

226. Invert Binary Tree

- Easy
- Tree
- Depth-First Search
- Breadth-First Search
- Binary Tree

Problem Description

The given problem is a classic [tree](#) manipulation problem which involves inverting a [binary tree](#). In other words, for every node in the tree, you need to swap its left and right children. This operation should be applied recursively to all nodes of the tree, thus flipping the structure of the entire tree. As a result, the leftmost child becomes the rightmost child and vice versa, effectively creating a mirror image of the original tree. The challenge lies not just in performing the swap, but also in traversing the tree correctly to ensure all nodes are covered. Your task is to implement a function that takes the root of the binary tree as input and returns the new root of the inverted tree.

Intuition

To achieve the inversion of the [tree](#), we have to traverse it and for each node visited, its left and right children are swapped. This is a typical use case for a [Depth-First Search](#) (DFS) traversal. The DFS algorithm starts at the root node and explores as far as possible along each branch before backtracking. This perfectly suits our need to reach every node in order to invert the entire tree.

The solution approach is recursive in nature:

- Start with the root node.
- Swap the left and right child nodes of the current node.
- Recursively apply the same procedure to the left child node (which after swapping becomes the right child node).
- Recursively apply the same procedure to the right child node (which after swapping becomes the left child node).
- Return back to the previous stack call and continue this process until all nodes are visited.

At the end of the recursion, all nodes have their children swapped, and hence the [tree](#) is fully inverted, respecting the mirror image condition. Since the inversion needs to happen at each node, the time complexity is $O(n)$, where n is the number of nodes in the tree, because each node is visited once.

Solution Approach

The solution leverages the [Depth-First Search](#) (DFS) algorithm to traverse the [tree](#) and invert it at each node. To explain this step-by-step:

- A helper function `dfs()` is defined which will carry out the depth-first traversal and inversion. This function takes one argument: the current `root` node being visited.
- Inside `dfs()`, a base case is present where if the `root` is `None` (indicating either an empty [tree](#) or the end of a branch), the function simply returns as there's nothing to invert.
- If the node is not `None`, the function proceeds to swap the left and right child nodes.
 - This swapping is done with the Python tuple unpacking syntax: `root.left, root.right = root.right, root.left`.
- After the swap, `dfs()` is recursively called first with `root.left` and then with `root.right`. Note that after the swap, the original right child is now passed as `root.left` and vice-versa, hence following the inverted structure.
 - These two recursive calls ensure that every child node of the current `root` will also get inverted.
- The recursion will reach the leaf nodes and backtrack to the root, effectively inverting the subtrees as it goes up the call stack.
- Finally, once the root node's children are swapped and the recursive calls for its children are done, the whole tree is inverted.

`dfs(root)` completes its execution and the modified root node is returned by the `invertTree()` function.

Data structure used:

- A binary [tree](#) data structure is utilized with nodes following the definition of `TreeNode` which includes the `val`, `left`, and `right` attributes representing the node's value and its pointers to its left and right children respectively.

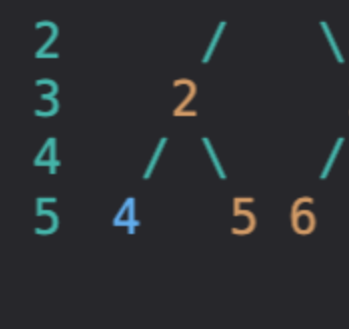
Pattern used:

- The pattern is recursion facilitated by DFS which is appropriate for [tree](#)-based problems where operations need to be performed on all nodes.

By applying this approach, each and every node in the [tree](#) is visited exactly once, and it is guaranteed that the tree will be inverted correctly. The time complexity of this approach is $O(n)$ where n is the total number of nodes in the tree, as each node is visited once during the traversal.

