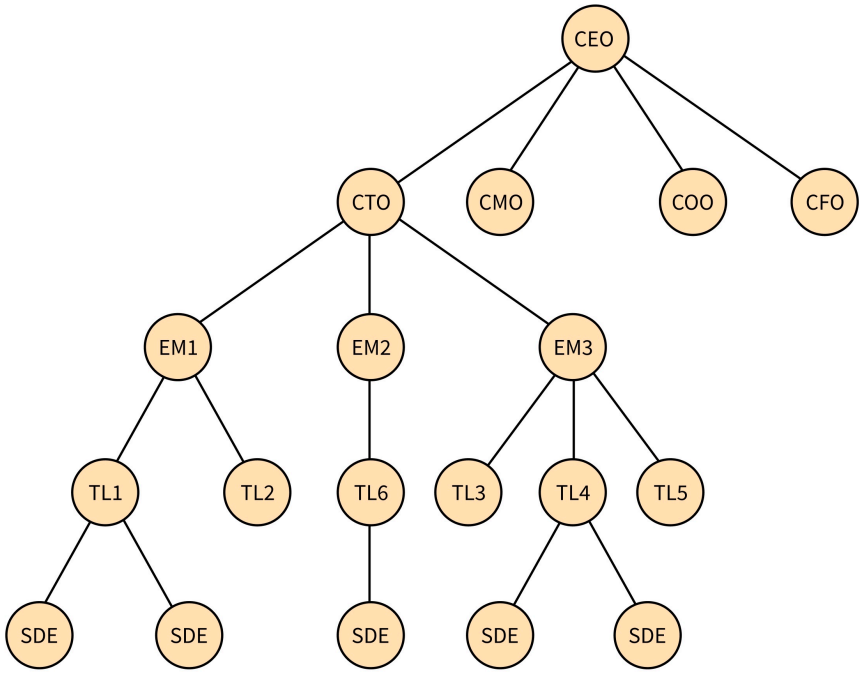
SCALER

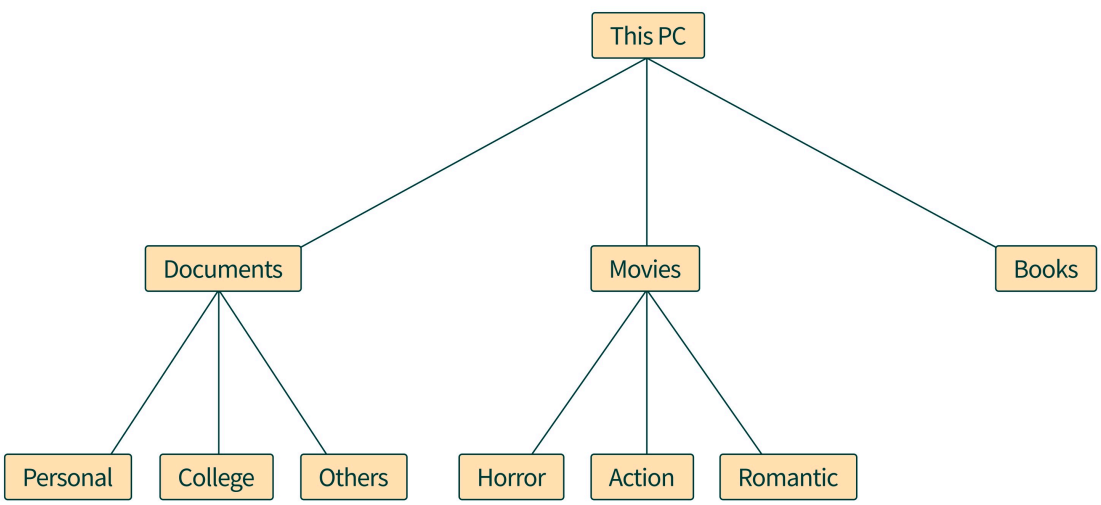
Trees - 1

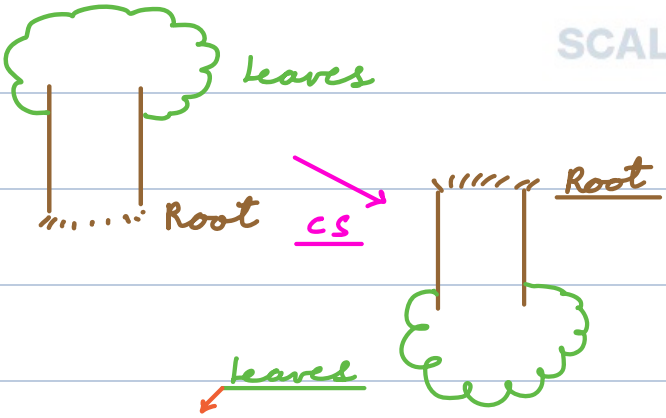
TABLE OF CONTENTS

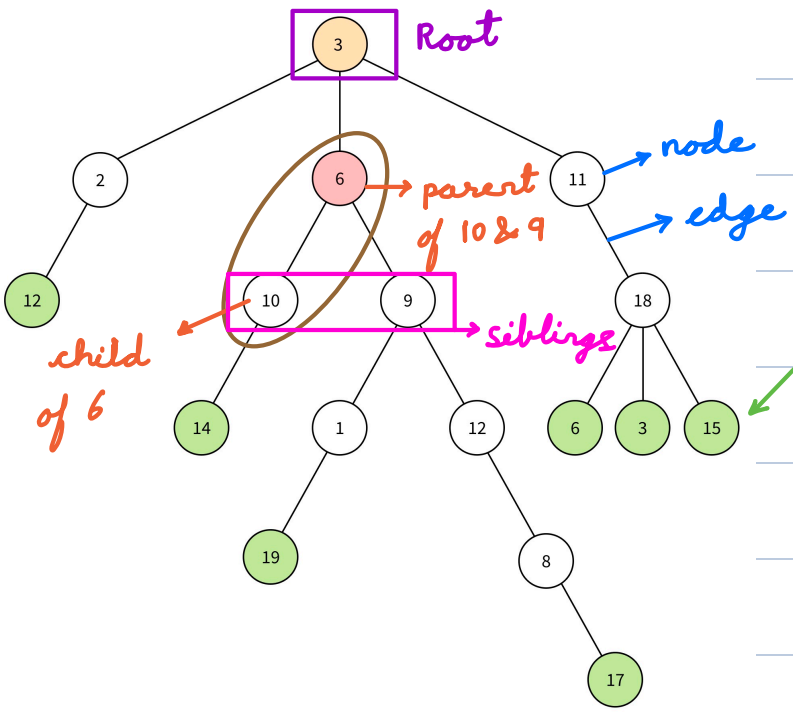
Trees

Hierarchical Data









Nodes without children.

Height of node → count of edges (distance) to travel from current node to farthest leaf.

leaf

height (18) = 1

height (9) = 3

height (leaf) = 0

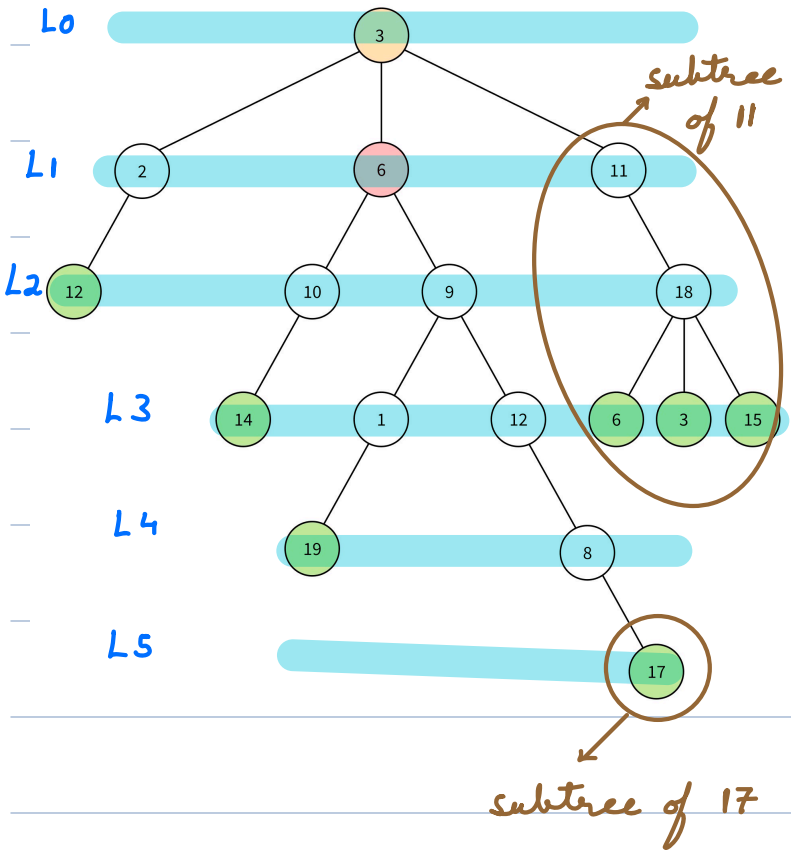
v.imp → Height of tree = Height (root) = 5

Depth / Level of node → count of edges (distance) to travel from root to reach the current node.

root

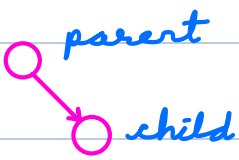
depth (9) = 2

depth (14) = 3



Subtree of a node x →

All the nodes we can travel from node x are part of subtree of x.

