Agenda

1. Level Order Traversal
2. Right View
3. Vertical Level Traversal
4. Top View
5. Types of Binary Tree
6. Height and Balanced Binary Tree

① Contest

22 Dec 9 - 10:30 PM

Comparator, Linked List, Stacks and Queue

② Wed class (20 Dec)

↓

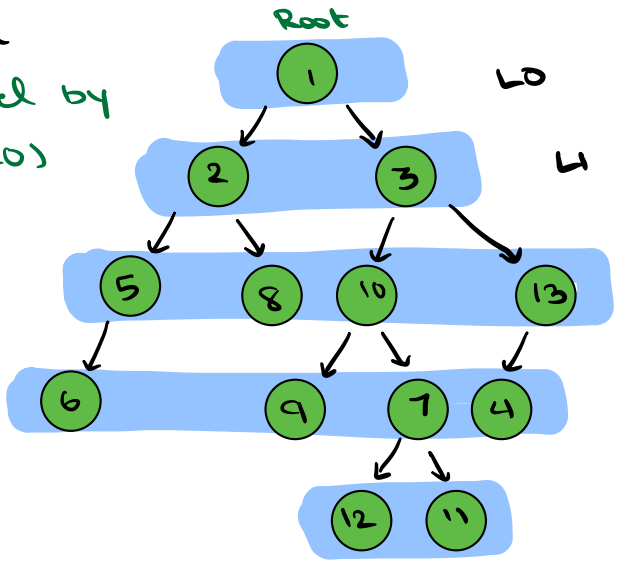
Tue (19 Dec) Trees 3

↓

BST

Level Order Traversal

(Print all nodes level by level starting from L0)



O/P

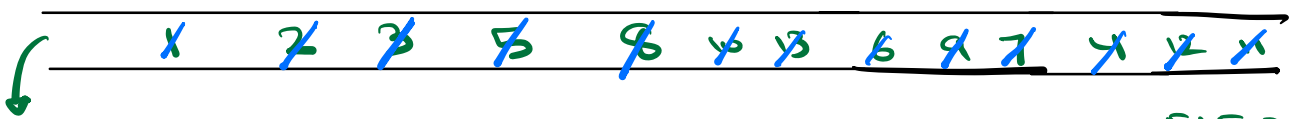
1

2 3

5 8 10 13

6 9 7 4

12 11



O/P → 1 2 3 5 8 10 13 6 9 7 4

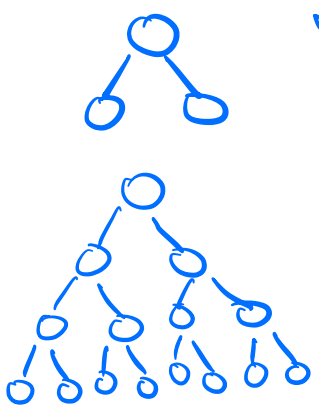
12 11

1. Queue → either nodes of cur level

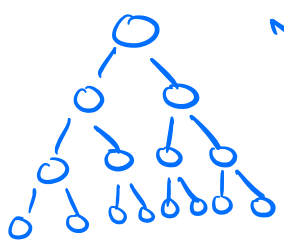
or

some nodes of cur level

and some nodes of next level



N	Last level
3	2
7	4
15	8



N

Last level → N/2

Queue <Node> q

q.enqueue(root)

while (!q.empty()) {

Node cur = q.front()

q.dequeue()

print(cur.data)

if (cur.left != NULL)

q.enqueue(cur.left)

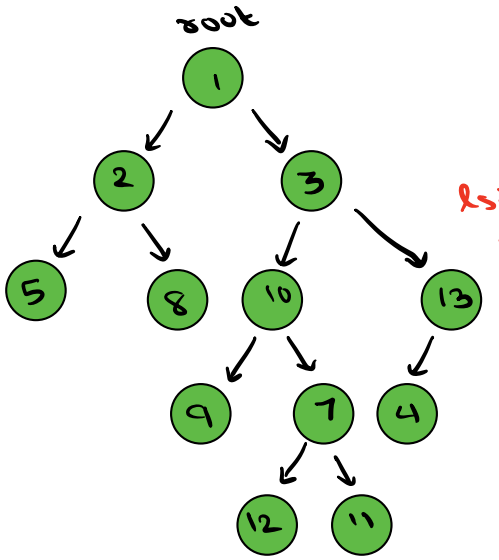
if (cur.right != NULL)

