Agenda

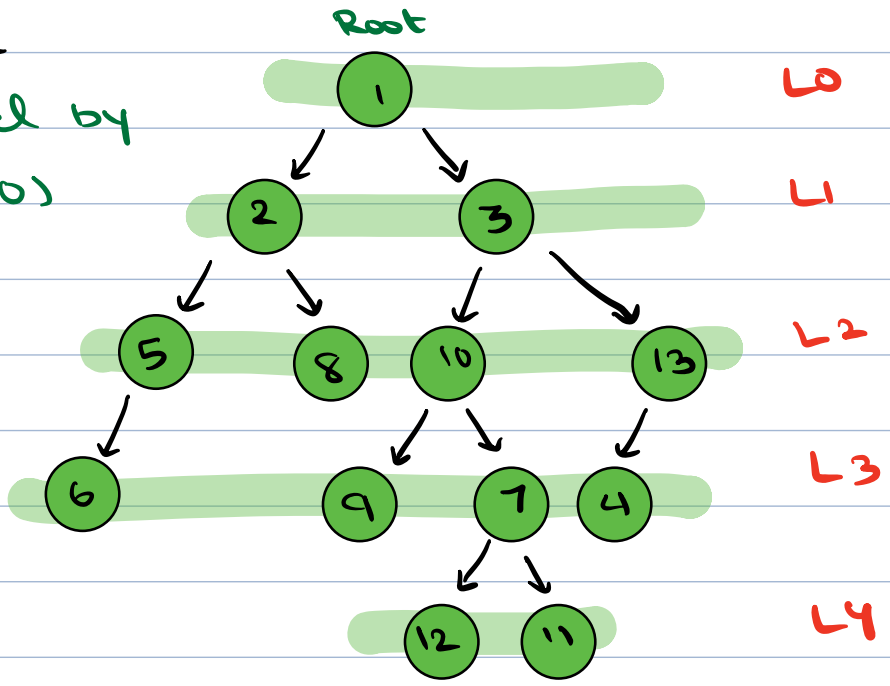
1. Level Order Traversal
2. Right view
3. Types of Binary Tree
4. Height and Balanced Binary Tree
5. Construct Binary Tree

Contest 17 Jan 9-10:30 PM

Sorting, Searching,
Linked List, Stacks and Queue

Level Order Traversal

(Print all nodes level by level starting from L0)



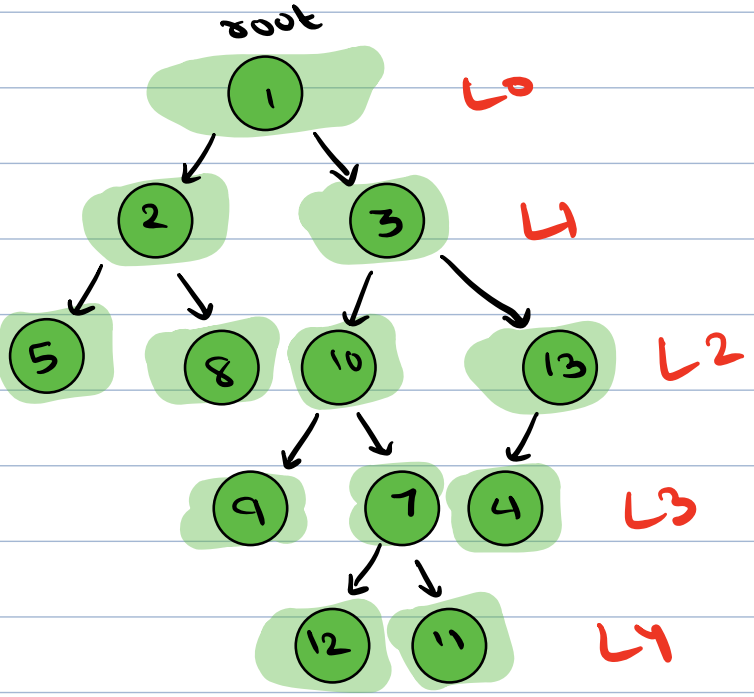
O/P :

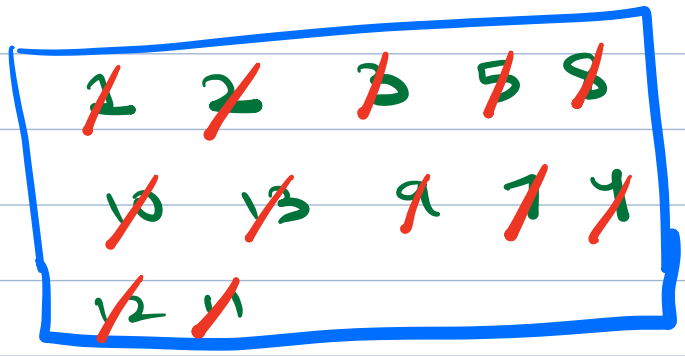
1, 2, 3, 5, 8, 10, 13,

6, 9, 7, 4, 12, 11

Will the last level node always be a leaf node ? YES

Which is preferred traversal to print all nodes from top to bottom ?





