Data Structures

arrays, arraylist, hashmap / hashset, LinKed list, Stack, Queue, Trees - 1

Problems → different requirement

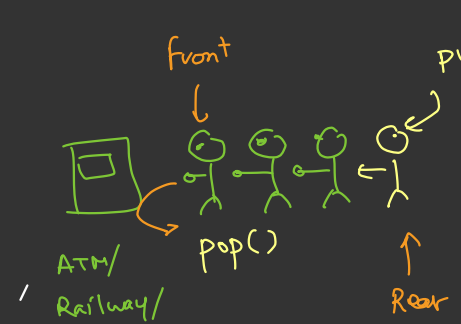
↳ fast look up ⇒ hashmap — unordered

↳ ordered fashion → random access → array / arraylist → ordered

↳ fast ins. at both ends → LinKed list → ordered

↳ LIFO ordering → Stack, → ordered

Queue



ATM/ Railway/ Doctor Visit/ Jobs

Deletion at front → pop / de queue

Insert from Rear → push / en queue

Behaviour

FIFO ordering ⇒ First In First Out

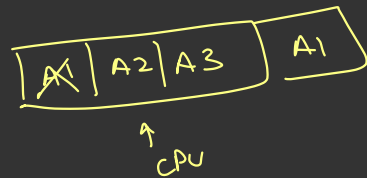
Software level

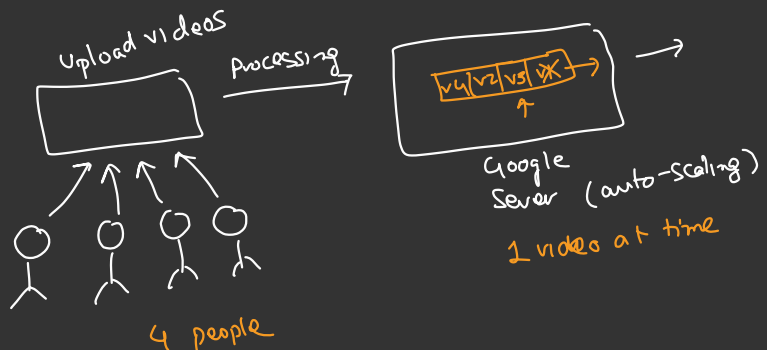
→ Operating systems, process scheduling

→ Servers, Multiple Requests from users (form a Queue) waiting to get served

→ Message Queue

3 AM

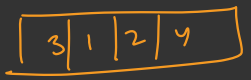


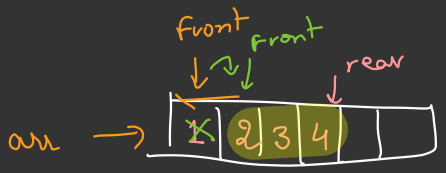


Queue

↓

using an array






```
class Queue {
  arr[N]                          O(1)   proper element   rear = 0,
                                         CS = 0,   N = size
  push()   ← add x to rear
  pop()    ← remove at front
  getFront()  ← read element at front
  empty()
}
```

Method

peek() in Java Collections.

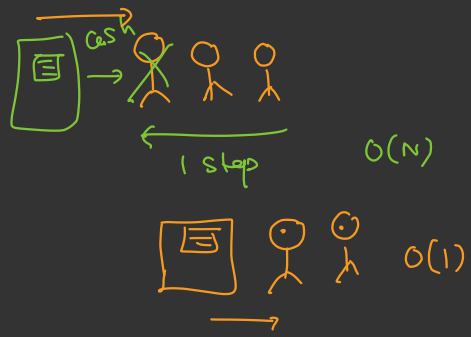
Fixed size array (n)

```
void push(int x) {
    if( is NOT -ull
        arr[rear] = x,
        rear = (rear + 1) % N
}
```

```
void pop() {
    if(q is not empty)
        front = (front + 1) % N
}
```

```
int getFront(){
    return a[front]
}
```