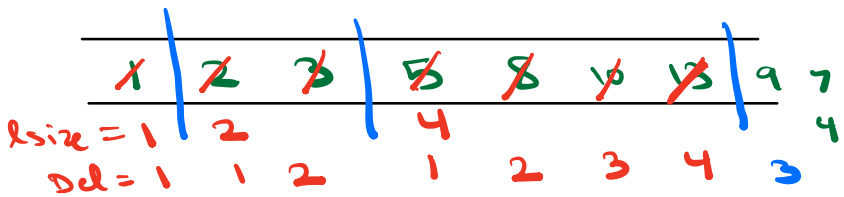
q.enqueue(cur.right)

}

TC: O(N)

SC: O(N)





levelsize :

↓

keep track of nodes in cur level

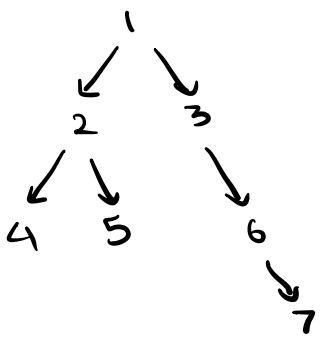
1 \n

2 3 \n

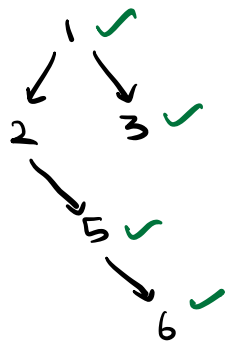
5 8 10 13 \n

```
Queue <Node> q
q.enqueue(root)
while (!q.empty()) {
    int levelsize = q.size()
    for (cnt = 1 ; cnt ≤ levelsize ; cnt++) {
        Node cur = q.front()
        q.dequeue()
        print (cur.data)
        if (cur.left != NULL)
            q.enqueue (cur.left)

        if (cur.right != NULL)
            q.enqueue (cur.right)
    }
    print ("\n")
}
```

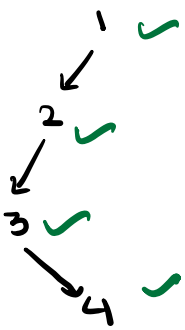
Right View of Binary Tree



1 3 6 7



1 3 5 6

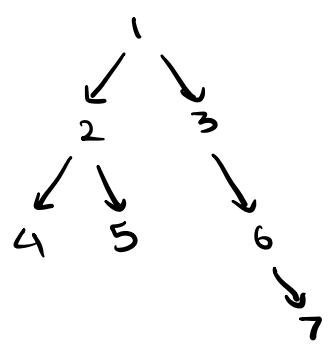


1 2 3 4

Print last node of every level

```
Queue <Node> q
q.enqueue(root)
while (!q.empty()) {
    int levelsize = q.size()
    for (cnt = 1; cnt <= levelsize; cnt++) {
        Node cur = q.front()
        q.dequeue()
        if (cnt == levelsize) {
            print (cur.data)
        }
        if (cur.left != NULL)
            q.enqueue(cur.left)
        if (cur.right != NULL)
            q.enqueue(cur.right)
    }
}
```

Left view (First node of every level)



