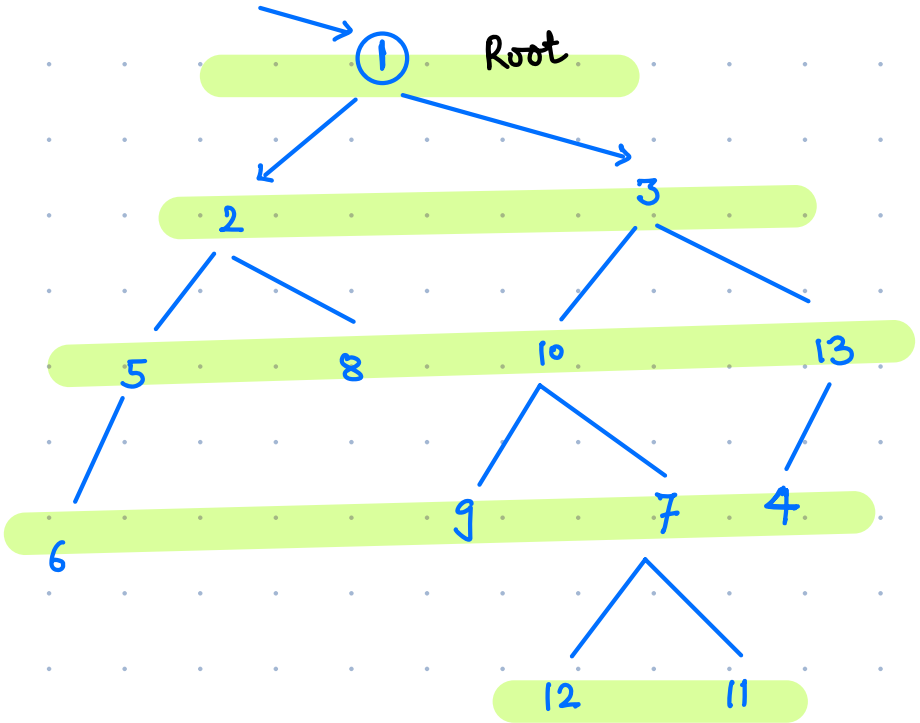
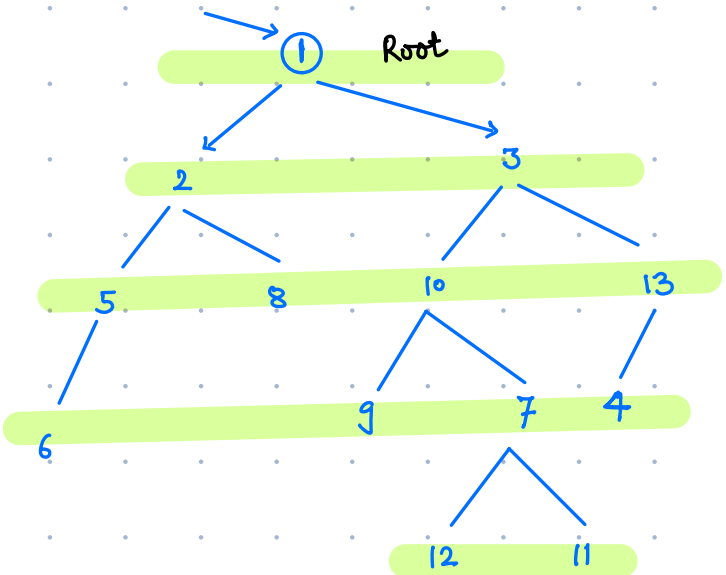
Q.

