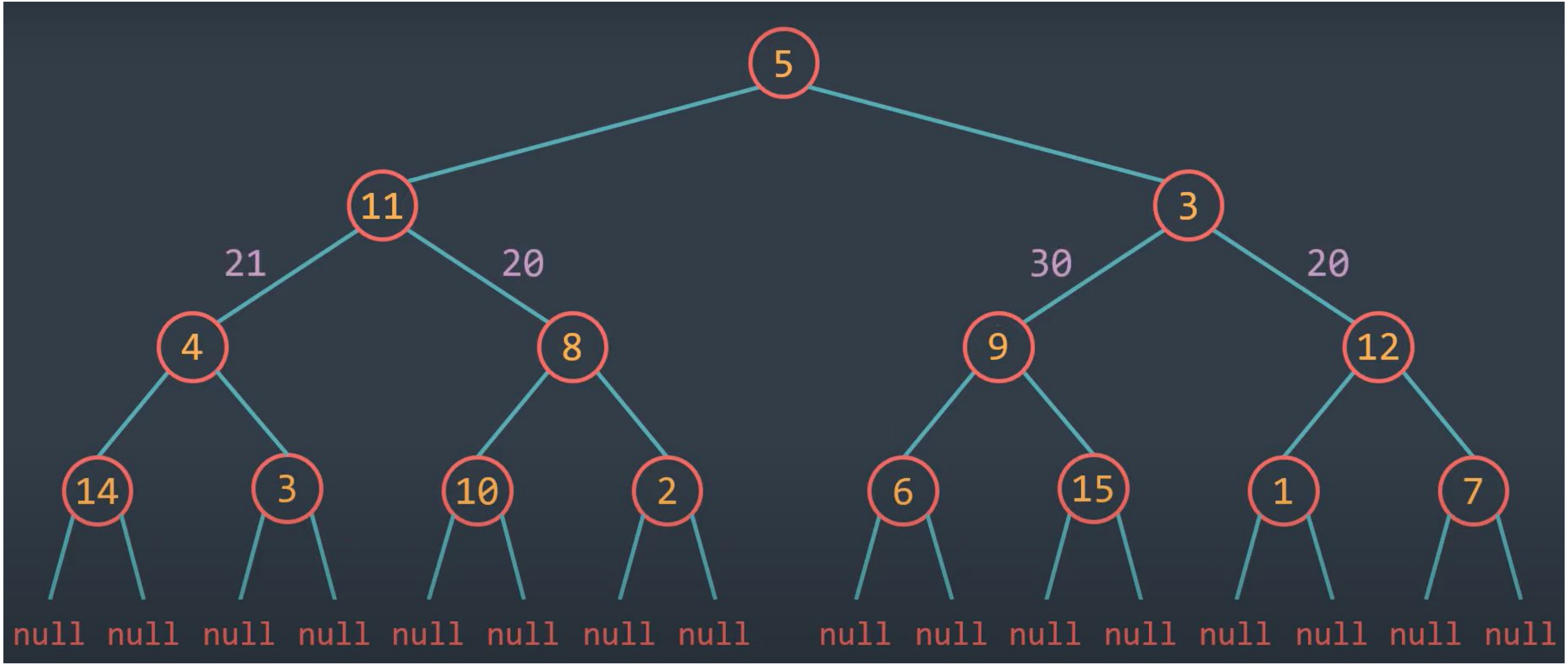

5
11
3
4
8
9
12
14
3
10
2
6
15
1
7
null null null null null null null null null null null null null null null null

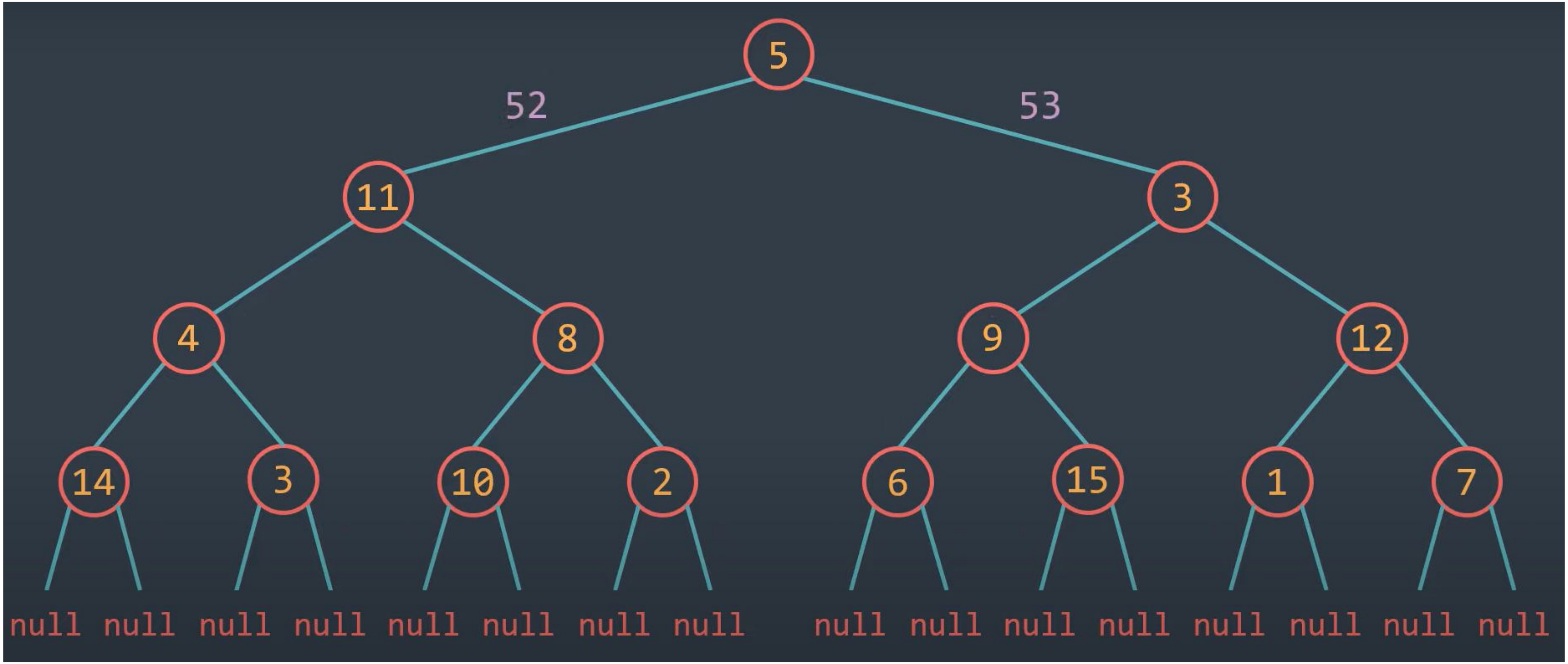

5
11
3
4
8
9
12
14
3
10
2
6
15
1
7
null null null null null null null null null null null null null null null null

