Todays Content:

- → Binary Tree
- → Traversal
- → Sizes

} 45-50

- → Iterative traversals
- → level order
- → Invert BT

Tree Basics:

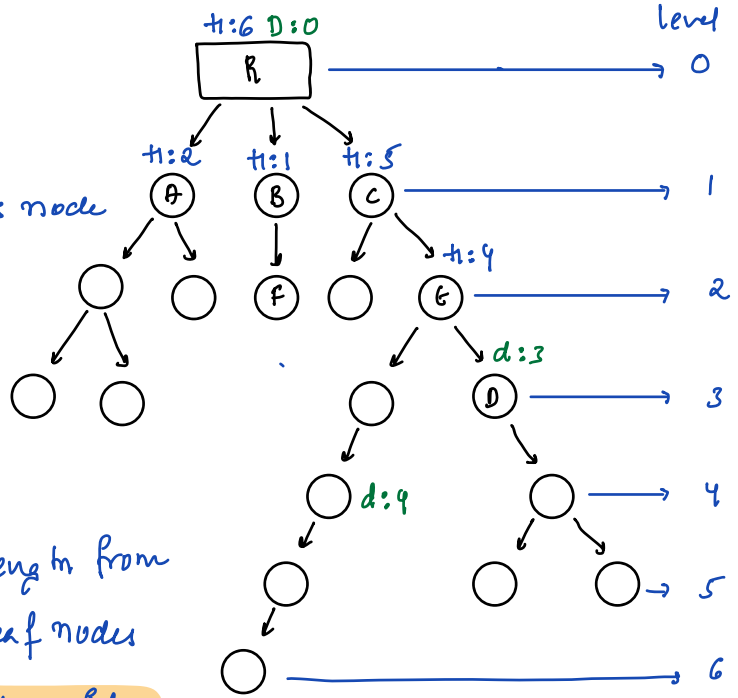
○ : node

→ : Edge, connects node

parent(F): B

Ancestor(D): E C R

Eg E are : nodes at same level



Height(Node): max length from node to any of it's leaf nodes

length calculated based on Edges

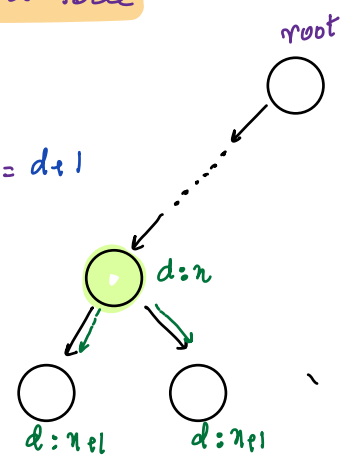
Height(Node) = 1+ max(Child node height)