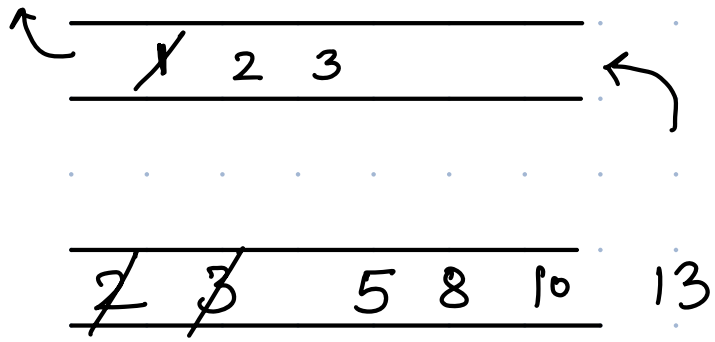


1
2 3
5 8 10 13
6 9 7 4
12 11



1. Process all nodes of current level before moving to next level

2. DS Used = Queue



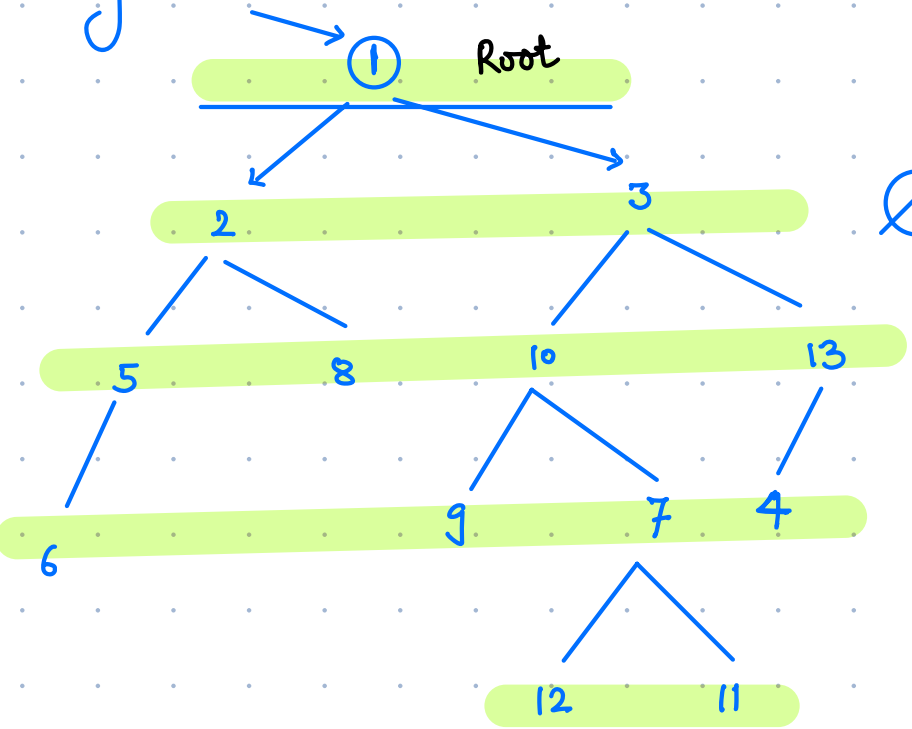
Pseudocode:

```
q.enqueue(root)
while(q.isEmpty() == false) {
        x = q.dequeue();
        print(x.data);
        if(x.left != NULL){q.enqueue(x.left)}
        if(x.right != NULL){q.enqueue(x.right)}
}
```

TC: $2N = O(N)$

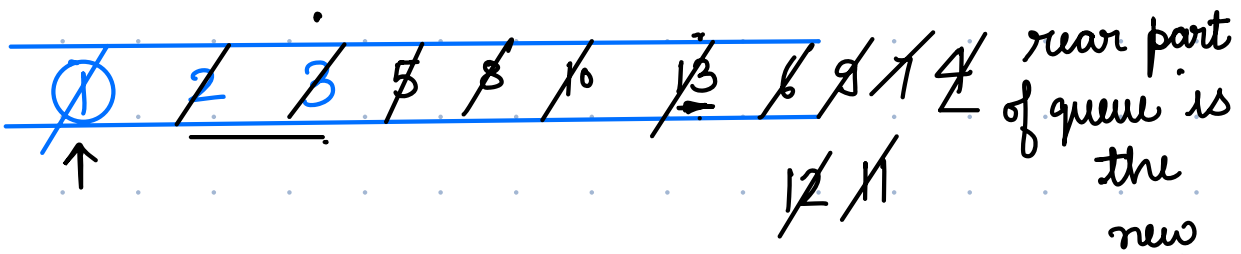
SC: $O(N)$

Dry run :



1 2 3 5 8 10 13 6 9 7 4 12 11

11 4 13 3 1 = last



1

✓ When you remove last node, next level is completely ready

✓ Whenever you put/update last node, add a new line!

