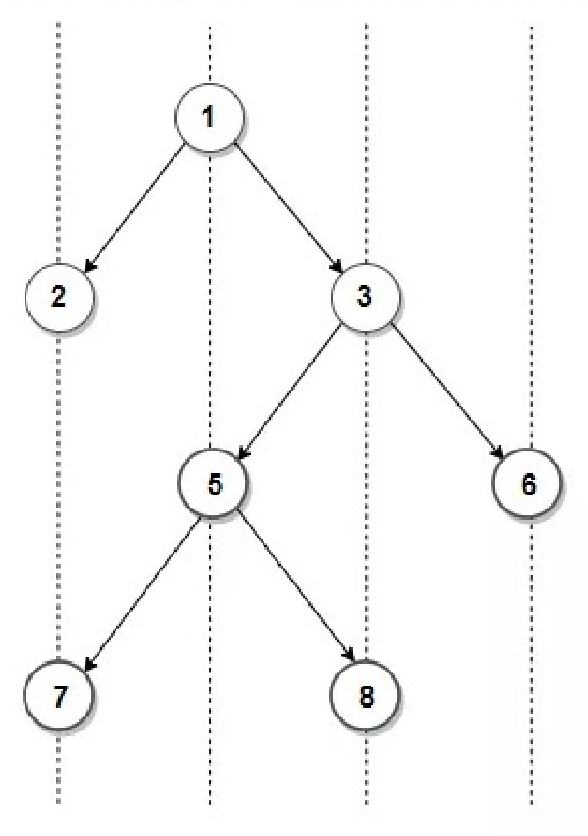
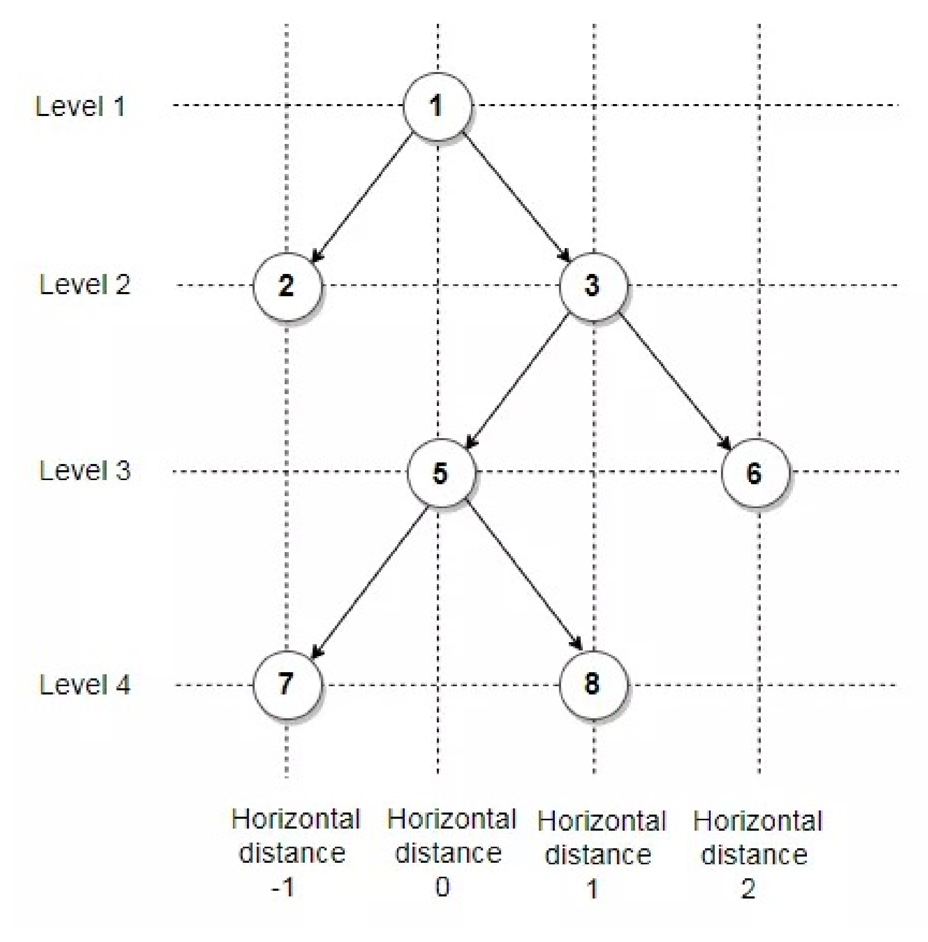
Editorial

X

Recursive Approach

The idea is to create an empty map where each key represents the relative horizontal distance of the node from the root node and value in the map maintains a pair containing node's value and its level number. Then we do a pre-order traversal of the tree and if current level of a node is less than maximum level seen so far for the same horizontal distance as current node's or current horizontal distance is seen for the first time, we update the value and the level for current horizontal distance in the map. For each node, we recurse for its left subtree by decreasing horizontal distance and increasing level by 1 and recurse for right subtree by increasing both level and horizontal distance by 1.





