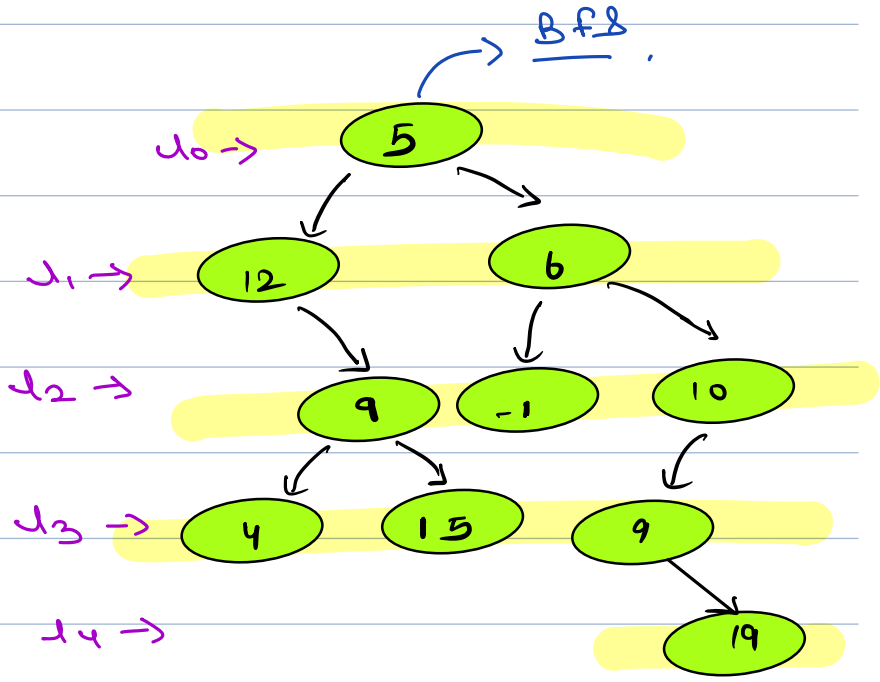
Today's content →

Level order Traversal
Left view & right view
Vertical Level Order
Top view & Bottom view
Types of B.T.
check Height Balanced

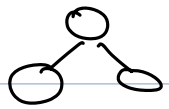
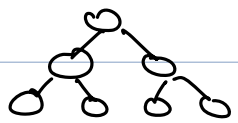
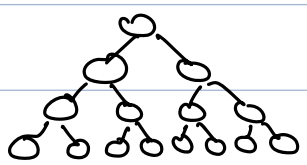
8) Level order Traversal

O/P :- 5 12 6 9 -1 10 4 15 9 19



~~5 12 6 9 -1 10 4 15 9 19~~

5 12 6 9 -1 10 4 15 9 19

	n	last level
	3	2
	7	4
	15	8

queue will have $(\frac{n}{2} + \frac{n}{4})$ nodes at 1 time.

```
void levelOrder(Node root){
    Queue<Node> q;
    q.add(root);
    while(q.size()>0){
        Node front = q.peek();
        q.remove();
        print(front.data);
        if(front.left!=null){
            q.add(front.left);
        }
        if(front.right!=null){
            q.add(front.right)
        }
    }
}
```

T.C → O(n)

S.C → O(n)

