Trees - 1

Amazon, M.S, Adobe

→ Recursion
→ Structure of the tree

LL of Trees

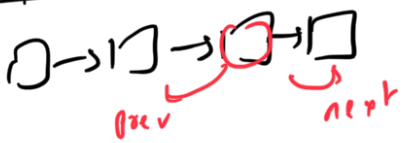
5 session
→ Trees 1, 2
→ B.S.T
→ LCA
→ Problems on Trees

Linear Data Structures :

Arrays, L.L, Stacks, Queue

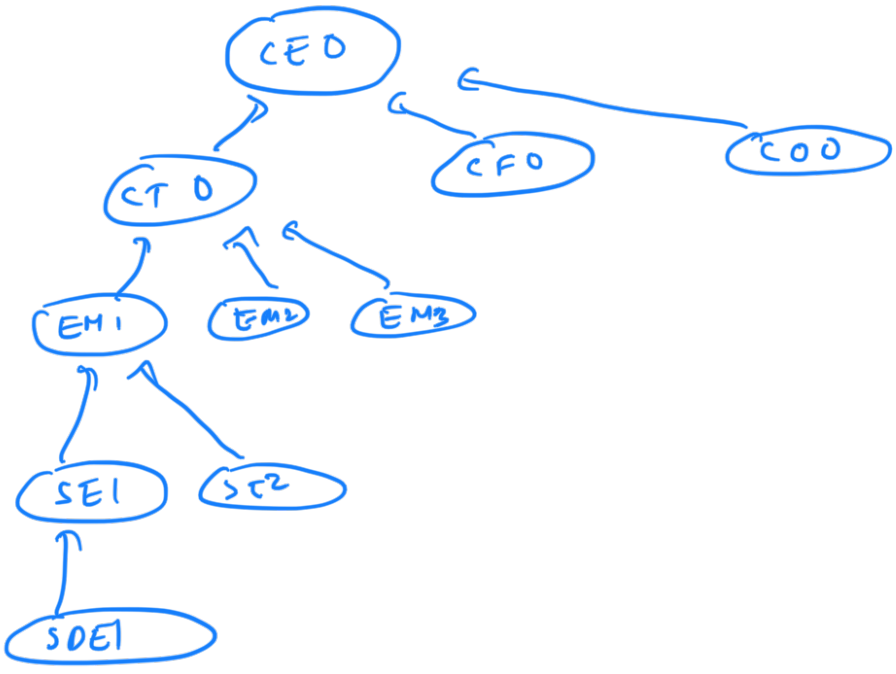
↳ Elements are arranged in a linear / sequential manner

Array: | 2 | 4 | 5 | 1 | 2 |

LL: 

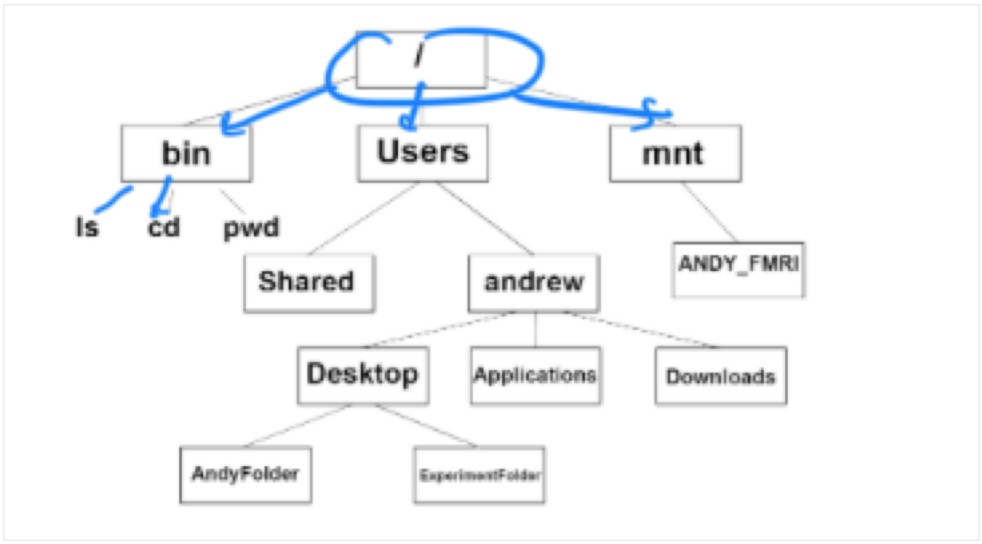
→ No Hierarchy

Trees come into picture



SDE2

Directory :

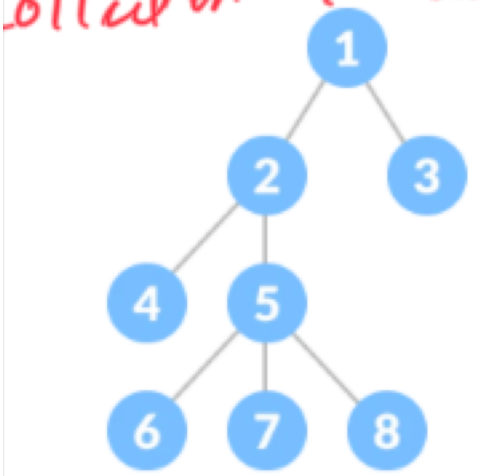


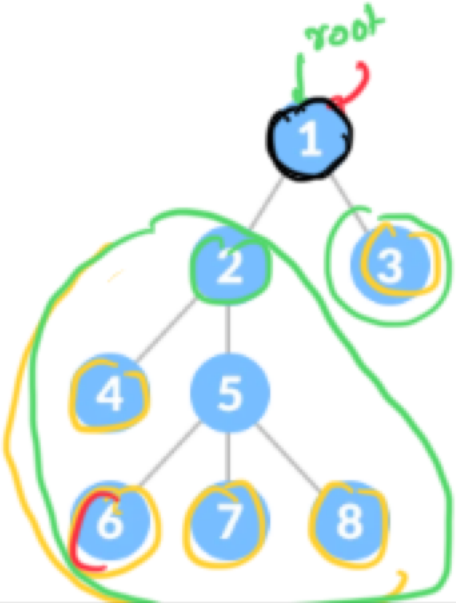
Non Linear D.S : Trees, Graphs

heap Segment Trees Trie

DP

Collection of nodes/entities which are linked together to form a hierarchy





Terminology

Root => 1 [No parent]

