958. Check Completeness of a Binary Tree Breadth-First Search Binary Tree Medium Tree

## Leetcode Link

# In this problem, we are provided with the root node of a binary tree. Our objective is to determine whether the binary tree is a

level order traversal.

Problem Description

"complete binary tree" or not. A complete binary tree has a few defining characteristics: All levels are completely filled, except possibly the last level.

- This definition means that in a complete binary tree, if we traverse the nodes level by level from left to right, we will encounter all the
- non-null nodes first before any null node. If any null nodes are found, there should not be any more non-null nodes after them in this

On the last level, if it's not completely filled, all the nodes are as far left as possible.

To sum it up, the task is to check if the given binary tree meets the complete binary tree criteria. Intuition

known as breadth-first search), we enqueue all children of each node and process each node from the left to the right of each level.

To determine if a binary tree is complete, we can perform a level order traversal on the tree. In an ordinary level order traversal (also

## The solution is based on an observation that in a complete binary tree, once we encounter the first null node (representing a missing

traversal must also be null. If we find a non-null node after encountering a null, the tree cannot be considered complete. Here's the step-by-step reasoning for arriving at the solution:

child in the last level or an indication of no more nodes on the last level), all subsequent nodes encountered during the level order

1. Initiate a queue to perform level order traversal. Start by enqueuing the root node. 2. Begin the level order traversal, dequeuing a node from the front of the queue at each step. 3. Check the current node:

o If the node is null, this indicates the end of all the non-null nodes encountered so far for the complete binary tree. Any

If it's not null, enqueue its left and right children (regardless of whether they're null or not) for later processing.

The algorithm will employ a deque (short for double-ended queue) as it allows efficient insertions and deletions from both ends,

subsequent nodes should be null to ensure the tree is complete. 4. After encountering a null node, check the rest of the queue:

which is perfect for our level order traversal needs.

If all remaining nodes in the queue are null, the tree is complete.

the loop as we should not find any more non-null nodes after this point.

values if the current node doesn't have a corresponding child.

elements left in the queue adhere to the complete binary tree property.

Let's assume we have a binary tree structured like this:

- If any non-null node is found, then it's not a complete binary tree.
- **Solution Approach**
- The solution approach implements a level order traversal of the binary tree using a queue data structure. The standard queue is

replaced with a deque from Python's collections module for efficient manipulation of the queue's endpoints.

Here's a step-by-step overview of the implementation: Initialize a deque and add the root node of the binary tree to it.

## For each iteration, dequeue (popleft) an element from the front of the queue.

1 q = deque([root])

2. Begin iterating through the queue:

 This element represents the current node being processed. 1 while q: node = q.popleft()

3. The key condition: If the current node is None, it signals that we've encountered a gap in our level order traversal. Break out of

4. If the current node is not None, enqueue its children (left and right) to the back of the queue. These operations will include None

1 if node is None: break

q.append(node.left) 2 q.append(node.right) 5. After breaking out of the loop, we must verify that all the remaining nodes in the queue are None. This would confirm that we did

This method relies heavily on the properties of a complete binary tree observed during a level order traversal. By utilizing these

not encounter any non-null nodes after the first None node, which is a requirement for a complete binary tree.

properties, the algorithm effectively determines the completeness of the binary tree with an efficient traversal and a simple while loop.

Example Walkthrough

1 return all(node is None for node in q)

Implementing this approach provides an intuitive and direct method to verify that a binary tree is a complete binary tree. The use of a queue to keep track of nodes for processing and the break condition for encountering None values combine to form an elegant solution for this problem.

The pattern used in this method is a common Breadth-First Search (BFS) pattern, adapted to check the continuity of node presence

would imply the binary tree isn't complete. After the first None is seen, the result of the all function confirms whether the rest of the

at each level of the tree. The presence of a None value in the queue is used as an indicator to signal potential discontinuity, which

This binary tree is a complete binary tree because all levels except possibly the last one are fully filled, and all nodes in the last level are as far left as possible. Now let's walk through the solution approach with this binary tree as an example:

4. We repeat the above step for node with value 3, enqueuing its child with value 6. Since node 3 has no right child, we insert None.

