Satya Sai Siva Rama krishna

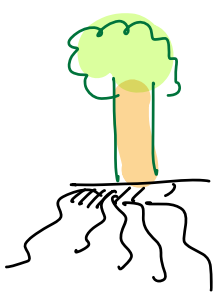
2+ years Teaching in DSA, 1 1/2 year Scaler

Today's Content: ⟶ { Pseudo codes
↳ Convert language of Choice }

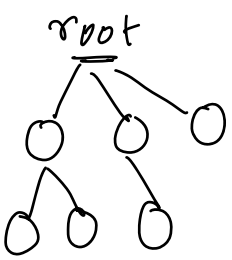
1) Trees Introduction
2) Binary Trees
3) Tree Traversals
4) finding path from root → src
5) least Common Ancestor
6) BST → [if time permits]

→ Scaler Student : Recording
→ Not a : Notes
→ { Not able to listen : Reload }

Quizes → { During Sessions , Leaderboard → { At the end }

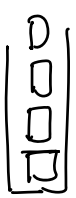


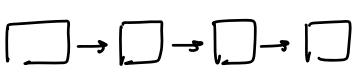
⟶ CS



→ Simple data Structures : ⟶ linear

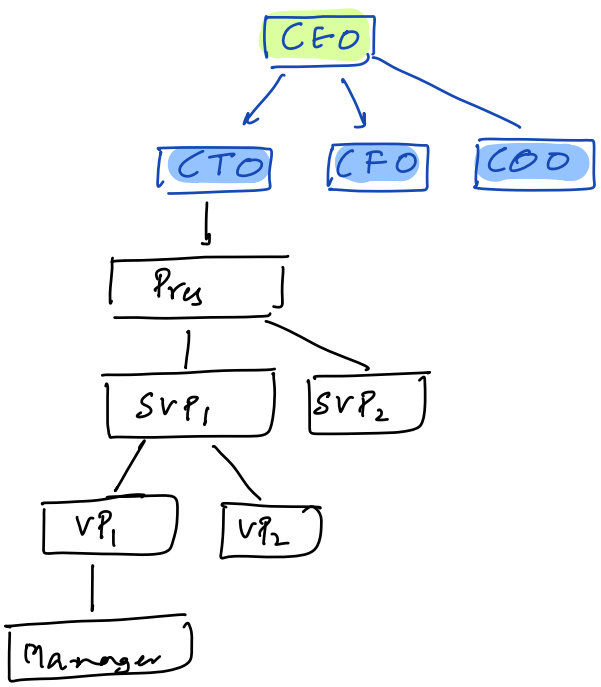
1) Array
2) Stacks
3) LinkedList
4) Queues



