Solution: Invert Binary Tree

Let's solve the Invert Binary Tree problem using the Tree Depth-first Search pattern.

We'll cover the following

- Statement
- Solution
  - Time complexity
  - Space complexity

Statement

Given the root node of a binary tree, convert the binary tree into its mirror image, and return the root of the converted tree.

Constraints:

- $0 \leq$ Number of nodes in the tree ≤ 100
- $-1000 \leq$ `Node.value` ≤ 1000

Solution

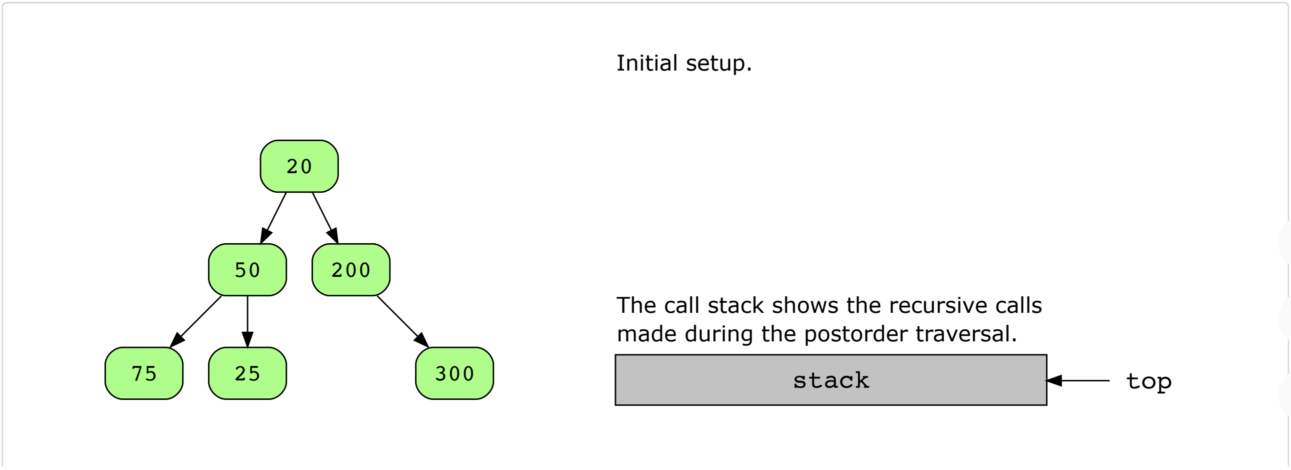
For this solution, we do a post-order (left, right, parent) traversal of the binary tree. The algorithm is as follows:

- Perform post-order traversal on the left child of the root node.
- Perform post-order traversal on the right child of the root node.
- Swap the left and right children of the root node.

Since we perform a depth-first search traversal, the children of any node are already mirrored even before we return the node itself.

Let’s look at an example below:

> **Note:** In this example, the node at the top of the stack is the current node.



Initial setup.
20
50
200
75
25
300
The call stack shows the recursive calls made during the postorder traversal.
stack
top

1 of 16

Java

main.java

BinaryTree.java

TreeNode.java

```java
class InvertTree {
    // Variables to support step-by-step printing
    static int change = 0;
    static TreeNode<Integer> masterRoot = null;
    // Function to mirror binary tree
    public static TreeNode<Integer> invertTree(TreeNode<Integer> root) {

        // Base case: end recursive call if current node is null
        if (root == null) {
           return null;
        }
        // We will do a post-order traversal of the binary tree.
        if(root.left!=null){
           invertTree(root.left);
        }
        if(root.right!=null){
          invertTree(root.right);
        }

        root.left = root.right;
        root.right = temp;

        // Only to demonstrate step-by-step operation of the solution
        if (masterRoot!=null && (root.left!=null || root.right!=null))
        {
            change++;
```

Invert Binary Tree

Time complexity

The time complexity of this solution is linear, $O(n)$, where n is the number of nodes in the tree.

> **Note:** Every subtree needs to be mirrored, so we visit every node once. Because of that, the runtime complexity is $O(n)$.

Space complexity

The space complexity of this solution is $O(h)$, where h is the height of the tree. This is because our recursive algorithm uses space on the call stack, which can grow to the height of the binary tree. The complexity will be $O(\log n)$ for a balanced tree and $O(n)$ for a degenerate tree.