2, 3 are children to 1

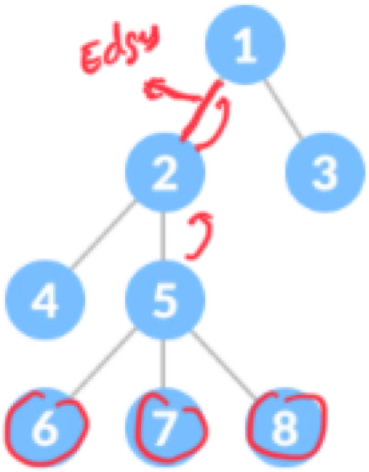
1 is the parent of 2

par(7) = 5, par(8) = 5

Nods with no children => leaf

=> Left Subtree & Right Subtree are trees

6, 7, 8 are called siblings



grandP(5) = 1

grandP(8) = 2

grandC(2) = 6, 7, 8

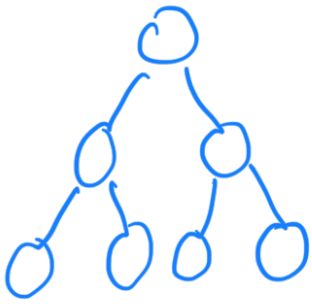
Ancestor(7) = 5, 2, 1

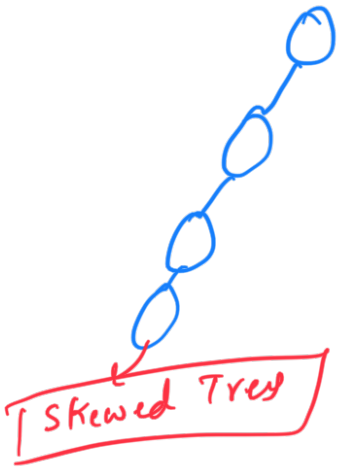
Ancestors(4) = 2, 1

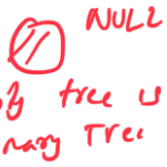
Successor(2) = 4, 5, 6, 7, 8

Binary Tree

It is a special tree where every node can have atmost 2 children. {0, 1, 2}

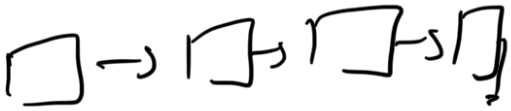


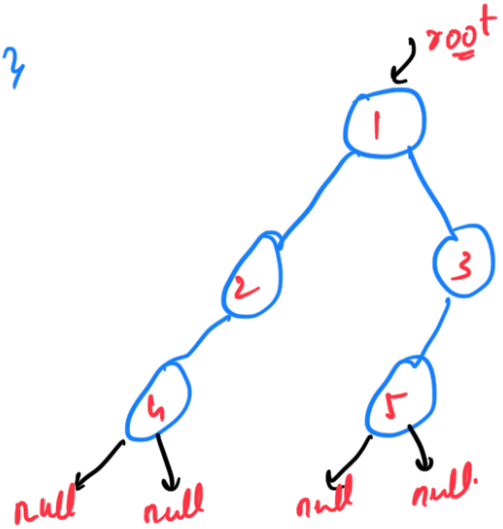




Structure

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



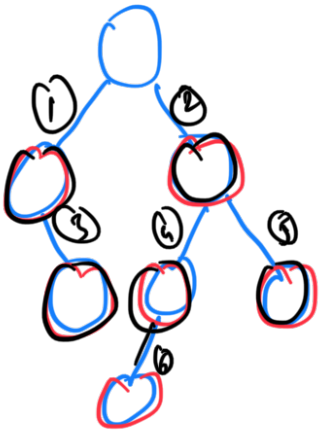


Properties:

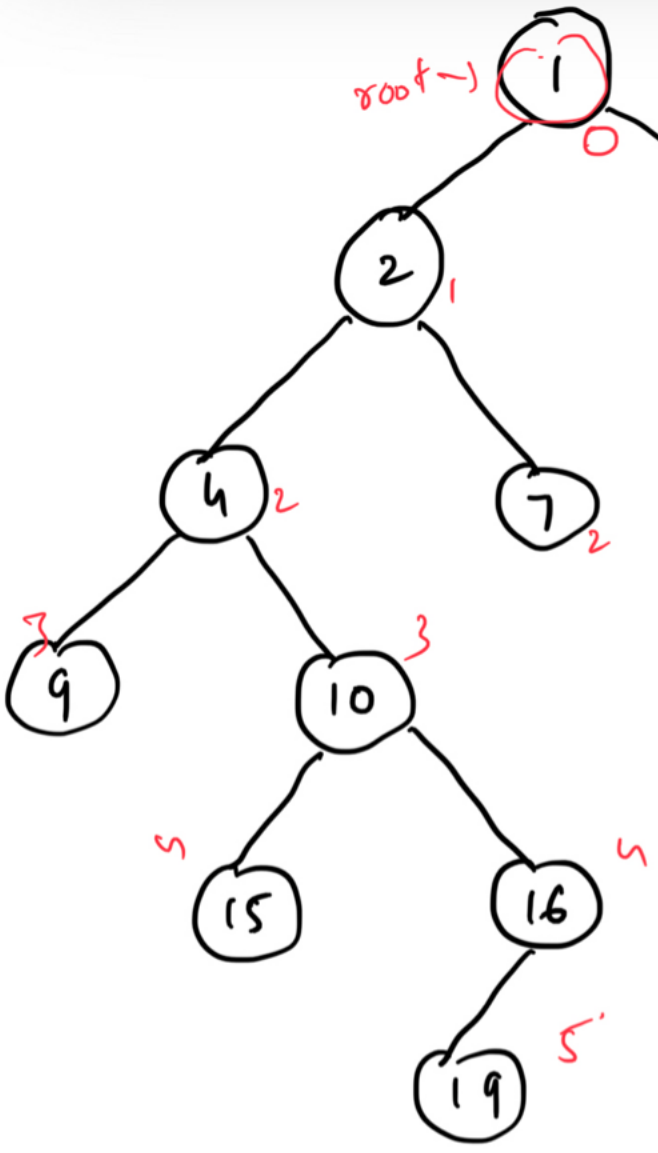
1) N nodes => (N-1) edges

→ Every node except root has an incoming edge => N-1

→ No node will have more than 1 incoming edge



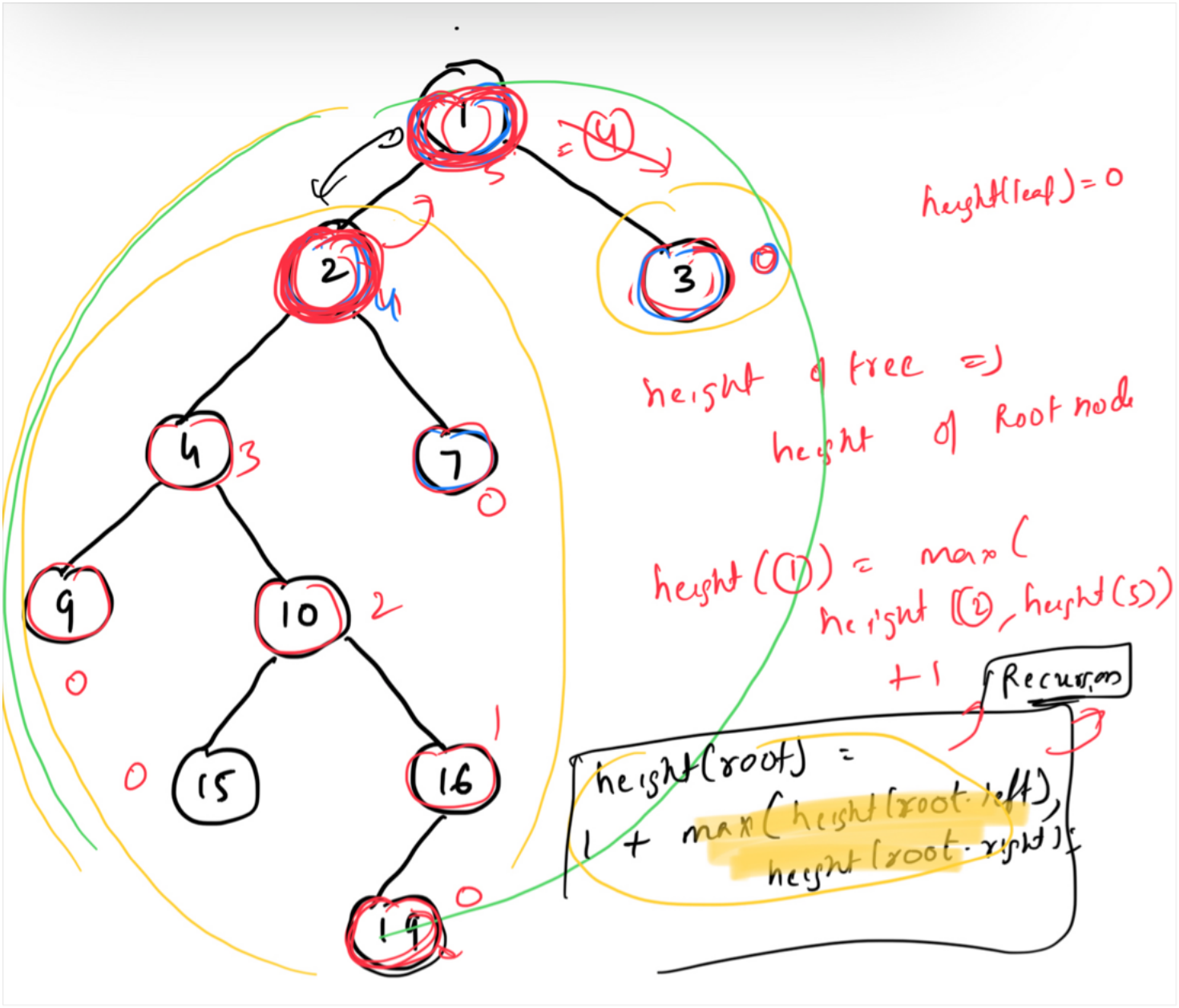
c) Depth of a Node :
Distance (No. of edges) from root to the current node