* Level Order Traversal - 2 :-

output

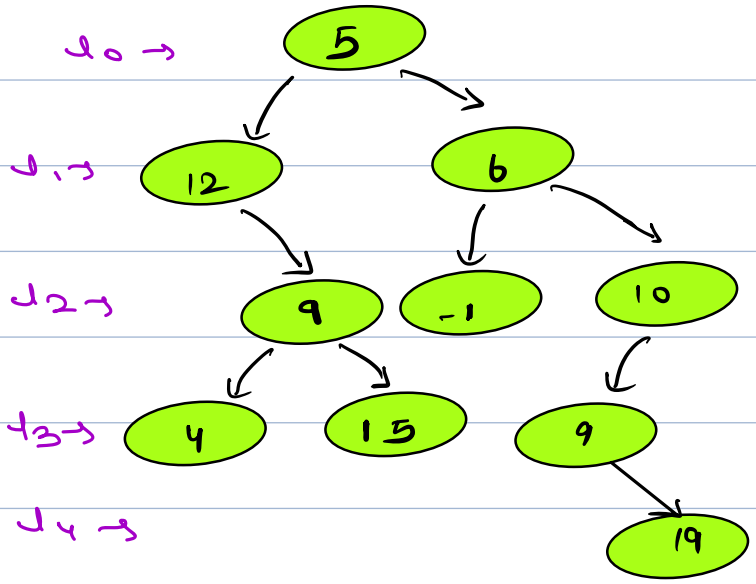
5 \n

12 6 \n

9 -1 10 \n

4 15 9 \n

19



current level Element = ~~5~~ ~~12~~ ~~6~~ ~~9~~ ~~-1~~ ~~10~~ ~~4~~ ~~15~~ ~~9~~ 19

~~5~~ ~~12~~ ~~6~~ ~~9~~ ~~-1~~ ~~10~~ ~~4~~ ~~15~~ ~~9~~ 19

---

5 \n

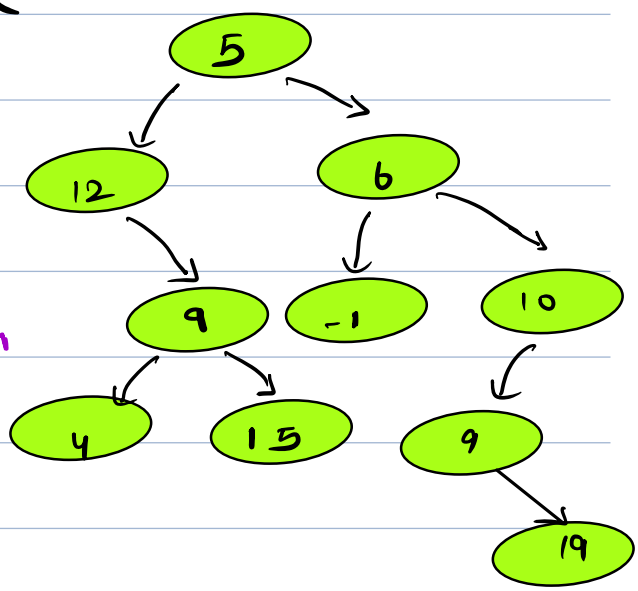
12 6 \n

9 -1 10 \n

4 15 9 \n

~~5~~ ~~12~~ ~~6~~ ~~9~~ ~~-1~~ ~~10~~ ~~4~~ ~~15~~ ~~9~~ ~~19~~

```
void levelOrder(Node root){
    Queue<Node> q;
    q.add(root);
    while(q.size()>0){
        int n = q.size();
        for(i=1; i<=n; i++){
            Node front = q.peek();
            q.remove();
            print(front.data);
            if(front.left!=null){
                q.add(front.left);
            }
            if(front.right!=null){
                q.add(front.right)
            }
        }
        print("\n");
    }
}
```



