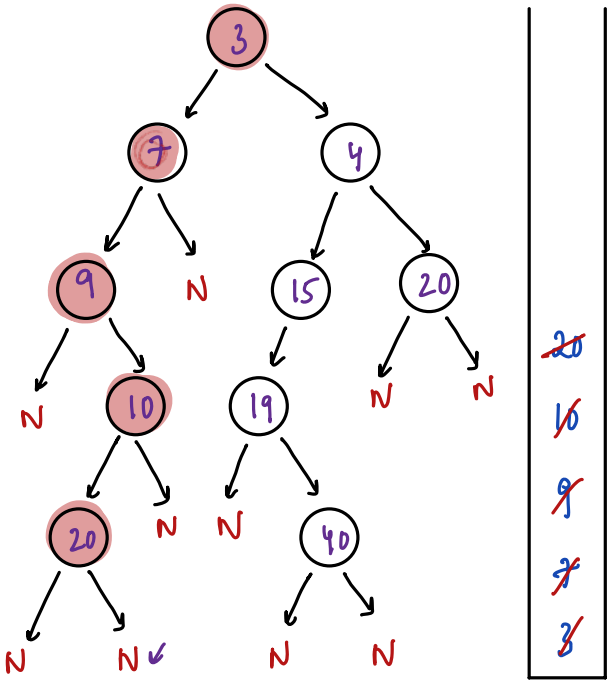
Todays Content:

a) Iterative Inorder
b) level Order
c) left View
d) Right View
e) Construct tree using pre order & inorder

Pre-requisits

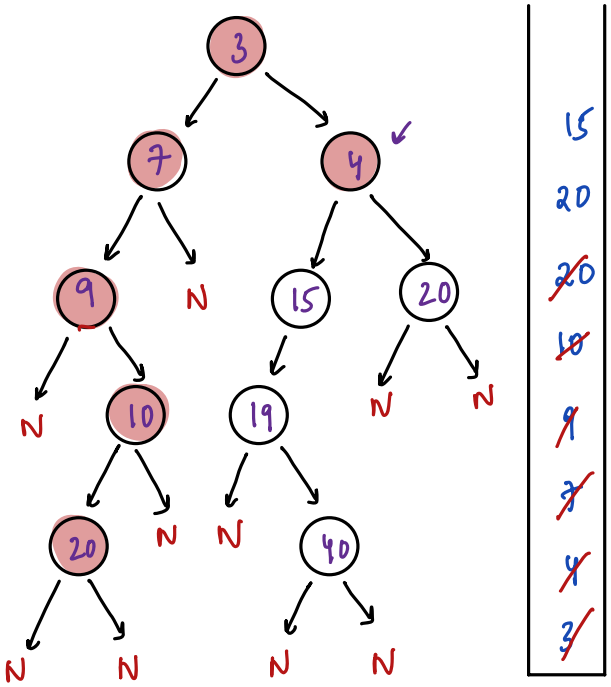
a) Tree
b) Tree Traversals

a) Iterative InOrder:




```
void inorder(Node root){ TC: O(N)
    Node curr = root;          SC: O(N)
    stack<Node> st;
    while(curr!=NULL || st.size()>0){
        while(curr!=NULL){ // curr=null
            st.push(curr)
            curr = curr.left
        }
        Node t = st.peek(); // t=3
        st.pop();
        print(t.data)
        curr = t.right   // curr=4
    }
}
```

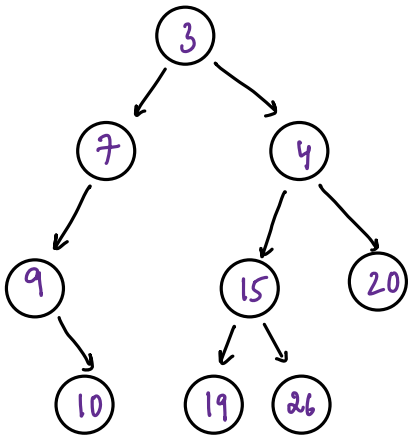
b) Iterative PreOrder: DLR




```
void inorder(Node root){ TC: O(N)
    stack<Node> st;          SC: O(N)
    st.push(root);
    while(st.size()>0){
        Node t = st.peek(); t=4
        st.pop();
        print(t.data)
        if(t.right!=NULL){
            st.push(t.right)
        }
        if(t.left!=NULL){
            st.push(t.left)
        }
    }
}
```

