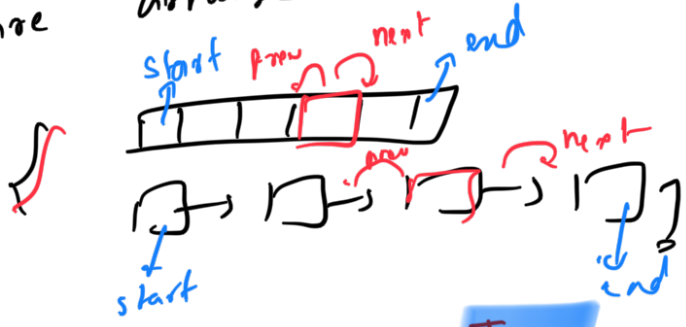
TREES - 1

Amazon / MS / Adobe

→ Recursion
→ Tree structure

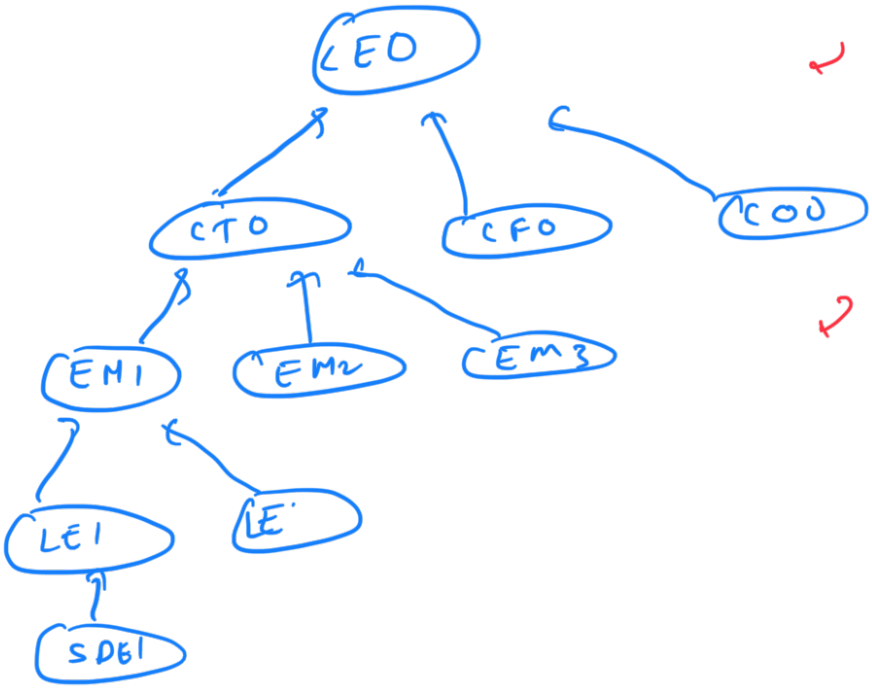
Linear D.S:

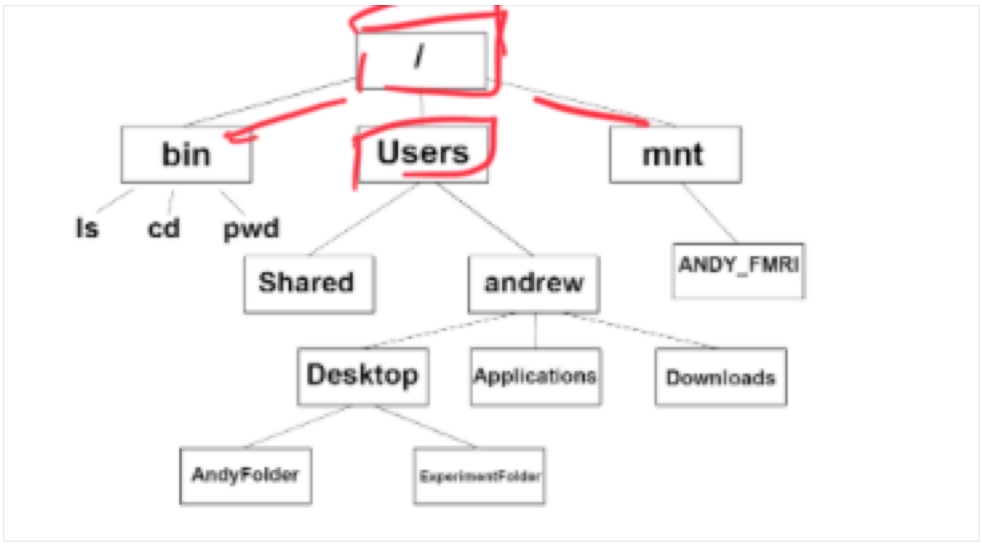
↳ Elements are Arrays, Linked, Queues, Stacks arranged in a sequential manner