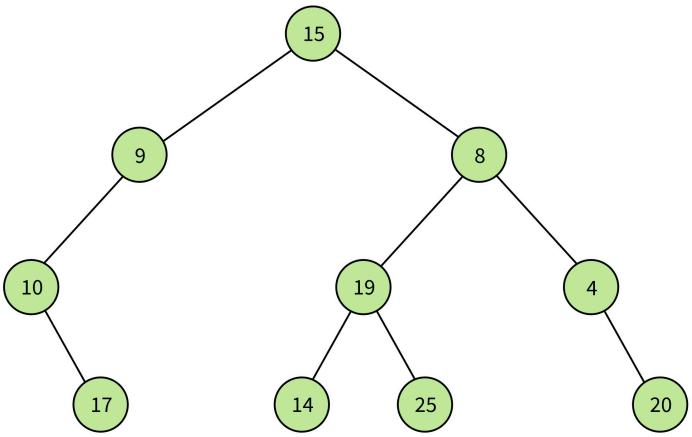


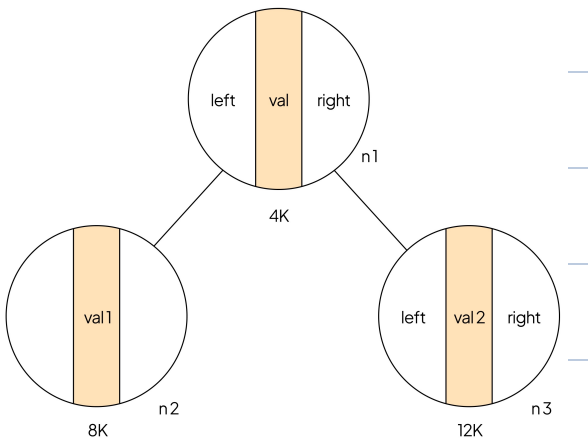
subtree (root) = complete tree

Binary Tree

∀ nodes, max # children = 2

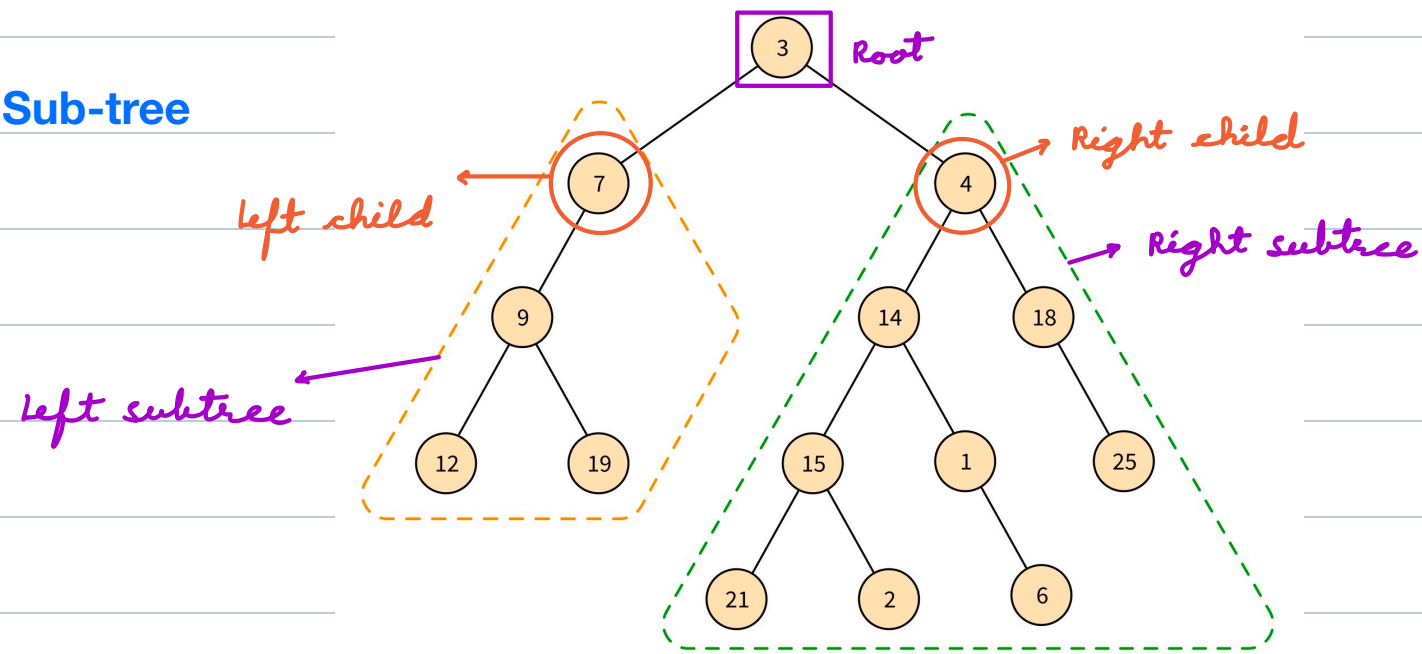
children = {0, 1, 2}






```
class TreeNode {
    int val;
    TreeNode left, right;
}
```

Sub-tree



Preorder N L R

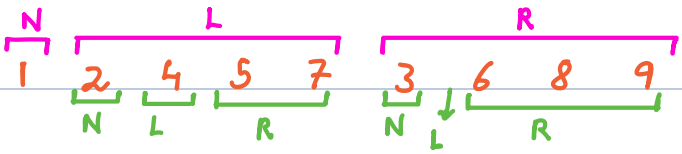
Inorder L N R

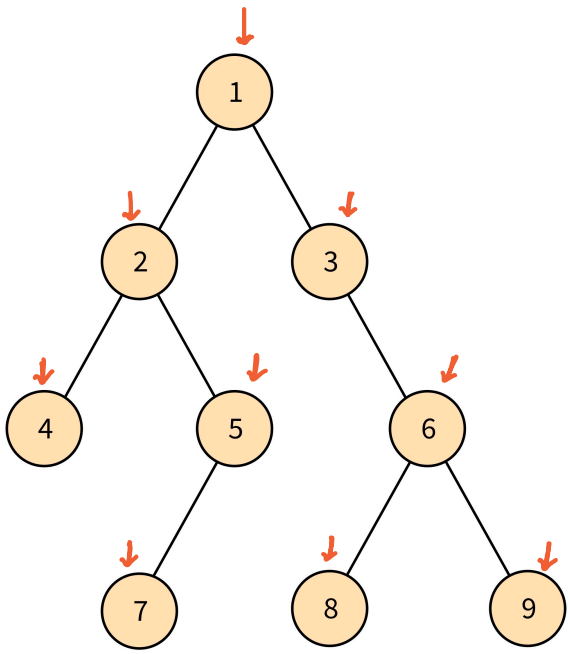
Postorder L R N

Levelorder → Next class

Binary Tree Traversal

1. Pre-order Traversal N L R

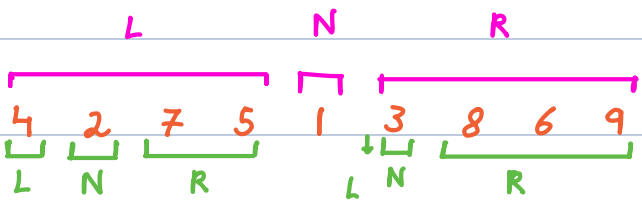


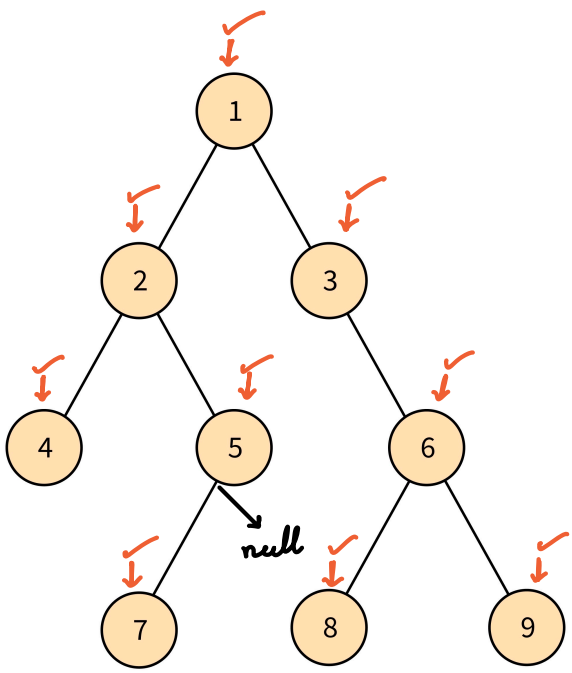



```
void preorder (root) {
    if (root == null) return
    print (root.val)          // Node
    preorder (root.left)      // Left
    preorder (root.right)     // Right
}
```

TC = O(N) SC = O(H)

2. In-order Traversal L N R

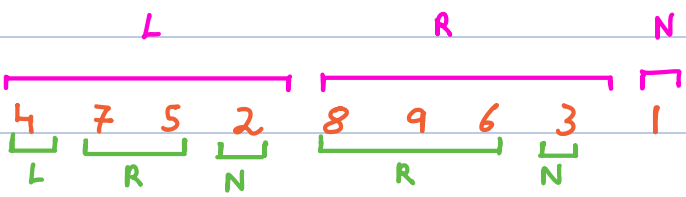


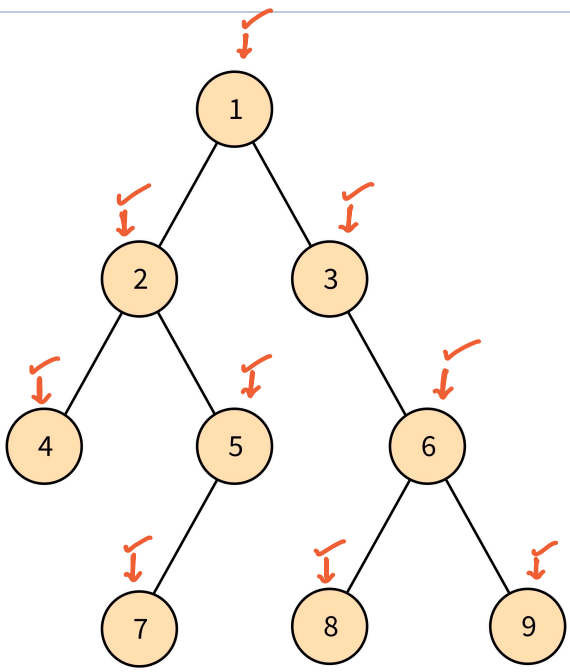


```
void inorder (root) {
    if (root == null) return
    inorder (root.left)  // Left
    print (root.val)     // Node
    inorder (root.right) // Right
}
```

TC = O(N) SC = O(H)

2. Post-order Traversal L R N





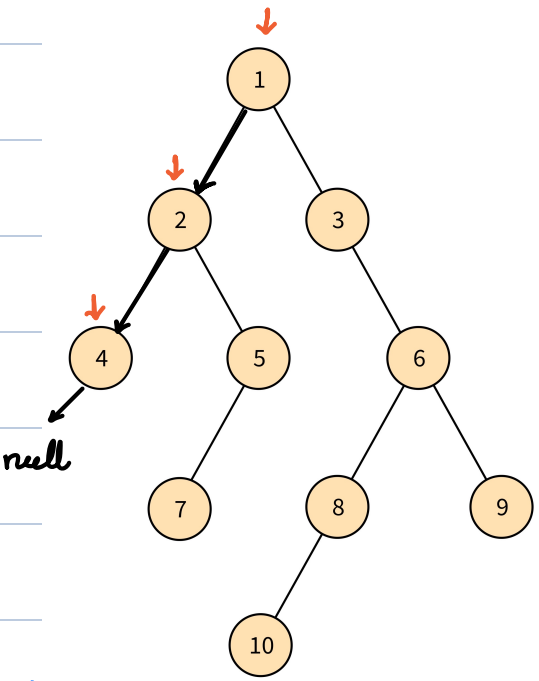

```
void postorder (root) {
    if (root == null) return
    postorder (root.left)  // Left
    postorder (root.right) // Right
    print (root.val)       // Node
}
```

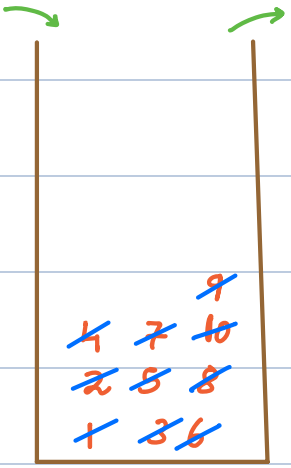
TC = O(N) SC = O(H)

Iterative In-order Traversal

L N R

DS to use → Stack





1) st.push(cur)

cur = cur.left

2) cur = st.pop()

print(cur.val)

cur = cur.right

cur = root

o/p → 4 2 7 5 1 3 10 8 6 9

4 null 2 5 7 null 7 null 5 null 1 3 null 3 6 8 10 null 10 null 8 null 6 9 null 9 null

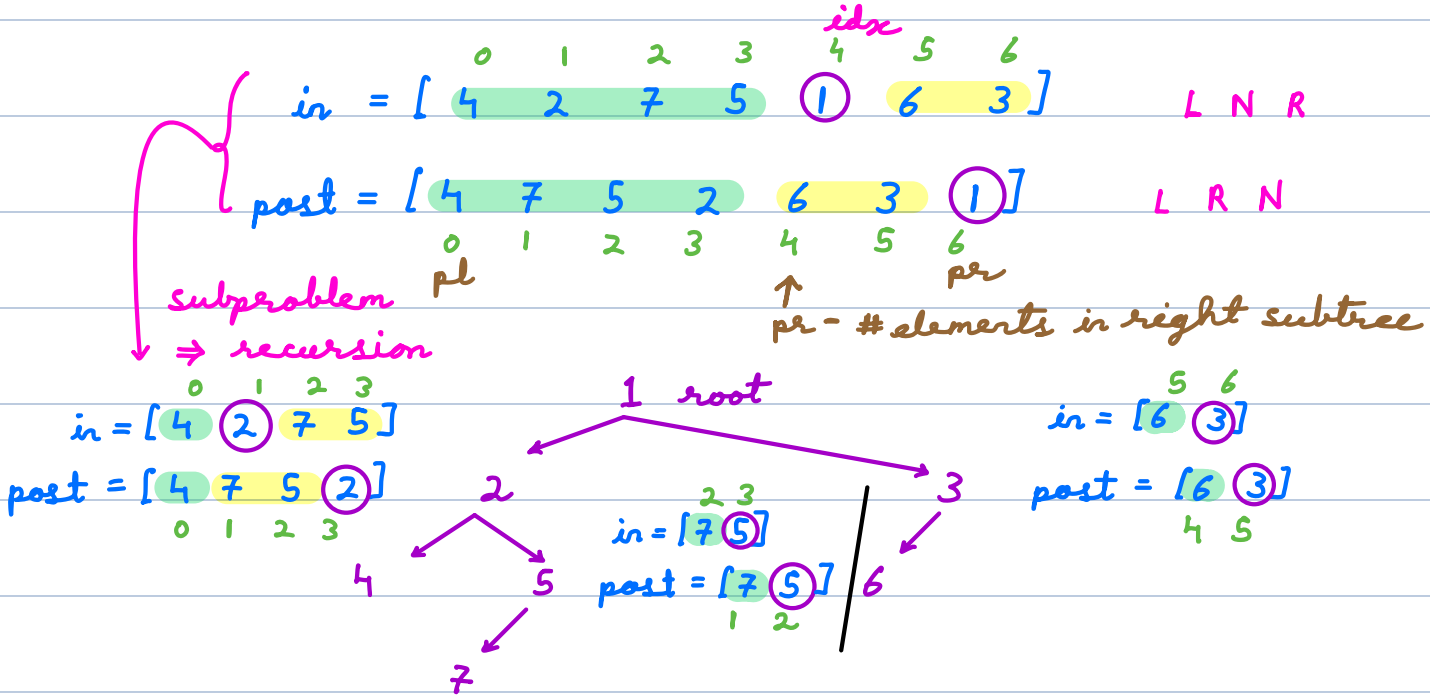
```
cur = root
while (cur != null || !st.isEmpty()) {
    if (cur != null) { st.push(cur)
                       cur = cur.left }
    else { cur = st.pop()
           print(cur.val)
           cur = cur.right }
}
```

TC = O(N) SC = O(H)

H.W → Try iterative code for preorder & postorder. ✓

< Question > : Construct the binary tree from in-order and post-order.