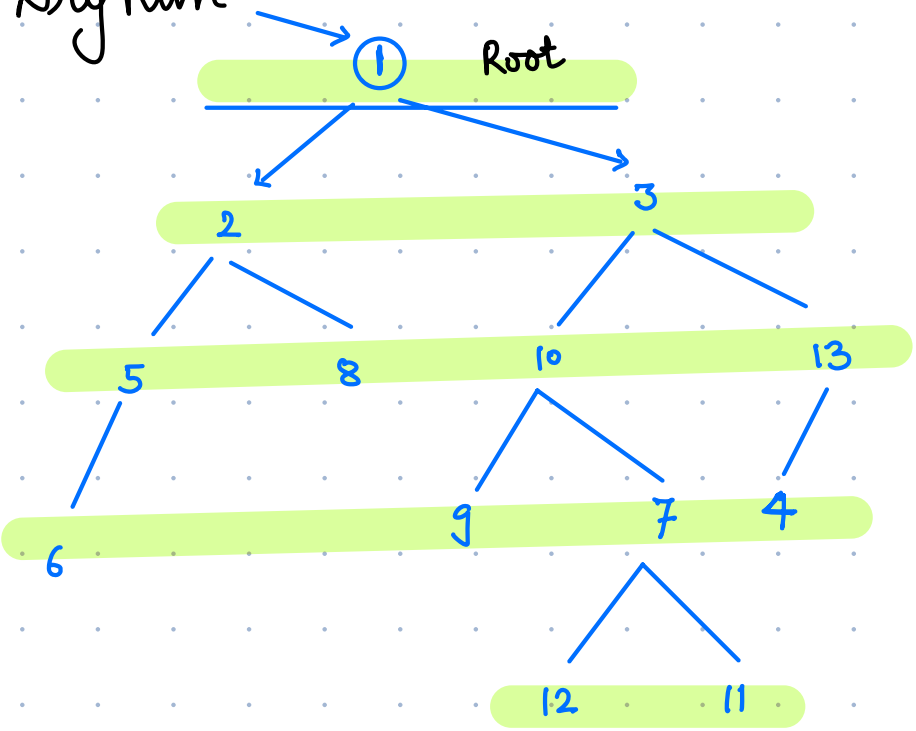
Pseudocode:

```
last = root
q.enqueue(root)
while (q.isEmpty() == false) {
        x = q.dequeue();
        print(x.data);
        if (x.left != NULL) { q.enqueue(x.left) }
        if (x.right != NULL) { q.enqueue(x.right) }

        if (x == last && q.isEmpty() == false) {
                print("\n")
                last = q.rear();
        }
}
```

- TC: 2N = O(N)
- SC: O(N)

Dry Run :



