Todays Content:

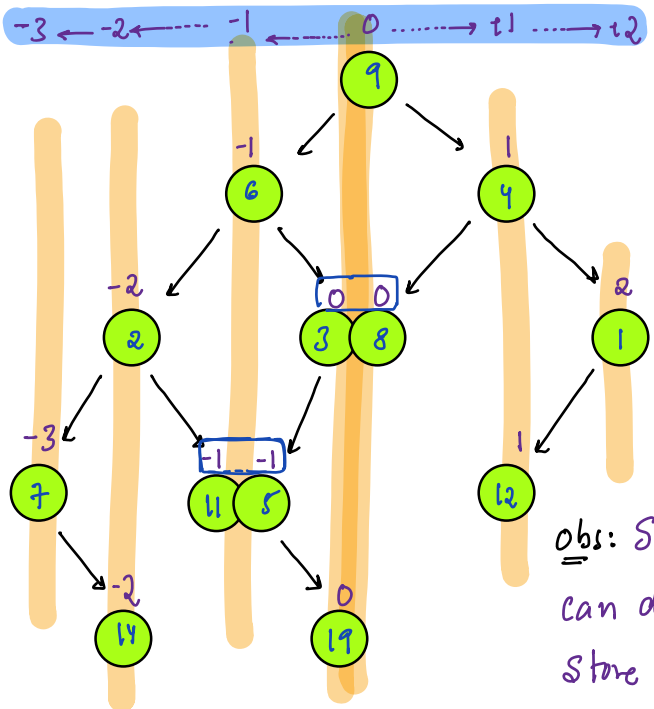
a) Vertical level order traversaal

- → top view
- → bottom view
- → diagnol view : TODO

b) Construct BT from inorder & preorder

Vertical level order traversal (left → right)

Note: When we go from left → right vertical level will increase



Expected:

7

2 14

6 11 5

9 3 8 19

4 12

1

Obs: Since none of the traversals can directly print required data, store vertical level order traversal & then print it

key → vertical level

value → nodes at that vertical level

Idea: HashMap<Int, List<Int>> HM

level nodes at level

minlevel = -3: 7

-2: 2 14

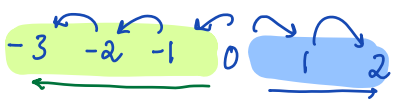
-1: 6 11 5

0: 9 3 8 19

1: 4 12

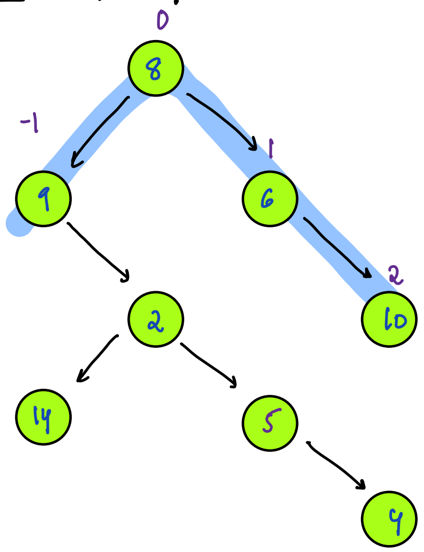
manlevel = 2: 1

All level from min to man should be in hashmap: True



Obs: Iterate from min level to man level & print data at that level using hashmap

Qn: Q: top view



Hashmap

-1 : 9 14
0 : 8 2
1 : 6 5
2 : 10 4

Inorder Hashmap

{9 14 2 5 4 8 6 10}

-1 : 9 14
0 : 2 8
1 : 5 6
2 : 4 10

} order incorrect

preorder Hashmap

8 9 2 14 5 4 6 10

-1 : 9 14
0 : 8 2
1 : 5 6
2 : 4 10

} order wrong

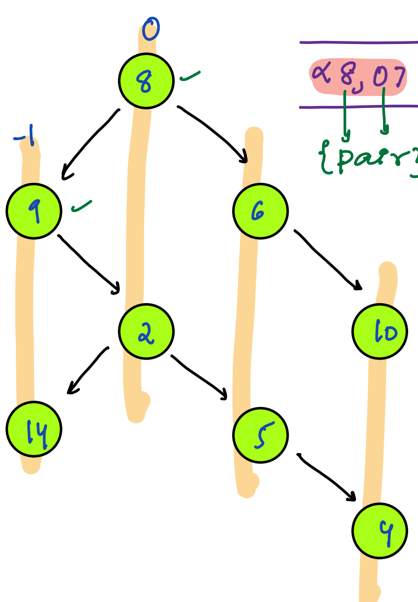
postorder Hashmap

14 4 5 2 9 10 6 8

-1 : 14
0 :
1 :
2 :