Given two arrays in[ ] , post[ ]. (unique elements)




```
TreeNode build ( in[], il , ir , post[], pl, pr) {
    if ( ir < il ) return null // empty array
    root = new TreeNode ( post[pr] )
    idx = index ( in , root.val ) // iterate <A[i], i> → hashmap ✓
    // in → left   l = il   r = idx - 1
    //      right  l = idx+1   r = ir
    cnt_r = ir - idx                    // R-L+1  [L R]
    // post → left   l = pl   r = pr - cnt_r - 1
    //        right  l = pr-cnt_r   r = pr - 1
    root.left = build (in, il, idx-1, post, pr-cnt_r-1)
    root.right = build (in, idx+1, ir, post, pr-1)
```

return root

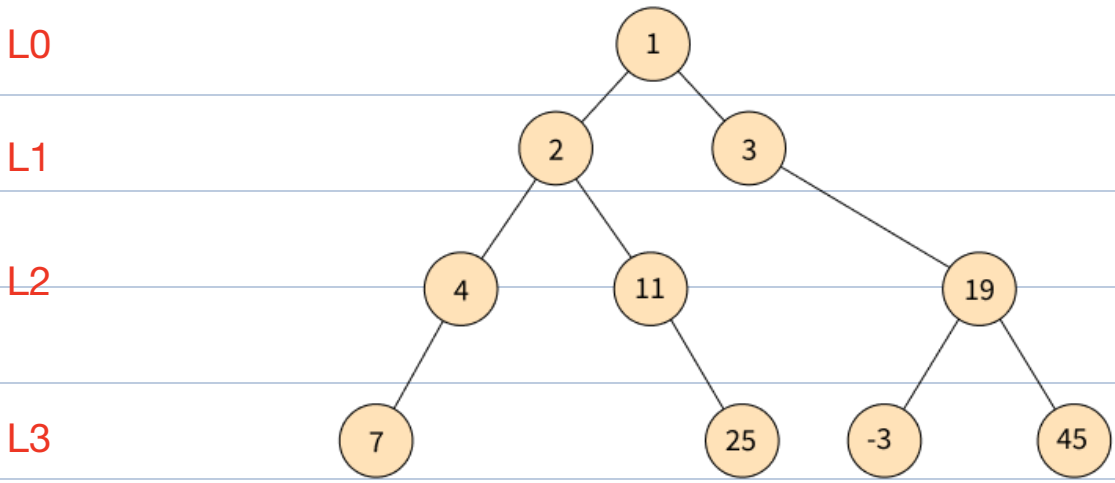
}

TC = O(N) SC = O(N)

Trees 2

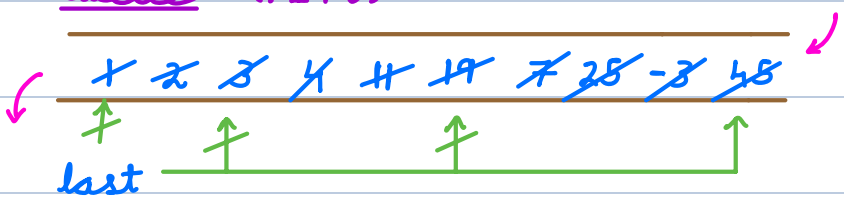
TABLE OF CONTENTS

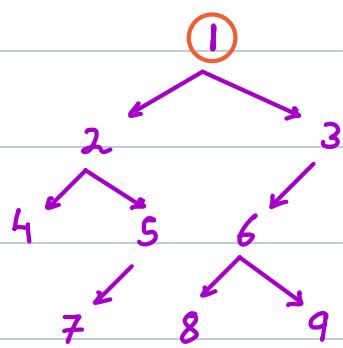
Level Order Traversal



o/p → 1
2 3
4 11 19
7 25 -3 45

Queue (FIFO)





last = 1 3 6 9

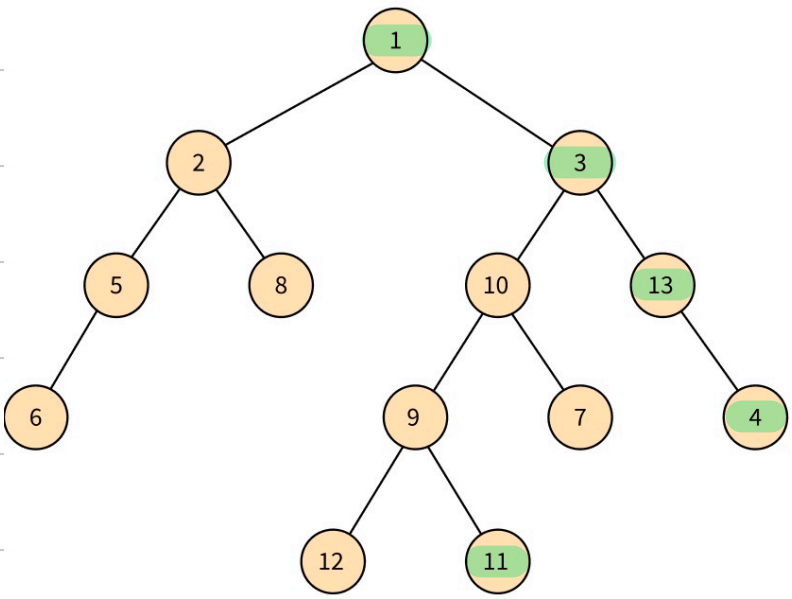
x = 1 2 3 4 5 6 7 8 9

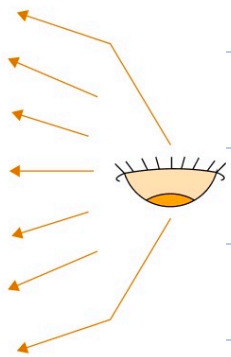
o/p → 1
2 3
4 5 6
7 8 9

```
q.enqueue(root)
last = root
while (!q.isEmpty()) {
    x = q.dequeue()
    print(x.val)
    if (x.left != null)  q.enqueue(x.left)
    if (x.right != null) q.enqueue(x.right)
    if (x == last && !q.isEmpty()) {
        print("\n")
        last = q.rear()
    }
}
```

TC = O(N) SC = O(N)

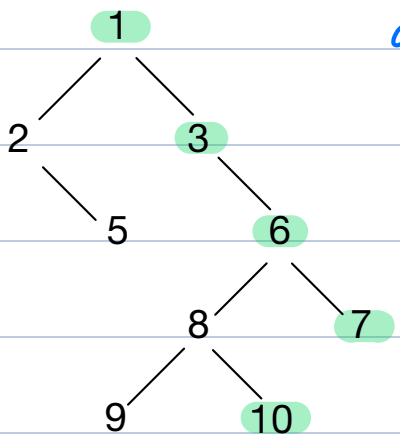
< Question > : Find right view of binary tree.





o/p → 1 3 13 4 11

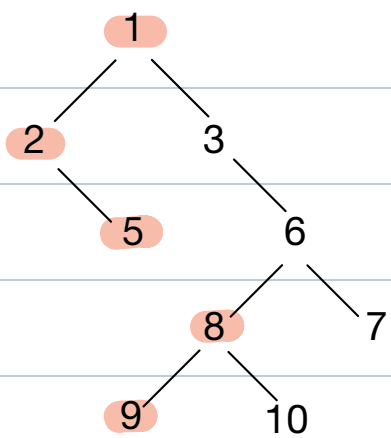
Quiz :



o/p → 1 3 6 7 10

Sol → Print last node of the level. ✓

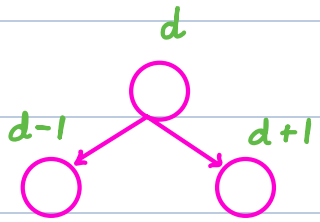
< Question > : Find left view of binary tree.



o/p → 1 2 5 8 9

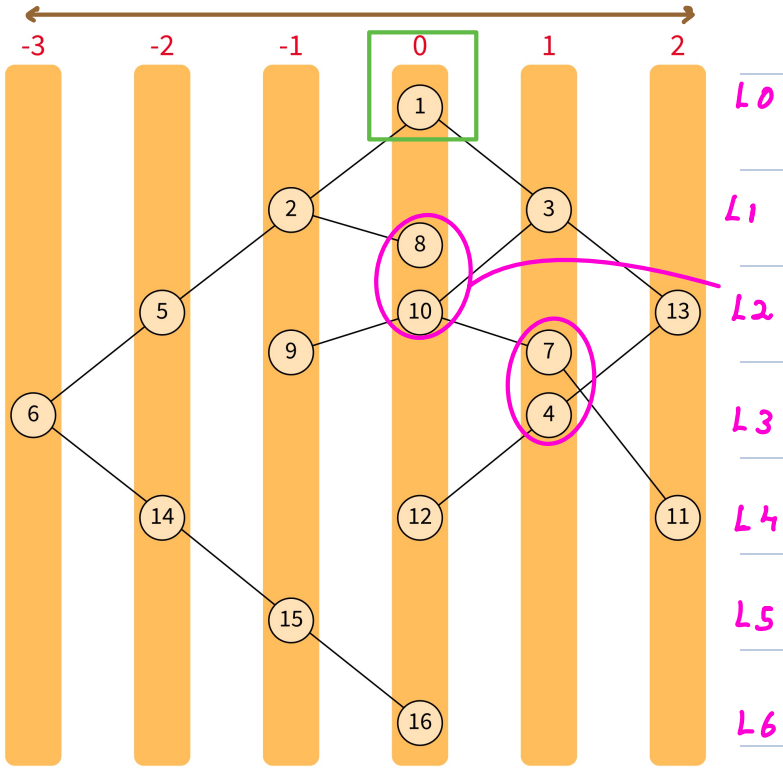
H.W ✓

Vertical Order Traversal of Binary Tree



Print each vertical line top to bottom.

