### 2331. Evaluate Boolean Binary Tree

**Binary Tree** 

**Depth-First Search** 

Here's how we can think about our approach:

implementation steps and algorithms used:

### **Problem Description**

In this problem, we're given the root of a full binary <u>tree</u> representing a boolean expression. This tree has two types of nodes: leaf nodes and non-leaf nodes. Leaf nodes can only have a value of 0 (representing False) or 1 (representing True). Non-leaf nodes, on the other hand, hold a value of either 2 (representing the 0R operation) or 3 (representing the AND operation). The task is to evaluate this tree according to the following rules:

• If the node is a leaf, its evaluation is based on its value (True for 1, and False for 0).

- For non-leaf nodes, you need to evaluate both of its children and then apply the node's operation (OR for 2, AND for 3) to these evaluations.
- The goal is to return the boolean result of this evaluation, starting at the root of the <u>tree</u>.

A full binary tree, by definition, is a binary tree where each node has exactly 0 or 2 children. A leaf node is a node without any

children.

Intuition

## The solution to this problem lies in recursion, which matches the tree's structural nature. We will perform a post-order traversal to evaluate the tree – this means we first evaluate the child nodes, and then we apply the operation specified by their parent node.

If we reach a leaf node (a node with no children), we return its boolean value (True for 1, False for 0).
 If the node is not a leaf, it will have exactly two children because the tree is a full binary tree. We evaluate the left child and the right child first (recursively calling our function).

- 3. Once both children are evaluated, we perform either an OR operation if the node's value is 2 or an AND operation if the node's value is 3.

  4. We continue this process, working our way up the tree, until we reach the root.
- We continue this process, working our way up the tree, until we reach the root.
   The result of evaluating the root node gives us the answer to the problem.
  - This recursive algorithm effectively simulates the process of evaluating a complex boolean expression, starting from the most basic sub-expressions (the leaf nodes) and combining them as specified by the connecting operation nodes.
- Solution Approach

The solution is based on a simple recursive <u>tree</u> traversal algorithm. The <u>evaluateTree</u> function is a recursive function that evaluates whether the boolean expression represented by the <u>binary tree</u> is <u>True</u> or <u>False</u>. Let's take a deeper dive into the

# 1. **Base Case (Leaf Nodes):** If we come across a leaf node (which has no children), the evaluateTree function simply returns the boolean equivalent of the node's value. In Python, the boolean value True corresponds to an integer value 1, and False

corresponds to 0. Hence, bool(root.val) is sufficient to get the leaf node's boolean value.

2. Recursive Case (Non-Leaf Nodes): For non-leaf nodes, since the tree is full, it is guaranteed that if a node is not a leaf, it will have both left and right children. We first recursively evaluate the result of the left child self.evaluateTree(root.left) and store this in the variable 1. Similarly, we evaluate the result of the right child self.evaluateTree(root.right) and store it in

- the variable r.

  Combining Results: Once we have the boolean evaluations of the left and the right children, we check the value of the current (parent) node:
- If the parent node's value is 3, we perform an AND operation on the results of the children. This is done by 1 and r in Python.
   Termination and Return Value: The entire process recurses until the root node is evaluated. The result of evaluating the root node is then returned as the result of the boolean expression represented by the binary tree.
   Here is a breakdown of the method in pseudocode:

o If the parent node's value is 2, we perform an OR operation on the results of the children. This is done by 1 or r in Python.

- def evaluateTree(node):
   if node is a leaf:
   return the boolean value of the leaf
   else:
  - right\_child\_result = evaluateTree(node's right child)
    if node's value is OR:
     return left\_child\_result OR right\_child\_result
    else if node's value is AND:
     return left\_child\_result AND right\_child\_result

The elegance of the recursive approach is in its direct mapping to the tree structure and the natural way it processes nodes

```
according to the rules of boolean logic expression evaluation. At the end of the recursive calls, the evaluateTree function gives us the final boolean evaluation for the entire tree starting from the root node. This aligns perfectly with the expected solution for the problem.

Example Walkthrough

Let's walk through an example to illustrate how the solution approach works.

Imagine a small binary tree representing a boolean expression, where leaf nodes are either 0 (False) or 1 (True), and non-leaf
```

nodes are 2 (OR) or 3 (AND). Here is the structure of our example tree:

left\_child\_result = evaluateTree(node's left child)

This tree represents the boolean expression (True OR (False AND True)).

