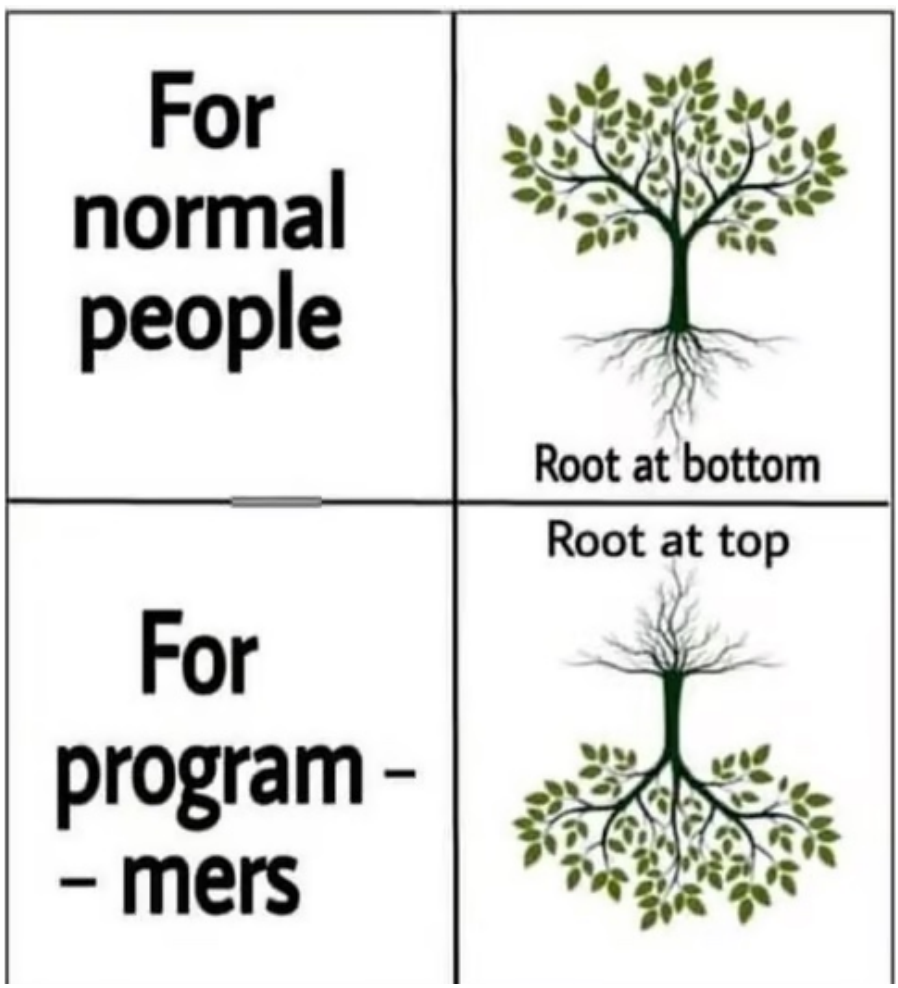
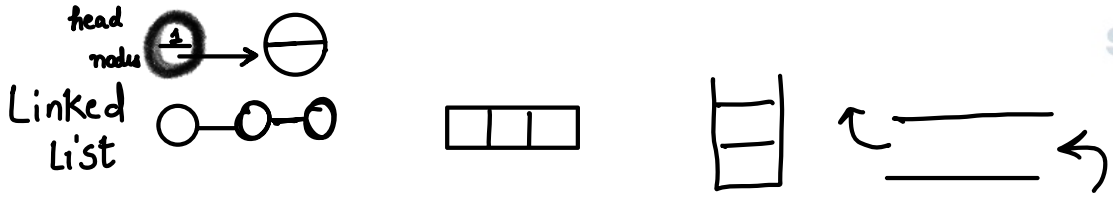
SCALER

