Invert a Binary Tree - Notes

Problem Statement:

Given the root of a binary tree, invert the tree by swapping the left and right subtrees of every node, and return the new root.

Example:

Example 1:

Input:

```
   4
  / \
 2   7
/ \ / \
1   3 6  9
```

Output:

```
   4
  / \
 7   2
/ \ / \
9   6 3  1
```

Example 2:

Input:

```
 2
/ \
1   3
```

Output:

```
 2
/ \
3   1
```

Example 3:

Input: []
Output: []

Approach 1: Recursive (Depth-First Traversal)

- **Base Case:** If the node is NULL, return.
- **Swap Subtrees:**
  - Store the right subtree in a temporary variable.
  - Assign the left subtree to the right child.
  - Assign the temporary right subtree to the left child.
- **Recursive Calls:**
  - Call the function recursively for root->right (previously left).
  - Call the function recursively for root->left (previously right).

Code Implementation (Recursive)

```
class Solution {
public:
  // Helper function to invert the tree
  void helper(TreeNode* root){
    if(root == NULL) return;  // Base case: If node is NULL, return

    // Swap left and right subtrees
    TreeNode* temp = root->right;
    root->right = root->left;
    root->left = temp;

    // Recursively invert left and right subtrees
    helper(root->right);
    helper(root->left);
  }

  TreeNode* invertTree(TreeNode* root) {
    helper(root);  // Call the helper function
    return root;   // Return the new root after inversion
  }
};
```

Time and Space Complexity:

- **Time Complexity: O(n)** (Each node is visited once.)
- **Space Complexity: O(h)** (Recursion depth = height of the tree, worst case O(n) for skewed tree, best case O(log n) for balanced tree.)

Approach 2: Iterative (Breadth-First Search using Queue)

Instead of recursion, we can use **BFS with a queue**.

Code Implementation (Iterative)

```
class Solution {
public:
   TreeNode* invertTree(TreeNode* root) {
      if (root == NULL) return NULL;

      queue<TreeNode*> q;
      q.push(root);

      while (!q.empty()) {
         TreeNode* curr = q.front();
         q.pop();

         // Swap left and right child
         swap(curr->left, curr->right);

         // Push children into the queue
         if (curr->left) q.push(curr->left);
         if (curr->right) q.push(curr->right);
      }

      return root;
   }
};
```

Time and Space Complexity:

- **Time Complexity: O(n)**
- **Space Complexity: O(n)** (In the worst case, the queue holds all leaf nodes.)

Key Takeaways:

1. **Recursive Approach (DFS)**: Simple and easy to implement.
2. **Iterative Approach (BFS with Queue)**: Uses explicit queue instead of recursion.
3. **Time Complexity**: **O(n)** in both approaches.
4. **Space Complexity**: Recursive approach has **O(h)** space usage, whereas the iterative approach has **O(n)** in the worst case.

This problem is a fundamental **binary tree traversal** problem and helps in understanding **tree transformations** using recursion or iteration. 🚀