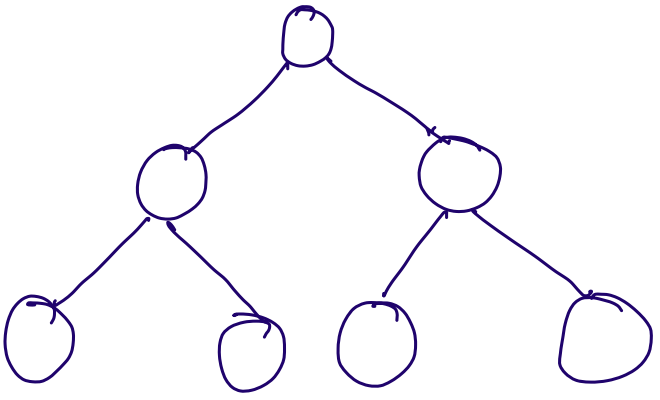
① Invert Binary tree

→



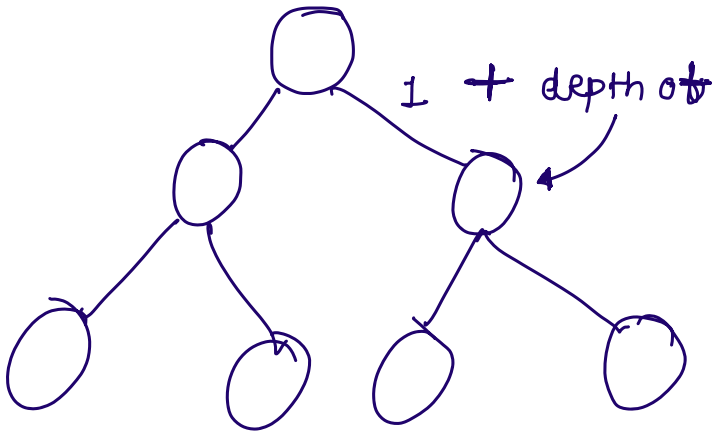
Swap root.left & root.right.

& now let recursion do the work.

invertTree (root.left)
invertTree (root.right)

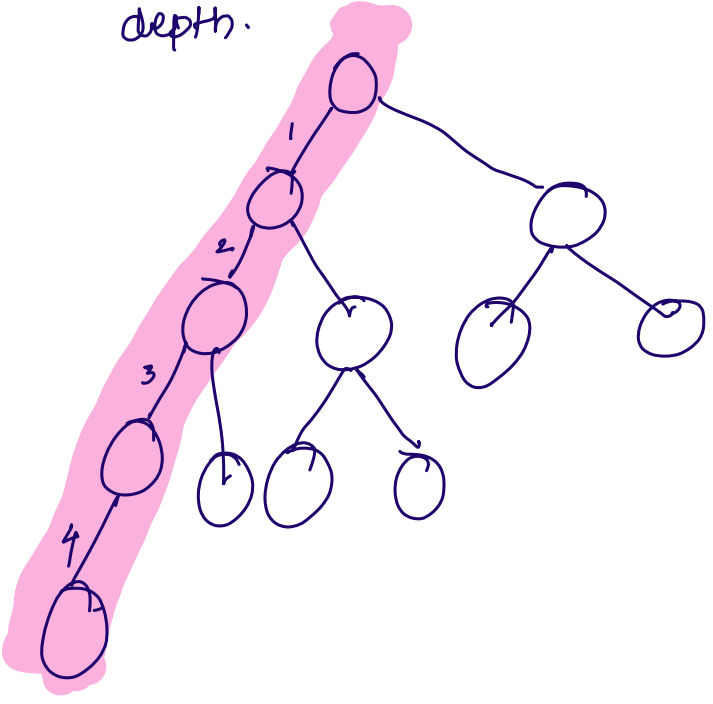
② Maximum depth of binary tree -

⟶



1+ maxdepth (root.left)

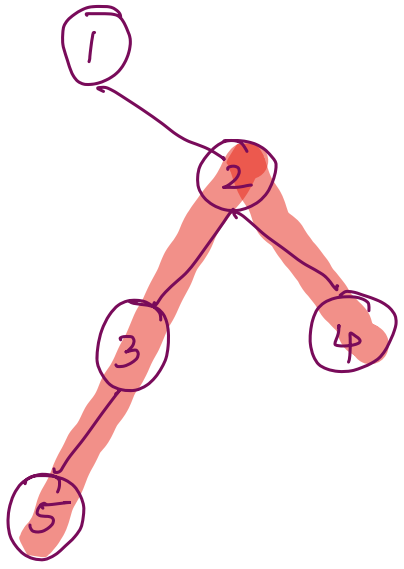
1+ maxdepth (root.right)



left = height (n.left)

right = height (n.right)

