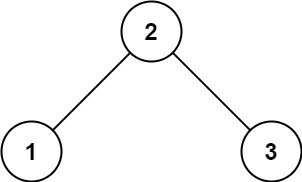
<https://leetcode.com/problems/find-bottom-left-tree-value/>

Given the root of a binary tree, return the leftmost value in the last row of the tree.

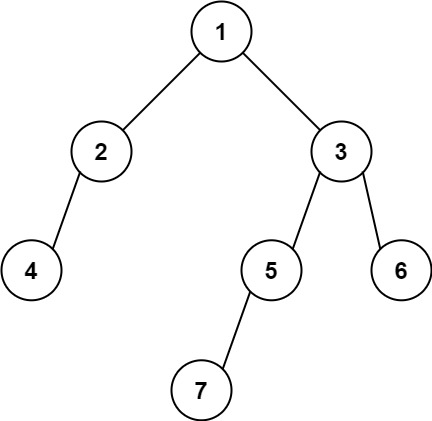
**Example 1:**



Input: root = [2,1,3]

Output: 1

**Example 2:**



Input: root = [1,2,3,4,null,5,6,null,null,7]

Output: 7

**Constraints:**

The number of nodes in the tree is in the range [1, 10^4].

-2^31 <= Node.val <= 2^31 - 1

**Attempt 1: 2023-06-23**

**Solution 1: BFS (10 min)**

**Style 1: Level Order Traversal**

**Traverse from left to right and only record the 1st element during each level traversal**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\*    int val;

\*    TreeNode left;

\*    TreeNode right;

\*    TreeNode() {}

\*    TreeNode(int val) { this.val = val; }

\*    TreeNode(int val, TreeNode left, TreeNode right) {

\*        this.val = val;

\*        this.left = left;

\*        this.right = right;

\*    }

\* }

\*/

class Solution {

    public int findBottomLeftValue(TreeNode root) {

        int result = 0;

        Queue<TreeNode> q = new LinkedList<TreeNode>();

        q.offer(root);

        while(!q.isEmpty()) {

            int size = q.size();

            for(int i = 0; i < size; i++) {

                TreeNode node = q.poll();

                if(i == 0) {

                    result = node.val;

                }

                if(node.left != null) {

                    q.offer(node.left);

                }

                if(node.right != null) {

                    q.offer(node.right);

                }

            }

        }

        return result;

    }

}

**Refer to**

<https://leetcode.com/problems/find-bottom-left-tree-value/solutions/98786/verbose-java-solution-binary-tree-level-order-traversal/>

Typical way to do binary tree level order traversal. Only additional step is to remember the first element of each level.

public class Solution {

    public int findLeftMostNode(TreeNode root) {

        if (root == null) return 0;

        int result = 0;

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

        queue.add(root);

        while (!queue.isEmpty()) {

            int size = queue.size();

            for (int i = 0; i < size; i++) {

                TreeNode node = queue.poll();

                if (i == 0) result = node.val;

                if (node.left != null) queue.add(node.left);

                if (node.right != null) queue.add(node.right);

            }

        }

        return result;

    }

}

**Style 2: Regular traversal but from right to left**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\*    int val;

\*    TreeNode left;

\*    TreeNode right;

\*    TreeNode() {}

\*    TreeNode(int val) { this.val = val; }

\*    TreeNode(int val, TreeNode left, TreeNode right) {

\*        this.val = val;

\*        this.left = left;

\*        this.right = right;

\*    }

\* }

\*/

class Solution {

    public int findBottomLeftValue(TreeNode root) {

        int result = 0;

        Queue<TreeNode> q = new LinkedList<TreeNode>();

        q.offer(root);

        while(!q.isEmpty()) {

            TreeNode node = q.poll();

            result = node.val;

            if(node.right != null) {

                q.offer(node.right);

            }

            if(node.left != null) {

                q.offer(node.left);