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

depth(root) = 0

3) Height of Node : Distance (No. of edges) from current node to the farthest Leaf Node.



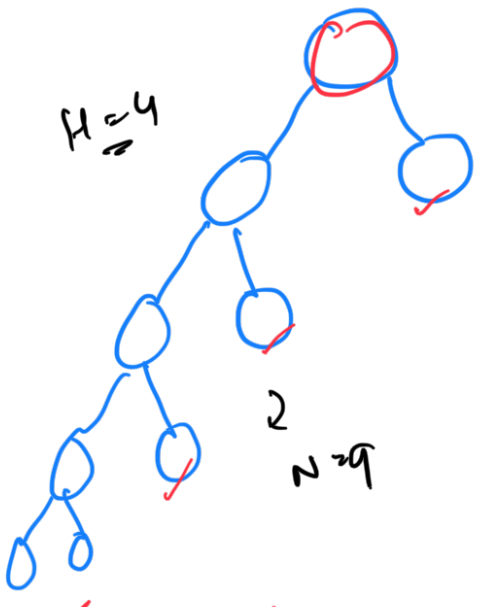
Height of a Tree = Depth of Farthest leaf from Root

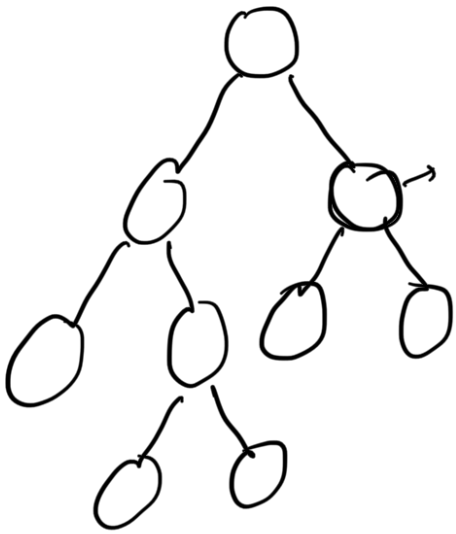
height(node) = 1 + max(height(node.left), height(node.right));

Types:

1) Proper Binary

A Binary Tree where every node has either 0 or 2 children.





Proper Height.

Google

N & 1 =>
= 1 => odd No
0 => Even No

N = N_1 + N_2

Leaf

N_1 : No. of nodes with 0 children

N_2 : No of Nodes with 2 children.

Edges = 0 + $2 \cdot N_2$.

$N_1 + N_2 - 1$

Nodes = Edges + 1

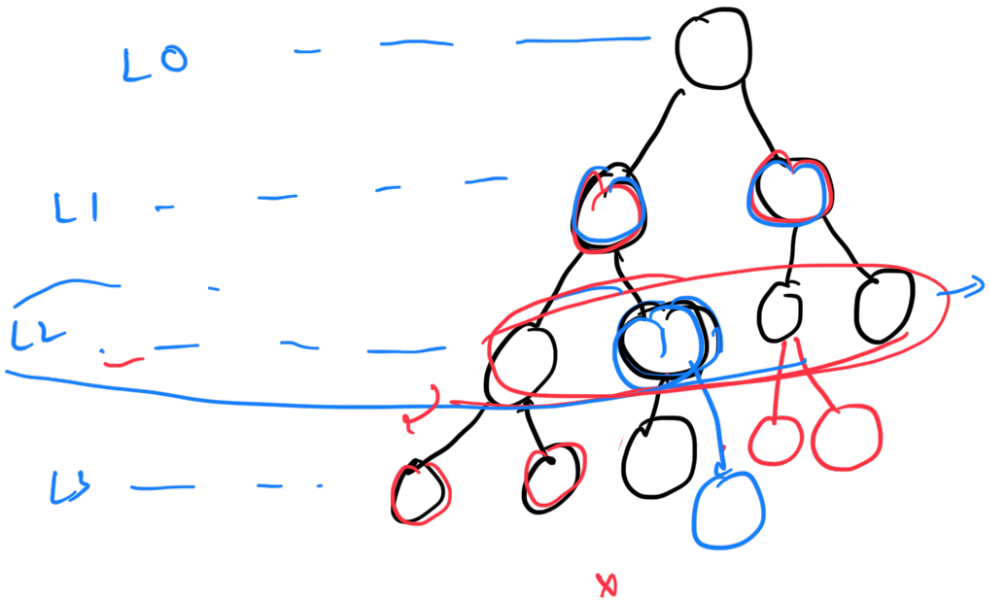
= 2Nₑ + 1 → odd Number

Even

2) Complete Binary:

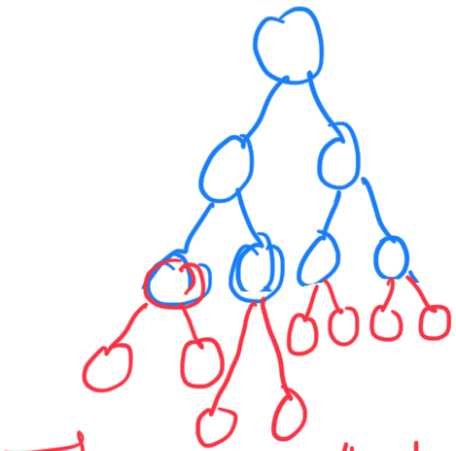
1) Every level is completely filled, except the last level (possibly):

⇒ In the last level, all the nodes are as left as possible.



