1609. Even Odd Tree

Medium | Tree | Breadth-First Search | Binary Tree | Leetcode Link

Problem Description

The problem provides a binary tree and introduces a special condition called *Even-Odd*. A binary tree meets the Even-Odd condition if:

- The root node is at level 0, its children are at level 1, their children at level 2, and so on.
- Nodes at levels with an even index (like 0, 2, 4, ...) must have odd integer values, and these values must be in strictly increasing order from left to right.
- Nodes at levels with an odd index (like 1, 3, 5, ...) must have even integer values, and these values must be in strictly decreasing order from left to right.

The goal is to write a function that takes the root node of a binary tree as an input and returns `true` if the tree satisfies the Even-Odd condition; otherwise, it should return `false`.

Intuition

To solve this problem, we use the Breadth-First Search (BFS) algorithm. BFS allows us to traverse the tree level by level. This fits our needs perfectly since the Even-Odd condition applies to individual levels of the tree.

Here's how we approach the solution:

1. Initialize a queue and add the root node to it. The queue is used to keep track of nodes to visit in the current level.
2. Traverse each level of the tree using a while loop that runs as long as there are nodes left in the queue:
   - For each node, check if the value meets the required criteria based on whether the current level is even or odd.
   - Maintain a `prev` value to compare with the current node's value ensuring the strictly increasing or decreasing order.
   - After processing all nodes in the current level, toggle the even/odd level flag using XOR with `1`.
3. If at any point a node does not meet the requirements, return `false`.
4. If the loop completes without finding any violations, return `true` since the tree meets the Even-Odd condition.

By using BFS, we are able to check the values level by level, and by maintaining a variable to track the previous node's value, we can easily check the strictly increasing or decreasing order requirement.

Solution Approach

The solution approach is based on a Breadth-First Search (BFS) pattern. Here's a walkthrough of the implementation details:

1. **Initialize Variables**: A flag variable `even`, which is initially set to `1`, is used to represent the current parity of the level being traversed (odd or even). A queue `q` is used to keep track of nodes in the current level.

   ```
   even = 1
   q = deque([root])
   ```

2. **BFS Loop**: The solution iterates through the tree by using a `while` loop that continues as long as there are nodes in the queue.

   ```
   while q:
   ```

3. **Level Traversal**: Within the loop, a `for` loop iterates over each node in the current level. The number of nodes is determined by the current size of the queue (`len(q)`).

   ```
   for _ in range(len(q)):
   ```

4. **Initialization Per Level**: At the beginning of each level, initialize `prev` to `0` if the current level is supposed to contain even values, and to `inf` for odd values.

   ```
   prev = 0 if even else inf
   ```

5. **Node Validation**: For each node, the algorithm checks if the current node's value meets the appropriate conditions:

   - For even levels, node values must be odd and increasing, checked by `root.val % 2 == 0 or prev >= root.val`.
   - For odd levels, node values must be even and decreasing, checked by `root.val % 2 == 1 or prev <= root.val`.

   If a node does not meet its corresponding condition, the function returns `False`.

   ```
   if even and (root.val % 2 == 0 or prev >= root.val):
       return False
   if not even and (root.val % 2 == 1 or prev <= root.val):
       return False
   ```

6. **Track Previous Value**: The `prev` value is updated to the current node's value for the next iteration.

   ```
   prev = root.val
   ```

7. **Queue Update**: The left and right children of the current node are added to the queue if they exist.

   ```
   if root.left:
       q.append(root.left)
   if root.right:
       q.append(root.right)
   ```

8. **Toggle Even/Odd Level**: After each level is processed, the `even` flag is toggled to `1` if it was `0`, or to `0` if it was `1`. The toggle is done by using the XOR operator (`even ^= 1`).

   ```
   even ^= 1
   ```

9. **Completion**: After the entire tree has been traversed without returning `False`, the function returns `True`, confirming that the tree satisfies the Even-Odd condition.

By using a queue to manage nodes, this approach maintains a clear delineation between levels and employs BFS efficiently to verify the tree against the provided conditions level by level.

Example Walkthrough

Let's take a small binary tree to illustrate the solution approach.

Suppose we have the following binary tree:

```
      5
     / \
    10  7
   /    / \
  6    12  2
```

Let's walk through the steps:

1. **Initialize Variables**: Set `even` to `1` and put the root node with value 5 in the queue.