            }

        }

        return result;

    }

}

**Refer to**

<https://leetcode.com/problems/find-bottom-left-tree-value/solutions/98779/right-to-left-bfs-python-java/>

Doing BFS right-to-left means we can simply return the **last** node's value and don't have to keep track of the first node in the current row or even care about rows at all. Inspired by @fallcreek's solution (not published) which uses two nested loops to go row by row but already had the right-to-left idea making it easier. I just took that further.

public int findLeftMostNode(TreeNode root) {

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

    queue.add(root);

    while (!queue.isEmpty()) {

        root = queue.poll();

        if (root.right != null)

            queue.add(root.right);

        if (root.left != null)

            queue.add(root.left);

    }

    return root.val;

}

**Note:**

**The difference between BFS and Level Order Traversal:**

**Level order traverse will add level nodes size calculation and for loop based on this size, which BFS no need**

**Solution 2: DFS (30 min)**

**How to use DFS to tracking the first node on each depth ?**

**We can use DFS preorder traversal to find the leftmost node value of the last level, the mechanism is if and only if the level increase happening, we will record that node's value, because in preorder: root -> left -> right, when level increase, current root is the first node of new level for sure**

**Style 1: Classic DFS Traversal, no return type required since we pass a global variable 'result' through recursion and receive the result**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\*    int val;

\*    TreeNode left;

\*    TreeNode right;

\*    TreeNode() {}

\*    TreeNode(int val) { this.val = val; }

\*    TreeNode(int val, TreeNode left, TreeNode right) {

\*        this.val = val;

\*        this.left = left;

\*        this.right = right;

\*    }

\* }

\*/

class Solution {

    // We can use DFS preorder traversal to find the leftmost node value

    // of the last level, the mechanism is if and only if the level increase

    // happening, we will record that node's value, because in preorder:

    // root -> left -> right, when level increase, current root is the first

    // node of new level for sure

    int curDepth = 0;

    int result = 0;

    public int findBottomLeftValue(TreeNode root) {

        helper(root, 1);

        return result;

    }

    private void helper(TreeNode root, int depth) {

        if(root == null) {

            return;

        }

        if(curDepth < depth) {

            curDepth = depth;

            result = root.val;

        }

        if(root.left != null) {

            helper(root.left, depth + 1);

        }

        if(root.right != null) {

            helper(root.right, depth + 1);

        }

    }

}

**Style 2: Still DFS Traversal, but with return type, but even with return type, its NOT Divide and Conquer, since its NOT based on "1.base case -> 2.递归成为更小的问题 -> 3.进行当前层的处理计算", we still do 3 before 2, and we don't pass a global variable 'result' through recursion and receive the result, but pass an object int[] {recorded curDepth, tracking value} to record depth and left most node value on each recursion**

**With if(root != null)**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\*    int val;

\*    TreeNode left;

\*    TreeNode right;

\*    TreeNode() {}

\*    TreeNode(int val) { this.val = val; }

\*    TreeNode(int val, TreeNode left, TreeNode right) {

\*        this.val = val;

\*        this.left = left;

\*        this.right = right;

\*    }

\* }

\*/

class Solution {

    // We can use DFS preorder traversal to find the leftmost node value

    // of the last level, the mechanism is if and only if the level increase

    // happening, we will record that node's value, because in preorder:

    // root -> left -> right, when level increase, current root is the first

    // node of new level for sure

    public int findBottomLeftValue(TreeNode root) {

        // To replace global variable "int curDepth = 0; int result = 0;" and

        // also keep tracking through the recursion, we have to use an object

        // and pass as a parameter into recursive method, the primitive type

        // cannot do this

        // initialize int[]{0, 0} as {recorded curDepth, tracking value}

        return helper(root, 1, new int[] {0, 0});

    }

    private int helper(TreeNode root, int depth, int[] result) {

        if(root == null) {

            return 0;

        }

        if(result[0] < depth) {

            result[0] = depth;

            result[1] = root.val;

        }

        // No return in branch !!!