Completely Filled

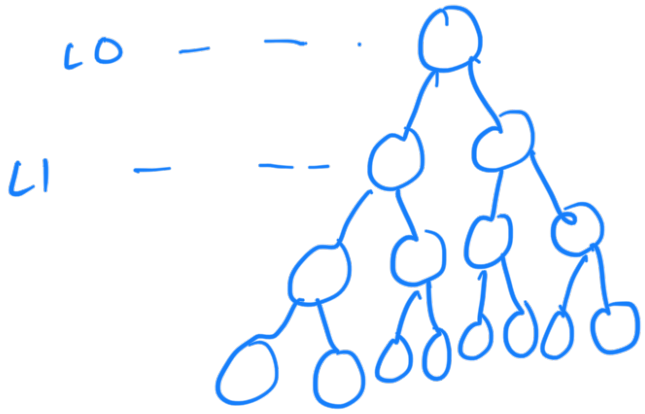
X



→ Fill from Left to Right

Perfect Binary Tree:

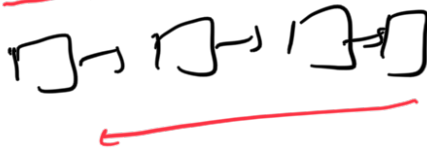
All the levels are completely filled



L(x) = k nodes

L(x+1) = 2k nodes

Linear Array



Array / LL

Traversals:

the process of visiting every node in any D.S

- Graphs
  - DFS
  - BFS

1) Preorder
2) Inorder Traversal
3) Post order
4) Level Order

1), 2), 3) → DFS

4) → BFS

Preorder

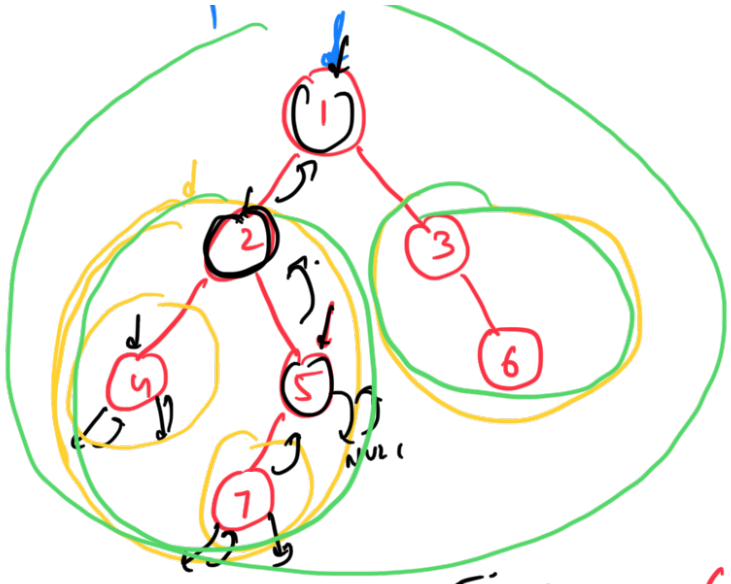
root, Left, Right

Inorder

Left, Root, Right

Postorder

Left, Right, Root



Recursion

Preorder: 1 2 4 5 7 3 6

Inorder:

Postorder:

Recursion

1) Assumption:
preorder (root) will print the preorder traversal

2) Main Logic:

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

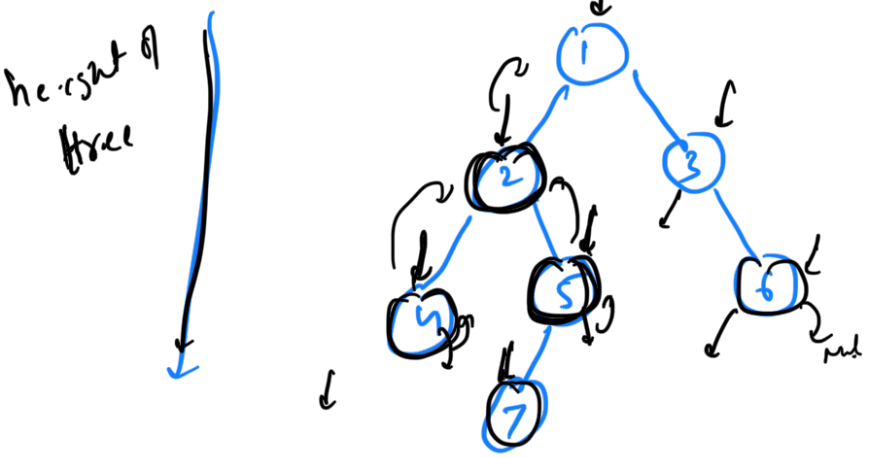
3) Base Case:

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

```
void preorder(root) {
    if(root == NULL) return;

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

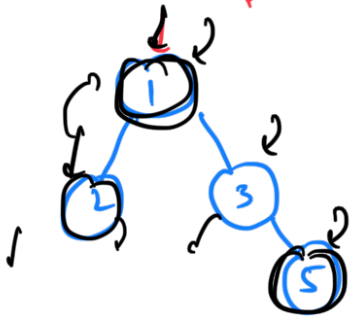
```
void Inorder (root) {
    if(root == NULL) return;
    inorder (root.left);
    print (root.data);
    inorder (root.right);
}

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

Recursion Stack



Stack:

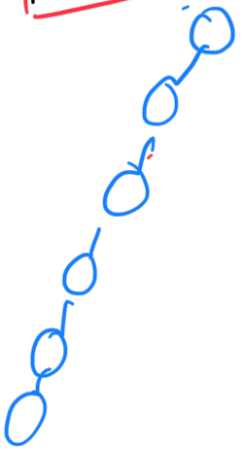
Inorder: 4 2 7 5 1 3 6



Postorder: 2 5 3 1

T.C: O(N)

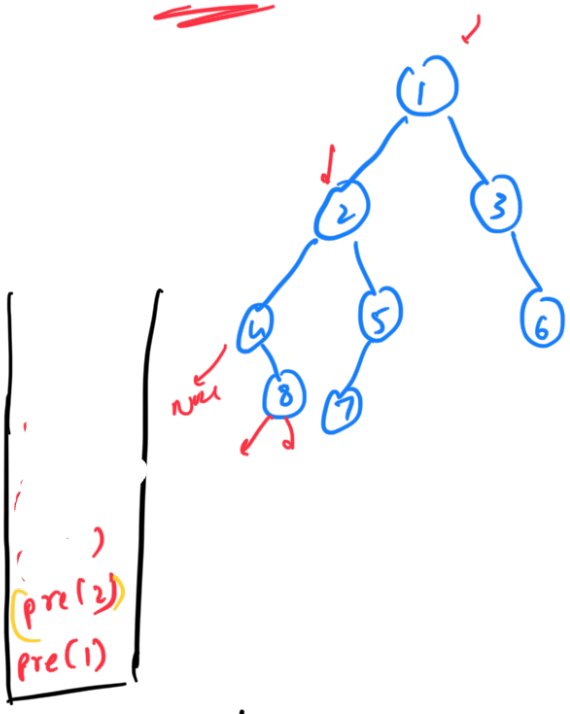
S.C: O(H)



N=

H = N-1

Iterative:



Recursion stack.