Circular Queue

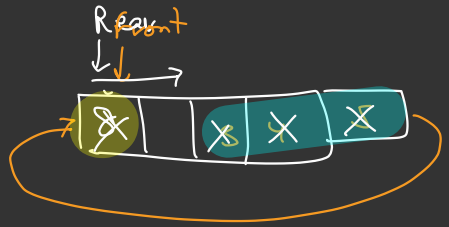


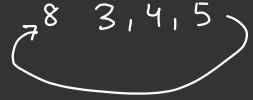
Challenges with above implementation

(1) Limited Size

Cap ⇒ 5
Store ⇒ 3

- push(1)
- push(2)
- push(3)
- push(4)
- push(5)
- push(6) → Full



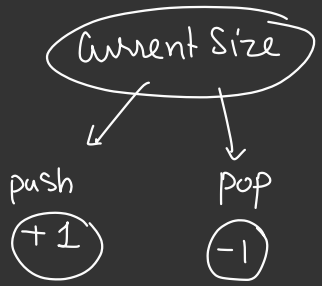


pop()
pop()
push(8)
pop() → 3
pop() → 4

② Maintain front & rear in a circular fashion

pop() → 5

pop → 8



queue is empty

—

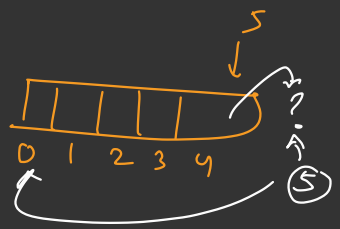
if (cs == 0)

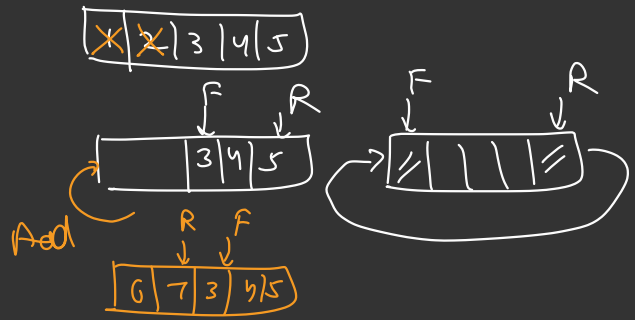
↳ queue is empty

queue can get full

if (cs == arr.size)

↳ Queue is full

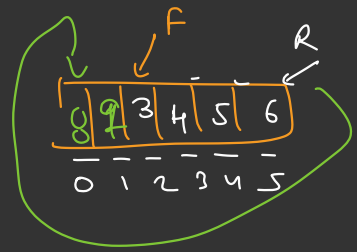




→ rear = (rear + 1) % 5

→ front = (front + 1) % 5

] at the end you will find at index 0.



n = 6

pop()
pop()
push(8)
push(9)

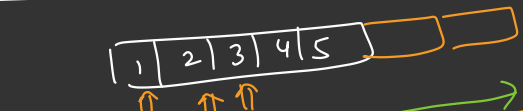
(Rear + 1) % 6

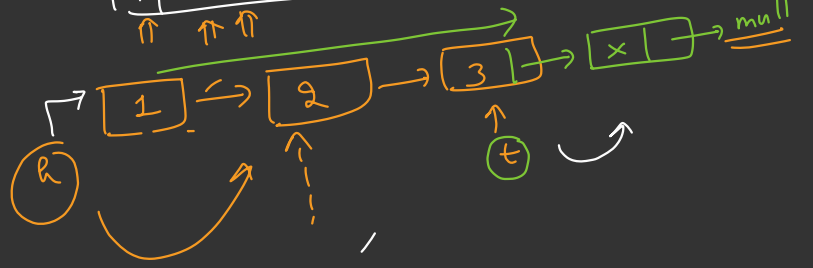
⇒ (5 + 1) % 6 ⇒

⇒ (0 + 1) % 6

Code → done

Queue using linked list





class Node{ 3

Class Queue{

→ Linked list → inbuit in Collection framework

OR → head, tail

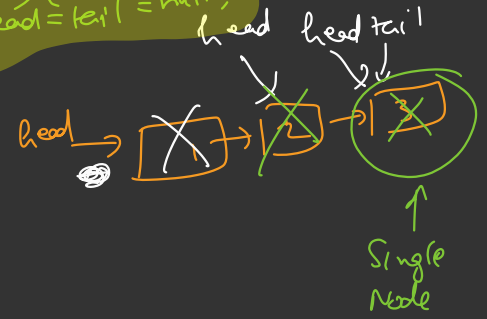

```
push (int x){
  if (head == null){ head = tail = new Node(x); }
  else{
    tail.next = new Node(x)
    tail = tail.next
  }
}
```

empty queue

automatically into next as null

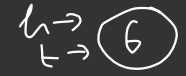
```
if (head != null && head == tail) {
    head = tail = null;
}
```