1 2 4 7

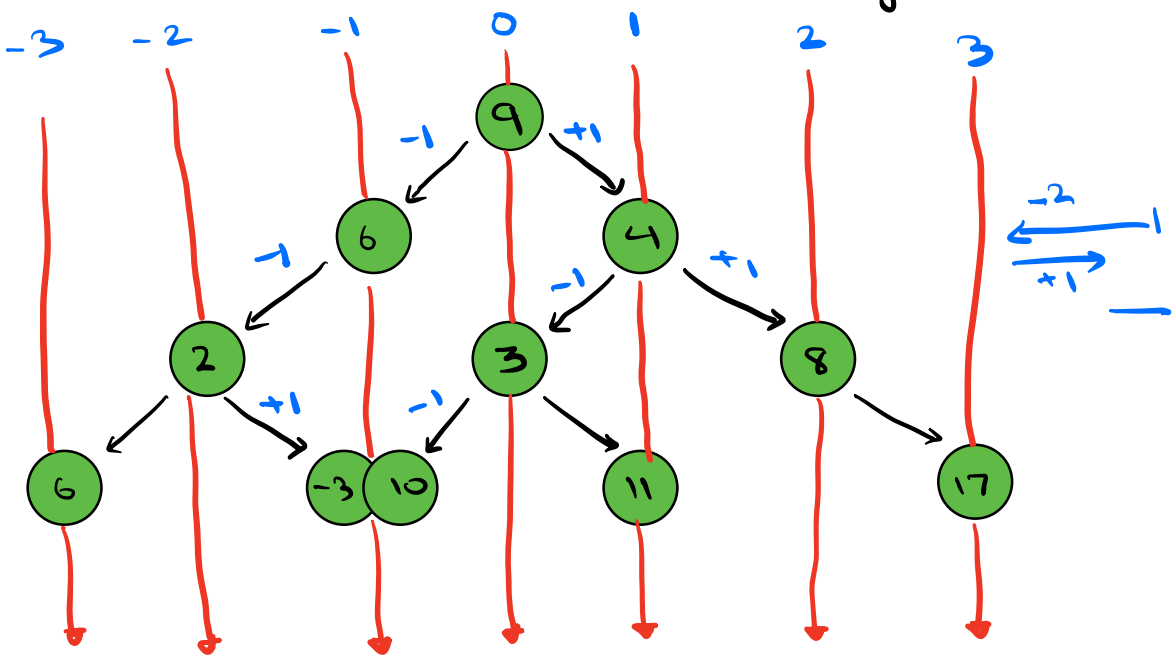
```
if (cnt == 1) {
    print (cur.data)
}
```