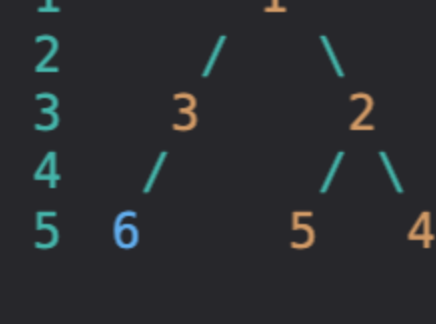
Example Walkthrough

Let's assume we have a simple binary tree:

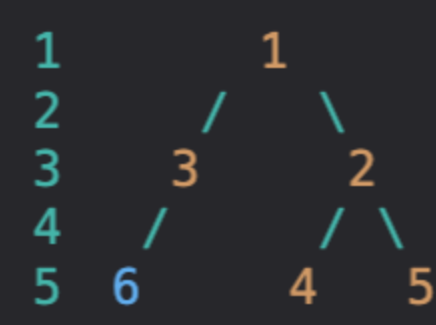


We want to invert this tree using the described solution approach. Here's how it happens step by step:

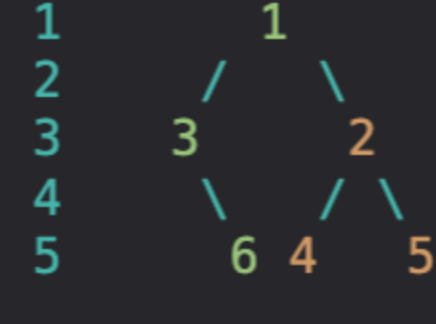
- We call `invertTree` on the root of the tree (node with value 1). This node is not `None`, so we proceed.
- Inside the `invertTree` function, we swap the children of node 1. Now the tree looks like this:



- We call `invertTree` recursively on the left child (node with value 3 which was originally the right child of 1). Node 3 also isn't `None`, so its children (node with value 6) are swapped, but since it's a leaf node with no children, the tree structure remains the same at this point.
- Next, we proceed to the right child of node 1 which is now the node with value 2. We swap the children of node 2. Now, the binary tree is:



- The node with value 2's left and right children (4 and 5) are leaf nodes and don't have children to swap. So they're left as is once reached by the recursive calls.
- After all recursive calls have completed, we have successfully inverted every node in the tree. The final structure of the binary tree is now:



The tree is now a mirror image of its original structure, and each step of our recursive DFS approach allowed us to visit and invert every node in the tree to achieve this. The `invertTree` function would then return the new root of this inverted tree, which completes the process.

Python Solution

```
1 class TreeNode:
2     """Definition for a binary tree node."""
3     def __init__(self, val=0, left=None, right=None):
4         self.val = val
5         self.left = left
6         self.right = right
7
8 class Solution:
9     def invertTree(self, root: Optional[TreeNode]) -> Optional[TreeNode]:
10         """Invert a binary tree.
11
12         Args:
13             root (Optional[TreeNode]): The root node of the binary tree.
14
15         Returns:
16             Optional[TreeNode]: The root node of the inverted binary tree.
17         """
18         def invert(node):
19             """Helper function to perform depth-first search and invert the tree.
20
21             Args:
22                 node (TreeNode): The current node to swap its children.
23             """
24             if node is None:
25                 return
26             # Swap the left child with the right child
27             node.left, node.right = node.right, node.left
28             # Recursively invert the left subtree
29             invert(node.left)
30             # Recursively invert the right subtree
31             invert(node.right)
32
33             # Start inverting the tree from the root
34             invert(root)
35             # Return the root of the inverted binary tree
36             return root
37
```