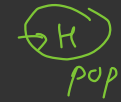
```
pop ( ) {
    if (head != null) {
        head = head.next;
    }
}
```



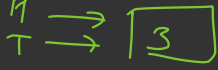
→ null
→ null

```
getFirst() {
    if (head != null) {
        return head.data;
    }
}
```



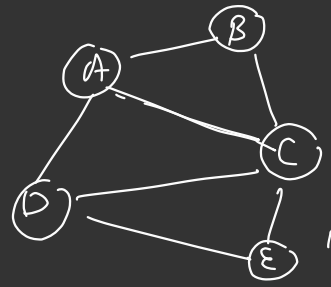


pop
push(=)

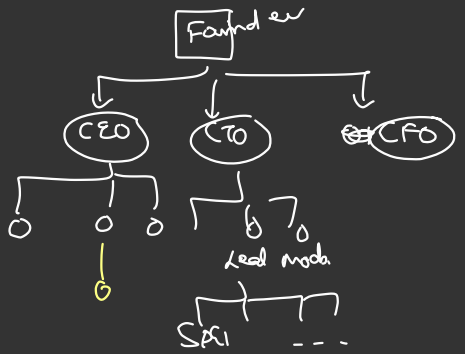


TREES

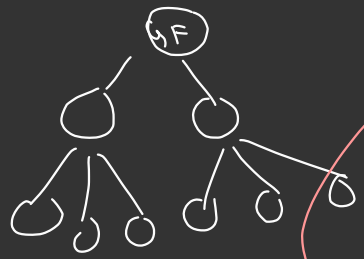
Hierarchical Data Structure



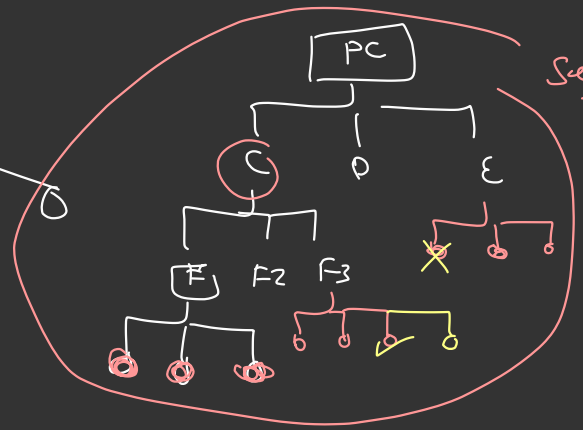
Network & Graph



Family



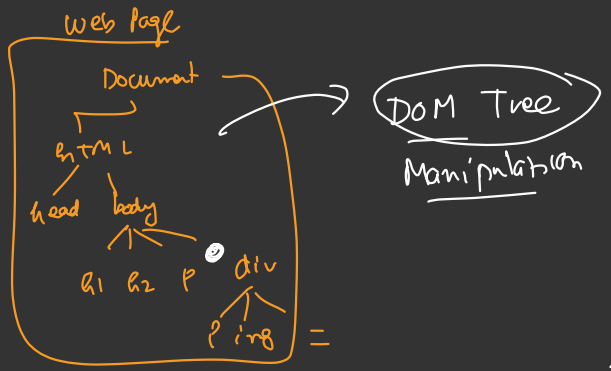
- Directory Structure

