314. Binary Tree Vertical Order Traversal

April 24, 2020 | 2.2K views

Average Rating: 5 (4 votes)

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]
]
```

Solution

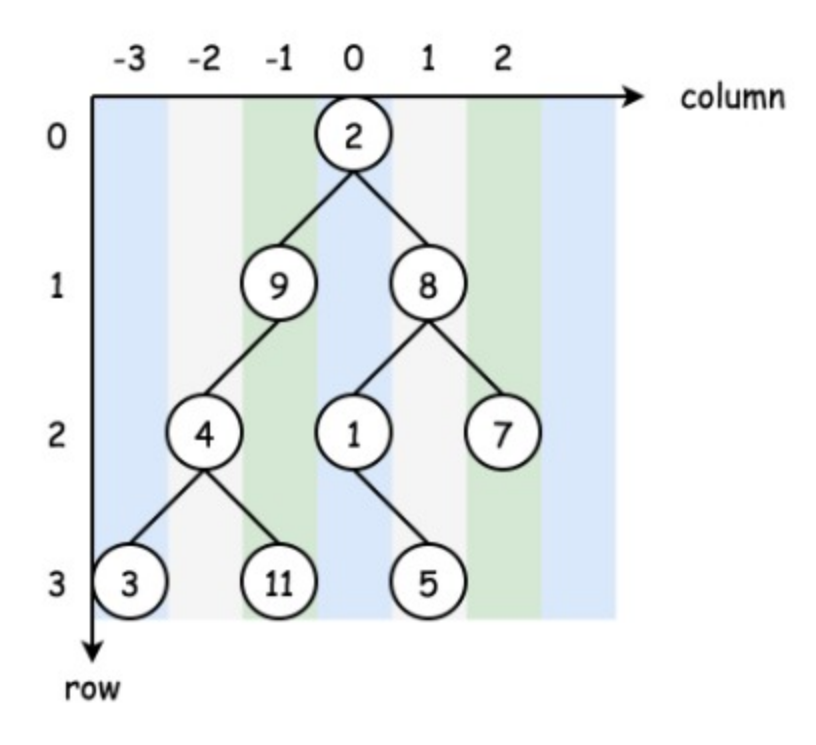
Overview

This is yet another problem about Binary Tree traversals. As one would probably know, the common strategies to traverse a *Tree* data structure are ***Breadth-First Search*** (*a.k.a* BFS) and ***Depth-First Search*** (*a.k.a.* DFS).

The DFS strategy can be further distinguished as *preorder DFS*, *inorder DFS* and *postorder DFS*, depending on the relative order of visit among the node itself and its child nodes.

If one is not familiar with the concepts of BFS and DFS, one can find the corresponding problems on LeetCode to practice with. Also, we have an Explore card called Queue & Stack where we cover both the BFS traversal as well as the DFS traversal. Hence, in this article, we won't repeat ourselves on these concepts.

In the problem description, we are asked to return the ***vertical*** order of a binary tree, which actually implies two sub-orders, where each node would have a 2-dimensional index (denoted as `<column, row>`)



- **column-wise order**

  If we look at a binary tree *horizontally*, each node can be aligned to a specific `column`, based on its relative *offset* to the root node of the tree.

  Let us assume that the root node has a column index of `0`, then its left child node would have a column index of `-1` and its right child node would have a column index of `+1`, and so on.

- **row-wise order**
  Now if we put the nodes into a *vertical* dimension, each node would be assigned to a specific `row`, based on its *level* (*i.e.* the vertical distance to the root node).

  Let us assume that the root node has a row index of `0`, then both its child nodes would have the row index of `1`.

> Given the above definitions, we can now formulate the problem as a task to order the nodes based on the 2-dimensional coordinates that we defined above.

More specifically, the nodes should be ordered by `column` first, and further the nodes on the same column should be ordered ***vertically*** based on their `row` indices.

Approach 1: Breadth-First Search (BFS)

Intuition

With the formulation of the problem in the overview section, one of the most intuitive solutions to tackle the problem would be applying the BFS traversal, where the nodes would be visited *level by level*.

With the BFS traversal, we naturally can guarantee the vertical order of the visits, *i.e.* the nodes at *higher* levels (large `row` values) would get visited later than the ones at lower levels.

However, we are still missing the horizontal order (the `column` order). To ensure this order, we need to do some additional processing during the BFS traversal.

> The idea is that we keep a hash table (let's denote it as `columnTable<key, value>`), where we keep the node values grouped by the `column` index.

The `key` in the hash table would be the `column` index, and the corresponding `value` would be a list which contains the values of all the nodes that share the same column index.

In addition, the values in the corresponding list should be ordered by their `row` indices, which would be guaranteed by the BFS traversal as we mentioned before.

Algorithm

We elaborate on the steps to implement the above idea.

- First, we create a hash table named `columnTable` to keep track of the results.
- As to the BFS traversal, a common code pattern would be to use a `queue` data structure to keep track of the order we need to visit nodes. We initialize the queue by putting the root node along with its column index value (`0`).
- We then run the BFS traversal with a loop consuming the elements from the queue.
- At each iteration within the BFS, we pop out an element from the queue. The element consists of a `node` and its corresponding `column` index. If the node is not empty, we then populate the `columnTable` with the value of the node. Subsequently, we then put its child nodes along with their respective column indices (*i.e.* `column-1` and `column+1`) into the queue.
- At the end of the BFS traversal, we obtain a hash table that contains the desired node values grouped by their `column` indices. For each group of values, they are further ordered by their `row` indices.
- We then sort the hash table by its keys, *i.e.* `column` index in ascending order. And finally we return the results *column by column*.

Java | Python3 | 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 defaultdict
class Solution:
    def verticalOrder(self, root: TreeNode) -> List[List[int]]:
        columnTable = defaultdict(list)
        queue = deque([(root, 0)])

        while queue:
            node, column = queue.popleft()

            if node is not None:
                columnTable[column].append(node.val)

                queue.append((node.left, column - 1))
                queue.append((node.right, column + 1))

        return [columnTable[x] for x in sorted(columnTable.keys())]
```