Will last level node always be a leaf node

YES

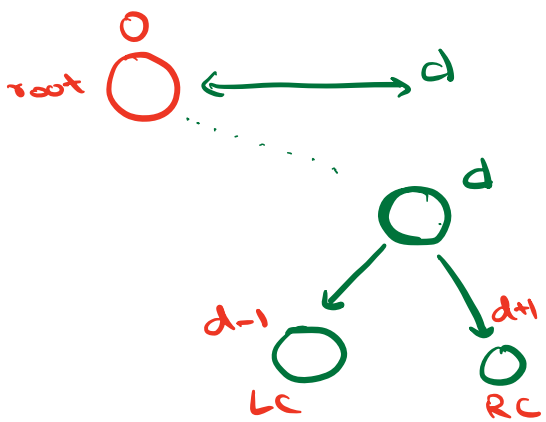
Vertical Order Traversal

Left → -ve
Right → +ve



In a level, top to bottom

-3	6		
-2	2		
-1	6	-3	10
0	9	3	
1	4	11	
2	8		
3	17		

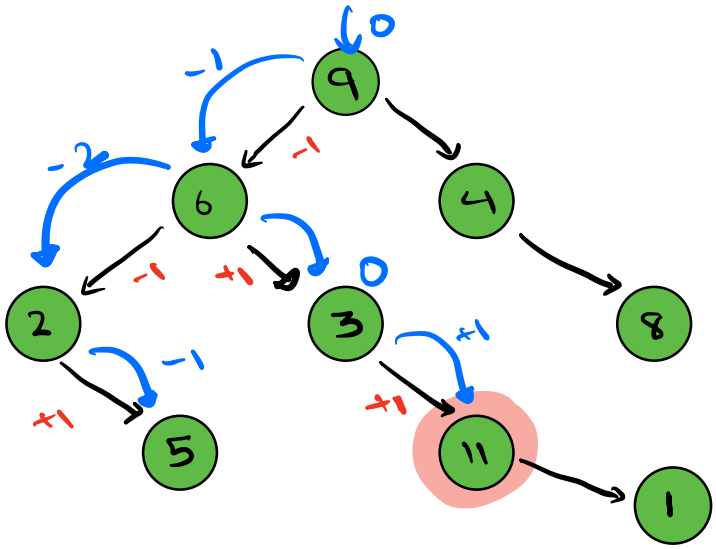


HM < int, list < Node >>

level → nodes

-1 : 6, -3, 10

— : —



Preorder

Recursive or In / Pre / Post

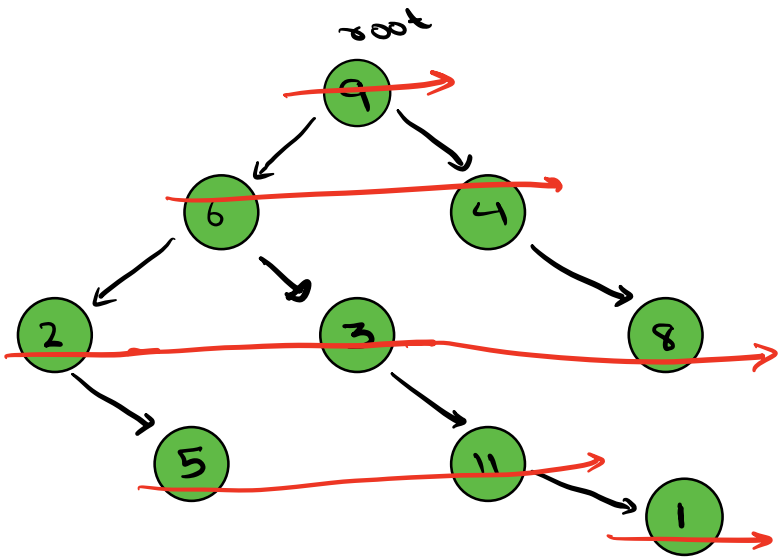
Level order

NLR

HM

0 : 9, 3
-1 : 6, 5
-2 : 2
1 : 11

Left subtree should be traversed first but here 11 is at bottom of 4



0 : 9, 3
-1 : 6, 5
1 : 4, 11
-2 : 2
2 : 8, 1

~~9,0~~ ~~6,-1~~ ~~4,1~~

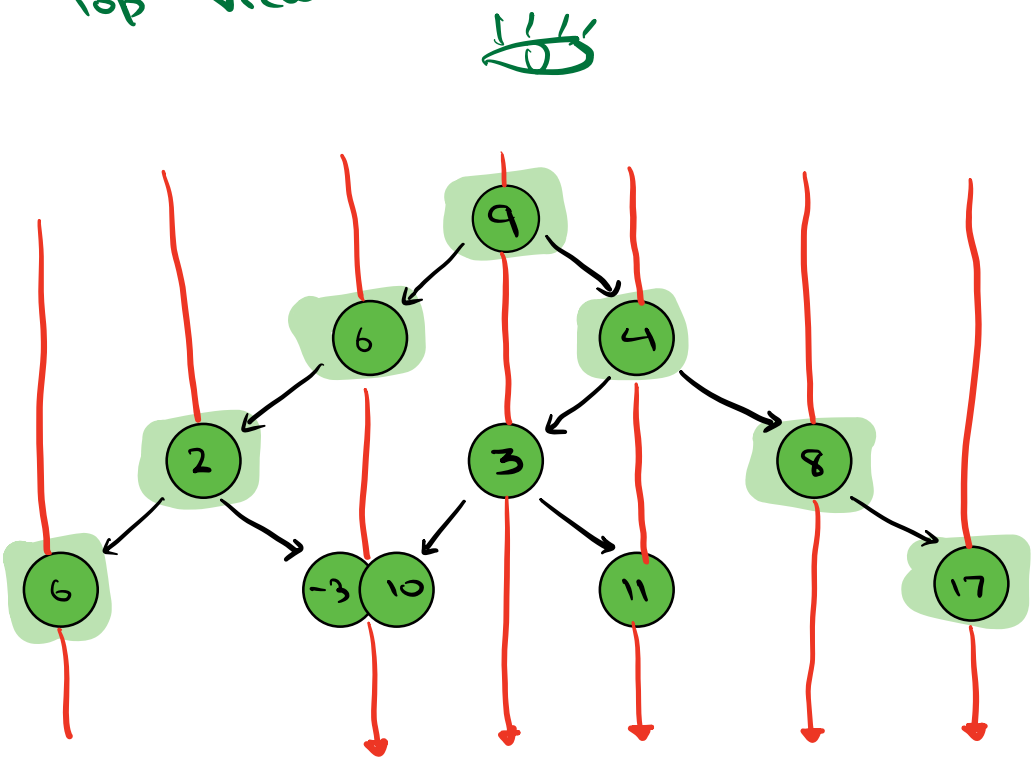
d d+1 d-1

node, level

~~2,-2~~ ~~3,0~~ ~~8,2~~ ~~5,-1~~ ~~11,1~~ ~~1,2~~

```
class Pair {
    Node n
    int vlevel
}

HM <int, list <Node>> hm

queue <Pair> q
int minl=0, maxl=0
Pair r = new Pair (root, 0)
q.enqueue(r)

while (!q.empty()) {                    TC: O(N)
    Pair f = q.front()                  SC: O(N)
