Solution: Level Order Traversal of Binary Tree

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

We'll cover the following

Statement

Given the root of a binary tree, display the values of its nodes while performing a level order traversal. Return the node values for all levels in a string separated by the character `:`. If the tree is empty, i.e., the number of nodes is 0, then return "None" as the output.

Constraints:

- The number of nodes in the tree is in the range $[0, 2000]$.
- $-1000 \leq$ `Node.data` ≤ 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 to first calculate the height of the tree and then recursively traverse it, level by level. The printing function would be called for every level in the range $[1, height]$.

In the case of a skewed tree, the time complexity for this approach would be $O(n^2)$, where n is the number of nodes. The space complexity is $O(n)$.

Optimized approach using tree breadth-first search

We can traverse a binary tree using the depth or breadth-first search pattern. The fundamental characteristic of the depth-first search pattern is that it travels as far as possible along each tree branch before exploring the others.

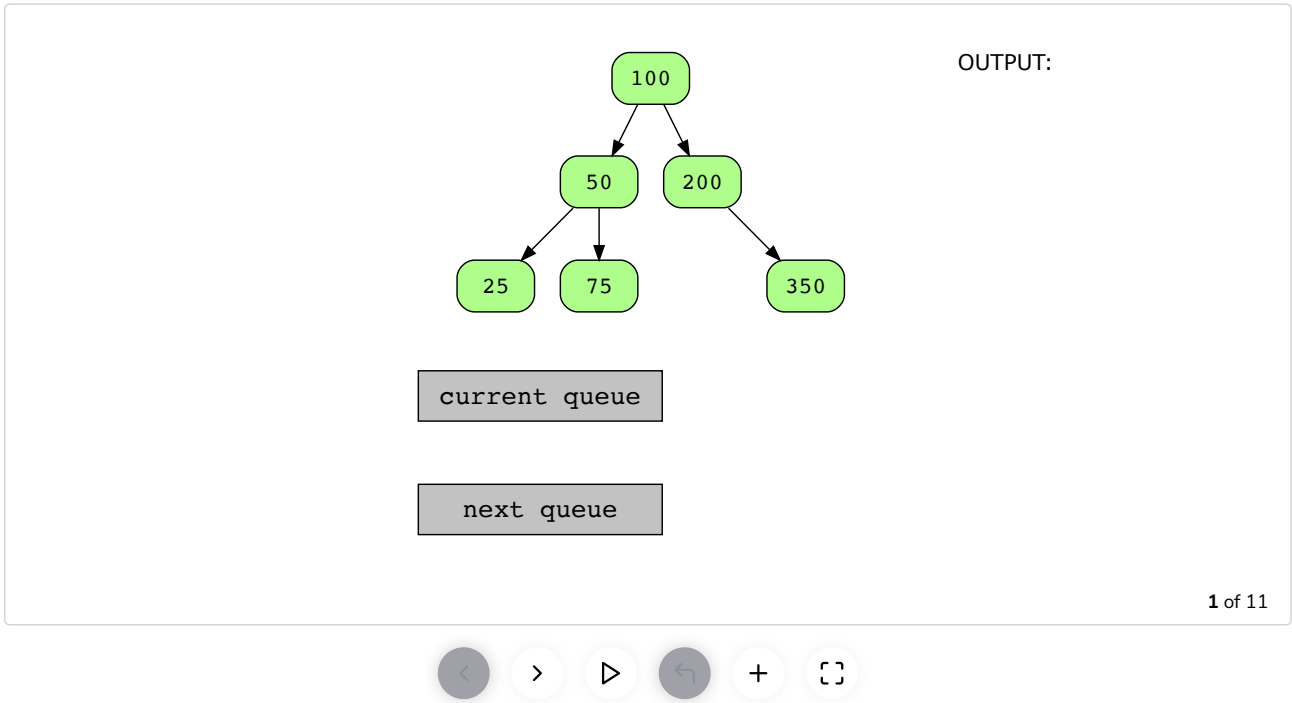
However, since we are asked to print all nodes at the same hierarchical level, this problem naturally falls under the tree breadth-first search pattern. It starts searching at the root node and moves down level by level, exploring adjacent nodes at level $k + 1$. We'll print all the nodes at each level and then move to the next level.