O/P: 1 2 3 5

8 10 13 9 7

4 12 11

BFT → nodes of a particular level

or

nodes of 2 consecutive levels

↓

Queue (Insertion → Front, Deletion → End)

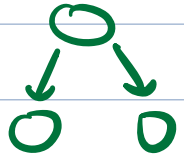
```
// I/P : Node root
Queue <Node > q
q. enqueue (root)

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)

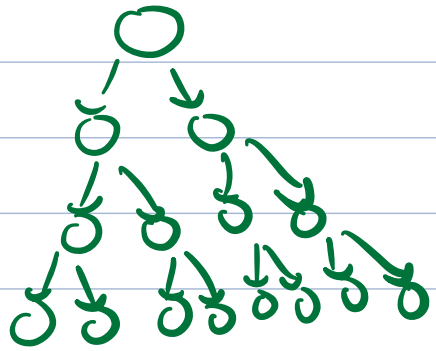
SC : O(N)

↓

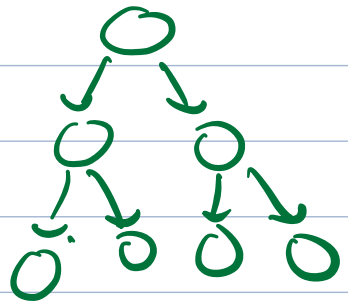
Queue



N = 3 Last level = 2

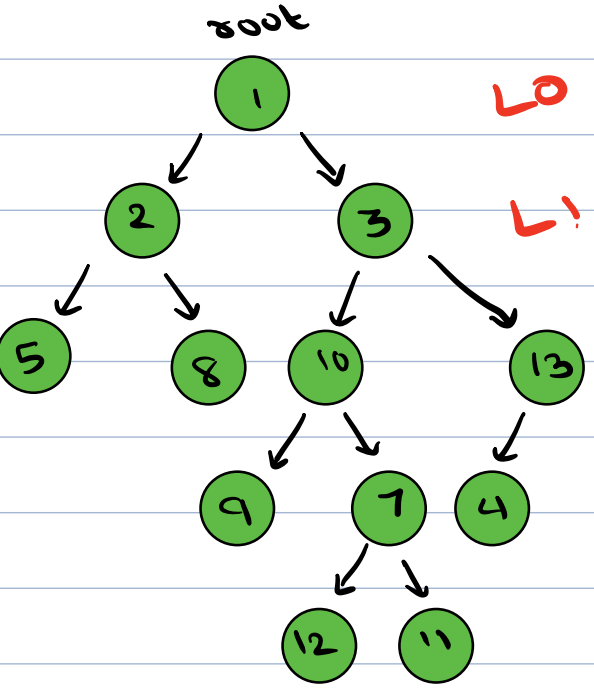


N = 15 Last = 8



7 4

Total = N
nodes in Last Level
= N/2



L0

L1

O/P:

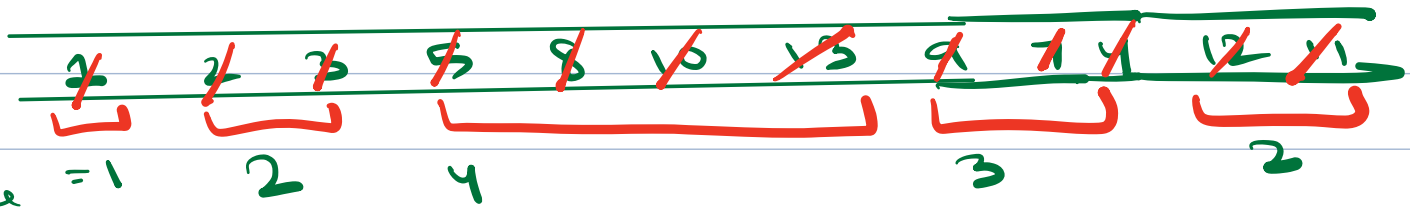
1 ln

2 3 ln

5 8 10 13 ln

9 7 4 ln

12 11



qsize
= levelsize

=1 2 4 3 2

cnt =1 1 2 1 2 3 4 1 2 3 1 2

1 ln

2 3 ln

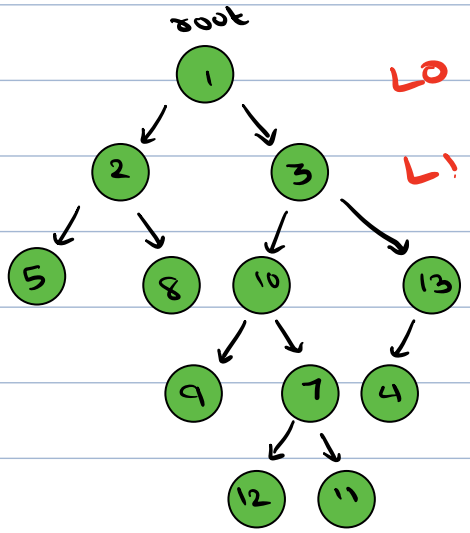
5 8 10 13 ln

9 7 4 ln

12 1

TC : O(N)
SC : O(N)