Trees - 1

TABLE OF CONTENTS



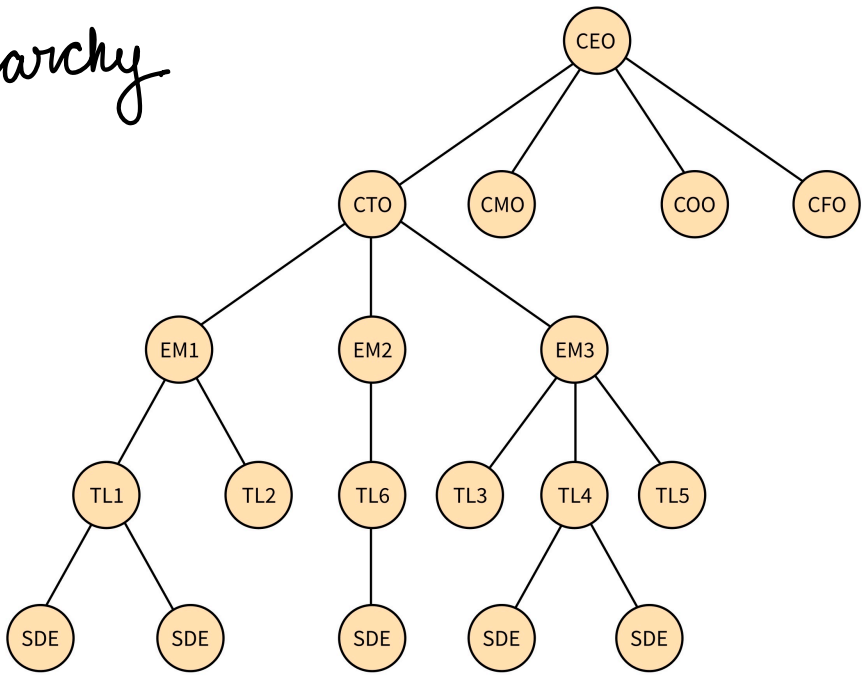


Trees

NonLinear

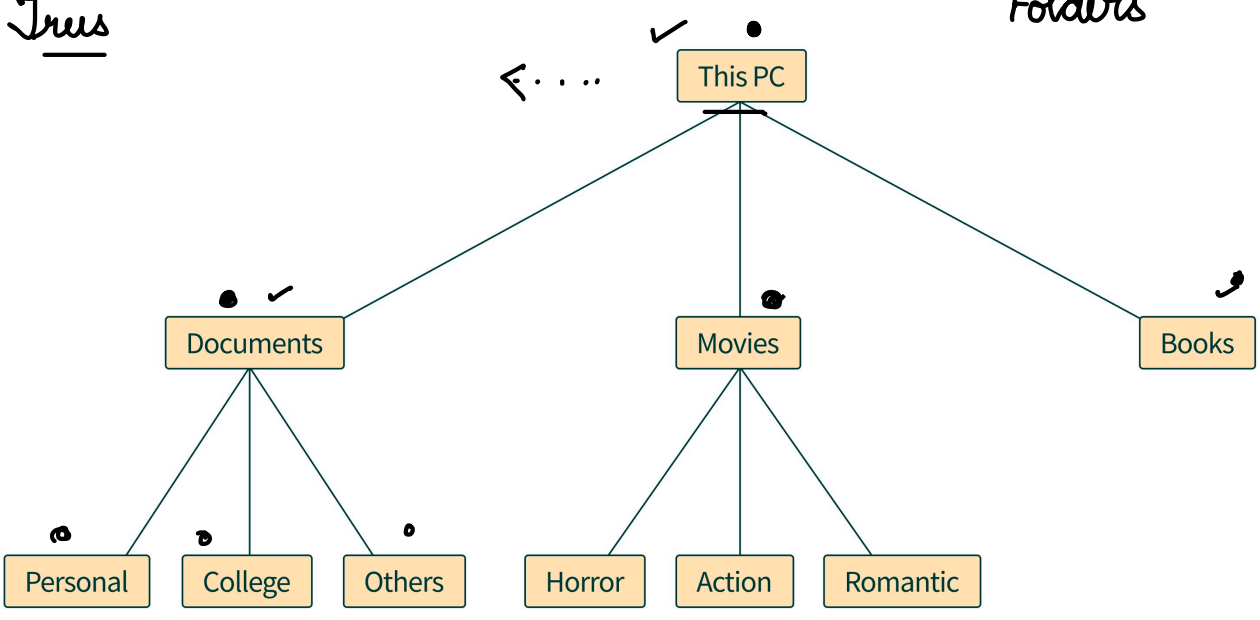
hierarchy

OFFICE

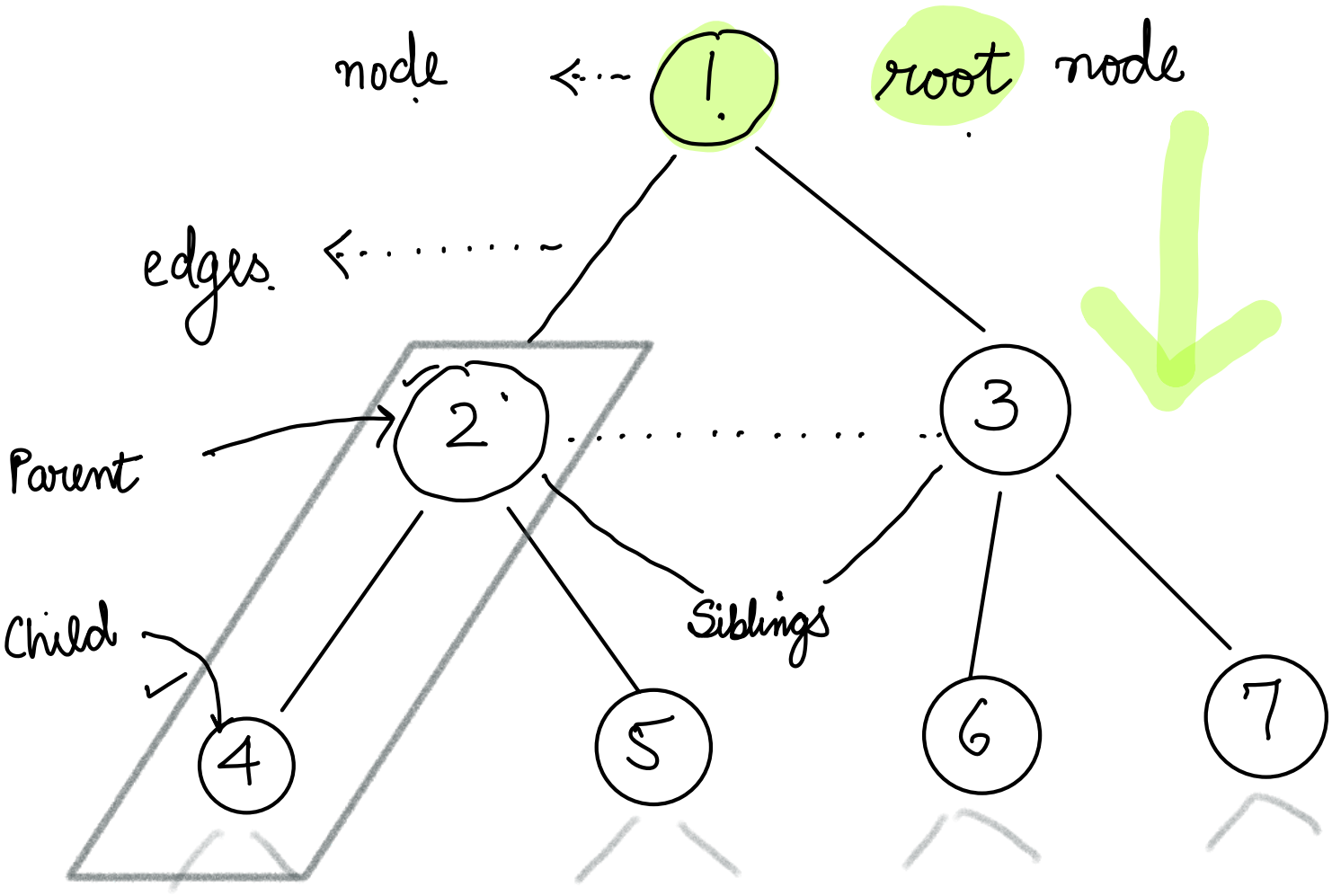


Trees

Folders



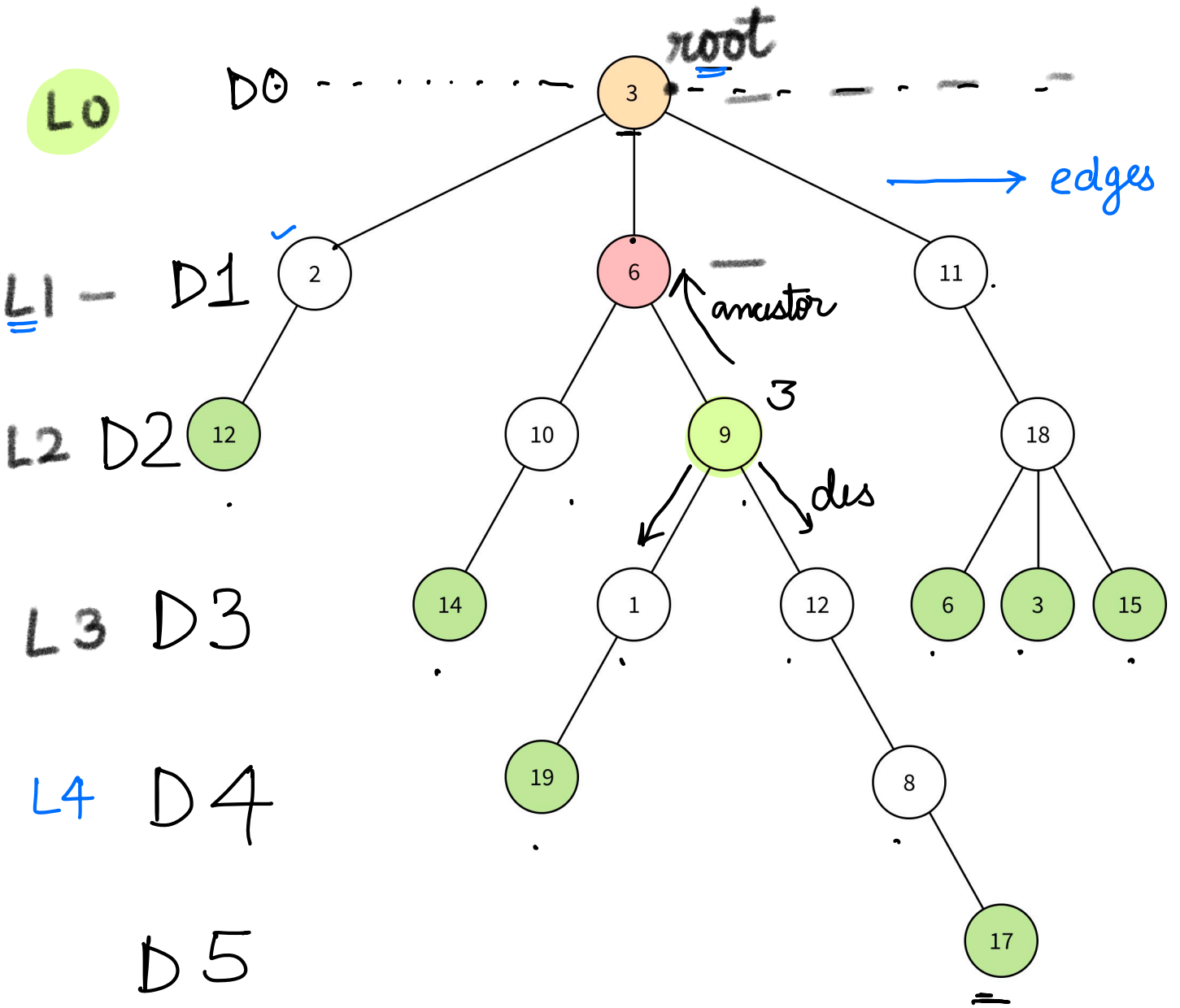
height &
depth



Leaf node = 0 children

Sibling

data - int



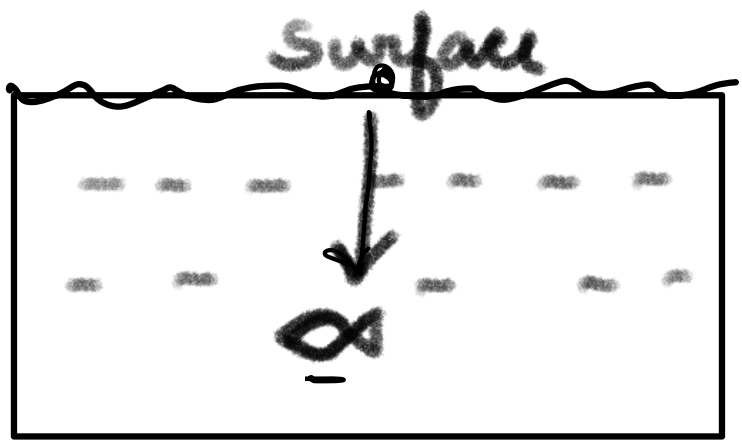
11 - Parent

18 - child node

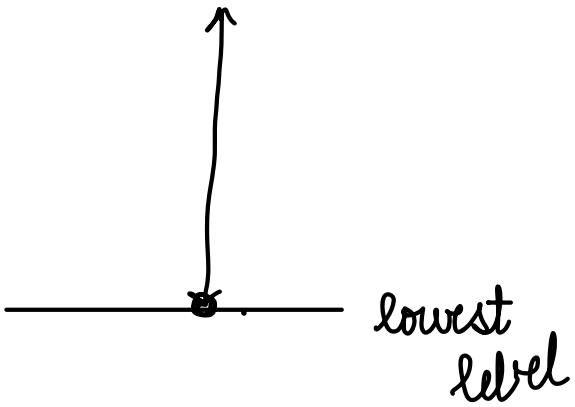
10 - 9 - 18

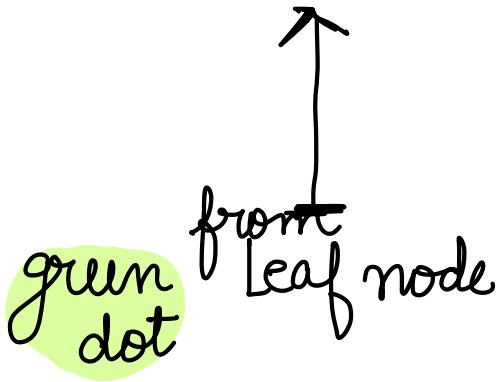
Siblings

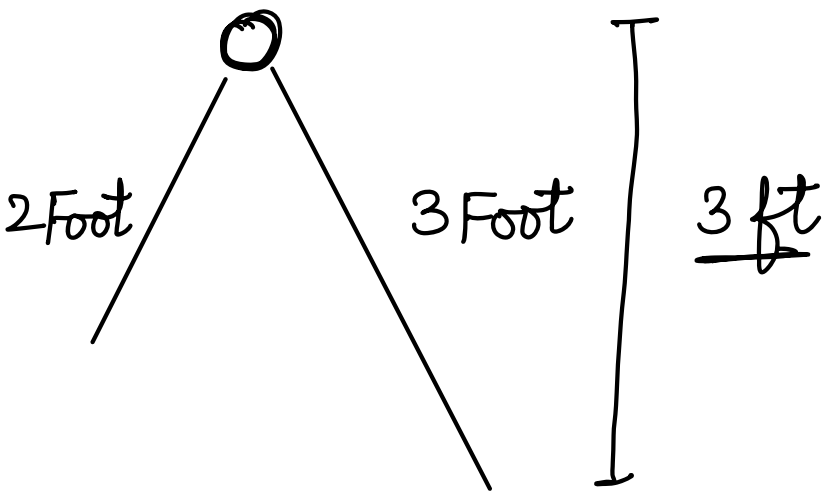
Depth



height :