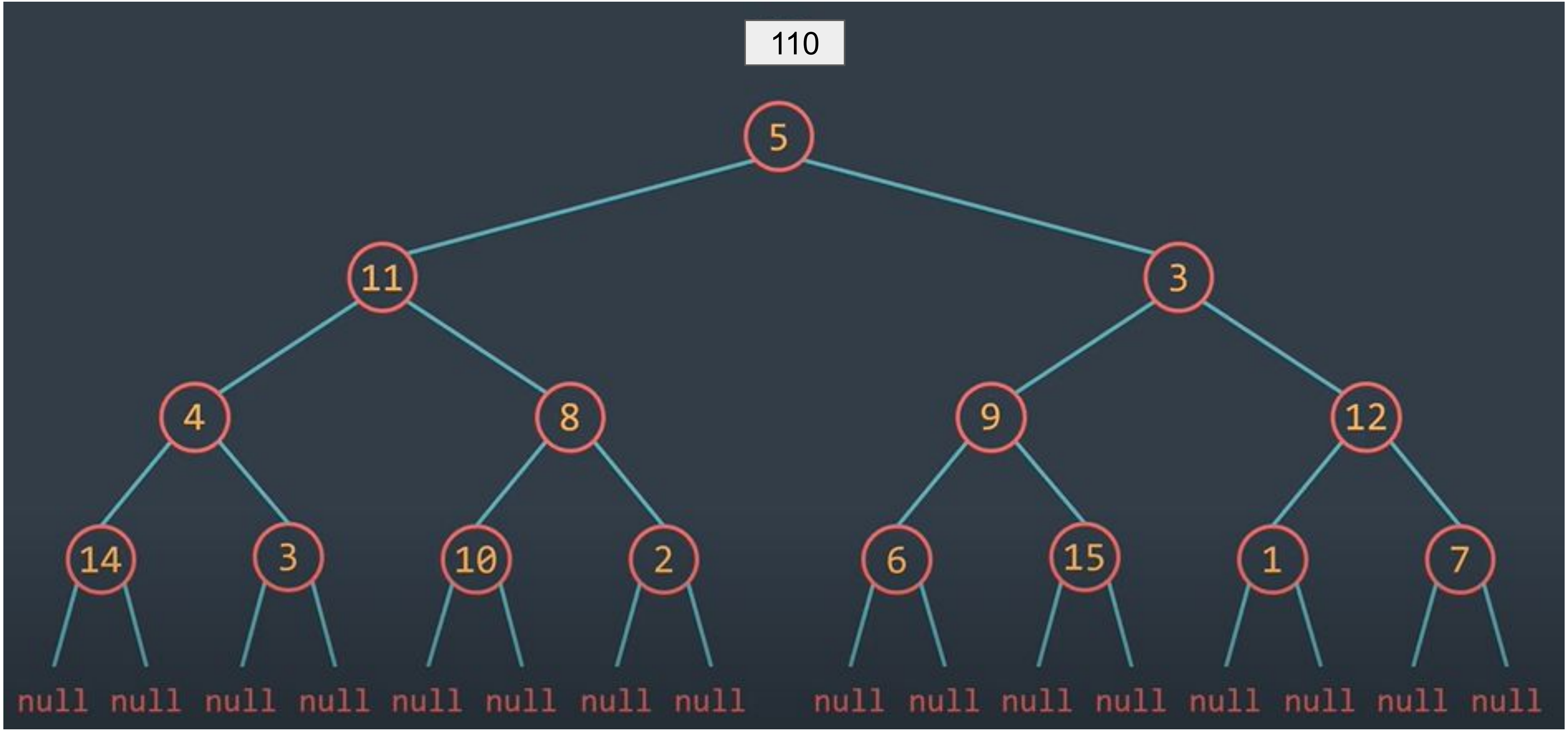

5
11
3
4
8
9
12
14
3
10
2
6
15
1
7
null null null null null null null null null null null null null null null null


5
11
3
4
8
9
12
14
3
10
2
6
15
1
7
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
null null null null null null null null null null null null null null null null


5
11
3
4
8
9
12
14
3
10
2
6
15
1
7
14
3
10
2
6
15
1
7
null null null null null null null null null null null null null null null null


5
11
3
21
20
30
20
4
8
9
12
14
3
10
2
6
15
1
7
null null null null null null null null null null null null null null null null


5
52
53
11
3
4
8
9
12
14
3
10
2
6
15
1
7
null null null null null null null null null null null null null null null null


110
5
11
3
4
8
9
12
14
3
10
2
6
15
1
7
null null null null null null null null null null null null null null null null

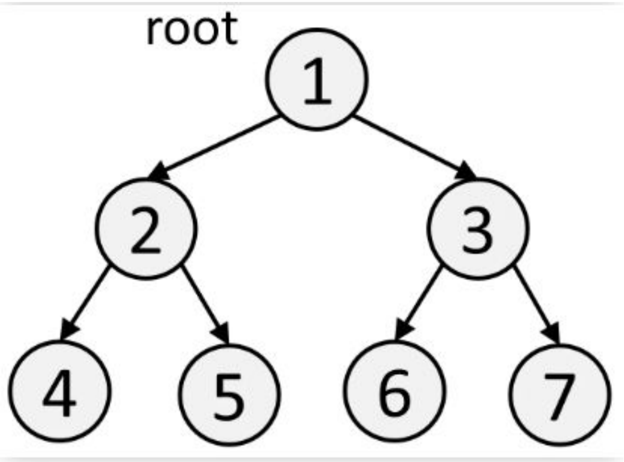
Steps in solving most DFS based Tree problems.

1. Figure out the Base conditions.
2. Call the recursive function on left sub-tree.
3. Call the recursive function on right sub-tree.
4. Combine to get the required results.

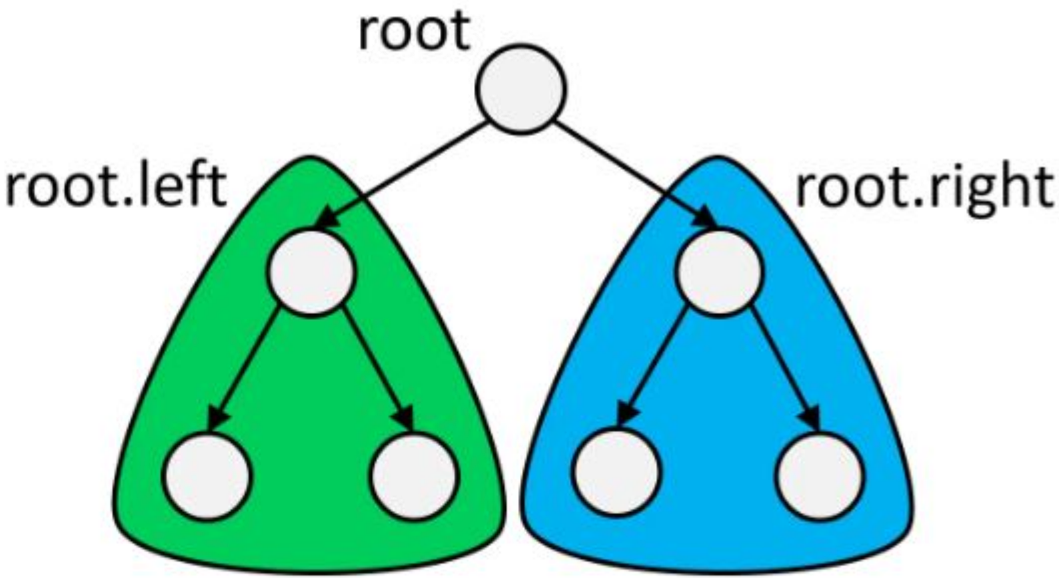
```
def sum_tree(root):

        # Base condition
        If root is None:
                return 0
        # Recursive condition
        return root.val + sum_tree(root.left) +
sum_tree(root.right)
```

