Previous Next

103. Binary Tree Zigzag Level Order Traversal

Dec. 26, 2019 | 41.7K views

Average Rating: 4.38 (16 votes)

Given a binary tree, return the *zigzag level order* traversal of its nodes' values. (ie, from left to right, then right to left for the next level and alternate between).

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

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

return its zigzag level order traversal as:

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

Solution

Approach 1: BFS (Breadth-First Search)

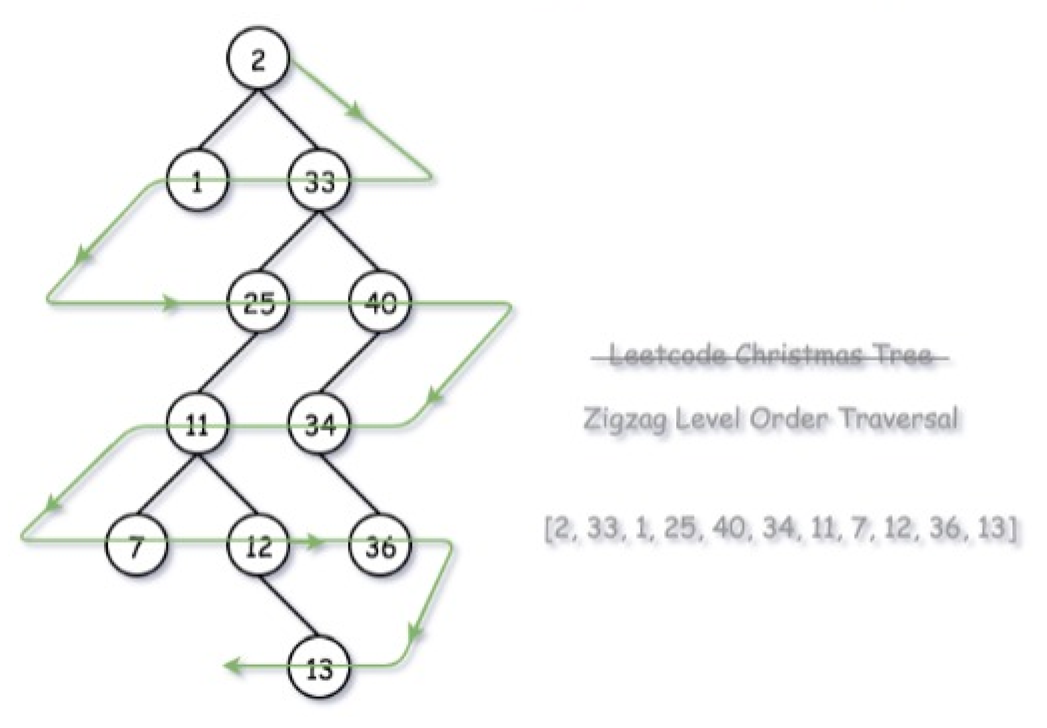
Intuition

Following the description of the problem, the most intuitive solution would be the *BFS* (Breadth-First Search) approach through which we traverse the tree level-by-level.

The default ordering of BFS within a single level is from left to right. As a result, we should adjust the BFS algorithm a bit to generate the desired zigzag ordering.

> One of the keys here is to store the values that are of the same level with the `deque` (double-ended queue) data structure, where we could add new values on either end of a queue.

So if we want to have the ordering of **FIFO** (first-in-first-out), we simply append the new elements to the *tail* of the queue, *i.e.* the late comers stand last in the queue. While if we want to have the ordering of **FILO** (first-in-last-out), we insert the new elements to the *head* of the queue, *i.e.* the late comers jump the queue.

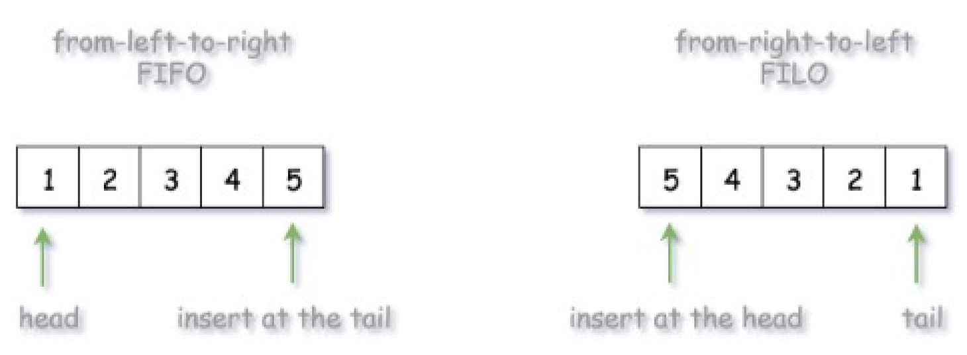


Algorithm

There are several ways to implement the BFS algorithm.

