Binary Tree Vertical Order Traversal

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:
Given binary tree `[3,9,20,null,null,15,7]`,

```
    3
   / \
  9  20
    /  \
   15   7
```

return its vertical order traversal as:

```
[
  [9],
  [3,15],
  [20],
  [7]
]
```

Given binary tree `[3,9,20,4,5,2,7]`,

```
    _3_
   /   \
  9    20
 / \   / \
4   5 2   7
```

return its vertical order traversal as:

```
[
  [4],
  [9],
  [3,5,2],
  [20],
  [7]
]
```

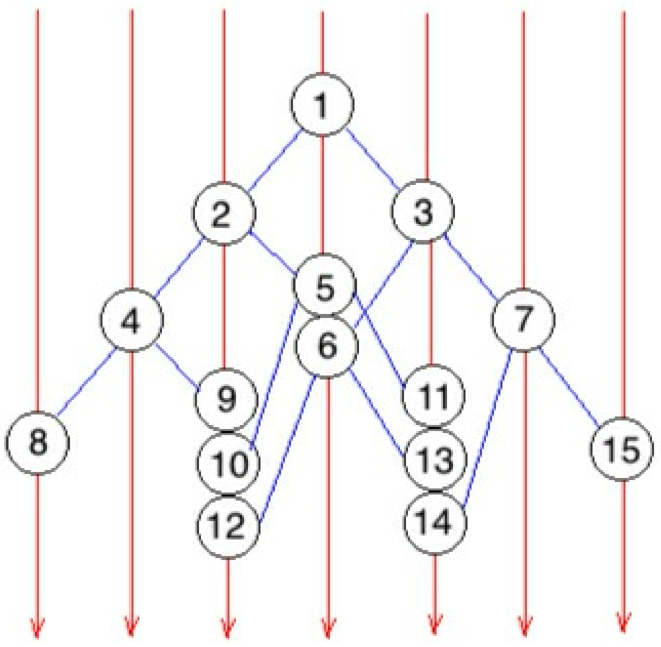
Solution 1

The following solution takes `5ms`.

- BFS, put `node`, `col` into queue at the same time
- Every left child access `col - 1` while right child `col + 1`
- This maps `node` into different `col` buckets
- Get `col` boundary `min` and `max` on the fly
- Retrieve `result` from `cols`

Note that `TreeMap` version takes `9ms`.

Here is an example of `[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]`. Notice that every child access changes one column bucket id. So `12` actually goes ahead of `11`.



```java
public List<List<Integer>> verticalOrder(TreeNode root) {
    List<List<Integer>> res = new ArrayList<>();
    if(root == null) return res;

    Map<Integer, ArrayList<Integer>> map = new HashMap<>();
    Queue<TreeNode> q = new LinkedList<>();
    Queue<Integer> cols = new LinkedList<>();

    q.add(root);
    cols.add(0);

    int min = 0, max = 0;
    while(!q.isEmpty()) {
        TreeNode node = q.poll();
        int col = cols.poll();
        if(!map.containsKey(col)) map.put(col, new ArrayList<Integer>());
        map.get(col).add(node.val);

        if(node.left != null) {
            q.add(node.left);
            cols.add(col - 1);
            if(col <= min) min = col - 1;
        }
        if(node.right != null) {
            q.add(node.right);
            cols.add(col + 1);
            if(col >= max) max = col + 1;
        }
    }

    for(int i = min; i <= max; i++) {
        res.add(map.get(i));
    }

    return res;
}
```

written by yavinci original link here

Solution 2

```
def verticalOrder(self, root):
    cols = collections.defaultdict(list)
    queue = [(root, 0)]
    for node, i in queue:
        if node:
            cols[i].append(node.val)
            queue += (node.left, i - 1), (node.right, i + 1)
    return [cols[i] for i in sorted(cols)]
```

written by StefanPochmann original link here

Solution 3

```java
private int min = 0, max = 0;
public List<List<Integer>> verticalOrder(TreeNode root) {
    List<List<Integer>> list = new ArrayList<>();
    if(root == null)    return list;
    computeRange(root, 0);
    for(int i = min; i <= max; i++) list.add(new ArrayList<>());
    Queue<TreeNode> q = new LinkedList<>();
    Queue<Integer> idx = new LinkedList<>();
    idx.add(-min);
    q.add(root);
    while(!q.isEmpty()){
        TreeNode node = q.poll();
        int i = idx.poll();
        list.get(i).add(node.val);
        if(node.left != null){
            q.add(node.left);
            idx.add(i - 1);
        }
        if(node.right != null){
            q.add(node.right);
            idx.add(i + 1);
        }
    }
    return list;
}
private void computeRange(TreeNode root, int idx){
    if(root == null)    return;
    min = Math.min(min, idx);
    max = Math.max(max, idx);
    computeRange(root.left, idx - 1);
    computeRange(root.right, idx + 1);
}
```

There is no difference when using HashMap. Since by using HashMap it need keep track of min and max as well, I'd rather directly insert into list by computing min and max in advance.

written by Jinx_boom original link here