<https://leetcode.ca/all/314.html>

Given a binary tree, return the *vertical order* traversal of its nodes' values. (ie, from top to bottom, column by column).

If two nodes are in the same row and column, the order should be from **left to right**.

**Examples 1:**

Input: [3,9,20,null,null,15,7]

3

/\

/ \

9 20

/\

/ \

15 7

Output:

[

[9],

[3,15],

[20],

[7]

]

**Examples 2:**

Input: [3,9,8,4,0,1,7]

3

/\

/ \

9 8

/\ /\

/ \/ \

4 01 7

Output:

[

[4],

[9],

[3,0,1],

[8],

[7]

]

**Examples 3:**

Input: [3,9,8,4,0,1,7,null,null,null,2,5] (0's right child is 2 and 1's left child is 5)

3

/\

/ \

9 8

/\ /\

/ \/ \

4 01 7

/\

/ \

5 2

Output:

[

[4],

[9,5],

[3,0,1],

[8,2],

[7]

]

**Attempt 1: 2024-06-16**

**Solution 1: BFS + Hash Table (10 min)**

import java.util.\*;

/\*\*

\* 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 {

// Definition of Pair class to hold a TreeNode and an Integer value together

private static class Pair {

TreeNode node;

Integer value;

public Pair(TreeNode node, Integer value) {

this.node = node;

this.value = value;

}

public TreeNode getKey() {

return node;

}

public Integer getValue() {

return value;

}

}

public List<List<Integer>> verticalOrder(TreeNode root) {

// Initialize result list

List<List<Integer>> result = new ArrayList<>();

// Early return if root is null

if (root == null) {

return result;

}

// Create a deque to perform a level-order traversal

Deque<Pair> queue = new ArrayDeque<>();

// Offering the root node with column value 0

queue.offer(new Pair(root, 0));

// TreeMap to hold nodes values grouped by their column number

TreeMap<Integer, List<Integer>> columnMap = new TreeMap<>();

// While there are nodes in the queue, process each level

while (!queue.isEmpty()) {

Pair currentPair = queue.pollFirst();

TreeNode currentNode = currentPair.getKey();

int column = currentPair.getValue();

// Add the current node's value to the corresponding column list

columnMap.computeIfAbsent(column, k -> new ArrayList<>()).add(currentNode.val);

// Offer the left child with column - 1 if it exists

if (currentNode.left != null) {

queue.offer(new Pair(currentNode.left, column - 1));

}

// Offer the right child with column + 1 if it exists

if (currentNode.right != null) {

queue.offer(new Pair(currentNode.right, column + 1));

}

}

// Add each column's list of nodes to result list and return it

result.addAll(columnMap.values());

return result;

}

}

Time Complexity: O(NlogN), sicne the order should be from left to right, we have to sort, so require NlogN

Space Complexity: O(N)

**Refer to**

<https://www.cnblogs.com/grandyang/p/5278930.html>

这道题让我们竖直遍历二叉树，并把每一列存入一个二维数组，看题目中给的第一个例子，3和 15 属于同一列，3在前，第二个例子中，3,5,2 在同一列，3在前，5和2紧随其后，那么隐约的可以感觉到好像是一种层序遍历的前后顺序，如何来确定列的顺序呢，这里可以把根节点给个序号0，然后开始层序遍历，凡是左子节点则序号减1，右子节点序号加1，这样可以通过序号来把相同列的节点值放到一起，用一个 TreeMap 来建立序号和其对应的节点值的映射，用 TreeMap 的另一个好处是其自动排序功能可以让列从左到右，由于层序遍历需要用到 queue，此时 queue 里不能只存节点，而是要存序号和节点组成的 pair 对儿，这样每次取出就可以操作序号，而且排入队中的节点也赋上其正确的序号，代码如下：

class Solution {

public:

vector<vector<int>> verticalOrder(TreeNode\* root) {

vector<vector<int>> res;

if (!root) return res;

map<int, vector<int>> m;

queue<pair<int, TreeNode\*>> q;

q.push({0, root});

while (!q.empty()) {

auto a = q.front(); q.pop();

m[a.first].push\_back(a.second->val);

if (a.second->left) q.push({a.first - 1, a.second->left});

if (a.second->right) q.push({a.first + 1, a.second->right});

}

for (auto a : m) {

res.push\_back(a.second);

}

return res;

}

};

**Refer to**

<https://algo.monster/liteproblems/314>

**Problem Description**

The given problem entails processing a binary tree to perform a vertical order traversal. The binary tree is a data structure where each node has at most two children referred to as the left and right child. In a vertical order traversal, we are required to group and print out the values of the nodes that are in the same vertical level—imagine drawing vertical lines through the nodes. The nodes that intersect with the same line are in the same vertical level and should be grouped together.