C++ Code for recursive approach

```
void topView1(node *root, int level, int dist, map<int, pair<int, int> >
    if(root == NULL) {
        return;
    }
    if(mp.find(dist) == mp.end() or level<mp[dist].second) {
        mp[dist] = {root->data, level};
    }
    topView1(root->left, level+1, dist-1, mp);
    topView1(root->right, level+1, dist+1, mp);
}
void topView(node *root)
{
   map<int, pair<int, int> >mp;
   topView1(root, 0, 0, mp);
   for(auto val:mp){
       cout<<val.second.first<<" ";
   }
}
```

Java Code for Recursive Approach

```
import java.util.*;

public class Practice {

    static private class TreeNode {
        int data;
        TreeNode left;
        TreeNode right;

        TreeNode(int d) {
            data = d;
            left = right = null;
        }
    }

    static Scanner sc = new Scanner(System.in);

    static TreeNode buildTreeLevelWise() {

        int d = sc.nextInt();

        TreeNode root = new TreeNode(d);
        Queue<TreeNode> q = new LinkedList<>();
        q.add(root);

        while (!q.isEmpty()) {

            TreeNode f = q.poll();
            int c1 = sc.nextInt();
            int c2 = sc.nextInt();

            if (c1 != -1) {
                f.left = new TreeNode(c1);
                q.add(f.left);
            }
            if (c2 != -1) {
                f.right = new TreeNode(c2);
                q.add(f.right);
            }
        }
        return root;
    }

    static void verticalOrderPrint(TreeNode root, Map<Integer, Integer>
        if (root == null) {
            return;
        }
        if (!m.containsKey(d)) {
            m.put(d, root.data);
        }
        verticalOrderPrint(root.left, m, d - 1);
        verticalOrderPrint(root.right, m, d + 1);
    }

    public static void main(String[] args) {
        TreeNode root = buildTreeLevelWise();
        Map<Integer, Integer> m = new TreeMap<>();
        verticalOrderPrint(root, m, 0);

        m.forEach((k, v) -> System.out.print(v + " "));
    }
}
```

Iterative Approach

Another approach can be using a queue.

The idea here is to observe that, if we try to see a tree from its top, then only the nodes which are at top in vertical order will be seen.
Start BFS from root. Maintain a queue of pairs comprising of node(Node *) type and vertical distance of node from root. Also, maintain a map which should store the node at a particular horizontal distance.
While processing a node, just check if any node is there in the map at that vertical distance.
If any node is there, it means the node can't be seen from top, do not consider it. Else, if there is no node at that vertical distance, store that in map and consider for top view.

C++ Code for Iterative Approach

```
void topViewIterative(node *root)
{
    queue<pair<node *, int>> qu;
    qu.push({root, 0});
    map<int, int> mp;
    while (!qu.empty())
    {
        node *temp = qu.front().first;
        int dist = qu.front().second;
        qu.pop();
        if (mp.find(dist) == mp.end())
        {
            mp[dist] = temp->data;
        }
        if (temp->left)
        {
            qu.push({temp->left, dist - 1});
        }
        if (temp->right)
        {
            qu.push({temp->right, dist + 1});
        }
    }

    for (auto x : mp)
    {
        cout << x.second << " ";
    }
}
```

Java Code for Iterative Approach

```
static void topViewIterative(TreeNode root) {
    Queue<NodeIntPair> qu = new LinkedList<>();
    qu.add(new NodeIntPair(root, 0));
    Map<Integer, Integer> mp = new TreeMap<>();
    while (!qu.isEmpty()) {
        TreeNode temp = qu.peek().first;
        int dist = qu.peek().second;
        qu.remove();
        if (!mp.containsKey(dist)) {
            mp.put(dist, temp.data);
        }
        if (temp.left != null) {
            qu.add(new NodeIntPair(temp.left, dist - 1));
        }
        if (temp.right != null) {
            qu.add(new NodeIntPair(temp.right, dist + 1));
        }
    }

    mp.forEach((k, v) -> System.out.print(v + " "));
}
```