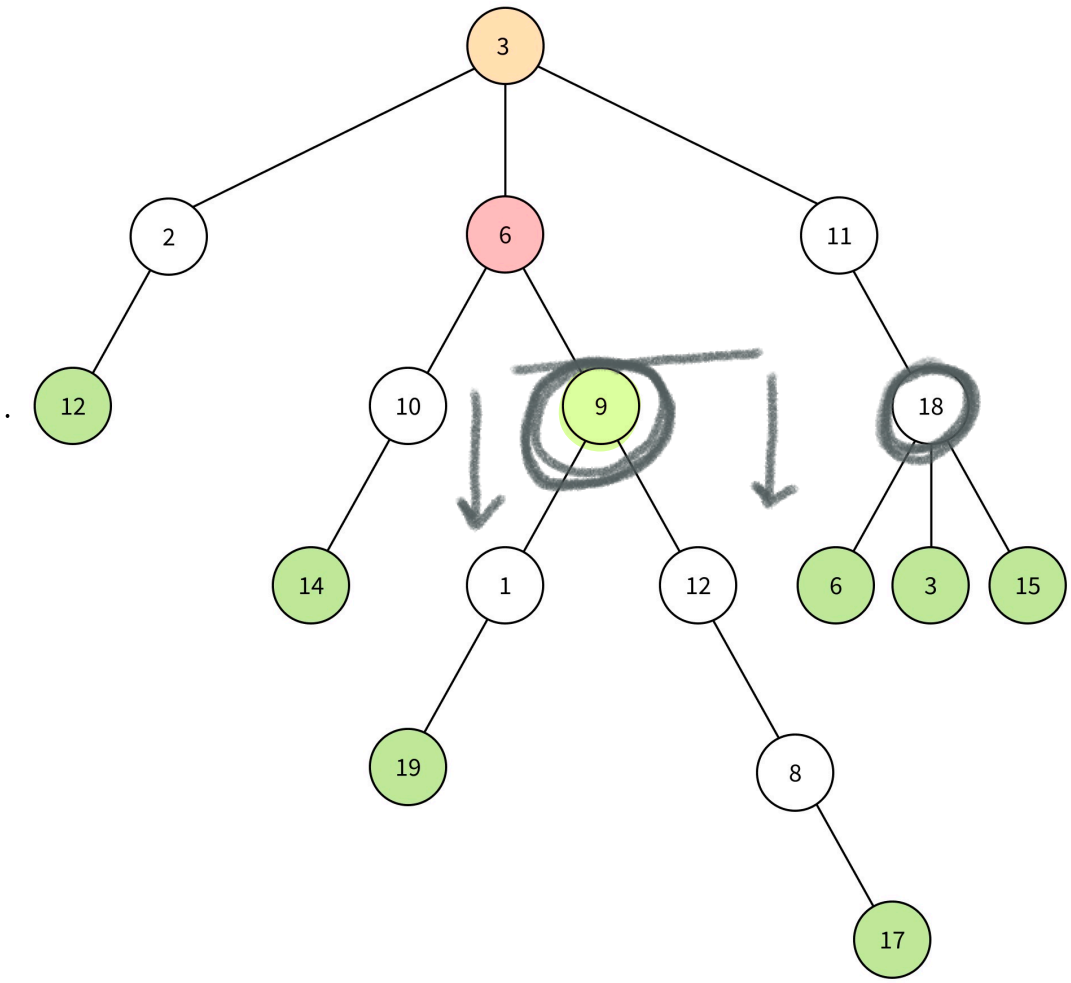




Top ——— Depth of node = from root

height = max dis from leaves

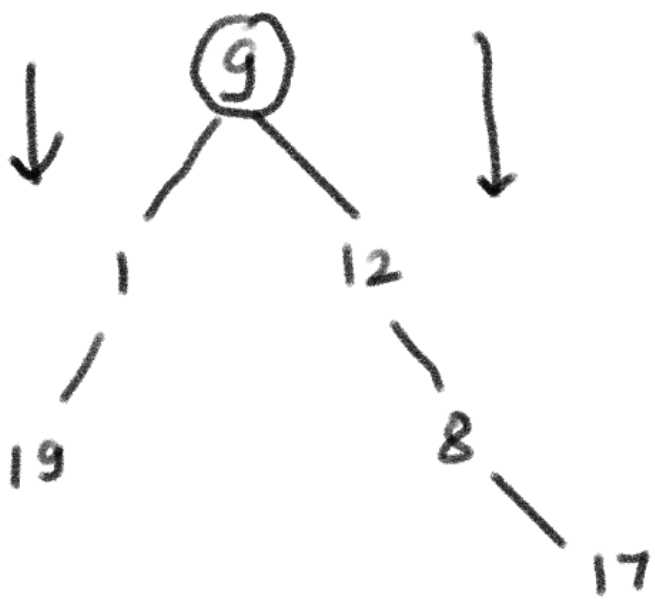
down ..... l1 ..... l2 .....



Subtree at 9

↓

any part of tree



Can leaf be a subtree?

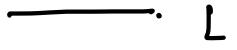
Yes

Do all nodes have parent node? No.

Who is root's parent? — No parent

⇒ Depth of root node = 0

⇒ height of leaf node = 0



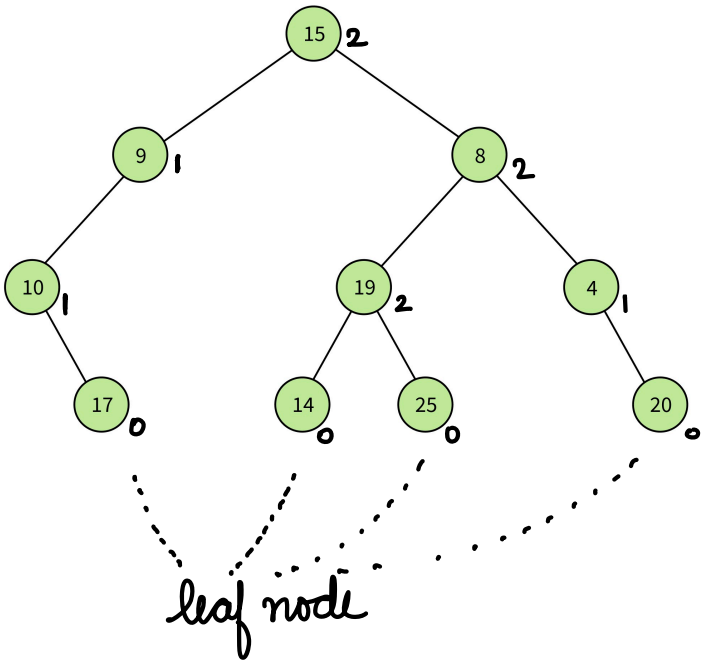
. . 2 → l , r

. . — 1 either l or r

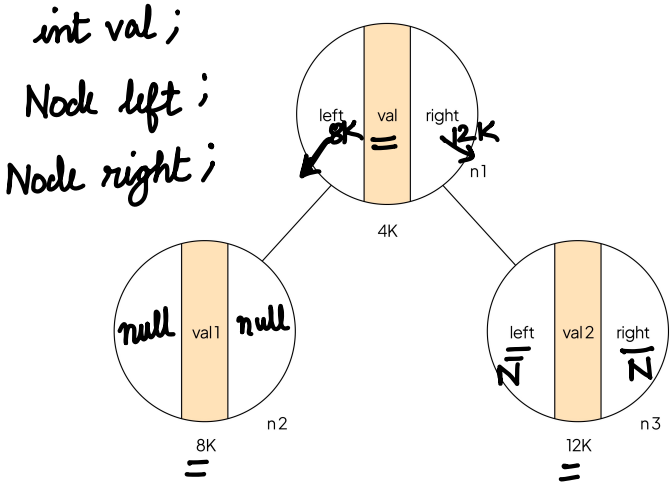
2, 1, 0 . . . . . l = null
r =

Binary Tree — maximum 2 connections

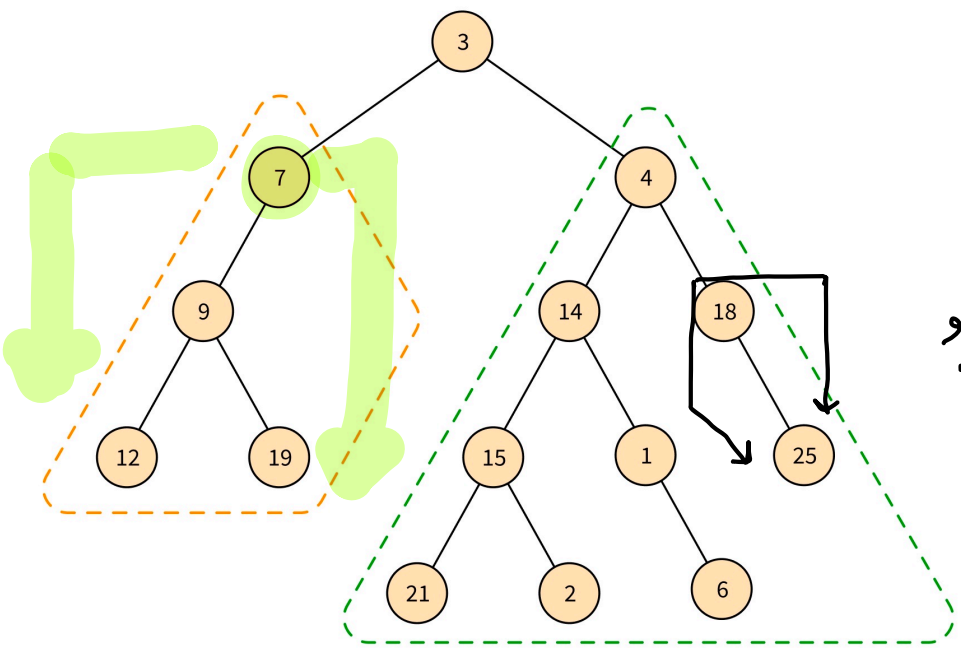
2




```
class Node {
    int val;
    Node left;
    Node right;
}
```