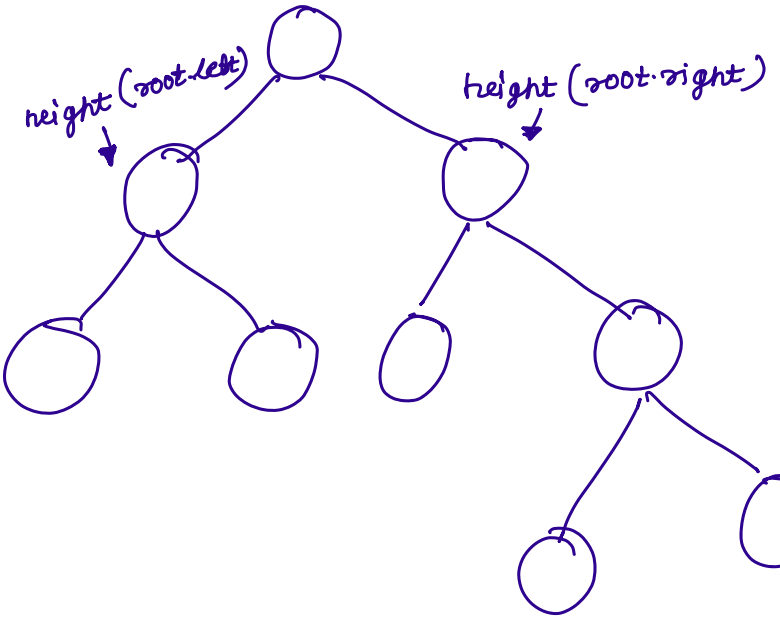
③ Diameter of Binary tree -



- diameter would be longest distance between two nodes

```
public int height (TreeNode n, int [] arr)
    int left = height (n.left, arr)
    int right = height (n.right, arr)
    arr[0] = max (left, right)
        1 + max (left, right)
```

(4) Balanced Binary tree -



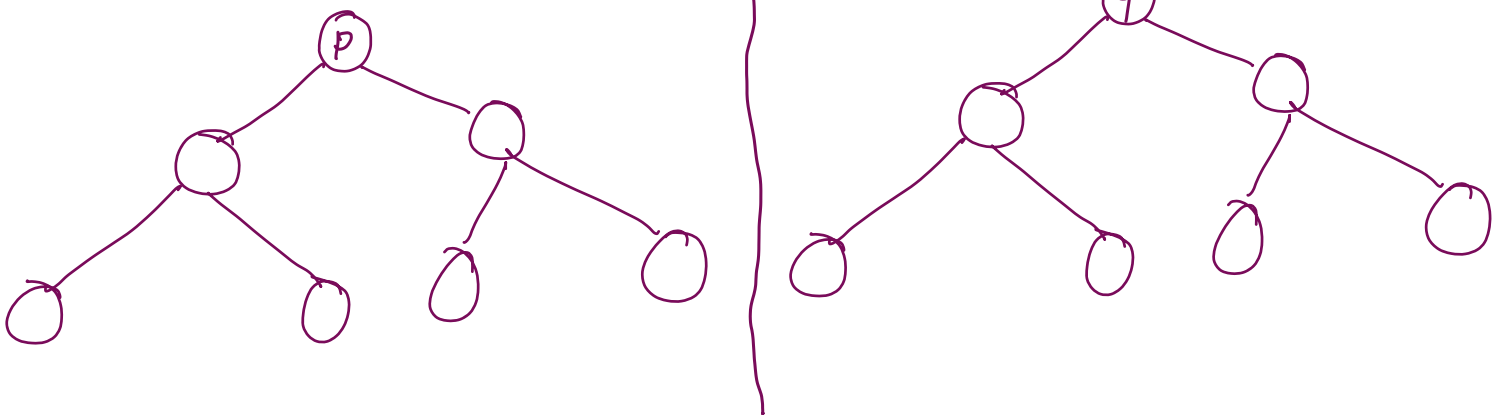
it

height (root.left) - height (root.right) > 1

return false;