When we refer to "from top to bottom, column by column," we mean that the output should be arranged so that values in higher rows of the tree are presented before lower ones, and for values in the same row, they should be listed from left to right. If we visualize the tree spatially, imagine that the root node is at position 0, nodes to the left are in negative positions (-1, -2, ...), and nodes to the right are in positive positions (+1, +2, ...). The goal is to list all nodes' values at position -n, then -n+1, all the way to 0, and then +1, +2, ... +n.

**Intuition**

To achieve the vertical order traversal, one intuitive approach is to perform a level-order traversal (also known as a breadth-first search), while at the same time tracking the horizontal distance from the root node for each node encountered. We need to keep a data structure to maintain groups of nodes that share the same vertical level—this can be facilitated by a dictionary where the key represents the vertical level (the horizontal distance) and the value is a list containing the nodes' values at that level.

Here are the steps we might intuitively follow to come up with the solution:

We start with the root node, which will be at horizontal distance 0.

We use a queue to perform the level-order traversal, and as we dequeue a node, we record its value in our dictionary under its associated horizontal distance.

For every node we process, we enqueue its left child (if not null) with the horizontal distance decremented by 1, and its right child (if not null) with the horizontal distance incremented by 1.

After we have traversed the entire tree, we will have a dictionary filled with vertical levels as keys and lists of node values as values. The only thing left would be to sort these keys and output the associated lists in the sorted order.

Using this approach, we ensure that nodes higher up in the tree are processed before lower nodes due to the nature of level-order traversal; and within the same level, from left to right as the left child is enqueued before the right child. In the end, when sorting the keys of our dictionary, we guarantee that the output respects the "column by column" requirement.

**Solution Approach**

To implement the vertical order traversal, the Reference Solution Approach uses several programming concepts and data structures:

Breadth-first search (BFS): This algorithm is used for traversing the binary tree level by level from the root. It utilizes a queue (in Python, a deque) to process nodes in a first-in-first-out (FIFO) manner, which aligns with the BFS traversal pattern.

Queue (deque): A deque (double-ended queue) is used here as it provides an efficient way to pop elements from the front (with popleft()) while new nodes are added to the back (with append()). For each iteration, nodes at the current level are popped and their children are added to the queue for subsequent levels.

Dictionary: The defaultdict is a type of dictionary that automatically creates a new entry with a default value (list in this case) if the key doesn't exist. In the given problem, the key is the horizontal offset and the value is a list that accumulates nodes' values at that offset.

Sorting: Finally, the keys representing the horizontal offsets need to be sorted to ensure the "column by column" output. This is done at the end of the BFS, once all the nodes have been processed.

Now, let's break down the steps of the implementation matching with the code:

Initialize an empty deque named q and a defaultdict named d. We start by adding a tuple containing the root node and its horizontal offset 0 to q. (q = deque([(root, 0)]))

A while loop is used to iterate over q until it's empty, indicating that all nodes have been processed. On each iteration, it processes all nodes at the current level before moving to the next.

Within the loop, iterate over the number of elements currently in the queue, which represents the nodes at the current level. For each element:

Pop it from the queue (root, offset = q.popleft()).

Append the node's value to the list in the dictionary associated with the current horizontal offset (d[offset].append(root.val)).

After processing the current node, if it has a left child, append this child along with the updated offset (offset - 1) to q. Similarly, if there's a right child, append it with offset + 1.

After the while loop, the dictionary now contains all the values grouped by their vertical levels but not necessarily in the correct order. Using list comprehension, we create a sorted list from the dictionary items ([v for \_, v in sorted(d.items())]), where \_ is a placeholder for the key (offset) which we are not explicitly using in the final list.

By sorting the items of the dictionary based on the keys (which are offsets), and then selecting only the values (lists of node values), we obtain the final vertical order traversal of the binary tree.

**Example Walkthrough**

Let's consider a simple binary tree as our example to illustrate the solution approach:

1

/ \

2 3

/ \

4 5

Here's how we would perform the vertical order traversal:

Begin with the root node 1, which is at horizontal distance 0. Our queue q will initially have the entry [(1, 0)] and our dictionary d is empty.

We process the nodes level by level. For node 1, we dequeue it, adding 1 to the list in d for key 0 (d[0] = [1]).