If there is overlap print the node coming from left side first.



o/p → 6

5 14

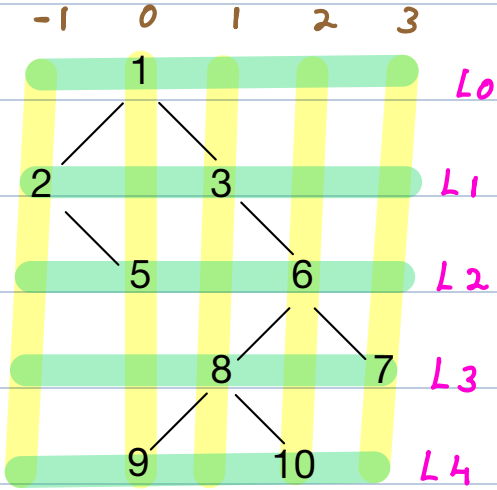
2 9 15

1 8 10 12 16

3 7 4

13 11

Quiz :



o/p → 2

1 5 9

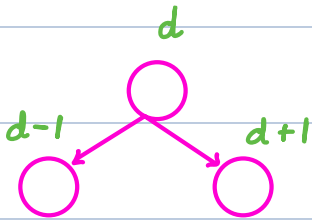
3 8

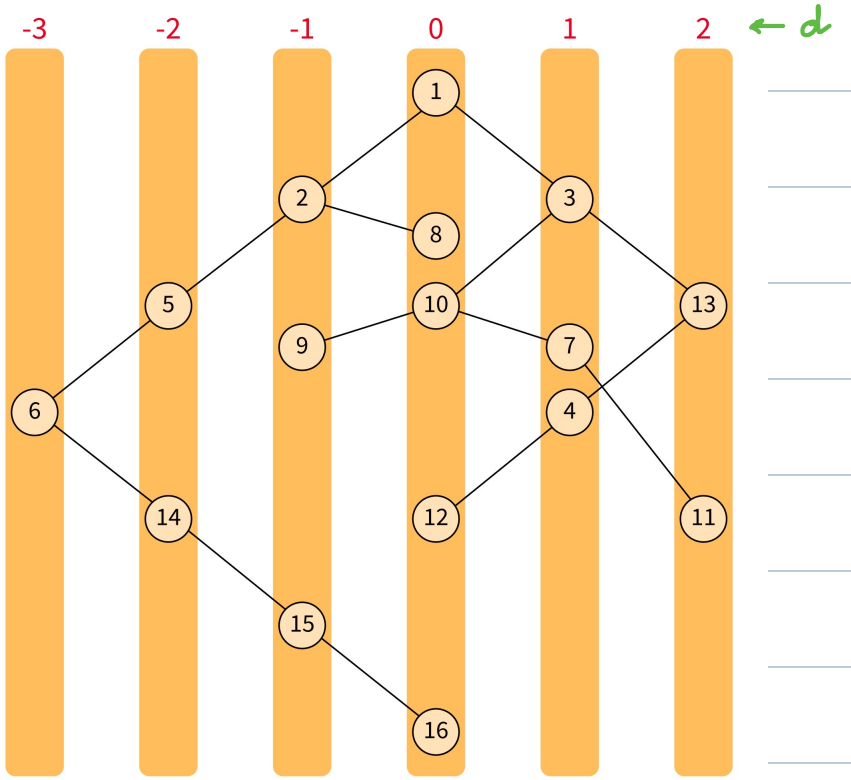
6 10

7

Print top to bottom

⇒ Level order Traversal





∀ nodes, store distance

⇒ Pair < Node, d > /

∀ i, d[i] → distance of 'i'. ✓

∀ level, store list of nodes ⇒ HashMap < Level, list of Nodes >

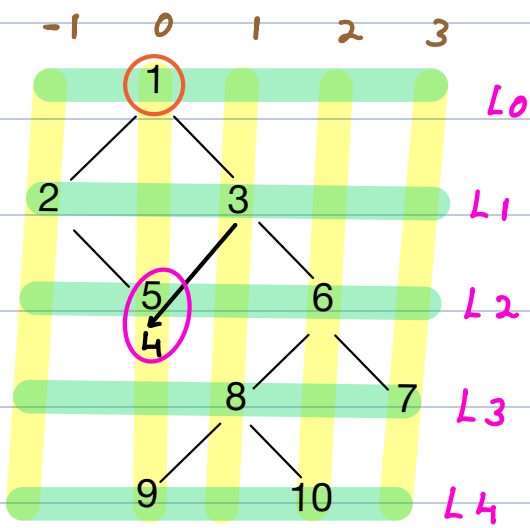
Queue

~~1~~ ~~2~~ ~~3~~ ~~5~~ ~~4~~ ~~6~~ ~~8~~ ~~7~~ ~~9~~ ~~10~~

d =

1	2	3	4	5	6	7	8	9	10
0	-1	1	0	0	2	3	1	0	2

Nodes → 1 to N



HashMap < dist, List of Nodes >

0 → {1, 5, 4, 9}

-1 → {2}

1 → {3, 8}

2 → {6, 10}

3 → {7}

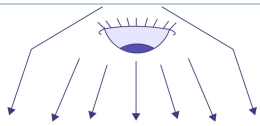
min_d → min horizontal distance // -1

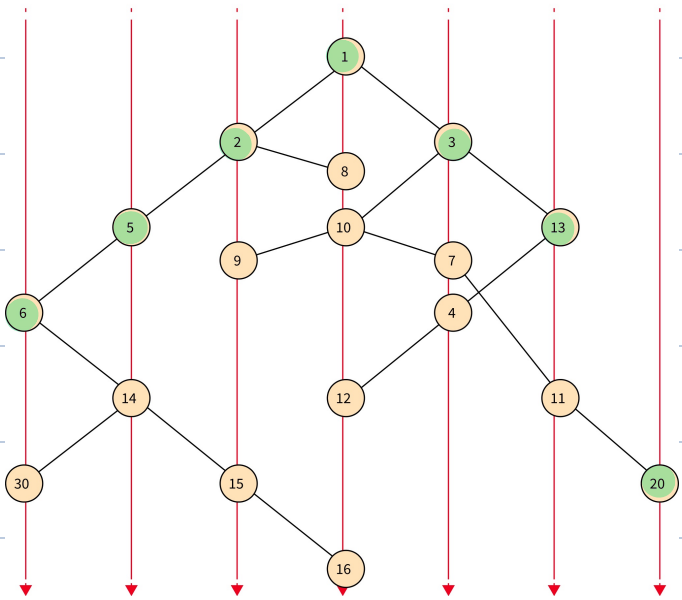
max_d → max horizontal distance // 3

```
for d → min_d to max_d {
    for x in hm.get(d) {  // for each loop
        print(x)
    }
    print("\n")
}
```

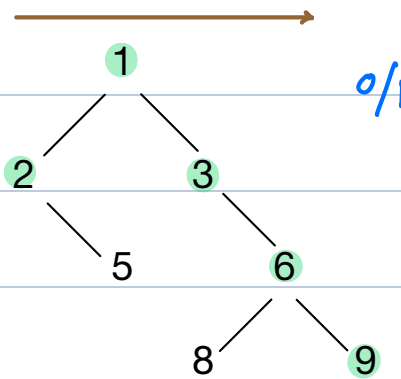
TC = O(N) SC = O(N+N+N) = O(N)

< *Question* > : Find top view of binary tree.





Quiz :



o/p → 2 1 3 6 9

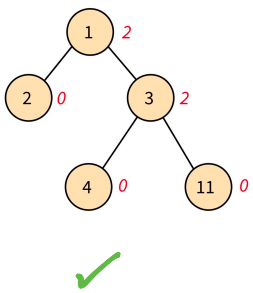
Sol → Print first element of every vertical line ⇒ first node ∀ list in hm. ✓

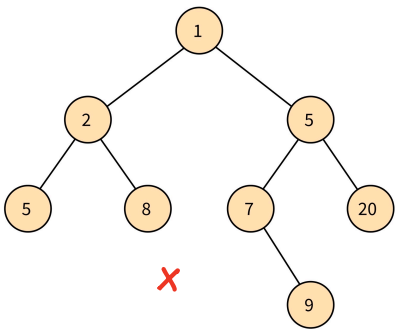
H.W → Bottom view .

Types of Binary Tree [Structure]

1. Proper/ Full Binary Tree

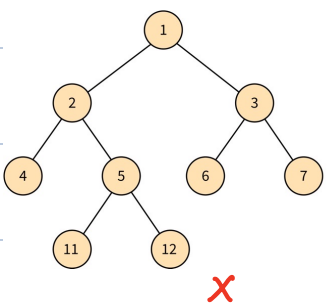
Every node has either 0 or 2 children.

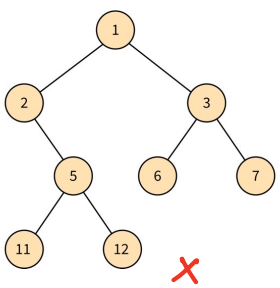


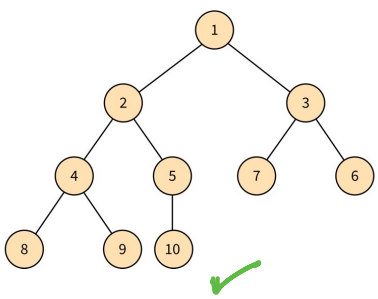


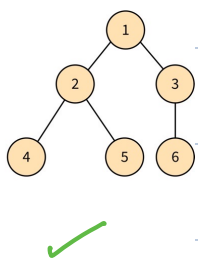
2. Complete Binary Tree (C.B.T)

All levels must be completely filled except possibly the last level and the last level must be filled from left to right.



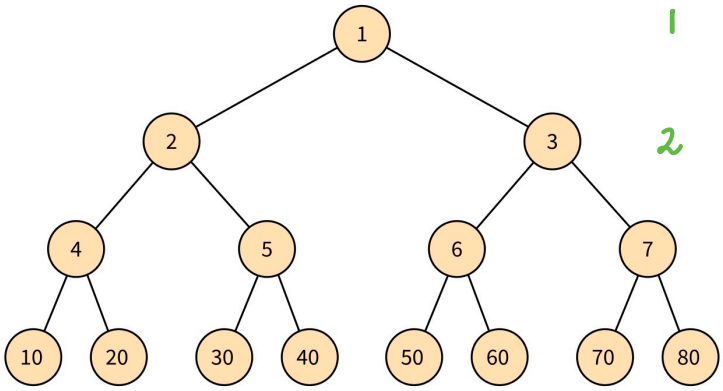






3. Perfect Binary Tree

All levels are completely filled.



$1 \quad 2^0$

$2 \quad 2^1$

$4 \quad 2^2$

$8 \quad 2^3$

$\vdots$

2^H

$N = 15$

$H = 3$

Q → Given a perfect binary tree with N nodes, find height of the tree.

$$1 + 2 + 4 + \ldots + 2^H = N$$

$$= \frac{1(2^{H+1} - 1)}{2 - 1} = 2^{H+1} - 1 = N$$

$$\Rightarrow 2^{H+1} = N + 1$$

$$\Rightarrow H + 1 = \log_2 (N+1)$$

$$\Rightarrow \boxed{H = \log_2 (N+1) - 1} \rightarrow O(\log(N))$$

Balanced Binary Tree

For all nodes

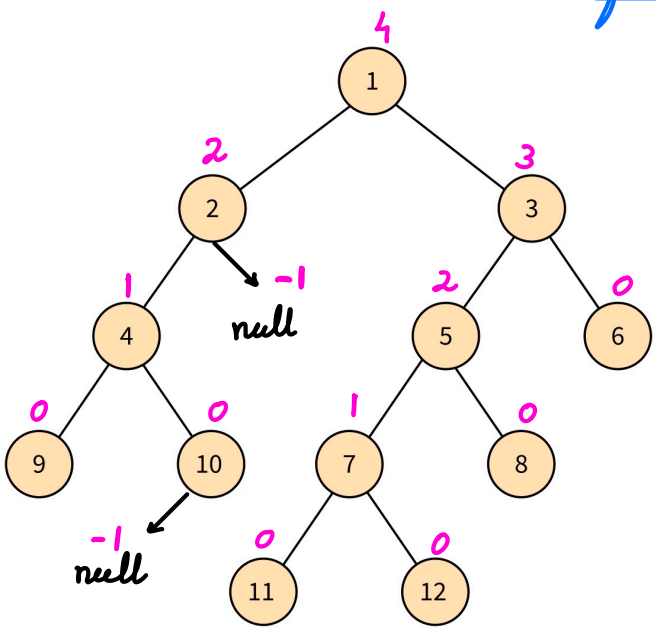
ht. of left child − ht. of right child	≤ 1

$|1 - (-1)| = 2 \neq 1$

< Question > : Given a binary tree, check if it is balanced or not.

Ans = false

Height → Distance of node to farthest leaf.