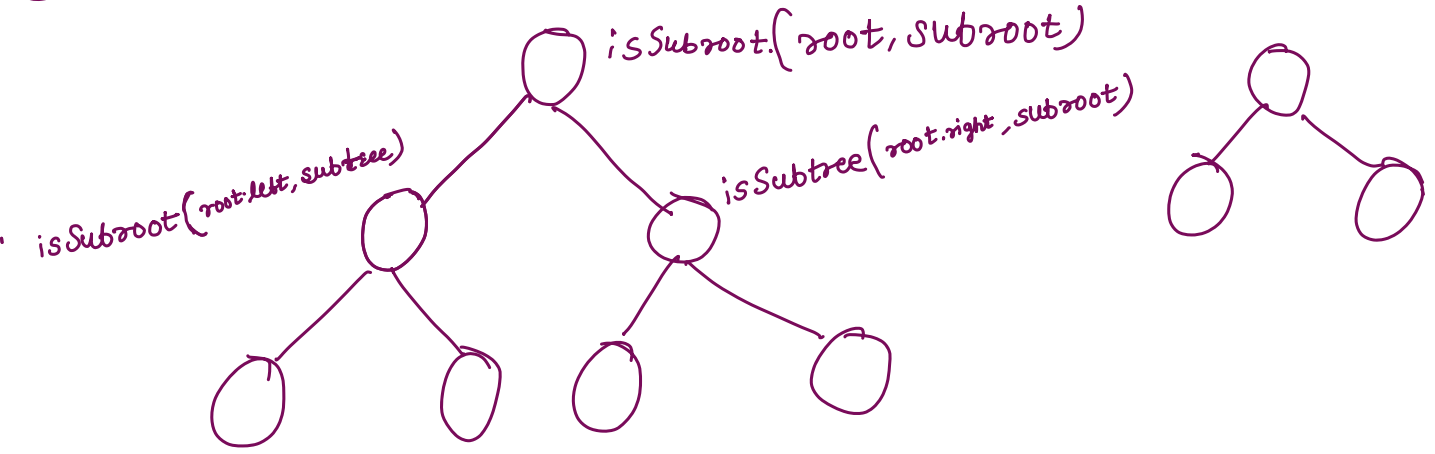
return isbalance (root.left) && isbalance (root.right

(5) same tree

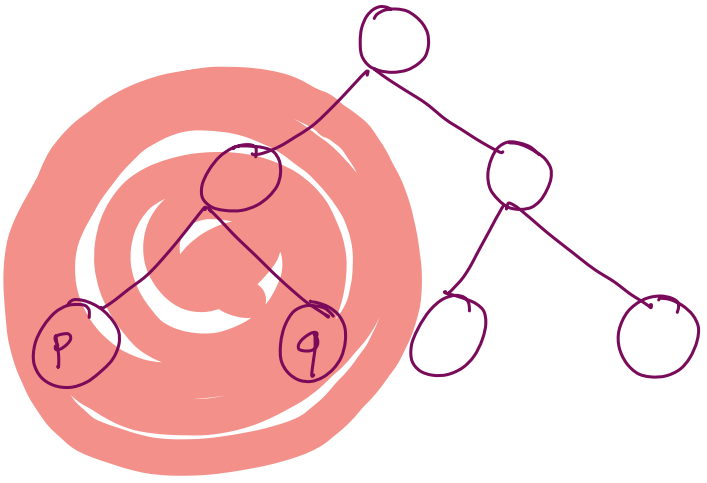



```
if (p == q)
    if recursion(p.left, q.left) &&
       recursion(q.right, q.right)
    return false.
else
    true.
```

⑥ Subtree of another tree



(7) Lowest common ancestor



if p < root.val & q < root.val

{ recursion(root.left, p, q)

suppose we are at root.