3. Postorder: TODO

3. level order traversal: Left → Right



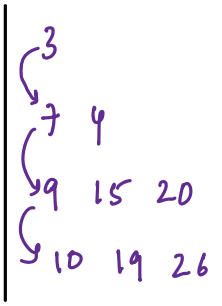
Output:

3

7 4

9 15 20

10 19 26



Idea: ~~3~~ | ~~7~~ ~~4~~ | ~~9~~ ~~15~~ ~~20~~ | ~~10~~ ~~19~~ ~~26~~

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

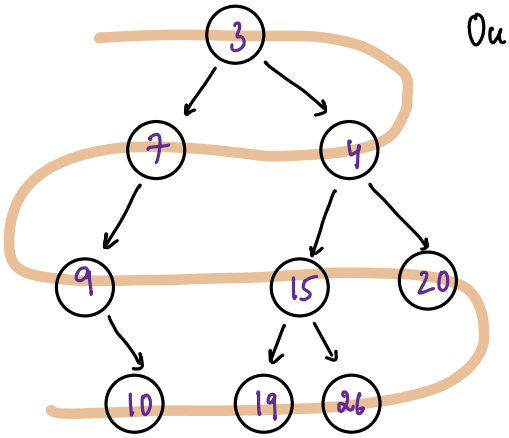
```
void levelOrder(Node root){
    Queue<Node> que;
    que.enque(root);
    while(que.size()>0){
        int N=que.size();
        for(int i=0; i<N; i++){
            Node t=que.front();
            print(t.data);
            que.deque();
            if(t.left!=NULL){que.enque(t.left)}
            if(t.right!=NULL){que.enque(t.right)}
        }
    }
}
```

	QueueSize	Pop & Insert child
level1	1	For 1 : Pop node print Insert child : nextline
level2	2	For 2 : Pop node print Insert child : nextline
level3	3	For 3 : Pop node print Insert child : nextline
level4	3	For 3 : Pop node print Insert child : nextline
	0	Stop

4. level order traversal: Right → left

a. Push Right child & left child Next.