Solution 1

We can use two queues to solve this problem:

- A current queue
- A next queue

The slides below illustrate how the algorithm runs:



> **Note**: In the following section, we will gradually build the solution. Alternatively, you can skip straight to just the code.

Step-by-step solution construction

- First, we'll declare an array of two queues. This step will help us swap values between our two queues later.
- We then declare and point `currentQueue` to the 0^{th} index and `nextQueue` to the 1^{st} index of the `queues` array.
- We'll start by pushing the root node to `currentQueue` and setting the tree level to zero.

Java

main.java

BinaryTree.java

TreeNode.java

```
class LevelOrderTraversal {

    public static void printQueue(Queue<TreeNode<Integer>> copy){
        TreeNode<Integer> node=null;
        Iterator<TreeNode<Integer>> through = copy.iterator() ;
        while(through.hasNext() ) {
            node = through.next();
            System.out.print("["+ node.data+ "]") ;
        }
        System.out.println() ;

    }

    public static void levelOrderTraversal(TreeNode<Integer> root) {
        // We print null if the root is null
        if (root == null) {
            System.out.print("null");
        } else {
            System.out.println("\n\tInitializing the queues");
            // Declaring an array of two queues
            ArrayList<Queue<TreeNode<Integer>>> queues = new ArrayList<Queue<
            queues.add(new ArrayDeque<TreeNode<Integer>>());
            queues.add(new ArrayDeque<TreeNode<Integer>>());
            System.out.println("\t\tqueues array: " + queues);
            // Initializing the current and next queues
            Queue<TreeNode<Integer>> currentQueue = queues.get(0);
            Queue<TreeNode<Integer>> nextQueue = queues.get(1);
```

Pushing the root node

We'll carry out the following operations until `currentQueue` is empty:

- Dequeue the first element of `currentQueue`.
- Enqueue its children to `nextQueue`.
- Store the dequeued node in `result`.

Java

main.java

BinaryTree.java

TreeNode.java

```
class LevelOrderTraversal {

    public static String printQueue(Queue<TreeNode<Integer>> copy) {
        if (copy != null && !copy.isEmpty()) {
            StringBuilder output = new StringBuilder("[");
            for (TreeNode<Integer> through : copy) {
                output.append(through.data).append(", ");
            }
            output.delete(output.length() - 2, output.length());
            output.append("]");
            return output.toString();
        }
        return "[]";
    }

    public static String levelOrderTraversal(TreeNode<Integer> root) {
        StringBuilder result = new StringBuilder();
        // Print None if the root doesn't exist
```

```
        if (root == null) {
            System.out.print("None");
        } else {
            System.out.println("\n\tInitializing the queues");
            // Declaring an array of two queues
            ArrayList<Queue<TreeNode<Integer>>> queues = new ArrayList<Queue<
            queues.add(new ArrayDeque<TreeNode<Integer>>());
            queues.add(new ArrayDeque<TreeNode<Integer>>());
            System.out.println("\t\tqueues array: " + queues);
            // Initializing the current and next queues
```

Dequeue operation

- The loop in the code above runs once, since we only have one node in `currentQueue`. To ensure all the tree nodes are visited, we'll swap `currentQueue` and `nextQueue` once the former becomes empty.
- We'll swap `currentQueue` with `nextQueue` and increment `levelNumber` by 1. Since we're moving to the next level, we'll also append `:` to `result`.
- We terminate the loop if `currentQueue` is still empty and return the result.