If we need hierarchy, then comes Trees

Hierarchy





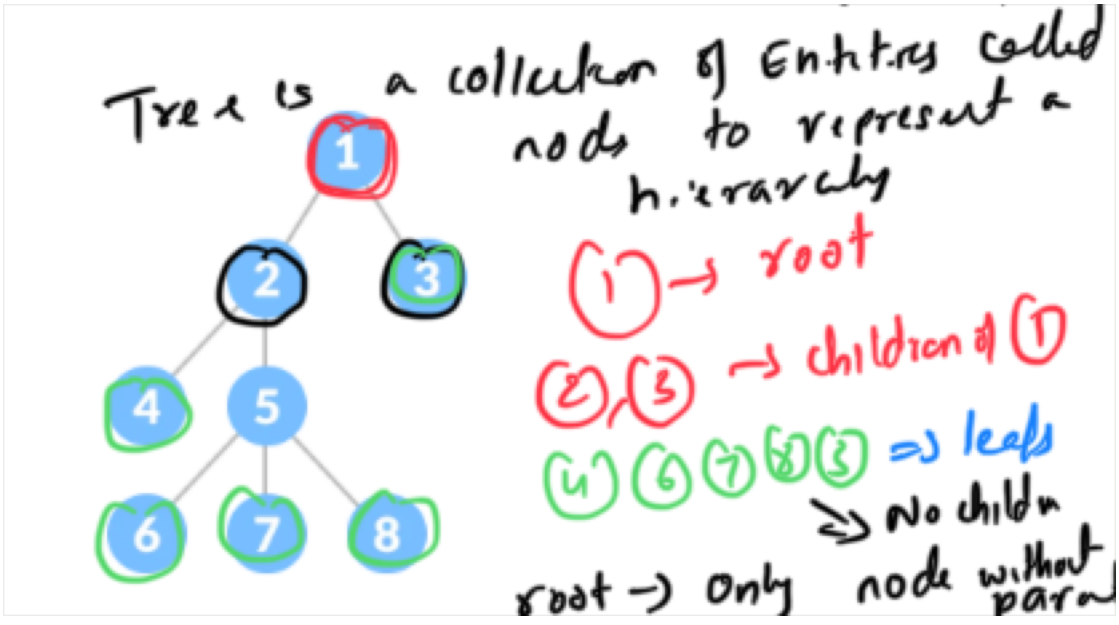
Non Linear D.S : Trees, Graphs

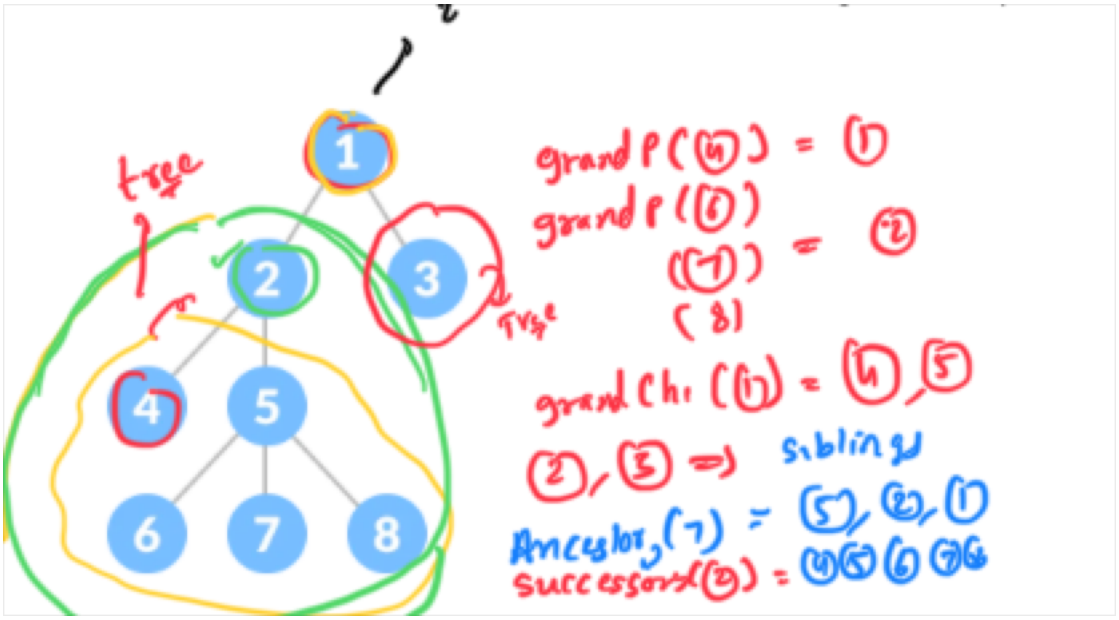
b-tree, Heaps, Tries, Segment trees

DP

1D arrays

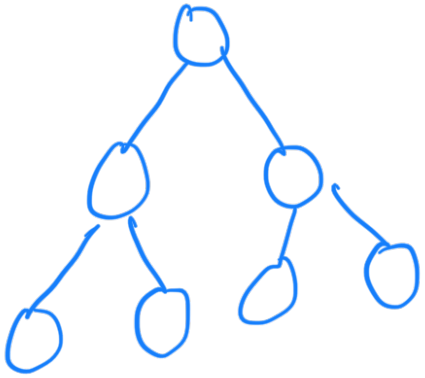
2D arrays

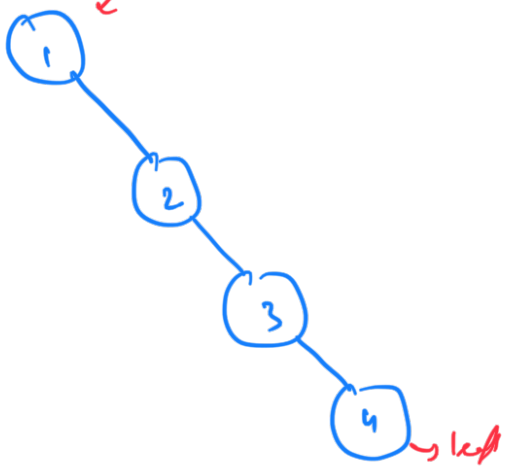



tree
1
2
3
4
5
6
7
8
tree
grandP(4) = 1
grandP(6)
(7) = 2
(8)
grandChi(1) = 4, 5
2, 3 => sibling
Ancestor(7) = 5, 2, 1
Successor(3) = 4 5 6 7 6

Binary Tree :

Every Node can have atmost 2 children {0, 1, 2}





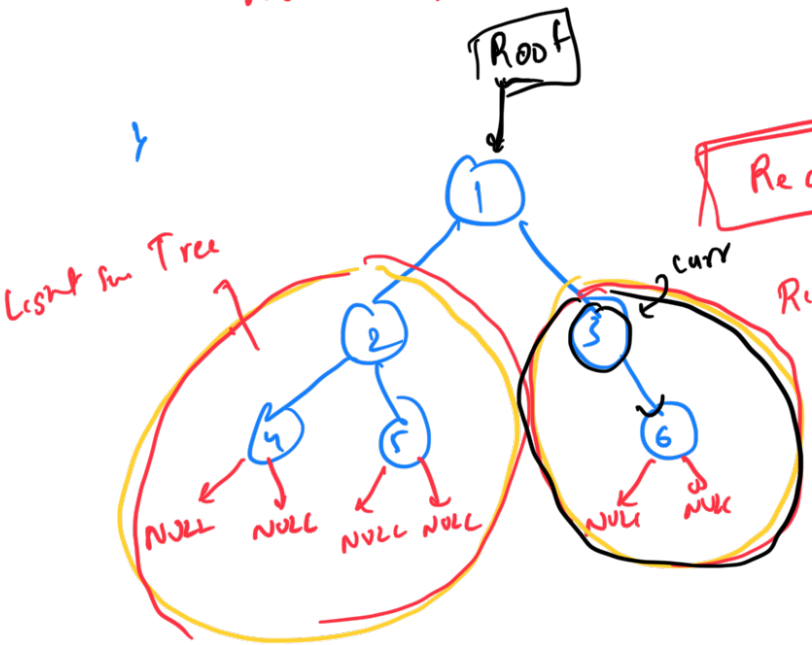
Binary Tree

1 → Binary

{0, 1, 2}

Empty Tree ⇒ Binary Tree

```
class Node {
    int data;
    Node left, right;
}
```



Recursion

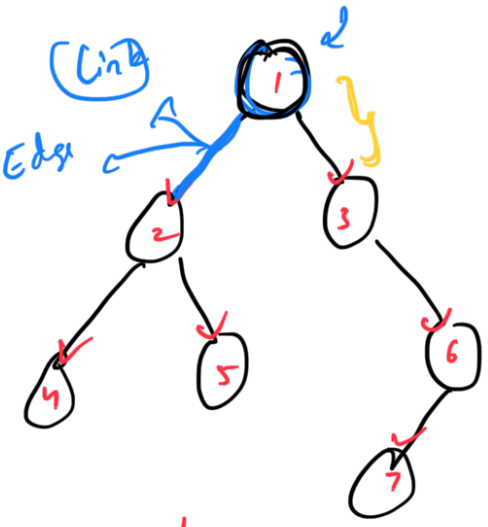
→ Linked
→ Vector

Properties:

