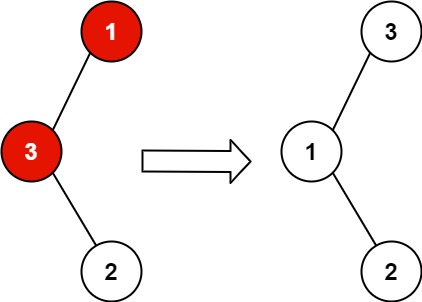
<https://leetcode.com/problems/recover-binary-search-tree/description/>

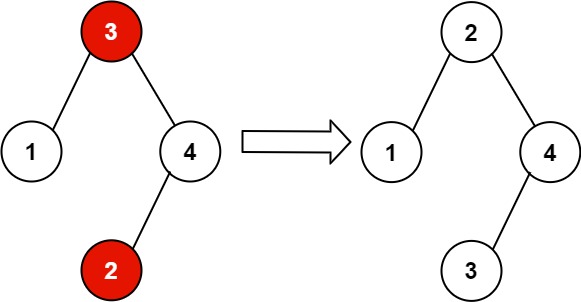
You are given the root of a binary search tree (BST), where the values of **exactly** two nodes of the tree were swapped by mistake. *Recover the tree without changing its structure*.

**Example 1:**



**Input:** root = [1,3,null,null,2]**Output:** [3,1,null,null,2]**Explanation:** 3 cannot be a left child of 1 because 3 > 1. Swapping 1 and 3 makes the BST valid.

**Example 2:**



**Input:** root = [3,1,4,null,null,2]**Output:** [2,1,4,null,null,3]**Explanation:** 2 cannot be in the right subtree of 3 because 2 < 3. Swapping 2 and 3 makes the BST valid.

**Constraints:**

The number of nodes in the tree is in the range [2, 1000].

-2^31 <= Node.val <= 2^31 - 1

 **Follow up:** A solution using O(n) space is pretty straight-forward. Could you devise a constant O(1) space solution?

**Attempt 1: 2025-03-25**

**Solution 1: Inorder Traversal O(N) Space Complexity Solution (30 min)**

**Style 1: Stack inorder traversal**

/\*\*

 \* Definition for a binary tree node.

 \* public class TreeNode {

 \*     int val;

 \*     TreeNode left;

 \*     TreeNode right;

 \*     TreeNode() {}

 \*     TreeNode(int val) { this.val = val; }

 \*     TreeNode(int val, TreeNode left, TreeNode right) {

 \*         this.val = val;

 \*         this.left = left;

 \*         this.right = right;

 \*     }

 \* }

 \*/

class Solution {

    public void recoverTree(TreeNode root) {

        List<TreeNode> inorderList = new ArrayList<>();

        Stack<TreeNode> stack = new Stack<>();

        TreeNode curr = root;

        while(curr != null || !stack.isEmpty()) {

            while(curr != null) {

                stack.push(curr);

                curr = curr.left;

            }

            curr = stack.pop();

            inorderList.add(curr);

            curr = curr.right;

        }

        TreeNode first = null;

        TreeNode second = null;

        for(int i = 0; i < inorderList.size() - 1; i++) {

            if(inorderList.get(i).val > inorderList.get(i + 1).val) {

                // The below logic design is for inorder list like

                // [1, 4, 3, 2, 5], the swapped nodes are 4 and 2

                if(first == null) {

                    first = inorderList.get(i);

                    second = inorderList.get(i + 1);

                } else {

                    second = inorderList.get(i + 1);

                    break;

                }

            }

        }

        int tmp = first.val;

        first.val = second.val;

        second.val = tmp;

    }

}

Time Complexity: O(n)

Space Complexity: O(n)

**Refer to Deepseek**

To solve the problem of recovering a Binary Search Tree (BST) where two nodes have been swapped, we can use an in-order traversal approach with O(n) space complexity. This method is more straightforward and easier to understand compared to the Morris Traversal approach, although it uses more space.

**Approach**

In-order Traversal: Perform an in-order traversal of the BST to collect all nodes in a list. This traversal should yield a sorted list in a valid BST.

Identify Swapped Nodes: Traverse the collected nodes to find the two nodes where the sorted order is violated. The first violation gives the larger misplaced node, and the second violation gives the smaller misplaced node.

Swap Values: Once the two misplaced nodes are identified, swap their values to restore the BST properties.