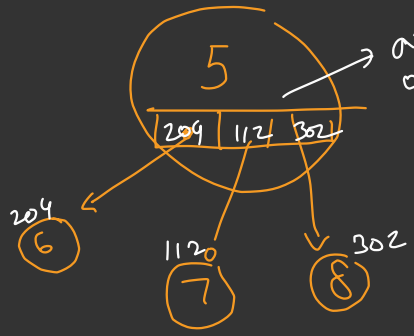


Search

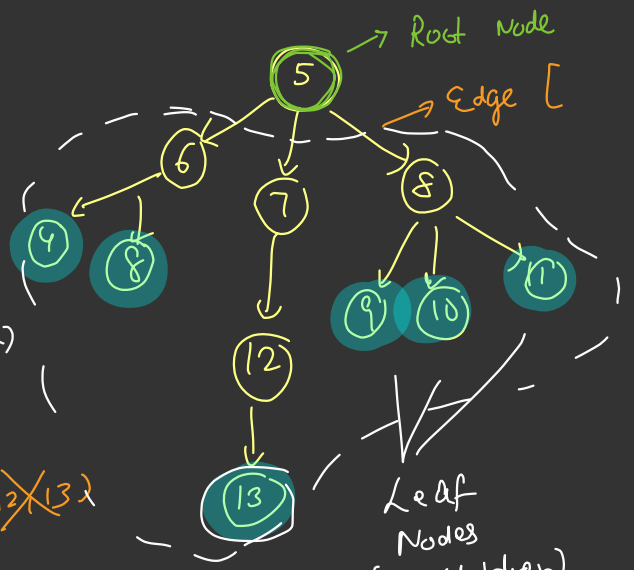
↑

Insertion, Delete





Tree / Generic Tree



5 → parent of 6
6 → child of 5

sibling
↓
nodes with same parent
(6, 7, 8)
(9, 10, 1)
(9, 8)

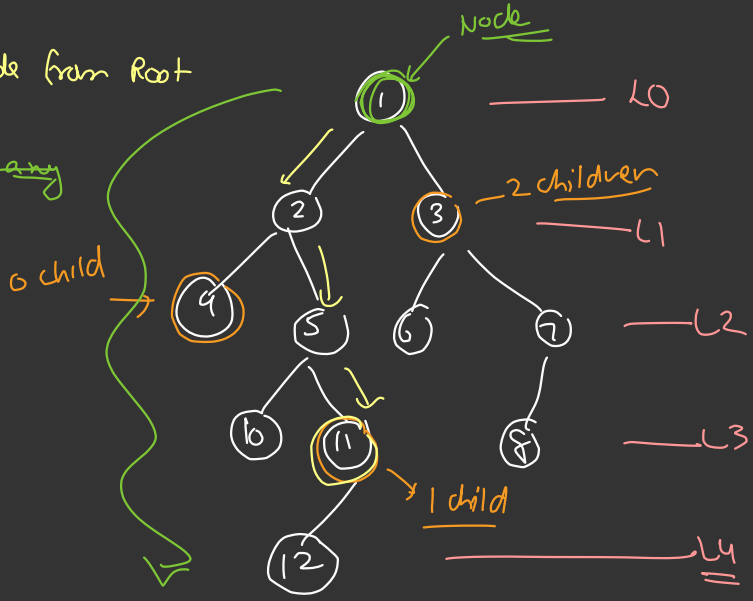
Ancestors $^{13}\rightarrow$ (12, 7, 5)

Descendants $^{5}\rightarrow$ All except 5

→ All nodes have single parent except Root Node

Depth of Node – Distance of node from Root

⇒ Height of Tree – Max Depth of ~~any~~ Leaf Node



Depth(2) = 1

Depth(11) = 3

Depth(1) = 0

Height(Tree) = [4 ← Edges]