Complexity Analysis

- Time Complexity: $\mathcal{O}(N \log N)$ where N is the number of nodes in the tree.

  In the first part of the algorithm, we do the BFS traversal, whose time complexity is $\mathcal{O}(N)$ since we traversed each node once and only once.

  In the second part, in order to return the ordered results, we then sort the obtained hash table by its keys, which could result in the $\mathcal{O}(N \log N)$ time complexity in the worst case scenario where the binary tree is extremely imbalanced (for instance, each node has only left child node.)

  As a result, the overall time complexity of the algorithm would be $\mathcal{O}(N \log N)$.

- Space Complexity: $\mathcal{O}(N)$ where N is the number of nodes in the tree.

  First of all, we use a hash table to group the nodes with the same column index. The hash table consists of keys and values. In any case, the values would consume $\mathcal{O}(N)$ memory. While the space for the keys could vary, in the worst case, each node has a unique column index, *i.e.* there would be as many keys as the values. Hence, the total space complexity for the hash table would still be $\mathcal{O}(N)$.

  During the BFS traversal, we use a `queue` data structure to keep track of the next nodes to visit. At any given moment, the queue would hold no more two levels of nodes. For a binary tree, the maximum number of nodes at a level would be $\frac{N+1}{2}$ which is also the number of leafs in a full binary tree. As a result, in the worst case, our queue would consume at most $\mathcal{O}(\frac{N+1}{2} \cdot 2) = \mathcal{O}(N)$ space.

  Lastly, we also need some space to hold the results, which is basically a reordered hash table of size $\mathcal{O}(N)$ as we discussed before.

  To sum up, the overall space complexity of our algorithm would be $\mathcal{O}(N)$.

Approach 2: BFS without Sorting

Intuition

In the previous approach, it is a pity that the sorting of results overshadows the main part of the algorithm which is the BFS traversal. One might wonder if we have a way to eliminate the need for sorting. And the answer is yes.

> The key insight is that we only need to know the **range** of the column index (*i.e.* `[min_column, max_column]`). Then we can simply ***iterate*** through this range to generate the outputs without the need for sorting.