We start at the root node, which is not a leaf and has a value of 2, representing the OR operation. Since this is not a leaf node,

We evaluate the left child first. Here, the left child is a leaf with a value of 1 (True). According to our base case, we return True

Next, we evaluate the right child, which is a non-leaf and has a value of 3, representing the AND operation. Since this is not a

### we need to evaluate its children.

for this node.

Solution Implementation

self.val = val

self.left = left

self.right = right

def evaluateTree(self, root: Optional[TreeNode]) -> bool:

and internal nodes are either 2 (OR) or 3 (AND).

if root.left is None and root.right is None:

# Recursively evaluate the left subtree

left\_value = self.evaluateTree(root.left)

right\_value = self.evaluateTree(root.right)

# Recursively evaluate the right subtree

return bool(root.val)

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

\* and AND operations, respectively.

TreeNode(int val, TreeNode left, TreeNode right) {

\* @param root The root node of the binary tree.

\* @return The boolean result of evaluating the binary tree.

// Function to evaluate the value of a boolean binary tree

// Recursively evaluate the left subtree

bool leftResult = evaluateTree(root->left);

bool rightResult = evaluateTree(root->right);

// Recursively evaluate the right subtree

bool evaluateTree(TreeNode\* root) {

if (!root->left) {

\* Evaluates the boolean value of a binary tree with nodes labeled either

\* 0 (false), 1 (true), 2 (OR), or 3 (AND). Leaves of the tree will always

\* be labeled with 0 or 1. Nodes with values 2 or 3 represent the logical OR

**Python** 

class Solution:

TreeNode() {}

class Solution {

/\*\*

this.val = val;

this.left = left;

this.right = right;

5.

Now, let's step through the algorithm:

leaf node, we need to evaluate its children.

4. We evaluate the left child of the AND node, which is a leaf with a value of 0 (False). As per our base case, we return False for this node.

6. Now that we have the results of both children of the AND node (False and True), we apply the AND operation. False AND True results in False.

down the complex expression into simple operations that we could easily evaluate.

8. True OR False gives us True. So, the final result of evaluating the entire tree is True.

Hence, the boolean expression represented by the binary tree evaluates to True. The recursive approach allowed us to break

Returning back to the root of the tree, we now have results from both children - True from the left and False from the right

Moving to the right child of the AND node, we find it is a leaf with a value of 1 (True). We return True for this leaf.

(from the AND operation). The root is an OR node, so we apply the OR operation to these results.

- # Definition for a binary tree node.
  class TreeNode:
   def \_\_init\_\_(self, val=0, left=None, right=None):
- :param root: The root node of the binary tree.
  :return: The boolean result of evaluating the binary tree.
  """
  # If the current node is a leaf, return the boolean equivalent of its value

Evaluates the boolean value of a binary tree where leaves are 0 (False) or 1 (True),

# If the current node's value is 2, perform an OR operation; otherwise, perform an AND operation
if root.val == 2:
 return left\_value or right\_value
else:
 return left\_value and right\_value

