12/05/2023

Q1 Create a tree from given inorder & pre-order traversal.

Inorder traversal ⇒ LNR

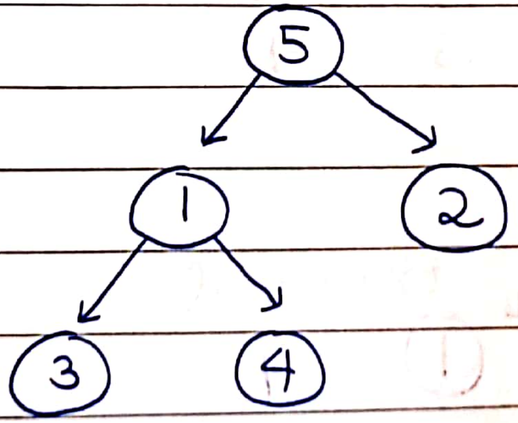
Preorder traversal ⇒ NLR

Postorder traversal ⇒ LRN

i/p → 3 1 4 5 2 } inorder

5 1 3 4 2 } preorder

O/p →



L → left

N → node

R → right

We can say that the 1st value in the pre-order traversal is the root node due to NLR.

↳ root node

Left subtree pos Right subtree

↓

3 1 4 5 2

5 1 3 4 2

↳ root

Now 5 is the root node. Now search for that node in the inorder traversal. The left elements to the root node in the inorder traversal is the left subtree & right element is the right subtree.

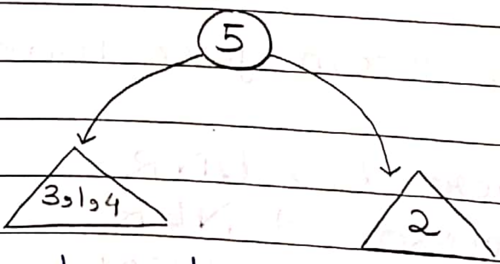
↙ inorderStart ↙ pos-1

3, 1, 4 ⇒ Left subtree

2 ⇒ Right subtree

↑

pos+1 to inorderEnd



We have broken down the problem into smaller subparts & recursion will solve this.

inorder ⇒ 3 1 4

preorder ⇒ 1 3 4

↙

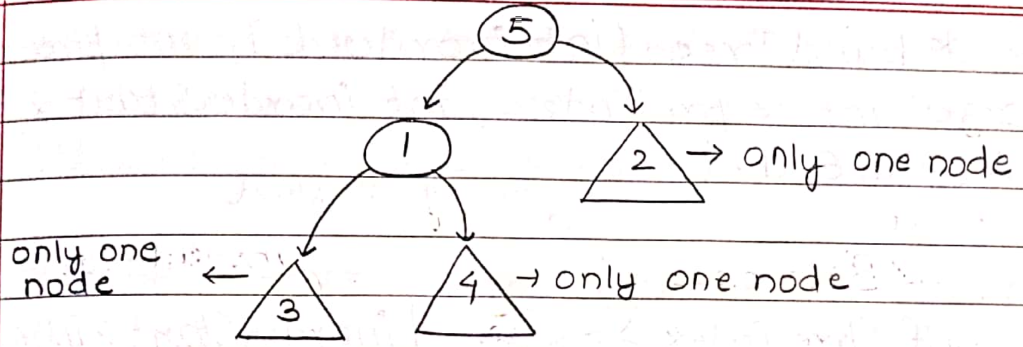
root

Check in inorder traversal.