Height(leaf) = 0

Depth(Node) = length of path from root node to given node

Depth(root) = 0

Depth(node) = level of a node



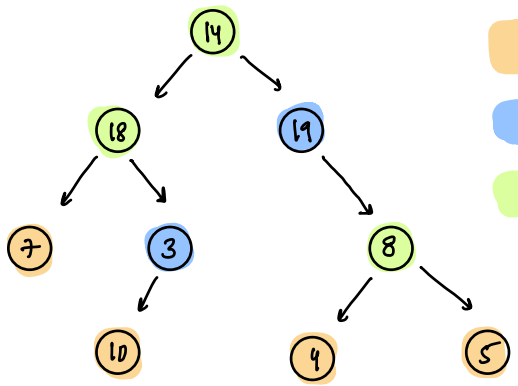
Depth(node) = d

Depth of it's child node = d+1

Obs: Root node is only node without a parent

Binary Tree: Any node can at max have 2 child nodes

0 1 2 child



→ node, with 0 child, leaf

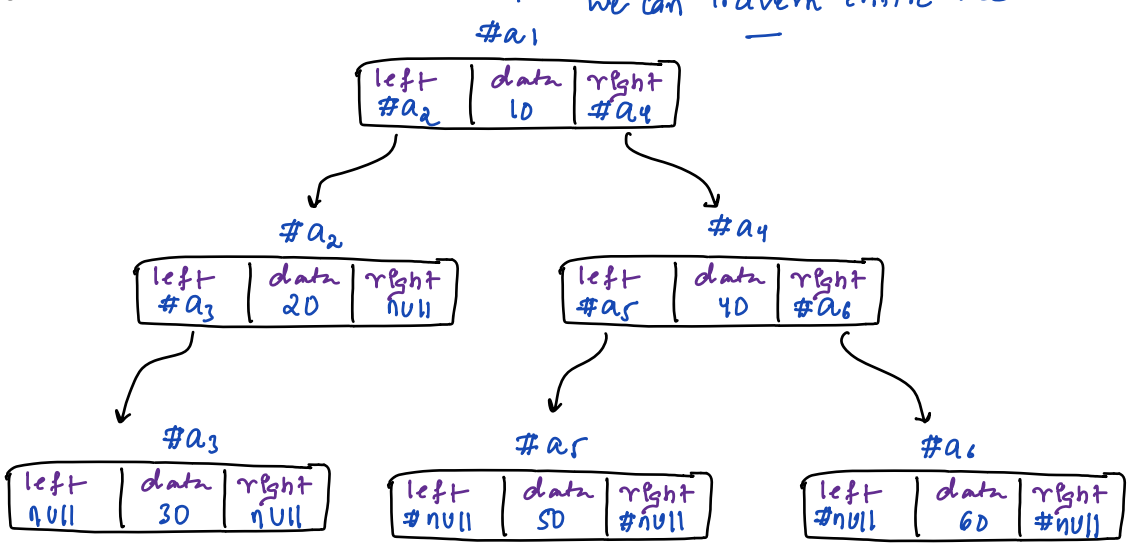
→ nodes, with 1 child

→ node, with 2 child

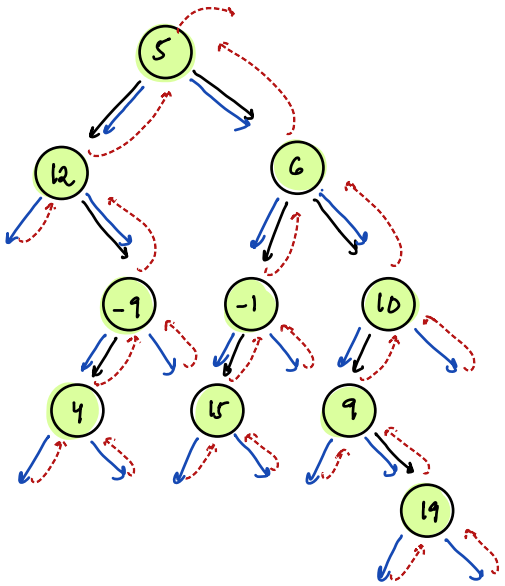
```
Class Node {
    int data
    Node left
    Node right
    Node(x) { // constructor: To initialize data members of class
        data = x
        left = null
        right = null
    }
}
```

obj reference can hold address of node object

// Only root node given, with that we can traverse entire tree



Tree Traversals :



preorder : D Left Right

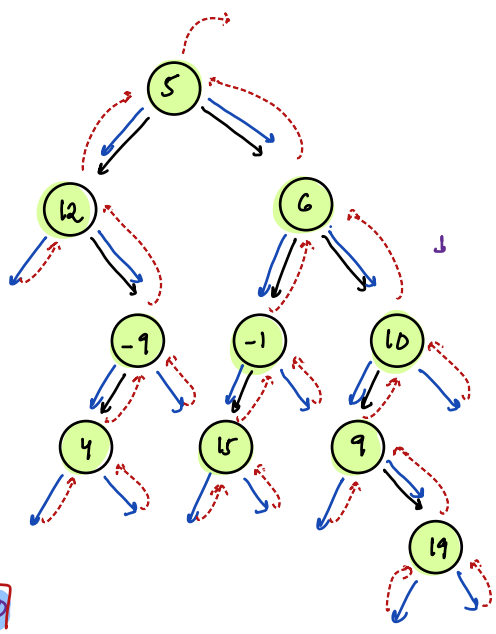
→ print node data

→ Go to left subtree
print entire left subtree in preorder

→ Go to right subtree
print entire right subtree in preorder

root | pre LST | pre RST

preorder: 5 | 12 -9 4 | 6 -1 15 10 9 19



inorder : left data right

→ Go to left subtree
print entire left subtree in inorder

→ print node data

→ Go to right subtree
print entire right subtree in inorder

in LST | root | in RST

inorder: 12 4 -9 | 5 | 15 -1 6 9 19 10

postorder : left right data

→ Go to left subtree
print entire left subtree in postorder

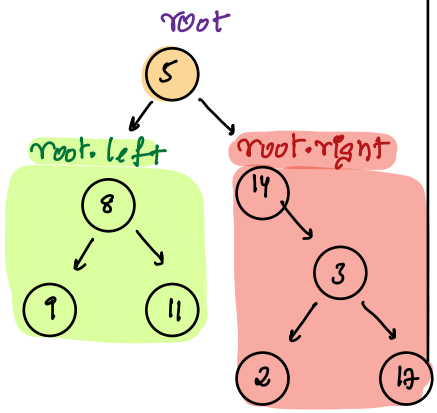
→ Go to right subtree
print entire right subtree in postorder