Sub-tree



subtree rooted at 18

18

25

rooted at 1

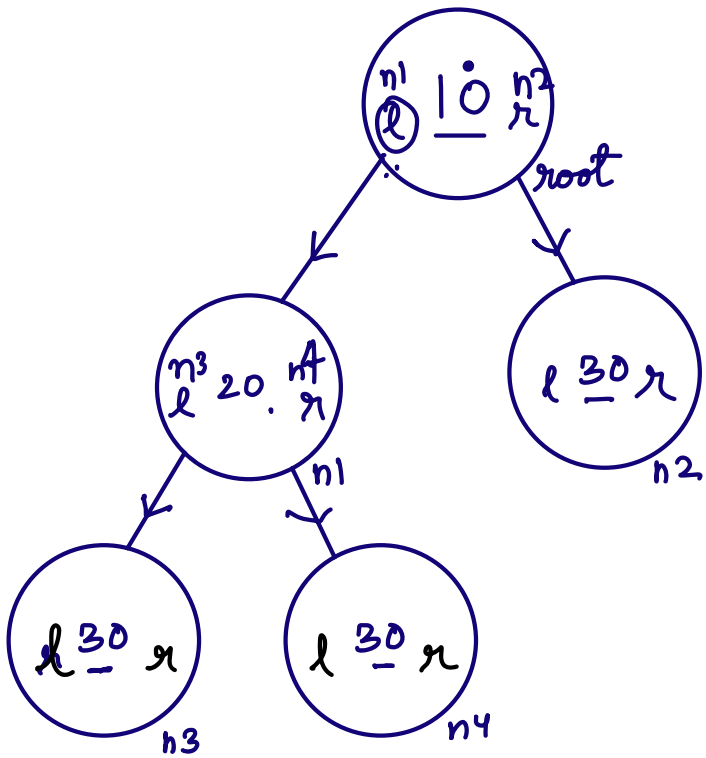
1

6

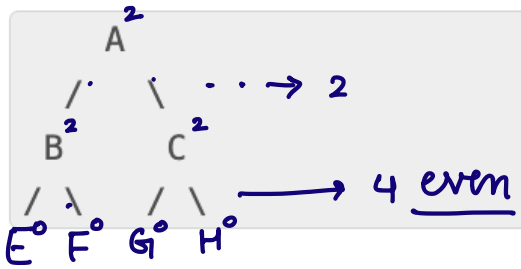
```python
class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None


print("Hello, World!")
root = Node(10)
n1 = Node(20)
n2 = Node(30)
n3 = Node(30)
n4 = Node(30)

root.left = n1
root.right = n2
n1.left = n3
n1.right = n4
```

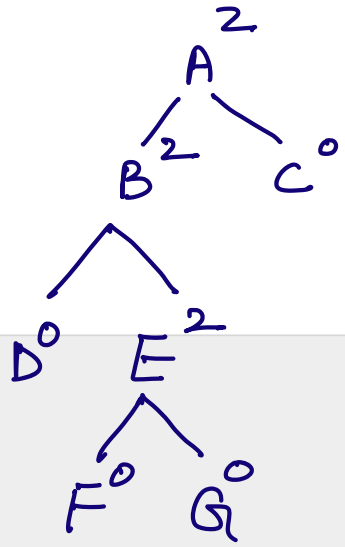
class Node {

def __init__(self, data):
self.data = data
self.left = none
self.right = none

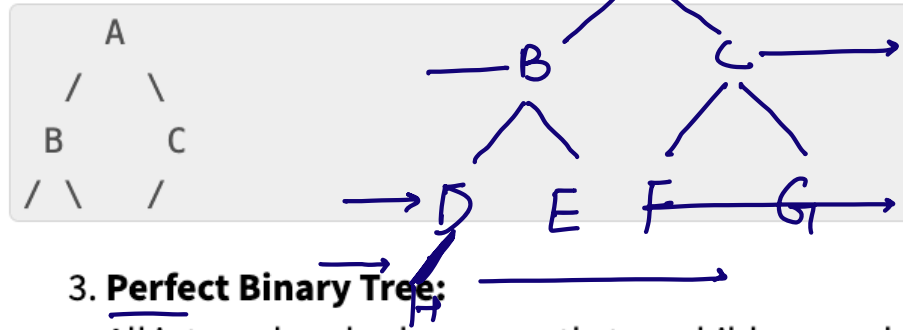


1. **Proper Binary Tree (Strict Binary Tree):**
   Every node has either 0 or 2 children (never 1 child).
   Diagram:

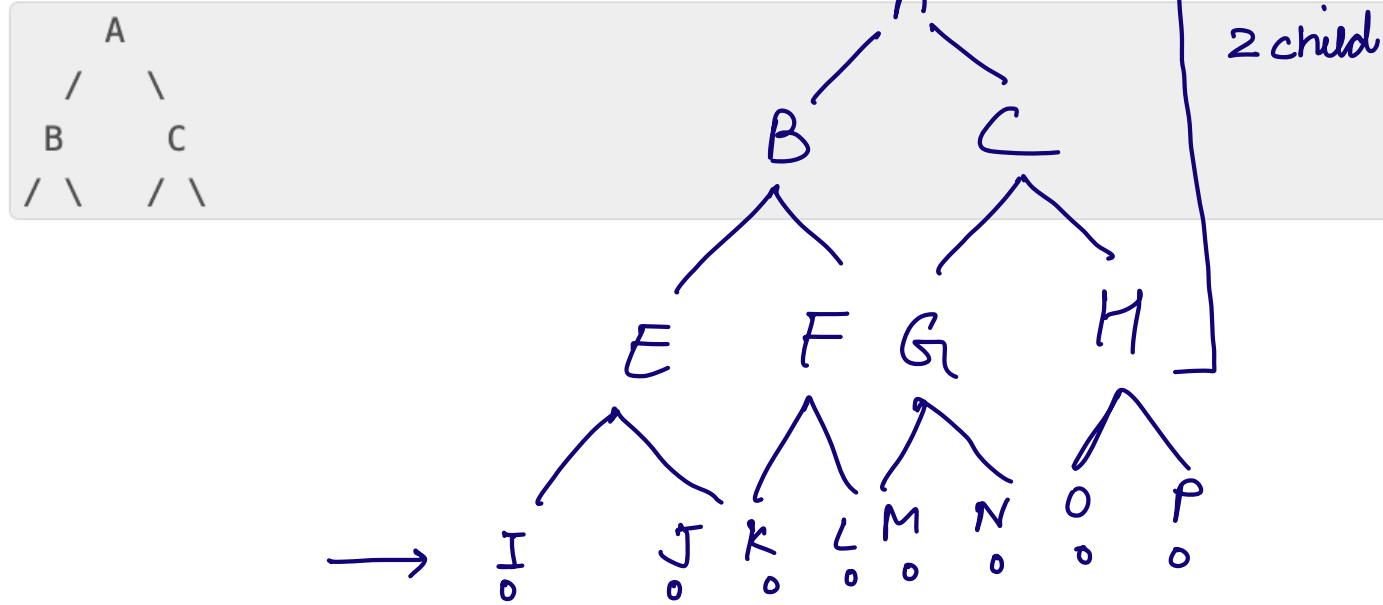




2. **Complete Binary Tree:**
   All levels are filled except possibly the last level, which is filled left to right.
   Diagram:

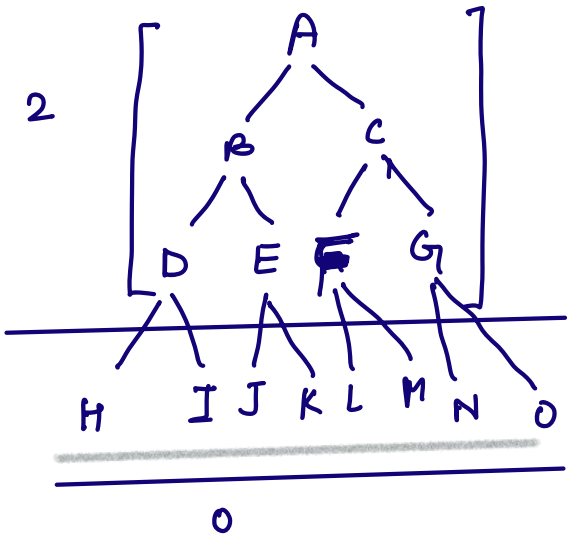


3. **Perfect Binary Tree:**
   All internal nodes have exactly two children, and all leaf nodes are at the same level.
   Diagram:



Quiz:

Perfect binary



2

0

Proper → 2

0

Complete Binary tree

Recursion - 1 level down

```
P  void traverse_tree (Node root) {
```

traverse_tree (root.left)

traverse_tree (root.right)

print (root)