        // Test case: root = [1,2,3,4,null,5,6,null,null,7]

        //          1

        //        /    \

        //      2      3

        //      /      / \

        //    4      5  6

        //            /

        //          7

        // output 4, expected 7

        //if(root.left != null) {

            // return helper(root.left, depth + 1, result);

            helper(root.left, depth + 1, result);

        //}

        //if(root.right != null) {

            // return helper(root.right, depth + 1, result);

            helper(root.right, depth + 1, result);

        //}

        return result[1];

    }

}

**Without if(root != null) but with if(root.left != null) and if(root.right != null) {**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\*    int val;

\*    TreeNode left;

\*    TreeNode right;

\*    TreeNode() {}

\*    TreeNode(int val) { this.val = val; }

\*    TreeNode(int val, TreeNode left, TreeNode right) {

\*        this.val = val;

\*        this.left = left;

\*        this.right = right;

\*    }

\* }

\*/

class Solution {

    // We can use DFS preorder traversal to find the leftmost node value

    // of the last level, the mechanism is if and only if the level increase

    // happening, we will record that node's value, because in preorder:

    // root -> left -> right, when level increase, current root is the first

    // node of new level for sure

    public int findBottomLeftValue(TreeNode root) {

        // To replace global variable "int curDepth = 0; int result = 0;" and

        // also keep tracking through the recursion, we have to use an object

        // and pass as a parameter into recursive method, the primitive type

        // cannot do this

        // initialize int[]{0, 0} as {recorded curDepth, tracking value}

        return helper(root, 1, new int[] {0, 0});

    }

    private int helper(TreeNode root, int depth, int[] result) {

        // Based on condition as if(root.left != null) / if(root.right != null),

        // if(root == null) won't reachable because after leaf node, the left

        // or right are both null, we will block early by above left/right != null

        // conditions

        //if(root == null) {

        //    return 0;

        //}

        if(result[0] < depth) {

            result[0] = depth;

            result[1] = root.val;

        }

        // No return in branch !!!

        // Test case: root = [1,2,3,4,null,5,6,null,null,7]

        //          1

        //        /    \

        //      2      3

        //      /      / \

        //    4      5  6

        //            /

        //          7

        // output 4, expected 7

        if(root.left != null) {

            // return helper(root.left, depth + 1, result);

            helper(root.left, depth + 1, result);

        }

        if(root.right != null) {

            // return helper(root.right, depth + 1, result);

            helper(root.right, depth + 1, result);

        }

        return result[1];

    }

}

**Refer to**

<https://leetcode.com/problems/find-bottom-left-tree-value/solutions/98802/simple-java-solution-beats-100-0/>

public class Solution {

    int ans=0, h=0;

    public int findBottomLeftValue(TreeNode root) {

        findBottomLeftValue(root, 1);

        return ans;

    }

    public void findBottomLeftValue(TreeNode root, int depth) {

        if (h<depth) {ans=root.val;h=depth;}

        if (root.left!=null) findBottomLeftValue(root.left, depth+1);

        if (root.right!=null) findBottomLeftValue(root.right, depth+1);

    }

}

No global variables, 6ms (faster):

public class Solution {

    public int findBottomLeftValue(TreeNode root) {

        return findBottomLeftValue(root, 1, new int[]{0,0});

    }

    public int findBottomLeftValue(TreeNode root, int depth, int[] res) {

        if (res[1]<depth) {res[0]=root.val;res[1]=depth;}

        if (root.left!=null) findBottomLeftValue(root.left, depth+1, res);

        if (root.right!=null) findBottomLeftValue(root.right, depth+1, res);

        return res[0];

    }

}

**Refer to**

[L104.Maximum Depth of Binary Tree (Ref.L222)](note://DFEAE5072F224B7F85934A11F3381C70)