} order wrong

Note : Since priority should be given to depth, we use level order traversal



<8, 0>	<9, -1>	<6, 1>	<2, 0>	<10, 2>	<14, -1>	<5, 1>	<4, 2>

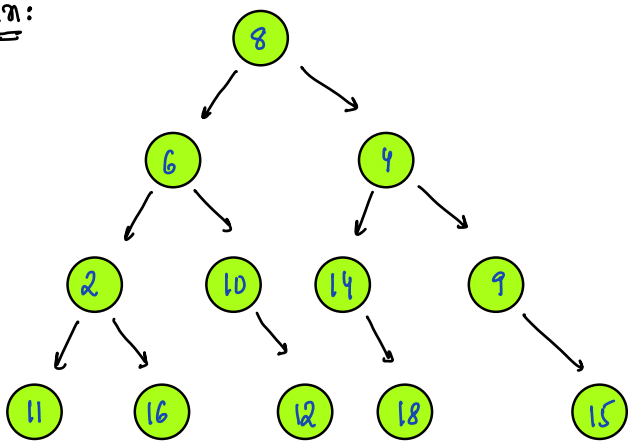
{pair}

-1 : 9 14
0 : 8 2
1 : 6 5
2 : 10 4

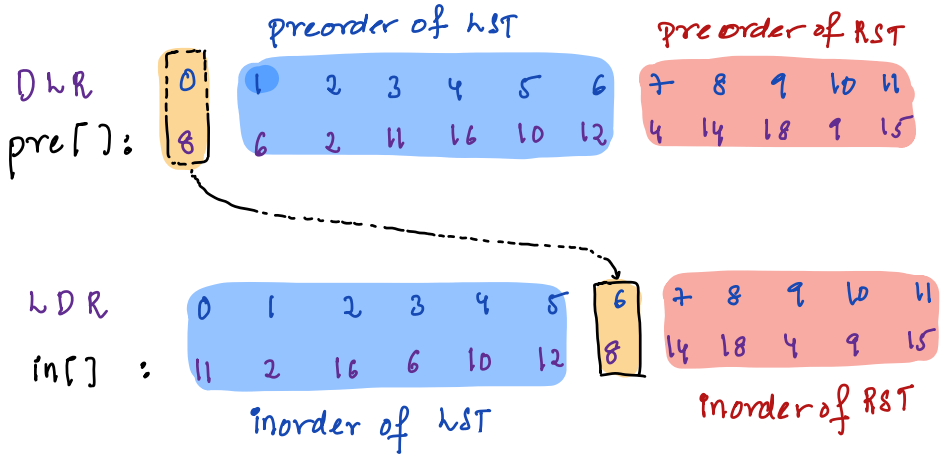
```
void Vertical level ( Node root ) {
    int minL = 0, manL = 0
    Hashmap< int, list<int>> hm // check syntax
    Queue< pair< Node, int >> q
    q.enque ( {root, 0} )
    while (q.size() > 0) {
        pair< Node, int> data = q.front()
        q.deque() // deleting front element
        Node temp = data.first } for node
        int l = data.second } vertical level = l
        hm[l].insert( temp.val)
        // list, inside that push node val
        minL = min (l, minL) } // update minL & manL
        manL = man (l, manL) }
        if( temp.left != NULL){
            q.enque ( {temp.left, l-1} )
        }
        if (temp.right != NULL){
            q.enque ( {temp.right, l+1} )
        }
    }
    i = minL ; i <= manL ; i++){
        // level = i, all nodes of level i are present at hm[i]
        hm[i] is a list // Iterate on list & print
        // print 1st element of hm[i] → Top view
        // print last element of hm[i] → bottom view
    }
}
```