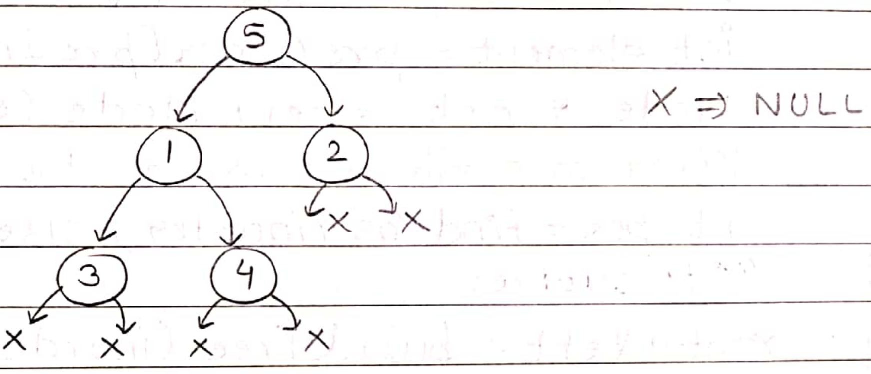
3 1 4

↑ ↑

left right



If only node (one) is present, simply make that as node. Final tree becomes by making 3, 4 & 2 as node.



Note → The 1st element in the pre-order traversal will always be the root node of the tree. If there is node on the left or right in the inorder traversal, this means we have to put NULL.

Code

```
//Linear search to find root in inorder traversal
int findPos (int arr[], int n, int element){

      for(int i=0; i<n; i++){
            if(arr[i] == element){
                  return i;
            }
      }
      return -1;
}
```

```
Node * buildTree (int inorder [], int preorder[],
int size, int & preIndex, int inorderStart,
int inorder End){
                               array finished
        // Base case                         invalid array
        if (preIndex >= size || inorderStart > inOrderEnd
                     return NULL;
      }
      // Find root from preorder & create root node
      int element = preOrder[preIndex ++];
      Node * root = new Node (element);
     //Find root element in inorder
     int pos = findPos (inorder, size, element);
    //Left subtree
    root→left = buildTree (inorder, preorder,
     size, preIndex, inorderStart, pos-1);
   //Right subtree
   root → right = buildTree (inorder, preorder,
   size, preIndex, pos+1, inorderEnd);
  //return root node
  return root;
}
```

Note→ preIndex is passed by reference because it should be updated else same node would be made as root of . tree which we don't want. (This happens while returning from recursive call)

Parameters

1) inorder & preorder we need to know to build the tree.

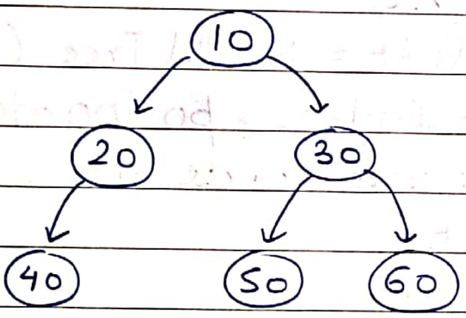
2) size to make sure we stay inside preorder array.

3) inorder Start & inorder End we need as we need to pass some part of the array in the recursive call & not the full array.

Q2 Create a tree from inorder & postorder traversal.

i/p ⇒ 40 20 10 50 30 60 } inorder
40 20 50 60 30 10 } postorder

o/p ⇒



The last node in the postorder traversal will be the root node due to L R N
↳ root node

L R N ←

In this first recursive call for right subtree and then for left subtree. Rest the logic remain same as that of Q1.

Code

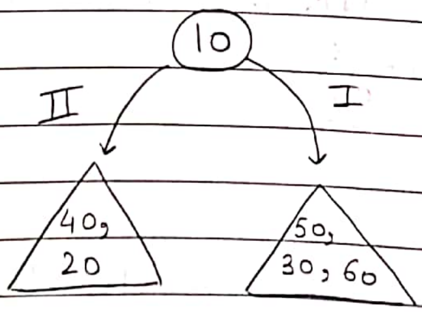
```
Node * build Tree (int inorder [], int postorder
[], int size, int &postIndex, int inorderStart,
int inorderEnd) {
        //Base case
        if (postIndex <0 || inorderStart >
                inorderEnd) {
                return NULL;
        }
//Find root and create node for that
int element = postorder [postIndex--];
Node * root = new Node (element);
//Find position of root in inorder
int pos = findPos (inorder, size, element);
//Right subtree first
root → right = buildTree (inorder, postorder,
 size, postIndex, pos+1, inOrderEnd);
//Left subtree
root → left = buildTree (inorder, postorder,
size, postIndex, inorderStart, pos-1);
//return the root node
return root;
}
```