Java

main.java

BinaryTree.java

TreeNode.java

```
class LevelOrderTraversal {
    public static String printQueue(Queue<TreeNode<Integer>> copy) {
        if (copy != null && !copy.isEmpty()) {
            StringBuilder output = new StringBuilder("[");
            for (TreeNode<Integer> through : copy) {
                output.append(through.data).append(", ");
            }
            output.delete(output.length() - 2, output.length());
            output.append("]");
            return output.toString();
        }
        return "[]";
    }

    public static String levelOrderTraversal(TreeNode<Integer> root) {
        // We print null if the root is null
        StringBuilder result = new StringBuilder();
        if (root == null) {
            System.out.println("None");
            return "None";
        } else {
            System.out.println("\n\tInitializing the queues");
            // Declaring an array of two queues
            ArrayList<Queue<TreeNode<Integer>>> queues = new ArrayList<Queue<
            queues.add(new ArrayDeque<TreeNode<Integer>>());
            queues.add(new ArrayDeque<TreeNode<Integer>>());
            System.out.println("\t\tqueues array: " + queues);
            // Initializing the current and next queues
```

Level Order Traversal of Binary Tree

Just the code

Here's the complete solution to this problem:

Java

main.java

BinaryTree.java

TreeNode.java

```java
class LevelOrderTraversal {

    public static String levelOrderTraversal(TreeNode<Integer> root) {

        StringBuilder result = new StringBuilder();
        if (root == null) {
            return "None";
        } else {
            ArrayList<Queue<TreeNode<Integer>>> queues = new ArrayList<Queue<
            queues.add(new ArrayDeque<TreeNode<Integer>>());
            queues.add(new ArrayDeque<TreeNode<Integer>>());

            Queue<TreeNode<Integer>> currentQueue = queues.get(0);
            Queue<TreeNode<Integer>> nextQueue = queues.get(1);

            currentQueue.add(root);
            int levelNumber = 0;
            int n = 0;

            while (!currentQueue.isEmpty()) {

                n += 1;
                TreeNode<Integer> temp = currentQueue.poll();
                result.append(String.valueOf(temp.data));

                if (temp.left != null) {
                    nextQueue.add(temp.left);
                }
```

Level Order Traversal of Binary Tree

Solution summary

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

1. Use two queues, `currentQueue` and `nextQueue`, to keep track of the nodes.
2. Dequeue nodes from `currentQueue` and push their children to the `nextQueue`.
3. Store the dequeued nodes in `result`.
4. If the former queue is empty in step 3, swap the two queues and increment the level number.
5. Return `result` once the loop terminates.

Time complexity

The time complexity of this solution is linear, $O(n)$, where n is the number of nodes, because every node is visited and printed only once.

Space complexity

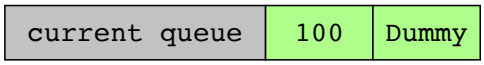
The space complexity of this solution is linear, $O(n)$, since the algorithm instantiates queues that take up space of up to $\lceil n/2 \rceil$ nodes. This is because the maximum queue size occurs at the level of the leaf nodes of a balanced binary tree.

Solution 2

In the previous solution, we used two queues to keep track of the current level. However, we'll only maintain one queue in this solution. An empty `currentQueue` indicates that we've processed all nodes at the current level. However, all we need is an end-of-level marker. This end-of-level marker can be an empty queue or a dummy node.

> **Note:** A dummy node acts as the end-of-level marker in the current queue.