5 \n

12 6 \n

9 -1 10 \n

4 15 9 \n

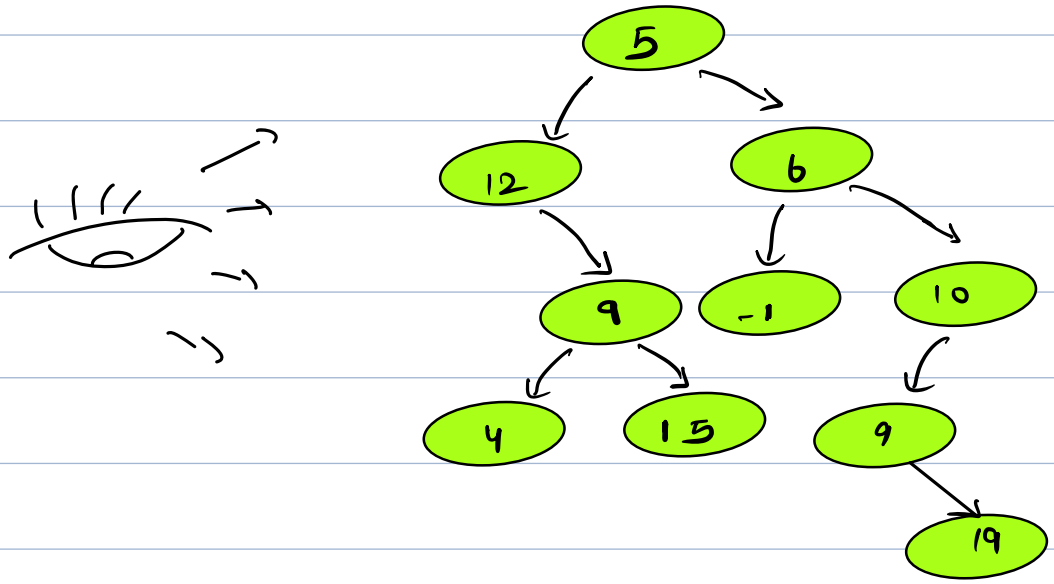
19 \n

T.C → O(n),

S.C → O(n)

Left View :-

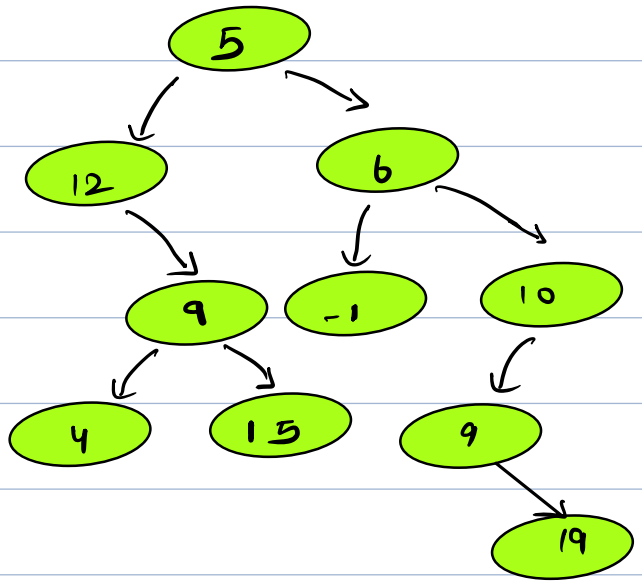
O/P → 5 12 9 4 19




```
void levelOrder(Node root){
    Queue<Node> q;
    q.add(root);
    while(q.size()>0){
        int n = q.size();
        for(i=1; i<=n; i++){
            Node front = q.peek();
            q.remove();
            print(front.data);
            if(front.left!=null){
                q.add(front.left);
            }
            if(front.right!=null){
                q.add(front.right);
            }
        }
        print("\n");
    }
}
```

if (i==1)

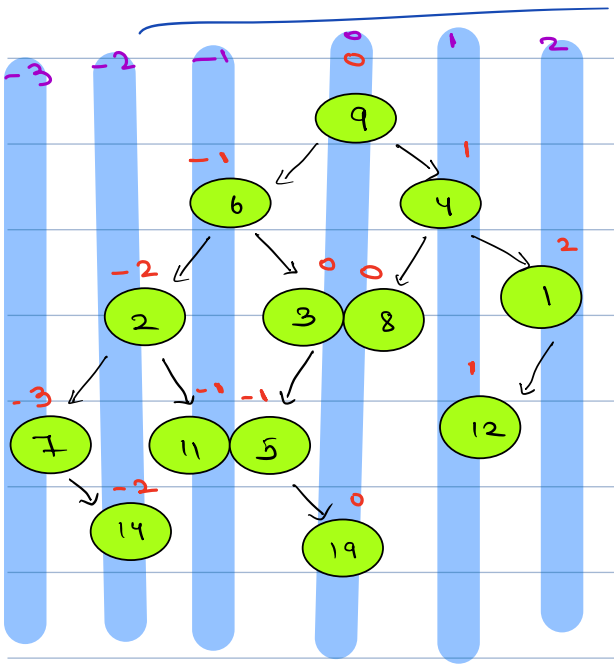
Right View :-




```
void levelOrder(Node root){
    Queue<Node> q;
    q.add(root);
    while(q.size()>0){
        int m = q.size();
        for(i=1; i<=m; i++){
            Node front = q.peek();
            q.remove();
            print(front.data);
            if(front.left!=null){
                q.add(front.left);
            }
            if(front.right!=null){
                q.add(front.right);
            }
        }
        print("\n");
    }
}
```

if (i==m)