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

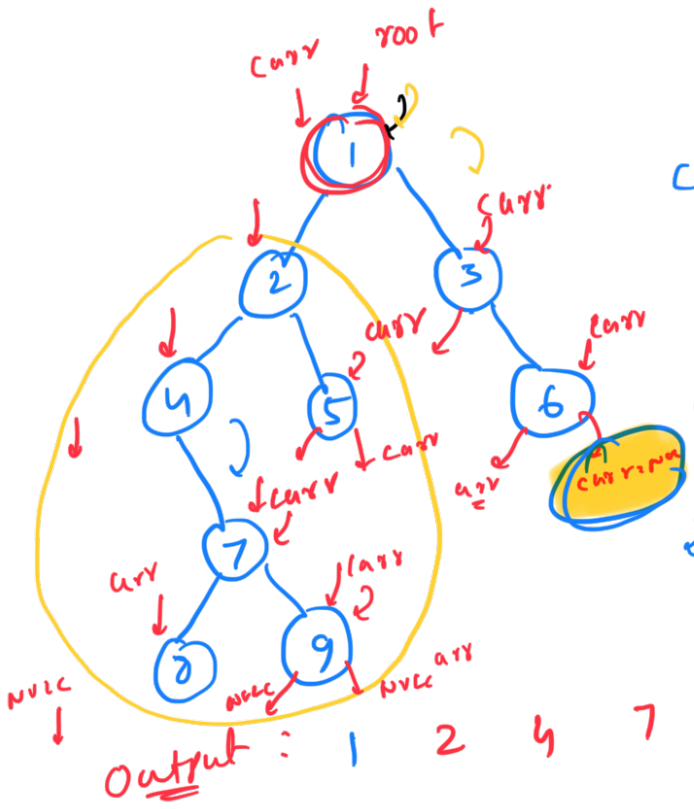
→ 2
→ 3
4

Output: 1 2 4 8 5 7

Iteration:

Stack

Stack



curr = root.

curr = curr.left

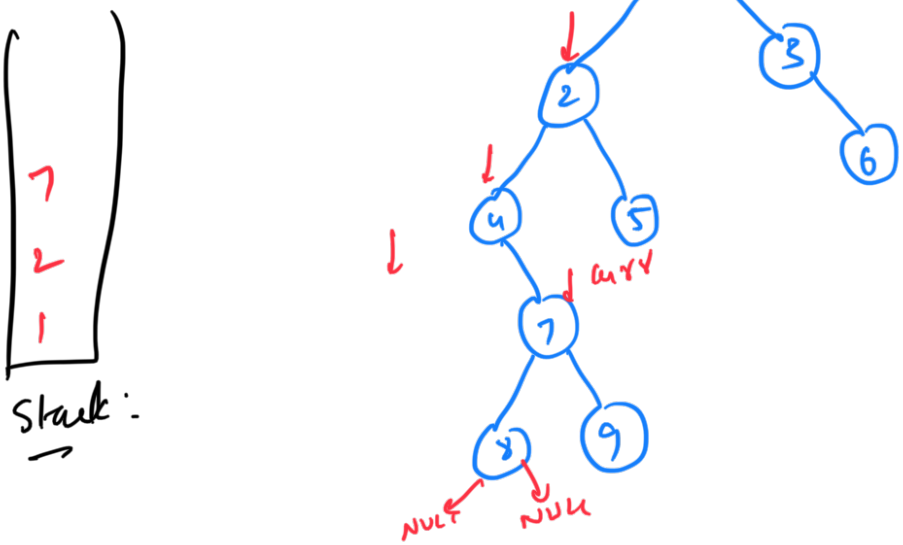
curr = top.right;

curr = 7.right

Output: 1 2 4 7 8 9 5 3 6

```
curr = root;
while ( !stk.empty() || curr != NULL ) {
    if (curr != NULL) {
```

```
        print ( curr.data);
        stk.push(curr);
        curr = curr.left;

    }

    else {
        top = stk.top();
        stk.pop();
        curr = top.right;

    }
}
```



Output: 4 _ 8

Postorder:

Preorder:

Root Left Right → Stack

Postorder:

Left Right Root

Reverse(Post order) =

Root Right Left

root = root.right.

1) Find Reverse of Post order
2) Print Reverse (ans)

2)

TC : O(n)

S.C : O(H)

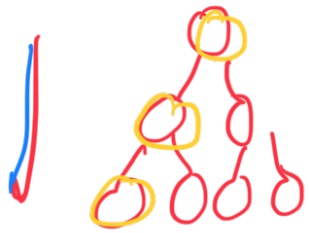
H ⇒ O(n)

⇒ Doubt

When to use Preorder / Postorder?

Preorder: First root is visited.

Control is going from top to bottom

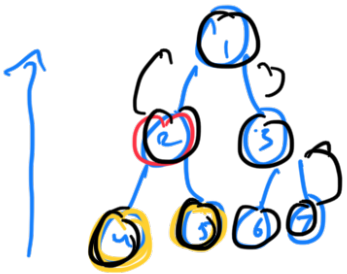


1) If we don't need info of Left & Right Subtree

2) Apply fast-fast method

Postorder:

Control is from Bottom to Top



Question: Find Sum of nodes in a B.T

sum(3) = 3 + sum(2) + sum(5)


```
int sum(root) {
    if(root == NULL) return 0;
    return root.data + sum(root.left) + sum(root.right);
}
```

3 + 2 + 5 + 6 + 2 + 8 + -1

Postorder Traversal

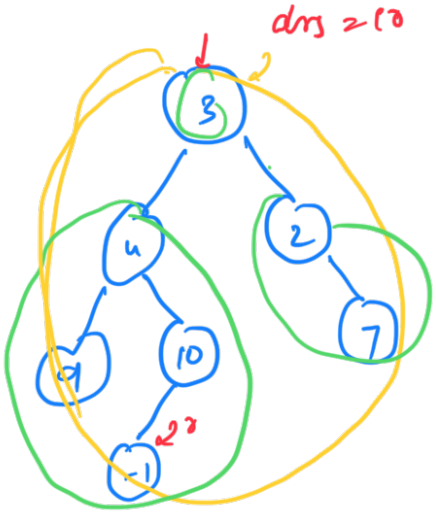
sum = 0;

Inorder,
Preorder,
Postorder

sum += root.data

Question: max value in a B.T.

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



Postorder

max(-1, 0, 0) = 0

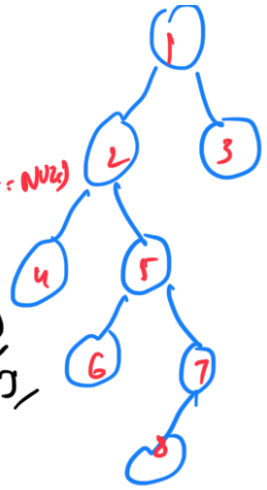
H = 4

Tree

Question: Height

```
int height(root) {
    if(root == NULL) return -1;
    if(root.left == NULL && root.right == NULL) return 0;
    return 1 + max(height(root.left), height(root.right));
}
```

(OR)



1 + max(0,0) = 1

Post order

1 + max(0,0)

H = 1

Count

```
count(root) = 1 + count(root.left) + count(right);
if(root == NULL) return 0;
```