3. level order traversal: Zig-Zag



Output: 3

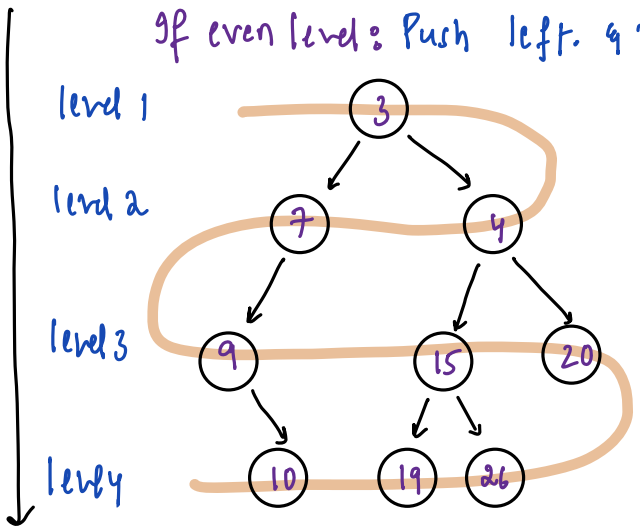
4 7

9 15 20

26 19 10

Ideal: If odd level: Push right & left * won't work

If even level: Push left. & right



~~3~~ | ~~4~~ ~~7~~ | 15 20 9

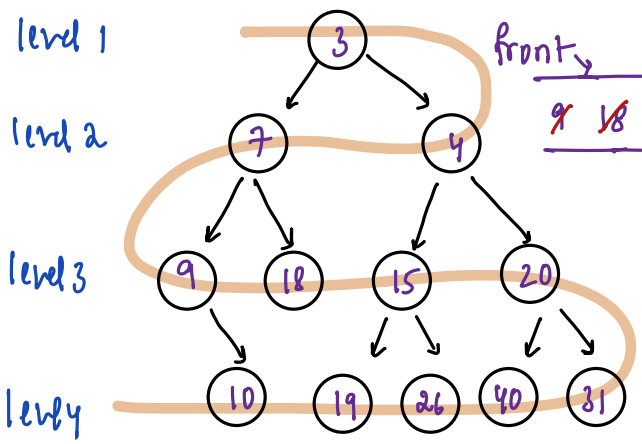
output

3

4 7

We should be able enque & deque from both sides = deque?

4. Using Deque: delete an node ⇒



front → ~~9~~ ~~18~~ ~~15~~ ~~20~~ ~~3~~ ~~7~~ ~~4~~ ~~10~~ ~~19~~ ~~26~~ ~~40~~ ~~31~~ ← rear

output

3

4 7

9 18 15 20

31 40 26 19 10

Note:

Odd level:

a) Delete front

b) Insert rear

c) left → right

Even level:

a) Delete rear

b) Insert front

c) right & left

```
void levelOrder(Node root){
    Deque<Node> dq;
    dq.enqueRear(root);
    int level=1;
    while(dq.size()>0){
        int N=dq.size();
        for(int i=0;i<N;i++){
            if(level%2==1){
                Node t=dq.frontC);
                dq.dequeFront();
                print(t.data)
                if(t.left!=NULL){dq.enqueRear(t.left)}
                if(t.right!=NULL){dq.enqueRear(t.right)}
            }else{
                TODO
            }
        }
        level++;
    }
}
```

front ↔ rear

enqueFront()	dequeFront()	front()
enqueRear()	dequeRear()	rear()

4. Left View:

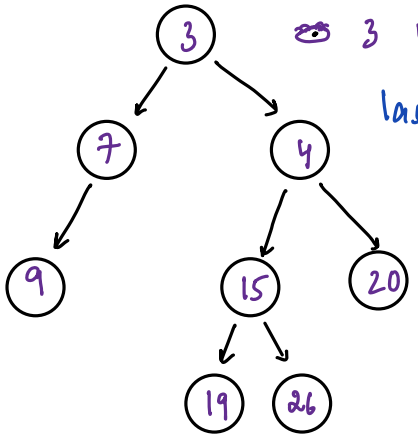
3 7 9 19

1st node of every level

5. Right View

3 4 20 26

last node at every level: TODO

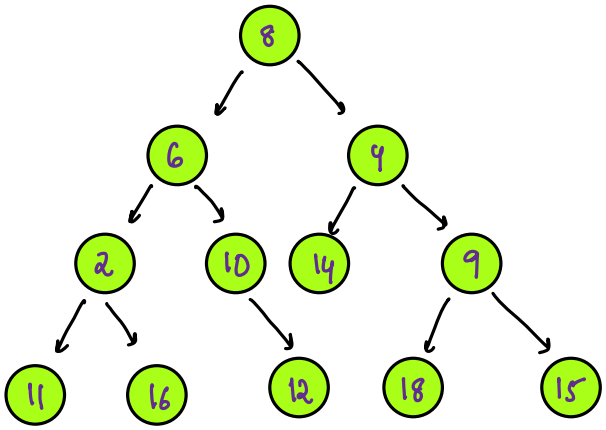



```
void leftView(Node root){
    Queue<Node> que;
    que.enque(root);
    while(que.size()>0){
        int N=que.size();
        for(int i=0; i<N; i++){ // Print a particular level
            Node t=que.front();
            if(i==0){ print(t.data);} // printing 1st node at each level
            que.deque();
            if(t.left!=NULL){ que.enque(t.left)}
            if(t.right!=NULL){ que.enque(t.right)}
        }
    }
}
```

6. Giving pre-order & inorder of BT of distinct values

Print Postorder traversal

Eg:



Ideal:

1. Create BT
2. print Postorder

given in[] & pre[]

a. Create Binary Tree?