2. **BFS Loop**: Start the while loop since the queue is not empty.

3. **Level Traversal**: Process nodes in the current level. We have one node, which is 5.

4. **Initialization Per Level**: At level 0 (even level), we initialize `prev` to `inf`.

5. **Node Validation**: We check the value of the root node:

   - Since it's an even level, we expect an odd value and as it's the first value, it doesn't need to be increasing. The root's value is 5, which is odd, so it passes. The function continues.

6. **Track Previous Value**: The `prev` value becomes the current node's value, which is `5`.

7. **Queue Update**: Nodes `10` and `7` are children of `5`, so add them to the queue.

8. **Toggle Even/Odd Level**: At the end of the level, toggle `even` to `0`.

9. **BFS Loop**: Now we have `10` and `7` in the queue; the while loop continues.

10. **Level Traversal**: There are two nodes in this level.

11. **Initialization Per Level**: Set `prev` to `0` because we are at an odd level now.

12. **Node Validation**: Evaluate the nodes in this odd level:

    - We start with the node `10`. We expect an even value and decreasing order (however, as it's the first node of this level, we only check for an even value). Node `10` has an even value, so it passes for now.
    - We move on to the next node in the same level, which is `7`. We expect an even value and a value less than `prev` (which is `10`), but `7` is odd, so it fails the condition.

Since we detected a violation at node `7`, the function should return `False`.

This small example contradicts the Even-Odd condition in the second level and illustrates the checking mechanism per level using BFS traversal.

Python Solution

```python
from collections import deque

# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def isEvenOddTree(self, root: TreeNode) -> bool:
        # Initialize the level to 0 (0-indexed, so even)
        level = 0

        # Initialize the queue with the root node
        queue = deque([root])

        # Traverse the tree by levels
        while queue:
            # Depending on the current level, set the previous value accordingly
            # For even levels, we start comparing from the smallest possible value (0)
            # For odd levels, we start comparing from the largest possible value (infinity)
            previous_value = 0 if level % 2 == 0 else float('inf')

            # Process all nodes in the current level
            for _ in range(len(queue)):
                node = queue.popleft()

                # Check the Even-Odd Tree condition for the current node
                # At even levels, values must be odd and strictly increasing
                if level % 2 == 0 and (node.val % 2 == 0 or previous_value >= node.val):
                    return False
                # At odd levels, values must be even and strictly decreasing
                if level % 2 == 1 and (node.val % 2 == 1 or previous_value <= node.val):
                    return False

                # Update the previous value for the next comparison
                previous_value = node.val

                # Add child nodes to the queue
                if node.left:
                    queue.append(node.left)
                if node.right:
                    queue.append(node.right)

            # Move to the next level
            level += 1

        # If all conditions are met, return True
        return True

```

Java Solution

```java
// Definition of the binary tree node.
class TreeNode {
    int value;
    TreeNode left;
    TreeNode right;

    TreeNode() {}

    TreeNode(int value) {
        this.value = value;
    }

    TreeNode(int value, TreeNode left, TreeNode right) {
        this.value = value;
        this.left = left;
        this.right = right;
    }
}

class Solution {
    public boolean isEvenOddTree(TreeNode root) {
        // Tracks whether the current level is even (starting with the root level as even).
        boolean isLevelEven = true;

        // Queue for performing level order traversal
        Deque<TreeNode> queue = new ArrayDeque<>();
        queue.offer(root);

        // Loop while there are nodes in the queue
        while (!queue.isEmpty()) {
            // 'previousValue' will store the last value seen at the current level to check the strictly increasing or decreasing orc
            int previousValue = isLevelEven ? 0 : 1000001;

            // Number of nodes at the current level
            for (int levelSize = queue.size(); levelSize > 0; --levelSize) {
                root = queue.poll();

                // For even levels, values must be odd and strictly increasing
                if (isLevelEven && (root.value % 2 == 0 || previousValue >= root.value)) {
                    return false;
                }

                // For odd levels, values must be even and strictly decreasing
                if (!isLevelEven && (root.value % 2 == 1 || previousValue <= root.value)) {
                    return false;
                }

                // Update the 'previousValue' with the current node's value
                previousValue = root.value;

                // Add the left and right children, if they exist, to the queue for the next level
                if (root.left != null) {
                    queue.offer(root.left);
                }
                if (root.right != null) {
                    queue.offer(root.right);
                }
            }

            // Toggle the level indication for the next level traversal
            isLevelEven = !isLevelEven;
        }

        // If all levels meet the condition, return true
        return true;
    }
}