    q.dequeue()

    minlvl = min (minlvl, f.vlevel)
    maxlvl = max (maxlvl, f.vlevel)
    hm[f.vlevel].add(f.n.data)
    if (f.n.left != NULL)
        q.enqueue(new Pair(
                  f.n.left, f.vlevel-1))

    if (f.n.right != NULL)
        q.enqueue(new Pair(
                  f.n.right, f.vlevel+1))
}

        min → -2        max → 2

for (lvl = min ; lvl ≤ max ; lvl++) {
    // access hm[lvl] → print it
}
```

Top view



Top view :

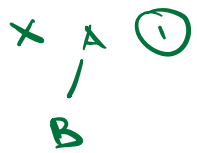
first node of each vertical level

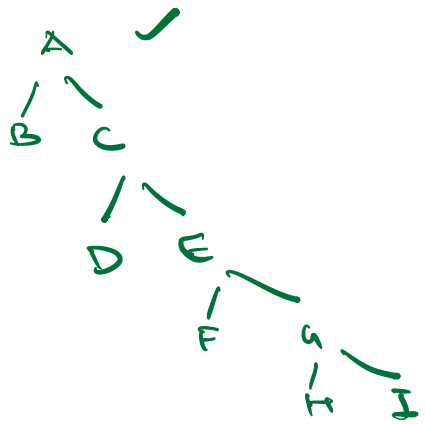
Bottom view :

last node of each vertical level

10:49

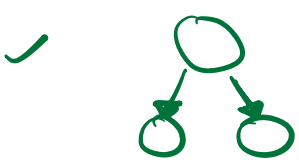
Proper BT → Every node has 0 or 2 children.

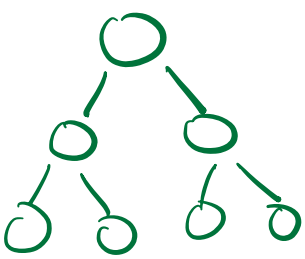


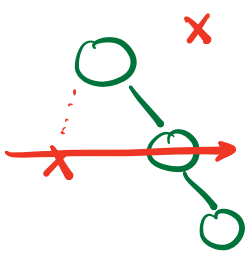


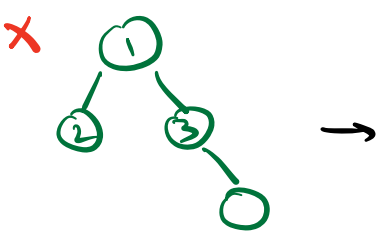
Complete BT : All levels are filled except last level

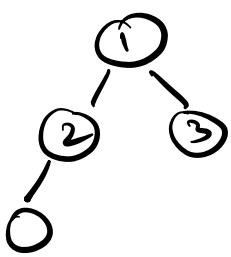
Last level can be partially filled (left to right)