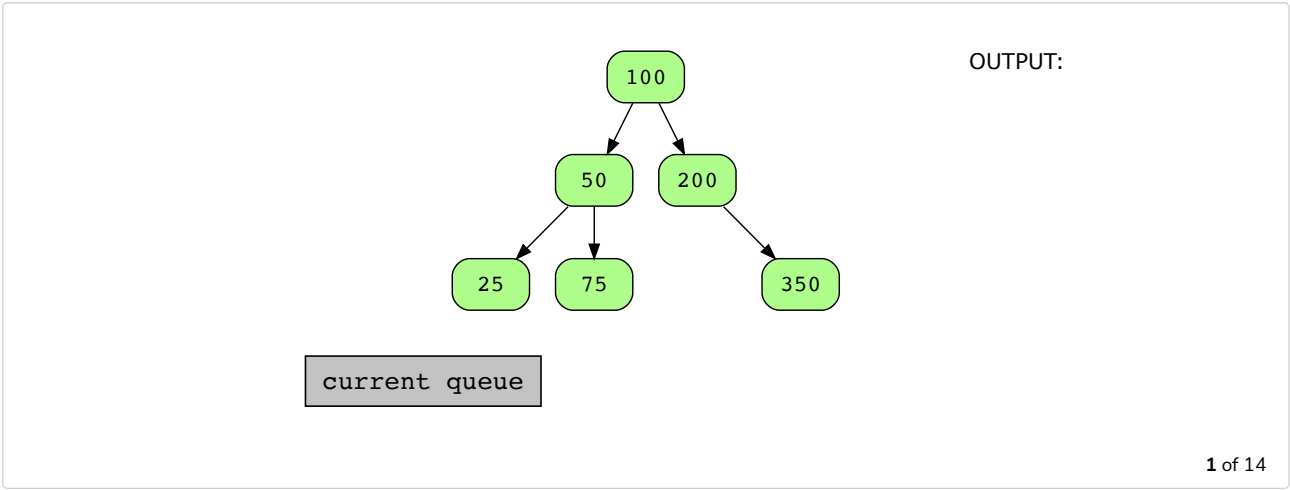
Here's what `currentQueue` initially looks like with the dummy marker at level zero:



The modified steps for this solution are as follows:

1. First, we declare `currentQueue` and a dummy node.
2. We then push the root and dummy nodes into `currentQueue`.
3. Next, we carry out the following operations until `currentQueue` is empty:
    - Dequeue the first element of the queue and print its data.
    - Enqueue the children of the node we dequeued in the previous step into `currentQueue`.
    - If the first element of `currentQueue` is the dummy node, we print a new line and remove the dummy node.
    - If `currentQueue` is not empty, add the dummy node back to the queue. Otherwise, we terminate the algorithm.

Here's an example to demonstrate the algorithm mentioned above:



Java

BinaryTree.java

TreeNode.java

```java
class LevelOrderTraversal {

    public static String levelOrderTraversal(TreeNode<Integer> root) {
        StringBuilder result = new StringBuilder();
        // Print None if the root doesn't exist
        if (root == null) {
            return "None";
        } else {
            // Initializing the current queue
            Queue<TreeNode<Integer>> current_queue = new ArrayDeque<TreeNode<

            // Initializing the dummy node
            TreeNode<Integer> dummyNode = new TreeNode<Integer>(0);

            // Enqueuing the root and dummy nodes into the current queue
            current_queue.add(root);
            current_queue.add(dummyNode);

            // Printing nodes in level-order until the current queue remains
            // empty
            while (!current_queue.isEmpty()) {
                // Dequeuing and printing the first element of queue
                TreeNode<Integer> temp = current_queue.poll();
                result.append(String.valueOf(temp.data));
                //System.out.print(temp.data);

                // Adding dequeued node's children to the next queue
                if (temp.left != null) {
```

Level Order Traversal of Binary Tree

Solution summary

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

1. Start by pushing the root to a queue.
2. Add a dummy node between each level of the tree.
3. Dequeue the first element of the queue and enqueue its children.

Time complexity

The time complexity of this solution is linear, $O(n)$, where n is the number of nodes, because every node is visited and printed only once.

Space complexity

The space complexity of this solution is linear, $O(n)$, since the algorithm instantiates queues that take up space of up to $\lceil n/2 \rceil$ nodes. This is because the maximum queue size occurs at the level of the leaf nodes of a balanced binary tree.

Back
Level Order Traversal ...

Next
Binary Tree Zigzag Le...

Mark as Completed