→ print node data

postorder : TODO for above example

Ass: Given root node, print entire tree in preorder

```
void preorder(Node root){
    1. if(root==null){return;}
    2. print(root.data)
    3. preorder(root.left)
    4. preorder(root.right)
}
```



preorder: 1 2 3 4 // Data left Right

inorder: 1 3 2 4 // left Data Right

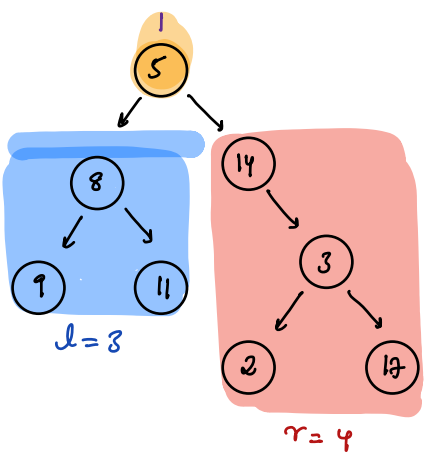
postorder: 1 3 4 2 // left right Data

Ass: Given a root node, return no: of nodes in Tree

```
int size(Node root){ TC: O(N) SC: O(H)
    if(root==null){return 0;}
    l = size(root.left)
    r = size(root.right)
    return l+r+1
}
```

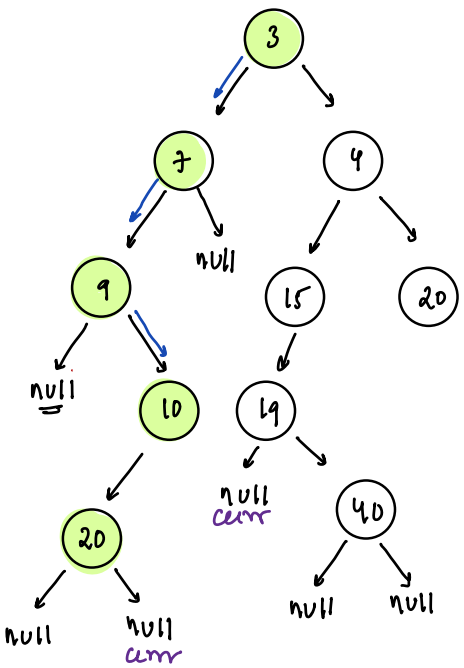
// at max stack size = H

Height of Tree



Inorder (Iteratively)

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

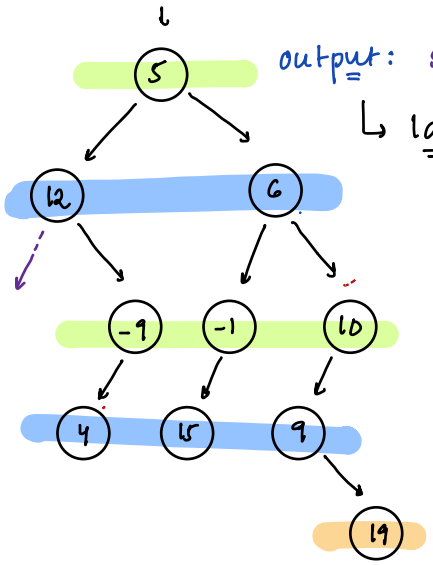



```
void inorder(Node root){
    Node curr = root
    Stack<Node> st; // storing address
    while(st.size()>0 || curr!=NULL){
        → curr = 4
        while(curr!=null){
            st.push(curr)
            curr = curr.left
        }
        curr = st.top() → curr = 19
        st.pop(); //deleting from stack
        print(curr.data)
        curr = curr.right → curr = 40
    }
}
```

Inorder: 9 20 10 7 3 19 ... soon

preorder / postorder : TODO / Iterative with Stack

level order traversal : level by level, left to right



output: 5 12 6 -9 -1 10 4 15 9 19

↳ Idea: get front ele in que, print it, pop it & push it's children, repeat process till que is empty