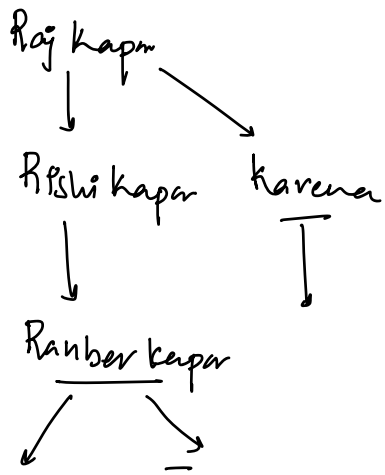


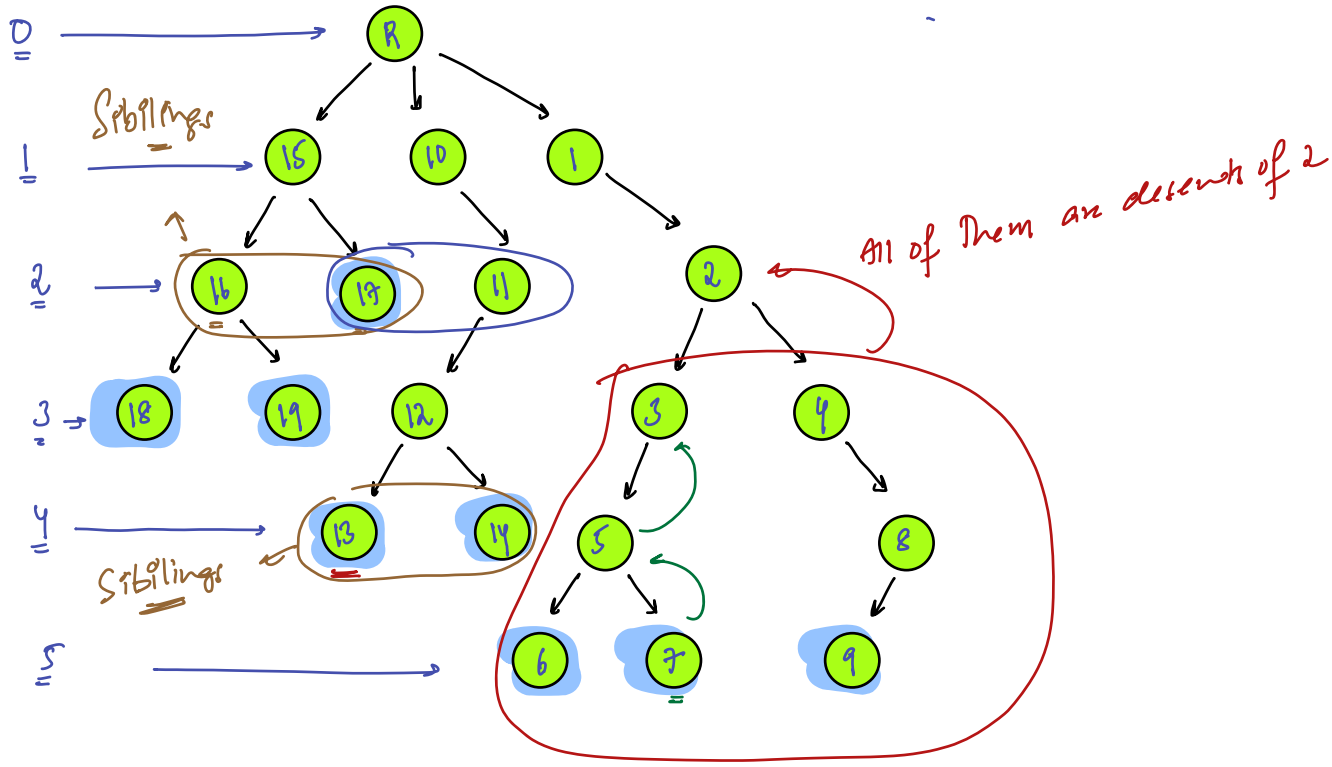
Hierarchical Data

Eg: Company organisation



Eg:





Parent: 2 is parent of 3 & 4 Child 11 is child of 10

Root Node: → R, node without parent

Leaf Node: → Node without any child Node are called leaf Node

Ancestr Node: { Ancestr → 7: 5, 3 2, 1, R }

{ Ancestr → 12, 11, 10, R }

Descendent Node:

Sibling: → Same parent Sibling nodes

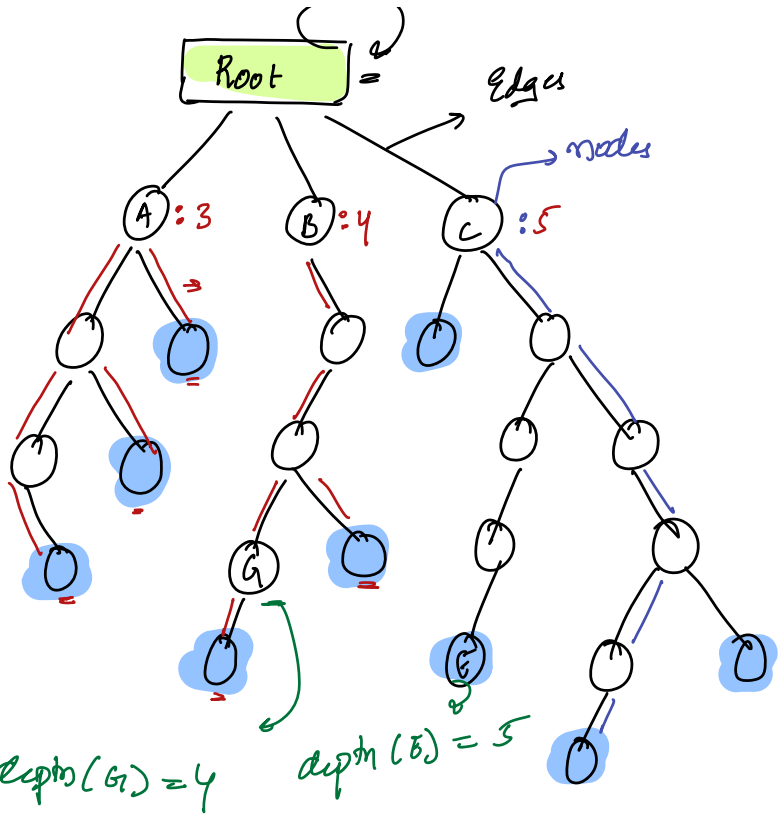
Same level Nodes:

// Node can have 1 parent

Height (Node):

: Length of (larger path) from Node to any of it's descendant leaf Nodes

: No: of Edges



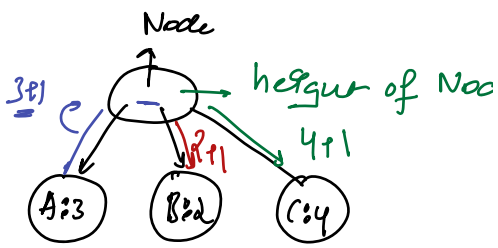
Height (A): 3

Height (B) : 4

Height (C) : 5

Height (root) = 6

Height (Node) = 1 + max height of (child Nodes)



Height (leaf Node) = 0

Depth (Node) = length of path from root Node → given Node

Depth (G) = 4

Depth (E) = 5

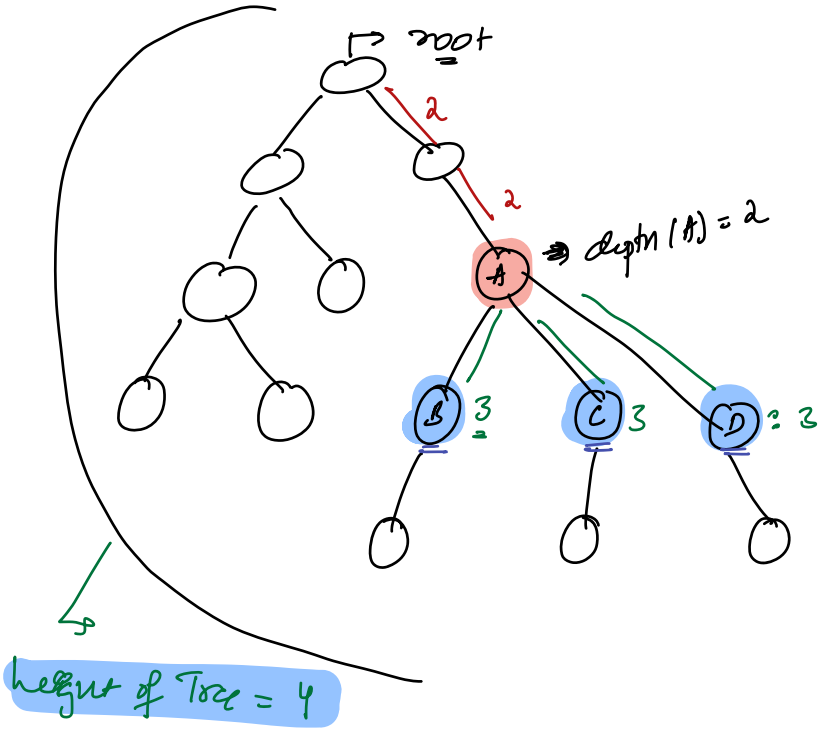
depth (Root Node) = 0

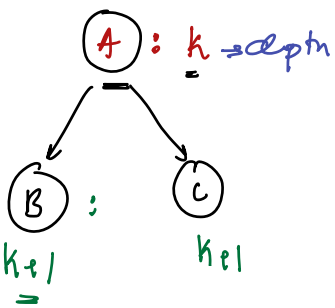
//

If

depth of Node = k

depth of child Node = —

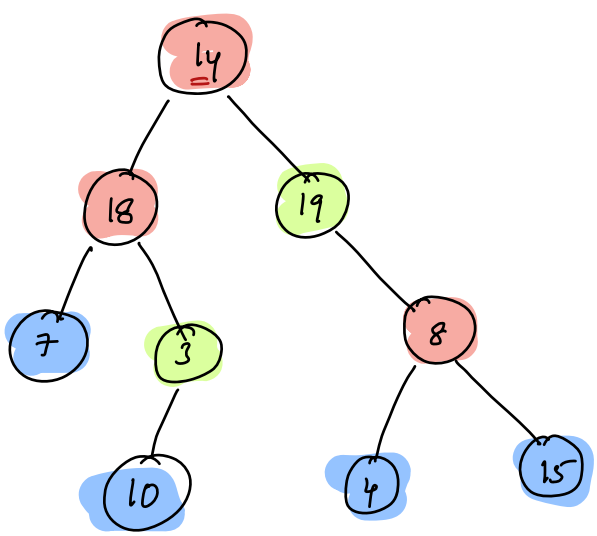




//

Binary Trees : → Every Node can atmost have 2 child Nodes

0 | 1 | 2 |



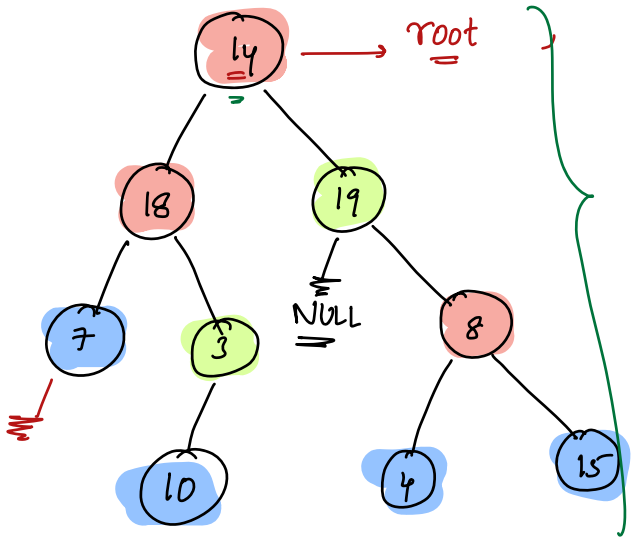
→ no child / leaf Node

