Documentation

The task is to invert a binary tree, meaning that for every node in the tree, the left and right child nodes need to be swapped. Given the root of the tree, the goal is to return the root of the inverted tree after swapping all left and right subtrees recursively. Inverting a binary tree is essentially a process of mirroring the tree. For each node, we swap its left and right children, and then we continue to do the same for those children's left and right subtrees. This is done until we reach the leaf nodes, which are the base case where the recursion stops.

To solve this problem, the approach used is recursion. The function checks if the current node is None, and if so, it returns None since an empty node does not need to be inverted. Otherwise, the left and right children of the current node are swapped. After swapping, the function is recursively called on the left and right children to ensure that the entire subtree under each child is also inverted. Once all nodes in the tree are processed, the root of the inverted tree is returned.

The time complexity of this approach is O(n), where n is the total number of nodes in the tree. This is because each node is visited exactly once during the inversion process. The space complexity depends on the depth of the recursion stack, which is determined by the height of the tree. In the best case of a balanced tree, the height is log(n), while in the worst case of a skewed tree, the height is n, making the space complexity O(h), where h is the height of the tree.

This problem is a classic example of a tree manipulation algorithm, and the recursive nature of the solution makes it intuitive for tree structures. By carefully swapping each node’s children and ensuring that the operation is applied to all subtrees, we achieve the desired result of an inverted binary tree.