1) No. of edges

N nodes

-) Every node except the root has exactly one incoming edge

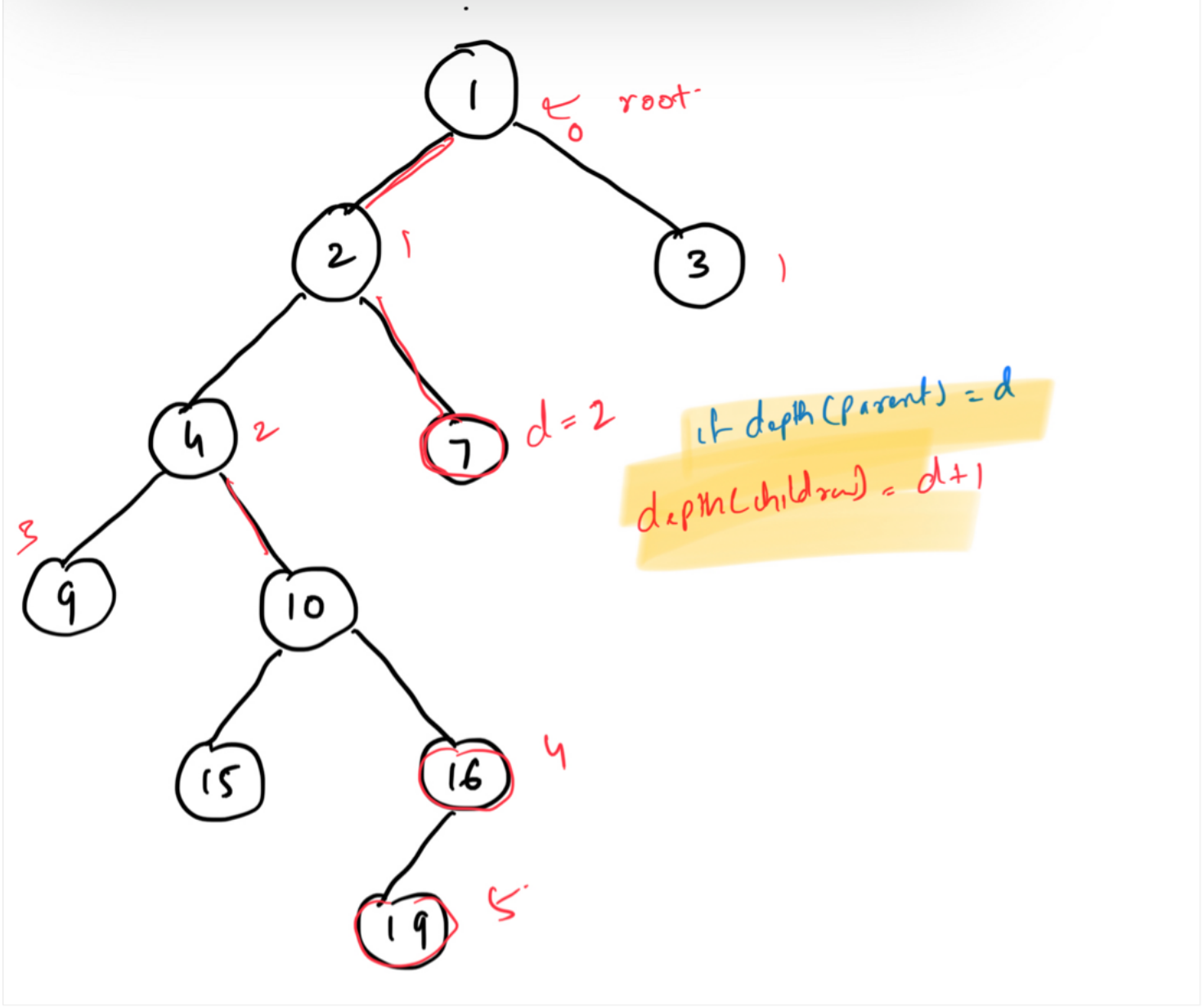


(N-1) nodes have 1 incoming edge

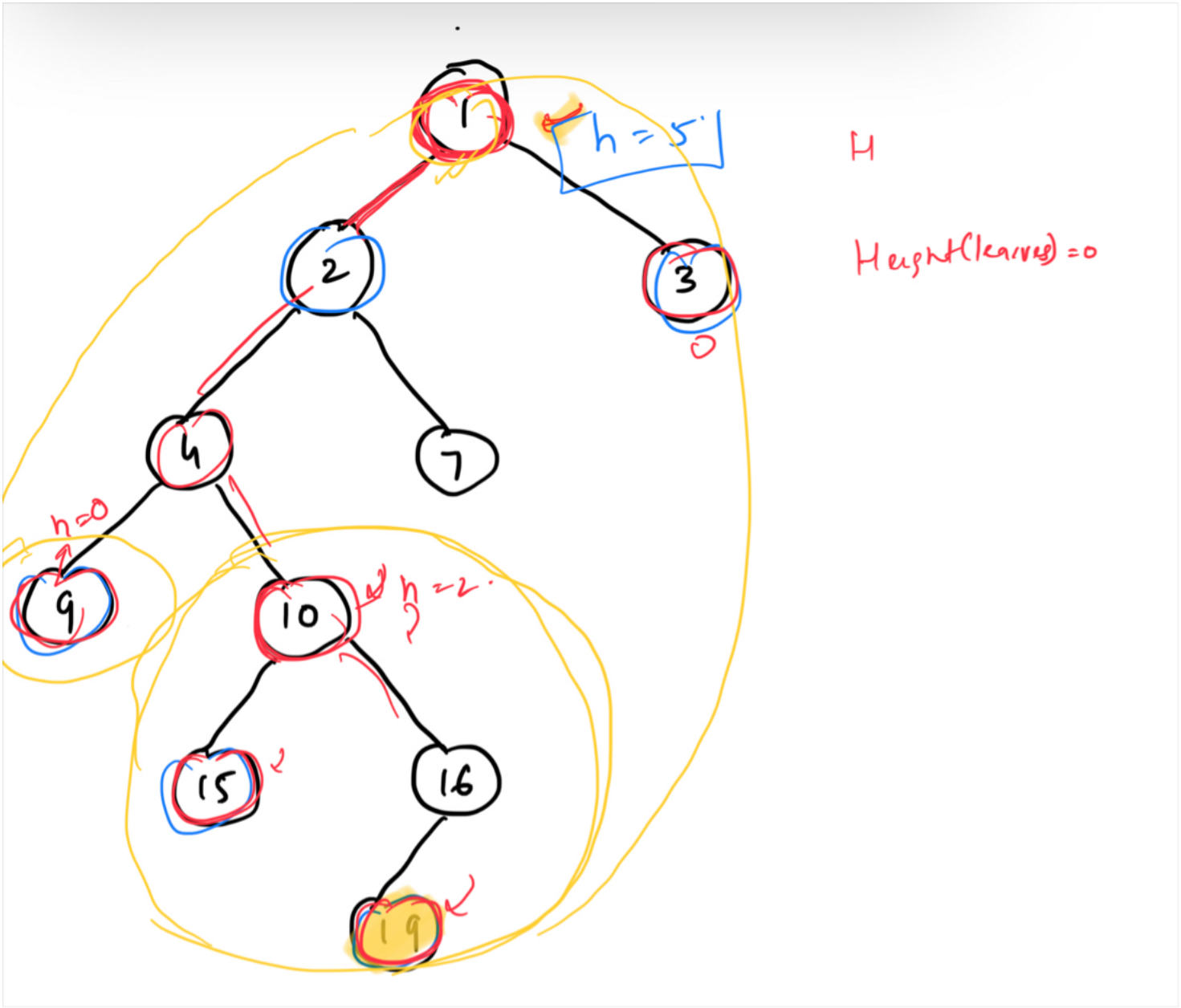
Edges = N-1

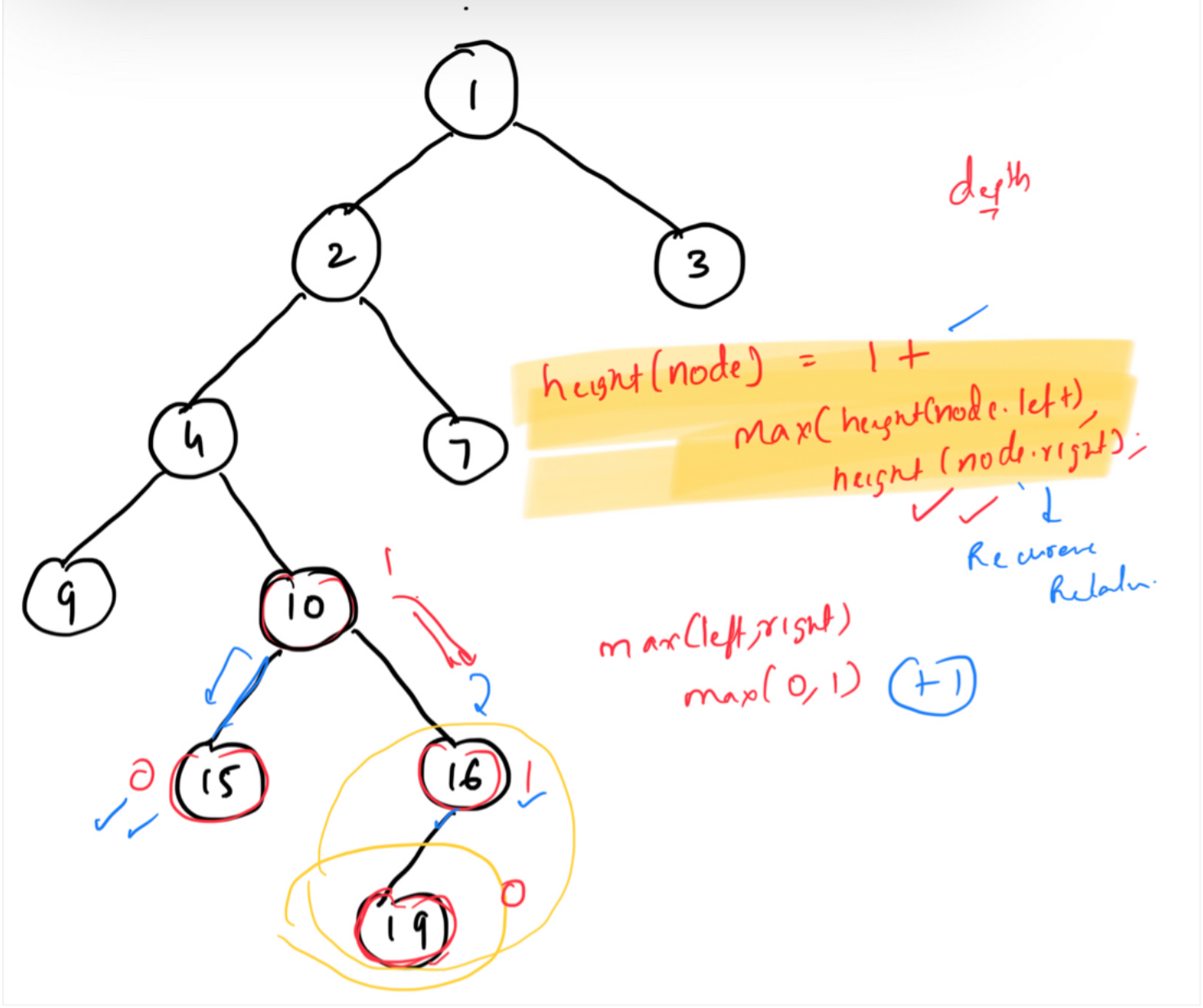
2) Depth of a Node:

Distance (No. of edges) from root to a node



3) Height: Distance (No. of edges) of the current node to the farthest leaf

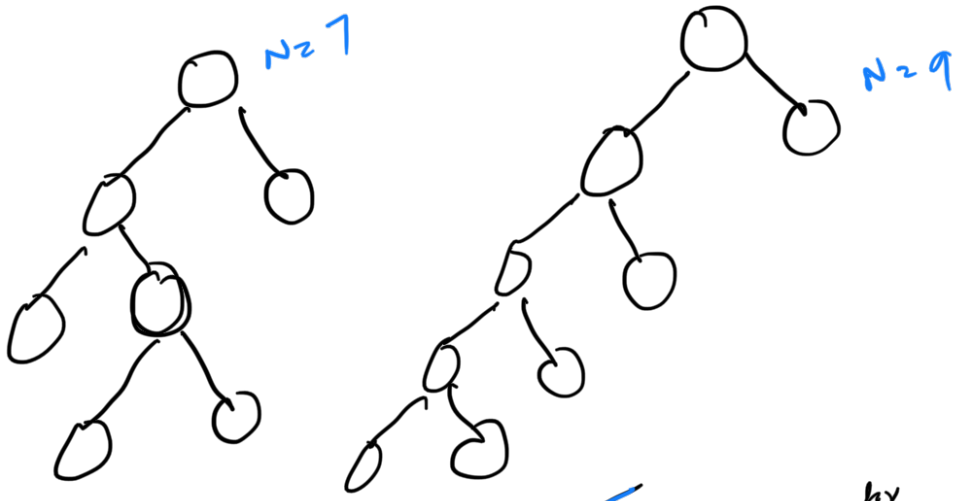

1
h = 5
2
3
4
7
h=0
9
10
h=2
15
16
19
H
Height(leaves) = 0


1
2
3
4
7
9
10
15
16
19
depth
height(node) = 1 +
max(height(node.left), height(node.right));
Recurrence Relation
max(left, right)
max(0, 1) (+1)

Types of Tree

1) Proper Binary Tree

Every node has either 0 or children



✓ ✓

Google

Bitwise AND operator

N & 1 ⇒ 0, 1

No. of nodes

N & 1 = 0 [Even]

↓ Last Bit of N = 0

$N = N_1 + N_2$

N_1 = No of nods with 2 children [$2N_1$]

N_2 = No. of nods with 0 children = $0 \times N_2$.

N & 1 = 1 [odd]

Edges = $2N_1$

Edges = Nods − 1

Nods = Edges + 1 → odd?

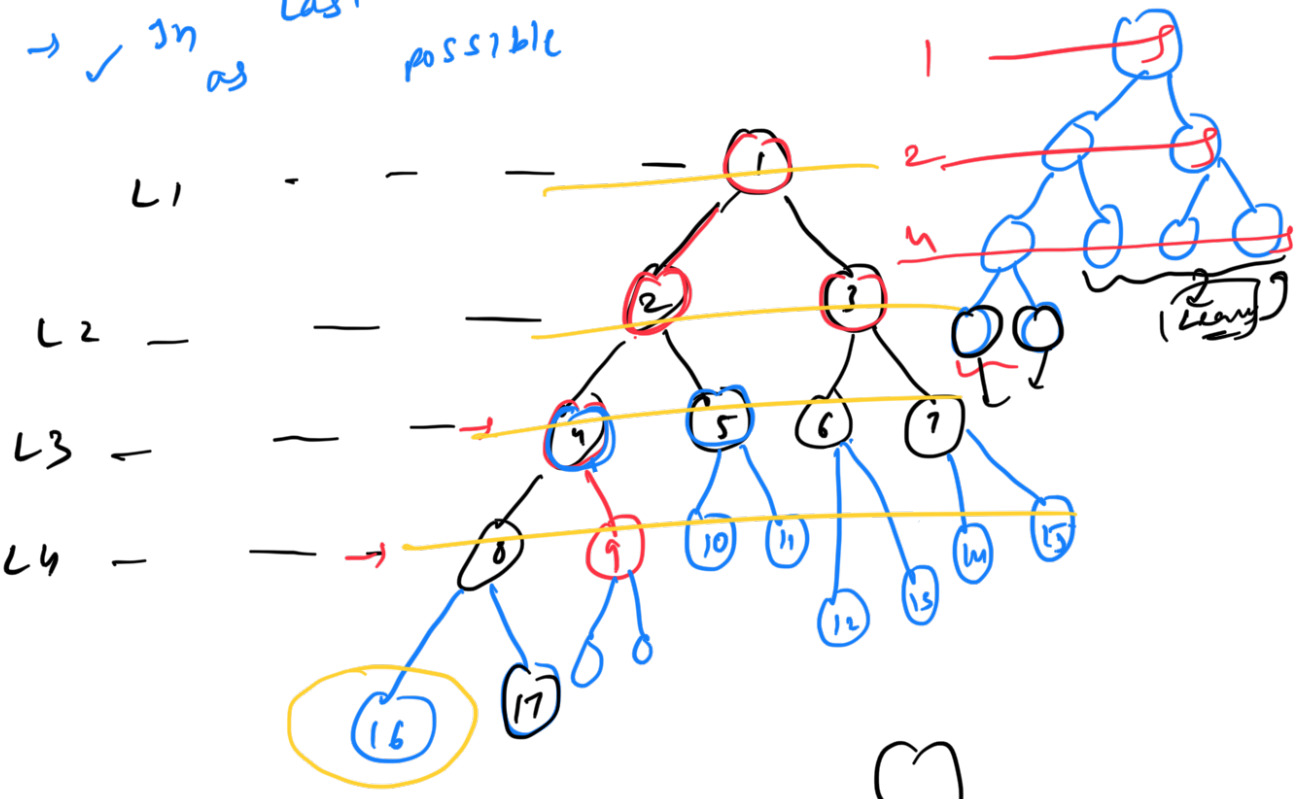
$= 2N + 1$

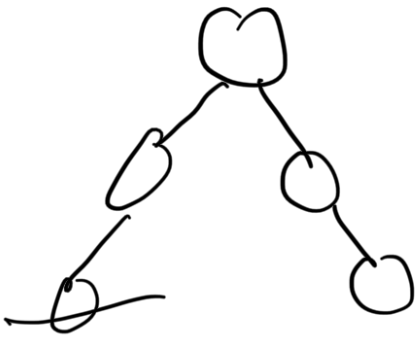
Elem

2) Complete Binary:

-> Every Level is completely filled except possibly the last level

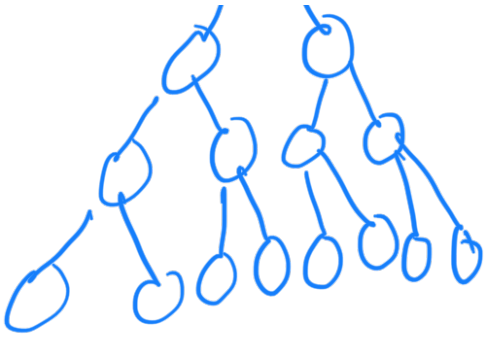
-> In Last level, the nodes should as left as possible





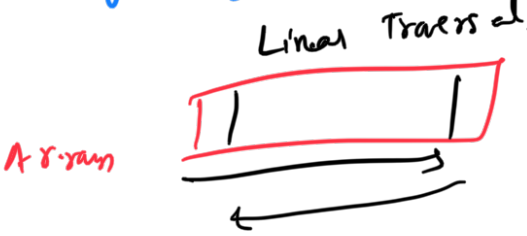
Perfect Binary Tree

All the levels are completely full



Traversal:

the process of visiting nodes of a DS

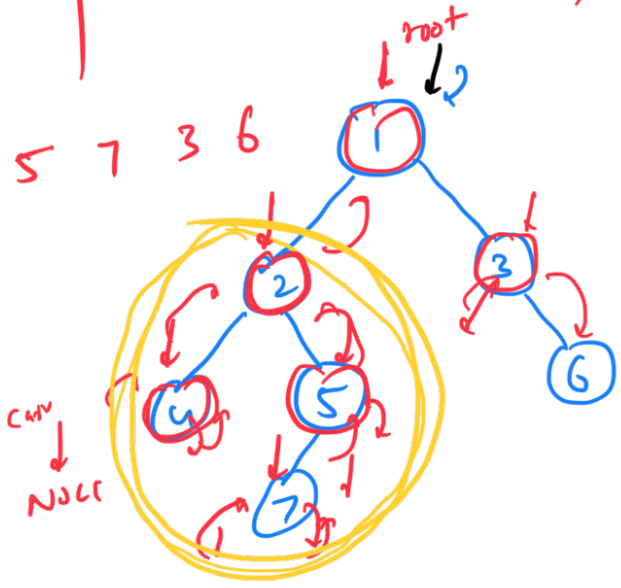


Preorder

Root, Left, Right

Inorder

Preorder: 1 2 4 5 7 3 6



Recursion

1) Assumption: preorder(root)

Lets trust this function that it prints preorder traversal of a tree

2) Main Logic / Recursion Relation.

```
print(root.data);
preorder(root.left);
preorder(root.right);
```

3) Base Case:

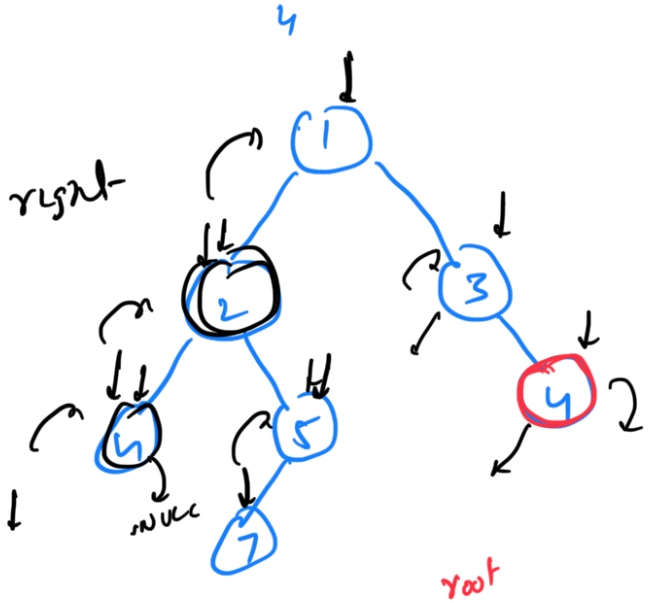
```
if(root == NULL) return;
```

```
void preorder(root) {
  if(root == null) return
  print(root.data);
  preorder(root.left);
  preorder(root.right)
}
```

```
postorder(root.left)
postorder(root.right)
print(root.data);
```

Inorder:

left, root, right



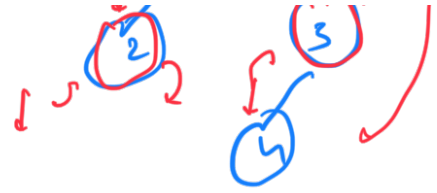
Inorder:

4 2 7 5 1 3 4

Postorder:

Postorder: 2 4 3 1

Left Right Root



T.C : O(n)

S.C : O(H)

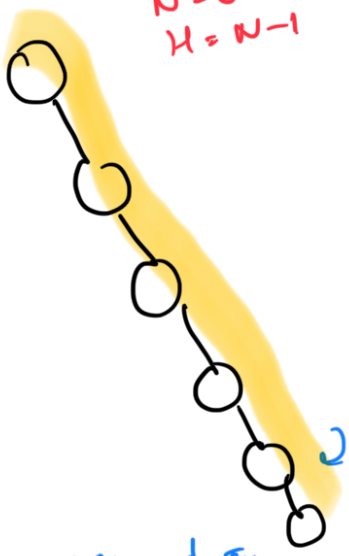
Worst Case : O(N)

Amortized

(H = = (log N)
Balanced Binary Tree

Recursion : Recursion call stack

N = 6
H = N - 1



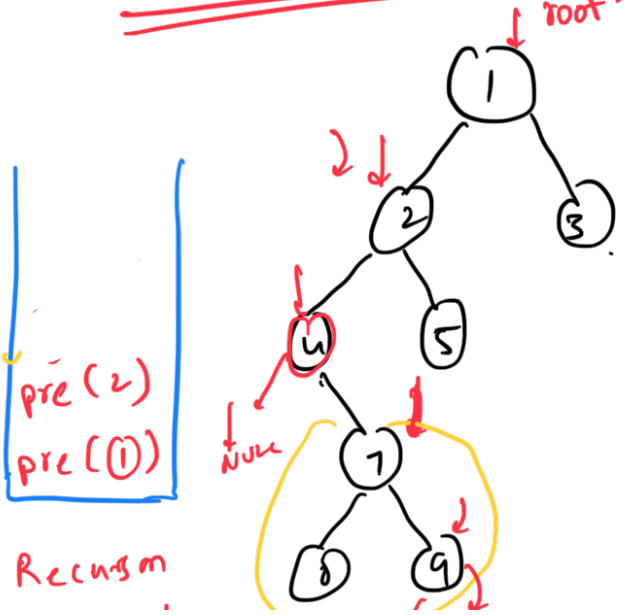
Skewed Tree

⇒ Serialization

Array

Complete Binary

Traversals Using Iteration

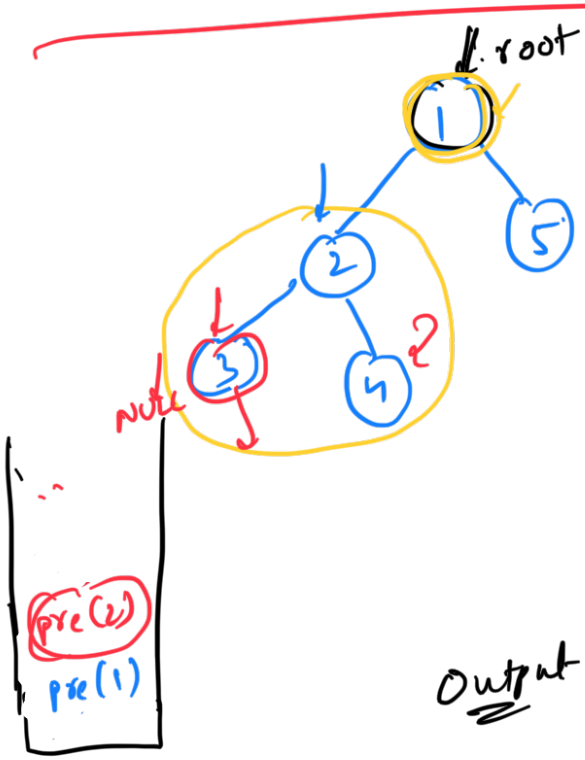


Recursion

Output :

