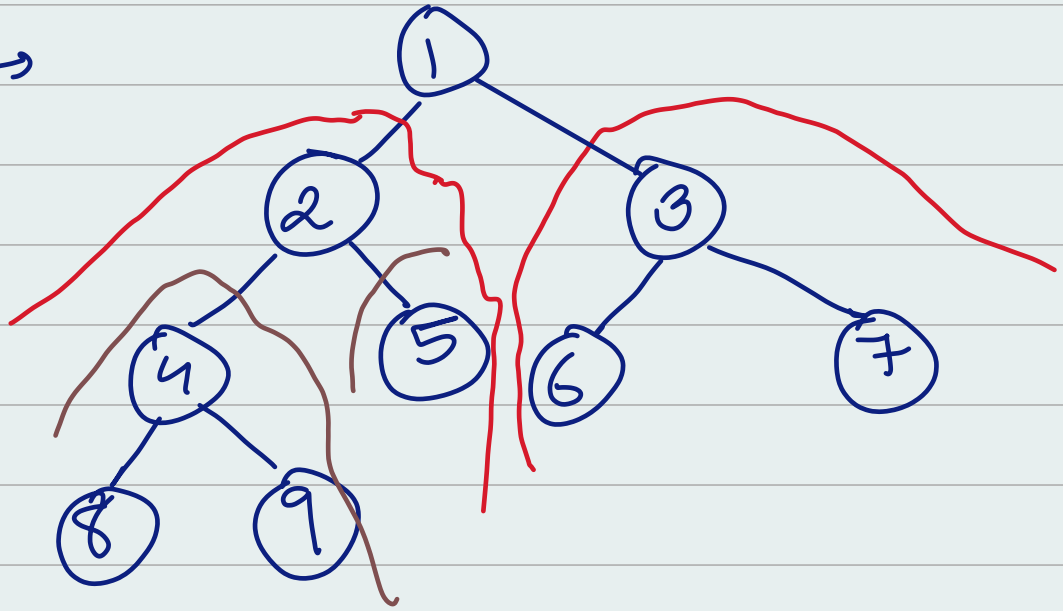
Advanced Tree Questions:-

8) Construct a binary tree from Preorder and InOrder Traversal

→



root

PreOrder :-

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

InOrder

left right

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

Left Right

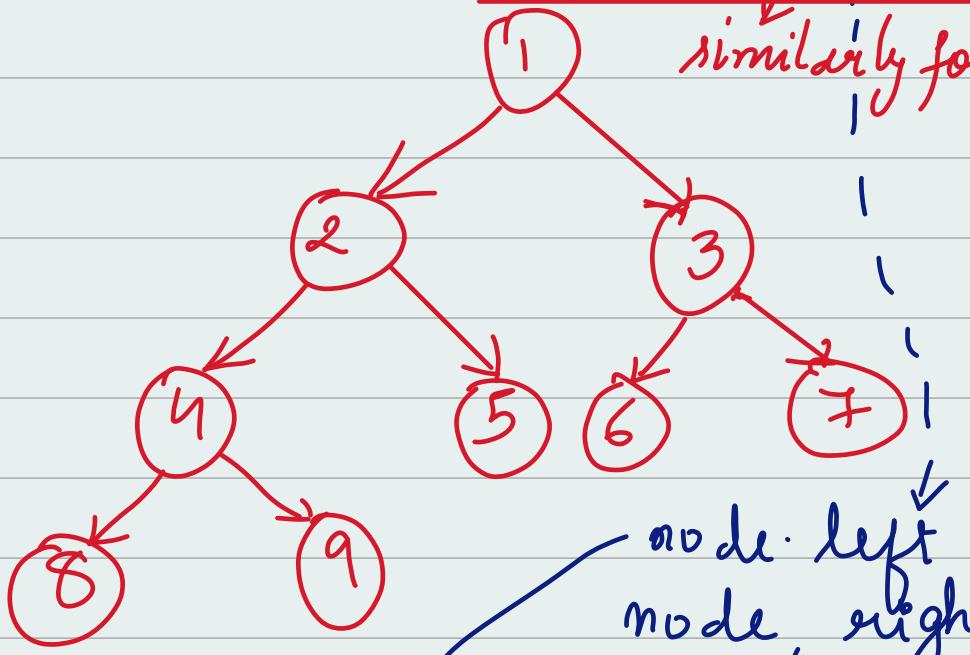
→ using these two create the above binary tree.