→ 1 child

→ 2 child nodes

// Structure of BT:

```
class Node {
    int data;      -> data  .data, .left, .right
    Node left;     } -> Both child Node
    Node right;
};
```



14.left = 18
18.left = 7
7.left = NULL / indicate no child
19.right = 8
3.right = NULL

-> Height of Tree = 3

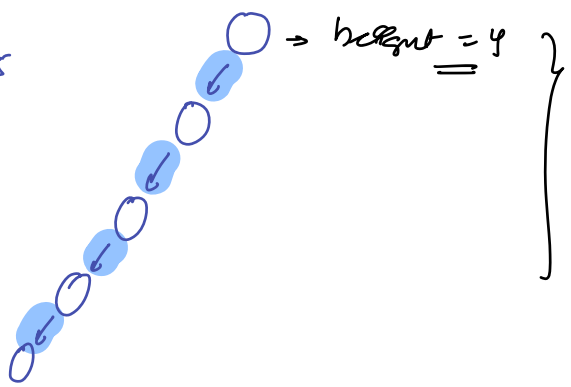
// Height of 14 : 3

// Height of Tree = Height of Root Node

// given N Nodes at max height of Tree: ?

All N Nodes are along 1 direction
i.e. N-1

N = 5



Problems:

Tree Traversals → {BT}

1) preorder : D L R

2) Inorder : L D R

3) postorder : L R D

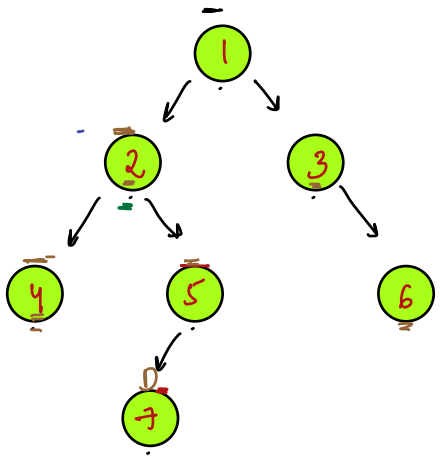
other versions

D R L

R D L

R L D

↳ No name as such

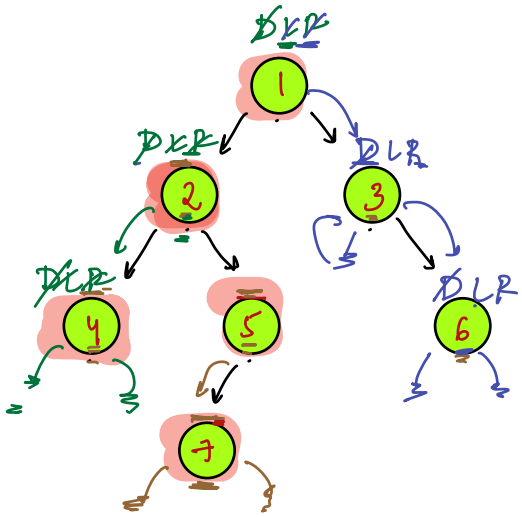


Inorder:

4 2 7 5 1 3 6

preorder: { D L R }

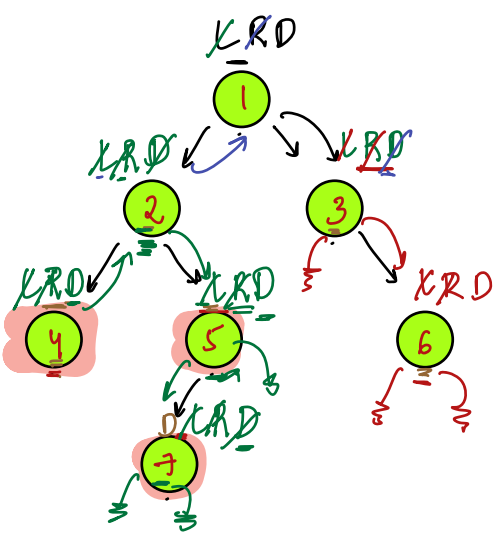
1 2 4 5 7 3 6



// postorder (L R D)

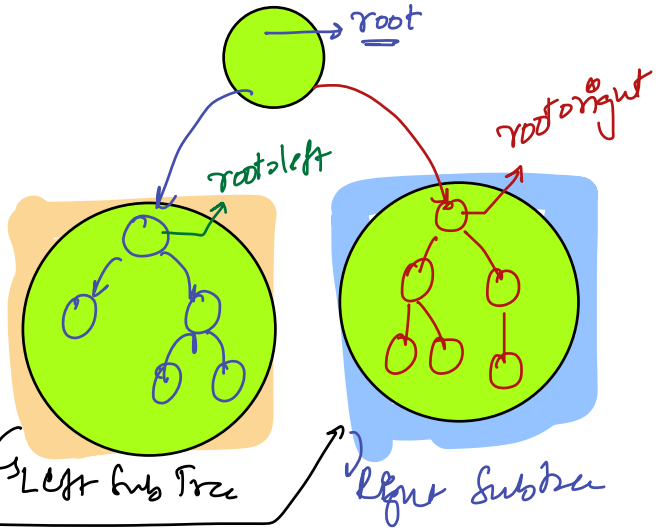
4 7 5 2 6 3 1

// preorder & postorder are not reverse of each other.



// Inorder

```
void Inorder(Node root) {
    if(root==NULL) {return;}
    Inorder(root.left)
    print(root.data)
    Inorder(root.right)
}
```