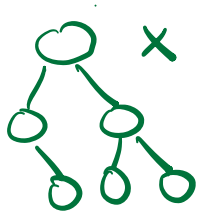




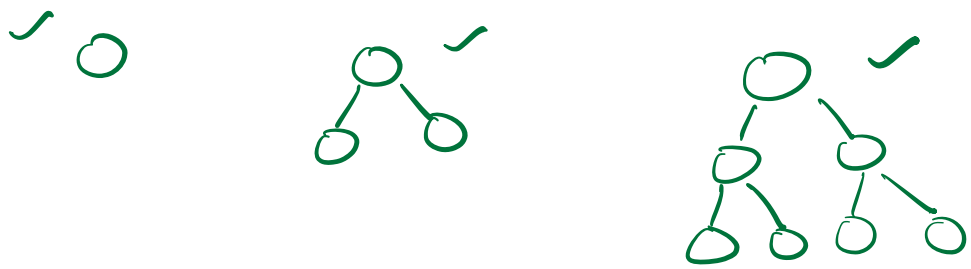








Perfect BT : All levels are completely filled



Leaf nodes → last level

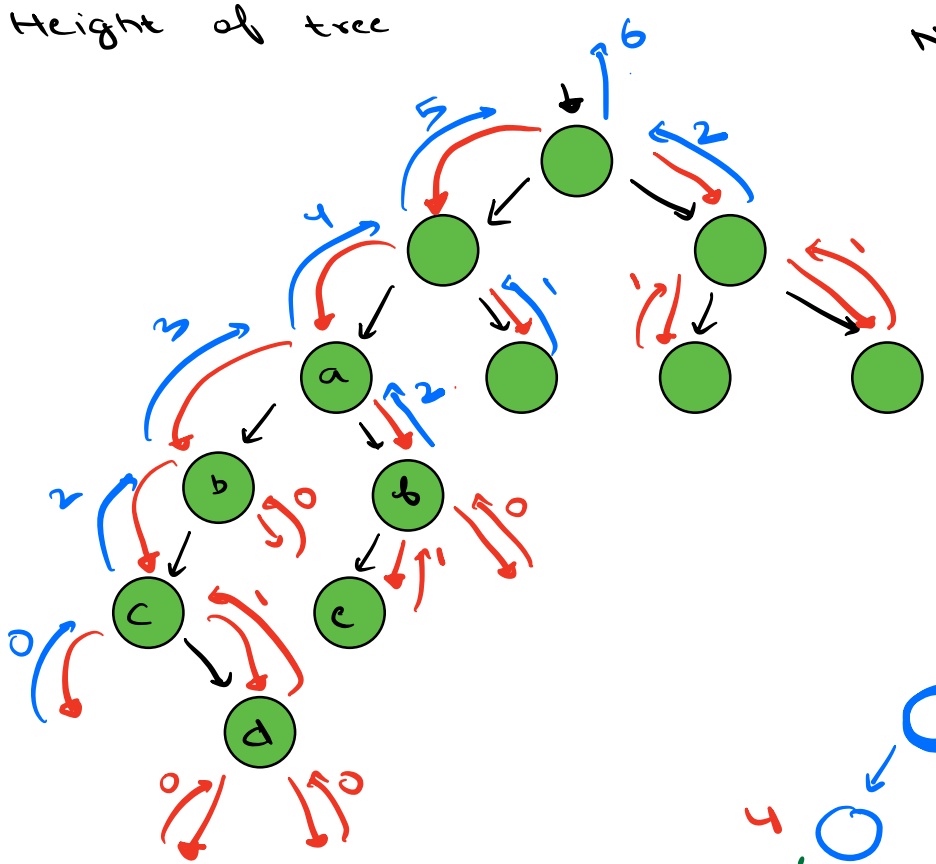
Perfect is complete and proper.

H of node = longest path from node to leaf

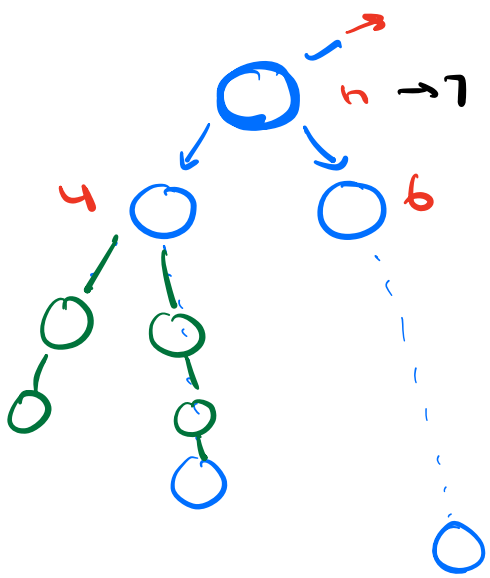
H (no. of nodes)

Height of tree

Nodes = 1 + Edges



LRN



h (Node) =

max (h (LC), h (RC))

+ 1

Post order traversal

```
int height (Node root) {
    if (root == NULL) return 0
    int lh = height (root.left)
    int rh = height (root.right)
    return max (lh, rh) + 1
}
```

TC: O(N)