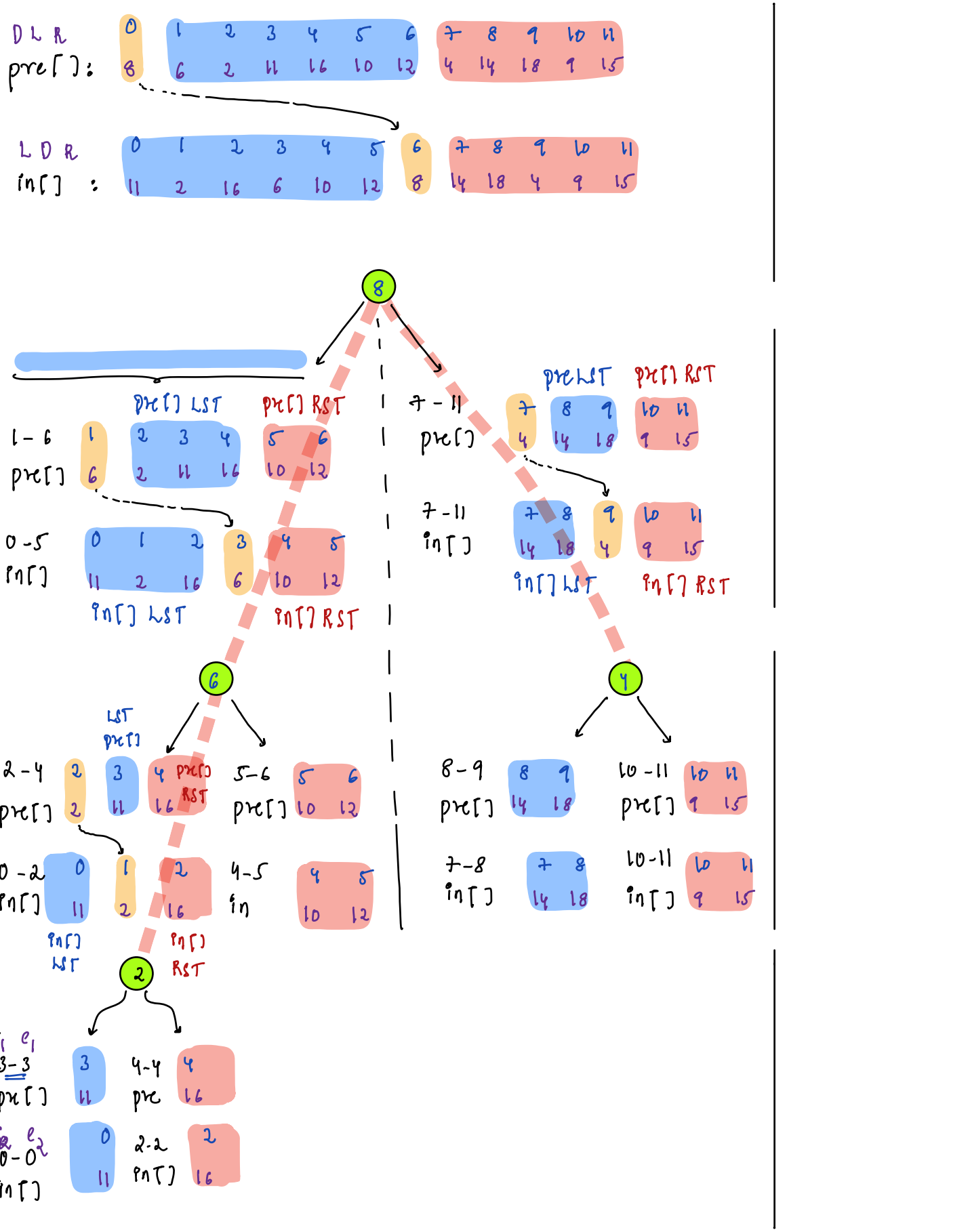
20) Given preorder & inorder of BT of distinct values
print postorder traversal