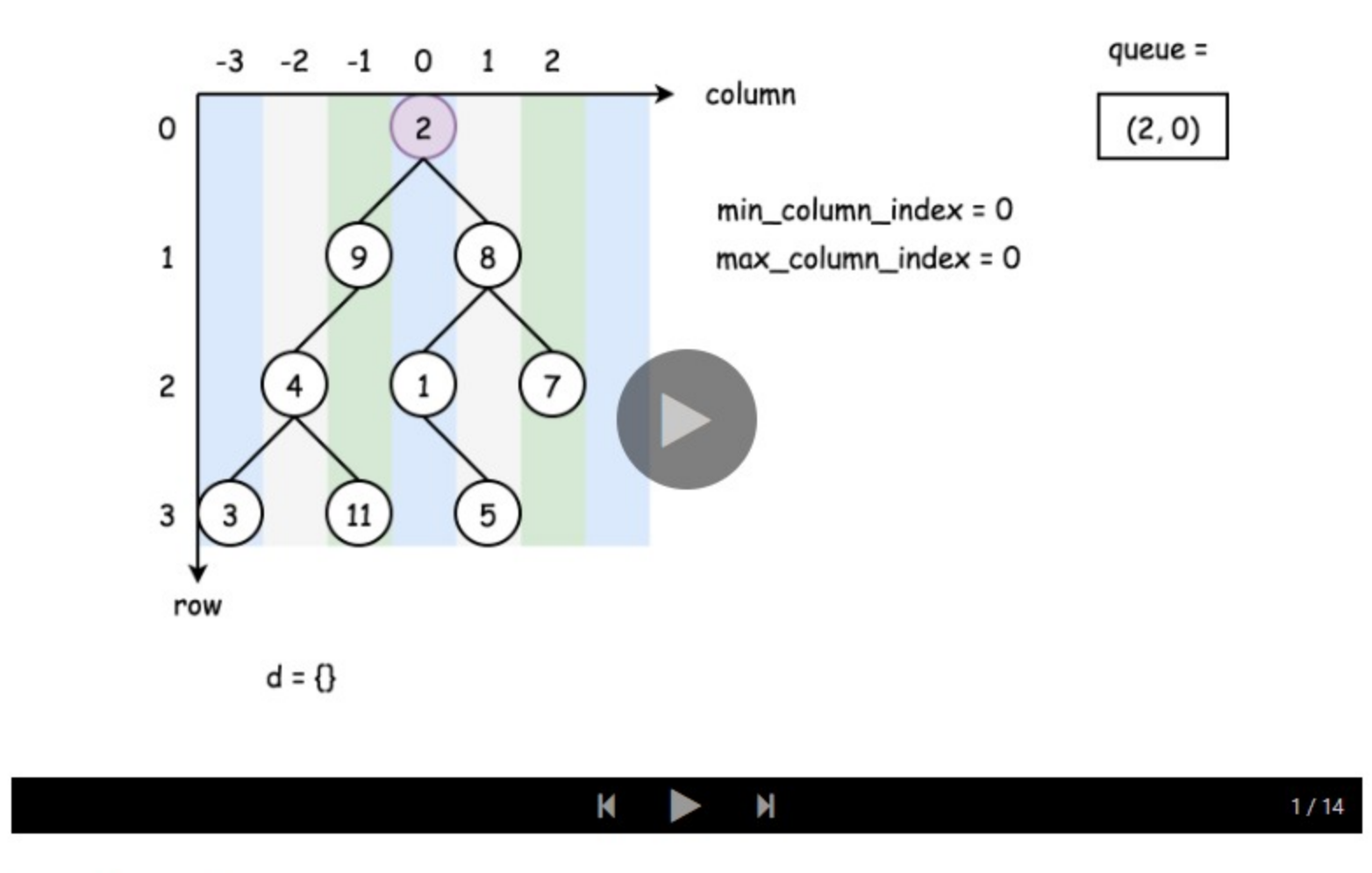
The above insight would work under the *condition* that there won't be any missing column index in the given range. And the condition always holds, since there won't be any broken branch in a binary tree.

Algorithm

To implement this optimization, it suffices to make some small modifications to our previous BFS approach.

During the BFS traversal, we could obtain the range of the column indices, *i.e.* with the variable of `min_column` and `max_column`.

At the end of the BFS traversal, we would then walk through the column range `[min_column, max_column]` and retrieve the results accordingly.