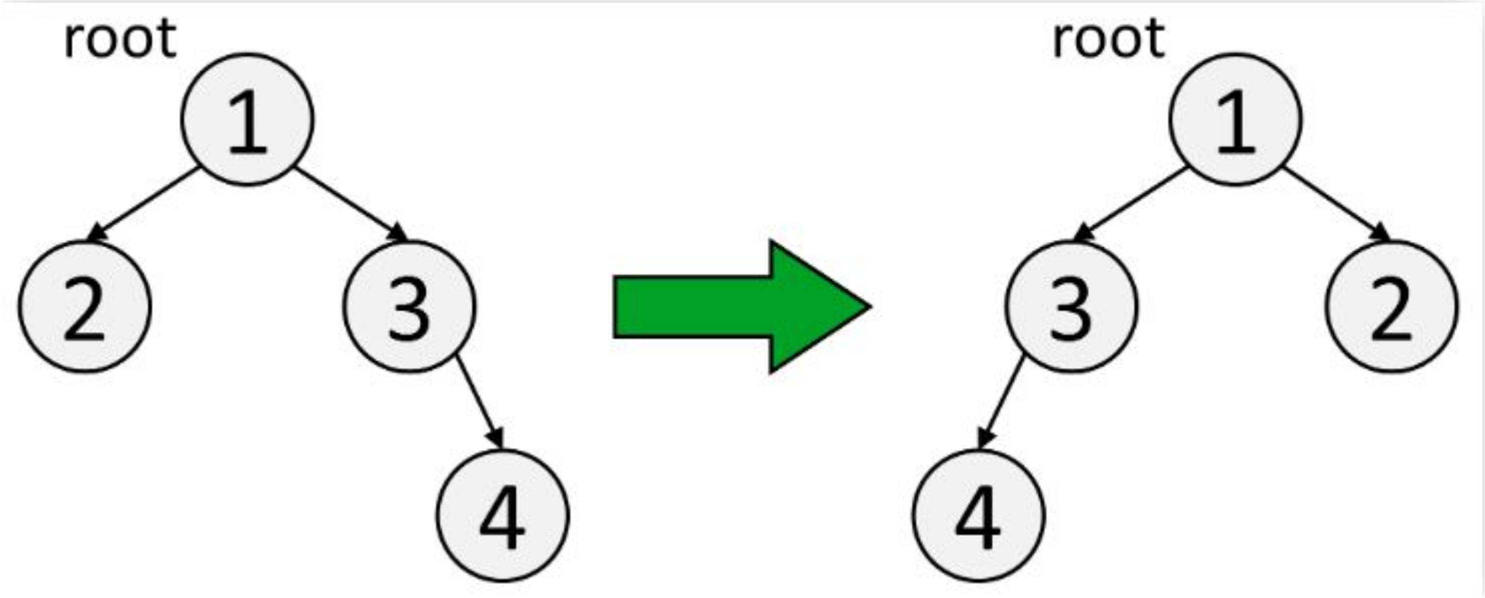
Question 1: Sum of number of nodes

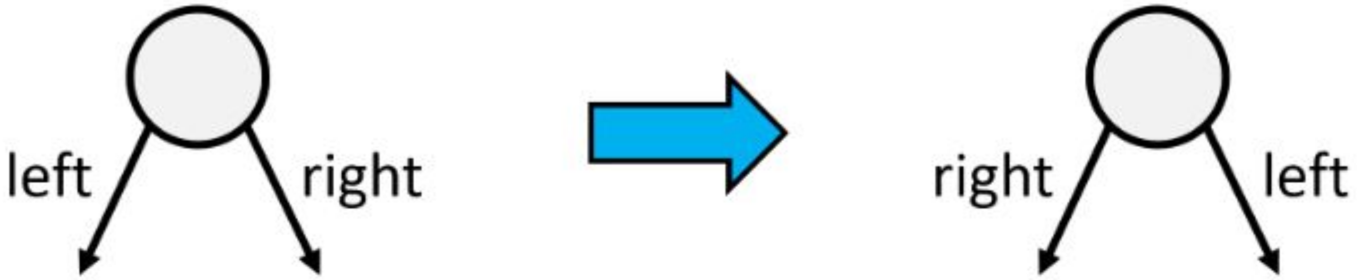


=> returns 7

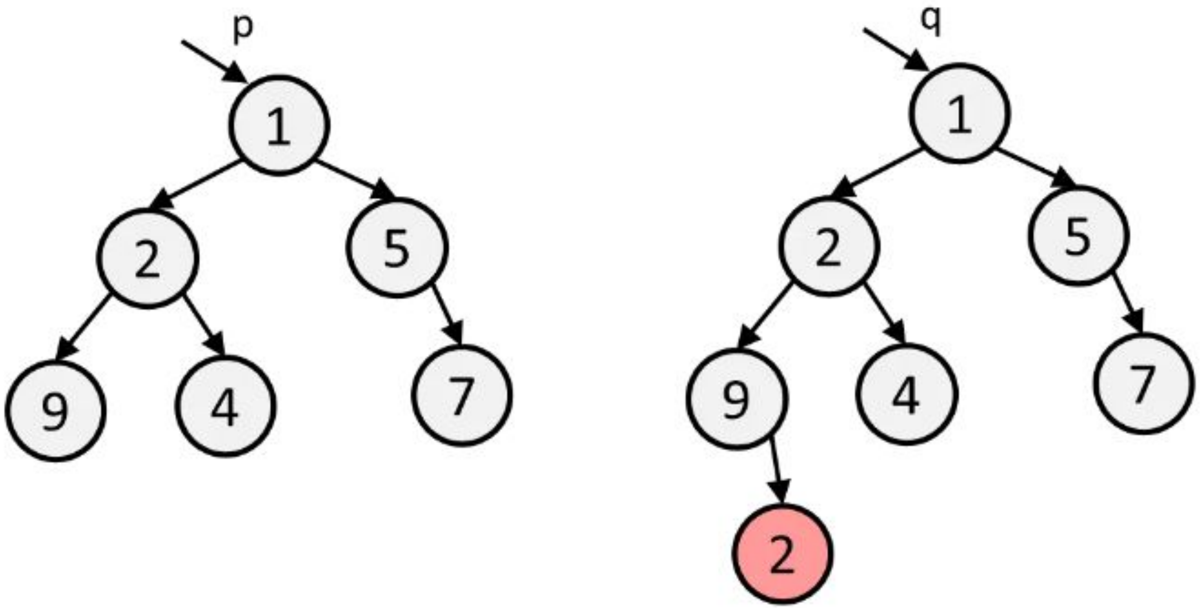

root
root.left
root.right

Question 2 : Mirror Tree




left
right
right
left

Question 3: Equal Trees?

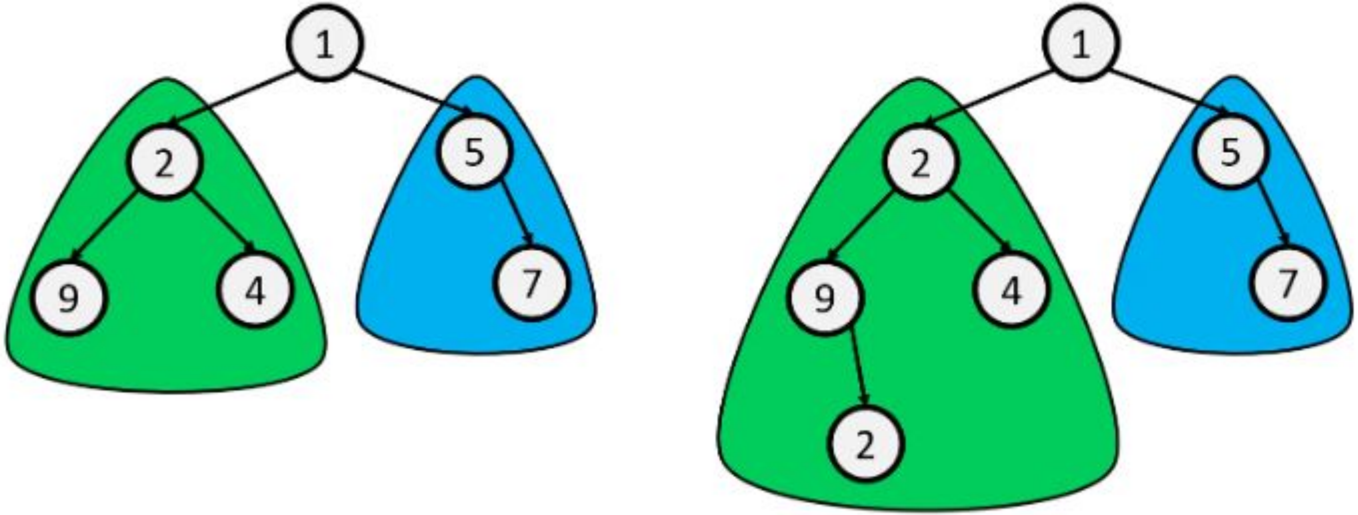


`=> returns false`

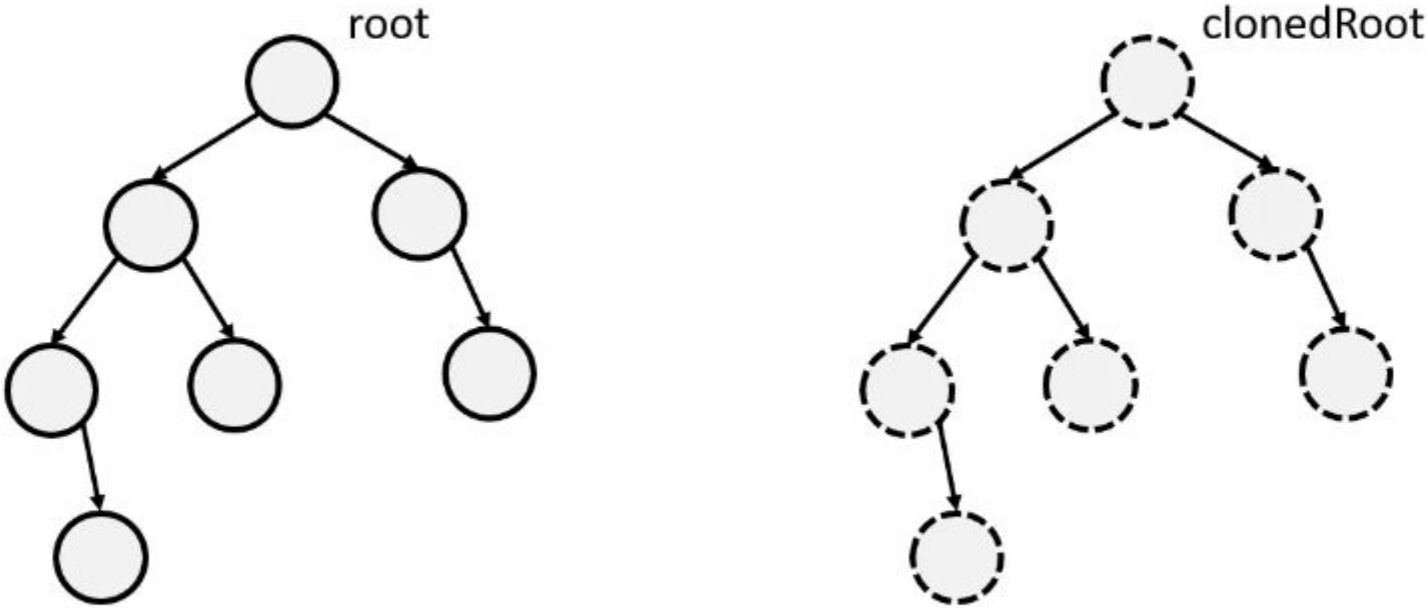
Values are equal?