```
Queue <Node> q
q.enqueue (root)
while (!q.isEmpty()) {
    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")
}
```



L0

L1

level = 1 2 4 3

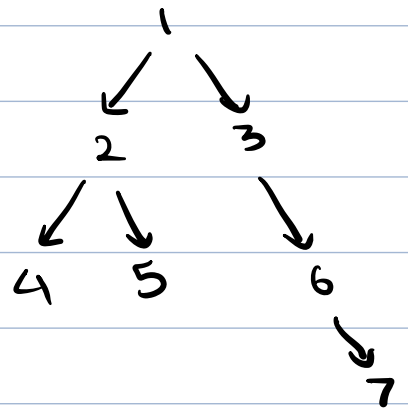
cnt = 1 2 2 1 2 3 4 1 2 3

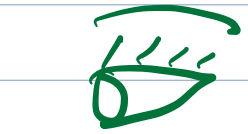
1 \n

2 3 \n

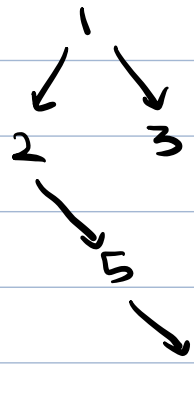
5 8 10 13 \n

Right View of Binary Tree

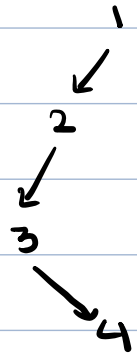




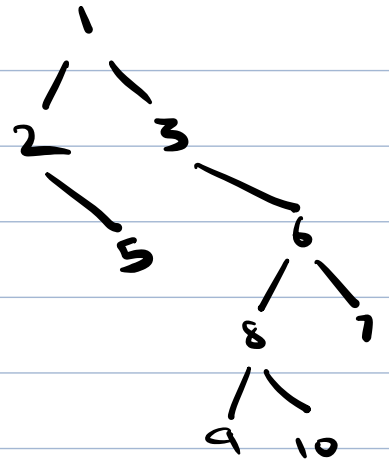
1 3 6 7



1 3 5 6



1 2 3 4



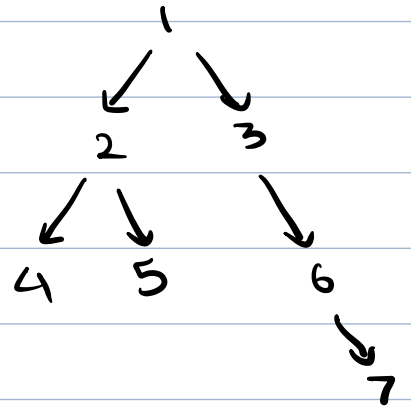
1 3 6 7 10

Print last node of every level

TC: O(N)
SC: O(N)

