Solution: Vertical Order Traversal of a Binary Tree

Let's solve the Vertical Order Traversal of a Binary Tree problem using the Tree Breadth-first Search pattern.

We'll cover the following

- Statement
- Solution
  - Naive approach
  - Optimized approach using tree breadth-first search
    - Solution summary
    - Time complexity
    - Space complexity

Statement

Find the vertical order traversal of a binary tree when the root of the binary tree is given. In other words, return the values of the nodes from top to bottom in each column, column by column from left to right. If there is more than one node in the same column and row, return the values from left to right.

Constraints:

- The number of nodes in the tree is in the range `[1, 1000]`.
- $0 \leq$ `Node.val` ≤ 1000

Solution

So far, you've probably brainstormed some approaches and have an idea of how to solve this problem. Let's explore some of these approaches and figure out which one to follow based on considerations such as time complexity and any implementation constraints.

Naive approach

The naive approach would be traversing the tree to get the maximum and minimum horizontal distance of the nodes from the root. Once we have these numbers, we can traverse over the tree again for each distance in the range `[maximum, minimum]` and return the nodes respectively.

The time complexity of the above algorithm is $O(n^2)$, where n is the number of nodes. The space complexity is $O(n)$.

Optimized approach using tree breadth-first search

Since we're dealing with a binary tree, the approach that comes to mind is either depth-first search (DFS) or breadth-first search (BFS). While both seem viable options, the breadth-first search is a better approach since we need to assign column values to each node. With BFS, we can assign the $column - 1$ value to the left child and the $column + 1$ to the first child. Hence, this problem falls under the tree breadth-first search pattern.

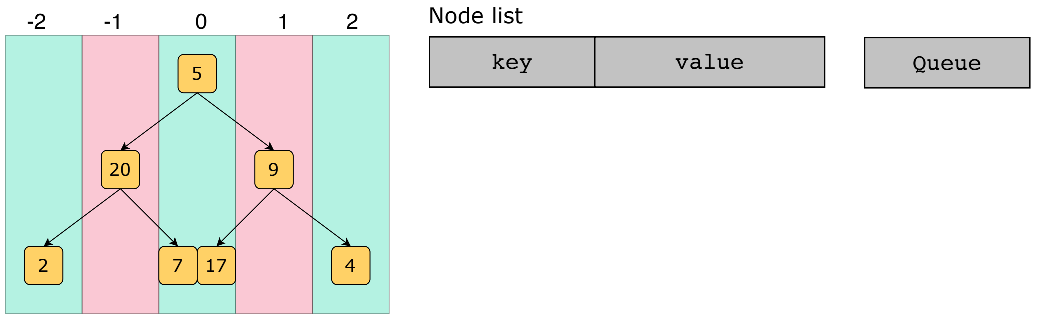
The intuition behind using BFS is that it guarantees that nodes at the top will be visited first and the nodes at the bottom will be visited last. To traverse the tree from left to right, we need to maintain a hash map that contains the list of all the nodes in a specific column. We'll denote these columns by their index, where the column of the root will be index `0`, the columns on the left side of the root will have a negative index, and the columns on the right side of the root will have a positive index. The index of each column will be a key.

We also need to store the value of the minimum and maximum indexes in two variables so that we can iterate over the indexes to get the nodes in the right order.

Let's go over the algorithm in detail below:

