Print Nodes in Top View of Binary Tree

Top view of a binary tree is the set of nodes visible when the tree is viewed from the top. Given a binary tree, print the top view of it. The output nodes can be printed in any order. Expected time complexity is O(n)

A node x is there in output if x is the topmost node at its horizontal distance. Horizontal distance of left child of a node x is equal to horizontal distance of x minus 1, and that of right child is horizontal distance of x plus 1.

```
       1
    /     \
   2       3
  /  \    / \
 4    5  6   7
Top view of the above binary tree is
4 2 1 3 7

        1
      /   \
    2       3
      \
        4
          \
            5
             \
               6
Top view of the above binary tree is
2 1 3 6
```

We strongly recommend to minimize your browser and try this yourself first.

The idea is to do something similar to vertical Order Traversal. Like vertical Order Traversal, we need to nodes of same horizontal distance together. We do a level order traversal so that the topmost node at a horizontal node is visited before any other node of same horizontal distance below it. Hashing is used to check if a node at given horizontal distance is seen or not.

```
// Java program to print top view of Binary tree
import java.util.*;

// Class for a tree node
class TreeNode
{
    // Members
    int key;
    TreeNode left, right;

    // Constructor
    public TreeNode(int key)
    {
        this.key = key;
        left = right = null;
    }
}

// A class to represent a queue item. The queue is used to do Level
// order traversal. Every Queue item contains node and horizontal
// distance of node from root
class QItem
{
   TreeNode node;
   int hd;
   public QItem(TreeNode n, int h)
   {
        node = n;
        hd = h;
   }
```

```
}

// Class for a Binary Tree
class Tree
{
    TreeNode root;

    // Constructors
    public Tree()  { root = null; }
    public Tree(TreeNode n) { root = n; }

    // This method prints nodes in top view of binary tree
    public void printTopView()
    {
        // base case
        if (root == null) {  return;  }

        // Creates an empty hashset
        HashSet<Integer> set = new HashSet<>();

        // Create a queue and add root to it
        Queue<QItem> Q = new LinkedList<QItem>();
        Q.add(new QItem(root, 0)); // Horizontal distance of root is 0

        // Standard BFS or level order traversal loop
        while (!Q.isEmpty())
        {
            // Remove the front item and get its details
            QItem qi = Q.remove();
            int hd = qi.hd;
            TreeNode n = qi.node;

            // If this is the first node at its horizontal distance,
            // then this node is in top view
            if (!set.contains(hd))
            {
                set.add(hd);
                System.out.print(n.key + " ");
            }

            // Enqueue left and right children of current node
            if (n.left != null)
                Q.add(new QItem(n.left, hd-1));
            if (n.right != null)
                Q.add(new QItem(n.right, hd+1));
        }
    }
}

// Driver class to test above methods
public class Main
{
    public static void main(String[] args)
    {
        /* Create following Binary Tree
             1
           /  \
          2    3
           \
            4
             \
              5
               \
                6*/
        TreeNode root = new TreeNode(1);
        root.left = new TreeNode(2);
        root.right = new TreeNode(3);
        root.left.right = new TreeNode(4);
        root.left.right.right = new TreeNode(5);
        root.left.right.right.right = new TreeNode(6);
        Tree t = new Tree(root);
        System.out.println("Following are nodes in top view of Binary Tree");
        t.printTopView();
```

```
        t.printTopView();
    }
}
```

Output:

```
Following are nodes in top view of Binary Tree
1 2 3 6
```

Time Complexity of the above implementation is O(n) where n is number of nodes in given binary tree. The assumption here is that add() and contains() methods of HashSet work in O(1) time.