last = 1 2 3 13 4 11

1 2 3 5 8 10 13 6

9 7 4

12 11

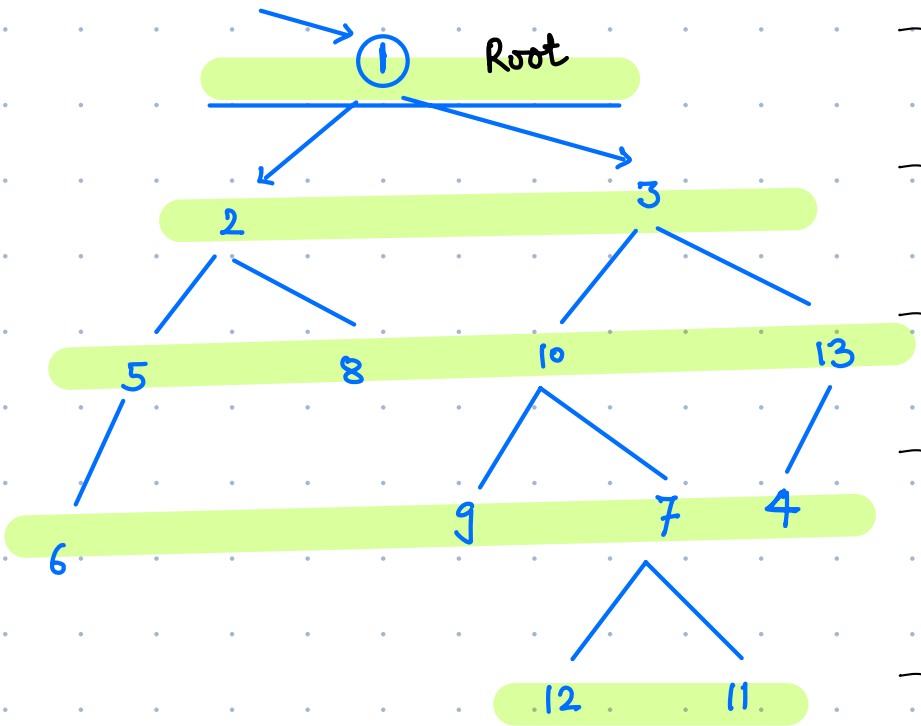
1 ↵

2 3 ↵

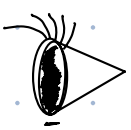
5 8 10 13 ↵

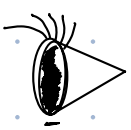
6 9 7 4 ↵

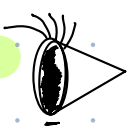
12 11

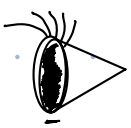


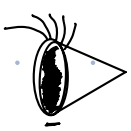
Right view

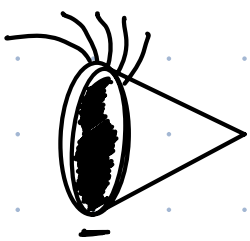
 = 1

 = 3

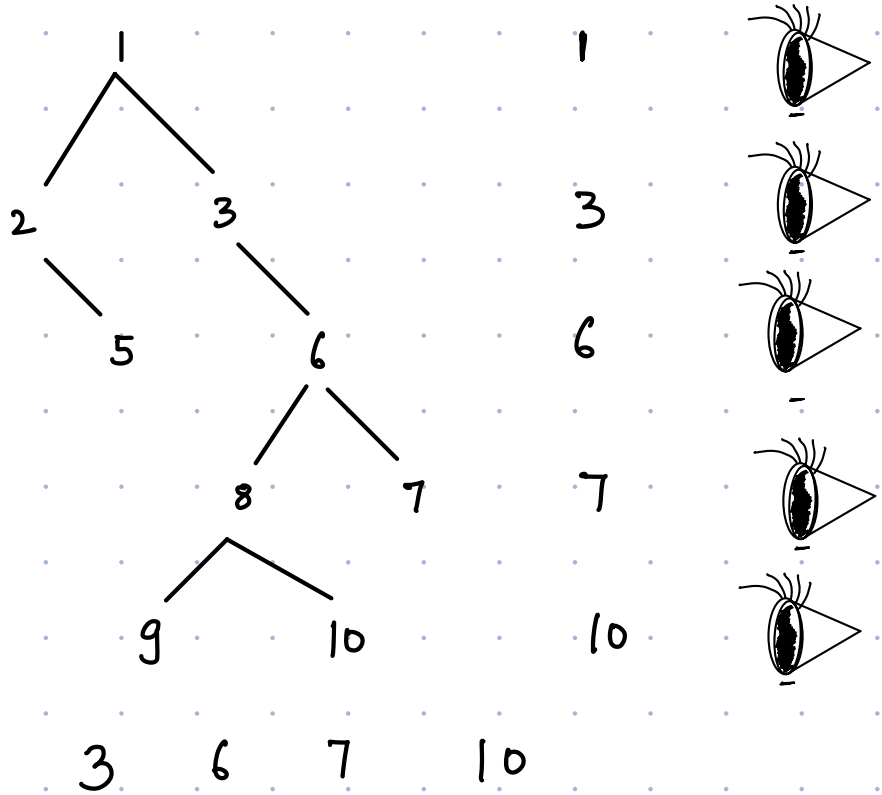
 = 13

 = 4

 = 11



Last Node



Pseudocode:

```
last = root
q.enqueue(root)
while (q.isEmpty() == false) {
        x = q.dequeue();

if (x.left != NULL) { q.enqueue(x.left) }
if (x.right != NULL) { q.enqueue(x.right) }
```

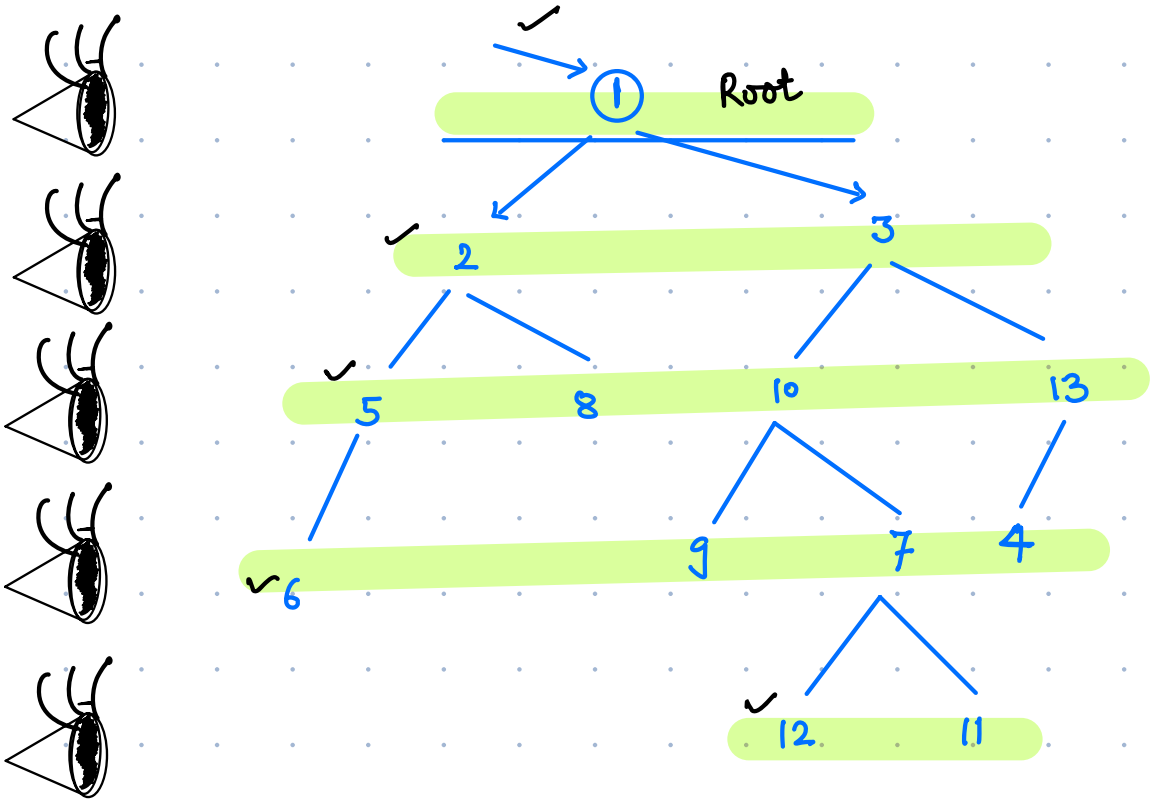
```
if ( x == last && q.isEmpty() == false){
        print (x.data);
        print ("\n")
        last = q.rear();
    }
}
```