Eg:



Idea: Using pre[] & in[] construct Tree, using Tree print postorder




DLR
pre[]:
LDR
in[]:
8
pre[] LST
pre[] RST
1-6
pre[]
7-11
pre[]
pre LST
pre[] RST
0-5
in[]
in[] LST
in[] RST
7-11
in[]
in[] LST
in[] RST
6
4
LST pre[]
2-4
pre[]
5-6
pre[]
pre[] RST
8-9
pre[]
10-11
pre[]
0-2
in[]
4-5
in
7-8
in[]
10-11
in[]
in[] LST
in[] RST
2
3-3
pre[]
4-4
pre
0-0
in[]
2-2
in[]

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 && s2 > e2) { // no data
                            return null; }
    // Note: if there is no data in in[], no data in pre[], i condition enough
    Node root = new Node(pre[s1]) // root node = pre[s1]
    //search for root in in[s2 - e2]
    int ind = -1;
    for(i = s2; i <= e2; i = i+1){
        if(in[i] == pre[s1]){
            ind = i; break;
        }
    }
```

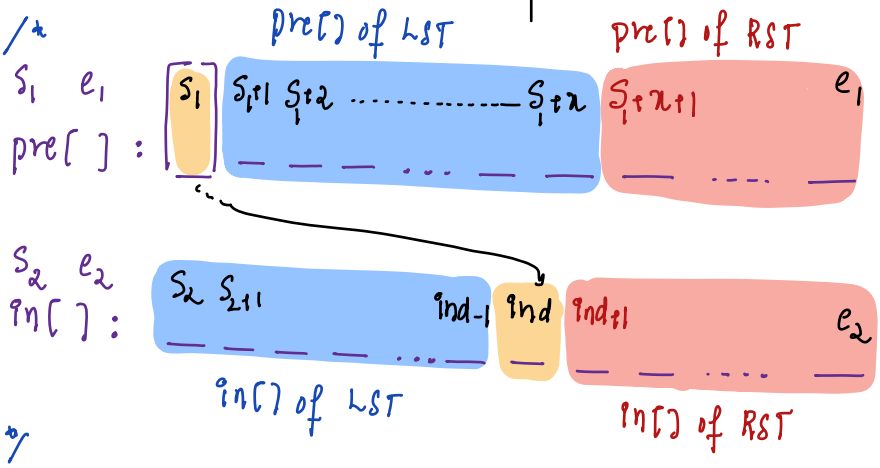
TC: N * N → O(N²)

1) N nodes, for every node search in inorder

// in in[], for an ele get index in inorder

↳ use hashmap<int, int> (inorder value → inorder index)

in[4]: { 8 3 2 9 } ⇒ HM: {8:0, 3:1, 2:2, 9:3}




```
    // No: of elements in LST: in[s2, ind-1] : ind-1 - s2 + 1
    int x = ind - s2; // ele in LST
    root.left = Tree(pre[], s1+1, s1+x, in[], s2, ind-1)
                // Construct LST & return root of LST
    root.right = Tree(pre[], s1+x+1, e1, in[], ind+1, e2)
                // Construct RST & return root of RST
    return root;
}
```

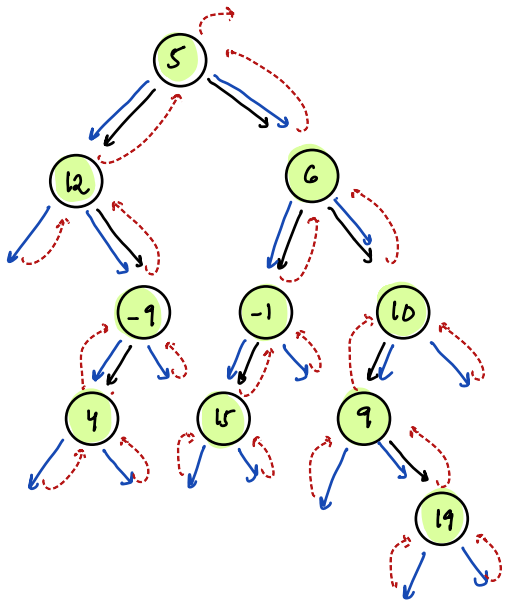
```
Node Tree(int pre[], int s1, int e1, int in[], int s2, int e2, hashmap hm)
{
    if (s1 > e1 && s2 > e2) { // no data
        return NULL }
    // Note: if there is no data in in[], no data in pre[], i condition enough

