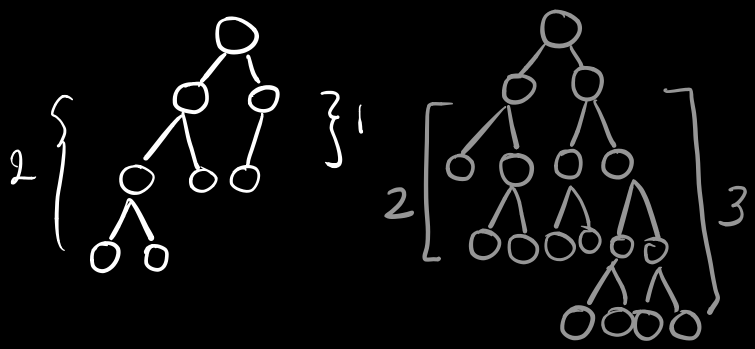
Check Balanced

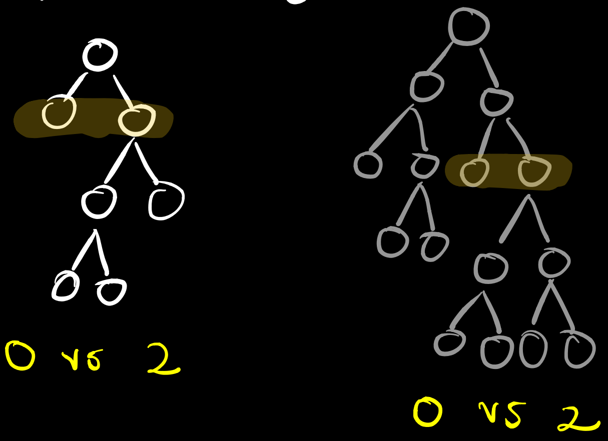
Implement a function to cheek if a binary tree is balanced.

Balanced tree: A tree such that the heights of the two subtrees of any node never differ by more than one.

Balanced tree

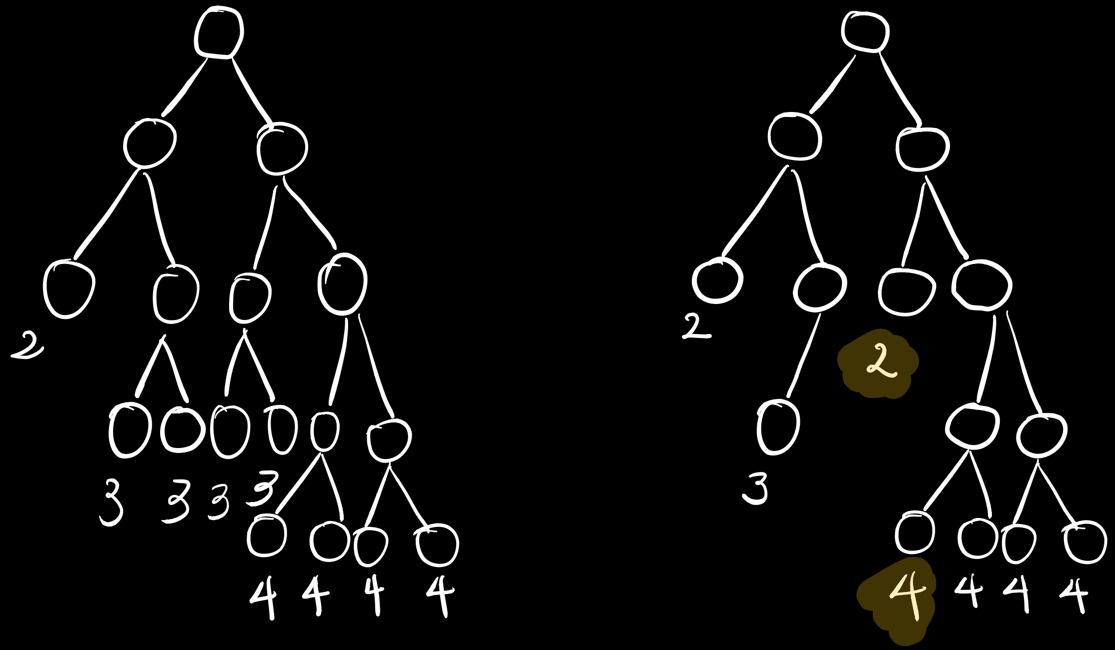


Unbalanced trees



Brute force ?