Vertical level order traversal (L-R)



Expected o/p :

→ 7
→ 2 14
→ 6 11 5
→ 9 3 8 19
→ 4 12
→ 1

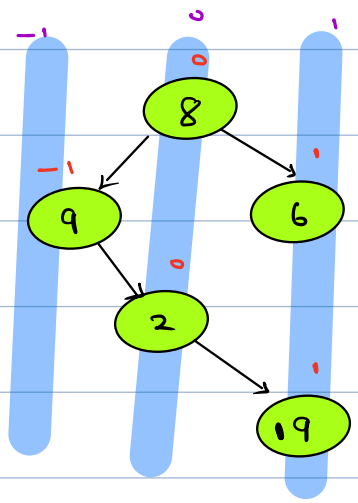
Top View

7
2
6
9
4
1

in(8, 0)

```
in(root, level) {
    in(root.left, level-1);
    hm[level].add(root);
    in(root.right, level+1);
}
```

Example :-



Expected o/p

-1 → 9
0 → 8 2
1 → 6 19

Preorder

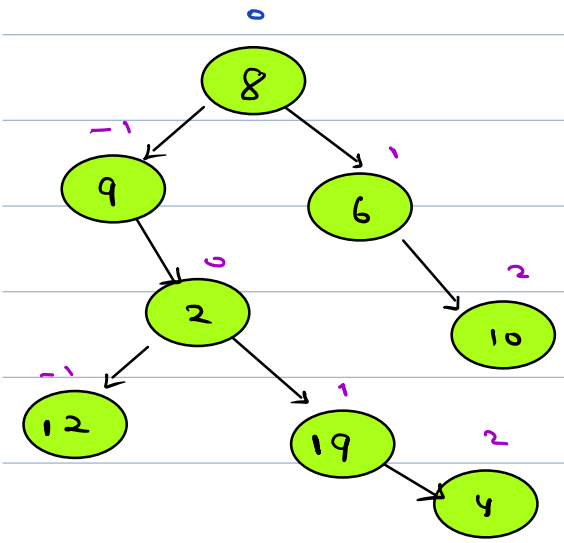
-1 → 9
0 → 8, 2
1 → 19, 6

Postorder

-1 → 9
0 → 2, 8
1 → 19, 6

Inorder

-1 → 9
0 → 2, 8
1 → 19, 6



HM

0 → 8, 2

-1 → 9, 12

1 → 6, 19

2 → 10, 4

minh = -1

maxh = 2

~~(8,0)~~ ~~(9,-1)~~ ~~(6,1)~~ ~~(2,0)~~ ~~(10,2)~~ ~~(12,-1)~~

~~(19,1)~~ ~~(4,2)~~

```
for ( i = minh; i <= maxh; i++) {
    print( hm[i] );
}
```

hm[i].getW()

```
class Pair {
    Node first;
    int second;

    class Pair (x, y) {
        first = x;
        second = y
    }
}
```

T.C → O(n)
S.C → O(n)

```
HashMap<int, List<int>> hmap;
Queue<pair> q;
int minl=0, maxl=0;
q.insert(new pair(root, 0));
while(q.size()>0){
    pair y = q.front();
    q.remove();
    Node t = y.first;   //node
    int l = y.second;   // level
    minl = min(minl, l), maxl = max(maxl, l);
    hm[l].add(t);
    if(t.left != null){
        q.insert(new pair(t.left, l-1));
    }
    if(t.right != null){
        q.insert(new pair(t.right, l+1));
    }
}
for(i=minl; i<=maxl; i++){
    Print(hm[i]);
}
```

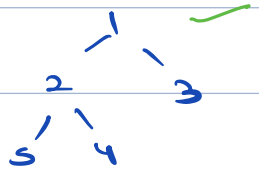
48

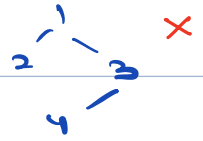
8:36 am - 8:46 am

Types of B.T.