Java | Python3 | 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 defaultdict
class Solution:
    def verticalOrder(self, root: TreeNode) -> List[List[int]]:
        if root is None:
            return []

        columnTable = defaultdict(list)
        min_column = max_column = 0
        queue = deque([(root, 0)])

        while queue:
            node, column = queue.popleft()

            if node is not None:
                columnTable[column].append(node.val)
                min_column = min(min_column, column)
                max_column = max(max_column, column)

                queue.append((node.left, column - 1))
                queue.append((node.right, column + 1))

```

Complexity Analysis

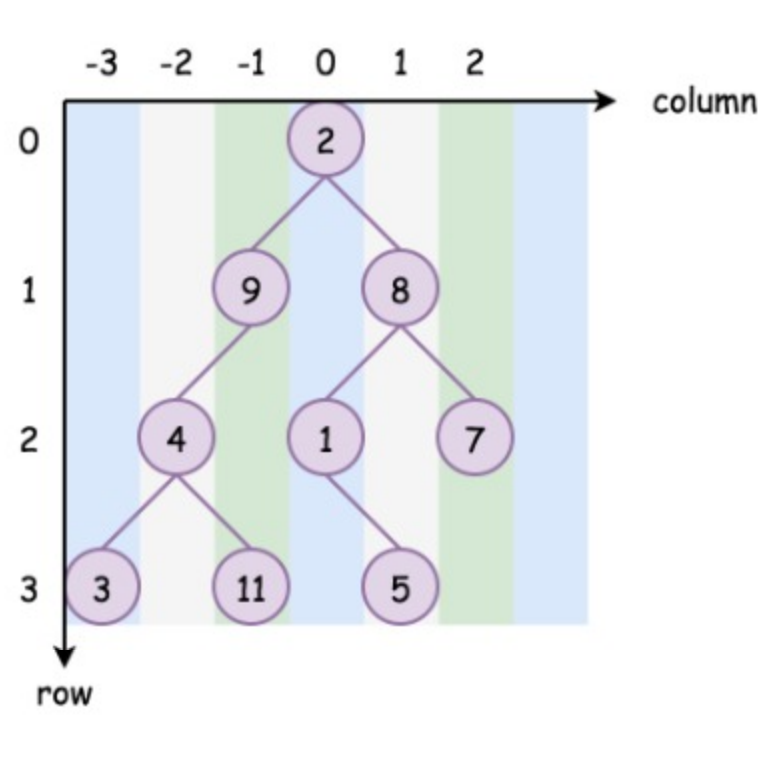
- Time Complexity: $\mathcal{O}(N)$ where N is the number of nodes in the tree.
  Following the same analysis in the previous BFS approach, the only difference is that this time we don't need the costy sorting operation (*i.e.* $\mathcal{O}(N \log N)$).
- Space Complexity: $\mathcal{O}(N)$ where N is the number of nodes in the tree. The analysis follows the same logic as in the previous BFS approach.

Approach 3: Depth-First Search (DFS)

Intuition

Although we applied a BFS traversal in both of the previous approaches, it is not impossible to solve the problem with a DFS traversal.

> As we discussed in the overview section, once we assign a 2-dimensional index (*i.e.* `<column, row>`) for each node in the binary tree, to output the tree in ***vertical*** order is to sort the nodes based on the 2-dimensional index, firstly by `column` then by `row`, as shown in the following graph.



column, row	value
(-3, 3)	3
(-2, 2)	4
(-1, 1)	9
(-1, 3)	11
(0, 0)	2
(0, 2)	1
(1, 1)	8
(1, 3)	5
(2, 2)	7

Compared to the DFS traversal, the BFS traversal gives us a head start, since the nodes in higher rows would be visited later than the ones in the lower lows. As a result, we only need to focus on the `column` order.

That being said, we could simply traverse the tree in any DFS order (preorder, inorder or postorder), then we sort the resulting list strictly based on two keys `<column, row>`, which would give us the same results as the BFS traversal.

> An important note is that two nodes might share the same `<column, row>`, in the case, as stated in the problem, the order between these two nodes should be from **left** to **right** as we did for BFS traversals. As a result, to ensure such a priority, one should make sure to visit the left child node before the right child node during the DFS traversal.

Algorithm

- Here we implement the above algorithm, with the trick that we applied in Approach 2 (BFS without sorting) where we obtained the range of `column` during the traversal.
- First, we conduct a DFS traversal on the input tree. During the traversal, we would then build a similar `columnTable` with the `column` index as the key and the list of `(row, val)` tuples as the value.
- At the end of the DFS traversal, we iterate through the `columnTable` via the key of `column` index. Accordingly, we have a list of `(row, val)` tuples associated with each key. We then sort this list, based on the `row` index.
- After the above steps, we would then obtain a list of node values ordered firstly by its `column` index and then by its `row` index, which is exactly the the *vertical* order traversal of binary tree as defined in the problem.

Java | Python3 | 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 defaultdict
class Solution:
    def verticalOrder(self, root: TreeNode) -> List[List[int]]:
        if root is None:
            return []

        columnTable = defaultdict(list)
        min_column = max_column = 0

        def DFS(node, row, column):
            if node is not None:
                nonlocal min_column, max_column
                columnTable[column].append((row, node.val))
                min_column = min(min_column, column)
                max_column = max(max_column, column)

                # preorder DFS
                DFS(node.left, row + 1, column - 1)
                DFS(node.right, row + 1, column + 1)

        DFS(root, 0, 0)
```