→ using (Hash map) we can store the element indices and find them in constant O(1) time

Pre : 1, 2, 4, 8, 9, 5, 3, 6, 7

In : 8, 4, 9, 2, 5, 1, 6, 3, 7

↳ 9 is in the R.H.S of 4. So similarly for 5 put it in R.H.S of 4.

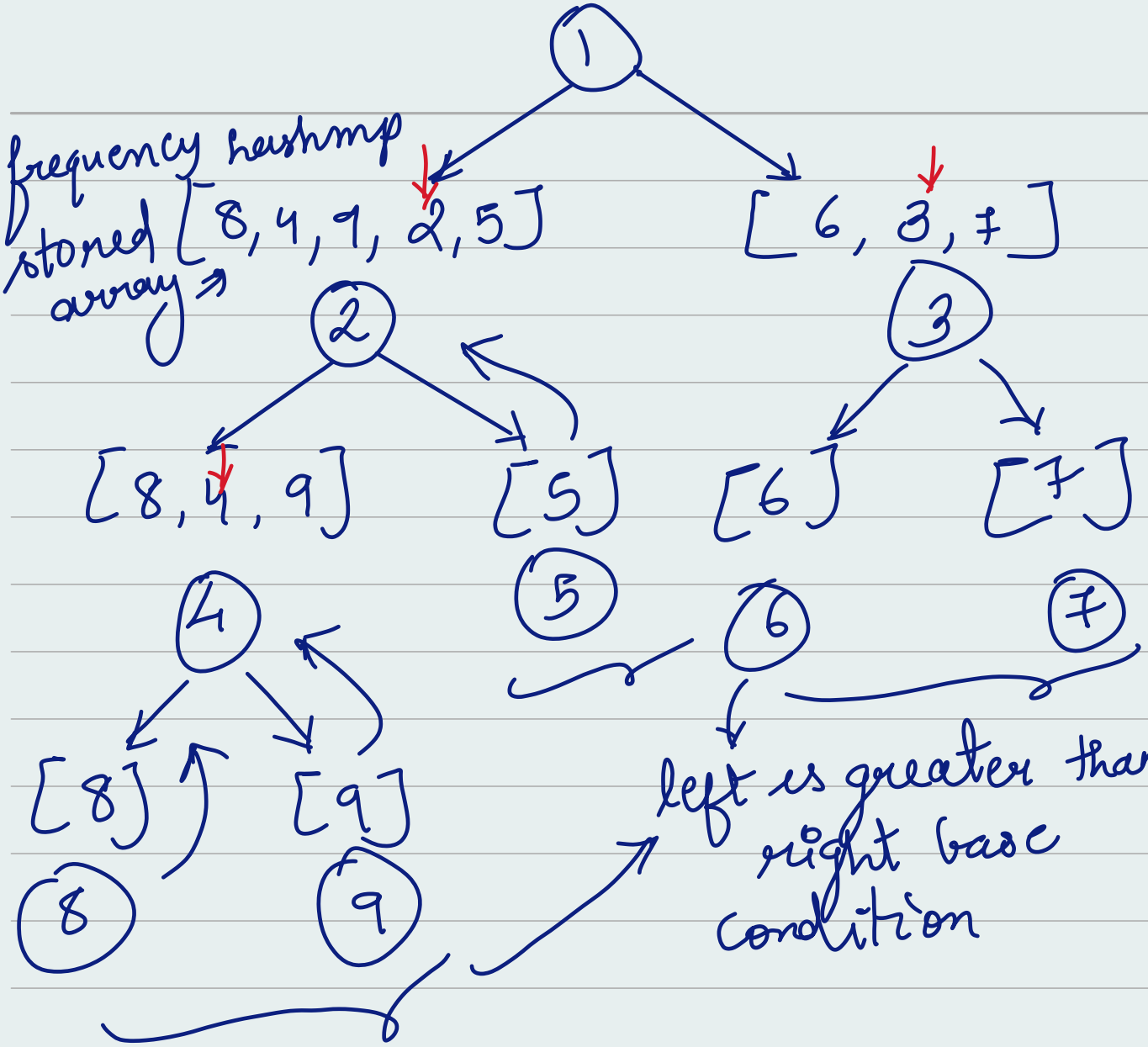


node.left = Recursion
node.right = Recursion

node.left = l, (i-1)
↳ start (left will not change)
↳ Right will change to index -1.

node.right = (i+1), r
↳ right will not change.
↳ left will change to i+1

→ Below is a recursive diagram.



Time complexity is O(N)
space complexity is O(N) for the hashmap that is created

Code :-

Class PreIn {

```
public TreeNode buildTree(
    int[] preOrder, int[] inOrder){

Hashmap<Integer, Integer> map = new
                    Hashmap<>();

  for(int i = 0; i< InOrder length -1; i++){
      map.put(inOrder[i], i);
  }
   int index = {0};

  return helper(preOrder, inOrder, 0,
          preOrder.length -1, map, index);
}

public TreeNode helper(int[] preOrder,