① Proper B.T.

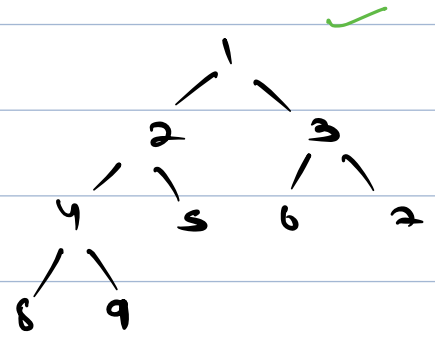
↳ ∀ nodes 0 or 2 children.

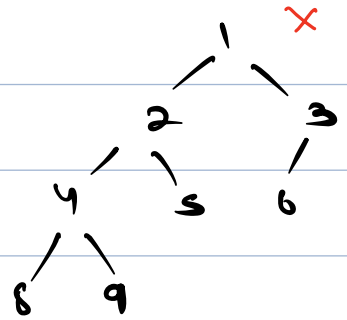


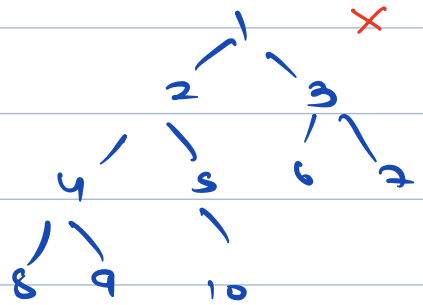


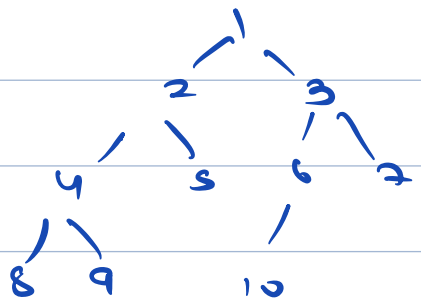
② Complete B.T.

All level must be completely filled, except may be last level but that should also be filled from L-R.



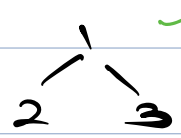


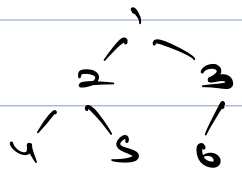




After first null in a level you can't have another node.

③ Perfect BT. → All levels are completely filled.





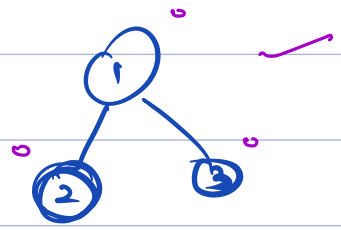
Perfect Binary tree is Proper as well as Complete Binary Tree.

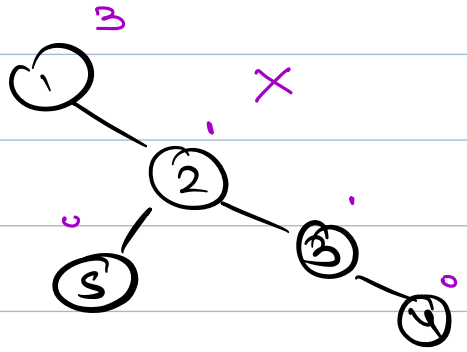
Ques **Balanced Binary Tree**

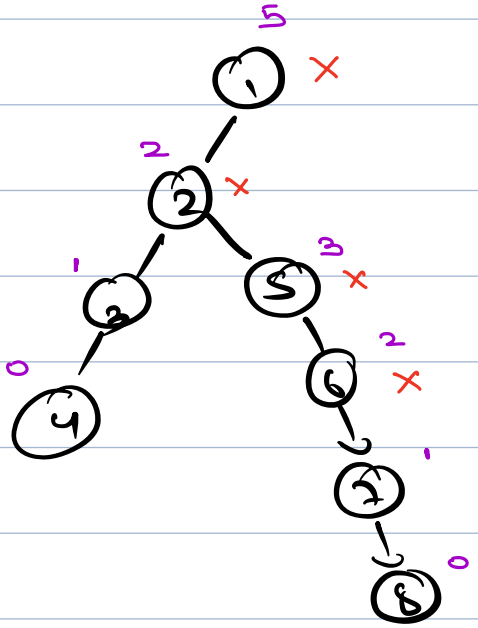
↳ A tree in which N nodes

$|LSTh - RSTh| <= 1$

Height in terms of nodes.

