Bottom View of a Binary Tree: - Given a Binary Tree, we need to print the bottom view from left to right. A node x is there in output if x is the bottommost 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.

20

/ \

8 22

/ \ \

5 3 25

/ \

10 14

For the above tree the output should be 5, 10, 3, 14, 25.

If there are multiple bottom-most nodes for a horizontal distance from root, then print the later one in level traversal. For example, in the below diagram, 3 and 4 are both the bottom-most nodes at horizontal distance 0, we need to print 4.

20

/ \

8 22

/ \ / \

5 3 4 25

/ \

10 14

For the above tree the output should be 5, 10, 4, 14, 25.

The following are steps to print Bottom View of Bianry Tree.  
1. We put tree nodes in a queue for the level order traversal.  
2. Start with the horizontal distance(hd) 0 of the root node, keep on adding left child to queue along with the horizontal distance as hd-1 and right child as hd+1.  
3. Also, use a TreeMap which stores key value pair sorted on key.  
4. Every time, we encounter a new horizontal distance or an existing horizontal distance put the node data for the horizontal distance as key. For the first time it will add to the map, next time it will replace the value. This will make sure that the bottom most element for that horizontal distance is present in the map and if you see the tree from beneath that you will see that element.

// Tree node class

class Node

{

    int data; //data of the node

    int hd; //horizontal distance of the node

    Node left, right; //left and right references

    // Constructor of tree node

    public Node(int key)

    {

        data = key;

        hd = Integer.MAX\_VALUE;

        left = right = null;

    }

}

//Tree class

class Tree

{

    Node root; //root node of tree

    // Default constructor

    public Tree() {}

    // Parameterized tree constructor

    public Tree(Node node)

    {

        root = node;

    }

    // Method that prints the bottom view.

    public void bottomView()

    {

        if (root == null)

            return;

        // Initialize a variable 'hd' with 0 for the root element.

        int hd = 0;

        // TreeMap which stores key value pair sorted on key value

        Map<Integer, Integer> map = new TreeMap<>();

         // Queue to store tree nodes in level order traversal

        Queue<Node> queue = new LinkedList<Node>();

        // Assign initialized horizontal distance value to root

        // node and add it to the queue.

        root.hd = hd;

        queue.add(root);

        // Loop until the queue is empty (standard level order loop)

        while (!queue.isEmpty())

        {

            Node temp = queue.remove();

            // Extract the horizontal distance value from the

            // dequeued tree node.

            hd = temp.hd;

            // Put the dequeued tree node to TreeMap having key

            // as horizontal distance. Every time we find a node

            // having same horizontal distance we need to replace

            // the data in the map.

            map.put(hd, temp.data);

            // If the dequeued node has a left child add it to the

            // queue with a horizontal distance hd-1.

            if (temp.left != null)

            {

                temp.left.hd = hd-1;

                queue.add(temp.left);

            }

            // If the dequeued node has a left child add it to the

            // queue with a horizontal distance hd+1.

            if (temp.right != null)

            {

                temp.right.hd = hd+1;

                queue.add(temp.right);

            }

        }

   // Extract the entries of map into a set to traverse an iterator over that.

        Set<Entry<Integer, Integer>> set = map.entrySet();

        // Make an iterator

        Iterator<Entry<Integer, Integer>> iterator = set.iterator();

        // Traverse the map elements using the iterator.

        while (iterator.hasNext())

        {

            Map.Entry<Integer, Integer> me = iterator.next();

            System.out.print(me.getValue()+" ");

        }

    }

}

public class BottomView

{

    public static void main(String[] args)

    {

        Node root = new Node(20);

        root.left = new Node(8);

        root.right = new Node(22);

        root.left.left = new Node(5);

        root.left.right = new Node(3);

        root.right.left = new Node(4);

        root.right.right = new Node(25);

        root.left.right.left = new Node(10);

        root.left.right.right = new Node(14);

        Tree tree = new Tree(root);

        System.out.println("Bottom view of the given binary tree:");

        tree.bottomView();

    }

}

Output:- Bottom view of the given binary tree:

5 10 4 14 25

**Exercise:-** Extend the above solution to print all bottommost nodes at a horizontal distance if there are multiple bottommost nodes. For the above second example, the output should be 5 10 3 4 14 25.

# 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

The idea is to do something similar to[**vertical Order Traversal**](http://www.geeksforgeeks.org/print-binary-tree-vertical-order-set-2/). Like [vertical Order Traversal](http://www.geeksforgeeks.org/print-binary-tree-vertical-order-set-2/), 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));

        }

    }

}

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();

    }

}

**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.

Asked in: [Amazon](http://practice.geeksforgeeks.org/company/Amazon/)