```
Queue <Node> q
q.enqueue(root)
while (!q.isEmpty()) {
    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)
    }
    print("\n")
}
```

Left view (First node of every level)

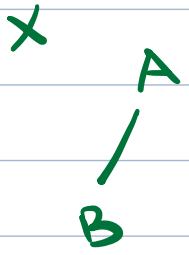


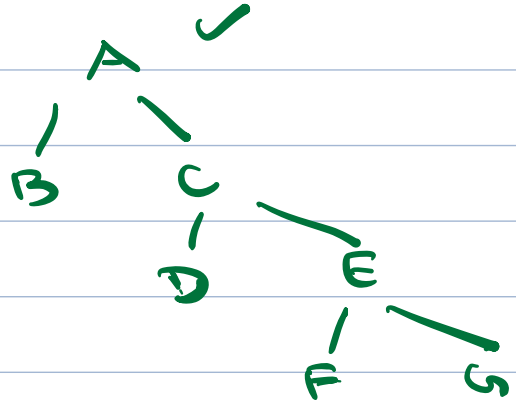
O/P : 1 2 4 7

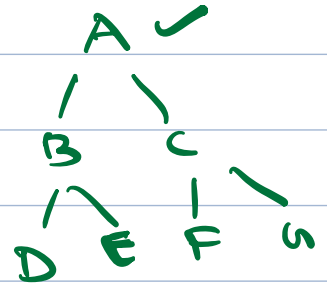
if (cnt == 1

print (cur.data)

Proper BT → Every node has 0 or 2 children.

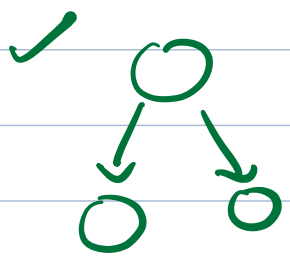


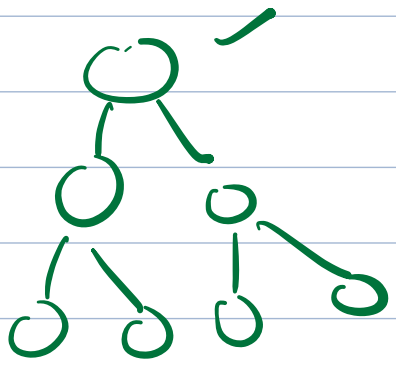


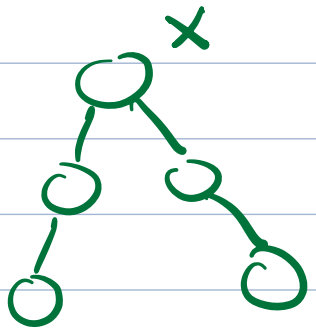


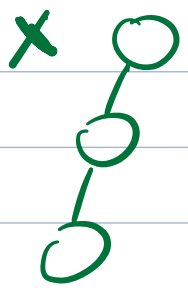