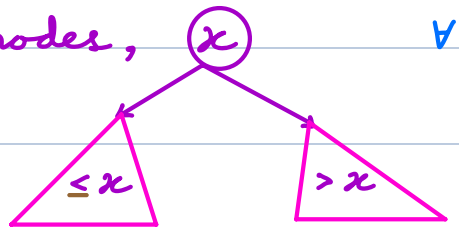
Left Right Node

isB = true

```
int height(root) {
    if (root == null) return -1
    L = height(root.left)
    R = height(root.right)
    if (abs(L - R) > 1)   isB = false
    return max(L, R) + 1
}
```

TC = O(N) SC = O(N)

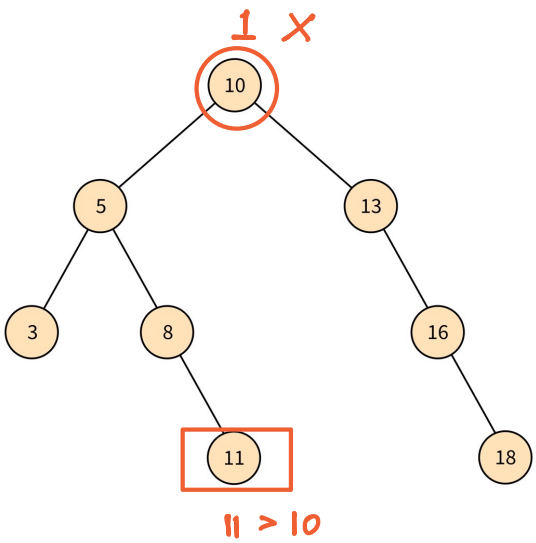
Binary Search Tree [BST]

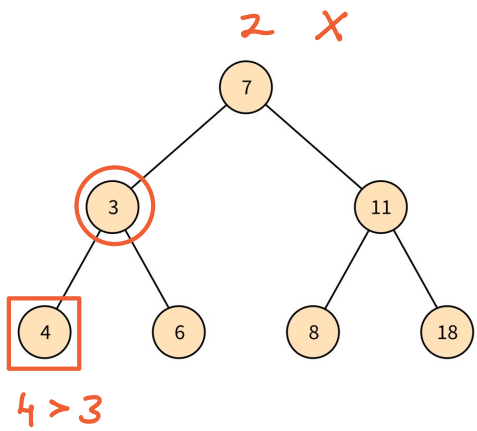


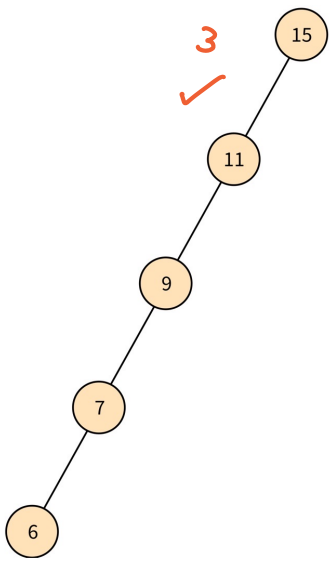
∀ nodes x,

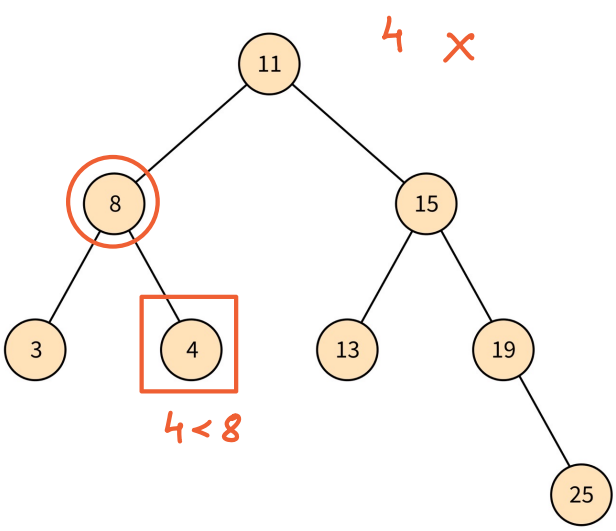
left subtree data <= x

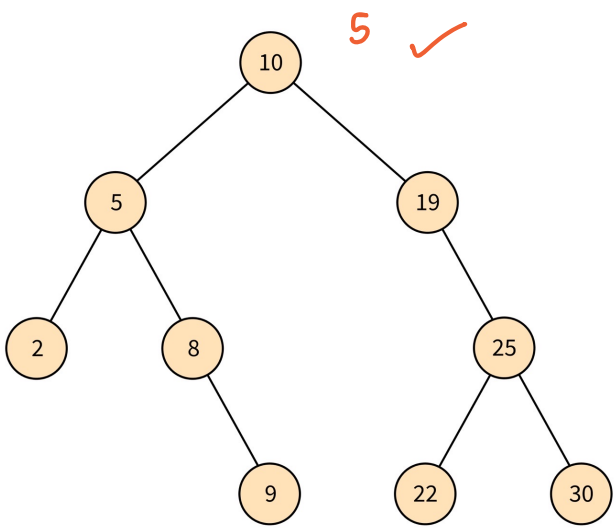
right subtree data > x











Scenario

Flipkart stores and manages the history of all the orders that were processed and stores them efficiently such that whenever some information is required regarding any order, it can be fetched easily.

Problem

Develop a system for Flipkart that efficiently stores and manages the history of all processed orders. The system should allow easy access to information regarding any specific order when needed.

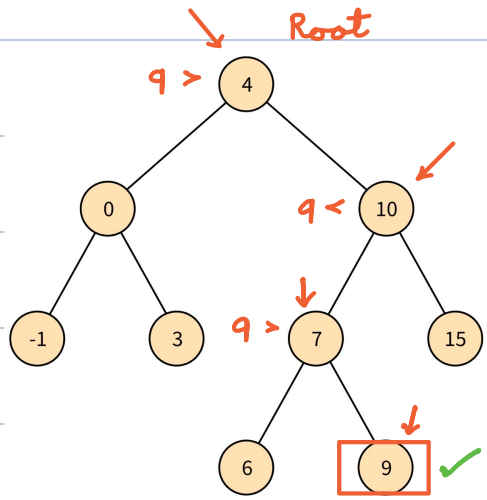
Requirements

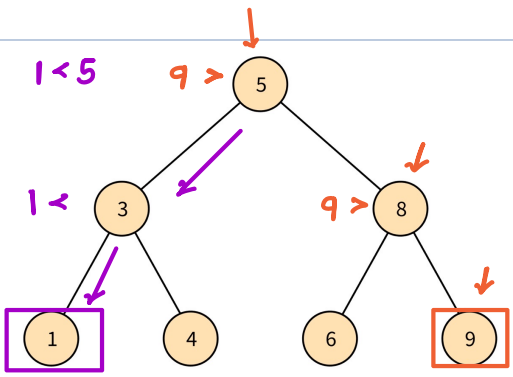
There are two types of requirements :

- **Add Order :** Insert a new record.
- **Find Order Time :** The system should be able to quickly retrieve the time of placing the order, given the unique order ID.

< Question > : Search an element K in Binary Search Tree.

K = 9





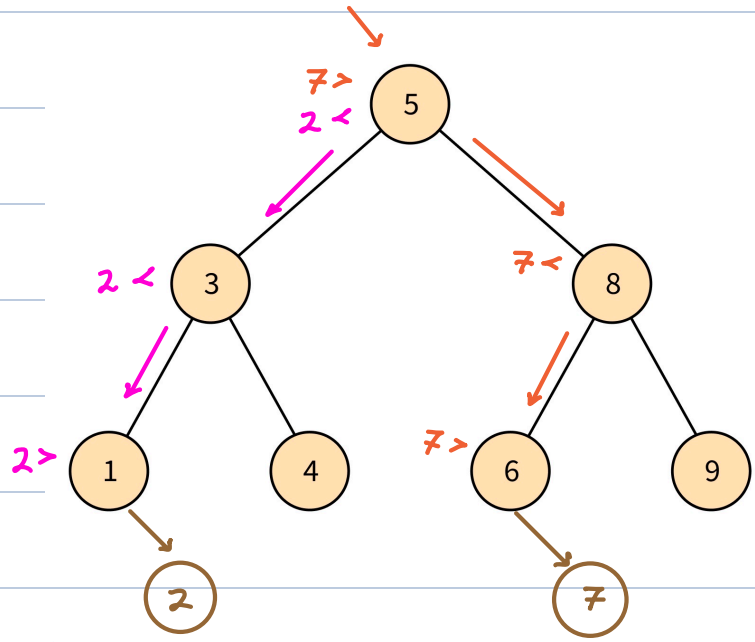

```
temp = root
while (temp != null) {
    if (temp.val == K)  return temp
    if (temp.val > K)   temp = temp.left
    else                temp = temp.right
}

return null
```

TC = O(H) SC = O(1)

Insertion in B.S.T

insert (7)

insert (2)



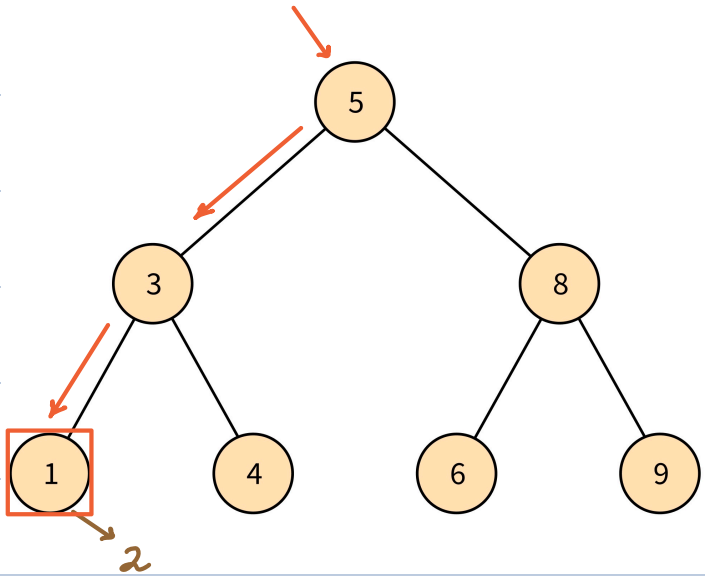
insert (7)

```
temp = root
while (temp != null) {
    if (K <= temp) {
        if (temp.left == null) {
            temp.left = new TreeNode(K)
            break
        } temp = temp.left
    } else {
        if (temp.right == null) {
            temp.right = new TreeNode(K)
            break
        } temp = temp.right
    }
}
```

TC = O(H) SC = O(1)

Find the Smallest Element in B.S.T

```
temp = root
while (temp.left != null) {
    temp = temp.left
}
return temp.
```



TC = O(H)

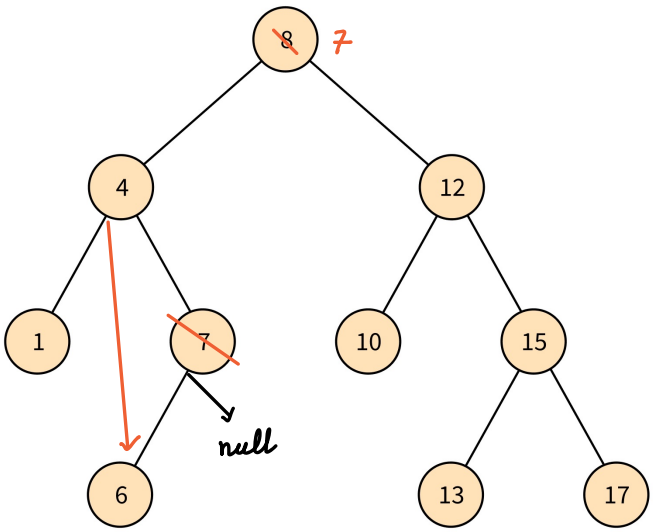
SC = O(1)

< *Question* > : How to find the largest element in Binary Search Tree.

```
temp = root
while (temp.right != null) {
    temp = temp.right
}
return temp
```

TC = O(H) SC = O(1)

Deletion in B.S.T



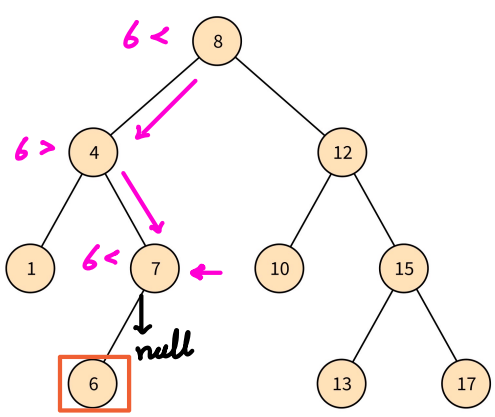
delete (6) ✓

delete (7) ✓

delete (8) ✓

1. Leaf node ✓
2. Node with single child ✓
3. Node with both the children ✓

Leaf Node



1) Travel till parent of K.

