Previous Next

429. N-ary Tree Level Order Traversal

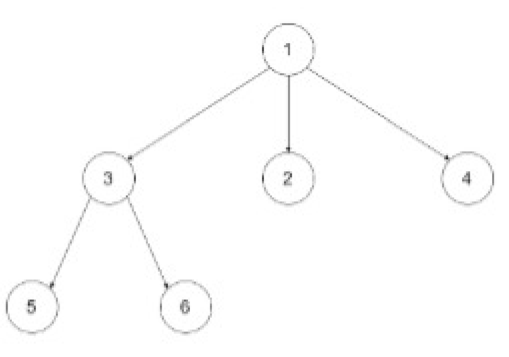
Nov. 6, 2019 | 5.7K views

Average Rating: 4.43 (7 votes)

Given an n-ary tree, return the *level order* traversal of its nodes' values.

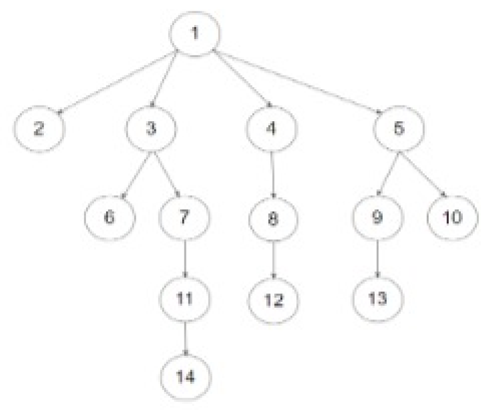
Nary-Tree input serialization is represented in their level order traversal, each group of children is separated by the null value (See examples).

Example 1:




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

Example 2:




```
Input: root = [1,null,2,3,4,5,null,null,6,7,null,8,null,9,10,null,null,11,null,12,null
Output: [[1],[2,3,4,5],[6,7,8,9,10],[11,12,13],[14]]
```

Constraints:

- The height of the n-ary tree is less than or equal to `1000`
- The total number of nodes is between `[0, 10^4]`

Solution

Approach 1: Breadth-first Search using a Queue

Intuition

We want to make a list of sub-lists, where each sub-list is the values from one *row* in the tree. The rows should be in the order they appear from top to bottom.

Because we're traversing the tree, starting with the nodes nearest to the root and then working our way down to the nodes furthest from the root, this is a type of **breadth-first search**. To do a breadth-first search, we use a **queue**. Recall that a queue is a data structure that we put items in one end, and take them out of the other. We call it First In, First Out (FIFO) because the first items that go in should be the first to come out. It works the same way as the queue you have to wait in to get into a busy stadium.