- One way would be that we run a two-level nested loop, with the *outer loop* iterating each level on the tree, and with the *inner loop* iterating each node within a single level.
- We could also implement BFS with a single loop though. The trick is that we append the nodes to be visited into a queue and we separate nodes of different levels with a sort of **delimiter** (*e.g.* an empty node). The delimiter marks the end of a level, as well as the beginning of a new level.

Here we adopt the *second* approach above. One can start with the normal BFS algorithm, upon which we add a touch of *zigzag* order with the help of `deque`. For each level, we start from an empty deque container to hold all the values of the same level. Depending on the ordering of each level, *i.e.* either from-left-to-right or from-right-to-left, we decide at which end of the deque to add the new element:



- For the ordering of from-left-to-right (FIFO), we *append* the new element to the ***tail*** of the queue, so that the element that comes late would get out late as well. As we can see from the above graph, given an input sequence of `[1, 2, 3, 4, 5]`, with FIFO ordering, we would have an output sequence of `[1, 2, 3, 4, 5]`.
- For the ordering of from-right-to-left (FILO), we *insert* the new element to the ***head*** of the queue, so that the element that comes late would get out first. With the same input sequence of `[1, 2, 3, 4, 5]`, with FILO ordering, we would obtain an output sequence of `[5, 4, 3, 2, 1]`.

Java | Python | Copy

```python
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None
from collections import deque

class Solution:
    def zigzagLevelOrder(self, root):
        """
        :type root: TreeNode
        :rtype: List[List[int]]
        """
        ret = []
        level_list = deque()
        if root is None:
            return []
        # start with the level 0 with a delimiter
        node_queue = deque([root, None])
        is_order_left = True

        while len(node_queue) > 0:
            curr_node = node_queue.popleft()

            if curr_node:
                if is_order_left:
```

Note: as an alternative approach, one can also implement the normal BFS algorithm first, which would generate the ordering of from-left-to-right for each of the levels. Then, at the end of the algorithm, we can simply ***reverse*** the ordering of certain levels, following the zigzag steps.

Complexity Analysis

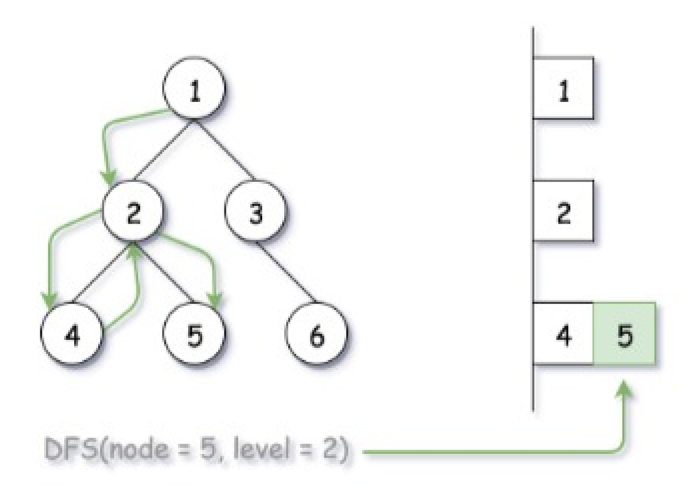
- Time Complexity: $\mathcal{O}(N)$, where N is the number of nodes in the tree.
  - We visit each node once and only once.
  - In addition, the insertion operation on either end of the deque takes a constant time, rather than using the array/list data structure where the inserting at the head could take the $\mathcal{O}(K)$ time where K is the length of the list.
- Space Complexity: $\mathcal{O}(N)$ where N is the number of nodes in the tree.
  - The main memory consumption of the algorithm is the `node_queue` that we use for the loop, apart from the array that we use to keep the final output.
  - As one can see, at any given moment, the `node_queue` would hold the nodes that are *at most* across two levels. Therefore, at most, the size of the queue would be no more than $2 \cdot L$, assuming L is the maximum number of nodes that might reside on the same level. Since we have a binary tree, the level that contains the most nodes could occur to consist all the leave nodes in a full binary tree, which is roughly $L = \frac{N}{2}$. As a result, we have the space complexity of $2 \cdot \frac{N}{2} = N$ in the worst case.

Approach 2: DFS (Depth-First Search)

Intuition

Though not intuitive, we could also obtain the *BFS* traversal ordering via the *DFS* (Depth-First Search) traversal in the tree.

> The trick is that during the DFS traversal, we maintain the results in a *global* array that is indexed by the level, *i.e.* the element `array[level]` would contain all the nodes that are at the same level. The global array would then be referred and updated at each step of DFS.