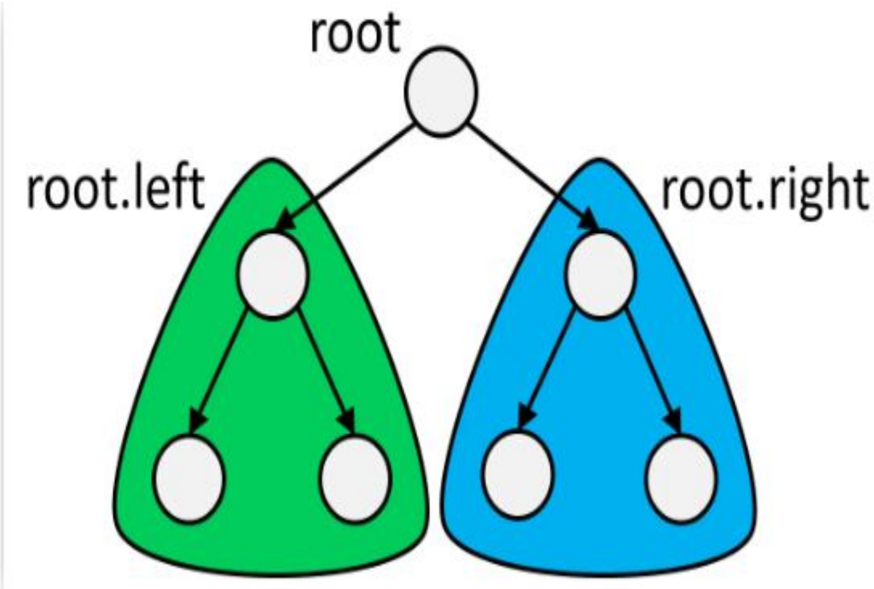
Left subtrees are the same?

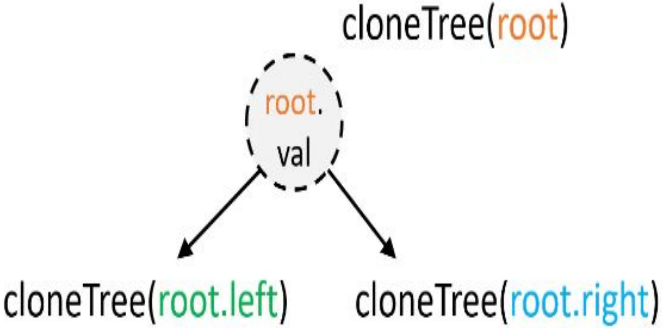
Right subtrees are the same?



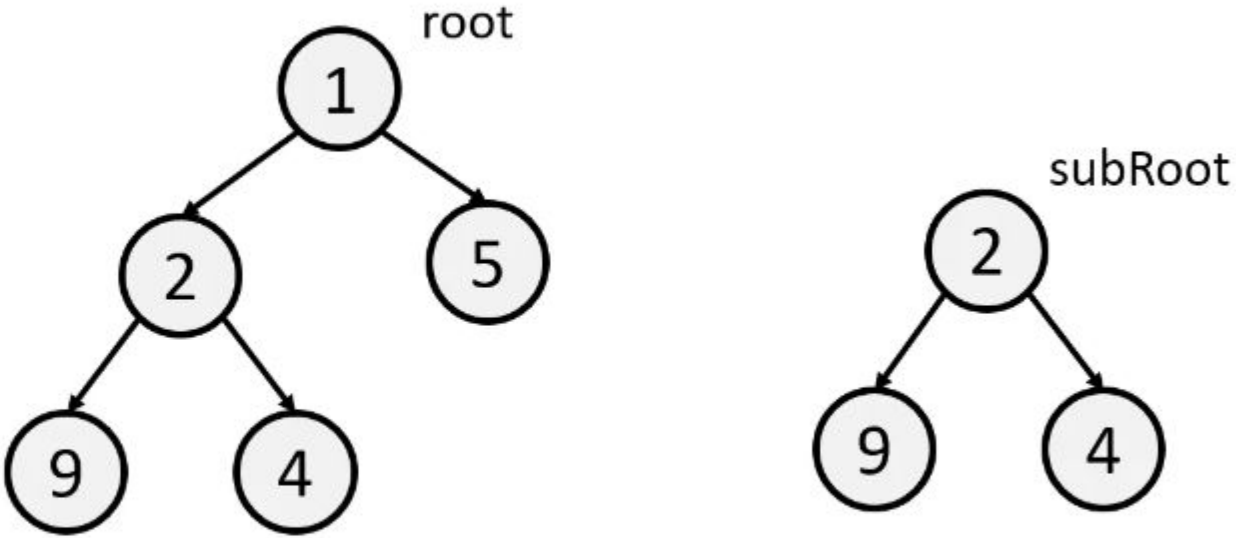
Question 4: Clone tree




root
root.left
root.right

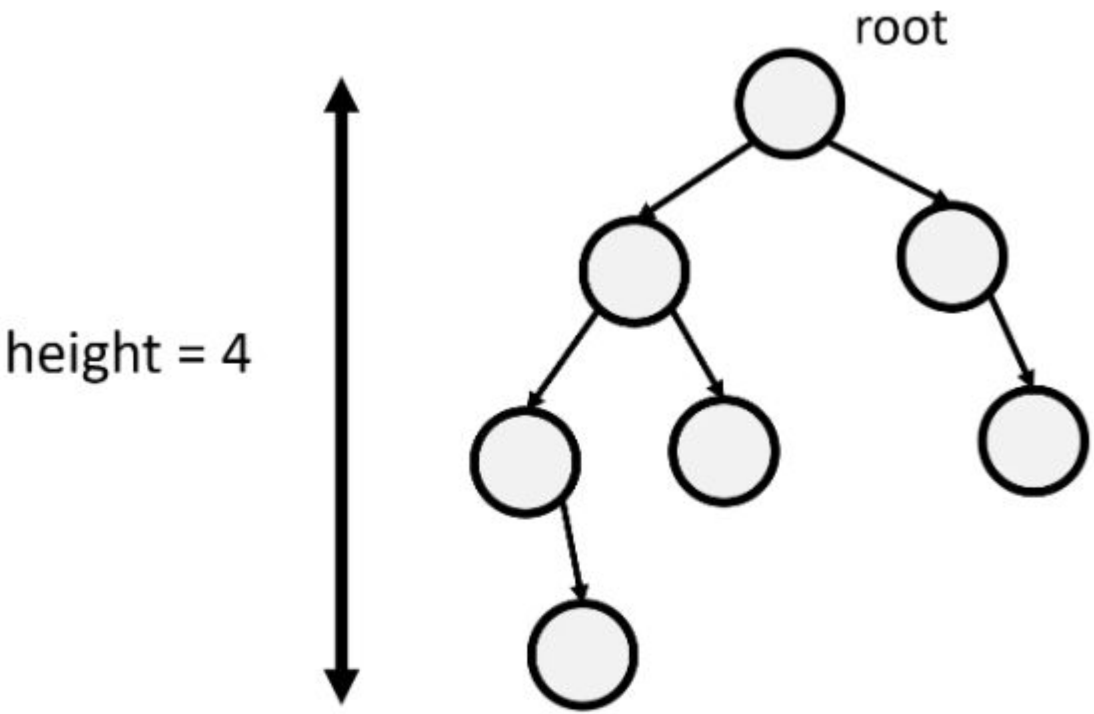

cloneTree(root)
root.
val
cloneTree(root.left)
cloneTree(root.right)