TC: $2N = O(N)$

SC: $O(N)$

Left view

last → next node → first node

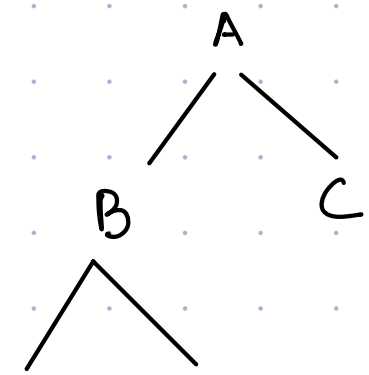


1 2 5 6 12

Types of binary tree —

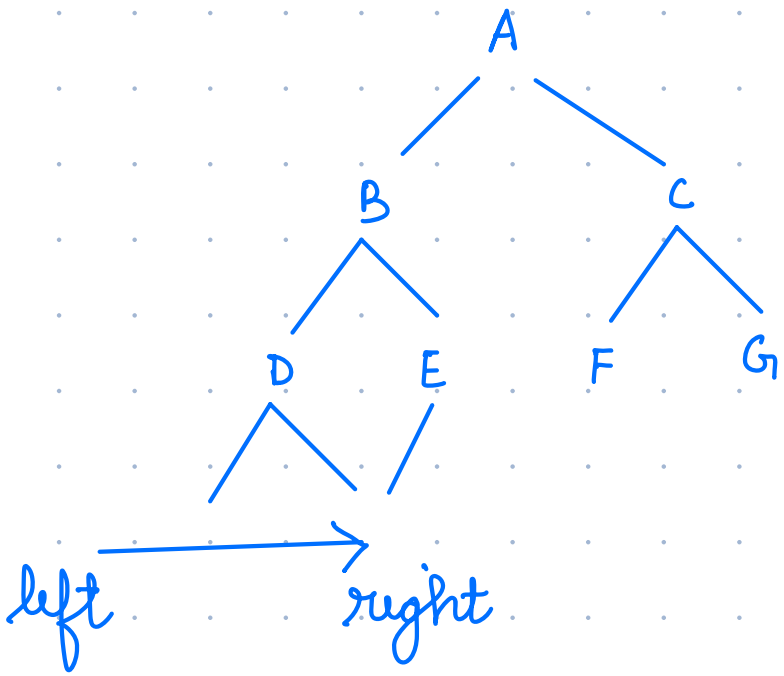
Proper binary tree (Strict BT)

0 or 2 children (never has 1 child)



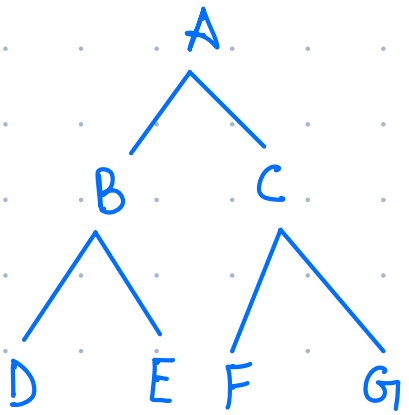
Complete Binary tree

All level of tree completely filled, except last level.



Perfect binary tree

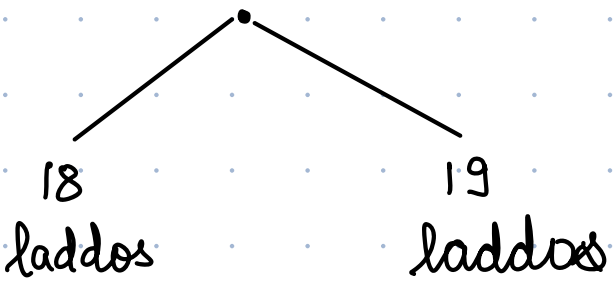
- All internal nodes have exactly 2 children
- all the leaf nodes are at same level