1 2 4 7 8 9

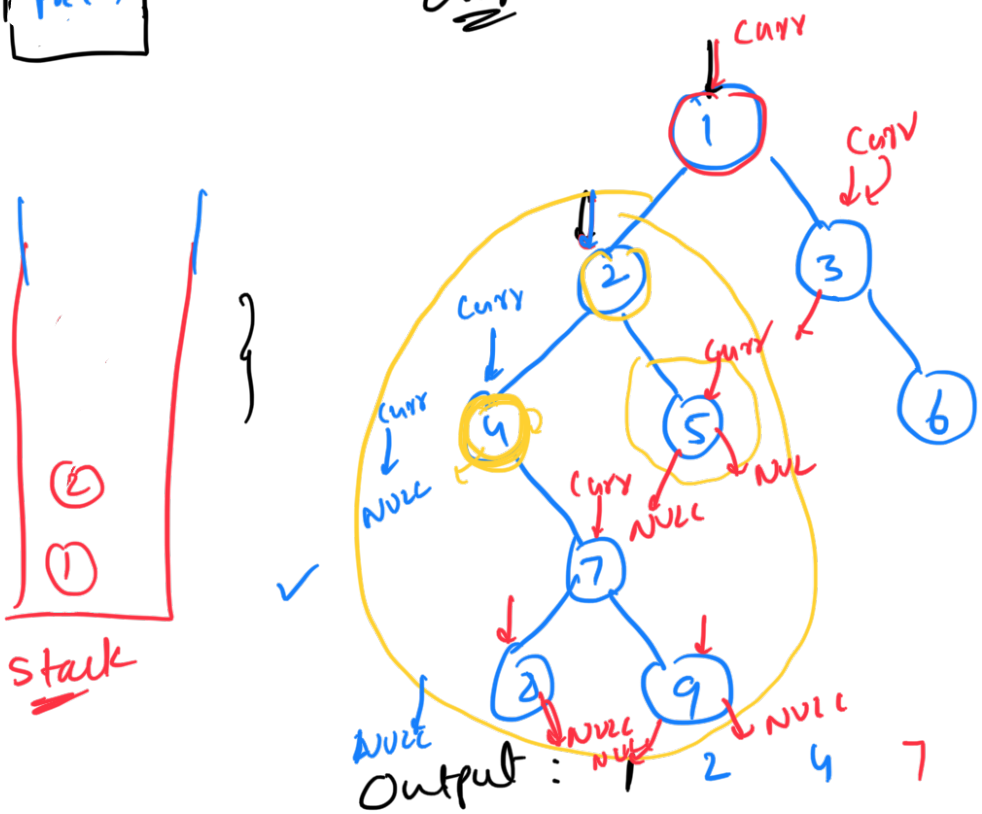
Call stack:




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

Output: 1 2 3

Root Left Right



curr = curr.left

curr = stack.top().right
stack.pop();

curr = top().right
stack.pop()

Output: 1 2 4 7 8 9 5 3

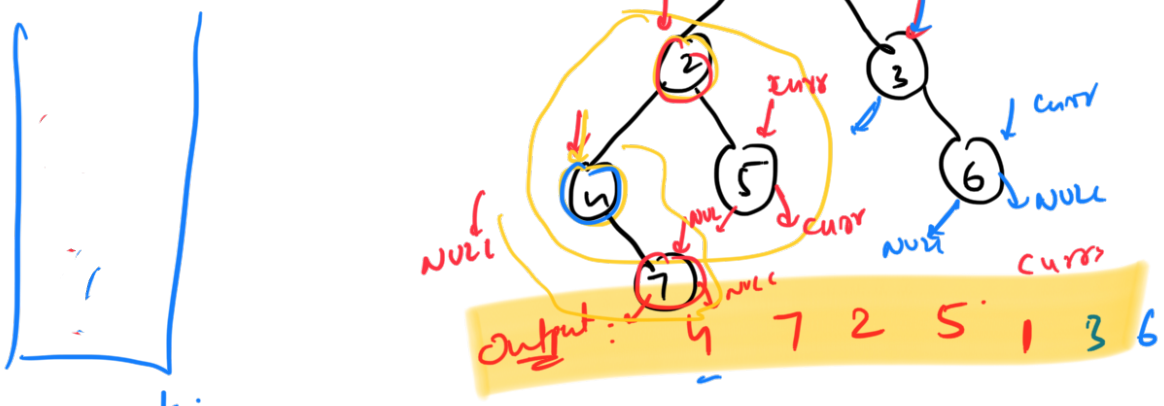
Stack<int>?

Stack<Node>

curr = root;

while (!stk.isEmpty() || curr != NULL) {

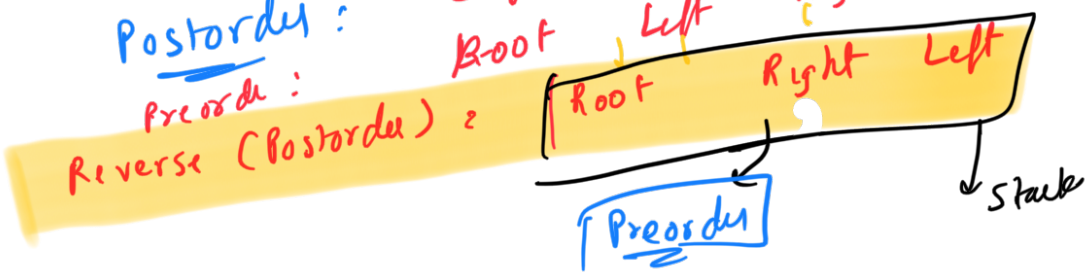
```
if(curr != NULL) {
    print(curr.data);
    stk.push(curr);
    curr = curr.left;
}
else {
    top = stack.top();
    stack.pop();
    curr = top.right;
}
}
```



Postorder: Left Right Root

Preorder: Root Left Right

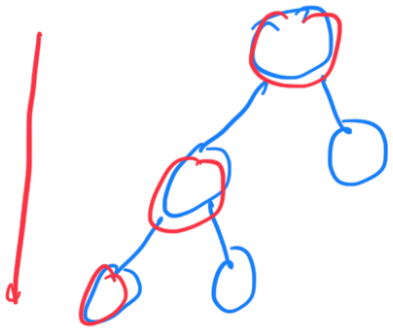
Reverse (Postorder): Root Right Left



When to use preorder / Postorder

Preorder:

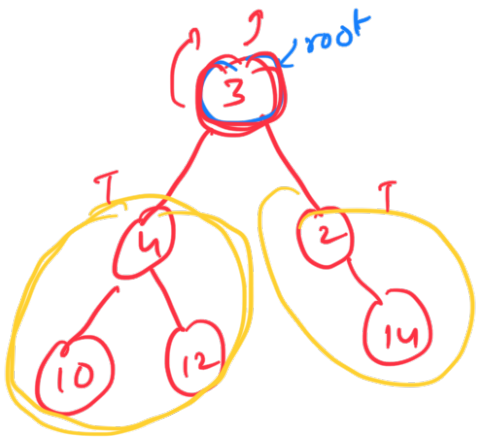
Control is going from top to bottom



... require information from

1) If we don't ... children

2) Apply Fail-Fast Method

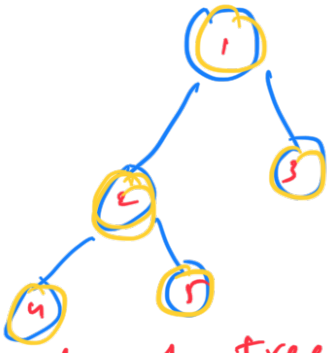


Postorder:

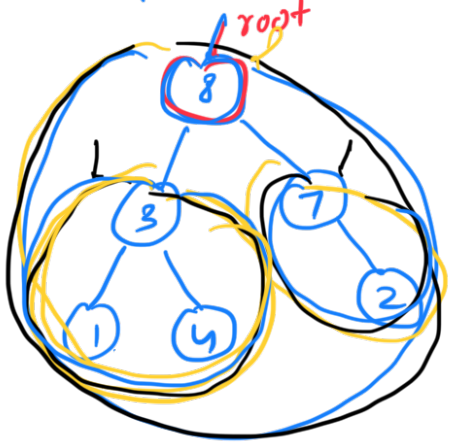
Control from bottom to up

Postorder: Left Right Root

sum(1)



Question: Find sum of all nodes of tree



sum(root) = root.data + sum(root.left) + sum(root.right)

```
int sum(root) {
    if(root == NULL) return;
    return root.data + sum(root.left) + sum(root.right)
}
```

Post order

my data + left + right

global sum = 0

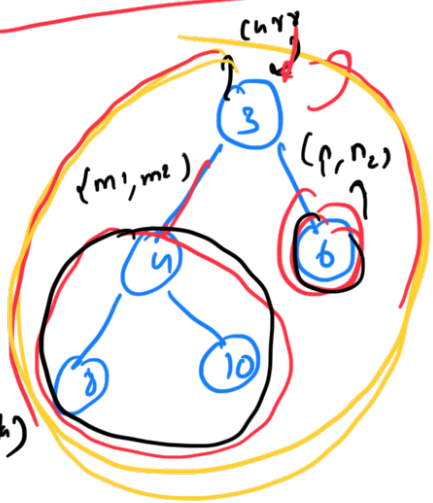
Preorder, Postorder, Inorder

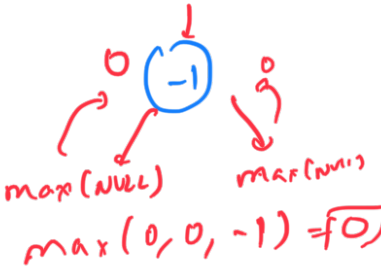
Global Value

Question: Max value of B.T

```
int max(root) {
    if(root==NULL) return -INF;
    left = max(root.left);
    right = max(root.right)
    return max(left, right, curr.data)
}
```

Post order



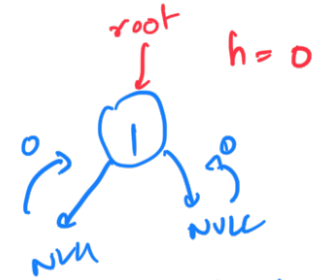


Question: Height

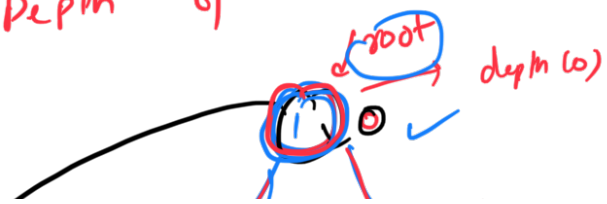
height(root) =

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

Post order

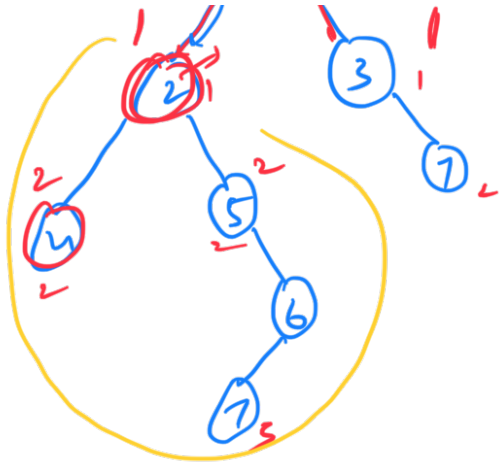


Question: Depth of all node




```
class Node {
    int data;
```

depth(root, 0)
depth(root.left,



int left, right;
int depth = 0;

node.depth=?
root, left, right

Preorder

depth(parent) = d
depth(children) = d+1

void depth (root, d) {
if(root == NULL) return;

root.depth = d;
depth (root.left, d+1);
depth (root.right, d+1);

node.left
node.rgt

queue<Node>
Node

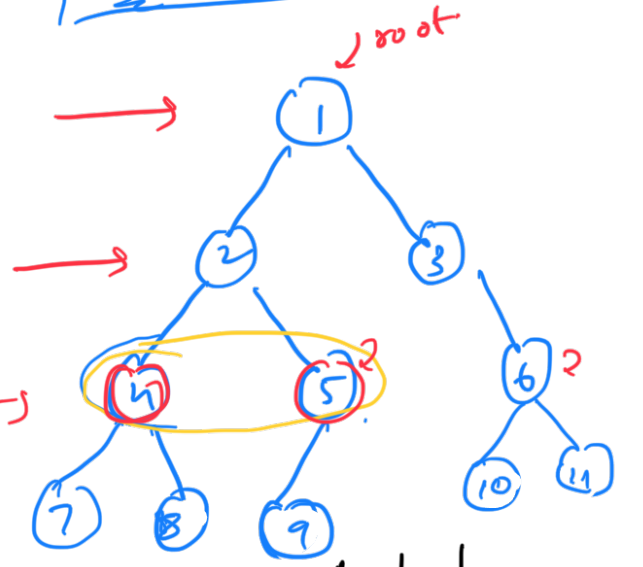
Integer queue

Output: 1 2 3 4 5

Level Order Traversal

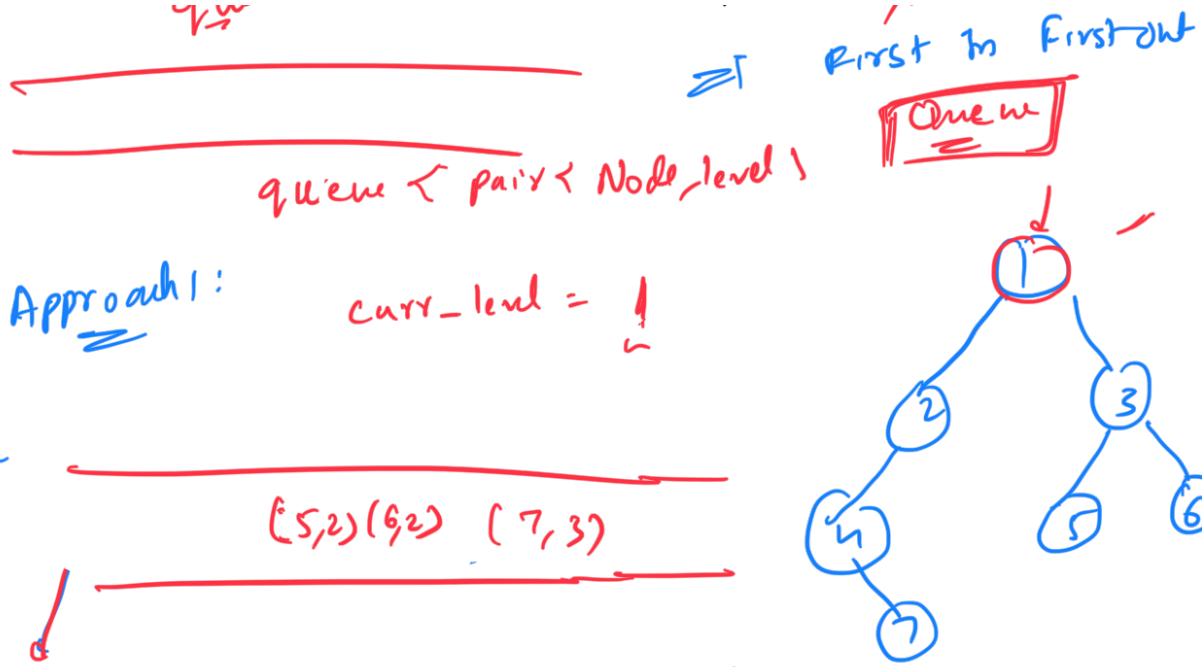
2D array

[
[1]
[2, 3]
[4, 5, 6]
[6, 7, 8]
]