   int[] inOrder, inb left, int right,
   Hashmap<Integer, Integer> map, int
  [] index){

    if(left > right){
          return null;
    }
```

// This check whether the current subtree is valid or not. If the left is greater than right it means indices have crossed each other and there are no elements so it will return null.

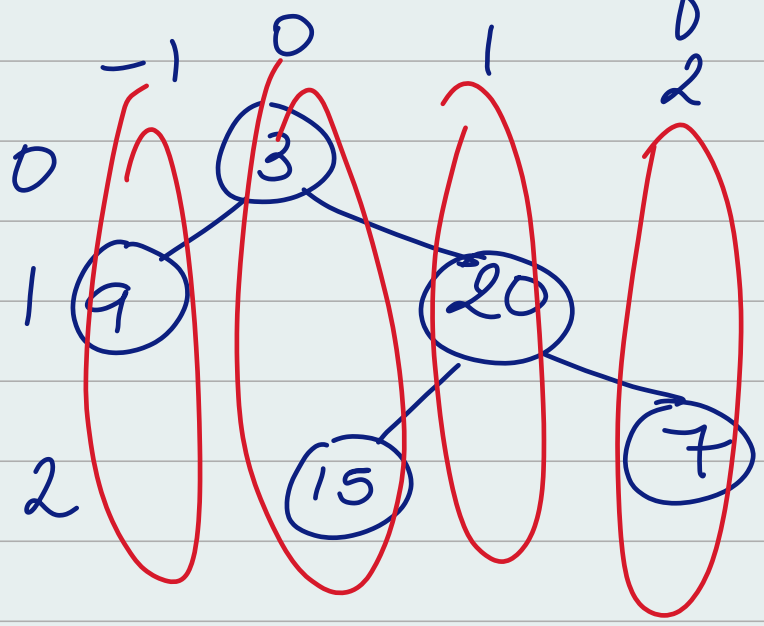
(8) → Here left is 0 and index is index-1 [leaf node] boundary crossed so return null.

```
int current = preOrder index[[0]];

Index[0]++;

TreeNode node = new TreeNode
                (current);

if (left == right){
    return node;
}

int inIndex = map.get(current);
```

// This code uses inIndex obtained from the map to correctly divide the inOrder array into left and right subtrees for the recursive calls

```
node left = helper(preOrder, inOrder,
    left, inIndex - 1, map, index);
node right = helper(preOrder, inOrder,
    inIndex + 1, right, map, index);
return node;
}
```