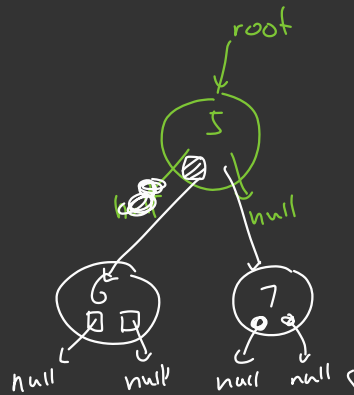
Max Depth ⇒ 4

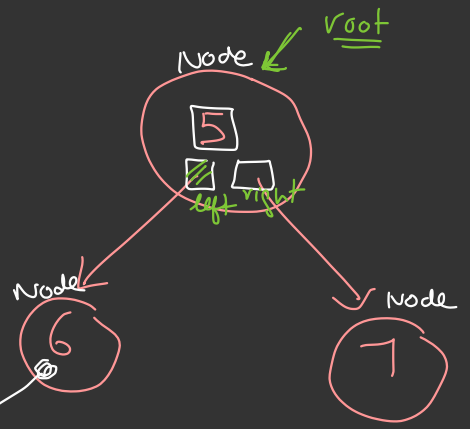
5 ← Nodes

Binary Tree → Max 2 children for every Node

1. ~~Build Tree~~

2. ~~Print / Traversal~~





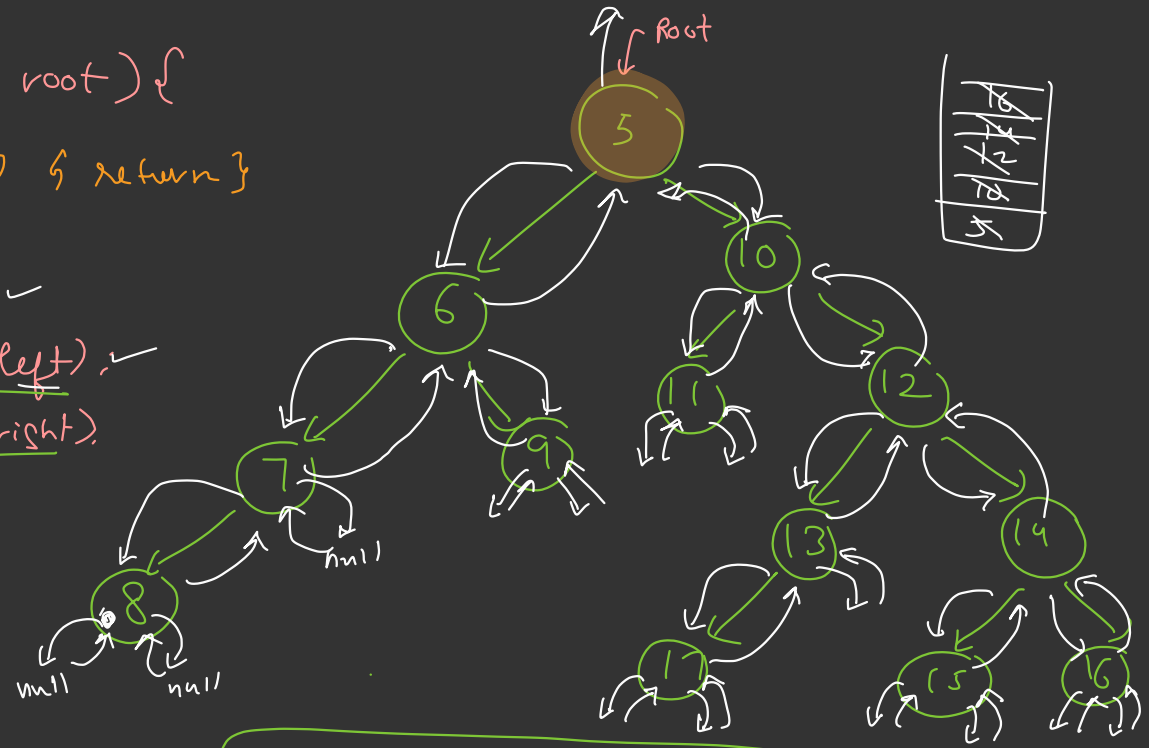

```
class Node {

    int data,

    Node Left,

    Node Right,

    Node (int d) {
        data = d,
        left = right = null,
    }

}
```