Complete BT : All levels are filled except possibly last level

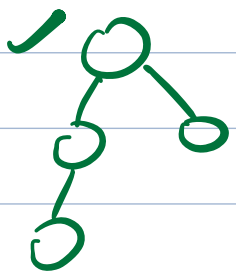
Last level can be partially filled (left to right)

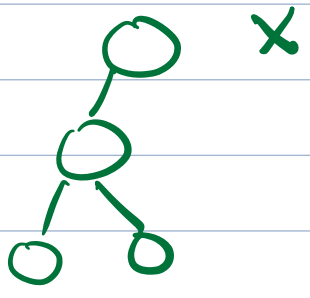




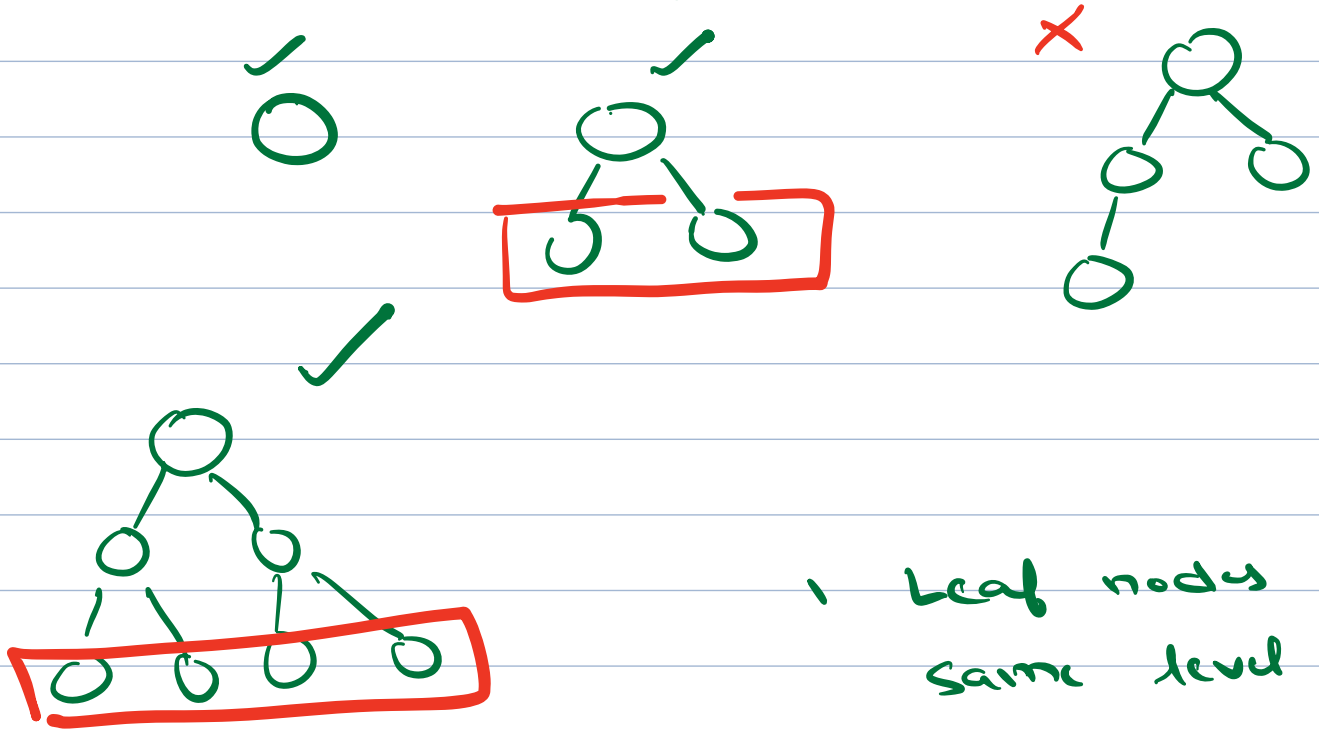








Perfect BT : All levels are completely filled



1. Leaf nodes are at same level

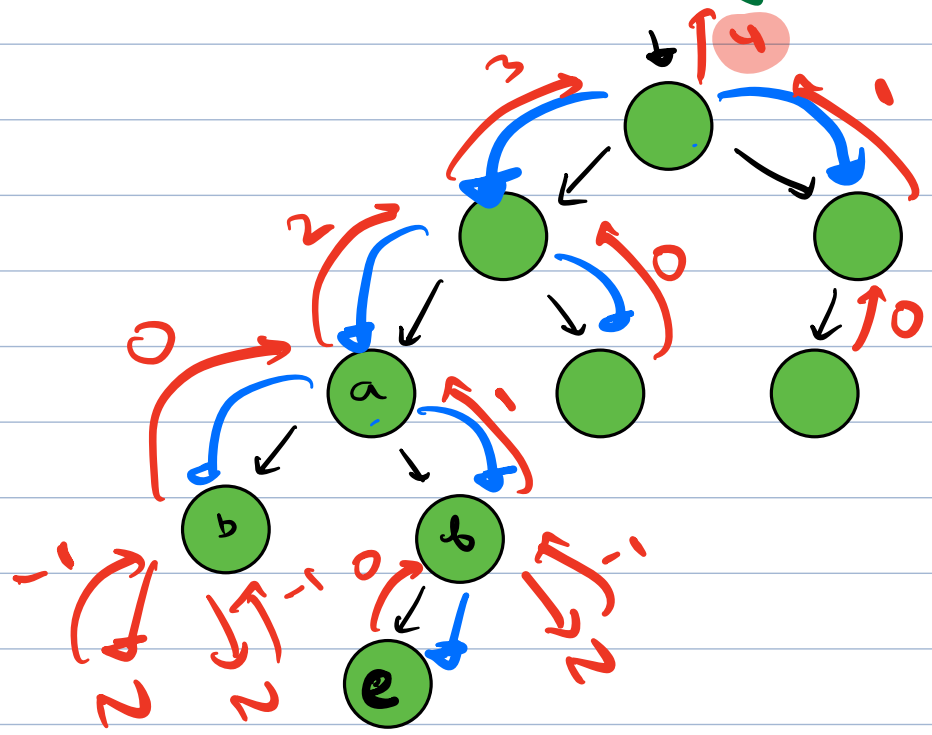
2. Internal nodes have exactly 2 children

Perfect BT is proper and complete.

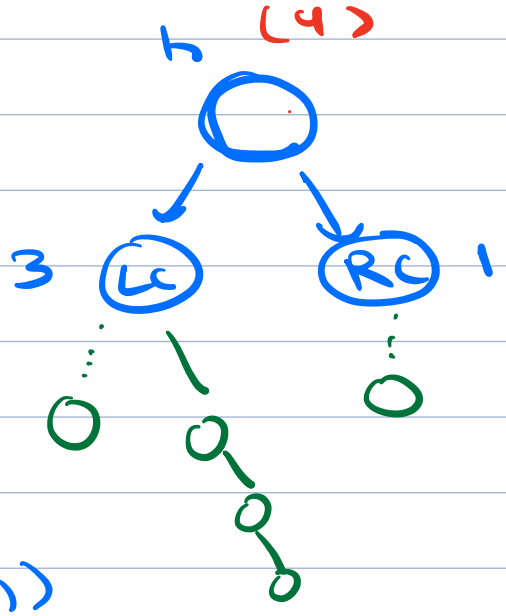
10:40

H of node = longest path from node to leaf (edges)