```

C++ Solution

```cpp
/**
 * Definition for a binary tree node.
 */
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode() : val(0), left(nullptr), right(nullptr) {}
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
    TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
};

class Solution {
public:
    /**
     * Checks whether a binary tree is an even-odd tree.
     * A binary tree is an even-odd tree if it meets the following conditions:
     * - At even-indexed levels, all nodes' values are odd, and they are
     *   increasing from left to right.
     * - At odd-indexed levels, all nodes' values are even, and they are
     *   decreasing from left to right.
     *
     * @param root The root of the binary tree.
     * @return True if the binary tree is an even-odd tree, otherwise false.
     */
    bool isEvenOddTree(TreeNode* root) {
        // 'isEvenLevel' indicates whether the current level is even or odd.
        bool isEvenLevel = true;
        // Queue for level order traversal.
        queue<TreeNode*> nodeQueue{{root}};

        while (!nodeQueue.empty()) {
            // 'previousValue' holds the previously encountered value in the current level.
            int previousValue = isEvenLevel ? 0 : INT_MAX;  // Initialize based on the level.
            // Process all nodes at the current level.
            for (int i = nodeQueue.size(); i > 0; --i) {
                root = nodeQueue.front();
                nodeQueue.pop();
                // Check whether the current node's value is appropriately odd or even.
                if (isEvenLevel && (root->val % 2 == 0 || previousValue >= root->val)) return false;
                if (!isEvenLevel && (root->val % 2 == 1 || previousValue <= root->val)) return false;

                previousValue = root->val;  // Update the 'previousValue'.
                // Add child nodes to the queue for the next level.
                if (root->left) nodeQueue.push(root->left);
                if (root->right) nodeQueue.push(root->right);
            }
            // Toggle the level indicator after finishing each level.
            isEvenLevel = !isEvenLevel;
        }

        return true;  // The tree meets the even-odd tree conditions.
    }
};

```

Typescript Solution

```typescript
interface TreeNode {
    val: number;
    left: TreeNode | null;
    right: TreeNode | null;
}

function isEvenOddTree(root: TreeNode | null): boolean {
    // 'isEvenLevel' indicates whether the current level is even or odd.
    let isEvenLevel = true;
    // Queue for level order traversal.
    let nodeQueue: (TreeNode | null)[] = [root];

    while (nodeQueue.length > 0) {
        // 'previousValue' holds the previously encountered value in the current level.
        let previousValue: number = isEvenLevel ? 0 : Number.MAX_SAFE_INTEGER; // Initialize based on the level.
        // Process all nodes at the current level.
        let levelSize = nodeQueue.length;
        for (let i = 0; i < levelSize; i++) {
            let node = nodeQueue.shift()!; // '!' asserts that 'node' won't be null.
            // Check whether the current node's value is appropriately odd or even.
            if (isEvenLevel && (node.val % 2 === 0 || previousValue >= node.val)) return false;
            if (!isEvenLevel && (node.val % 2 === 1 || previousValue <= node.val)) return false;

            previousValue = node.val; // Update the 'previousValue'.
            // Add child nodes to the queue for the next level.
            if (node.left) nodeQueue.push(node.left);
            if (node.right) nodeQueue.push(node.right);
        }
        // Toggle the level indicator after finishing each level.
        isEvenLevel = !isEvenLevel;
    }

    return true; // The tree meets the even-odd tree conditions.
}

```

Time and Space Complexity

// The time complexity of the provided code is `O(n)` where `n` is the number of nodes in the binary tree. This is because the code traverses each node exactly once. Each node is popped from the queue once, and its value is checked against the conditions for an even-odd tree, which takes constant time. The enqueue operations for the children of the nodes also happen once per node, maintaining the overall linear complexity with respect to the number of nodes.

// The space complexity of the code is `O(m)` where `m` is the maximum number of nodes at any level of the binary tree, or the maximum breadth of the tree. This is because the queue `q` can at most contain all the nodes at the level with the maximum number of nodes at some point during execution, which determines the maximum amount of space needed at any time.