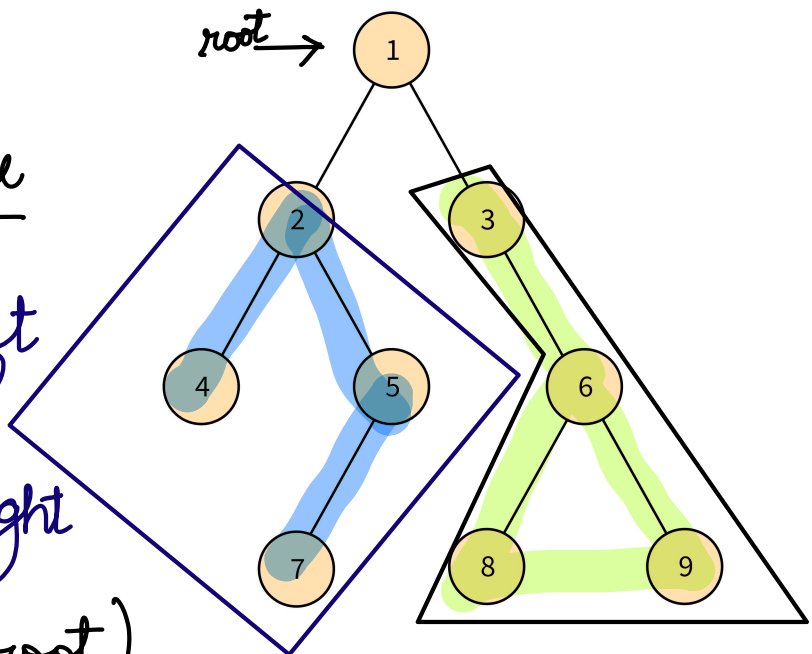
```
}
```

P - print tree

SP1 = root.left
2 4 5 7

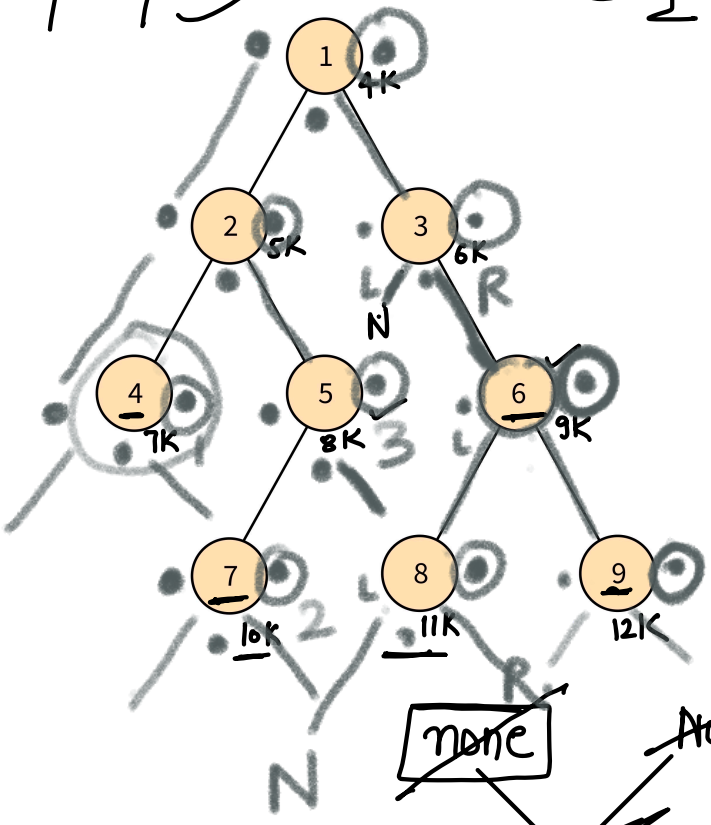
SP2 = root.right
3 6 8 9

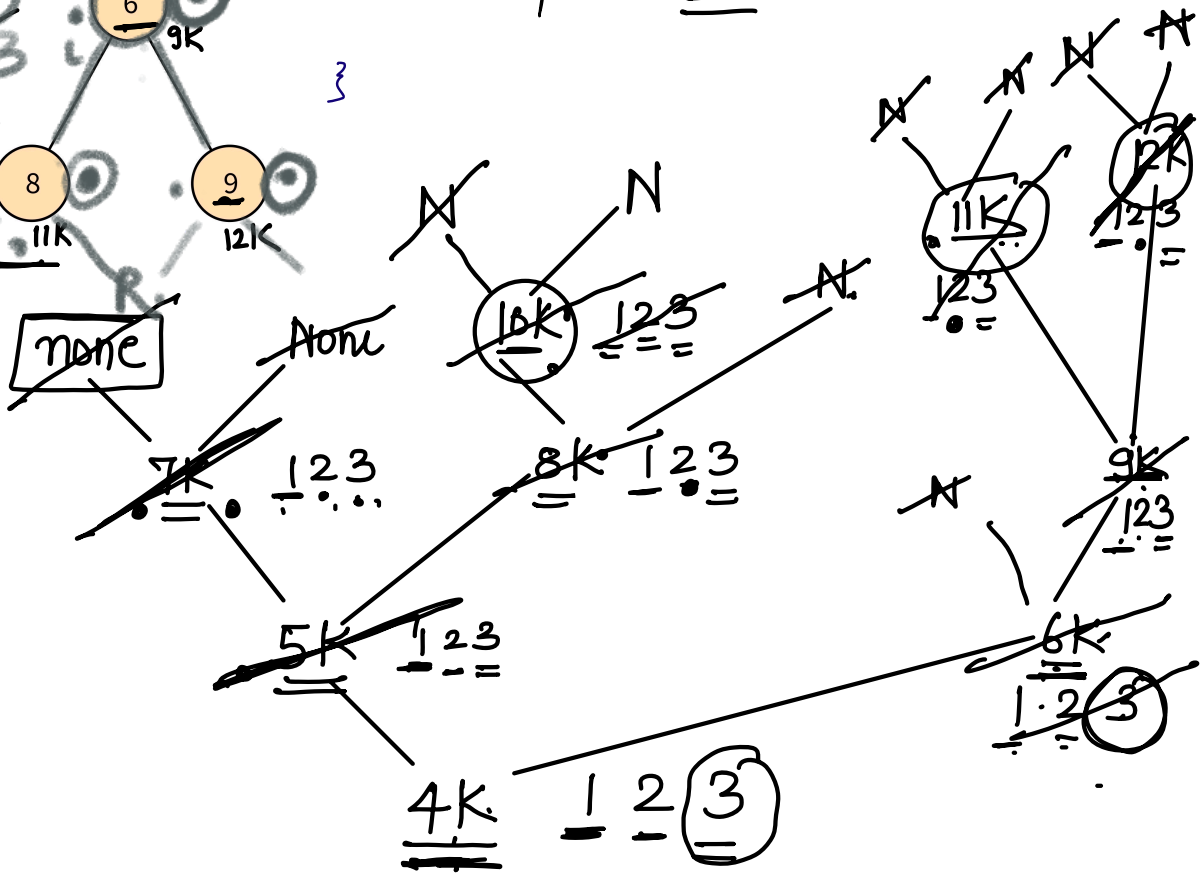
EW = print (root)

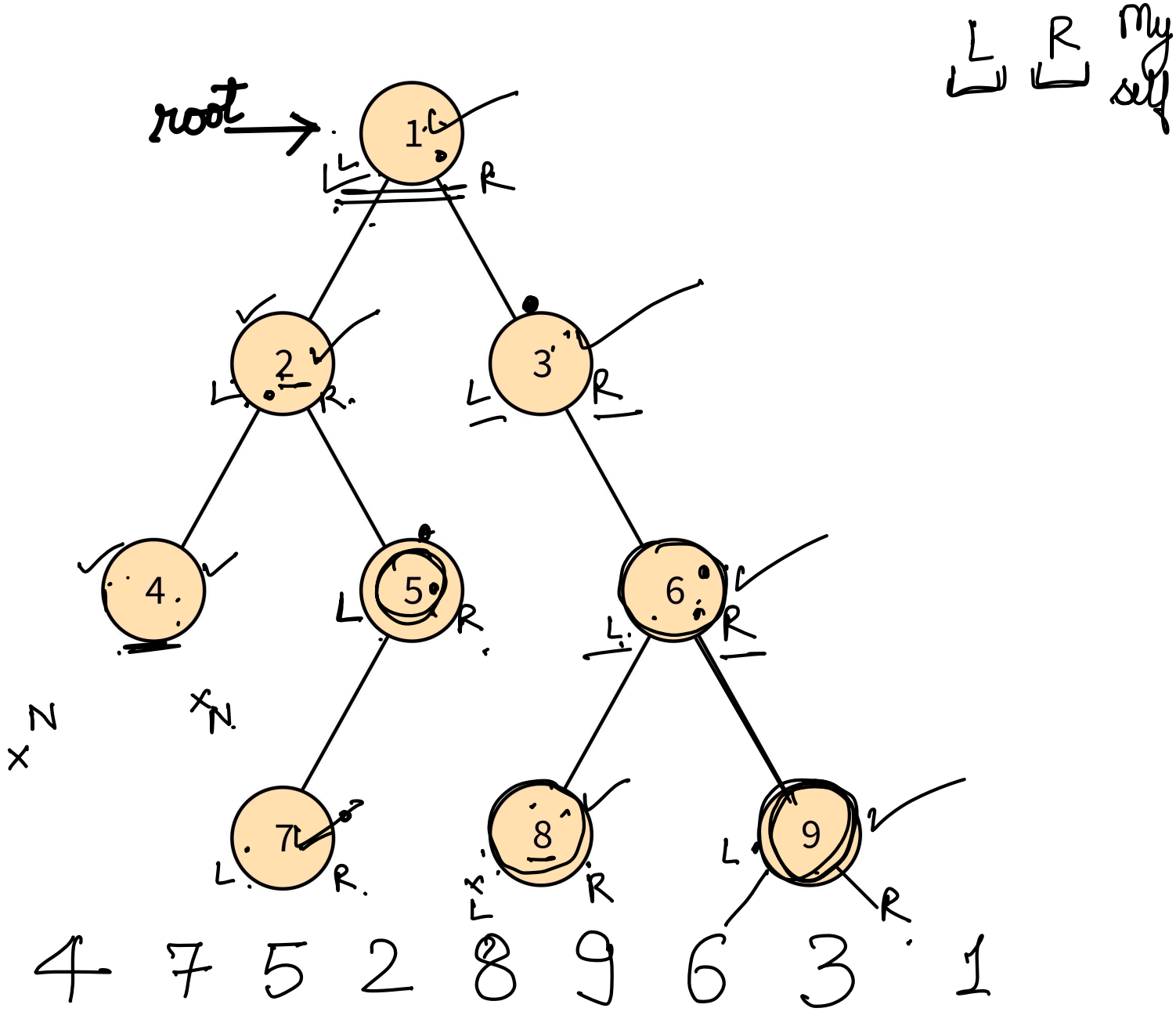


4 7 5 2 8 9 6 3 1

```
void traverse_tree (Node root) {
    if root == None : return
    traverse_tree (root.left)   1
    traverse_tree (root.right)  2
    print (root.data)           3
}
```






L R My self
root
1
2
3
4
5
6
7
8
9
4 7 5 2 8 9 6 3 1

P

```
void printTree(Node root) {
    [redacted]
SP1 printTree(root.left)
SP2 printTree(root.right)
→   print(root.data)
}
```

```
void traverse_tree(Node root) {
    if root == None : return
    traverse_tree(root.left)   1
    traverse_tree(root.right)  2
    print(root.data)           3
}
```

Post order

```
void traverse_tree(Node root) {
    if root == None : return
    traverse_tree(root.left)   1
    print(root.data)           2
    traverse_tree(root.right)  3
}
```

