# Question

Given a binary tree, check whether it is a mirror of itself (ie, symmetric around its center).

For example, this binary tree [1,2,2,3,4,4,3] is symmetric:

1

/ \

2 2

/ \ / \

3 4 4 3

But the following [1,2,2,null,3,null,3] is not:

1

/ \

2 2

\ \

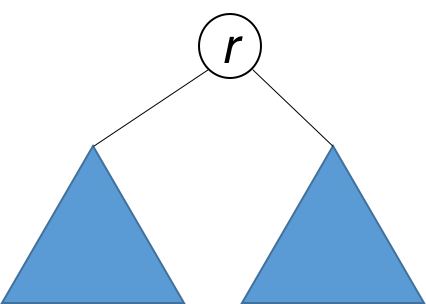
3 3

**Follow up:** Solve it both recursively and iteratively.

# Solution

#### **Approach 1: Recursive**

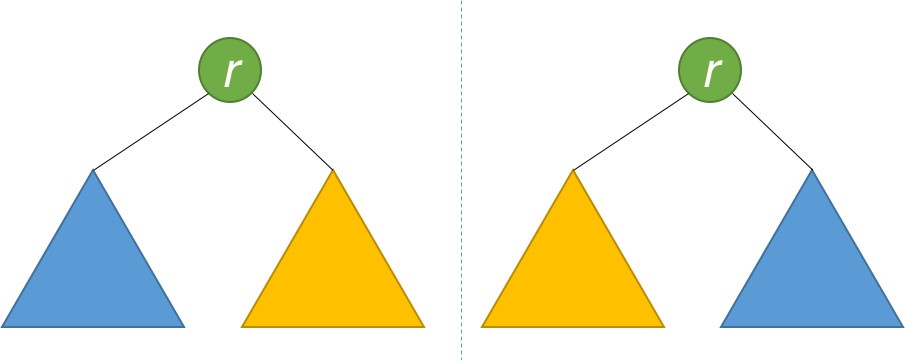
A tree is symmetric if the left subtree is a mirror reflection of the right subtree.



Therefore, the question is: when are two trees a mirror reflection of each other?

Two trees are a mirror reflection of each other if:

1. Their two roots have the same value.
2. The right subtree of each tree is a mirror reflection of the left subtree of the other tree.



This is like a person looking at a mirror. The reflection in the mirror has the same head, but the reflection's right arm corresponds to the actual person's left arm, and vice versa.

The explanation above translates naturally to a recursive function as follows.

|  |
| --- |
| public boolean isSymmetric(TreeNode root) {  return isMirror(root, root);  }  public boolean isMirror(TreeNode t1, TreeNode t2) {  if (t1 == null && t2 == null) return true;  if (t1 == null || t2 == null) return false;  return (t1.val == t2.val)  && isMirror(t1.right, t2.left)  && isMirror(t1.left, t2.right);  } |

**Complexity Analysis**

* Time complexity : O(n)*O*(*n*). Because we traverse the entire input tree once, the total run time is O(n)*O*(*n*), where n*n* is the total number of nodes in the tree.
* Space complexity : The number of recursive calls is bound by the height of the tree. In the worst case, the tree is linear and the height is in O(n)*O*(*n*). Therefore, space complexity due to recursive calls on the stack is O(n)*O*(*n*) in the worst case.

#### **Approach 2: Iterative**

Instead of recursion, we can also use iteration with the aid of a queue. Each two consecutive nodes in the queue should be equal, and their subtrees a mirror of each other. Initially, the queue contains root and root. Then the algorithm works similarly to BFS, with some key differences. Each time, two nodes are extracted and their values compared. Then, the right and left children of the two nodes are inserted in the queue in opposite order. The algorithm is done when either the queue is empty, or we detect that the tree is not symmetric (i.e. we pull out two consecutive nodes from the queue that are unequal).

|  |
| --- |
| public boolean isSymmetric(TreeNode root) {  Queue<TreeNode> q = new LinkedList<>();  q.add(root);  q.add(root);  while (!q.isEmpty()) {  TreeNode t1 = q.poll();  TreeNode t2 = q.poll();  if (t1 == null && t2 == null) continue;  if (t1 == null || t2 == null) return false;  if (t1.val != t2.val) return false;  q.add(t1.left);  q.add(t2.right);  q.add(t1.right);  q.add(t2.left);  }  return true;  } |

**Complexity Analysis**

* Time complexity : O(n)*O*(*n*). Because we traverse the entire input tree once, the total run time is O(n)*O*(*n*), where n*n* is the total number of nodes in the tree.
* Space complexity : There is additional space required for the search queue. In the worst case, we have to insert O(n)*O*(*n*) nodes in the queue. Therefore, space complexity is O(n)*O*(*n*).