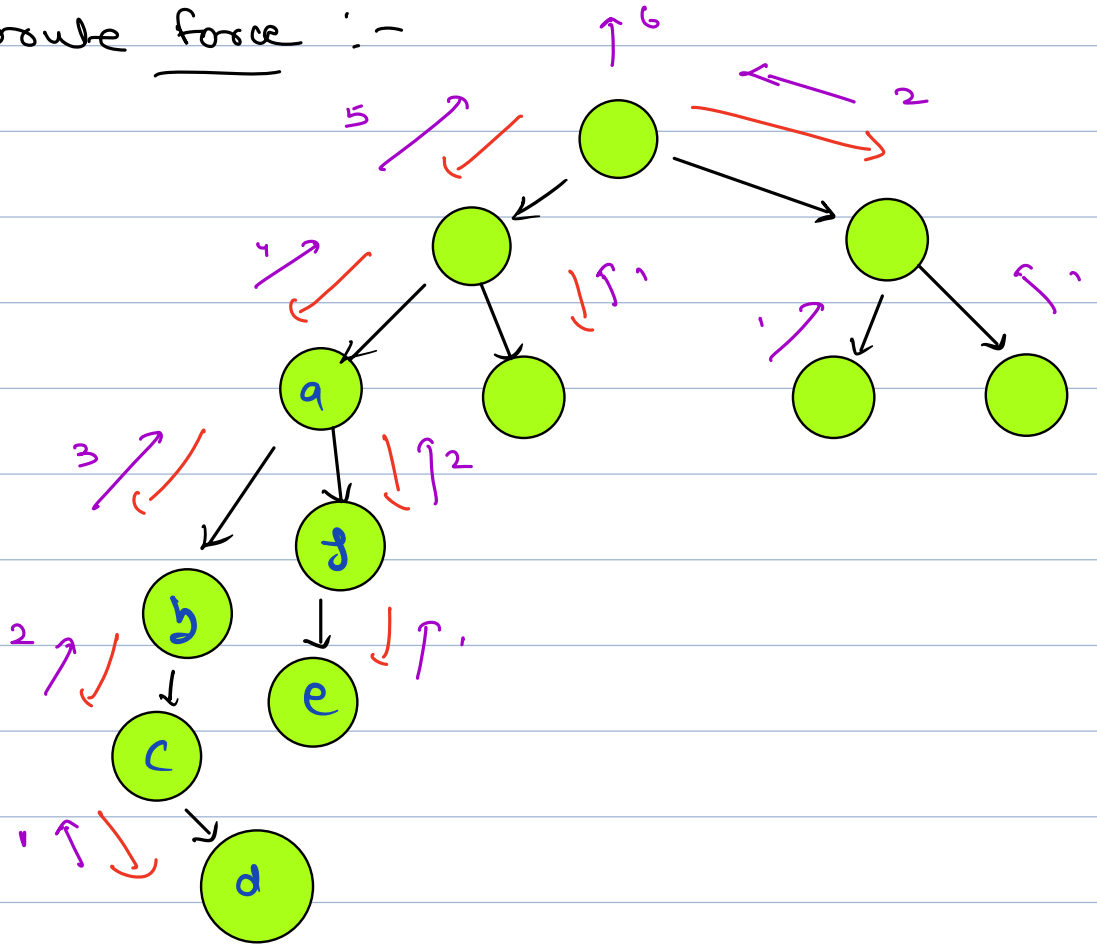






Ques Check if a tree is height balanced.