Java Solution

```
1 // Definition for a binary tree node.
2 class TreeNode {
3     int val; // The value contained in the node
4     TreeNode left; // Reference to the left child
5     TreeNode right; // Reference to the right child
6
7     // Constructor for creating a leaf node
8     TreeNode() {}
9
10    // Constructor for creating a node with a specific value
11    TreeNode(int val) { this.val = val; }
12
13    // Constructor for creating a node with a specific value and left/right children
14    TreeNode(int val, TreeNode left, TreeNode right) {
15        this.val = val;
16        this.left = left;
17        this.right = right;
18    }
19 }
20
21 // A solution class containing the method to invert a binary tree.
22 class Solution {
23
24     // Inverts a binary tree and returns the root of the inverted tree.
25     public TreeNode invertTree(TreeNode root) {
26         // Start the depth-first search inversion from the root node
27         depthFirstSearchInvert(root);
28         // Return the new root after inversion
29         return root;
30     }
31
32     // A helper method that uses Depth-First Search to invert the given binary tree recursively.
33     private void depthFirstSearchInvert(TreeNode node) {
34         // Base case: If the current node is null, there's nothing to invert; return immediately
35         if (node == null) {
36             return;
37         }
38
39         // Swap the left and right children of the current node
40         TreeNode tempNode = node.left;
41         node.left = node.right;
42         node.right = tempNode;
43
44         // Recursively invert the left subtree
45         depthFirstSearchInvert(node.left);
46         // Recursively invert the right subtree
47         depthFirstSearchInvert(node.right);
48     }
49 }
50
```

C++ Solution

```
1 #include <functional> // Include the functional header for std::function
2
3 // Definition for a binary tree node.
4 struct TreeNode {
5     int val;
6     TreeNode *left;
7     TreeNode *right;
8     // Constructor to initialize the node values
9     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
10    // Constructor to initialize the node values with given left and right children
11    TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
12 };
13
14 class Solution {
15 public:
16     // Method to invert a binary tree
17     TreeNode* invertTree(TreeNode* root) {
18         // Lambda function to recursively traverse the tree in a depth-first manner and invert it
19         std::function<void(TreeNode*>> depthFirstSearch = [&](TreeNode* node) {
20             // If the node is null, return immediately as there is nothing to invert
21             if (!node) {
22                 return;
23             }
24
25             // Swap the left and right children of the current node
26             std::swap(node->left, node->right);
27
28             // Invert the left subtree
29             depthFirstSearch(node->left);
30             // Invert the right subtree
31             depthFirstSearch(node->right);
32         });
33
34         // Start depth-first search from the root to invert the entire tree
35         depthFirstSearch(root);
36
37         // Return the root of the inverted tree
38         return root;
39     }
40 };
41
```

Typescript Solution

```
1 // TreeNode class definition
2 class TreeNode {
3     val: number;
4     left: TreeNode | null;
5     right: TreeNode | null;
6
7     constructor(val?: number, left?: TreeNode | null, right?: TreeNode | null) {
8         this.val = (val === undefined ? 0 : val); // Assign the node's value or default it to 0
9         this.left = (left === undefined ? null : left); // Assign the left child or default it to null
10        this.right = (right === undefined ? null : right); // Assign the right child or default it to null
11    }
12 }
13
14 /**
15  * Inverts a binary tree by swapping all left and right children.
16  *
17  * @param {TreeNode | null} treeRoot - The root of the binary tree to invert.
18  * @return {TreeNode | null} - The new root of the inverted binary tree.
19  */
20 function invertTree(treeRoot: TreeNode | null): TreeNode | null {
21     // Recursive function to traverse the tree and swap children
22     function invertNode(node: TreeNode | null): void {
23         if (node === null) {
24             return; // If the node is null, do nothing
25         }
26         [node.left, node.right] = [node.right, node.left]; // Swap the left and right children
27         invertNode(node.left); // Recursively invert the left subtree
28         invertNode(node.right); // Recursively invert the right subtree
29     }
30
31     invertNode(treeRoot); // Start inverting from the root node
32     return treeRoot; // Return the new root after inversion
33 }
34
```

Time and Space Complexity

The time complexity of the provided code is $O(n)$, where n is the number of nodes in the binary tree. This is because the function `dfs` is called exactly once for each node in the tree.

The space complexity of the code is also $O(n)$ in the worst case, corresponding to the height of the tree. This happens when the tree is skewed (i.e., each node has only one child). In this case, the height of the stack due to recursive calls is equal to the number of nodes. However, in the average case (a balanced tree), the space complexity would be $O(\log n)$, as the height of the tree would be logarithmic with respect to the number of nodes.