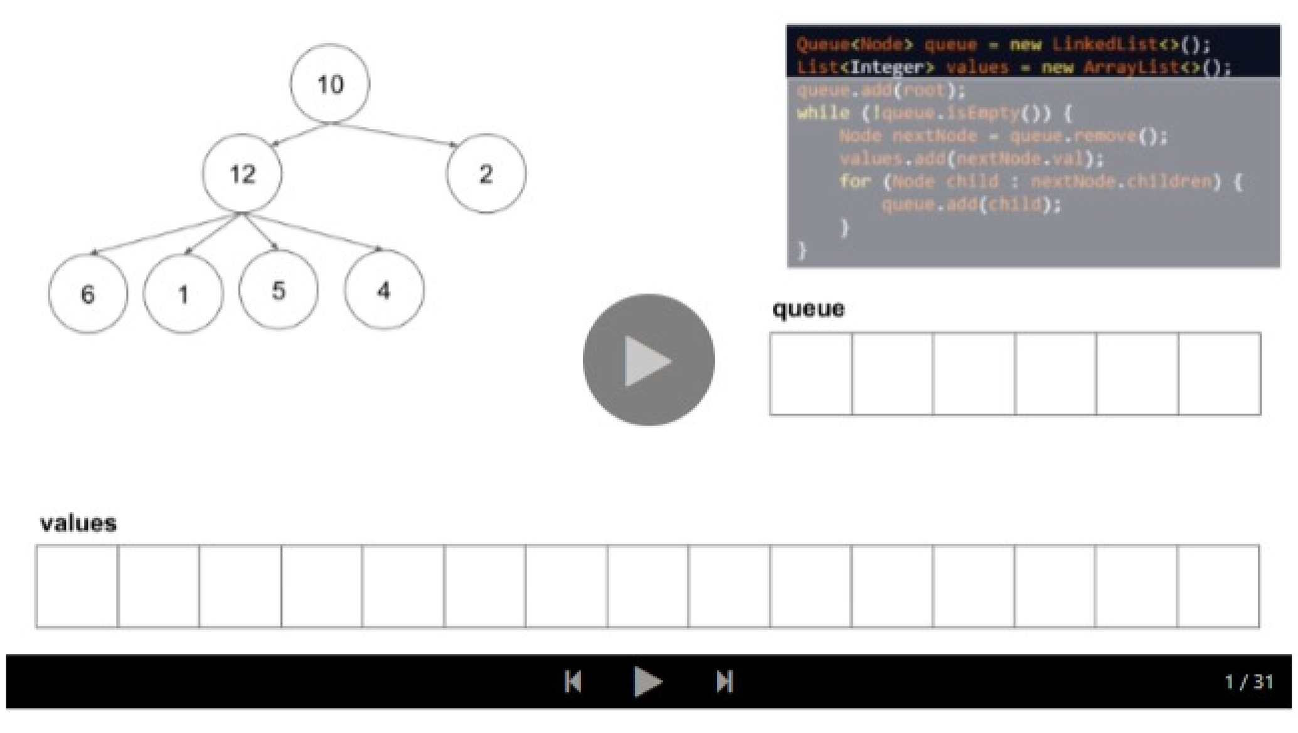
A **stack** would be the *wrong* data structure to use here. Stacks are used for **depth-first search** (there is a convoluted way you could do it, but it's not *sensible*).

Let's start by using the most basic of queue-based traversal algorithms on the tree to see what it does. This is a fundamental algorithm you should be aiming to *memorize*.

```
List<Integer> values = new ArrayList<>();
Queue<Node> queue = new LinkedList<>();
queue.add(root);
while (!queue.isEmpty()) {
    Node nextNode = queue.remove();
    values.add(nextNode.val);
    for (Node child : nextNode.children) {
        queue.add(child);
    }
}
```

Make a list to put integers in, and a queue to put nodes on. Put the root node onto the queue, and then while the queue is not empty, take a node off the queue, add its value to the list, and add each of its children onto the queue. Note that we are putting **Node** objects onto the queue, *not* integers. If we were to only put the integer values, we'd have no way of getting the child nodes out of them.

Let's see what we get when we use this algorithm to traverse the tree (don't worry about the inner lists yet, we're ensuring we have a way of getting the nodes from left to right and then top to bottom).



Indeed, it does return the nodes from left to right, then top to bottom. Next we'll be looking at how we can take this basic algorithm and modify it so that we have each level in its own sub-list. Have a go at modifying the above algorithm on your own first.

Algorithm

The basic breadth-first search algorithm above got us part of the way, but we still need to do those sub-lists, and also make sure our code works if the root is null (a tree with no data).

We need to create a new sub-list each time we're starting a new layer, and we need to insert all nodes from the layer into that sub-list. A good way we can do this is by checking the current `size` of the queue at the start of the `while` loop body. Then, we can have another loop that processes that number of nodes. This way, we are guaranteed to be processing *one layer* for *each* iteration of the `while` loop so can put all nodes within the same iteration into the same sub-list. On the first iteration of the while loop, we only have **1** node: the root node. So we'll loop around the inner loop once, removing the root node, and put all of its children onto the queue. Then in the second iteration, we'll remove all the children from the queue (as that's the number of times we'll loop around the inner loop) and put all the grandchildren onto the queue. And so forth.

It's very important to use a **Queue** type for `nodesToExplore`, and *not* a Vector, List, or Array. Removing items off the front of those other data structures is an $O(n)$ operation because all the remaining elements are moved along to fill the gap. A Queue is designed so that it is $O(1)$.