TC: O(N) — we are printing all nodes

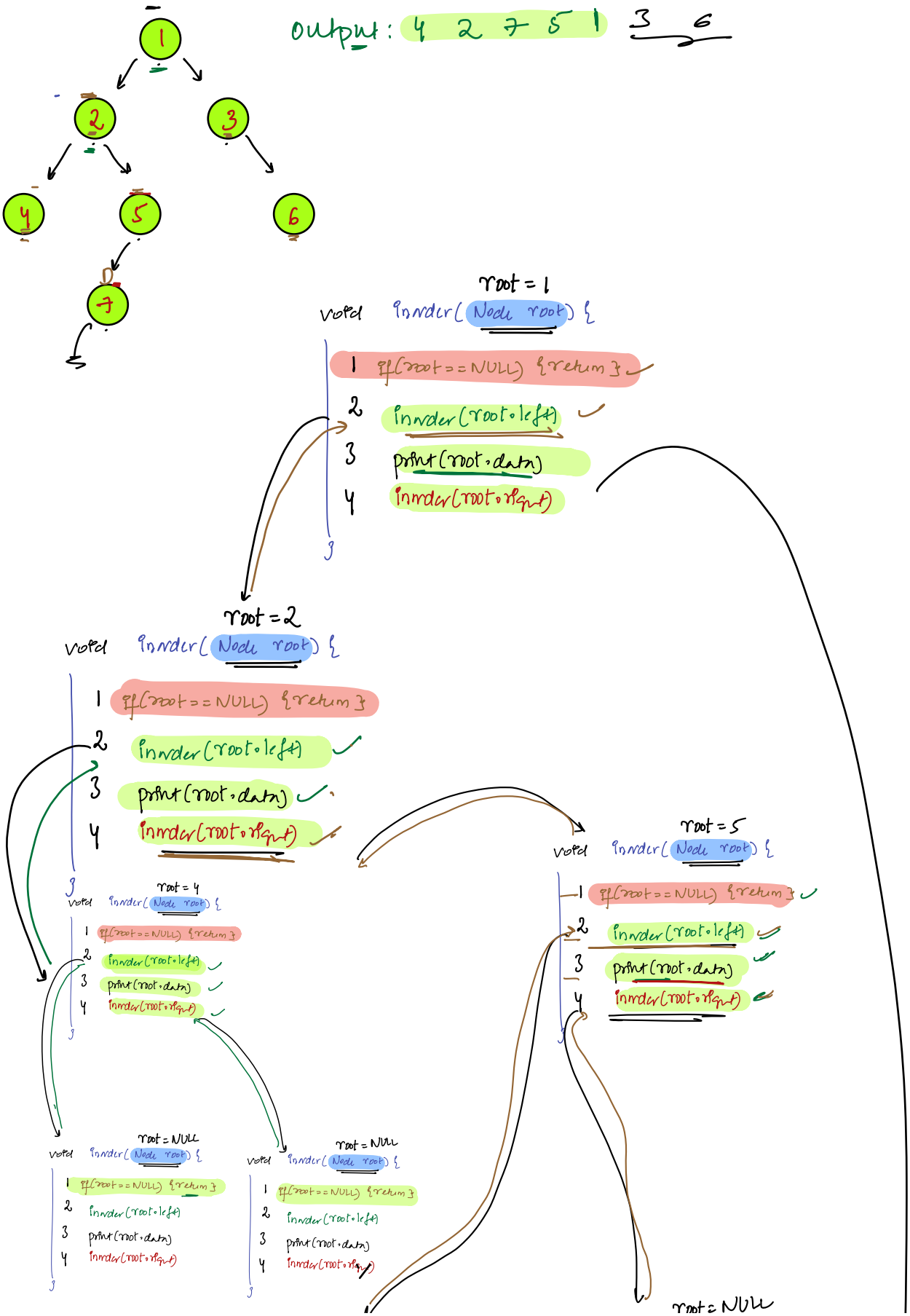
SC: ~

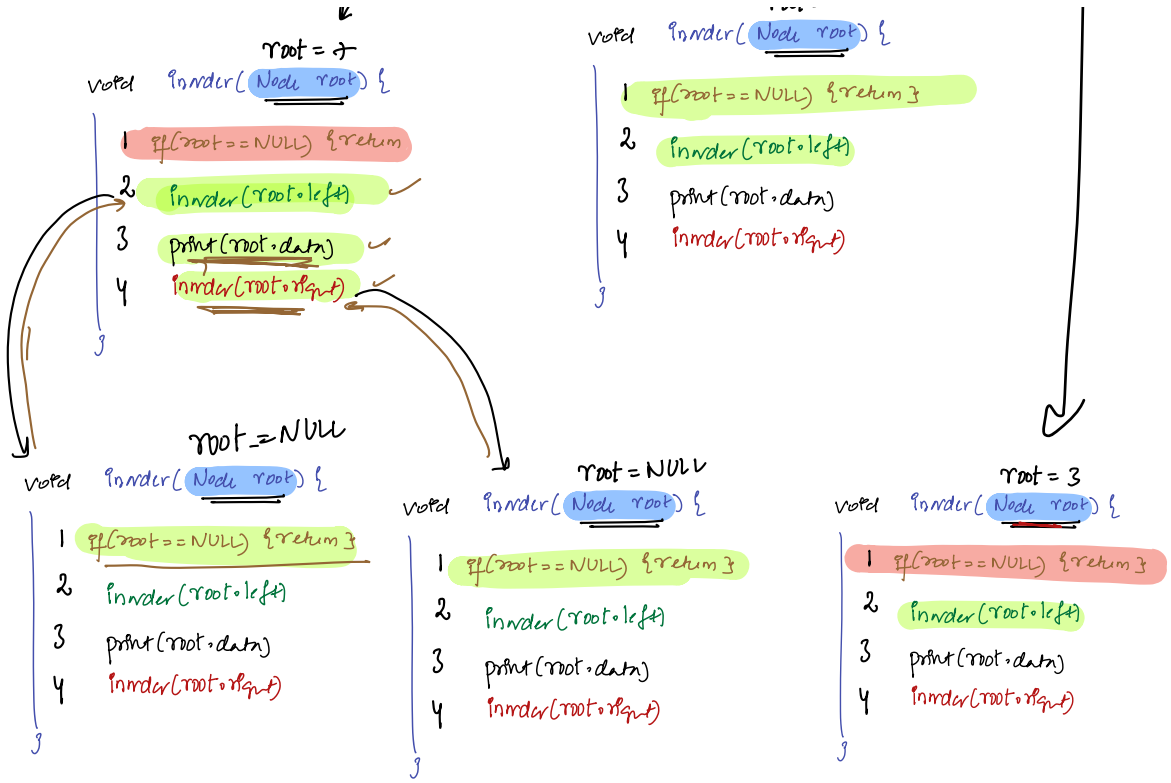
obs1:

root is root node of Entire Tree

root.left is root node of Entire LST

root.right is root node of Entire RST


output: 4 2 7 5 1 3 6
root = 1
void inorder(Node root) {
1 if(root == NULL) {return}
2 inorder(root.left)
3 print(root.data)
4 inorder(root.right)
}
root = 2
root = 4
root = 5
root = NULL
root = NULL
root = NULL



// inorder → iteratively

Break: 9:35pm – 9:45pm

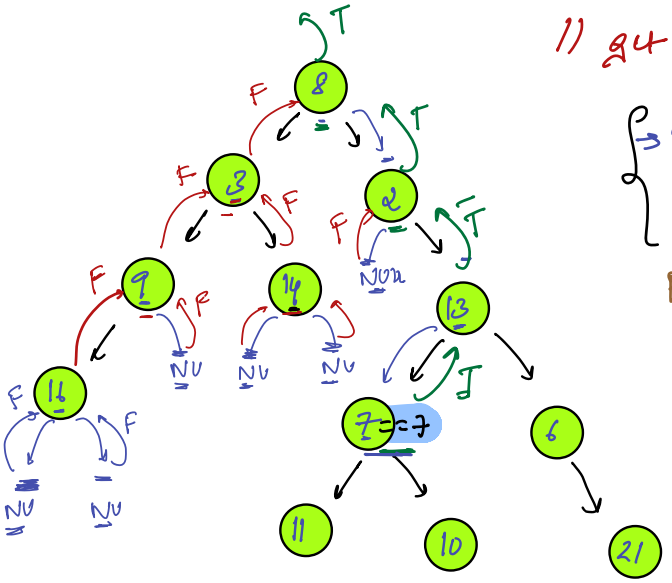
→ Search for a node:

→ Path from S → D

→ LCA

// Given Binary Tree Root Node & data k, check if k is present or Not

→ all value are unique



// get the path to 7 from 8

{ → nodes which are returning True
they belong to path. }

path:

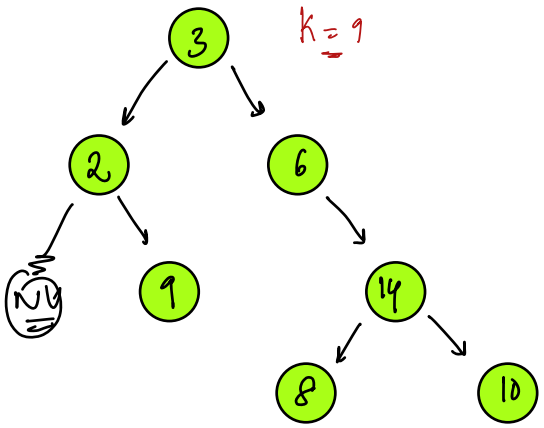
{ 7 13 2 8 }

path we get: node to root Node

Reverse: { Root Node → given Node }

```
// List<Int> path; → { global variable }

boolean search(Node root, int k){
    if (root == NULL) { return false }
    if (root.data == k) { path.add(root.data) return True }
    if ( search(root.left, k) ){
        path.add(root.data) return True
    }
    if ( search(root.right, k) ){
        path.add(root.data) return True
    }
    return false
}
```