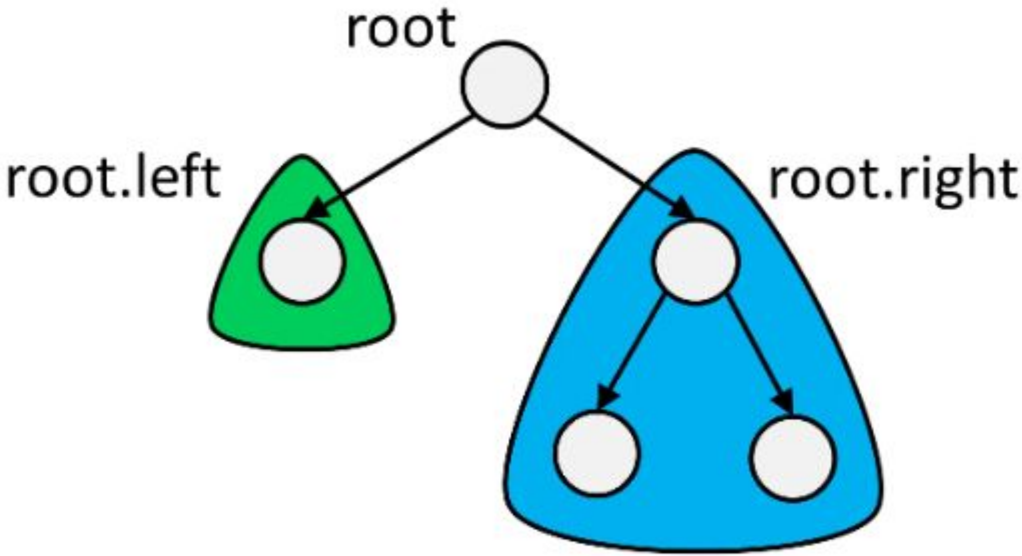
Question 5: Tree contains Sub-Tree



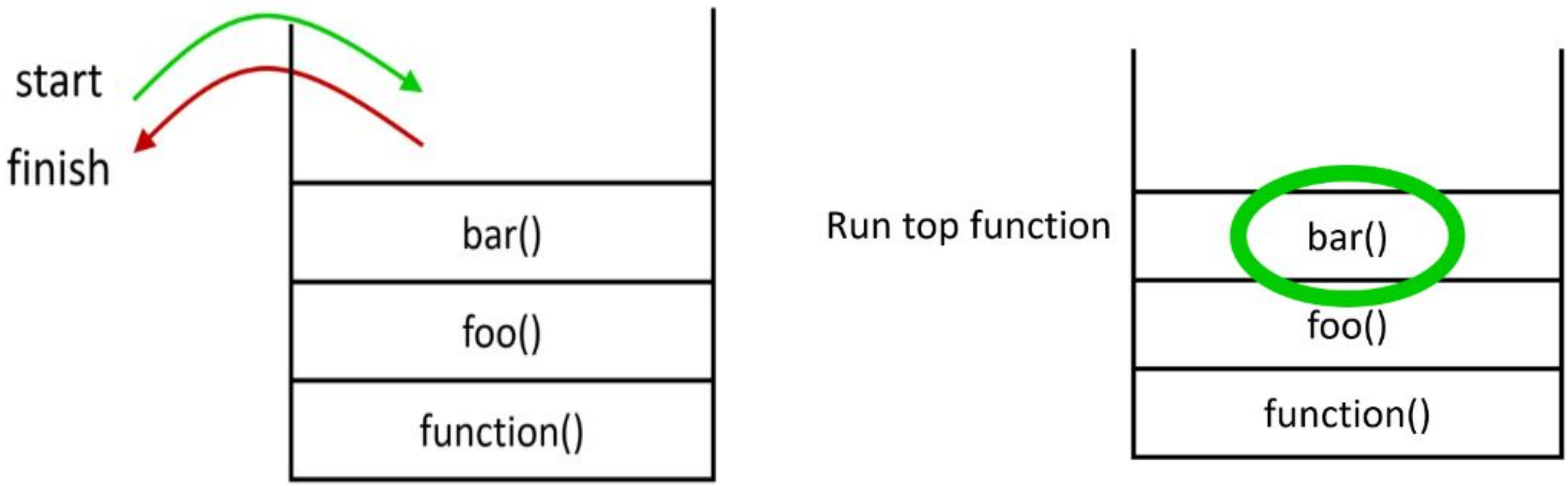
=> returns True

Question 6: Height of a tree

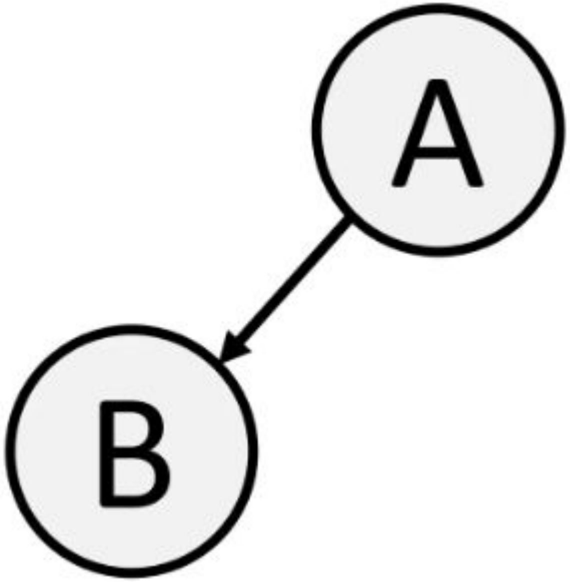



root
root.left
root.right

Call Stack: Stack of function calls.

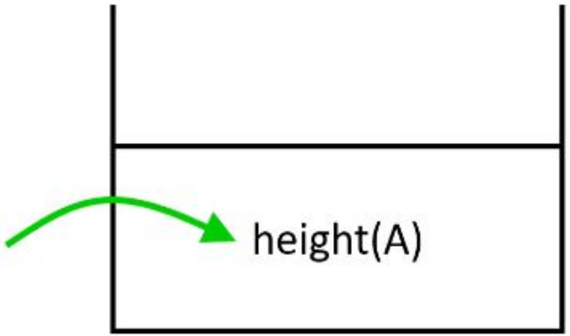


Calculate the height of the tree




```python
def height(node):
    if node == None:
        return 0
    heightL = height(node.left)
    heightR = height(node.right)
    return max(heightL, heightR) + 1
```

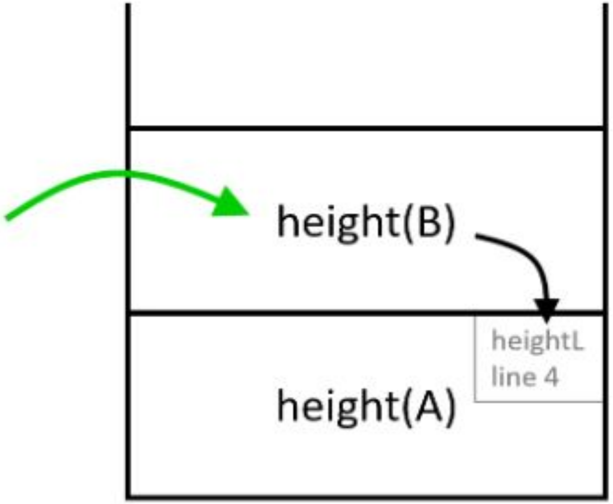
The very first thing that happens when we call `height(A)` is that the computer adds it to the Call Stack.



The computer then sees that `height(A)` is at the top of the Call Stack, so it starts computing `height(A)`. Eventually, the computer gets to line 4, and has to make *another* function call `height(B)`.

```
                 A
def height(root):
    if root == None:
        return 0                   B
    heightL = height(root.left)
    heightR = height(root.right)
    return max(heightL, heightR) + 1
```

The computer adds `height(B)` it to the Call Stack. It saves information on how to resume height(A) when height(B) finishes:



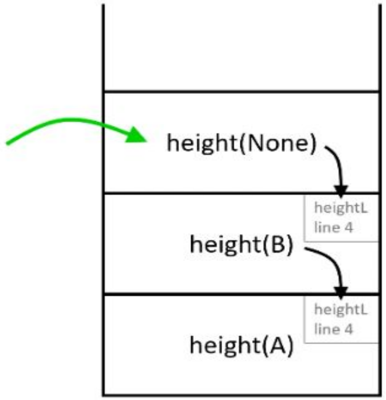
Now `height(B)` is on top of the Call Stack, so the computer starts running it. It eventually reaches another function call `height(None)`:

B

None

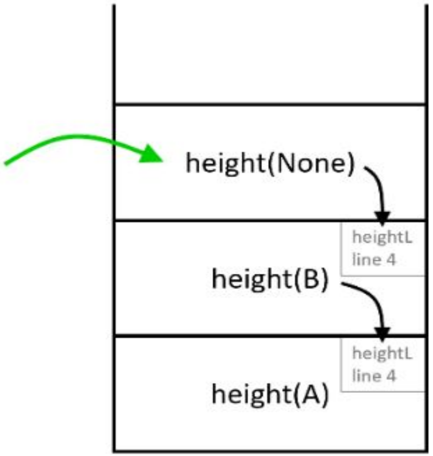
```
1  def height(root):
2      if root == None:
3          return 0
4      heightL = height(root.left)
5      heightR = height(root.right)
6      return max(heightL, heightR) + 1
```

So it and adds it to the Call Stack:



Now the function call `height(None)` is on top of the Call Stack, so the computer runs it. It gets to the base case, and returns `0`

Now `height(None)` finished, so the computer removes it from the Call Stack. It updates the variables inside of the `height(B)` function accordingly.

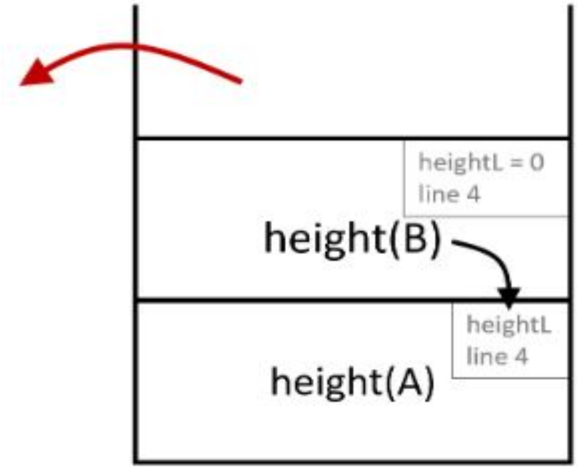


Now the function call `height(None)` is on top of the Call Stack, so the computer runs it. It gets to the base case, and returns `0`.