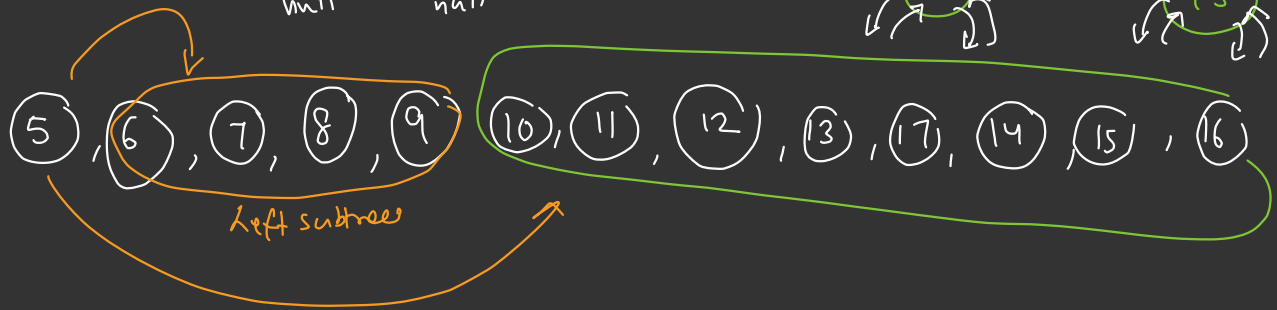

```
Node root = new Node(5),

root.left = new Node(6);

root.right = new Node(7);
```

↓

Tree with 3 Nodes

```
void preorder ( Node root ) {
    L1    if (root == null) { return }
    L2 →  Sout (Node)
    L3 →  preorder (root.left)
    L4 →  preorder (root.right)
}
```





3 popular Traversals

↳ Preorder

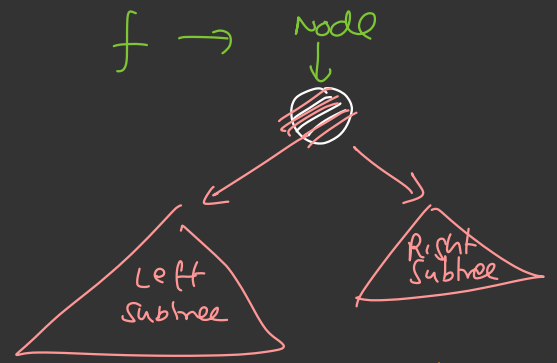
Node
Left
Right

↳ Inorder

Left
Node
Right

↳ Post order

Left
Right
Node



f(root){ → if(root == NULL) { Return }

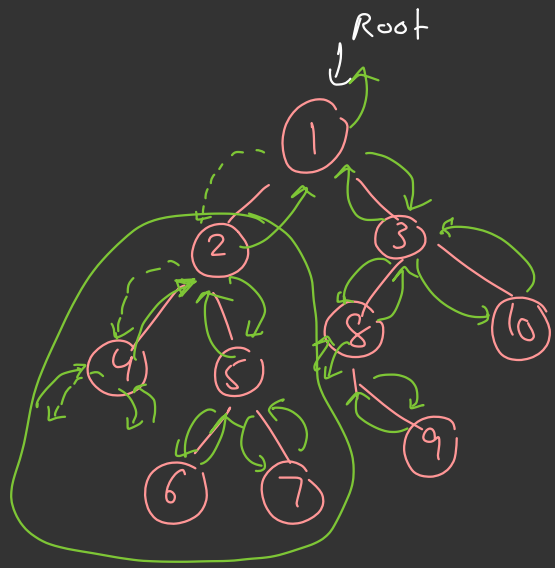
Sout(Node → data)

Two Rec Calls. → [f(root left)
f(root right)

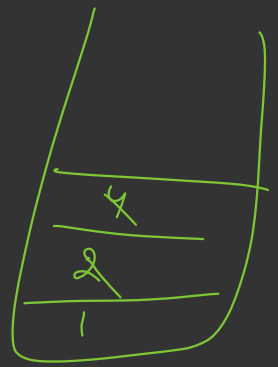
3

```
void inOrder ( Node root) {
    if ( Root == null ) { return }

    ⇒ inOrder (root.left)
    → sout (root.node)
    ⇒ inOrder (root.right)
}
```