Height of tree = height of root = 4



Nodes = edges + 1



h(node) = max(h(LC), h(RC)) + 1

h(leaf) = 0

Postorder
L R N

Node's height is dependent on children

TC: O(N)
SC: O(H)

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

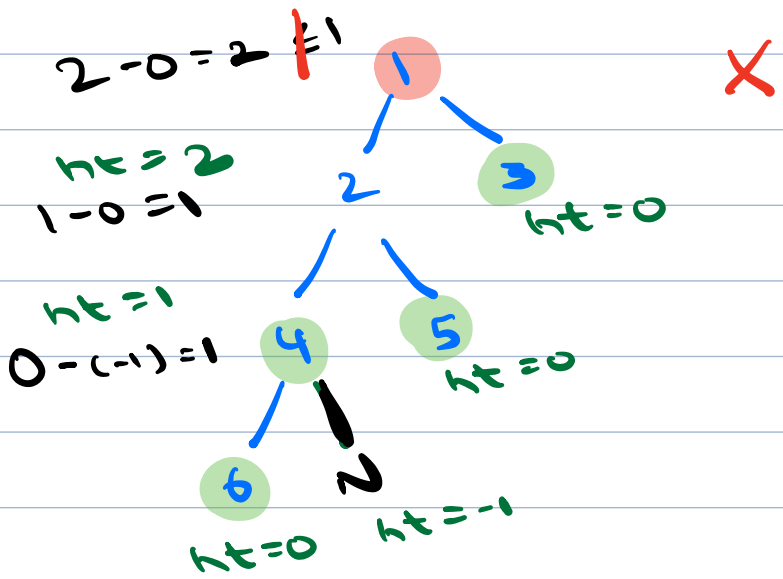
Balanced Binary Tree

Check if a tree is height balanced

↓

A tree in which for every node

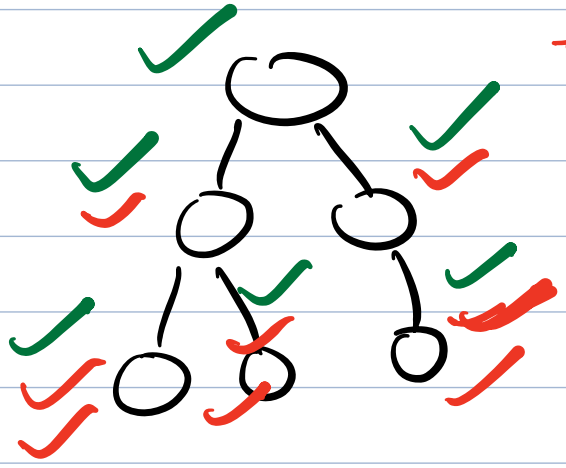
$$|LST_h - RST_h| \leq 1$$



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) &&
           isBalanced (root.right)
}
```