None

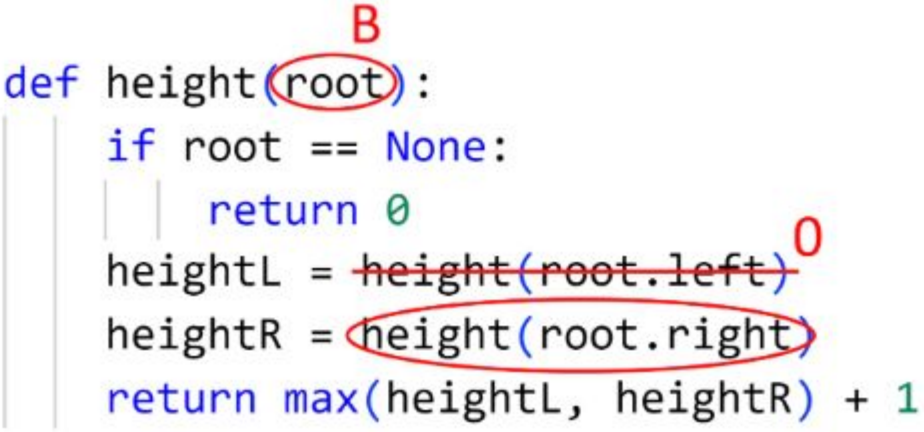
```
def height(root):
    if root == None:
        return 0
    heightL = height(root.left)
    heightR = height(root.right)
    return max(heightL, heightR) + 1
```

Now `height(None)` finished, so the computer removes it from the Call Stack. It updates the variables inside of the `height(B)` function accordingly.

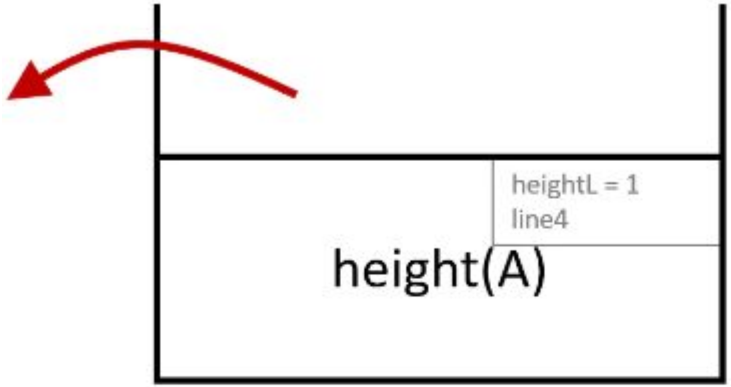


The Call Stack now has `height(B)` on top, so it resumes this function where it left off. Now it knows that `heightL = 0`.

So the computer resumes at line 5:



The computer finds that `heightR = 0`. The computer then returns `max(0, 0) + 1`, which is `1`. Since the computer finished `height(B)`, it removes it from the Call Stack. It updates the variables in `height(A)` accordingly.



It now resumes `height(A)` on line 4, and knows that `heightL = 1`.

A

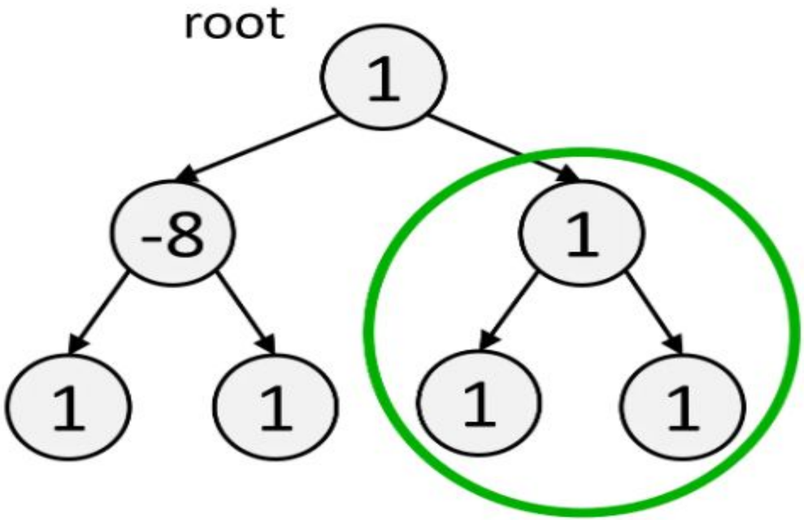
```
1   def height(root):
2       if root == None:
3           return 0                       1
4       heightL = height(root.left)
5       heightR = height(root.right)
6       return max(heightL, heightR) + 1
```

A

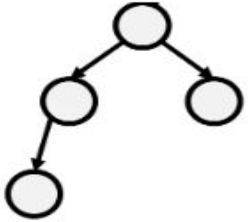
```
1   def height(root):
2       if root == None:
3           return 0                       1
4       heightL = height(root.left)
5       heightR = height(root.right)        0
6       return max(heightL, heightR) + 1       2
```

Recursion Hijacking

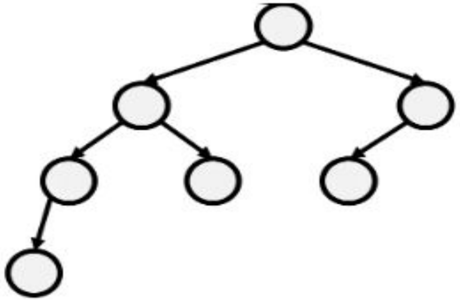
Max sub-tree sum:



Height balanced tree

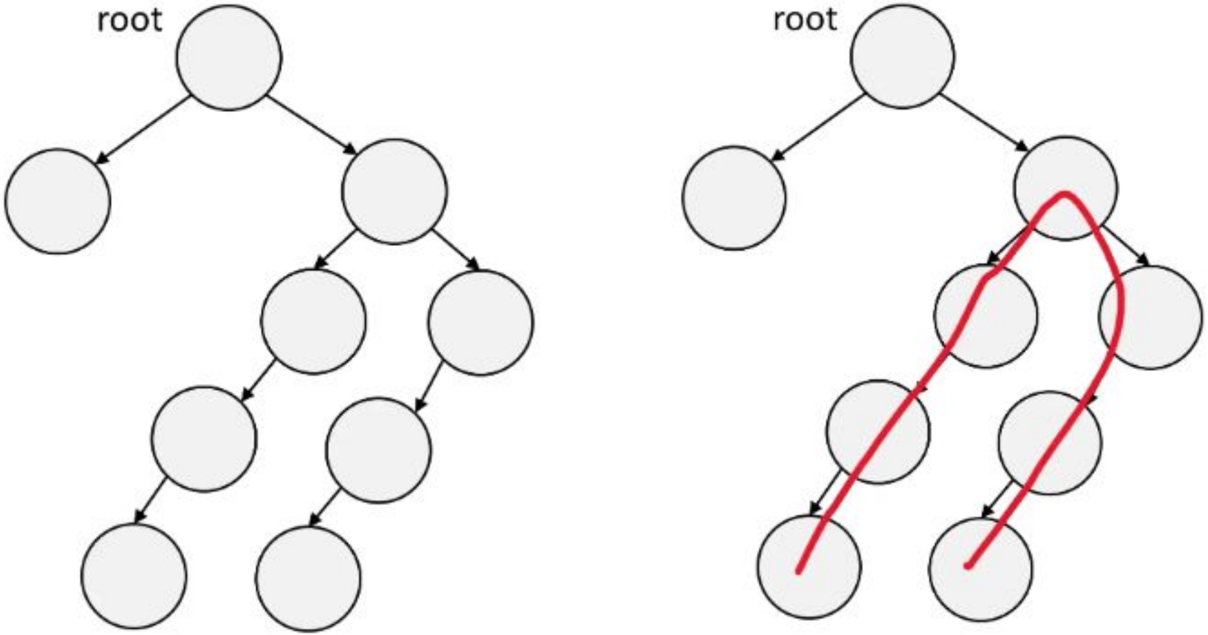


```
=> returns True
```



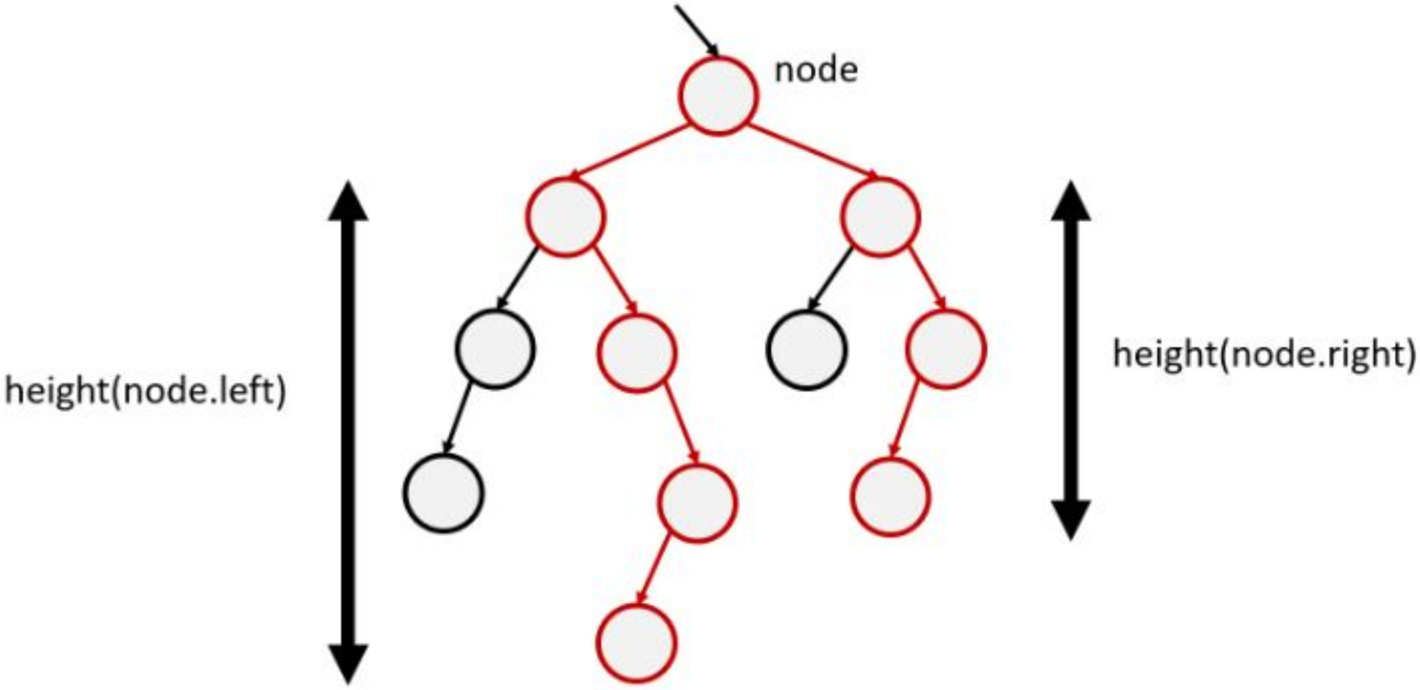
```
=> returns True
```

Diameter of a tree

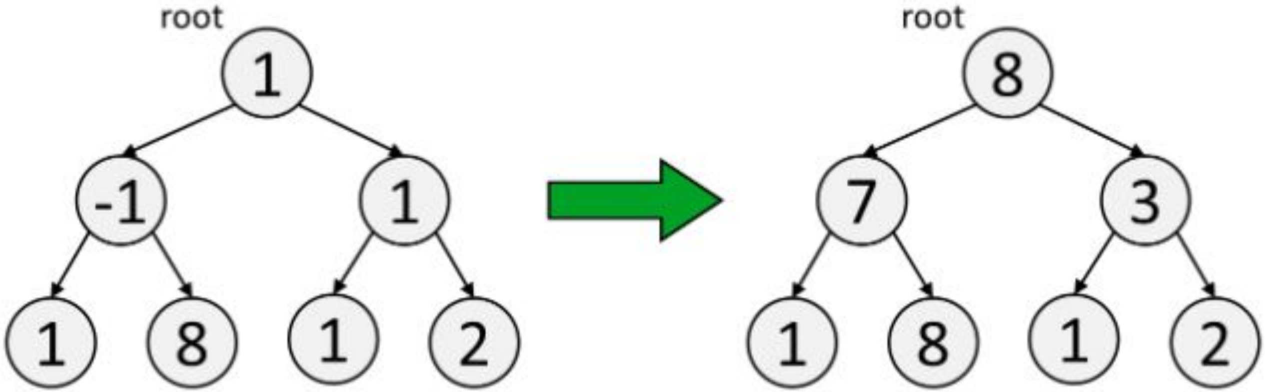


`=> returns 6 (red path)`

Diameter of a tree

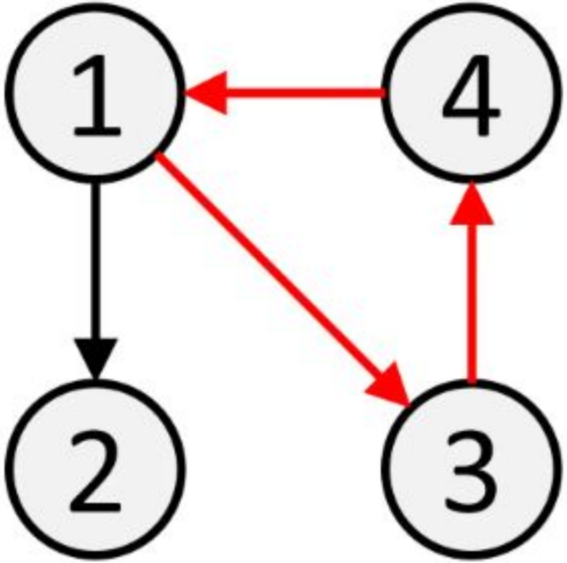


Max node to leaf sum



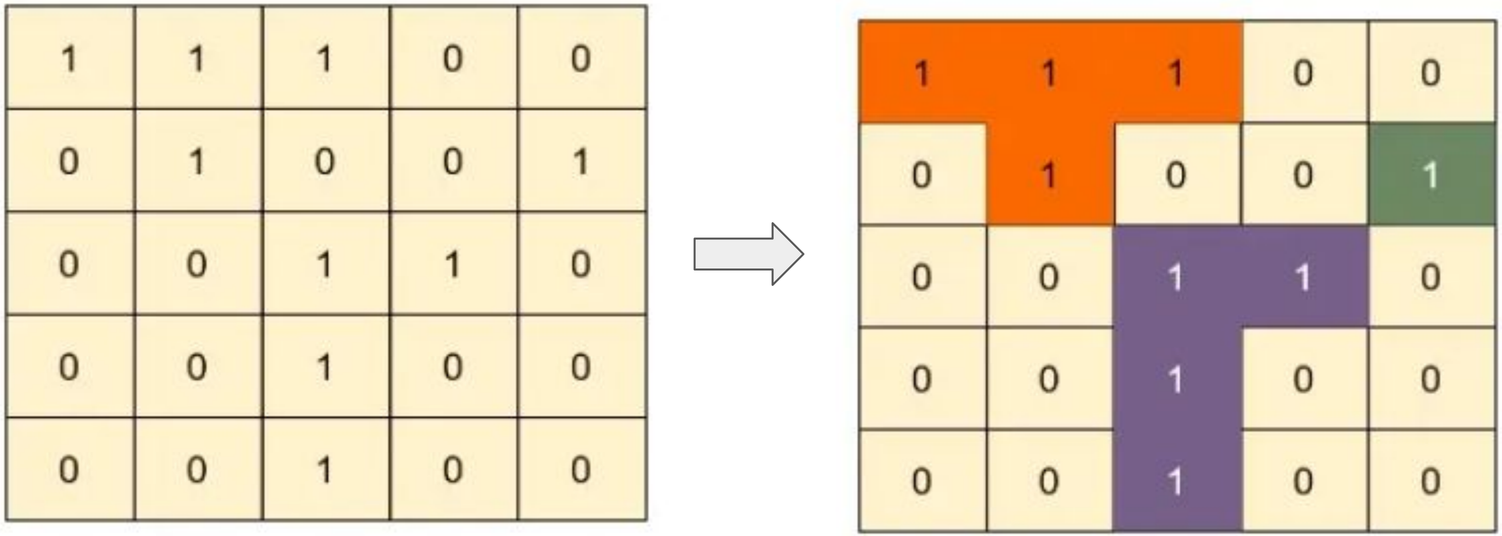
Graphs

Number of nodes in the graph

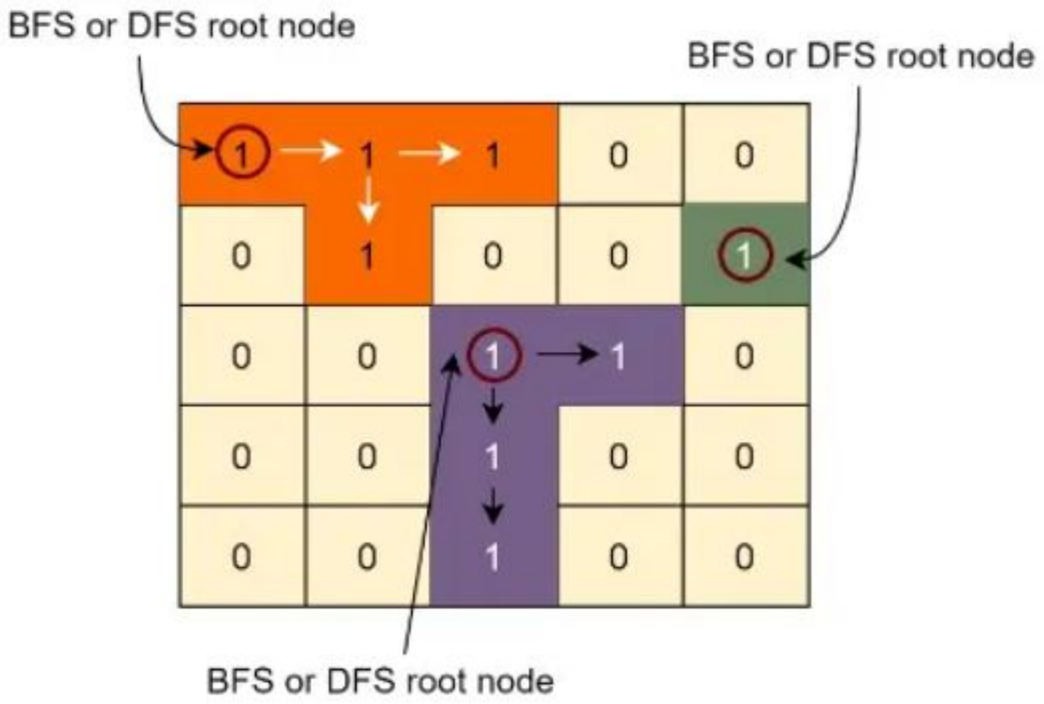


Returns 4

Calculate the number of islands

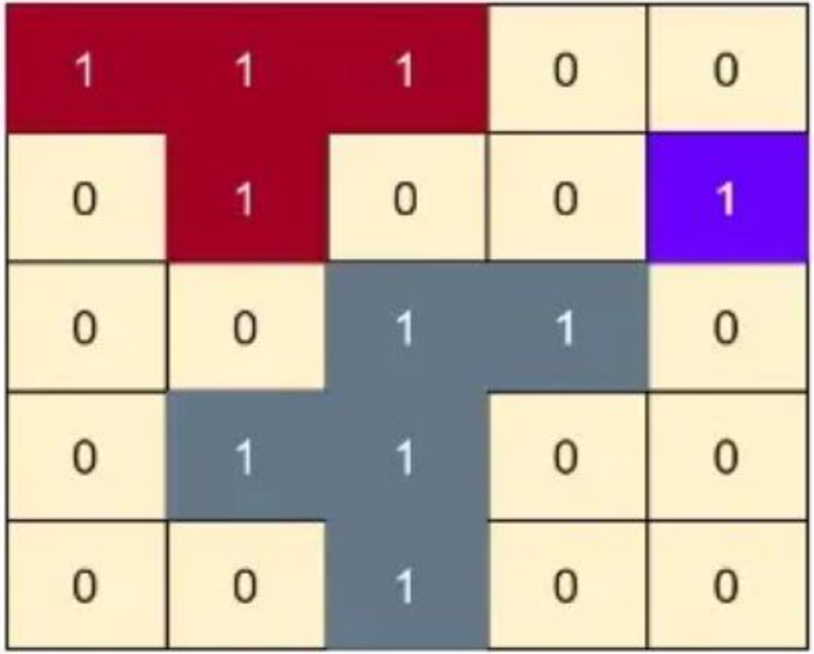