if p & q both are less than root.val

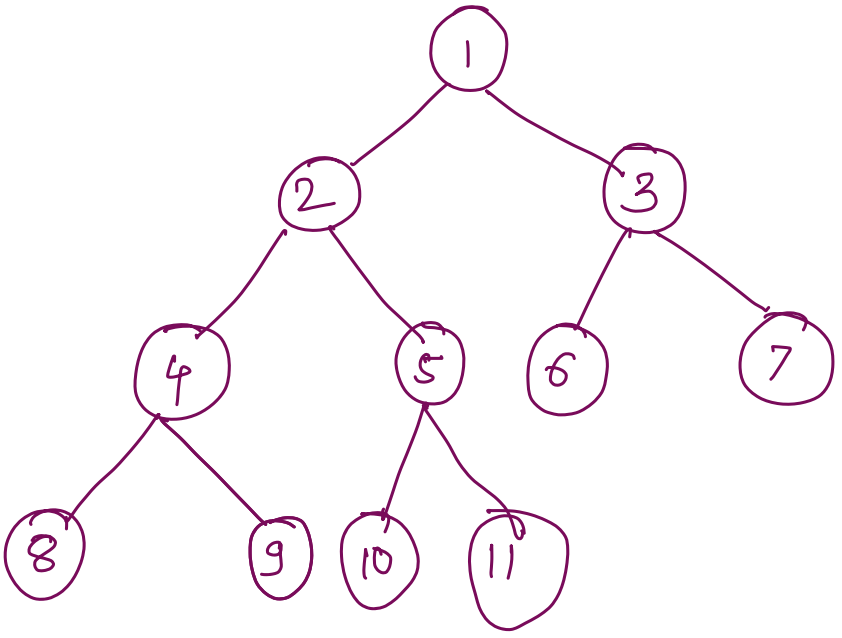
we know they would be in left side

so, we apply recursion on root.left.

likewise, if p & q are greater than root.val

we apply recursion on right side of tree.

⑧ Level order traversal -.



queue.

put root in queue.

1

queue-

put the neighbour

remove

1 2 3

queue-

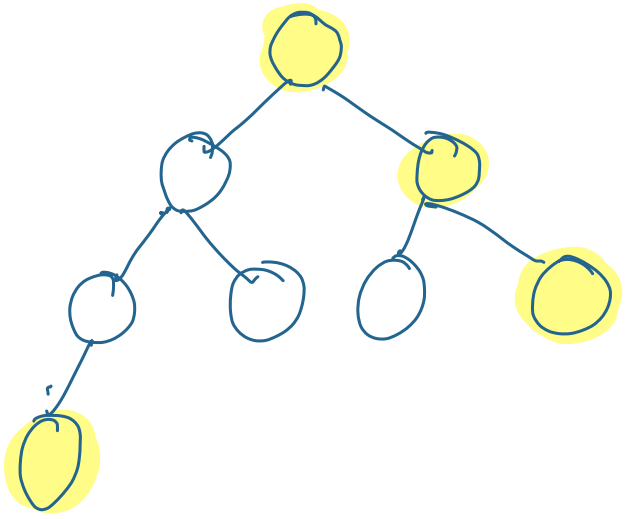
1

2 3

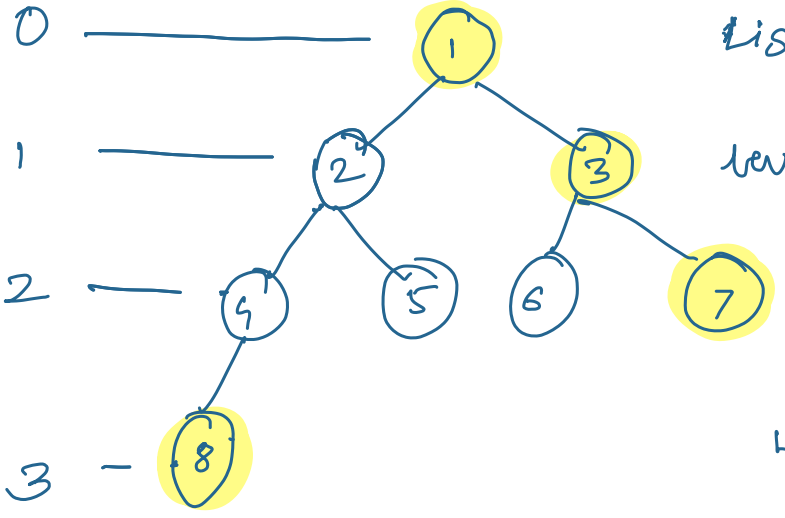
for '2' times call neighbours at 2 & 3.

3

(10) Binary tree right side view



we go right
because we want right side view.
&
if the level we are on matches the size of our result list
we add that node.



List[1].

level = 1 & List.size() = 1 = [1,3]

level = 2 & List.size() = 2
[1,3,7]

Level = 3 & List.size = 3.
[1,3,8].