### Since it is not None, we enqueue its children nodes with values 2 and 3. 1 while q:

1 q = deque([1])

2. We begin iterating through the queue.

node = q.popleft() # node with value 1

q.append(node.left) # node with value 2

q.append(node.right) # node with value 3

3. Since node with value 1 is not None, we continue with the loop:

We enqueue its children nodes with values 4 and 5.

1 node = q.popleft() # node with value 2 q.append(node.left) # node with value 4 3 q.append(node.right) # node with value 5

We again popleft() on the deque, this time getting the node with value 2.

1. We initialize a deque and add the root node of the binary tree (the node with value 1) to it.

• We dequeue an element from the front of the queue (popleft()), which is the root node with value 1.

5. Next, we process nodes 4, 5, and 6. Since they are at the last level and have no children, they would add None values to the

1 return all(node is None for node in q) # returns True, confirming the tree is complete

Eventually, we encounter the first None value after all nodes on the last level.

1 node = q.popleft() # node with value 4, adds None values to the queue

2 node = q.popleft() # node with value 5, adds None values to the queue

3 node = q.popleft() # node with value 6, adds None values to the queue

tree. In this case, all remaining elements are None, so the tree is complete.

1 from collections import deque # Import deque, as it's used for the queue.

# Initialize a queue with the root node to check level by level.

# When a None is encountered, the check for completeness starts.

# Append left and right children to the queue for further level checking.

return True # The tree is complete as all nodes after the first None are None.

# At this point, all remaining nodes in the queue should be None for a complete tree.

# This loop checks if there is any node that is not None after the first None was found.

return False # The tree is not complete as there is a node after a None.

# Iterate over the nodes until you find the first None, which signals incomplete level.

def isCompleteTree(self, root: TreeNode) -> bool:

current\_node = nodes\_queue.popleft()

nodes\_queue.append(current\_node.left)

nodes\_queue.append(current\_node.right)

4 node = q.popleft() # None, indicates end of non-null nodes

6. Since we encountered a None, we break out of the loop. 7. Now we check that the rest of the queue only contains None elements. If it does, we confirm that we have a complete binary

1 node = q.popleft() # node with value 3

3 q.append(node.right) # None

queue.

our example.

6

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

28

29

30

31

32

33

34

1 /\*\*

9

10

11

12

\*/

Python Solution

class Solution:

self.right = right

while nodes\_queue:

break

for node in nodes\_queue:

if node is not None:

36 # tree = TreeNode(1, TreeNode(2), TreeNode(3))

\* Definition for a binary tree node.

38 # print(solution.isCompleteTree(tree)) # Output: True

nodes\_queue = deque([root])

if current\_node is None:

2 g.append(node.left) # node with value 6

# Definition for a binary tree node. class TreeNode: def \_\_init\_\_(self, value=0, left=None, right=None): self.value = value self.left = left

By following this level order traversal and checking for the first occurrence of None, we have verified the binary tree is complete using

# Java Solution

class TreeNode {

int val;

TreeNode left;

TreeNode() {}

TreeNode right;

TreeNode(int val) {

this.val = val;

35 # Example usage:

37 # solution = Solution()

```
13
14
15
       TreeNode(int val, TreeNode left, TreeNode right) {
           this.val = val;
16
           this.left = left;
17
           this.right = right;
18
19
20
21
   class Solution {
23
       /**
        * Determine if a binary tree is complete.
24
25
        * A binary tree is complete if all levels are completely filled except possibly the last,
26
        * which is filled from left to right.
27
        * @param root The root node of the binary tree.
28
        * @return true if the tree is complete, false otherwise.
29
30
       public boolean isCompleteTree(TreeNode root) {
31
32
           // Queue to hold tree nodes for level order traversal
33
           Deque<TreeNode> queue = new LinkedList<>();
           queue.offer(root);
34
35
           // Traverse the tree using breadth-first search.
37
           // Keep adding children nodes until a null is encountered.
38
           while (queue.peek() != null) {
                TreeNode currentNode = queue.poll();
39
                queue.offer(currentNode.left);
40
                queue.offer(currentNode.right);
42
43
           // Remove any trailing nulls from the queue.
44
           // If any non-null nodes are found afterwards it means that the tree is not complete.
45
           while (!queue.isEmpty() && queue.peek() == null) {
46
                queue.poll();
48
49
50
           // If the queue is empty, then all the levels of the tree are fully filled.
51
           // Otherwise, the tree is not complete.
52
           return queue.isEmpty();
53
54 }
55
C++ Solution
   #include <queue>
   // Definition for a binary tree node.
   struct TreeNode {
       int val;
       TreeNode *left;
```