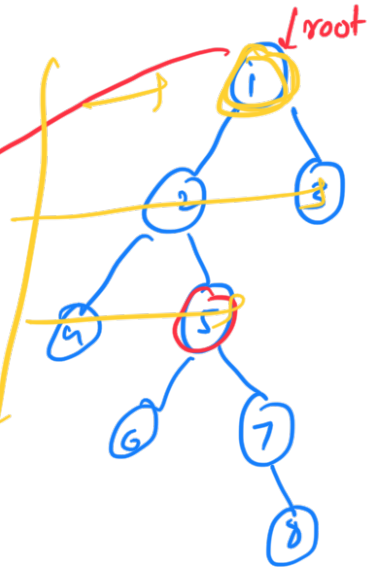
Question: Depth of node

depth(parent) = d

depth(children) = d+1

depth(root) = 0

depth(1, 0)




```
void depth(root, depth) {
    if(root == NULL) return;
    root.depth = depth;
    depth + 1)
```

Node {

Preorder

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

int data
Node left, right,
int depth = 0;

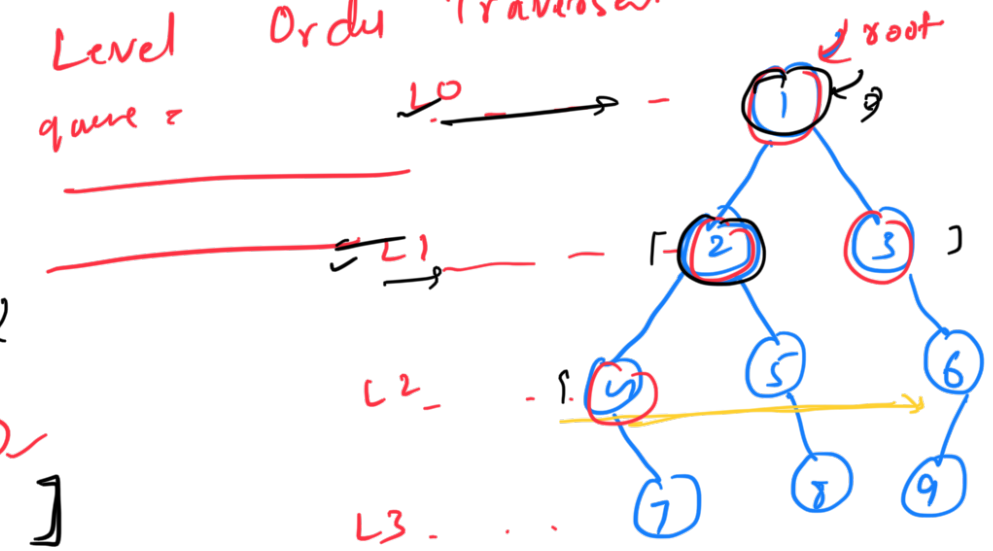
Level Order Traversal

Question:

2D array

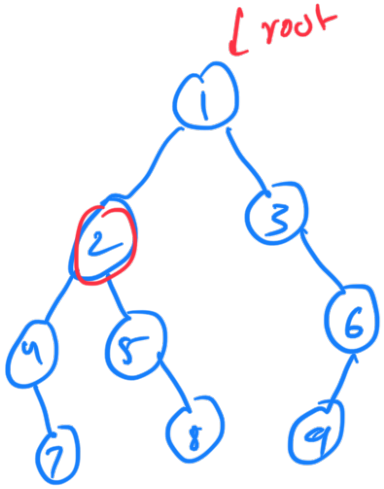
[[1] ✓
[2, 3] ✓
[4, 5, 6] ✓
[7, 8, 9]]

queue =



Level Order: 1 2 3 4 5 6

First In First Out



currDepth = 3

queue

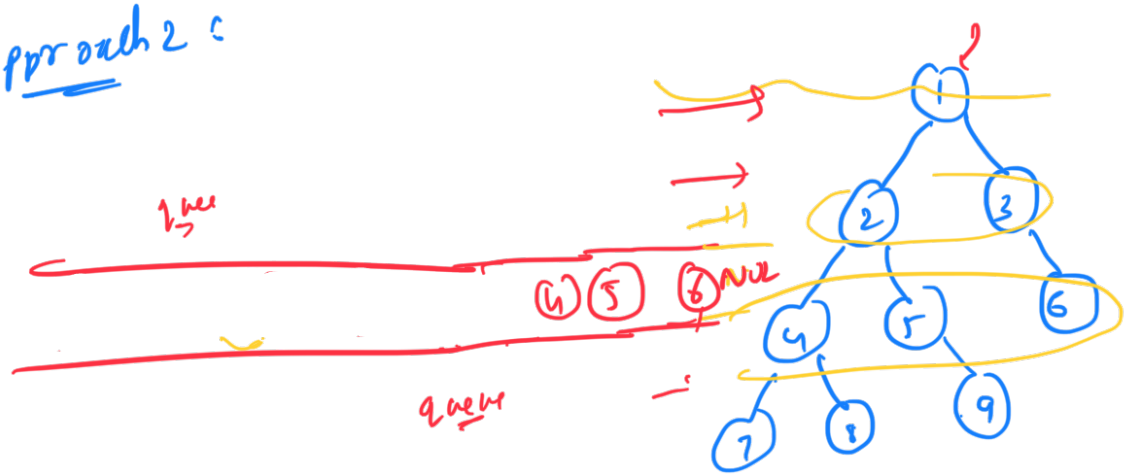
[7, 3] [8, 3] [9, 3]

[1]
[2, 3]
[4, 5, 6]
[7, 8, 9]

Approach:

Maintain a queue of <Node, depth>

Approach 2 :



-> 1

-> 2 3

-> [ ]

Vector<Node>

Hashmap <int, ?>

Hashmap :

0 : [1, 5, 6]

-1 : [2,

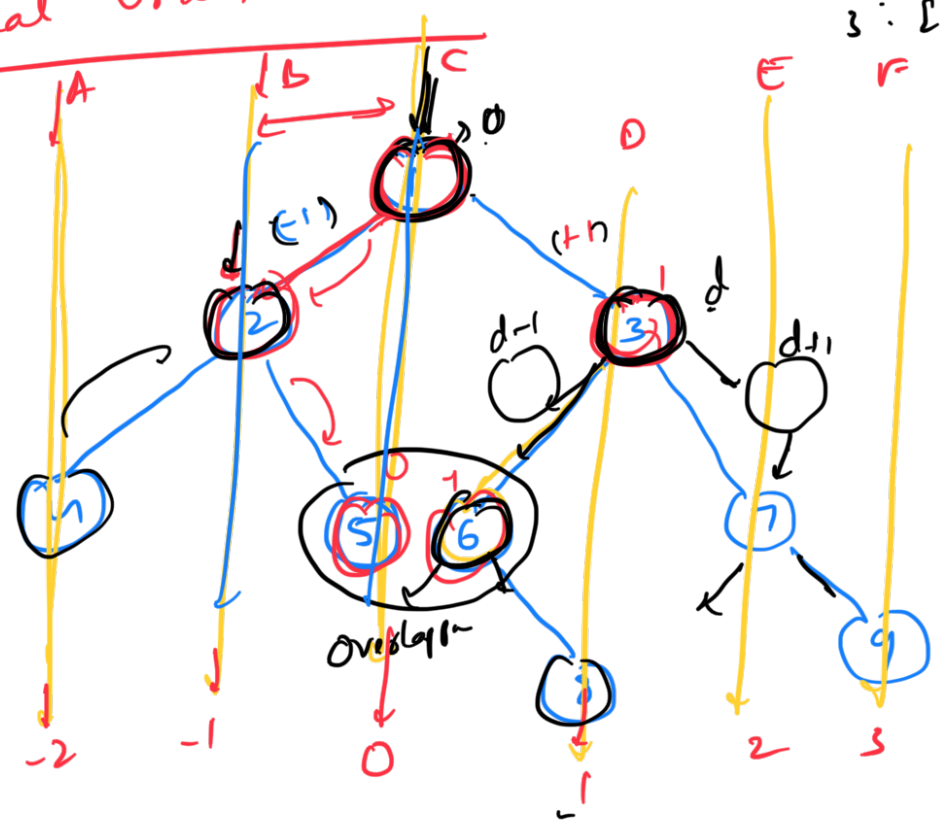
-2 : [4,

1 : [3, 8]