SC: O(H) → N

Balanced Binary Tree

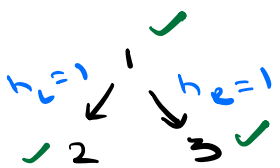
Check if a tree is height balanced

↓

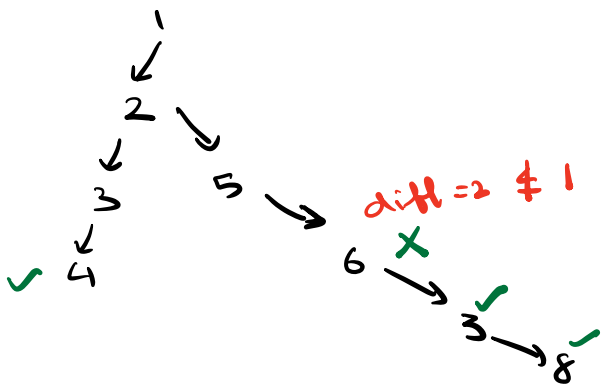
A tree in which for every node

$$|LST_h - RST_h| \leq 1$$

Height (in terms of nodes)



Tree → Balanced



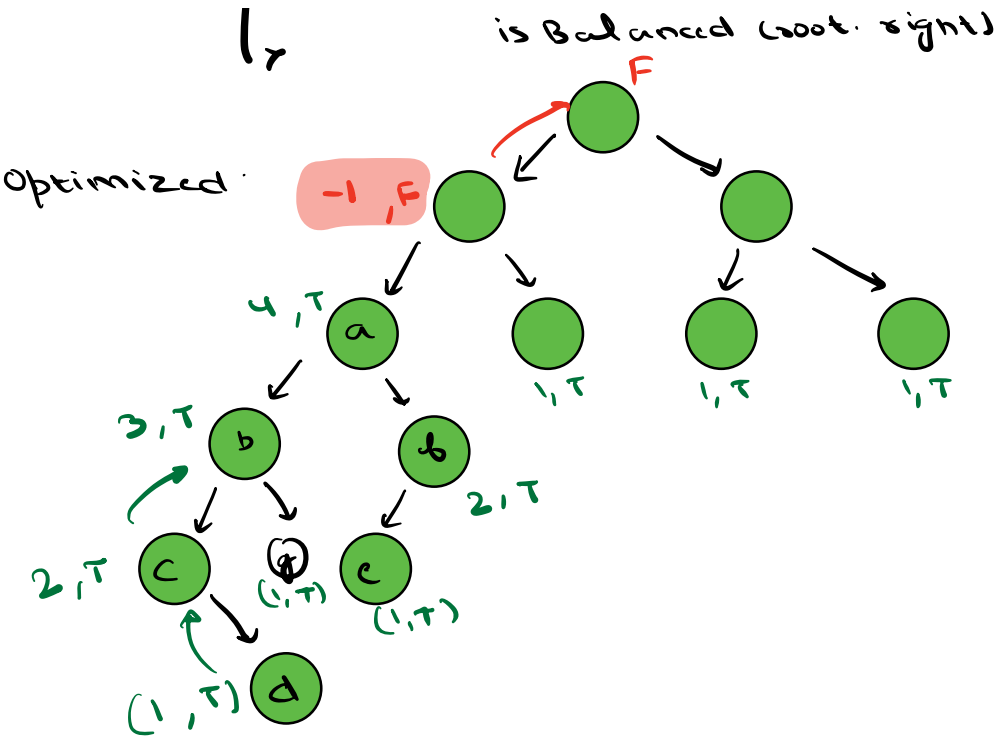
Tree → Not Balanced

BF : Go to every node and check whether its balanced or not

```
bool isBalanced (Node root) {
    if (root == NULL)
        return true

    int lh = height(root.left)
    int rh = height(root.right)
    if (abs(lh - rh) > 1)
        return false

    return isBalanced(root.left) &&
```

TC : $O(N^2)$

SC : $O(H)$




```
class Pair {
    int height,
    bool isBal
};
```

```
bool isBalanced (Node root) {
    return helper(root).isBal
};
```

```
Pair helper(Node root){
    if(root==NULL)
        return new Pair(0, true)

    Pair lp= helper(root.left)
    Pair rp= helper(root.right)

    if(lp.isbal==false || rp.isbal==false)
        return new Pair(-1, false)

    else if(|lp.height - rp.height|>1)
        return new Pair(-1, false)

    else
      return new Pair(
    max(lp.hight,rp.height)+1    , true)
}
```

TC : O(N)

SC : O(H)