    Node root = new Node(pre[s1]) // root node = pre[s1]
    //search for root in in[s2 - e2]
     int ind = hm[pre[s1]]
    // No: of elements in LST : in[s2, ind-1] : ind - s2 + 1
     int n = ind - s2 // ele in LST
     root.left = Tree(pre[], s1+1, s1+n, in[], s2, ind-1, hm)
                 // Construct LST & return root of LST
     root.right = Tree(pre[], s1+n+1, e1, in[], ind+1, e2, hm)
                 // Construct RST & return root of RST
     return root;
}
```

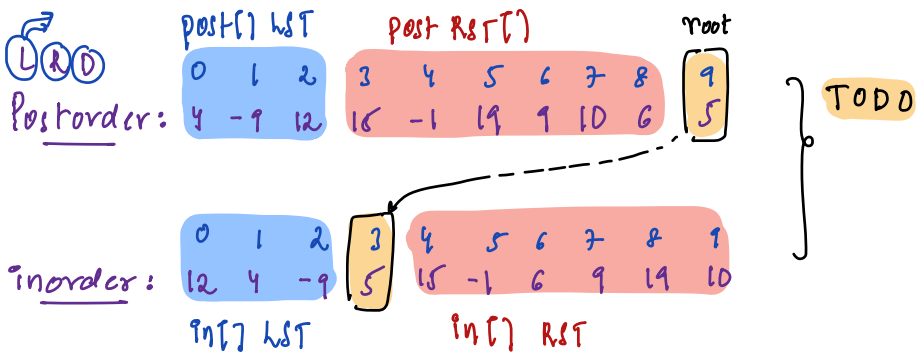
TC: $N * \{1 + 1\} \Rightarrow O(N)$

SC: $N + N \Rightarrow O(N)$

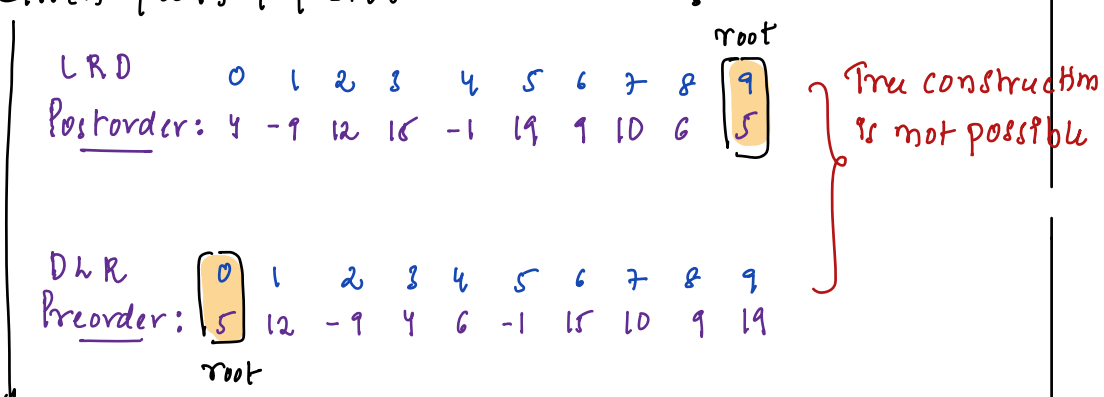
```
void printpostorder(int pre[n], int in[n]) {
    hashmap<int, int> hm
    for(i=0; i<n; i++) { // inserting inorder in hashmap
        hm[in[i]] = i
    }
    Node root = Tree(pre[], 0, n-1, in[], 0, n-1, hm)
    postorder(root) // call postorder function to print
}
```

4Q) Given in[] & post[] construct tree[]





5Q) Given pre[] & post[] construct tree[]?



Note: in[] + { pre[] / post[] } & data distinct, we can construct Tree