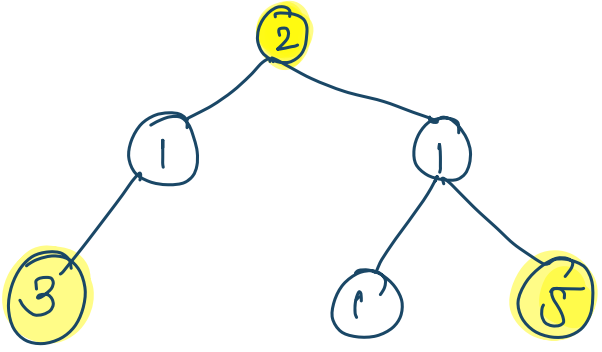
Base condition → level == list.size()

add

recursion(root.right, list, level+1)
recursion(root.left, list, level+1)

(11) Count good nodes in binary tree :

Good → path to that node from root should be like there should not be any node bigger than that node.

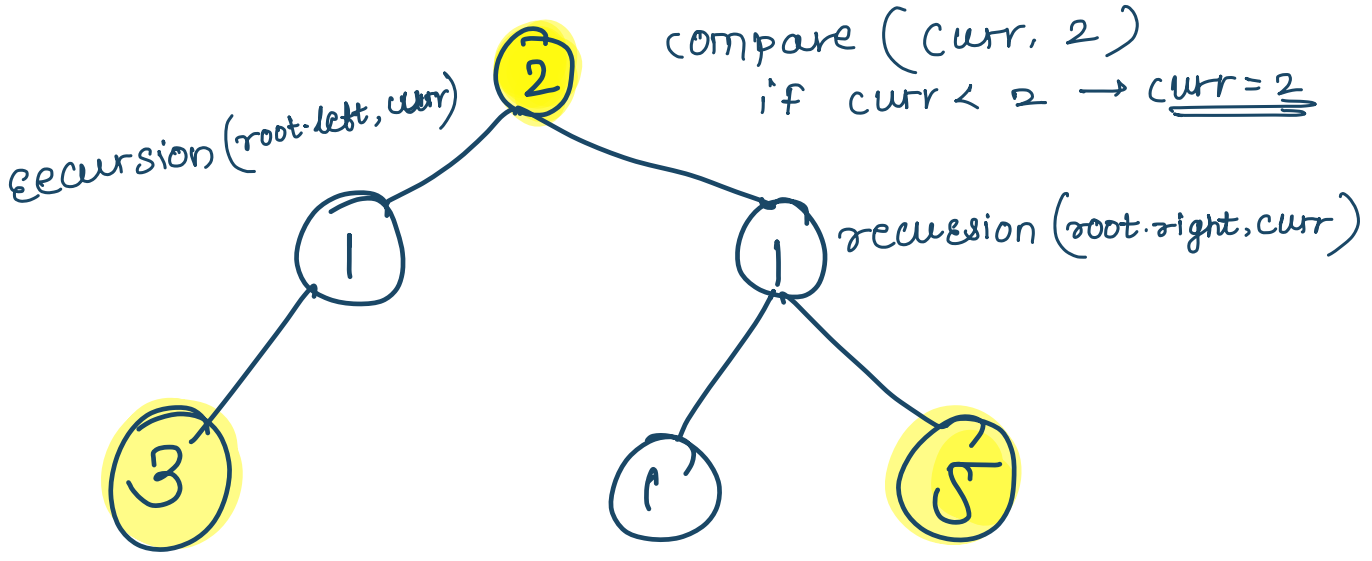


we will declare a variable curr = Integer.min.

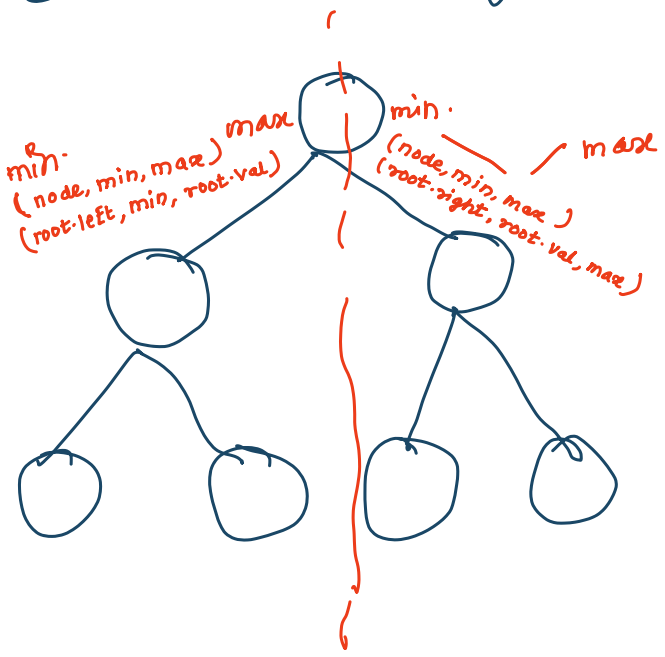
Base Case → if root == null return 0;