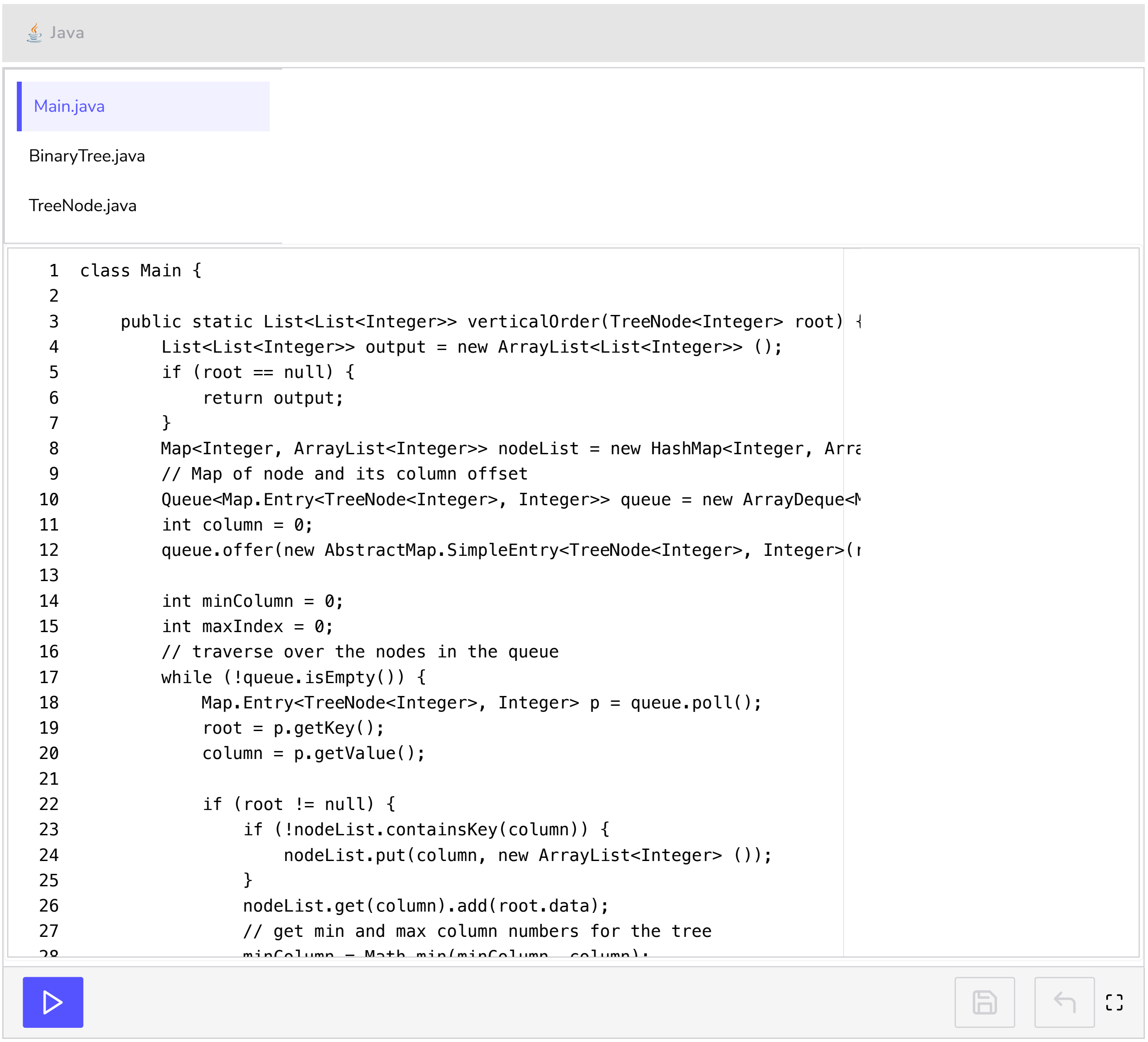
1. We need to create a hash map called `nodeList` in which the key will be the index of the column and the value will be the list of nodes in that column.
2. We initialize two variables to store the minimum and maximum values of the index.
3. During BFS, we also maintain a queue to keep track of the order of nodes that need to be visited. We'll initialize the queue by adding the root to it along with the column index of the root, which is `0`.
4. Next, we carry out BFS using a loop until the queue is empty. We iterate over the queue elements and pop them. The values in the queue consist of a node and the index of the column.
5. If the queue contains a node that is not NULL, we add the value of the node to `nodeList` along with the index of the column. We also keep updating the minimum and maximum values of the index. Then, we append the children of this node to the queue with their index values. The left child has an index value of the current index -1, and the right child has a value of the current index $+1$.
6. At the end of all the iterations, the `nodeList` hash map contains the values of all the nodes in all the columns, but the values of each column are not in the correct order.
7. We'll iterate over the index range using the variables that stored the minimum and maximum indexes. Finally, we return the values of the nodes in the correct order.

We'll traverse the tree level by level and store the nodes and their column numbers in a hash map. The root is assigned the column number = 0, the left child = parent's column - 1, and the right child = parent's column + 1.



1 of 10

Let's look at the code for this solution below:



```java
class Main {

    public static List<List<Integer>> verticalOrder(TreeNode<Integer> root) {
        List<List<Integer>> output = new ArrayList<List<Integer>> ();
        if (root == null) {
            return output;
        }
        Map<Integer, ArrayList<Integer>> nodeList = new HashMap<Integer, Arra
        // Map of node and its column offset
        Queue<Map.Entry<TreeNode<Integer>, Integer>> queue = new ArrayDeque<M
        int column = 0;
        queue.offer(new AbstractMap.SimpleEntry<TreeNode<Integer>, Integer>(r

        int minColumn = 0;
        int maxIndex = 0;
        // traverse over the nodes in the queue
        while (!queue.isEmpty()) {
            Map.Entry<TreeNode<Integer>, Integer> p = queue.poll();
            root = p.getKey();
            column = p.getValue();

            if (root != null) {
                if (!nodeList.containsKey(column)) {
                    nodeList.put(column, new ArrayList<Integer> ());
                }
                nodeList.get(column).add(root.data);
                // get min and max column numbers for the tree
                minColumn = Math.min(minColumn, column);
```

Solution summary

To recap, the solution to this problem can be divided into the following steps:

1. Traverse the tree level by level, starting from the root.
2. Maintain a hash map that contains the list of all the nodes in a specific column.
3. Store the value of the minimum and maximum indexes in two variables so we can iterate over the indexes to get the nodes in the right order.

Time complexity

We are using BFS, which goes through each node precisely once and has a time complexity of $O(n)$, where n is the number of nodes in the binary tree.

Space complexity

In this scenario, there are three data structures—a hash map, a queue, and an ordered hash map.

The first is a hash map that stores the nodes' values against the column indexes. In the worst-case scenario, every node will be in a different column. That means its space complexity will be $O(n)$, where n is the number of nodes in the binary tree.

At any time, the queue will contain a maximum of nodes of two levels, and the maximum a level can contain is $(n + 1)/2$. So, in the worst-case scenario, the space complexity will be $(n + 1)/2 + (n + 1)/4 = O(n)$.

We also have an ordered hash table that stores the result. It will have a space complexity of $O(n)$.

The overall space complexity will be $O(n)$.

← Back
Vertical Order Travers...

Next →
Connect All Siblings o...

✓ Mark as Completed