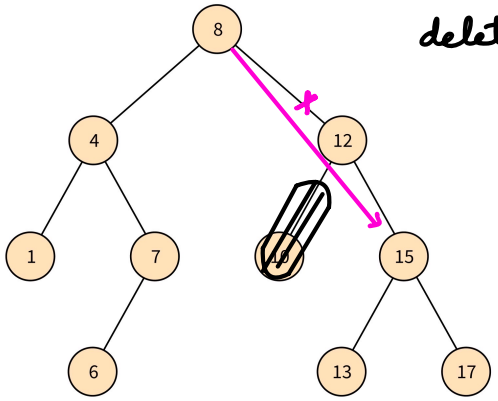
2) Update the child to null.

delete (6) ✓

~~delete (7)~~

~~delete (8)~~

Node with single child



delete (12)

delete (6)

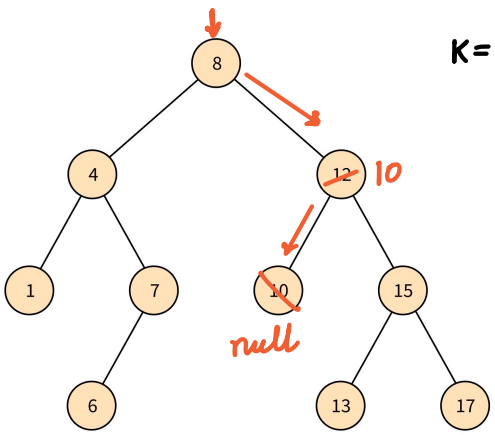
delete (7)

delete (8)

1) Travel till parent of K.

2) Update link to K with link to child of K.

Node with both children



K = 12

~~delete (6)~~

~~delete (7)~~

~~delete (8)~~

1) Travel till node K.

2) Find largest node in left subtree & delete it.

3) Replace K with largest node in left subtree.

```
root = delete ( root , K)
```

```
                                    int
TreeNode  delete ( root , K ) {
    if ( root == null )  return null
    if ( K < root . data )
        root . left = delete ( root . left , K)  ✓
    else if ( K > root . data )
```

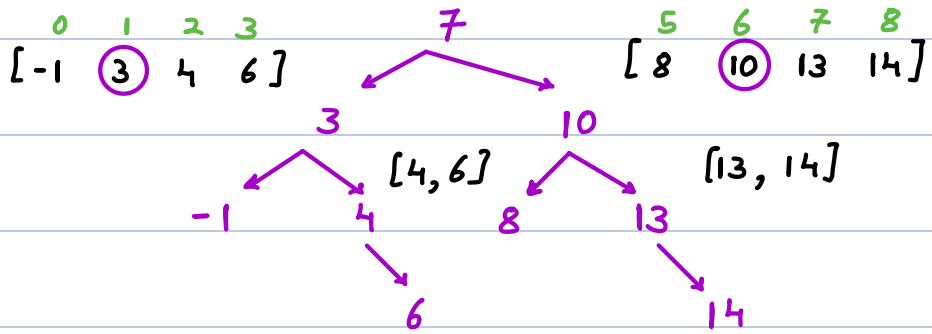
```
    root.right = delete(root.right, K) ✓
else {  // K == root.data
    if (root.left == null &&
        root.right == null)   return null
    if (root.left == null)   return root.right
    if (root.right == null)  return root.left

    temp = root.left
    while (temp.right != null)
        temp = temp.right
    }
    root.data = temp.data
    root.left = delete(root.left, temp.data)
    return root
}
```

TC = O(H) SC = O(H)

H.W → Read about Red-Black Tree & AVL Tree } Balanced BST.

Construct B.S.T from sorted array

arr → [-1 , 3 , 4 , 6 , 7 , 8 , 10 , 13 , 14]
0 1 2 3 4 5 6 7 8

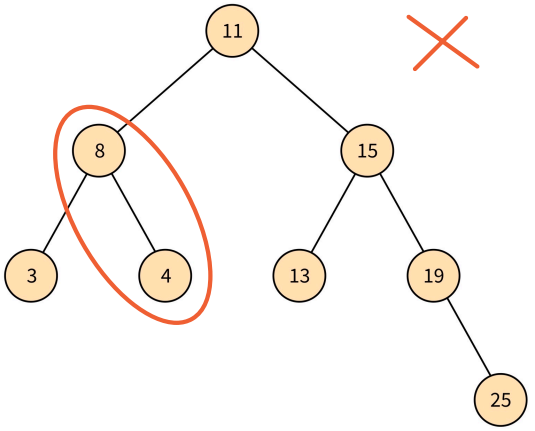


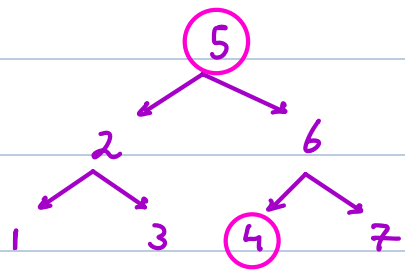

```
TreeNode build ( A[], L, R) {
    if ( L > R )  return null
    m = (L+R) / 2
    root = new TreeNode(A[m])
    root.left = build (A, L, m-1)
    root.right = build (A, m+1, R)
    return root
}
```

(L: 0, R: N-1)

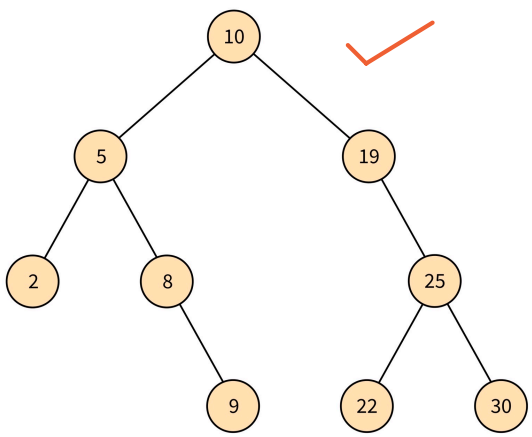
TC = O(N) SC = O(log (N))

Check if given Binary Tree is a B.S.T





Not a BST



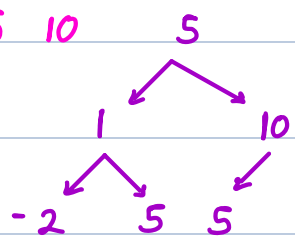
Sol 1 → Left Node Right

< >

Inorder traversal is sorted.

Unique data ✓

→ in = -2 1 5 5 5 10



Not a BST

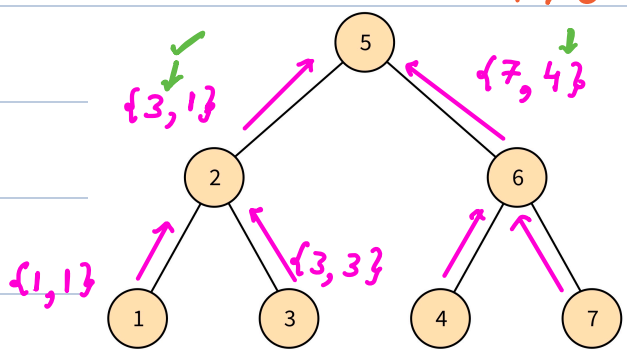
4 ≯ 5

Sol 2 →

∀ nodes x,

left subtree data <= x

right subtree data > x



largest in left subtree <= x {max, min}

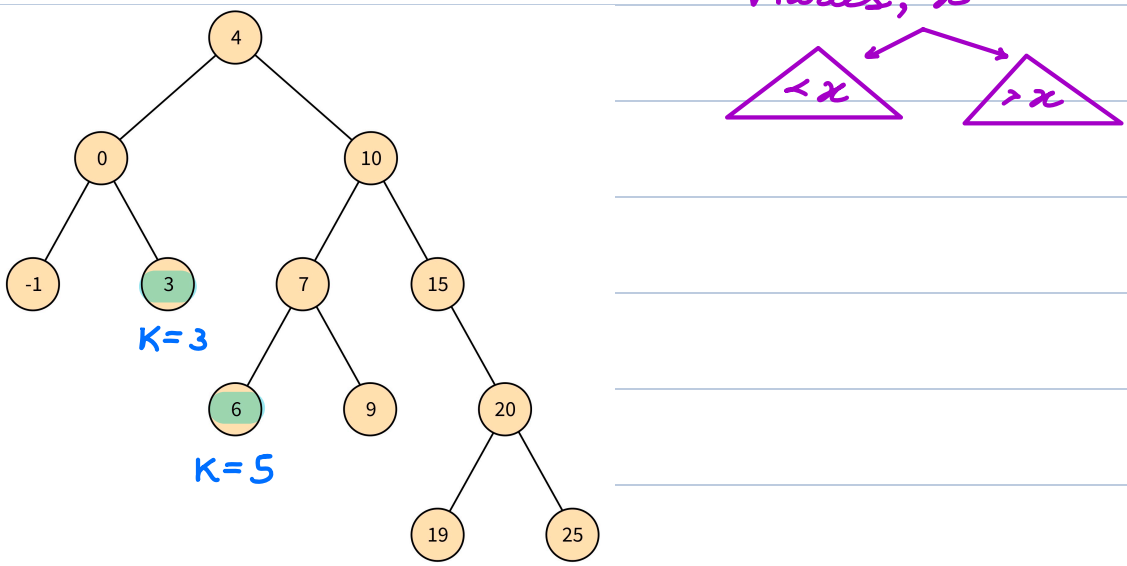
& smallest in right subtree > x ✓

SCALER

Trees - 4

TABLE OF CONTENTS

< Question > : Find Kth smallest element in B.S.T K = 3



Sol → Inorder → Left Node Right

K^{th} element in inorder traversal.

TC = O(N) SC = O(H)

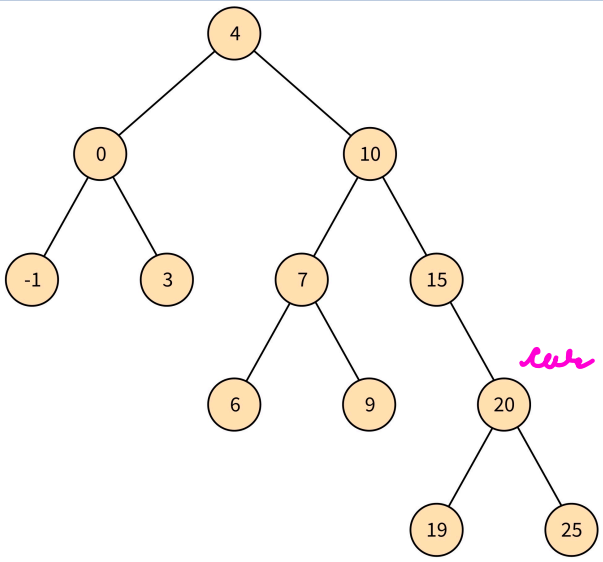
O(1)

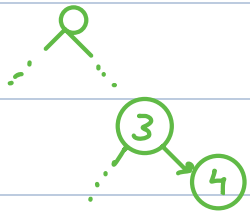
Morris Traversal

Left Node Right

In-Order Traversal of a Binary Tree

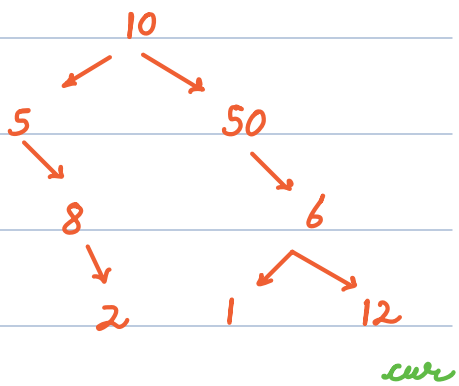
Expected S.C → O(1)