Output: 1 2 3 4 5 6 7 8 9

1 2 3 4 5 6 7 8 9 10 11



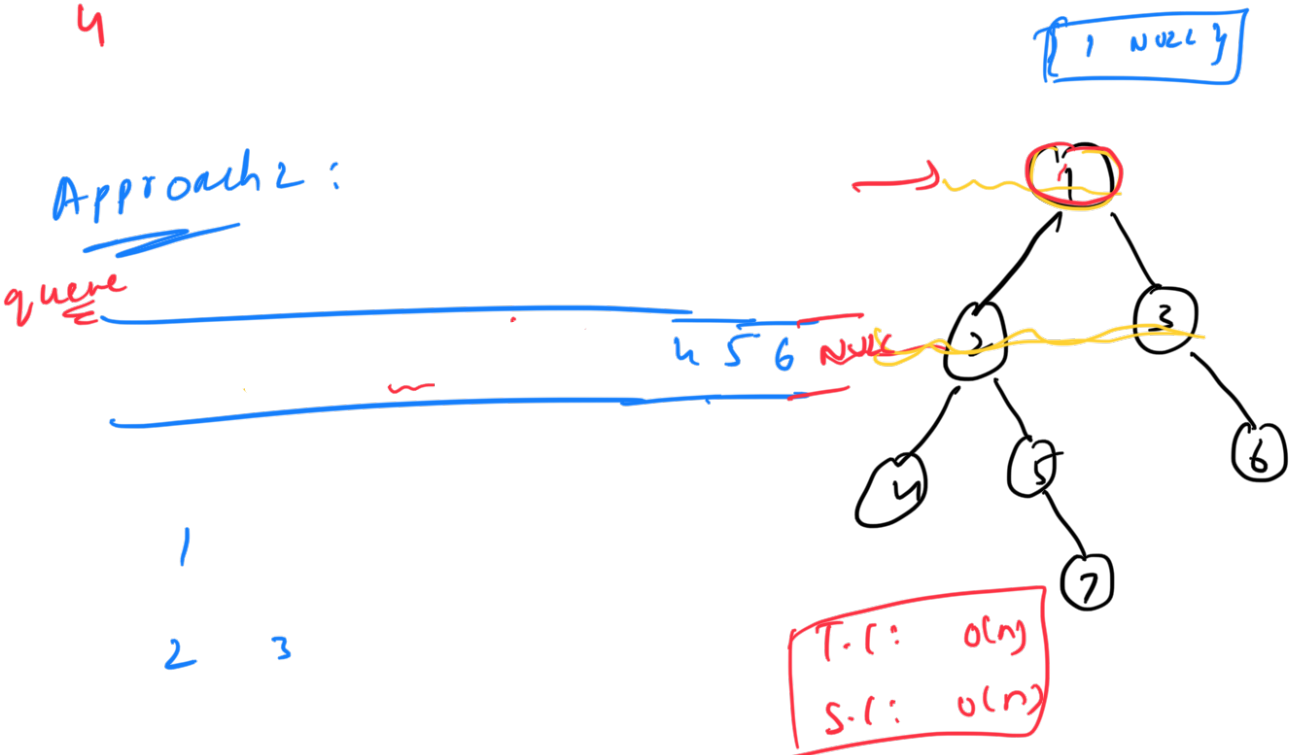
T.C: O(N)

S.C: O(N) ⇒ Queue Space

1

2 3

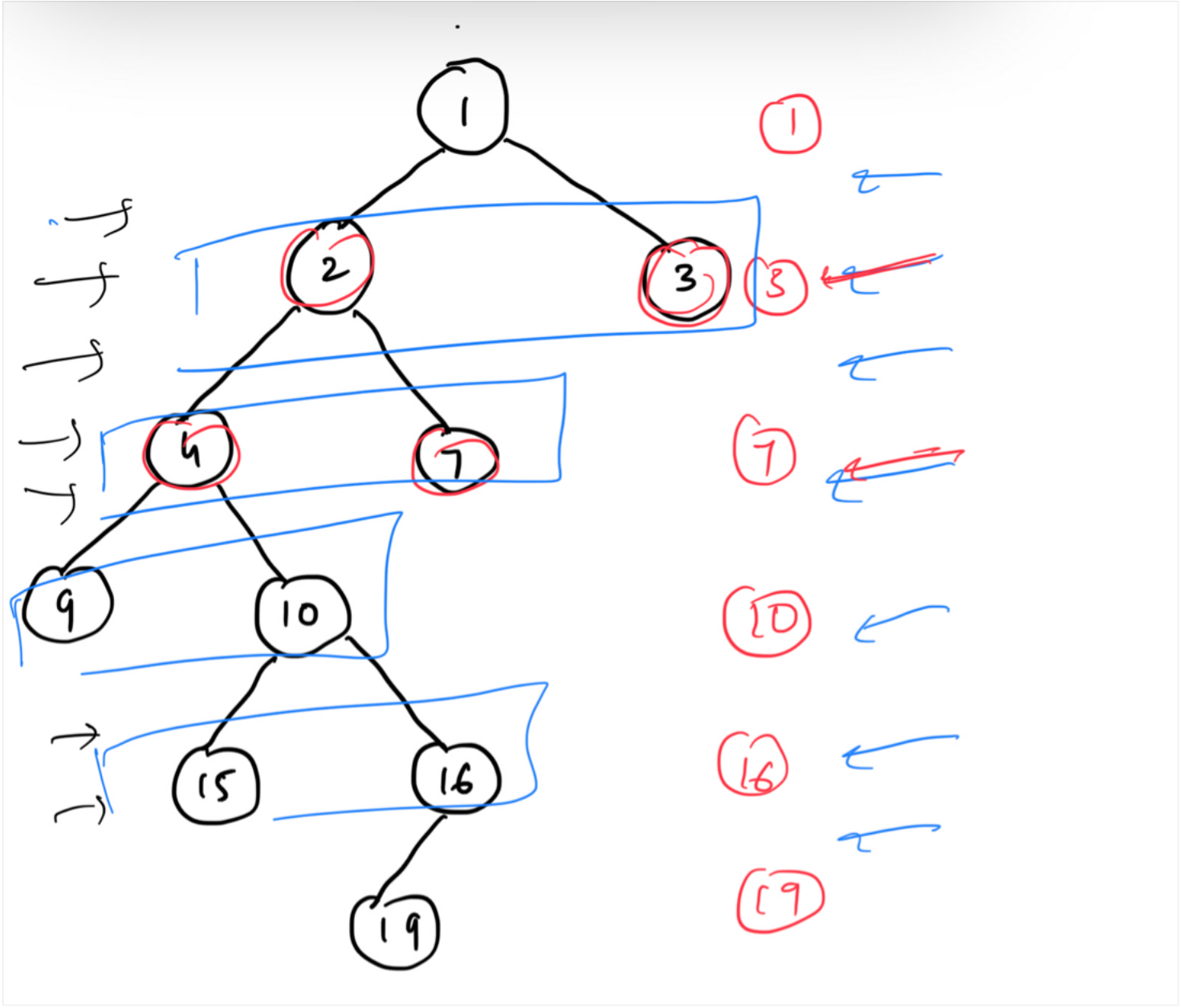
4



1

2 3

View of a BT



Linked In: Yahnit Sirineni

Level Order Traversal : By storing level of every node

```
level_order_pair(Node* root){
	queue<pair<Node*, int>> q;
	cur_level = 0;
	q.push(<root, 0>);

	while(!q.empty()){
		top = q.front();
		q.pop();
		if(top.level != curr_level){
			cout << "\n";
			curr_level++;
		}
		cout << top->val << endl;
		if(top.first->left) q.push(<top.node->left, top.level+1>)
		if(top.first->right) q.push(<top.node->right, top.level+1>)
	}
}
```

Level Order Traversal : By using a delimiter

```
levelOrder(Node* root){
	queue<Node*> q;
	q.push(root);
	q.root(NULL);

	while(q.size() > 1){
		Node* f = q.front();
		q.pop();

		if(f != NULL){
			cout << t->val;
			if(t->left) q.push(t->left);
			if(t->right) q.push(t->right);
		}
		else{
			cout << "\n";
			q.push(NULL);
		}
	}
}
```