Similar with the above modified BFS algorithm, we employ the `deque` data structure to hold the nodes that are of the same level, and we alternate the insertion direction (*i.e.* either to the head or to the tail) to generate the desired output ordering.

Algorithm

Here we implement the DFS algorithm via *recursion*. We define a recursive function called `DFS(node, level)` which only takes care of the current `node` which is located at the specified `level`. Within the function, here are three steps that we would perform:

- If this is the first time that we visit any node at the `level`, *i.e.* the deque for the level does not exist, then we simply create the deque with the current node value as the initial element.
- If the deque for this level exists, then depending on the ordering, we insert the current node value either to the head or to the tail of the queue.
- At the end, we *recursively* call the function for each of its child nodes.

Java | Python | Copy

```python
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None
from collections import deque

class Solution:
    def zigzagLevelOrder(self, root):
        """
        :type root: TreeNode
        :rtype: List[List[int]]
        """
        if root is None:
            return []

        results = []
        def dfs(node, level):
            if level >= len(results):
                results.append(deque([node.val]))
            else:
                if level % 2 == 0:
                    results[level].append(node.val)
                else:
                    results[level].appendleft(node.val)

```

It might go without saying that, one can also implement the DFS traversal via ***iteration*** rather than recursion, which could be one of the followup questions by an interviewer.

Complexity Analysis

- Time Complexity: $\mathcal{O}(N)$, where N is the number of nodes in the tree.
  - Same as the previous BFS approach, we visit each node once and only once.
- Space Complexity: $\mathcal{O}(H)$, where H is the height of the tree, *i.e.* the number of levels in the tree, which would be roughly $\log_2 N$.
  - Unlike the BFS approach, in the DFS approach, we do not need to maintain the `node_queue` data structure for the traversal.
  - However, the function recursion would incur additional memory consumption on the *function call stack*. As we can see, the size of the call stack for any invocation of `DFS(node, level)` would be exactly the number of `level` that the current node resides on. Therefore, the space complexity of our DFS algorithm is $\mathcal{O}(\log_2 N)$ which is much better than the BFS approach.

Rate this article:

Previous Next

Comments: 15 Sort By

Type comment here... (Markdown is supported)
Preview Post

sheldon123z ★ 43 December 29, 2019 12:00 AM
I think it is FIFO and "LIFO" rather than "FILO"
25 Share Reply
SHOW 1 REPLY

doctordanger ★ 26 January 14, 2020 1:01 AM
simple BFS/Level order traversal

```
class Solution {
    public List<List<Integer>> zigzagLevelOrder(TreeNode root) {
        List<List<Integer>> list = new LinkedList<>();
```

Read More
17 Share Reply
SHOW 2 REPLIES

conanlemony1 ★ 6 February 16, 2020 1:03 PM
would Approach 2 have O(N) space complexity to store result?
3 Share Reply
SHOW 1 REPLY

sampleaccountpage ★ 3 January 27, 2020 6:16 AM
The DFS (extra) space complexity would be O(log_2(N)) only if the tree is relatively balanced. If it is completely unbalanced like a stick, the call stack would be of size n. Yet the worst extra space complexity for the BFS method is when there are 2 levels and could easily be better than O(n).
3 Share Reply

robinali34 ★ 5 April 17, 2020 4:07 AM
Level traversal:

```
class Solution {
    public List<List<Integer>> zigzagLevelOrder(TreeNode root) {
        List<List<Integer>> rtn = new ArrayList<>();
```

Read More
2 Share Reply
SHOW 2 REPLIES

moksha038 ★ 0 January 9, 2020 6:07 AM
Similar idea as first one, but straight forward.

```
from collections import deque
class Solution:
    def zigzagLevelOrder(self, root: TreeNode) -> List[List[int]]:
```

Read More
0 Share Reply

eaiman ★ 1 2 days ago
Time: O(n)
Space: O(n)

the code:

Read More
0 Share Reply

shazo ★ 0 July 6, 2020 1:46 AM
Space complexity for the second approach seems wrong to me.

How can the space complexity be any less than O(N)? The return is a list of list - where the inner elements are ALL the nodes of the tree (which is N).

Read More
0 Share Reply

kzhang2014 ★ 1 June 5, 2020 10:46 AM
In method # 2, shouldn't time complexity be higher than O(N)? Even though each node is only visited once, for every odd level, results.get(level).add(0, node.val) is called. And insert at list head is not o(1) because every element in list need to be shifted to the right. Can some explain for me?
0 Share Reply
SHOW 1 REPLY

sea0920 ★ 176 May 31, 2020 3:03 AM
Would any interviewer question that this solution isn't actual "Zigzag Level Order Traversal"? We are just changing the way outputs are inserted.
0 Share Reply

‹ 1 2 ›