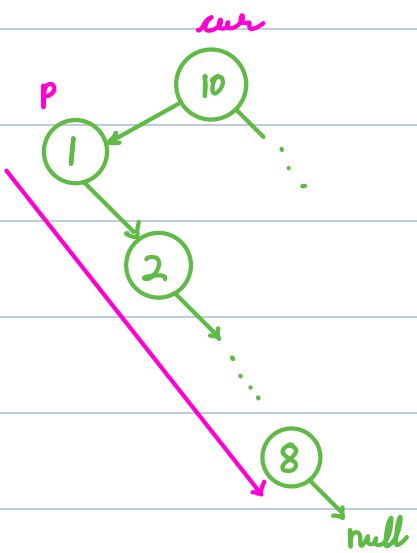
o/p → -1 0 3 4 6 7 9 10 15 . . .

```
cur = root
while ( cur != null ) {
    if ( cur.left == null ) {
        print (cur.val)
        cur = cur.right
    } else {
        p = cur.left
        while ( p.right != null && p.right != cur )
            p = p.right
        if ( p.right == null ) {    // First time
            p.right = cur
```



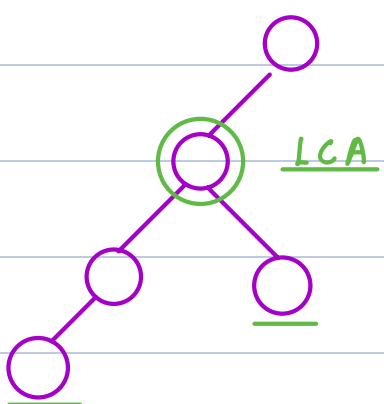
o/p → 5 8 2 10 50 1 6 12

```
        cur = cur.left
    } else {            // second time
        print(cur.val)
        cur = cur.right
        p.right = null
    }
}
```

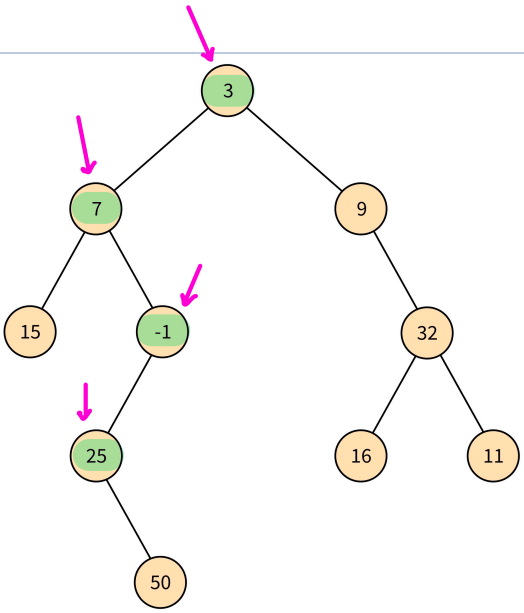


If cur != 10, do we visit 1 to 8 while finding inorder predecessor? → No

TC = O(N)



Path from Root to Node



K = 25

```
void travel (root, found) {
    if (root == null || found)
        return
    a.add(root)
    if (root.val == K) {
        found = true
        return
    }
    travel(root.left, found)
    travel(root.right, found)
    a.removeLast()
}
```

false

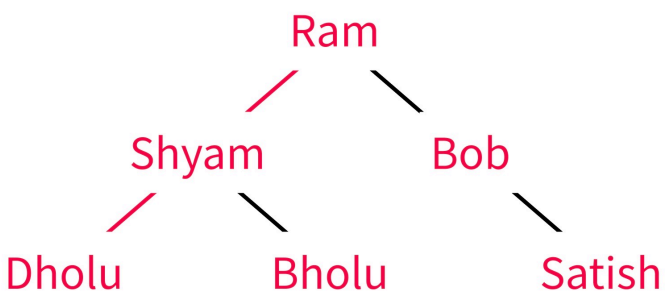
TC = O(N)

SC = O(H)

Problem Statement

It is said that we all humans are related through some common ancestor at some point of time. Assume that a person can have 0, 1 or 2 children only.

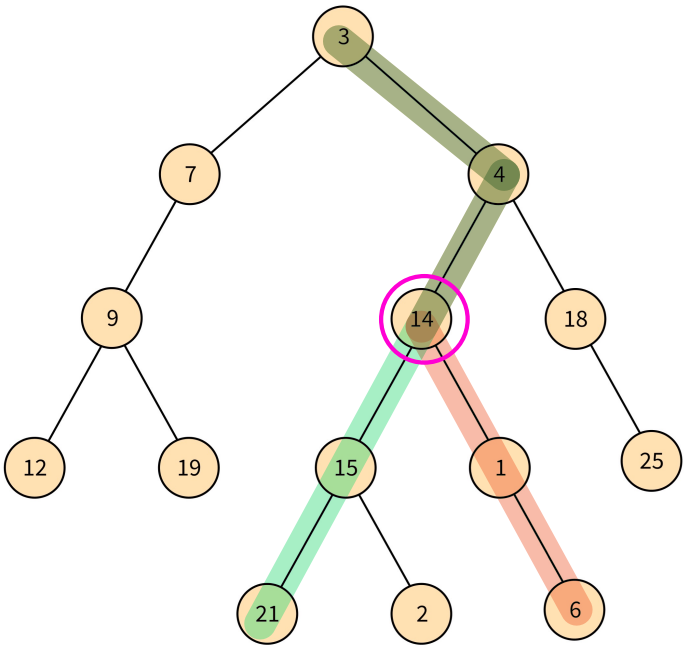
Given the Binary tree A representing the family tree, discover the earliest common family member who connects two given people B and C in a family tree.



- Find LCA of Dholu and Bob
- Answer = Ram

Lowest Common Ancestor

Flipkart | Acollite | Amazon | Microsoft | OYO Rooms | Snapdeal | MakeMyTrip | PayU | Times Internet | Cisco | PayPal | Expedia | Twitter | American Express



L.C.A(21 , 6) → 21 → 3 4 14 15 21
6 → 3 4 14 1 6

L.C.A(2 , 6) → 14

L.C.A(21 , 4) → 21 → 3 4 14 15 21
4 → 3 4

L.C.A(12 , 6) → 3

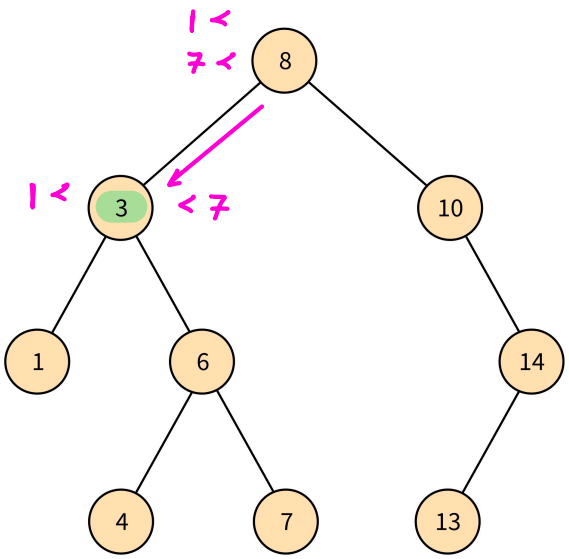
LCA (x, y)

Sol → 1) Find root to node path for x & y. ✓

2) Ans = last common node.

TC = O(N) SC = O(H)

L.C.A in B.S.T



Above sol → TC = O(H) SC = O(H)

O(1)

LCA (1, 7) = 3

```
cur = root
while (cur != null) {
    if (x < cur.val && y < cur.val)
        cur = cur.left
    else if (x > cur.val && y > cur.val)
        cur = cur.right
    else
        return cur
}
```

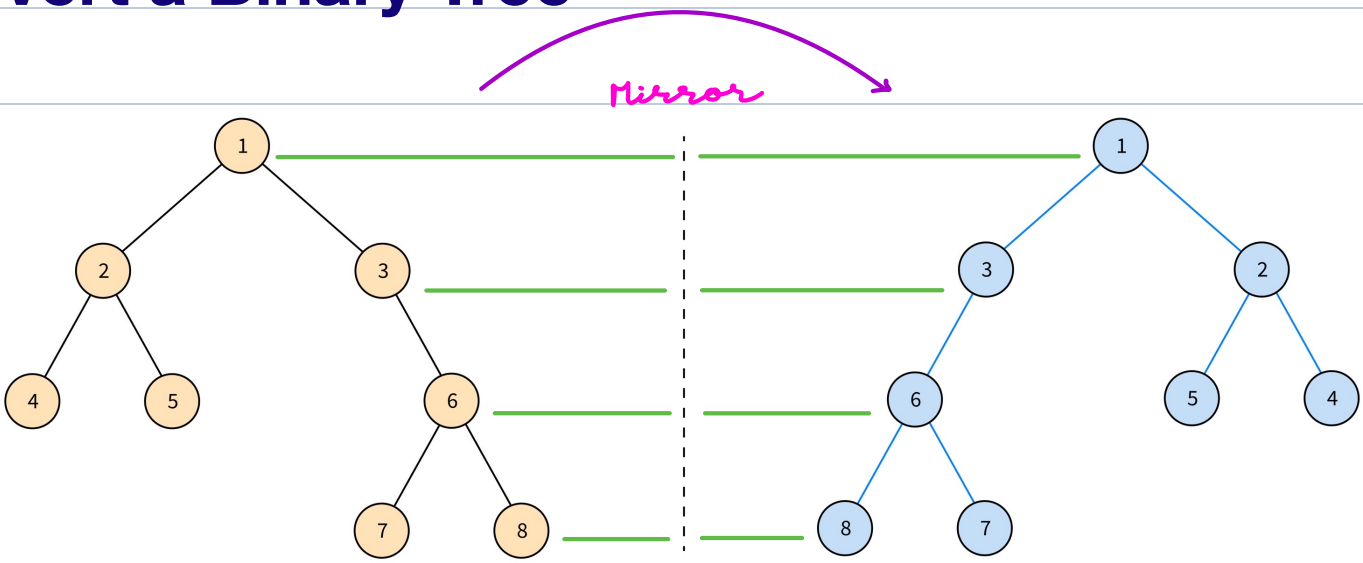
TC = O(H) SC = O(1)

SCALER

Trees - 5

TABLE OF CONTENTS

Invert a Binary Tree



Is the below inversion correct for given binary tree ?

```
Given
     4
    / \
   2   7
  / \   \
 1   3   9

Inverted
     4
    / \
   7   2
  \   / \
   3 1   9
```

No

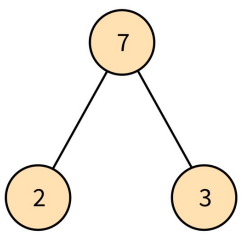
```
void invert (root) {
    if (root == null)
        return
    temp = root.left
    root.left = root.right
    root.right = temp
    invert (root.left)      // Left
    invert (root.right)     // Right
}
```

Node

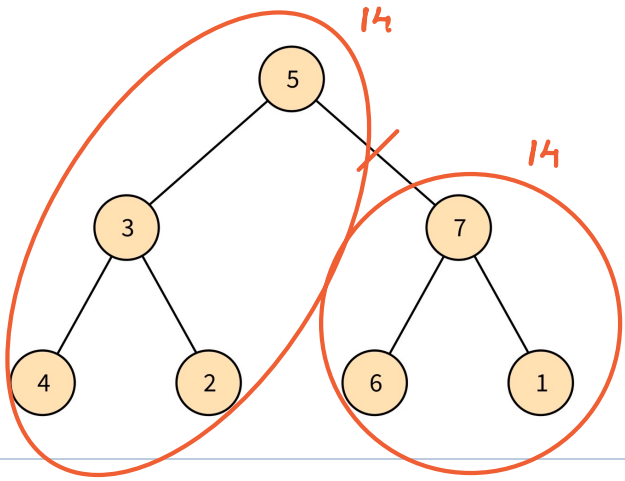
TC = O(N) SC = O(H)

Equal tree Partition

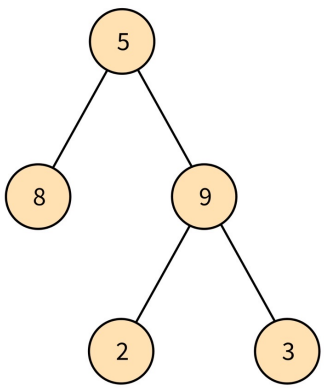
- Check if it is possible to remove an edge from the given binary tree such that sum of resultant two trees is equal. ← half of total.