for each node, determine the heights of the left & right subtrees



DFS - for each path from the root, left to right, if we detect a difference > 1 between the subtrees, then the tree is unbalanced

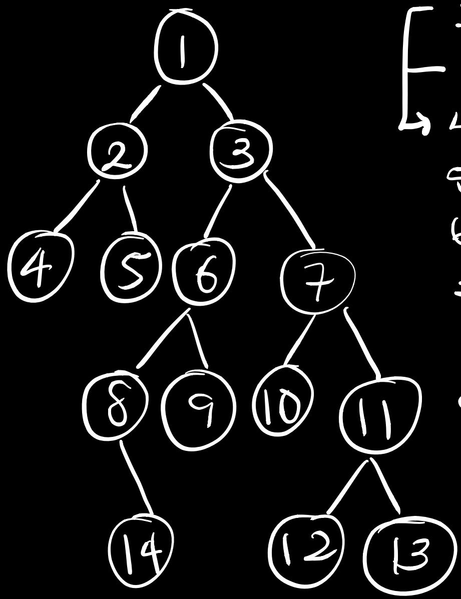
- Seems recursive? if we are returning to the previous node

- Base traversal would be left, right, root = postorder
- Have a separate function for counting height

```
function getHeight (node, currHeight){
      let leftHeight, rightHeight;
      if(node.left === null && node.right === null){
         return currHeight;
      }

      leftHeight = node.left !== null ?
             getHeight(node.left, currHeight+1) : currHeight;
      rightHeight = node.right !== null ?
             getHeight(node.right, currHeight+1) : currHeight;
      if( leftHeight -1 !== rightHeight ||      //heights are
         rightHeight -1 !== leftHeight){        // not ±1 of
         return false ;                         // each other
      }                                        // alternatively, use abs
      return (leftHeight > rightHeight) ?
                        leftHeight : rightHeight;

}


function isBalanced (root){
     return bool(this.getHeight(root,0));
}
```

let's test it!

given:



1) node = 1 · h = 0 · lh = gH(2, 0+1)
2) node = 2 · h = 1 · lh = gH(4, 1+1)
3) node = 4, return h = 2
4) node = 2 · h = 1 · lh = 2 . rh = gH(5, 1+1)
5) node = 5, return h = 2
6) node = 2, h = 1, lh = 2, rh = 2, return 2
7) node = 1, h = 0, lh = 2, rh = gH(3, 0+1)
8) node = 3. h = 1, lh = GH(6, 1+1)
9) node = 6, h = 2, lh = gH(8, 2+1)
10) node = 8, h = 3, lh = 3, rh = gH(14, 3+1)
11) node = 14, return 4
12) node = 8, h = 3, lh = 3, rh = 4, return 4
13) node = 6, h = 2, lh = 4, rh = gH(9, 2+1)
14) node = 9, h = 3, return 3
15) node = 6, lh = 4, rh = 3, return 4
16) node = 3, h = 1, lh = 4, rh = gH(7, 1+1)
17) node = 7, h = 2, lh = gH(10, 2+1)
18) node = 10, h = 3, return 3
19) node = 7, h = 2, lh = 3, rh = gH(11, 2+1)
20) node = 11, h = 3, lh = gH(12, 3+1)
21) node = 12, return 4
22) node = 11, lh = 4, rh = gH(13, 3+1)
23) node = 13, return 4
24) node = 11, lh = 4, rh = 4, return 4
25) node = 7, lh = 3, rh = 4, return 4
26) node = 3, lh = 4, rh = 4, return 4
27) node = 1, lh = 2, rh = 4, return false //end

Complexity

Time: O(N) where N = the number of nodes in the tree

Space: O(h) where h = the height of the tree, due to the recursive calls being made