Dry run

40 20 10 50 30 60

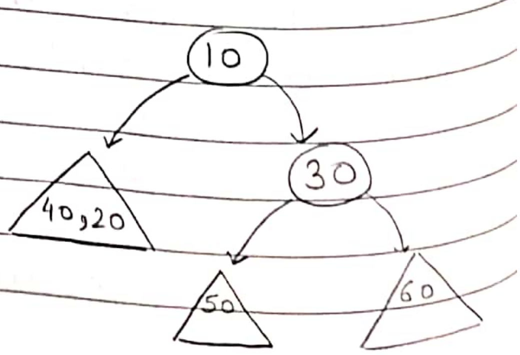
40 20 50 60 30 10

↳ root

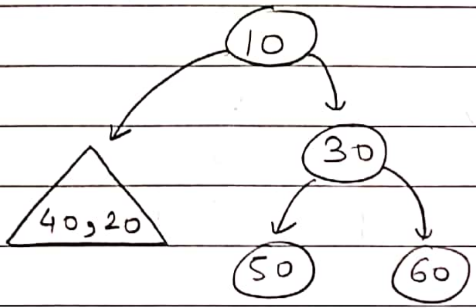


50, 30, 60 } inorder

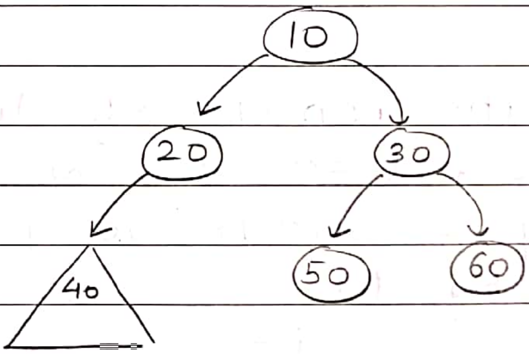
50, 60, 30 } postorder



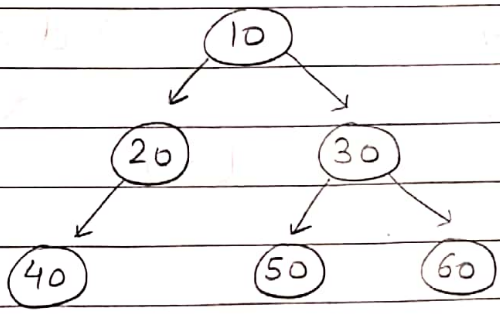
50 and 60 are only one node & hence make node.



40, 20 } inorder
40, $\boxed{20}$ → root } postorder



↳ only node & hence make node



The above tree is the final tree constructed from inorder and postorder traversal.

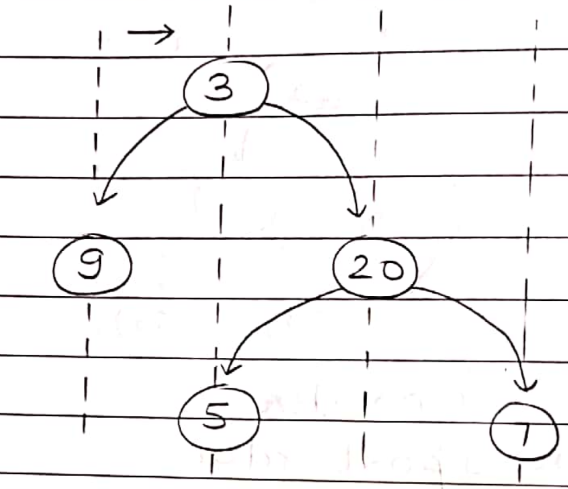
Using map

```
void createMapping (unordered_map <int, int>&
mapping, int inorder [], int n) {
        for (int i = 0 ; i < n ; i++) {
            mapping [inorder [i]] = i ;
        }
}
```