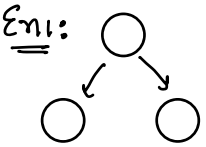
front ↓ rear

Idea: ~~5~~ | ~~12 6~~ | ~~-9 -1 10~~ | ~~4 15~~ | ~~9 19~~ : q is empty stop

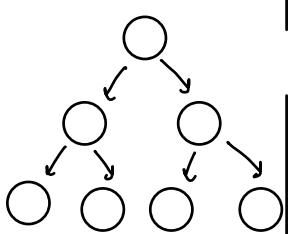
output: 5 12 6 -9 -1 10 4 15 9 19

```
void levelorder(Node root){ TC: O(N)  SC: max nodes in a level ?
                                          ↳ = (N+1)/2 = O(N)
    if(root==null){return}
    Queue<Node> q
    q.enque(root)
    while( q.size()>0 ){
        Node f = q.front();
        q.deque()
        print(f.data);
        if( f.left != NULL){
            q.enque( f.left)
        }
        if( f.right != NULL){
            q.enque( f.right)
        }
    }
}
```

Ex1:

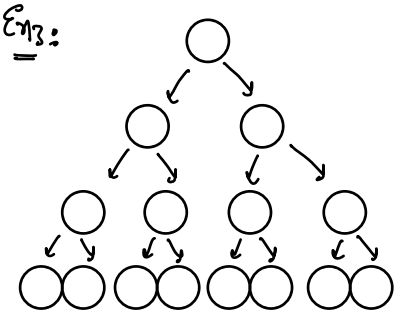


N = 3 M = 2



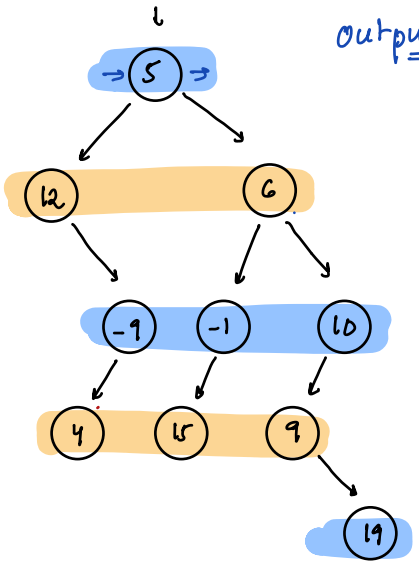
N = 7 M = 4

Ex3:



N = 15, M = 8

level order traversal : 2 → { After every level goto new line }



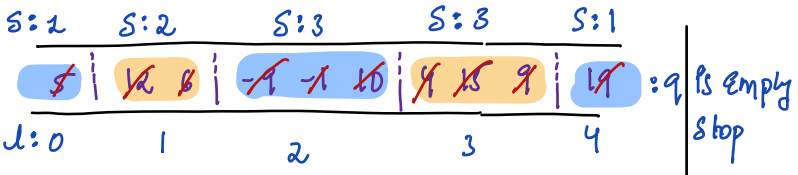
output: → 5 → ideal:

→ 12 6 → ideal:

→ -9 -1 10

→ 4 15 9

→ 19

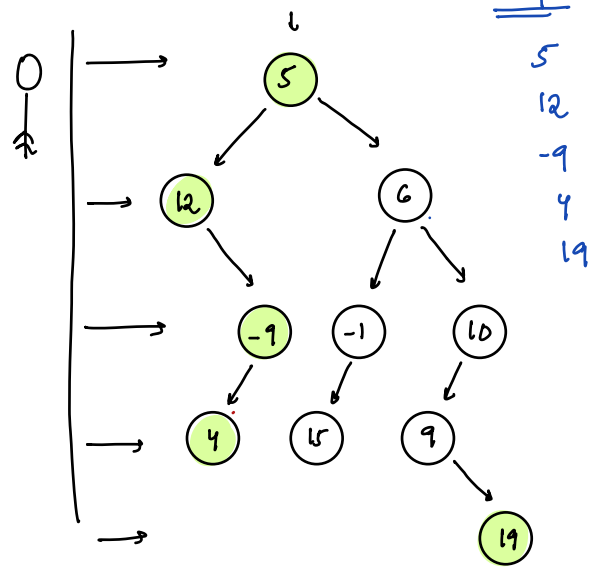


void levelorder(Node root){ TC: O(N) SC: max nodes in a level ?

```
if(root==null){return}
Queue<Node> q
q.enque(root)
while( q.size()>0 ){
    int n = q.size()
    i=1; i<=n; i++){
        Node f = q.front();
        q.deque()
        print(f.data);
        if( f.left != NULL){
            q.enque( f.left)
        }
        if( f.right != NULL){
            q.enque( f.right)
        }
    }
    print("\n") //new line
}
```