Then, we add its children nodes 2 with horizontal distance -1 and 3 with horizontal distance +1 to the queue (q = [(2, -1), (3, +1)]).

Proceeding with the BFS, we dequeue the next element from q, which is (2, -1). Node 2's value is then added to d[-1]. After this, we enqueue its child 4 with horizontal distance -2 (q = [(3, +1), (4, -2)]).

Next, we dequeue (3, +1) from the queue and add 3 to d[1]. Node 3 has a right child 5, which we will enqueue with horizontal distance +2 (q = [(4, -2), (5, +2)]).

Continue until the queue is empty; dequeue (4, -2) and add 4 to d[-2]. 4 has no children, so the queue now is [(5, +2)].

Finally, dequeue (5, +2) and add 5 to d[2]. Now the queue is empty, and our dictionary d is filled with the sorted keys/values like so:

{

-2: [4],

-1: [2],

0: [1],

1: [3],

2: [5]

}

The last step is to sort the keys of the dictionary and arrange the values. The keys when sorted give -2, -1, 0, 1, 2. Taking the associated lists, we can now create our vertically ordered list:

[

[4], // for horizontal distance -2

[2], // for horizontal distance -1

[1], // for horizontal distance 0

[3], // for horizontal distance 1

[5] // for horizontal distance 2

]

Thus, the vertical order traversal of the tree is [[4], [2], [1], [3], [5]]. This output respects the vertical levels of the tree from left to right and top to bottom.

**Solution Implementation**

import java.util.\*;

/\*\*

\* 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 {

// Definition of Pair class to hold a TreeNode and an Integer value together

private static class Pair {

TreeNode node;

Integer value;

public Pair(TreeNode node, Integer value) {

this.node = node;

this.value = value;

}

public TreeNode getKey() {

return node;

}

public Integer getValue() {

return value;

}

}

public List<List<Integer>> verticalOrder(TreeNode root) {

// Initialize result list

List<List<Integer>> result = new ArrayList<>();

// Early return if root is null

if (root == null) {

return result;

}

// Create a deque to perform a level-order traversal

Deque<Pair> queue = new ArrayDeque<>();

// Offering the root node with column value 0

queue.offer(new Pair(root, 0));

// TreeMap to hold nodes values grouped by their column number

TreeMap<Integer, List<Integer>> columnMap = new TreeMap<>();

// While there are nodes in the queue, process each level

while (!queue.isEmpty()) {

Pair currentPair = queue.pollFirst();

TreeNode currentNode = currentPair.getKey();

int column = currentPair.getValue();

// Add the current node's value to the corresponding column list

columnMap.computeIfAbsent(column, k -> new ArrayList<>()).add(currentNode.val);

// Offer the left child with column - 1 if it exists

if (currentNode.left != null) {

queue.offer(new Pair(currentNode.left, column - 1));

}

// Offer the right child with column + 1 if it exists

if (currentNode.right != null) {

queue.offer(new Pair(currentNode.right, column + 1));

}

}

// Add each column's list of nodes to result list and return it

result.addAll(columnMap.values());

return result;

}

}

**Time and Space Complexity**

**Time Complexity**

The time complexity of the code is determined by the number of nodes in the binary tree and the number of operations performed for each node.

Since the code traverses each node exactly once using a breadth-first search (BFS) approach, the iteration over nodes contributes O(N) to the time complexity, where N is the number of nodes in the tree.

For each node, few constant-time operations are performed, such as appending to a list and adding nodes to the queue. Therefore, these operations don't affect the overall time complexity which remains linear with respect to the number of nodes.

In addition, after the BFS traversal is done, the dictionary d that has been constructed is sorted by keys (vertical columns indices). If there are k entries (unique vertical indices), the sorting will take O(k log k) time.

Combining the two parts, the total time complexity is O(N + k log k). However, since in the worst case, k could be as large as N (imagine a skewed tree), the time complexity can be simplified to O(N log N).

**Space Complexity**

The space complexity is determined by the space needed to store the output and the data structures used during the BFS traversal.

The space taken by the dictionary d is O(N), as it stores a list of nodes for each unique vertical index. In a complete binary tree, k can be O(N), hence space complexity for d can reach O(N).

The queue q used for BFS might at most contain all the nodes at the widest level of the tree. In a perfect binary tree, this could be as much as N/2 (the last level of the tree). Therefore, the space complexity for q is O(N).

Since the output list is basically the values from the dictionary sorted by keys and does not occupy additional space unrelated to input, and considering both q and d, the overall space complexity of the algorithm is O(N).