Runs only once

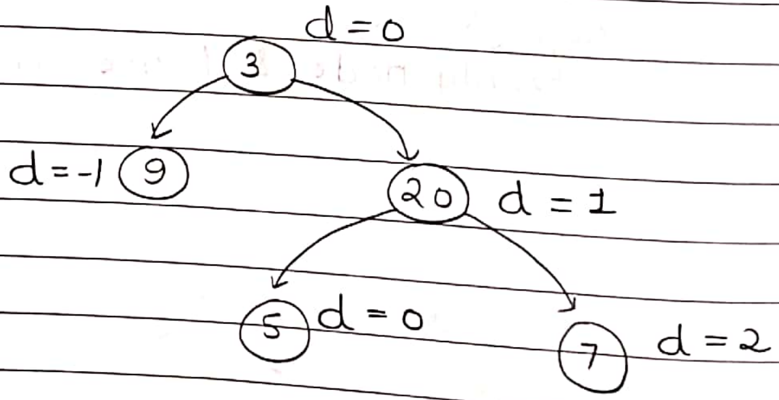
Q3 Vertical order traversal

i/p→



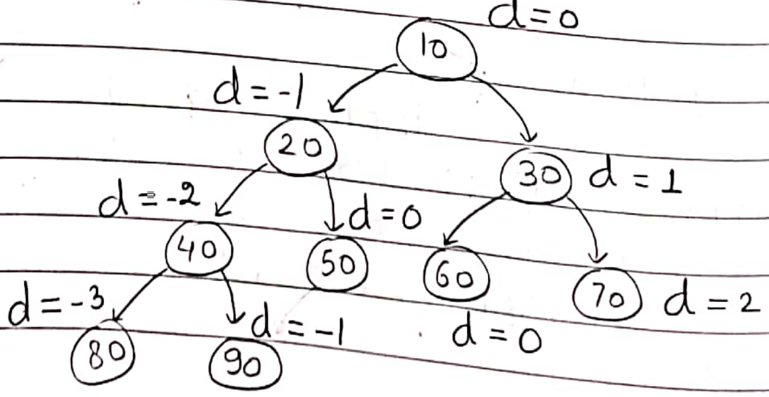
o/p→ 9 3 5 20 7

Here we use the concept of distance. At the root node, d = 0. As we go left d reduced by 1 and if we go right, d increases by 1.