root

DLR

pre[] =	0	1	2	3	4	5	6
8	6	2	11	16	10	12

pre[] LST

pre[] RST

7	8	9	10	11
4	14	18	9	15

LDR

in[] = | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| --- | --- | --- | --- | --- | --- | --- |
| 11 | 2 | 16 | 6 | 10 | 12 | 8 |

in[] LST

in[] RST

7	8	9	10	11
14	18	4	9	15

pre[] LST

1	2	3	4	5	6
6	2	11	16	10	12

pre[] RST

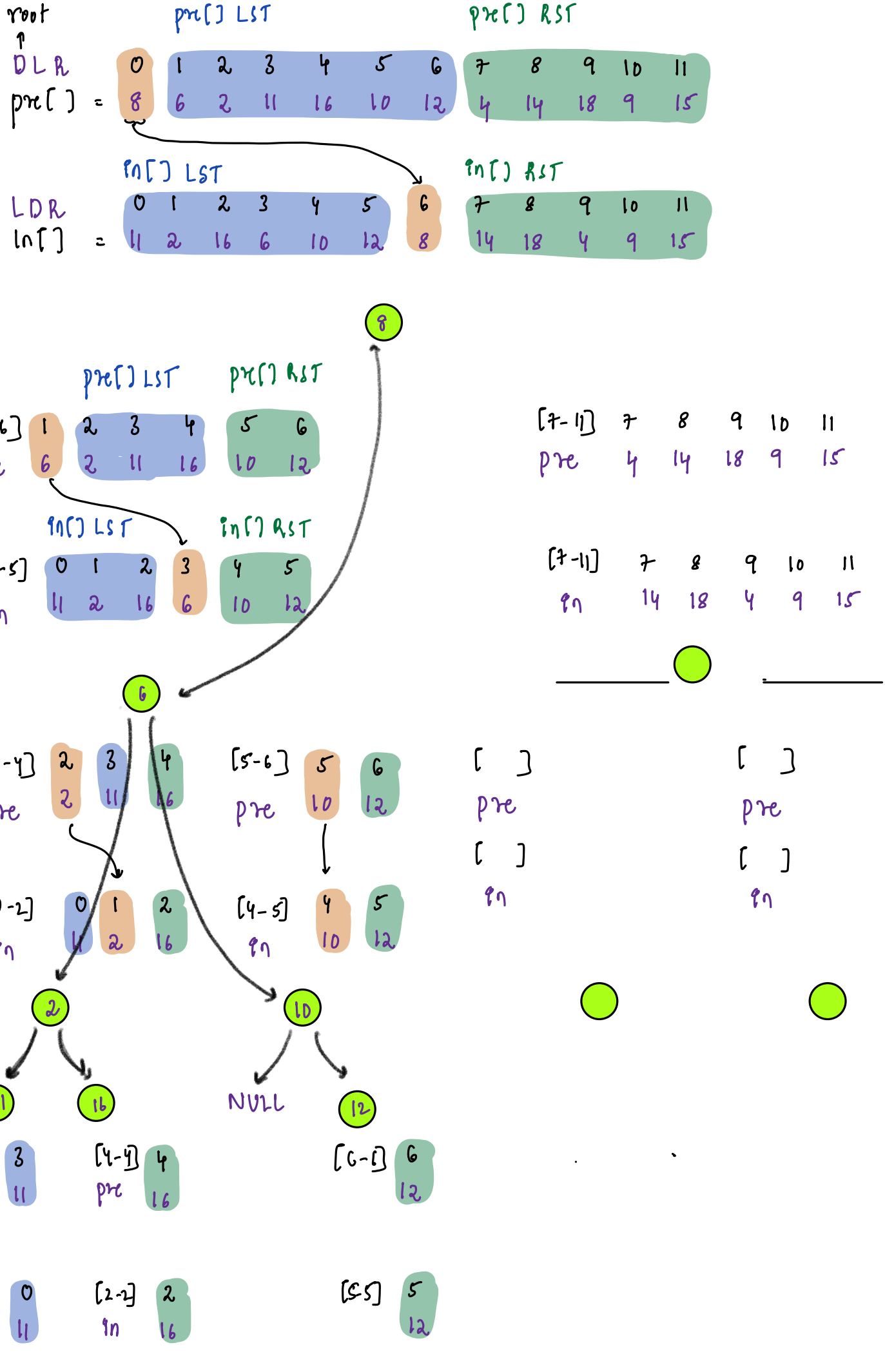
7	8	9	10	11
4	14	18	9	15

in[] LST

0	1	2	3	4	5
11	2	16	6	10	12

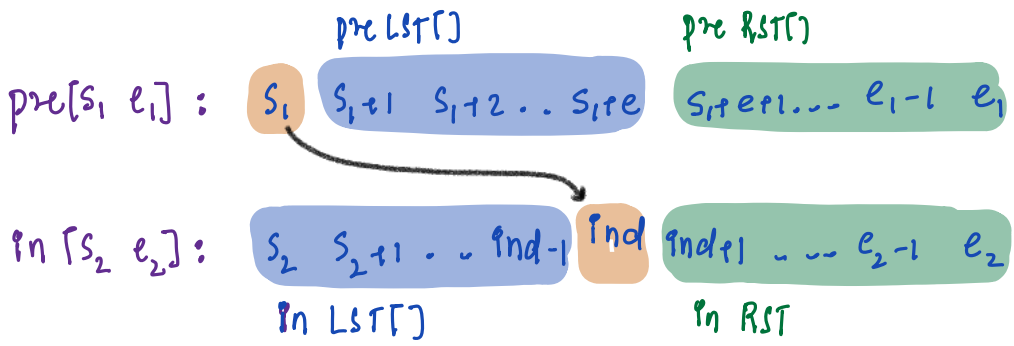
in[] RST

7	8	9	10	11
14	18	4	9	15


root
DLR
pre[] = 8
pre[] LST
0 1 2 3 4 5 6
8 6 2 11 16 10 12
pre[] RST
7 8 9 10 11
4 14 18 9 15
LDR
in[] =
in[] LST
0 1 2 3 4 5 6
11 2 16 6 10 12 8
in[] RST
7 8 9 10 11
14 18 4 9 15
8
pre[] LST
pre[] RST
[1-6] 1 2 3 4 5 6
pre 6 2 11 16 10 12
in[] LST
in[] RST
[0-5] 0 1 2 3 4 5
in 11 2 16 6 10 12
[7-11] 7 8 9 10 11
pre 4 14 18 9 15
[7-11] 7 8 9 10 11
in 14 18 4 9 15
6
[2-4] 2 3 4
pre 2 11 16
[5-6] 5 6
pre 10 12
[ ]
pre
[ ]
in
[ ]
pre
[ ]
in