// Constructor initializes the node with a value, and the left and right pointers as null.

TreeNode(int x, TreeNode \*left, TreeNode \*right) : val(x), left(left), right(right) {}

nodeQueue.pop(); // Remove front of the queue after saving the node.

// After finding the first null pointer, all subsequent nodes must also be null.

TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}

TreeNode\* currentNode = nodeQueue.front();

// We use a queue to do a level order traversal of the tree.

// Continue level order traversal until we find a null pointer.

// Add child nodes to the queue for further processing.

while (!nodeQueue.empty() && nodeQueue.front() == nullptr) {

// Method to check if a binary tree is complete.

while (nodeQueue.front() != nullptr) {

nodeQueue.push(currentNode->left);

// Pop all null pointers from the queue.

nodeQueue.push(currentNode->right);

bool isCompleteTree(TreeNode\* root) {

nodeQueue.push(root);

nodeQueue.pop();

std::queue<TreeNode\*> nodeQueue;

### 36 37 // If the queue is empty, it means there were no non-null nodes after the first null 38 // which indicates the tree is complete. Otherwise, the tree is not complete. 39 return nodeQueue.empty(); 40

Typescript Solution

class TreeNode {

1 // Definition for a binary tree node.

TreeNode \*right;

10

12

16

17

18

19

20

22

23

24

25

26

27

28

29

30

31

32

33

34

35

41 };

42

11 };

14 public:

13 class Solution {

```
val: number;
       left: TreeNode | null;
       right: TreeNode | null;
       // Constructor initializes the node with a value, and the left and right pointers as null.
       constructor(val: number, left?: TreeNode | null, right?: TreeNode | null) {
           this.val = val;
9
           this.left = left === undefined ? null : left;
10
           this.right = right === undefined ? null : right;
11
12
13 }
14
   // Method to check if a binary tree is complete.
   function isCompleteTree(root: TreeNode | null): boolean {
       if (!root) {
           // If the root is null, the tree is trivially complete.
18
           return true;
20
21
22
       // We use a queue to do a level order traversal of the tree.
23
       const nodeQueue: (TreeNode | null)[] = [root];
24
25
       let foundNull = false;
26
27
       // Continue level order traversal until the queue is empty.
28
       while (nodeQueue.length > 0) {
29
           const currentNode = nodeQueue.shift();
30
           if (!currentNode) {
31
32
               // We encountered a null node; from this point forward, all nodes must be null.
33
               foundNull = true;
           } else {
34
35
               if (foundNull) {
36
                   // If we previously encountered a null node and now we see a non-null node,
37
                   // then the tree is not complete.
                   return false;
38
39
               // Add child nodes to the queue for further processing.
40
               nodeQueue.push(currentNode.left);
41
42
               nodeQueue.push(currentNode.right);
43
44
45
       // If we finished processing all nodes without finding a non-null
       // node after a null node, the tree is complete.
48
       return true;
Time and Space Complexity
```

omitted in Big O notation, the space complexity remains O(n).

```
The time complexity of the given code is O(n), where n is the number of nodes in the binary tree. This is because each node is
processed exactly once when it is dequeued for examination.
The space complexity of the code is also 0(n). In the worst-case scenario, the queue could contain all the nodes at the last level of a
complete binary tree, just before the loop terminates upon encountering a None node. This means that the number of elements in the
queue could approach the number of leaves in the full binary tree, which is approximately n/2. Considering the constant factor is
```