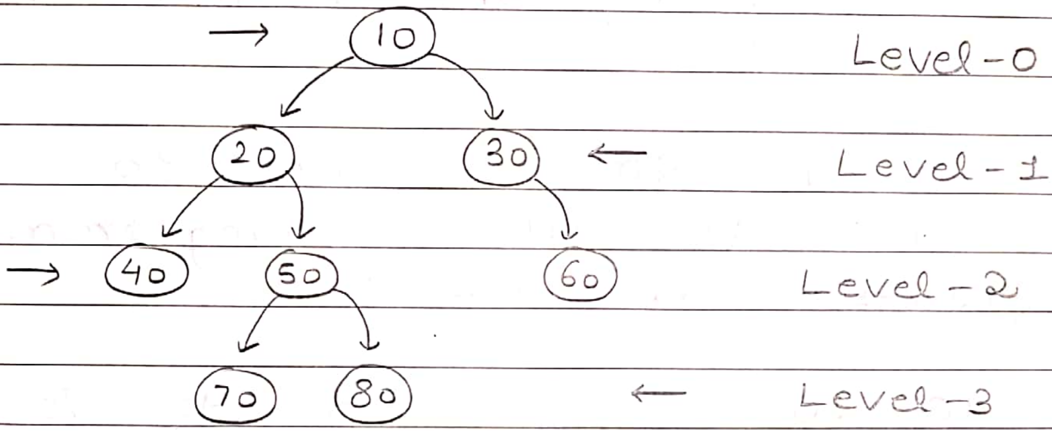
-1	⇒	9
0	⇒	3 5
1	⇒	20
2	⇒	7

Ex→



-3 ⇒	80		
-2 ⇒	40		
-1 ⇒	90	20	
0 ⇒	10	50	60
1 ⇒	30		
2 ⇒	70		

Q4 Zig-zag traversal

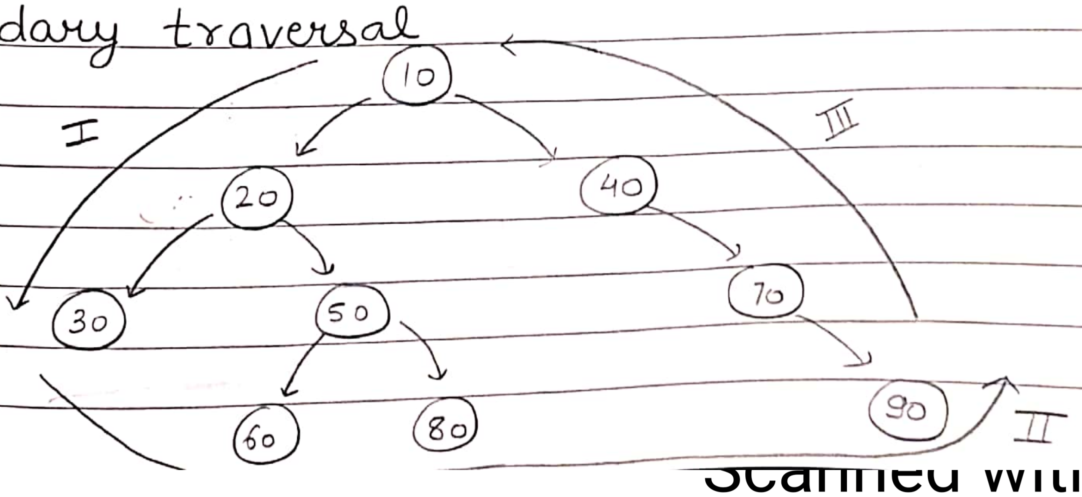


10 30 20 40 50 60 80 70

Even level ⇒ first right insert in queue
Odd level ⇒ first insert Left in queue.

This can be done via the level-order traversal but we have to take care of the levels i.e even or odd.

Q5 Boundary traversal



1) Print the left nodes first.
2) Print the leaf nodes after.
3) Print the right nodes after.

* Start from root node. Print node & go left but if leaf node found, stop.