InOrder

```
void traverse_tree (Node root) {
    if root == None : return
    print (root.data)
    traverse_tree(root.left)
    traverse_tree (root.right)
}
```

pre ✓

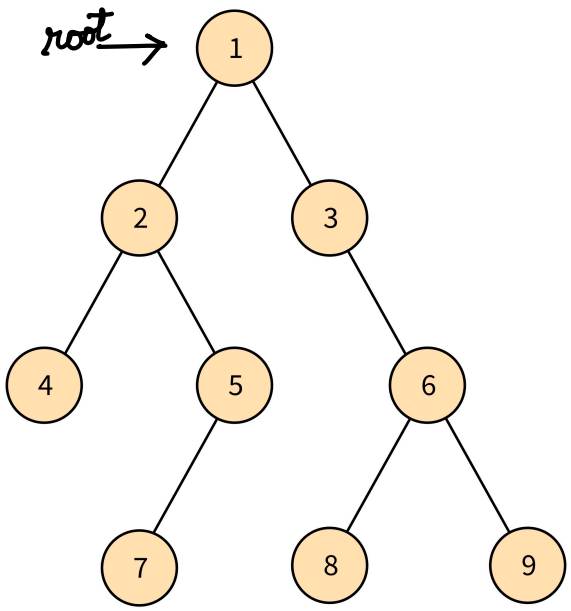
Binary Tree Traversal

Pre-order Traversal - Root Left Right

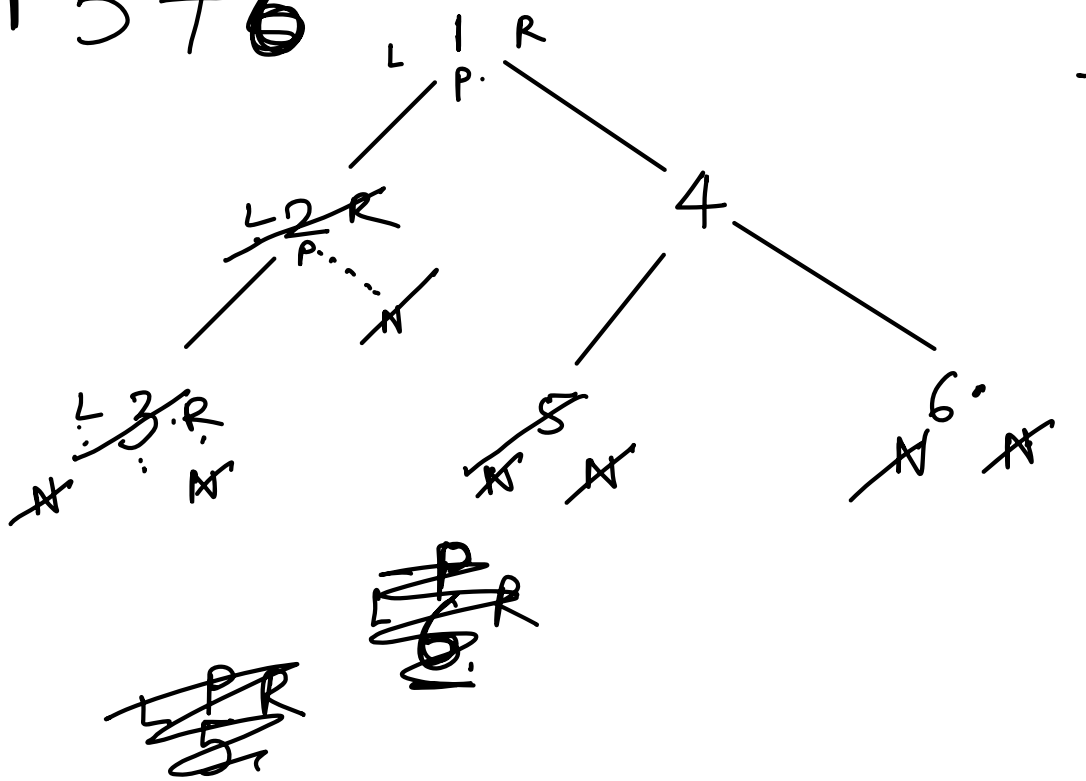
In-order Traversal - Left Root Right

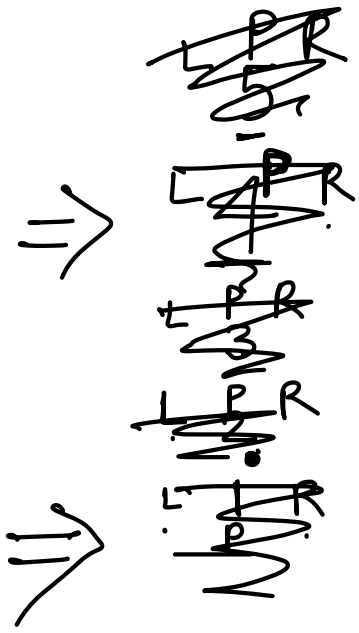
Post-order Traversal - Left Right Root

Recursive traversal



321546



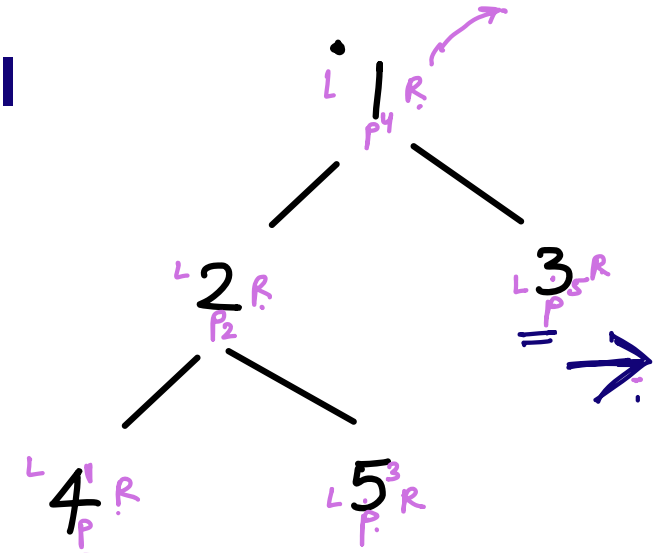


321546

Iterative In-order Traversal

Recursion Stack

our own Stack



4 2 5 1 3

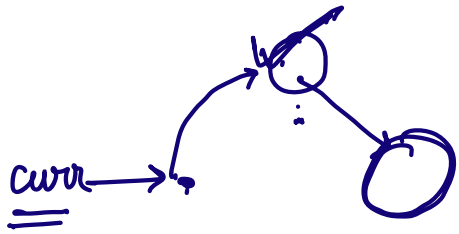
4 2 5 1 3

if (p == null)
pop
print
pop.right

```
def inorder_iterative(root):
    curr = root
    st = []

    while curr != None or len(st) > 0:

        if curr is not None:
            st.append(curr)
            curr = curr.left

        else:
            curr = st.pop()
            print(curr.data)
            curr = curr.right
```

TC: O(N)

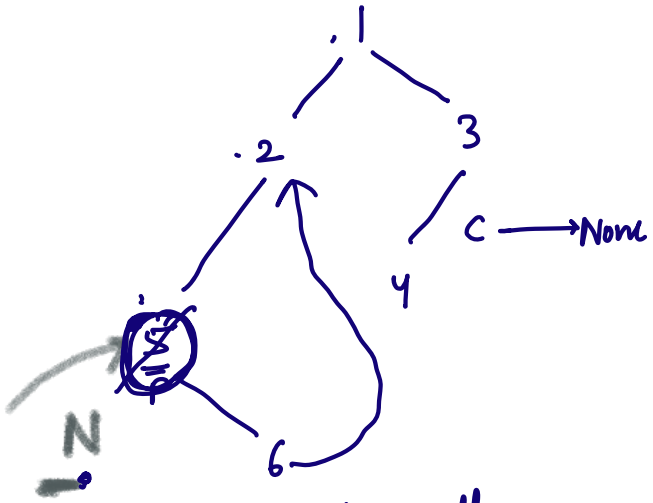
SC: O(N)



Pseudocode:

```
while ( st.isEmpty() == false || curr != null) {
    if ( curr != null) {
        st.push(curr)
        curr = curr.left
    } else {
        curr = st.pop()
        print (curr.data)
        curr = curr.right
    }
}
```

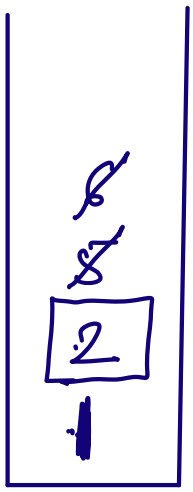
```
def inorder_iterative (root):
    curr = root
    st = []
    while curr != None or len(st) > 0:
        if curr is not None:
            st.append(curr)
            curr = curr.left
        else:
            curr = st.pop()
            print (curr.data)
            curr = curr.right
```