3



inorder 4, 2, 6, 5, 7, 1, 8, 9, 3, 10



postorder(Node) {

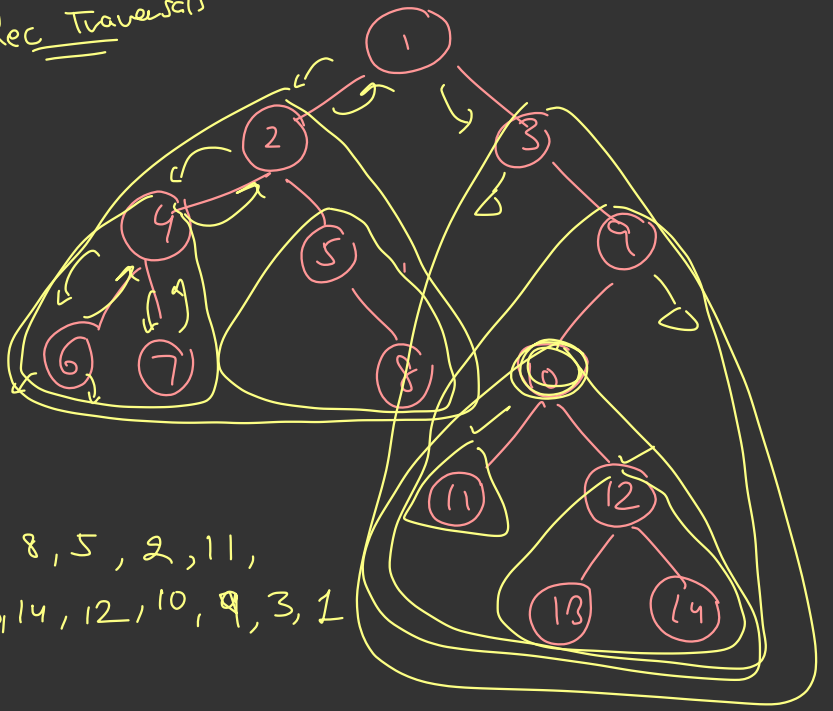
↳ postorder(Left)

↳ postOrder(Right)

↳ sout(Node)

}

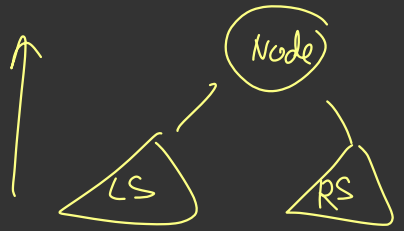
Rec Traversals



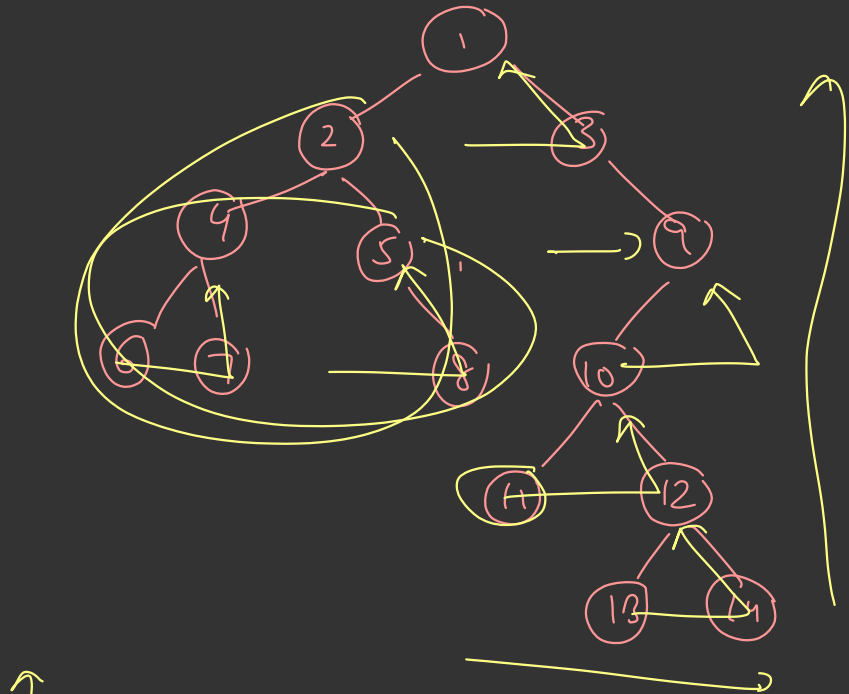
Post order

6, 7, 4, 8, 5, 2, 11,

13, 14, 12, 10, 9, 3, 1



Bottom up



6, 7, 4, 8, 5, 2, 11, 13, 14, 12, 10, 9, 3, 1

Next class

Trees

L to R