curr. = Int. max

compare (curr, 2)
if curr < 2 → curr = 2



(12) Validate Binary search tree.



Start with two values

min → min. Integer

max → max. Integer.

Recursion (root, min, max)

(13) Construct Binary tree from preorder & Inorder.

preorder = [3, 9, 20, 15, 7]

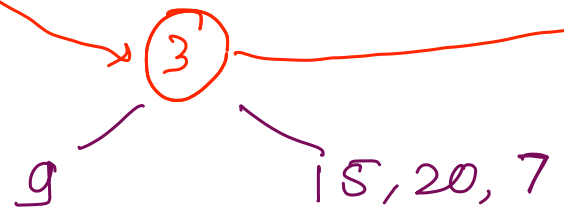
Basically Preorder → root, left, right

Preorder will give us root.

once we know the root Inorder helps us figure out which elements are in right side of tree & left side of tree

preorder = [3, 9 20, 15, 7] Inorder = [9, 3, 15, 20, 7]

left right.



tree builds up like this -

Eg

Preorder

[1, 2, 4, 5, 3, 6]

Inorder

[4, 2, 5, 1, 3, 6].

if you decide root → '1'

& see the index of root in inorder

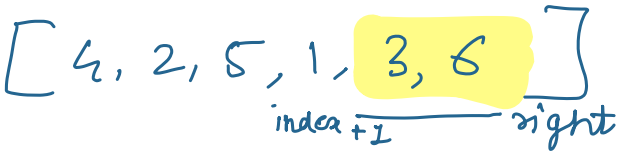
[4, 2, 5, 1, 3, 6]

Now we know for the left side of '1' we would only search the order from

[4, 2, 5, 1, 3, 6]

left index-1

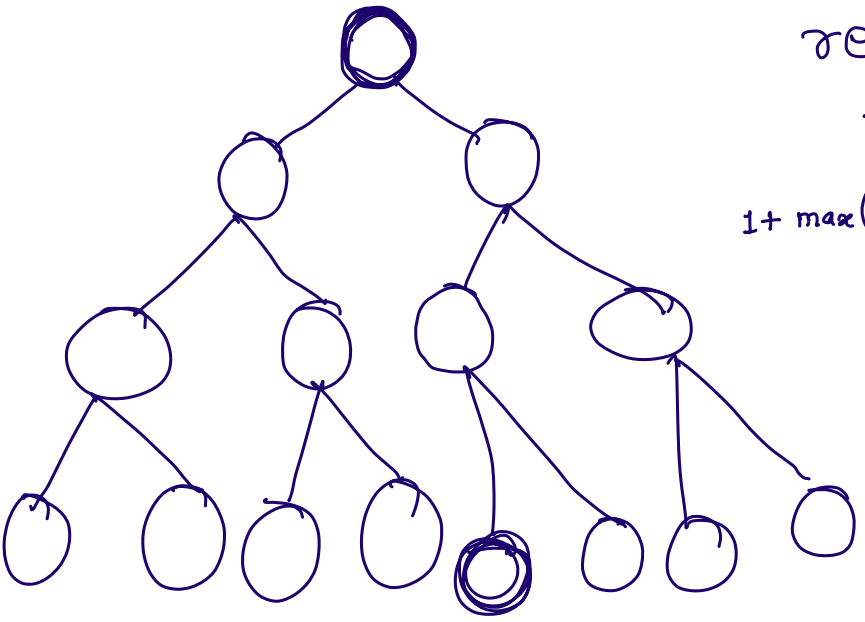
Similarly for the right side of the root (1) we would only search in range (index+1, right)



so,

we would form the root & make two recursive calls to the left of root & right of root.

(14) Binary tree maximum path sum -



recursion →

for root →

1 + max(recursion(root.left, max), recursion(root.right, ma

max = max(max, left + right + x.val)