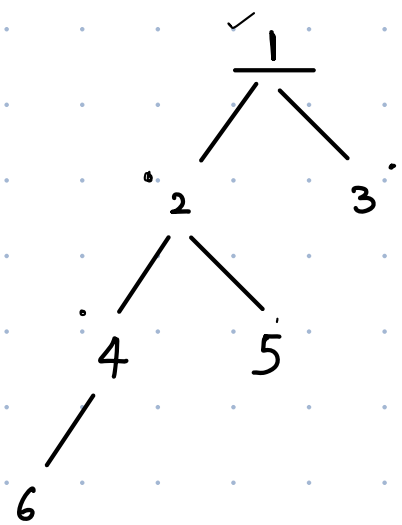
Q. Check height balanced tree

For all nodes if

$$\left| h_l - h_r \right| = 0 \text{ or } 1$$



18 laddos	19 laddos
19	18
18	18
→ 17	19 ✓



L0 last = 1

L1 last = 3

L2 last = 5

L3 last = 6

0 1 2 3

Number of edges = 3

Height of tree = Height of root node

```
function calcHeight ( root ) {
    if (root is leaf ) { return 0; }
    ✓ hl = calcHeight (root.left);
    ✓ hr = calcHeight (root.right)
    return Max( hl, hr) + 1 ← Post Order
}
```

```
function heightbalanced ( root ) {
    if ( root == null) {
        return true;
    }
    hl = calcHeight (root.left)
    hr = calcHeight (root.right)
    if (abs( hl - hr) > 1 ) {
        return false
    }

    return heightbalanced ( root.left )
        && heightbalanced (root.right)
}
```

At any node ——

$$|h_l - h_r| > 1$$

global variable ⟶ false

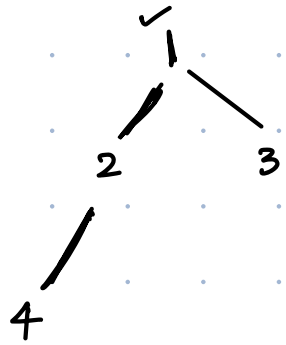
```
boolean ishb = true

function hight ( root, ——) {
    if (root is leaf ) { return 0; }
    if (root == null) { return -1 }
    l = hight (root.left)
    r = hight ( root.right)
    if ( abs (l - r) > 1)  ishb=false
    return max (l,r) + 1
}
```

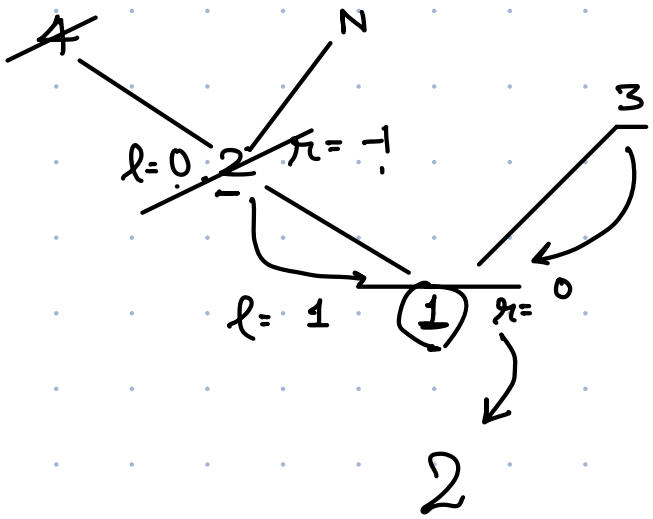
Dry Run ⟶

boolean ishb = true

```
function hight ( root, ___ ) {
    if (root is leaf ) { return 0; }
    if (root == null) { return -1 }
    l  = hight (root.left)
    r  = height ( root.right)
  • if ( abs (l - r) > 1) ishb=false
    return max (l,r) + 1
}
```

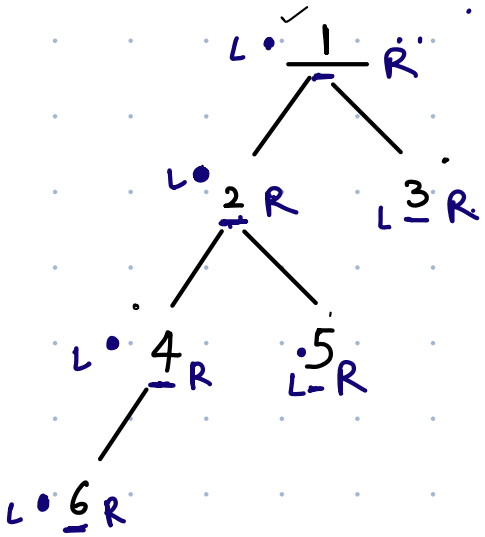


ishb = true



TC: O(N)