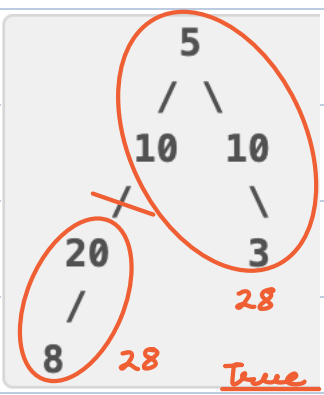
False



True



False




```
sum = getSum(root)
if (sum % 2 == 1) return false
ans = false
checkSum(root, sum/2)
return ans
```

```
int getSum(root) {
    if (root == null) return 0
    return root.val + getSum(root.left) +
                getSum(root.right)
}
```

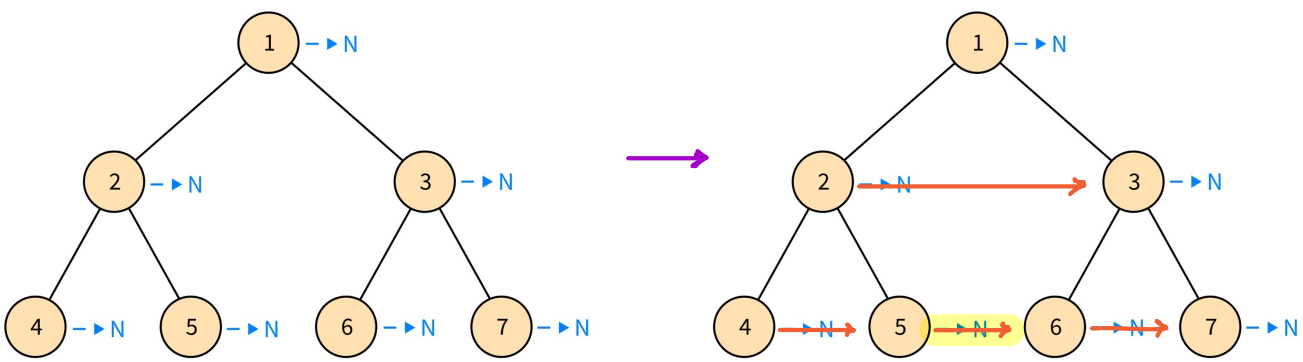
```
int checkSum(root, half_Sum) {
    if (root == null) return 0
    s = root.val + checkSum(root.left, half Sum) +
        checkSum(root.right, half Sum)
    if (s == half Sum) ans = true
    return s
}
```

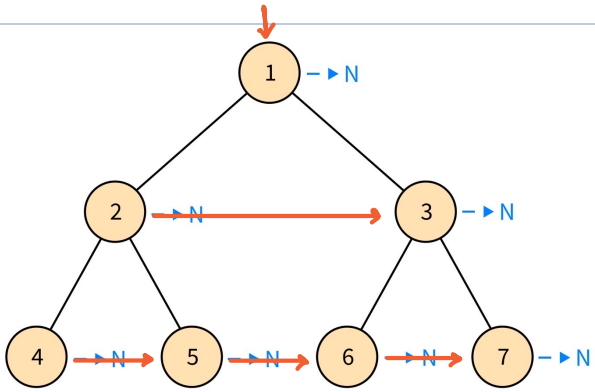
$TC = O(N + N) = O(N)$

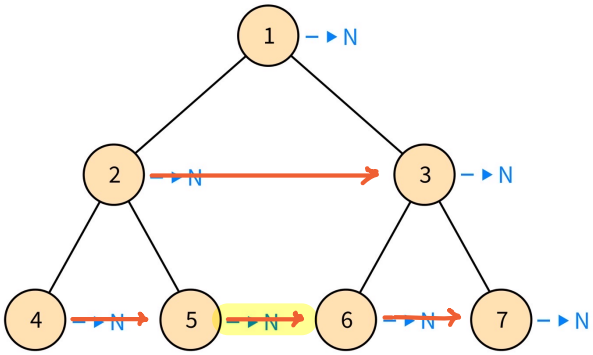
$SC = O(H)$

Next Pointer in Binary Tree

- Given a perfect tree initially with all next pointers set to nullptr, modify the tree in-place to connect each node's next pointer to the next node in the same level from left to right, following an in-order traversal.



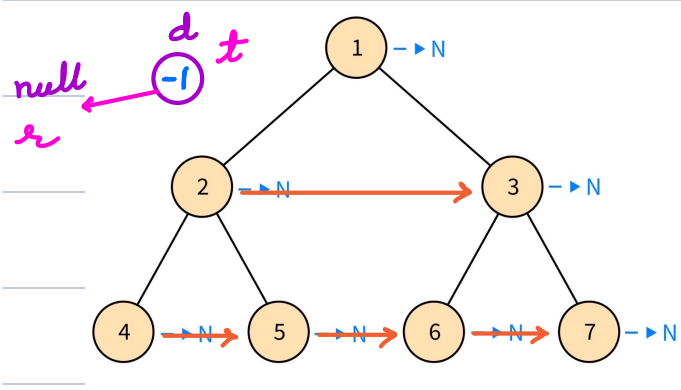


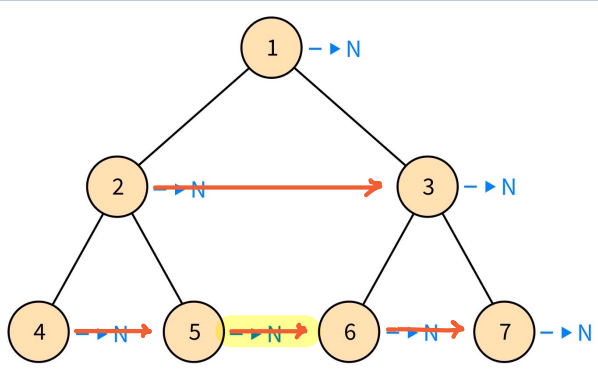


Level Order Traversal → Queue


```
if (root == null) return
// Queue → q
q.enqueue(root)
last = root
while (!q.isEmpty()) {
    cur = q.dequeue()
    if (cur.left != null) q.enqueue(cur.left)
    if (cur.right != null) q.enqueue(cur.right)
    if (cur != last) cur.next = q.front()
    else last = q.rear()
}
```

TC = O(N) SC = O(N)






```
if (root == null) return
dummy = new TreeNode (-1)
temp = dummy
while (root != null) {
    if (root.left != null) {
        temp.next = root.left
        temp = temp.next
    } if (root.right != null) {
        temp.next = root.right
        temp = temp.next
    } root = root.next
if (root == null) {
    root = dummy.next
    dummy.next = null
    temp = dummy
}
}
```

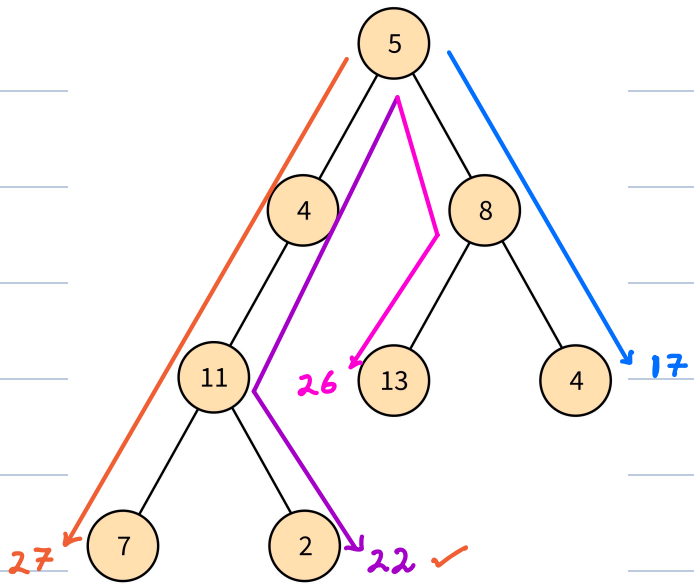
TC = O(N)

SC = O(1)

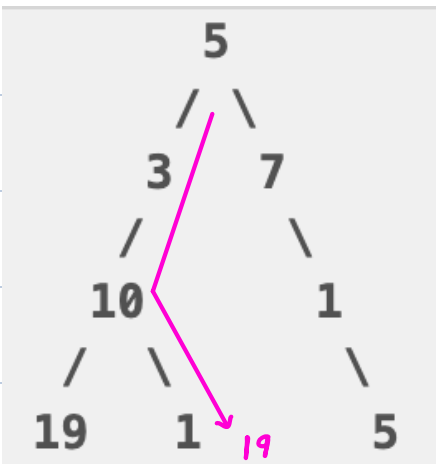
< Question > : Check if root to leaf path sum equals to K.

a. K = 22 → true

b. K = 19 → false




```
boolean check (root, K) {
    if (root == null) return false
    if (root.left == null &&
        root.right == null)
            return (root.val == K)
    return check (root.left, K-root.val) ||
           check (root.right, K-root.val)
}
```

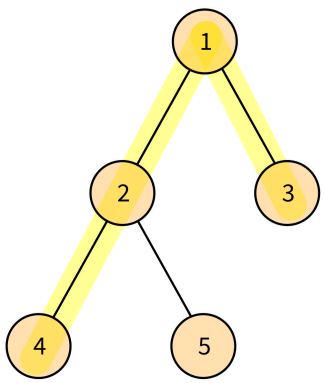


TC = O(N) SC = O(H)

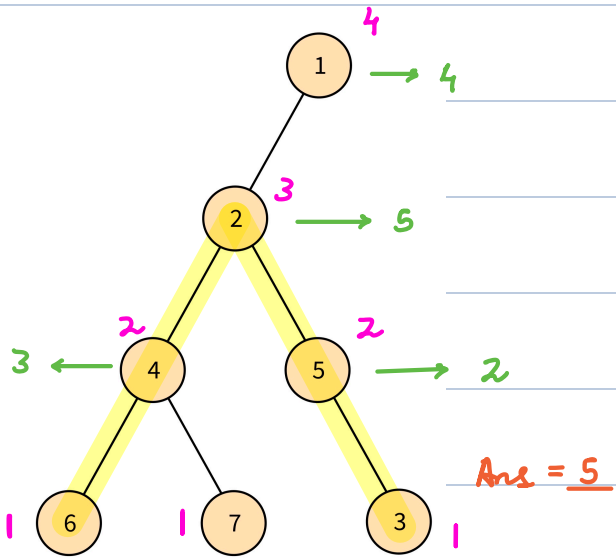
Diameter of Binary Tree

- **Definition of Diameter :**

The diameter of a binary tree is defined as the number of nodes along the longest path between any two leaf nodes in the tree. This path may or may not pass through the root.

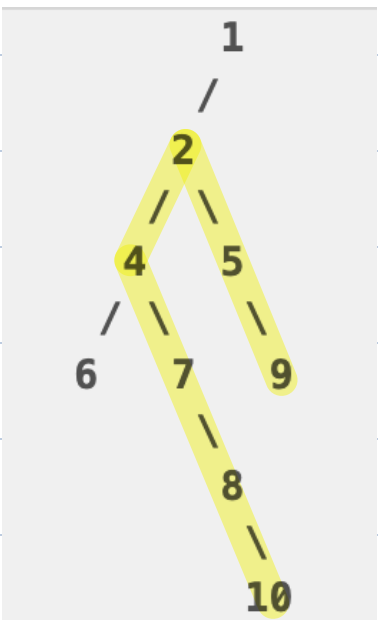


Ans = 4



Ans = 5

Hint → use height of tree. height(leaf) = 1



height(node) = max(left, right) + 1

diameter(node) = height(left) + height(right) + 1

Ans = 7