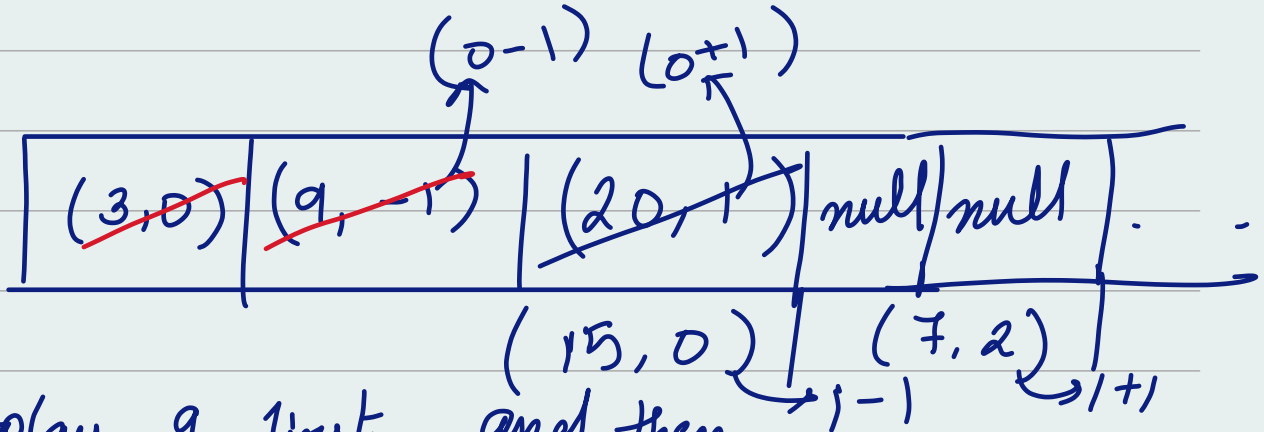
8) Vertical Order traversal of a binary tree



output:

[[9], [3], [15], [20], [7]]

We will use Breadth First Search by using Queue



Display 9 first and then and then rest of it vertical level by level

Store it in a hashmap The key will be col, value is nodes

column	nodes
0	[3, 15]
-1	[9]
1	[20]
2	[7]

Sort these keys by maintaining min and max

min = -1

max = 2

run a for loop (min to max)!

9
3, 15
20
7

Ans

Space complexity is total of Hashmap, Queue, & output. Total is O(N)

O(N) + O(N) + O(N) - O(N)

Code :-

```
class verticalTraversal {

public List<List<Integer>> verticalTraversal(TreeNode node){

List<List<Integer>> ans =
        new ArrayList<List<Integer>>();

if (node == null){
    return ans;
```

```
int col = 0;

Queue<Map.Entry<TreeNode, Integer>>
        queue = new ArrayDeque<>();

Map<Integer, ArrayList<Integer>>
        map = new HashMap();

queue.offer(new AbstractMap.
        SimpleEntry<>(node, col));

int min = 0,
 int max = 0;

while (!queue.isEmpty()){

    Map.Entry<TreeNode, Integer>
        removed = queue.poll();

    node = removed.getKey();
    col = removed.getValue();

    if (node != null){
        if (!map.containsKey(col)){
            map.put(col, new ArrayList
                <Integer>());
```

```
}

map.get(col).add(node.val);

min = Math.min(min, col);
max = Math.max(max, col);

queue.offer(new AbstractMap.SimpleEntry
        <>(node.left, col-1));

queue.offer(new AbstractMap.SimpleEntry<>
        (node.right, col+1));

    }
  }

  for(int i = min, i<= max; i++){
      ans.add(map.get(i));
  }
  return ans;
 }
}
```

Word Ladder - Hard!

words :- hit ⟶ cog

~~dot~~, ~~hot~~, ~~dog~~, lot, ~~log~~, cog

Using set cause it is easy to look up

ans

0 1
2
3
4
+1
5

~~hit~~	~~hot~~	~~dot~~	~~lot~~	~~dog~~	~~log~~

Here every word is compared with the above set so n x n times the word comparisons x the character comparison so Time Complexity is $O(N^2 \times m)$ Time

& O(N) is space complexity.