N L R



TC : O(N * N) = O(N²)

SC : O(H + H)

↓ isBalanced

Height

✓ → B

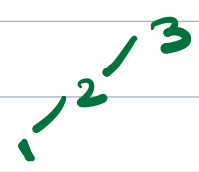
✓ → H

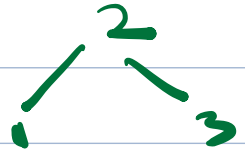
Optimized :

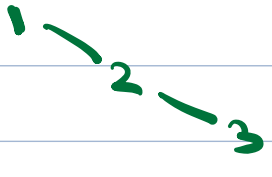
```
bool isBalanced = true                          TC : O(N)
                                                SC : O(H)
int height (Node root) {
    if (root == NULL)  return -1

    int lh = height (root.left)
    int rh = height (root.right)
    if (abs (lh - rh) > 1)
            isBalanced = false
    return  max (lh, rh) + 1
}
```

Inorder : 1 2 3

LNR

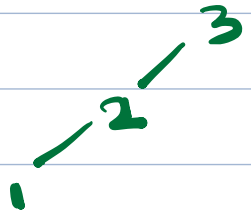


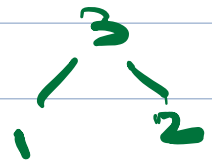


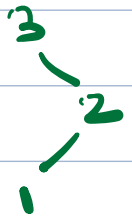


Postorder : 1 2 3

LRN



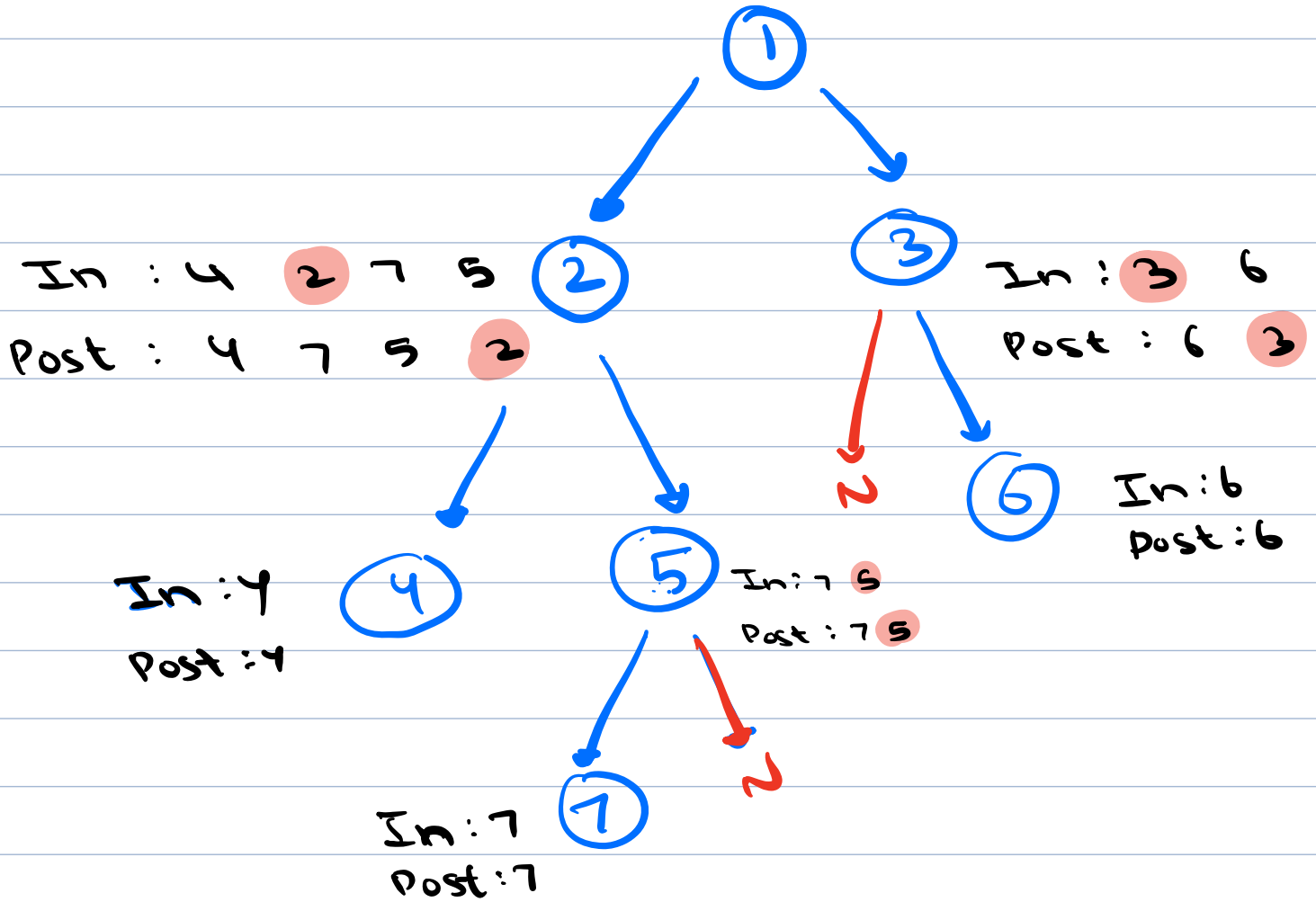




Q. Construct binary tree from inorder and post order traversal. (unique values)

LNR Inorder : 4 2 7 5 1 3 6

LRN Postorder : 4 7 5 2 6 3 1



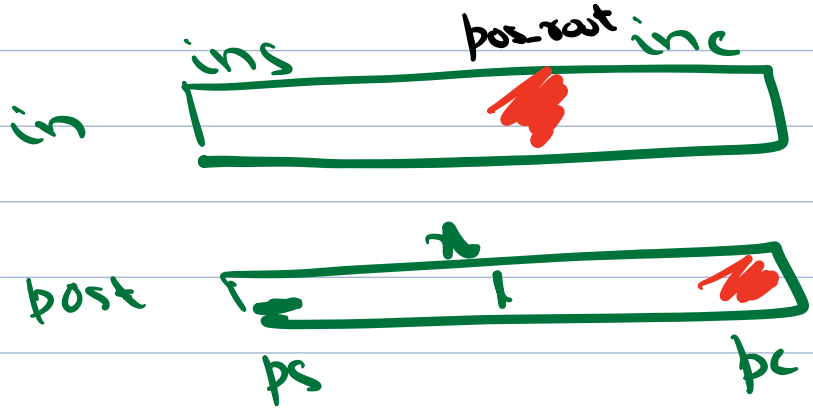
if in[ ] = N

return

Construct (in, post, 0, n-1, 0, n-1)