Java

/\*\*
 \*\* Definition for a binary tree node.
 \*/
class TreeNode {
 int val; // The value of the node
 TreeNode left; // Reference to the left child
 TreeNode right; // Reference to the right child
 // Constructors

```
public boolean evaluateTree(TreeNode root) {
       // Base case: If the node has no children, return true if it's value is 1, false otherwise.
       if (root.left == null && root.right == null) {
           return root.val == 1;
       // Recursively evaluate the left and right subtrees.
       boolean leftEvaluation = evaluateTree(root.left);
       boolean rightEvaluation = evaluateTree(root.right);
       // Determine the value of the current expression based on the current node's value
       // If the node value is 2, perform logical OR, otherwise logical AND (as per the problem, value will be 3).
       return (root.val == 2) ? leftEvaluation || rightEvaluation : leftEvaluation && rightEvaluation;
C++
// Definition for a binary tree node.
struct TreeNode {
                        // Value of the node
   int val;
   TreeNode *left; // Pointer to the left child
   TreeNode *right;
                        // Pointer to the right child
   // Constructor to initialize a node with no children
   TreeNode() : val(0), left(nullptr), right(nullptr) {}
   // Constructor to initialize a node with a specific value and no children
   TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
   // Constructor to initialize a node with a specific value and specified children
   TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
class Solution {
public:
```

// based on the value of the root (0, 1, and 2 corresponding to false, true, and OR/AND operations)

// If the root does not have a left child, it must be a leaf node (value 0 or 1)

return root->val; // Return the value of the leaf node as the result

```
// If the root's value is 2, we perform an OR operation; otherwise, we perform an AND operation
       return root->val == 2 ? (leftResult || rightResult) : (leftResult && rightResult);
};
TypeScript
// Definition for a binary tree node.
interface TreeNode {
  val: number;
  left: TreeNode | null;
  right: TreeNode | null;
/**
* Evaluates the boolean value of a binary logic tree where leaves represent
* boolean values and other nodes represent logical operators.
 * @param {TreeNode | null} node - A node in a binary tree.
 * @return {boolean} - The evaluated boolean value of the tree.
*/
function evaluateTree(node: TreeNode | null): boolean {
 // Check if the node is null, which should not happen in a valid call
 if (!node) {
    throw new Error('Invalid node: Node cannot be null');
 // If the node is a leaf node (i.e., both children are null), return true if it's value is 1, else false
 if (node.left === null && node.right === null) {
    return node.val === 1;
 // Check the value of the node to determine the logical operator
 if (node.val === 2) {
   // Logical OR operator
    return evaluateTree(node.left as TreeNode) || evaluateTree(node.right as TreeNode);
 } else if (node.val === 3) {
   // Logical AND operator
    return evaluateTree(node.left as TreeNode) && evaluateTree(node.right as TreeNode);
```

// Note: The code assumes that the tree is a full binary tree and all non—leaf nodes have both left and right children.

# If the current node's value is 2, perform an OR operation; otherwise, perform an AND operation

// It also assumes that the leaf nodes have the value 1 (true) or 0 (false), while other nodes have the value 2 (OR) or 3 (AND).

```
def evaluateTree(self, root: Optional[TreeNode]) -> bool:
    """
    Evaluates the boolean value of a binary tree where leaves are 0 (False) or 1 (True),
    and internal nodes are either 2 (OR) or 3 (AND).

:param root: The root node of the binary tree.
    :return: The boolean result of evaluating the binary tree.
    """

# If the current node is a leaf, return the boolean equivalent of its value
    if root.left is None and root.right is None:
        return bool(root.val)
```

# Recursively evaluate the left subtree

left\_value = self.evaluateTree(root.left)

right\_value = self.evaluateTree(root.right)

# Recursively evaluate the right subtree

return left\_value or right\_value

return left\_value and right\_value

def \_\_init\_\_(self, val=0, left=None, right=None):

# Definition for a binary tree node.

self.val = val

self.left = left

self.right = right

if root.val == 2:

else:

class TreeNode:

class Solution:

throw new Error('Invalid node value: Node value must be either 1, 2, or 3');

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 function

evaluateTree visits each node exactly once to perform the evaluation.

The space complexity is O(h), where h is the height of the binary tree. This space is used by the call stack due to recursion. In the worst case of a skewed tree, where the tree is essentially a linked list, the height h is n, making the space complexity O(n). In the best case, with a balanced tree, the height h would be O(n), resulting in a space complexity of O(O(n)).