Complexity Analysis

- Time Complexity: $\mathcal{O}(W \cdot H \log H))$ where W is the width of the binary tree (*i.e.* the number of columns in the result) and H is the height of the tree.
  In the first part of the algorithm, we traverse the tree in DFS, which results in $\mathcal{O}(N)$ time complexity.

  Once we build the `columnTable`, we then have to sort it *column by column*.

  Let us assume the time complexity of the sorting algorithm to be $\mathcal{O}(K \log K)$ where K is the length of the input. The maximal number of nodes in a column would be $\frac{H}{2}$ where H is the height of the tree, due to the zigzag nature of the node distribution. As a result, the upper bound of time complexity to sort a column in a binary tree would be $\mathcal{O}(\frac{H}{2} \log \frac{H}{2})$.

  Since we need to sort W columns, the total time complexity of the sorting operation would then be $\mathcal{O}(W \cdot (\frac{H}{2} \log \frac{H}{2})) = \mathcal{O}(W \cdot H \log H)$. Note that, the total number of nodes N in a tree is bounded by $W \cdot H$, *i.e.* $N < W \cdot H$. As a result, the time complexity of $\mathcal{O}(W \cdot H \log H)$ will dominate the $\mathcal{O}(N)$ of the DFS traversal in the first part.

  At the end of the DFS traversal, we have to iterate through the `columnTable` in order to retrieve the values, which will take another $\mathcal{O}(N)$ time.

  To sum up, the overall time complexity of the algorithm would be $\mathcal{O}(W \cdot H \log H)$.

  An interesting thing to note is that in the case where the binary tree is completely imbalanced (*e.g.* node has only left child.), this DFS approach would have the $\mathcal{O}(N)$ time complexity, since the sorting takes no time on columns that contains only a single node. While the time complexity for our first BFS approach would be $\mathcal{O}(N \log N)$, since we have to sort the N keys in the `columnTable`.

- Space Complexity: $\mathcal{O}(N)$ where N is the number of nodes in the tree.

  We kept the `columnTable` which contains all the node values in the binary tree. Together with the keys, it would consume $\mathcal{O}(N)$ space as we discussed in previous approaches.

  Since we apply the recursion for our DFS traversal, it would incur additional space consumption on the function call stack. In the worst case where the tree is completely imbalanced, we would have the size of call stack up to $\mathcal{O}(N)$.

  Finally, we have the output which contains all the values in the binary tree, thus $\mathcal{O}(N)$ space.

  So in total, the overall space complexity of this algorithm remains $\mathcal{O}(N)$.

Analysis written by @liaison and @andvary

Rate this article:

Comments: 2 Sort By

nd2n8 34 May 1, 2020 12:02 PM

This problem should be classified as hard. Thanks for the detailed solution.

3 Share Reply

8080dk 33 April 24, 2020 10:52 PM

Related topics should be 'BFS' or 'DFS' instead of hashtable. Hashtable is actually not required.

1 Share Reply

SHOW 1 REPLY