[0-2] 0 1 2
in 11 2 16
[4-5] 4 5
in 10 12
2
10
11
16
NULL
12
[3-3] 3
pre 11
[4-4] 4
pre 16
[6-6] 6
12
[0-0] 0
in 11
[2-2] 2
in 16
[5-5] 5
12

Ass: Given pre[] & in[] construct tree, return root node of tree

```
Node Tree(int pre[], int s1, int e1, int in[], int s2, int e2){
    if(s1>e1){return NULL} // no element, no tree
```




```
    Node root = new Node(pre[s1])
    int ind=0;
    for(int i=s2; i<=e2; i++){
        if(in[i]==pre[s1]){
            ind=i; break;
        }
    }
```

a b b-a+1

int e = ind-s2; // No:of ele in LST $[s_2\ \ s_2+1\ \ ..\ \ ind-1]$ =

```
    root.left = Tree(pre, s1+1, s1+e, in, s2, ind-1) // Create LST & return root of LST
    root.right = Tree(pre, s1+e+1, e1, in, ind+1, e2) // Create RST & return root of RST
    return root;
}
```

TC: O(N) * O(N) ≈ O(N²)

↳ Searching for an ele in in[] takes O(n) time.

↳ Since data is distinct:

→ Store in[] in hashmap<key, value>

key: ele, value: index

TC: O(N) : If we can use hashmap.