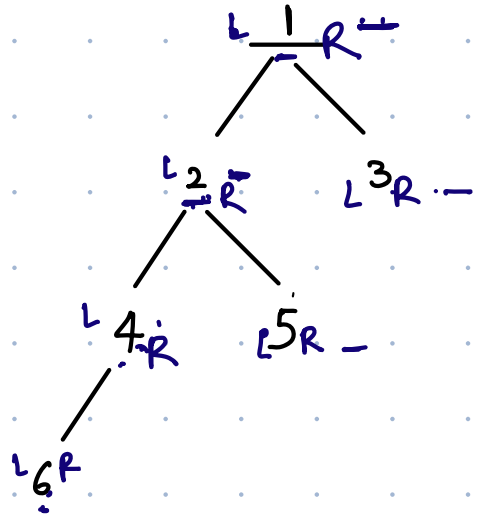
Q Consider inorder & post order — you have to construct binary tree



6 4 2 5 1 3

Inorder
Go to left
Print
Go to right

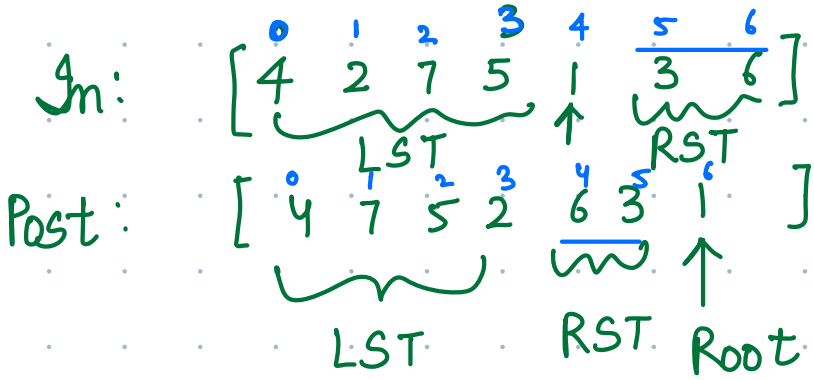
PostOrder
Go to left
Go to right
Print

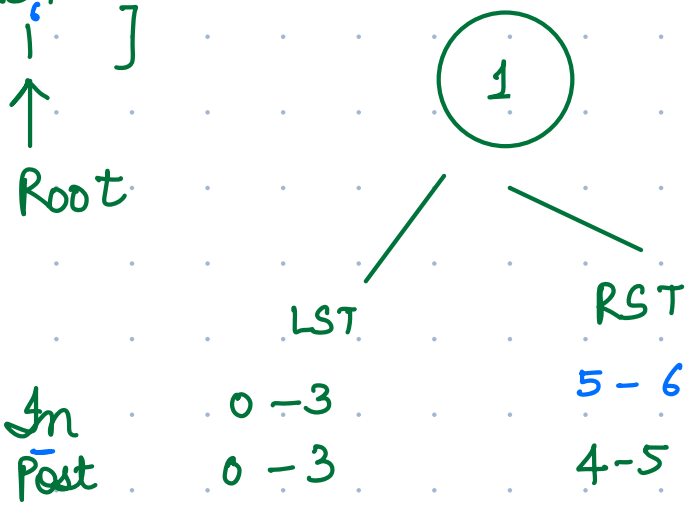


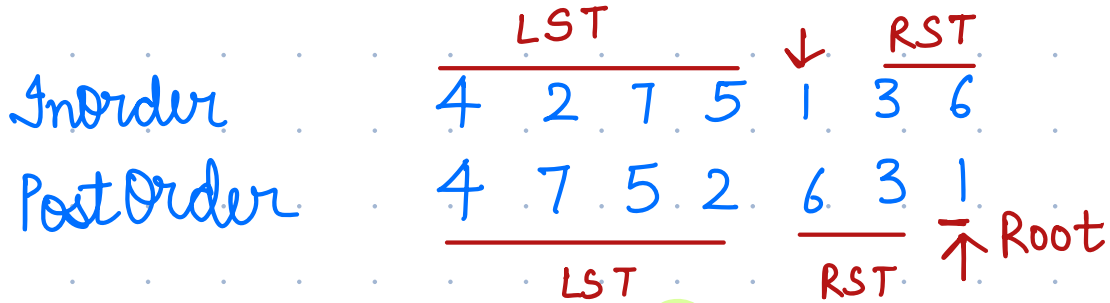
6 4 5 2 3 1

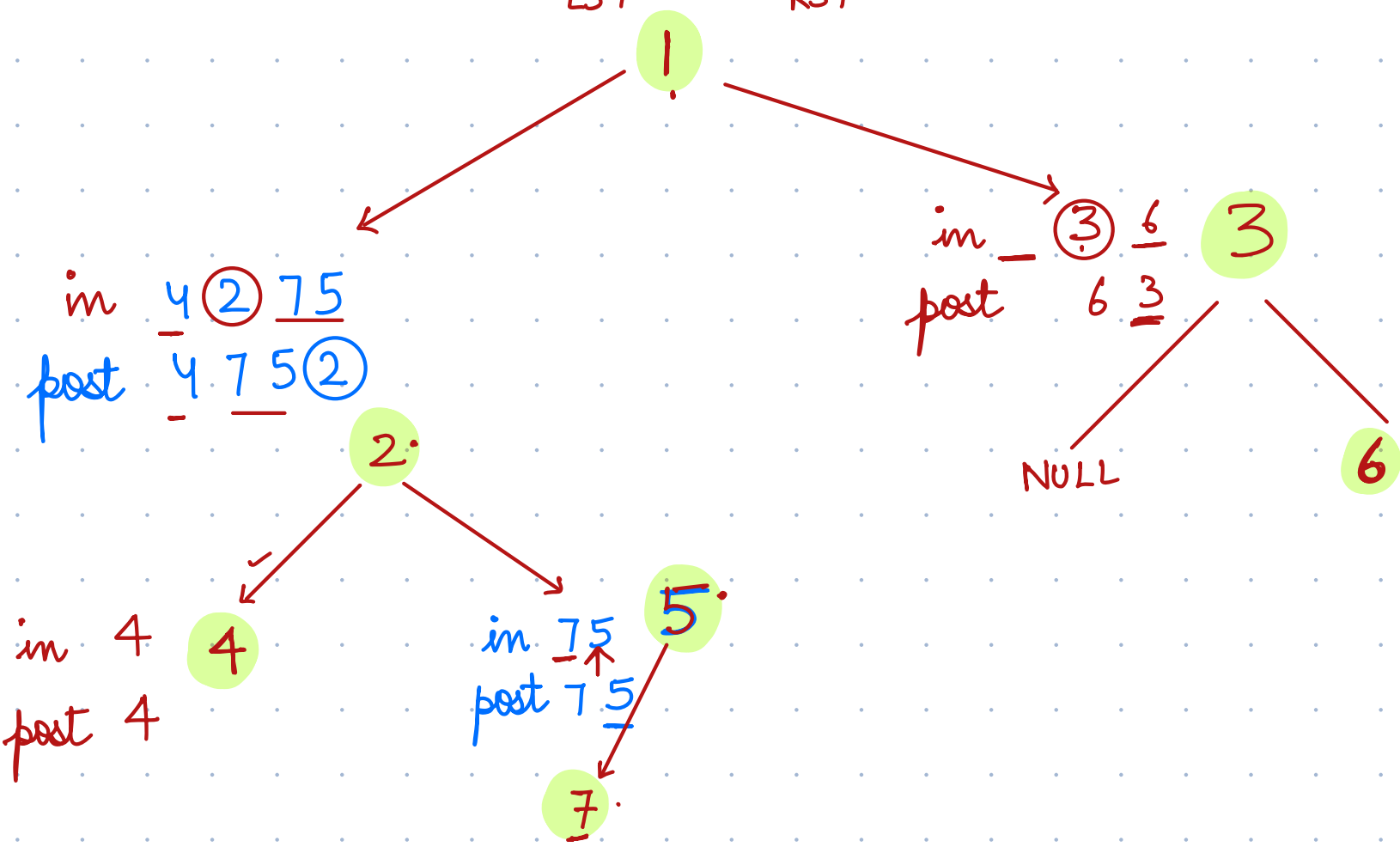
Post Order

Idea:


In:
0 1 2 3 4 5 6
4 2 7 5 1 3 6
LST
RST
Post:
0 1 2 3 4 5 6
4 7 5 2 6 3 1
LST
RST
Root


1
LST
RST
In
Post
0 -3
0 -3
5 - 6
4-5


LST
RST
Inorder
4 2 7 5 1 3 6
PostOrder
4 7 5 2 6 3 1
Root
LST
RST


1
in 4 2 7 5
post 4 7 5 2
in 3 6
post 6 3
3
NULL
6
2
in 4
post 4
4
in 7 5
post 7 5
5
7

Pseudocode :

```
function Node buildtree (inorder, postOrder) {
  if length of postOrder {return null }
    rootValue = postOrder.last
    root = new Node (rootValue)

    // find inder of rootValue in inOrder
    // Split it into left ST & RST
    rootIndex = indexOF (inOrder, rootValue);

root.left = build tree (subarray (inorder,
                                  start, rootIndex-1),
                        subarray (postOrder, start,
                                  rootIndex - 1))
```

```
root.right = buildTree(subarray(inorder,
                                rootIndex+1, end),
                       subarray(postOrder,
                                rootIndex,
                                end - 1))

    return root

}
```