Calculate the number of islands



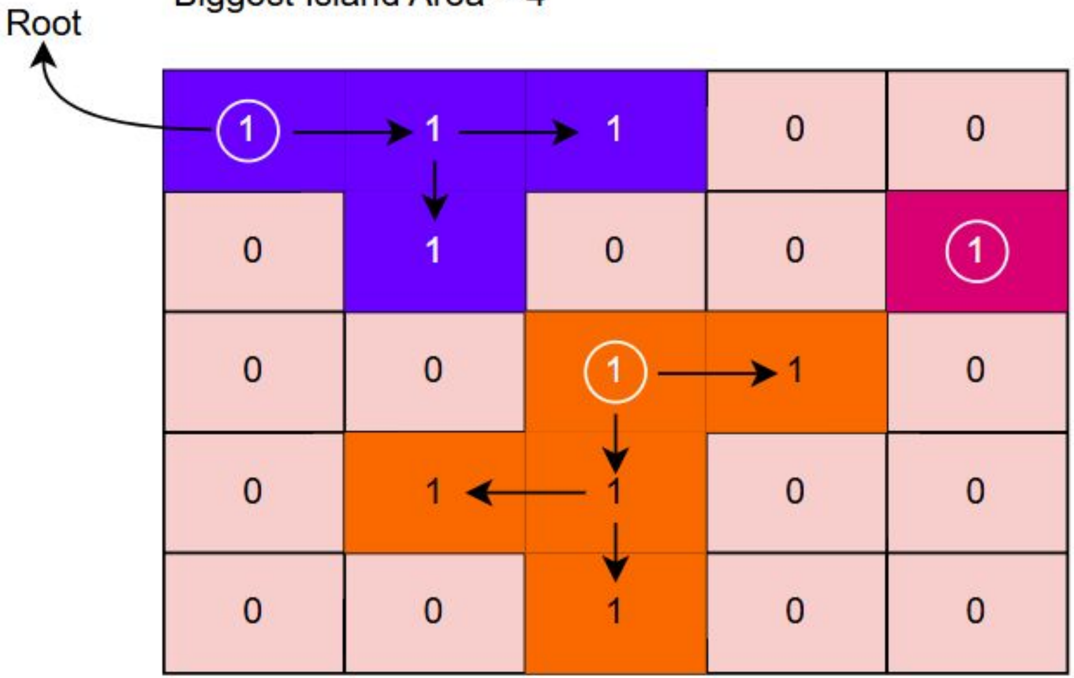
Biggest island among the different islands

1	1	1	0	0
0	1	0	0	1
0	0	1	1	0
0	1	1	0	0
0	0	1	0	0



Biggest island among the different islands

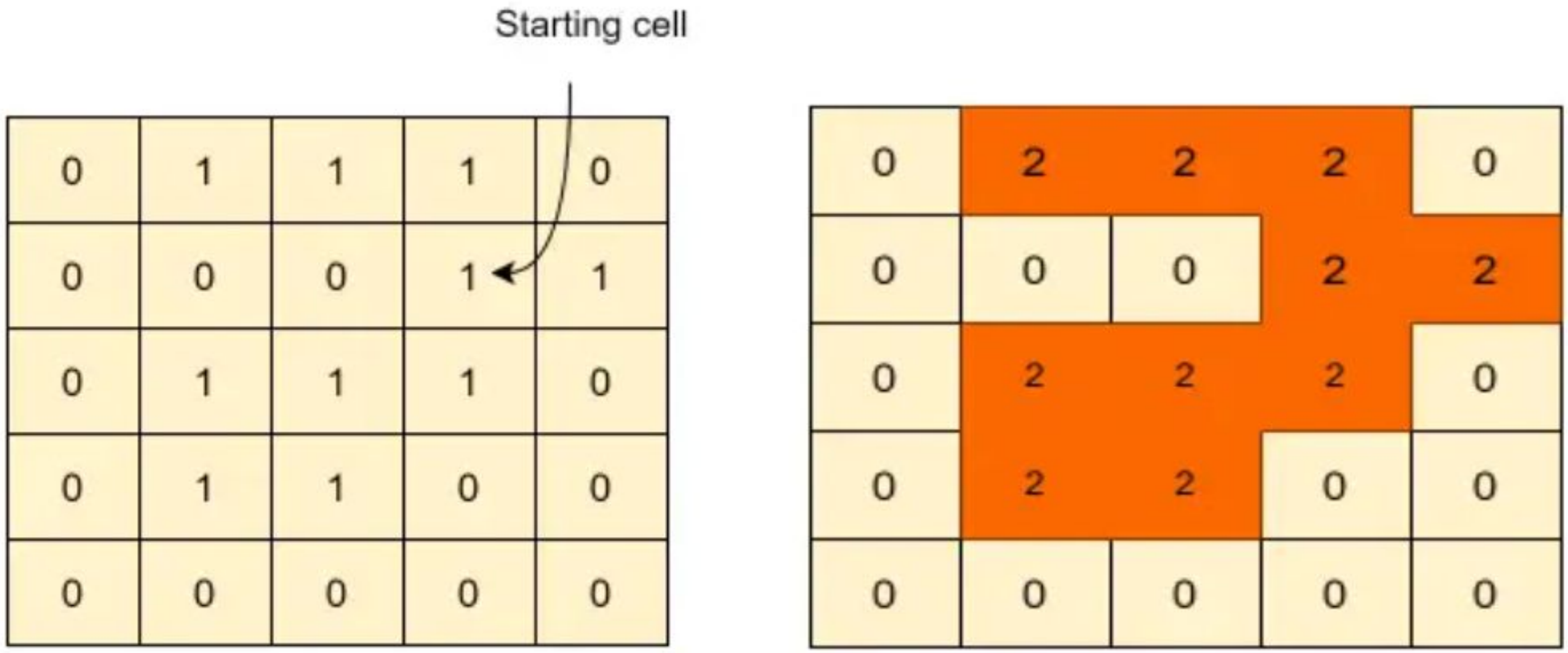
Biggest Island Area = 4



Biggest Island Area will not updated here because 4 is already max.

Here the Biggest Island Area=5 because 5 nodes are connected to each other according to the requirement.

Flood fill



starting cell = (1, 3)
new color = 2

Flood fill