2 : [7]

3 : [9]

Vertical Order Traversal



-> 4 => A

-> 2 => B

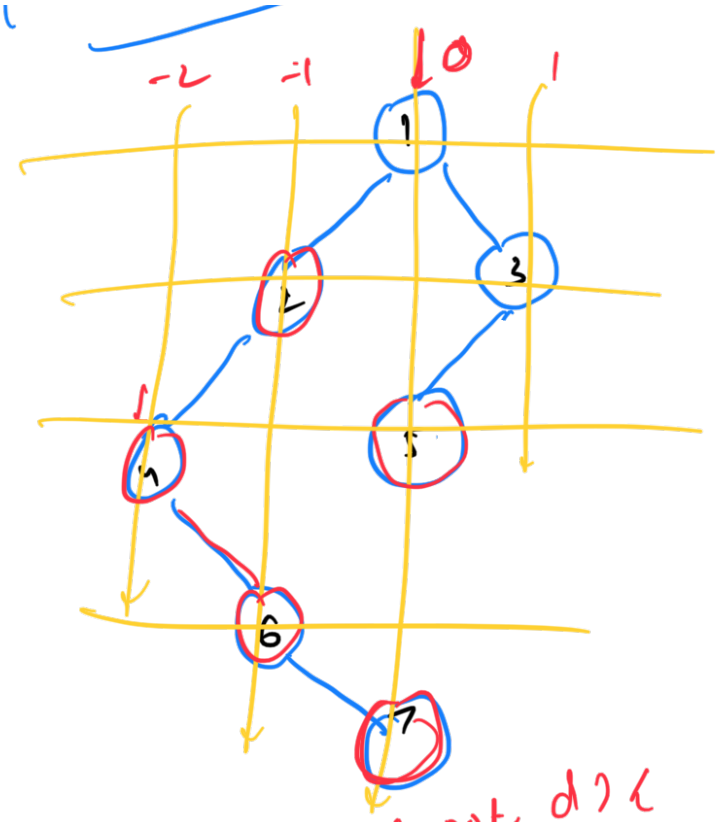
-> 1 5 6 => C

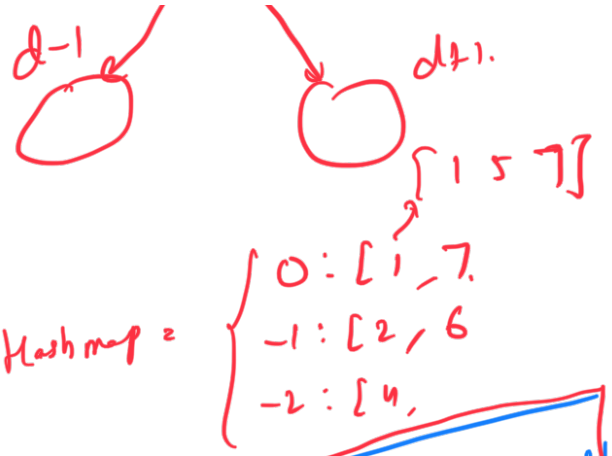
-> 3 8 => D

-> 7 => E

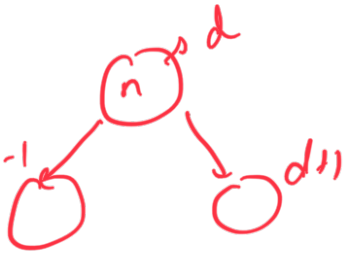
-> 9 => F

preorder: root, Left, Right





Level Order Traversal



verticalOrder (root, d) {

verticalOrder (root.left, d-1)

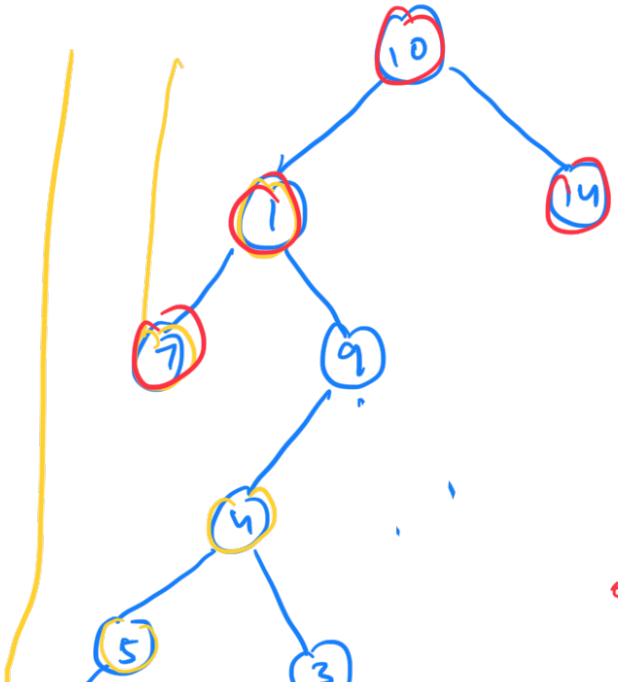
verticalOrder (root.right, d+1)

(left, d-1) (Right, d+1)