1) Brute force :-



T.C → O(n)
S.C → O(H)

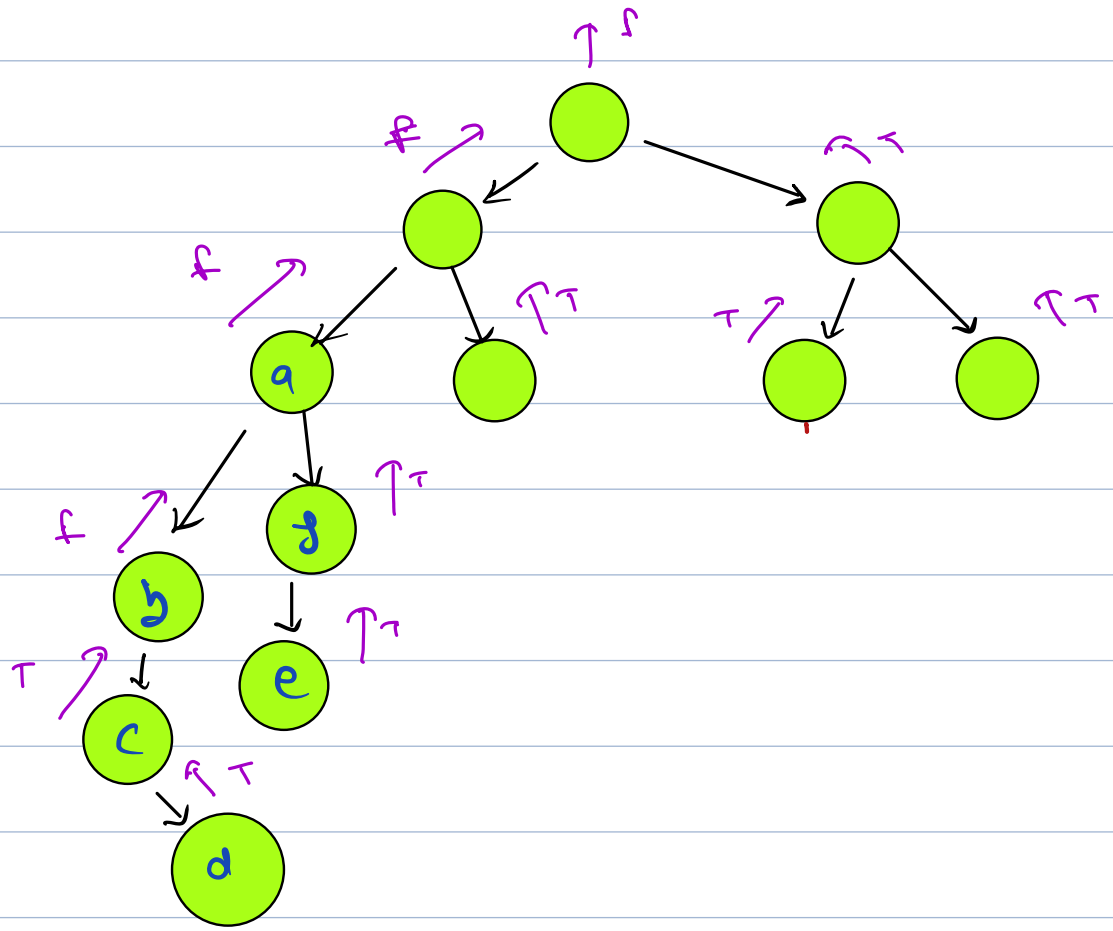
```
int height (node root) {
    if (root == null) return 0;

    lh = height (root.left);
    rh = height (root.right);

    return max (lh, rh) + 1
}
```