10 20

* Apply inorder and print nodes which are leaf

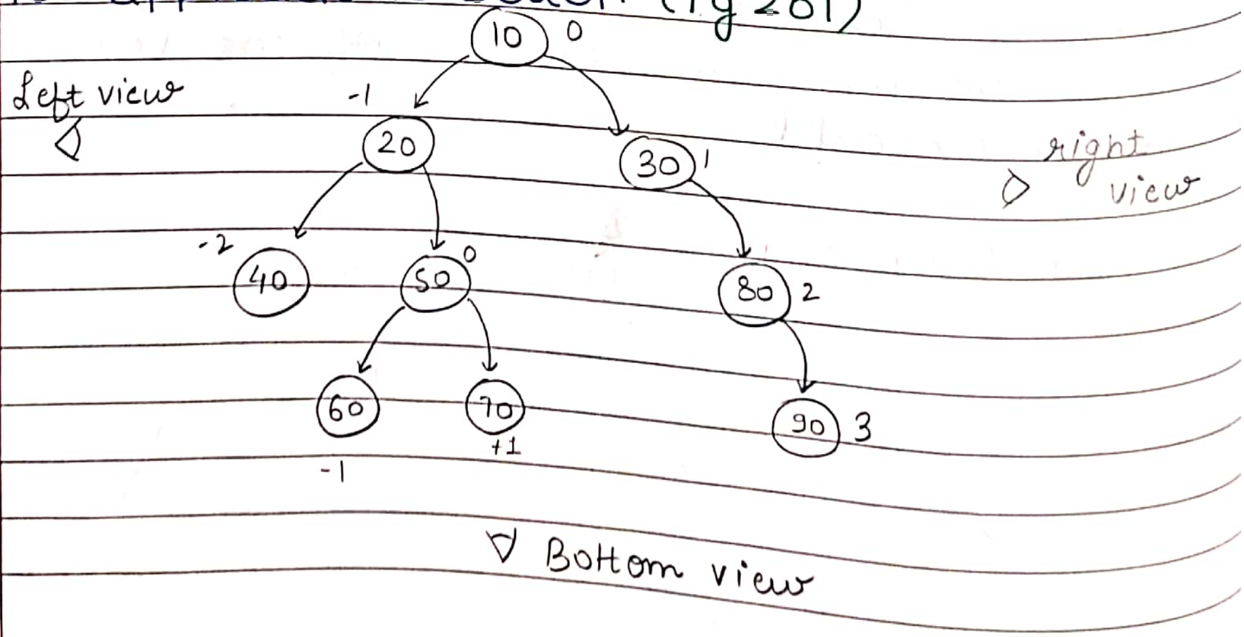
10 20 30 60 80 90

* Print nodes while returning from the recursive call. (RLN)

10 20 30 60 80 90 70 40 10

Here 10 gets printed twice. Handle it.

We can get stuck in the above code. Better way is using left view and right view. (X)
1st approach is better. (Pg 201)



Left view ⇒ 10 20 40 60
Right view ⇒ 10 30 80 90
Bottom view ⇒ 40 60 50 70 80 90
Top view ⇒ 40 20 10 30 80 90

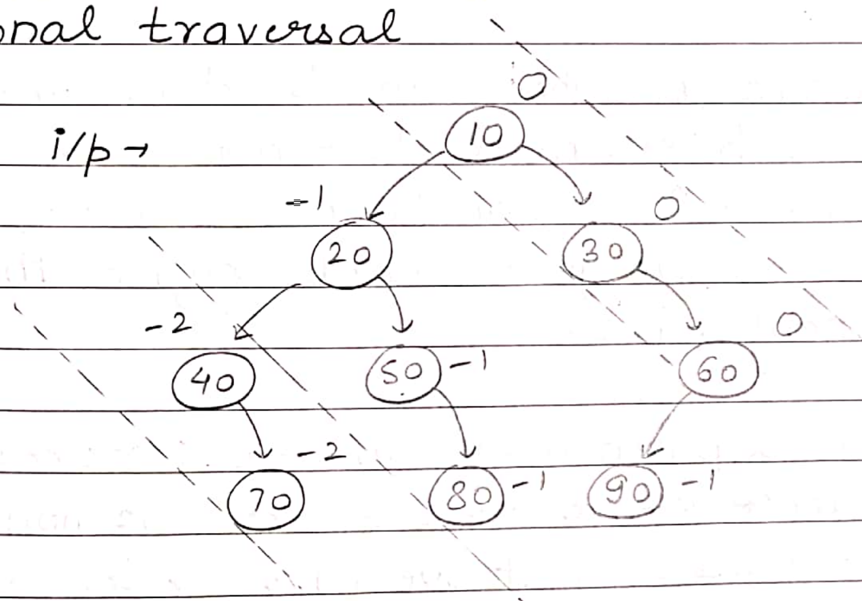
Now boundary traversal is easy.
✓ Left view
✓ Leaf nodes
✓ Right view in reverse order

} Some nodes will be printed twice & we need to handle it.

We need to put some conditions here. Hence this approach is not good.

Q6 Diagonal traversal

i/p →



When we go right, do nothing but when we go left then reduce d by -1.

0 ⇒ 10 30 60
-1 ⇒ 20 50 80 90
-2 ⇒ 40 70