L P R

Print

5 6 2 1 4 3

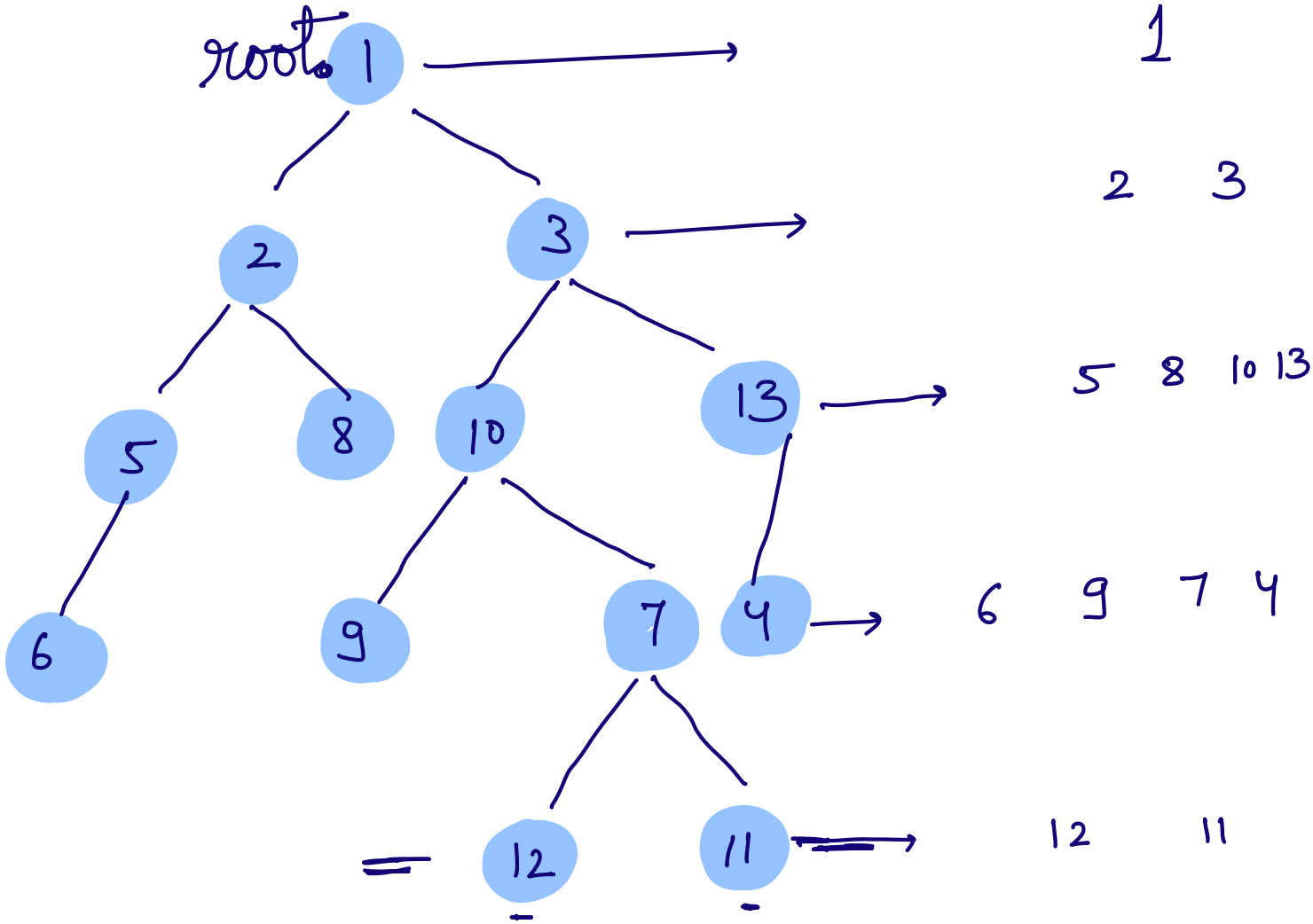


L

C != null
push
left

C - null
pop
print
right

Level Order traversal



while q.isEmpty() == false

R

P ✓

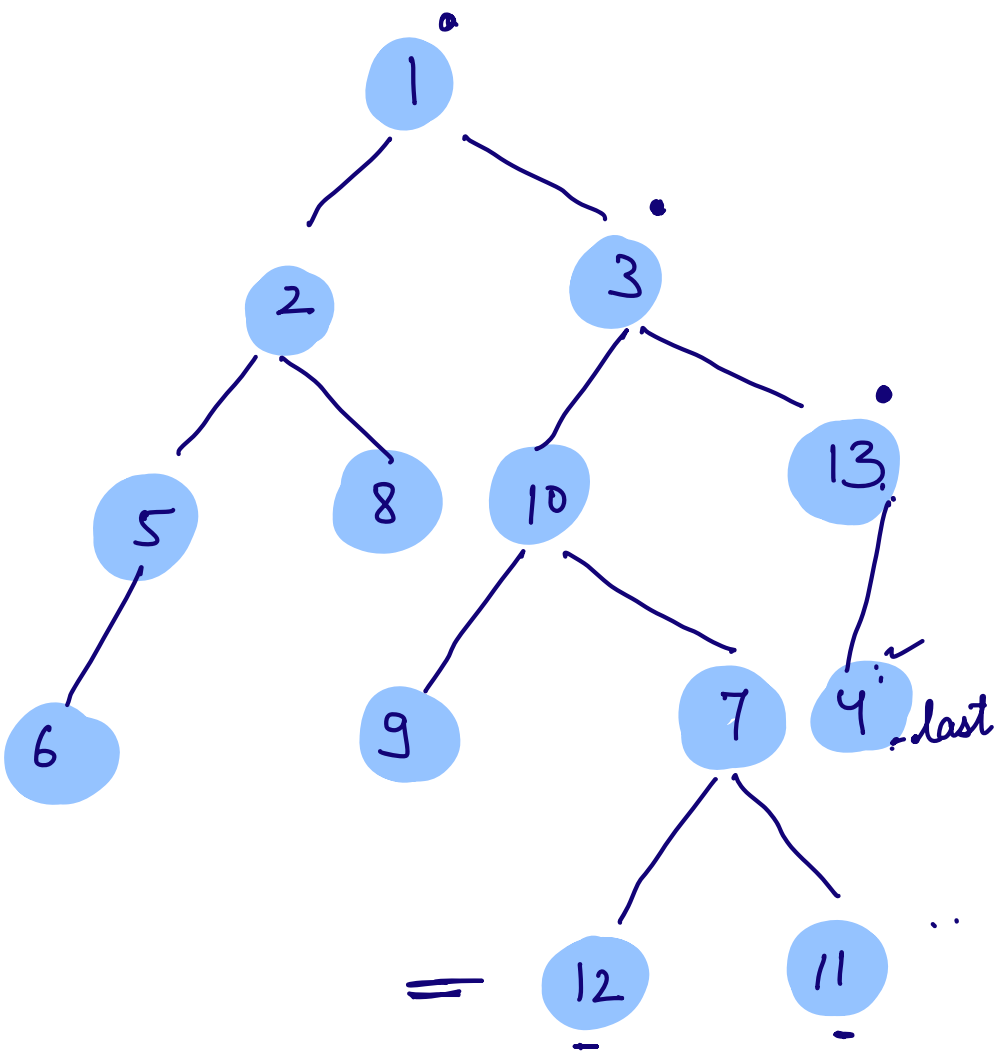
A L

A R

```
q.enqueue(root)
while (not q.empty())
    x = q.dequeue()
    print (x.data)
    if x.left != none :    q.enqueue(x.left)
    if x.right != none :   q.enqueue(x.right)
```

```
q.enqueue(root)                                   last

while ( not q.empty())
|       x = q.dequeue()
|       print (x.data)
|       if x.left != none :      q.enqueue(x.left)
|       if x.right != none :     q.enqueue(x.right)
|
```