T.C → $O(n^2)$

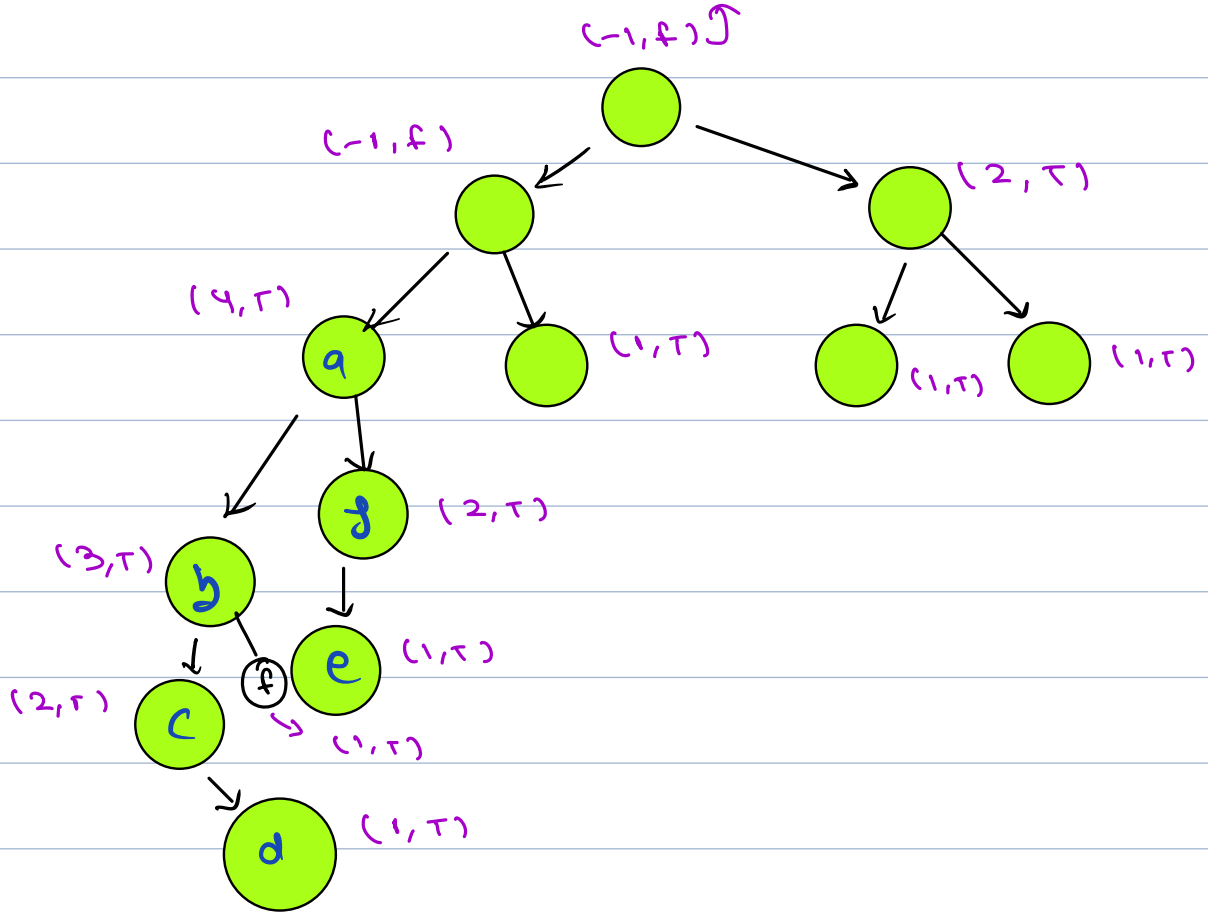
```
boolean isHeightBalanced (node root) {
    if (root == null) return True;
    lheight = height (root.left);
    rheight = height (root.right);

    if (abs (lheight - rheight) > 1) {
        return false;
    }
}
```

3

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



2) Optimized idea.




```
class Pair {
    int height;
    boolean isBalanced
}
```

→ `bool isBalanced b.T ( node root){`

`return helper(root).isBalanced.`

`}`

T.C -> O(n), S.C -> O(h);

```
private pair helper(node A){
    if(A==null){ return new pair(0,T)}

    pair lp = helper(A.left);
    pair rp = helper(A.right);

    if(lp.isBalanced== false || rp.isBalanced==false){
        return new pair(-1,f);
    }
    else if(|lp.height - rp.height|>1){
        return new pair(-1,f);
    }
    else{
        return new pair(max(lp.height,
                            rp.height)+1, T);
    }
}
```