}

Views of a Binary Tree

First Node of every level is level order

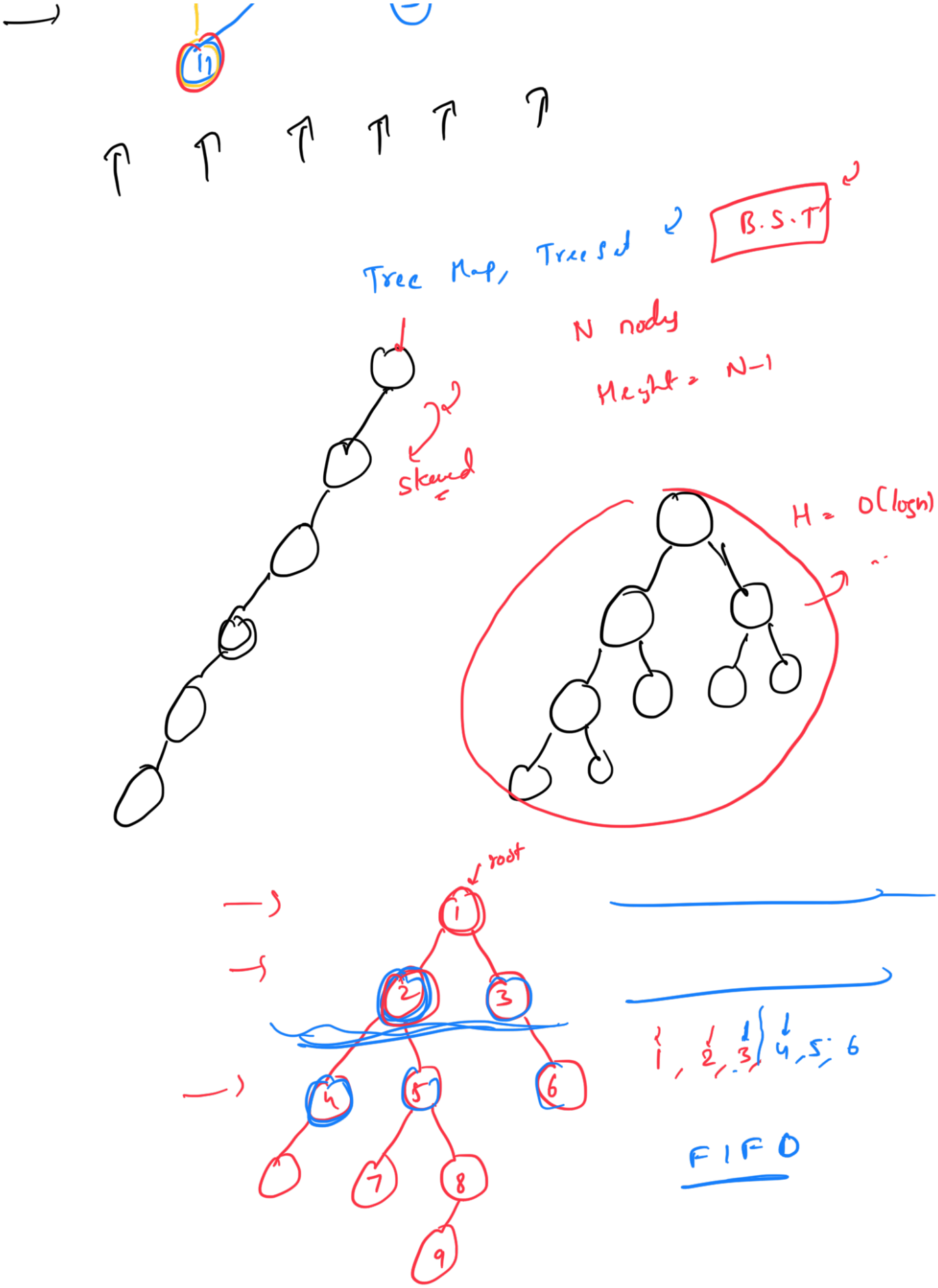


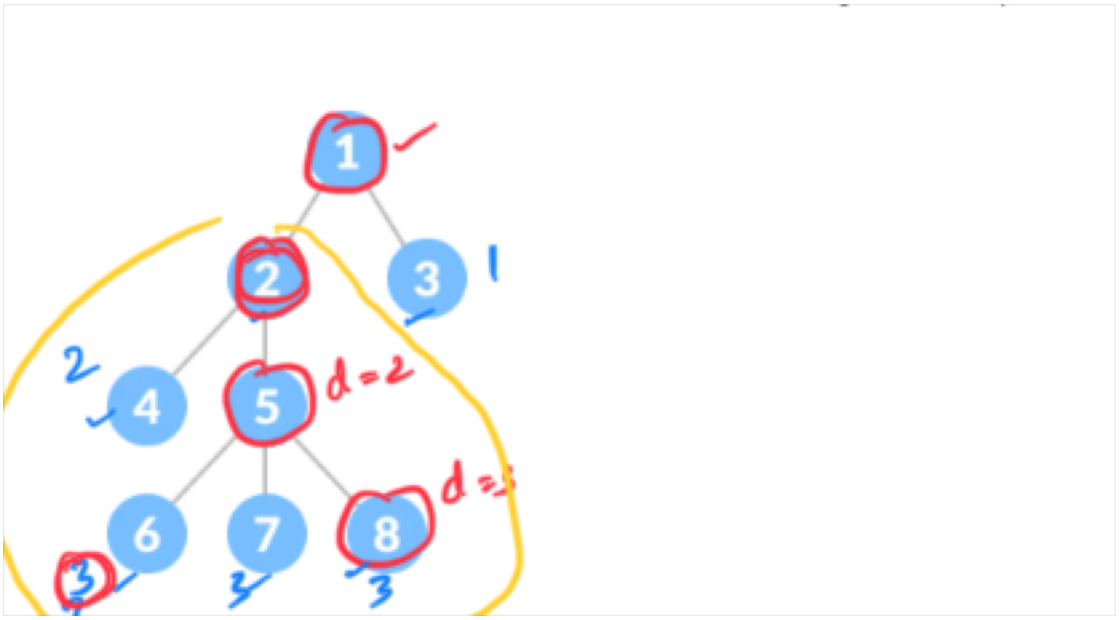
11 7 1 10 14

map<int, vector<Node>>

map<int, Node>

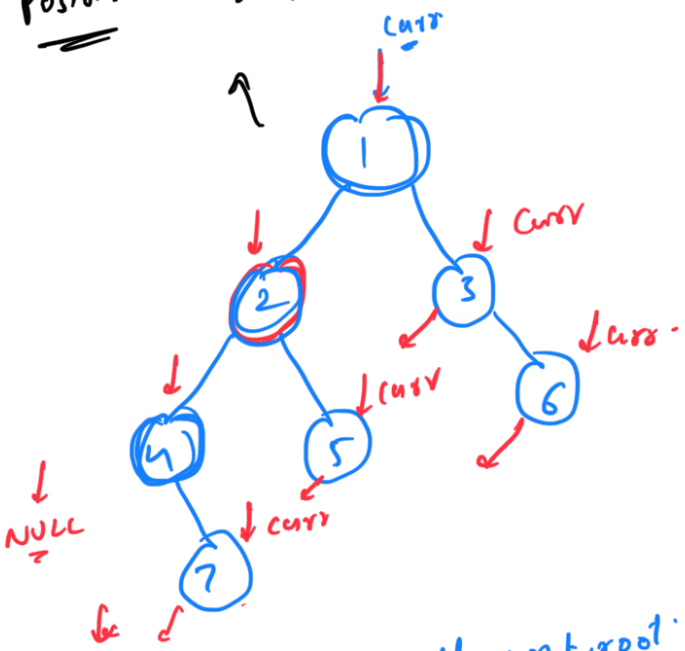
Last node of a level level order Traversal

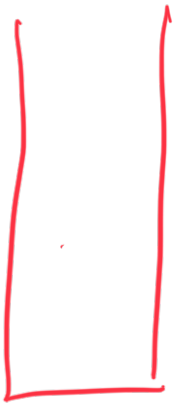

Tree Map, Tree Set
B.S.T
N nodes
Height = N-1
skewed
H = O(logn)
root
1
2
3
4
5
6
7
8
9
{1, 2, 3} {4, 5, 6}
FIFO



Postorder Traversal

Postorder: 7 4 5 2 6 3 1





Stack.

Output: 7 4 5 2 6 3 1

Postorder: Left, right, root.

curr = stk.top().right;

curr = 5

Vertical Order Traversal

Do it using level order traversal to preserve the order in vertical order traversal.

```
verticalOrder(Node* root, int dist){
      if(root == NULL) return;

      // Preorder traversal
      mp[dist].push_back(root->data);
      verticalOrder(root->left, dist - 1);
      verticalOrder(root->right, dist + 1);
}
print(map);
```

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);
		}
	}
}
```