last = ①



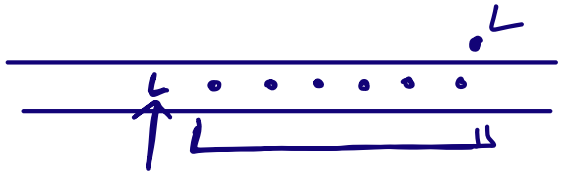
Case 1 → if last node → (R) & L

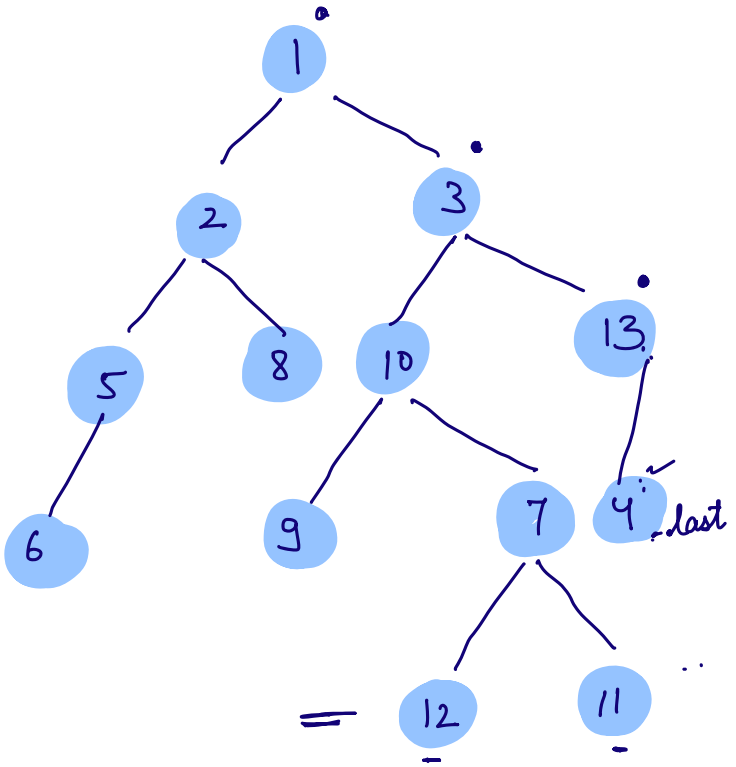
Last node of next level = R

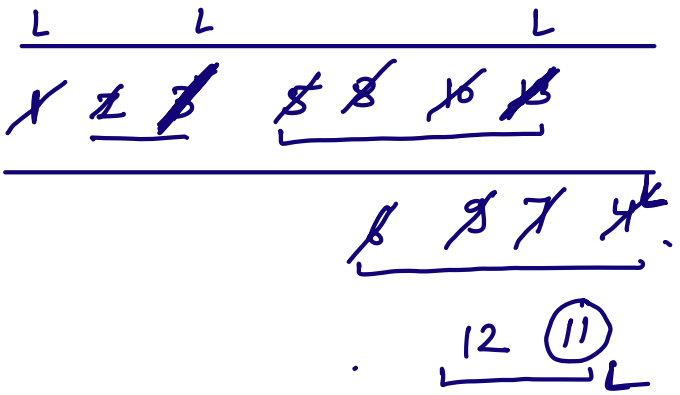
Case 2 → Last Node → L

Last of next level = L

Case 3 → Last Node → None





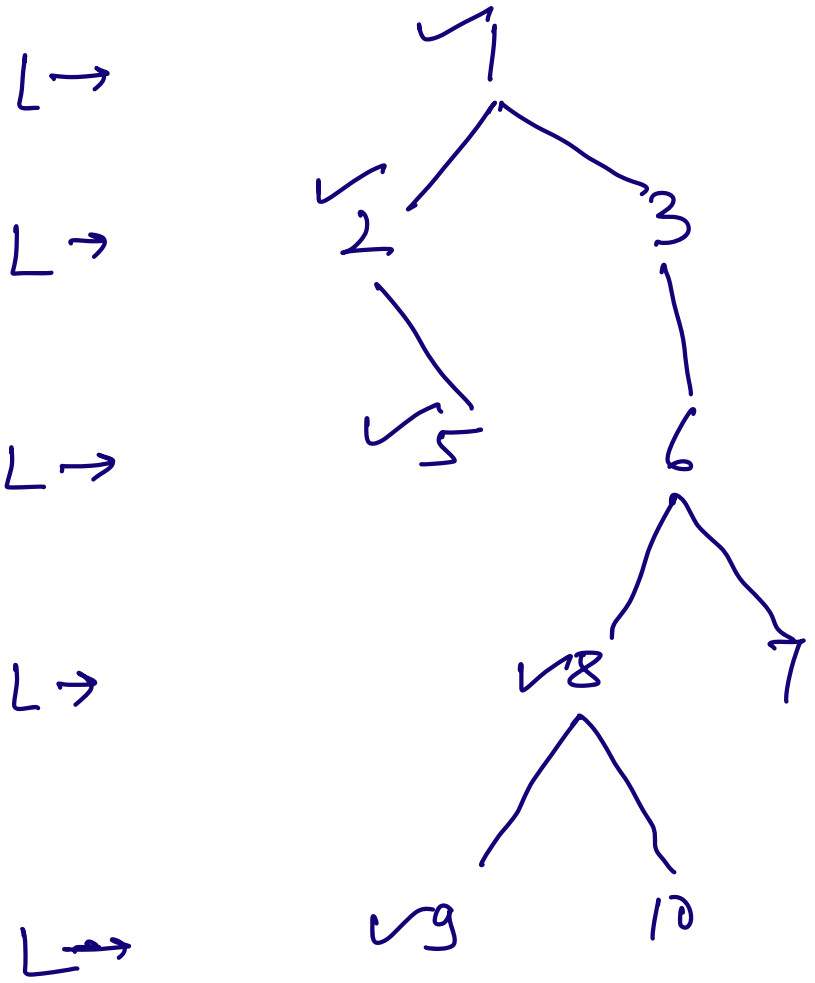


when L is removed, next level is completely inside queue

```
q.enqueue(root)
last = root
while ( not q.empty())
    x = q.dequeue()
    print (x.data)
    if x.left != none :   q.enqueue(x.left)
    if x.right != none :  q.enqueue(x.right)
    if (x == last) { print ("\n")
                     last = q.rear() }
```

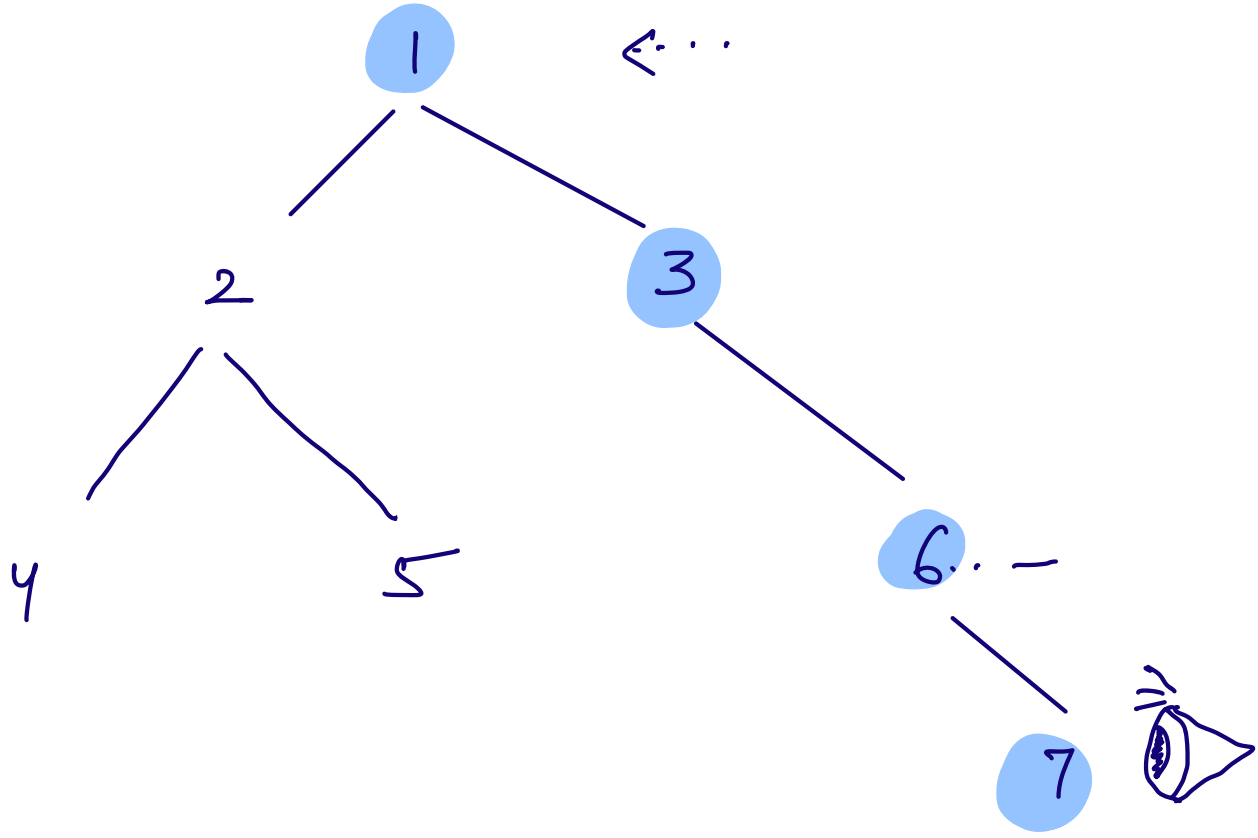
2 more questions ⟶

Left view of tree



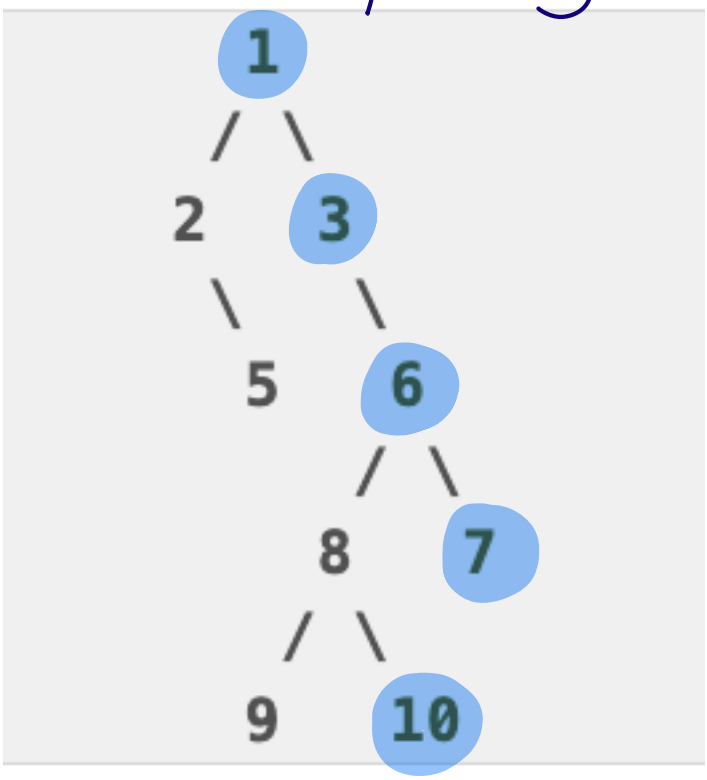
first node of every level

Right view of tree



1 3 6 7

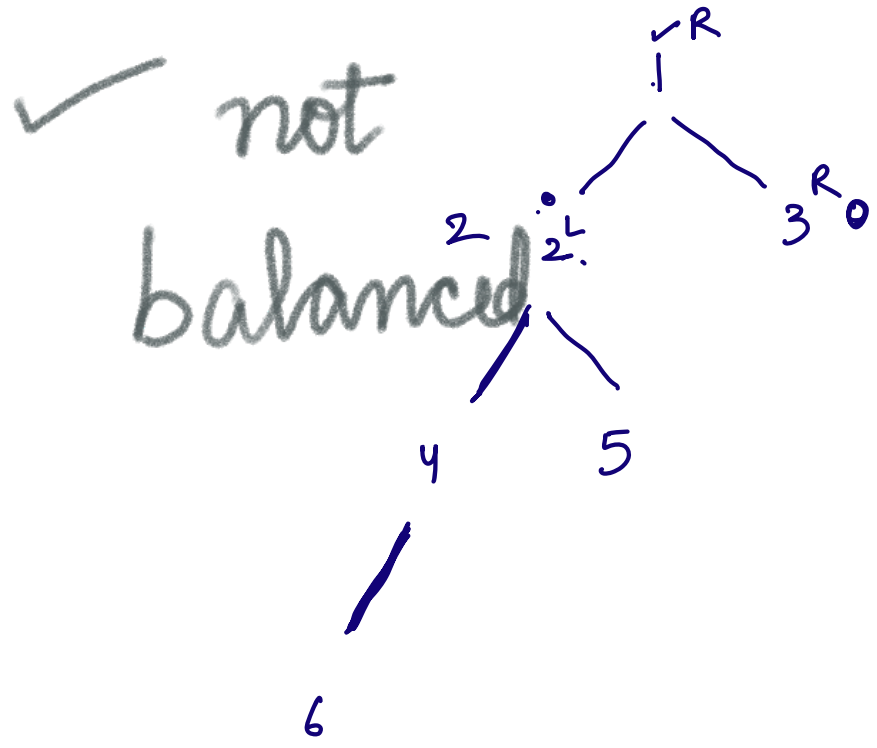
Q.



Right view

= Last node of every level

Q. Check if tree is height balanced



balanced

$|h_L - h_R| <= 1$

> 1

not balanced

- h_l
- h_r

work