size	
1	pop que, push childre 1 time, print(new line)
2	pop que, push childre 2 time print(new line)
3	pop que, push childre 3 time print(new line)
3	pop que, push childre 3 time print(new line)
1	pop que, push childre 1 time print(new line)

left view:



output : 1st node of every level

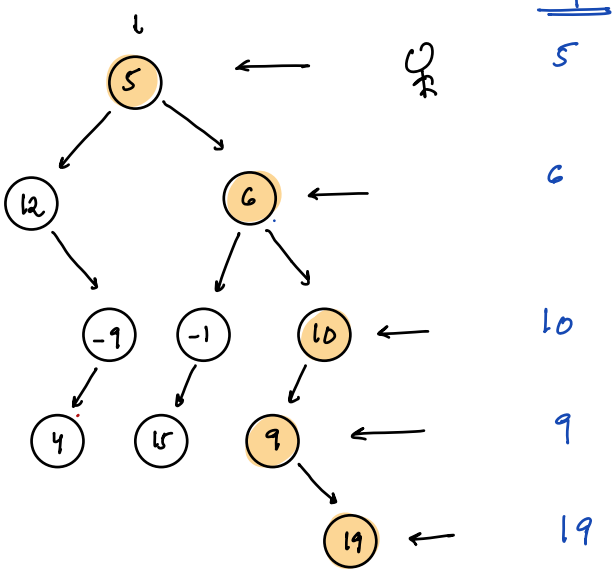
5
12
-9
4
19

void leftview (Node root) { TC: O(N) SC: max nodes in a level ?

```
if (root == null) { return }
Queue<Node> q
q.enque(root)
while ( q.size() > 0 ) {
    int n = q.size()
    i = 1; i <= n; i++ {
        Node f = q.front();
        q.deque()
        if (i == 1) { print(f.data); }   //node will be 1st node
        if ( f.left != NULL) {
            q.enque ( f.left)
        }
        if ( f.right != NULL) {
            q.enque ( f.right)
        }
    }
    print("\n") //new line
}
```

right view:

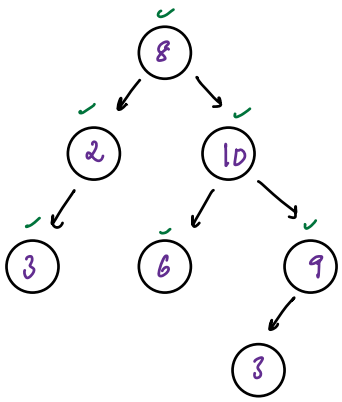
output : last node of every level

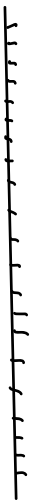


5
6
10
9
19

```
void rightview (Node root) {
    if (root == null) { return }
    Queue<Node> q
    q.enque(root)
    while ( q.size() > 0 ) {
        int n = q.size()
        i = 1; i <= n; i++ ) {
            Node f = q.front();
            q.deque()
            if (i == n) { print(f.data); }   //node will be last node
            if ( f.left != NULL) {
                q.enque ( f.left)
            }
            if ( f.right != NULL) {
                q.enque ( f.right)
            }
        }
        print("\n") //new line
    }
```

Invert BT / Mirror view of BT : TODO?

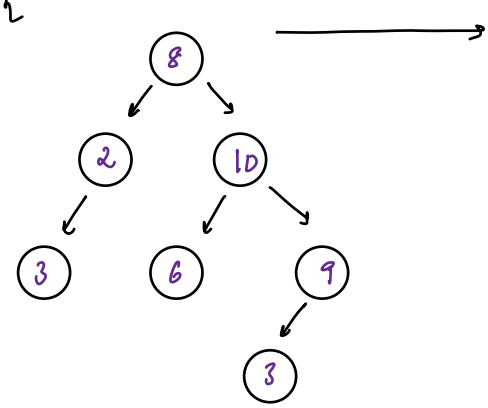




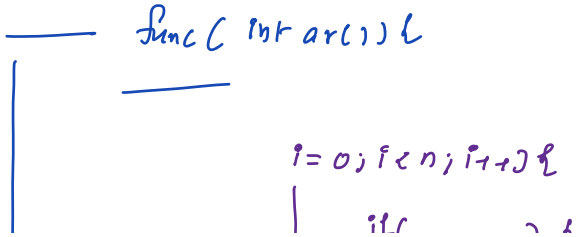
Ideal:

Ass:

void mirror(Node root){



}


func(int ar[]){
i=0; i<n; i++){