Java | Python | Copy

```
def levelOrder(self, root: 'Node') -> List[List[int]]:
    if root is None:
        return []
    result = []
    queue = collections.deque([root])
    while queue:
        level = []
        for _ in range(len(queue)):
            node = queue.popleft()
            level.append(node.val)
            queue.extend(node.children)
        result.append(level)
    return result
```

Complexity Analysis

- Time complexity : $O(n)$, where n is the number of nodes.

  Each node is getting added to the queue, removed from the queue, and added to the result exactly once.

- Space complexity : $O(n)$.

  We are using a queue to keep track of nodes we still need to visit the children of. At most, the queue will have 2 layers of the tree on it at any given time. In the worst case, this is all of the nodes. In the best case, it is just 1 node (if we have a tree that is equivalent to a linked list). The average case is difficult to calculate without knowing something of the trees we can expect to see, but in balanced trees, half or more of the nodes are often in the lowest 2 layers. So we should go with the worst case of $O(n)$, and know that the average case is probably similar.

Approach 2: Simplified Breadth-first Search

Intuition

A variant of the above approach is to make a new list on each iteration instead of using a single queue. This makes the code slightly simpler because we lose the `size` variable and the counting loop, which are a potential source of off-by-one errors.

Algorithm

Java | Python | Copy

```
def levelOrder(self, root: 'Node') -> List[List[int]]:
    if root is None:
        return []

    result = []
    previous_layer = [root]

    while previous_layer:
        current_layer = []
        result.append([])
        for node in previous_layer:
            result[-1].append(node.val)
            current_layer.extend(node.children)
        previous_layer = current_layer
    return result
```

Complexity Analysis

- Time complexity : $O(n)$, where n is the number of nodes.
- Space complexity : $O(n)$. Same as above, we always have lists containing levels of nodes.

Approach 3: Recursion

Intuition

We can use recursion for this problem. Often we can't use recursion for a breadth-first search (which is what a level-order traversal is). That is because breadth-first search is based on using a **queue**, whereas recursion is using the runtime **stack** and so is suited to depth-first search. In this case, however, we are putting all the values into a list before returning it. This means it's okay for us to get them in a different order to what they'll appear in the final list. As long as we know what level each node is from, and ensure they are in the correct order within each level, it will work.

Algorithm

Java | Python | Copy

```
def levelOrder(self, root: 'Node') -> List[List[int]]:

    def traverse_node(node, level):
        if len(result) == level:
            result.append([])
        result[level].append(node.val)
        for child in node.children:
            traverse_node(child, level + 1)

    result = []

    if root is not None:
        traverse_node(root, 0)
    return result
```

The iterative approach traversed the nodes in level order, whereas the recursive approach traversed them from left to right. While it was still easy to put them into the correct order using the recursive approach for this particular question, it could be problematic in practice. Often when we do a level-order traversal (or a breadth-first search), we are using the **Iterator** pattern and instead of storing the values in a list like we did here, the nodes are obtained one-by-one and processed. The iterator approach getting the nodes in the correct order will be much more useful for this use case. This is especially true with huge trees (e.g. links on a web page that you need to crawl and index).

Stack-based *iterative* approaches using a similar strategy to the recursion are *possible* too, however, they are not a good approach because this is supposed to be a breadth-first search problem (which uses queues) and they don't have the elegance of the recursive approach. If you used a stack for this question in an interview, you might leave your interviewer wondering why you felt the need to make the problem more difficult than it needed to be! For this reason, I've chosen not to include them in this article.

Complexity Analysis

- Time complexity : $O(n)$, where n is the number of nodes.
- Space complexity : $O(\log n)$ average case and $$O(n) worst case. The space used is on the runtime stack.

Rate this article:

Previous Next

Comments: 3 Sort By

Type comment here... (Markdown is supported)

Preview Post

itsthemakattack ★ 75 February 22, 2020 5:41 AM

Similar to solutions #1 and #2 but slightly simpler to read:

```
class Solution:
    def levelOrder(self, root: 'Node') -> List[List[int]]:
        q = collections.deque()
```

Read More

3 Share Reply

asperas ★ 183 March 30, 2020 8:56 PM

This question is medium and the same question with postorder traversal is labeled as easy.. Huh.. Really?

2 Share Reply

shikzheng ★ 42 November 10, 2019 7:38 AM

nice.

1 Share Reply