// Given in[] and post[], construct a tree from given values & return root

```
Node construct(in[], post[], ins, inc, ps, pe){

    if (ins > inc)  return NULL

    int element = post[pe]
    Node root = new Node(element)
    int pos-root
    for (i = ins, i ≤ inc, i++){
        if (in[i] == element){
            pos-root = i   break;
        }
    }

    int x = pos-root + ps - ins -1

root.
left = construct(in, pos, ins, pos-root-1,
                 ps, x)

root.
right = construct(in, pos, pos-root+1, inc)
                  x+1, pe-1)

    return root

}
```



LST in : ins → pos_root - 1

post : ps → r

RST in : pos_root + 1 → inc

post : r + 1 → pe - 1

r - ps = pos_root - 1 - ins

r = pos_root + ps - ins - 1

LNR in : 4 3

LRN post : 3 4

Construct (0 →(in) 1, 0 →(post) 1)

4

ins, root - 1

0 -1

pos_root = 0

ins = 0 inc = 1

ps = 0 pe = 1

TC : O(N²)
SC : O(H)

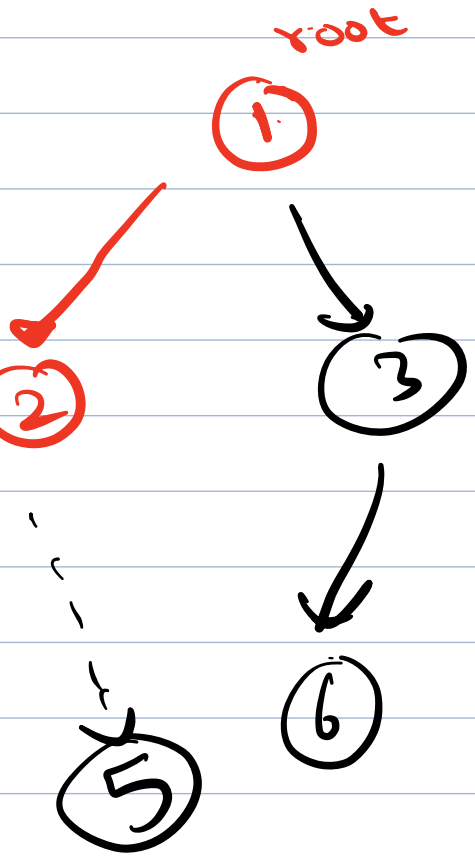
	0	1	2	3	4	5
in :	4	2	5	1	6	3
post :	4	5	2	6	3	1

N = 6

construct (ins ine ps pe
0 → 5, 0 → 5)

pos - root = 3

root



(ins ine , ps → pe
(0 → 2 , 0 → 2)

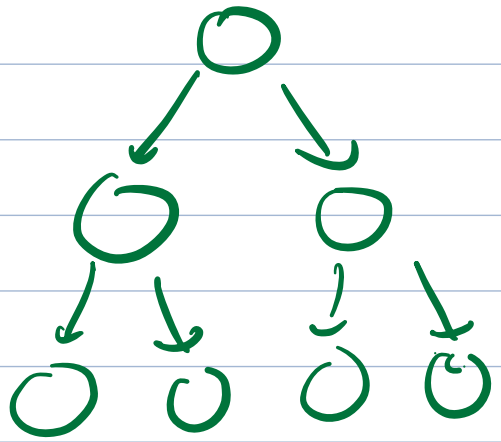
x = pos - root → ps - ins - 1

pos - root = 1

(ins → ine ps → pe
(0 → 0, 0 → 0)

pos - root = 0

(ins, ine ps, pe
(0, -1, 0, -1) N



$0 \rightarrow 2^0$

$1 \rightarrow 2^1$

$2 \rightarrow 2^2$

$h \rightarrow 2^h$

root-pos

	0	1	2	3	4	5	6
In	4	2	7	5	1	3	6
Post.	4	7	5	2	6	3	1

ins → ins

In : 0 → 6

p : ps pe
0 → 6

①

root-post+1 → ine

In : 5 6

post : 4 5

ins → root-pos-1

0 → 3

elem in LST → root-pos-1 - ins + 1

= root_pos - ins

ps → ps + elem -1

n = ps + root-pos -ins -1