True

root = 3 k = 9

```
boolean search(Node root, int k){
1) if(root == NULL){ return false }
2) if(root.data == k){ return True }
3) if(search(root.left, k)){
       return True
   }
   if(search(root.right, k)){
       return True
   }
   return false
}
```

root = 2 k = 9

```
boolean search(Node root, int k){
1) if(root == NULL){ return false }
2) if(root.data == k){ return True }
3) if(search(root.left, k)){
       return True
   }
   if(search(root.right, k)){
       return True
   }
   return false
}
```

root = NULL k = 9

```
boolean search(Node root, int k){
1) if(root == NULL){ return false }
2) if(root.data == k){ return True }
3) if(search(root.left, k)){
       return True
   }
   if(search(root.right, k)){
       return True
   }
   return false
}
```

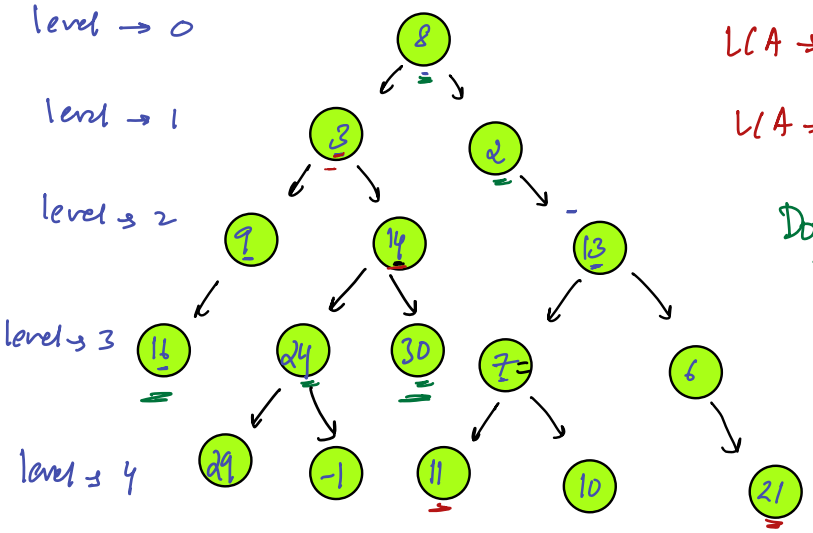
root = 9 k = 9

```
boolean search(Node root, int k){
1) if(root == NULL){ return false }
2) if(root.data == k){ return True }
3) if(search(root.left, k)){
       return True
   }
   if(search(root.right, k)){
       return True
   }
   return false
}
```

// LCA: given 2 node find least common ancestr

→ path of A from root

↳ { Common ancestr at max level }



LCA → {11, 21} → 13

LCA → {29, 30} → 14

Don't type :

Query → LCA → {16, 30}

LCA (16, -1) → -3

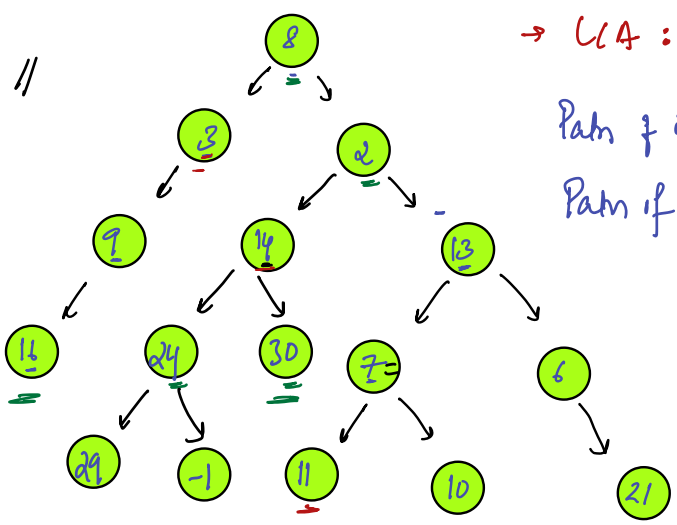
LCA (30, 13) →

// How to get LCA 11, 21

Path of 11 from root Node : 8 → 2 → 13 → 7 → 11

Path of 21 from root Node : 8 → 2 → 13 → 6 → 21

least common ancestr



→ LCA : (29, 7) :

Path of 29 from 8 → 8 → 2 → 14 → 24 → 29

Path of 7 from 8 → 8 → 2 → 13 → 7

least common Ancestr

// LCA (A, B)

: get path of A from root

: get path of B from root

: (Iterate in both paths & get last common node)