**Solution Code**

class Solution {

public void recoverTree(TreeNode root) {

List<TreeNode> inOrder = new ArrayList<>();

Stack<TreeNode> stack = new Stack<>();

TreeNode curr = root;

// Perform in-order traversal using a stack

while (curr != null || !stack.isEmpty()) {

while (curr != null) {

stack.push(curr);

curr = curr.left;

}

curr = stack.pop();

inOrder.add(curr);

curr = curr.right;

}

TreeNode first = null, second = null;

// Find the two misplaced nodes

for (int i = 0; i < inOrder.size() - 1; i++) {

if (inOrder.get(i).val > inOrder.get(i + 1).val) {

if (first == null) {

first = inOrder.get(i);

second = inOrder.get(i + 1);

} else {

second = inOrder.get(i + 1);

break;

}

}

}

// Swap the values of the two nodes

int temp = first.val;

first.val = second.val;

second.val = temp;

}

}

**Explanation**

In-order Traversal: The code uses a stack to perform an in-order traversal iteratively. Each node is visited in left-root-right order, and nodes are collected into a list.

Finding Violations: The list of nodes is scanned to find where the BST property is violated. The first violation identifies the larger node, and the second violation (if any) updates the smaller node.

Swapping Values: After identifying the two nodes, their values are swapped to correct the BST. This approach ensures the tree structure remains unchanged while restoring the correct node values.

This method uses O(n) space due to the list storing all nodes and the stack used for traversal, making it suitable for cases where simplicity and readability are prioritized over space efficiency.

**Style 2: Recursion inorder traversal - Action separate than inorder traversal**

/\*\*

 \* Definition for a binary tree node.

 \* public class TreeNode {

 \*     int val;

 \*     TreeNode left;

 \*     TreeNode right;

 \*     TreeNode() {}

 \*     TreeNode(int val) { this.val = val; }

 \*     TreeNode(int val, TreeNode left, TreeNode right) {

 \*         this.val = val;

 \*         this.left = left;

 \*         this.right = right;

 \*     }

 \* }

 \*/

class Solution {

    public void recoverTree(TreeNode root) {

        List<TreeNode> inorderList = new ArrayList<>();

        helper(inorderList, root);

        TreeNode first = null;

        TreeNode second = null;

        for(int i = 0; i < inorderList.size() - 1; i++) {

            if(inorderList.get(i).val > inorderList.get(i + 1).val) {

                // The below logic design is for inorder list like

                // [1, 4, 3, 2, 5], the swapped nodes are 4 and 2

                if(first == null) {

                    first = inorderList.get(i);

                    second = inorderList.get(i + 1);

                } else {

                    second = inorderList.get(i + 1);

                    break;

                }

            }

        }

        int tmp = first.val;

        first.val = second.val;

        second.val = tmp;

    }

    private void helper(List<TreeNode> list, TreeNode node) {

        if(node == null) {

            return;

        }

        helper(list, node.left);

        list.add(node);

        helper(list, node.right);

    }

}

Time Complexity: O(n)

Space Complexity: O(n)

**Refer to Deepseek**

To solve the problem of recovering a Binary Search Tree (BST) with two swapped nodes using an O(n) space approach, we can utilize a recursive in-order traversal. This method is straightforward and leverages the inherent properties of BSTs, where an in-order traversal yields a sorted sequence. By identifying the nodes that disrupt this order, we can correct the BST.

**Approach**

Recursive In-order Traversal: Perform a recursive in-order traversal to collect all nodes into a list. This traversal visits nodes in left-root-right order, which should produce a sorted list in a valid BST.

Identify Swapped Nodes: Traverse the collected nodes to find the two nodes where the sorted order is violated. The first violation identifies the larger misplaced node, and the second violation (if present) identifies the smaller misplaced node.

Swap Values: Once the two misplaced nodes are identified, swap their values to restore the BST properties.

**Solution Code**

class Solution {

public void recoverTree(TreeNode root) {

List<TreeNode> inOrder = new ArrayList<>();

inorderTraversal(root, inOrder); // Populate the list using recursive in-order traversal

TreeNode first = null, second = null;

// Find the two misplaced nodes

for (int i = 0; i < inOrder.size() - 1; i++) {

if (inOrder.get(i).val > inOrder.get(i + 1).val) {

if (first == null) {

first = inOrder.get(i);

second = inOrder.get(i + 1);

} else {

second = inOrder.get(i + 1);

break;

}

}

}

// Swap the values of the two nodes

int temp = first.val;

first.val = second.val;

second.val = temp;

}

// Helper method for recursive in-order traversal

private void inorderTraversal(TreeNode node, List<TreeNode> list) {

if (node == null) return;

inorderTraversal(node.left, list);

list.add(node);

inorderTraversal(node.right, list);

}

}

**Explanation**

Recursive In-order Traversal: The inorderTraversal method recursively visits each node in left-root-right order, appending nodes to a list. This list captures the in-order sequence of the BST.

Identify Violations: After collecting nodes, we iterate through the list to find where the BST property is violated (i.e., a node has a value greater than its successor). The first violation marks the larger misplaced node (first), and the second violation (if any) updates the smaller misplaced node (second).

Swap Values: The values of the identified nodes are swapped to correct the BST. This ensures the tree structure remains intact while restoring the correct node values.

This approach uses O(n) space due to the list storing all nodes, making it intuitive and easy to understand, though not space-optimal. It is ideal for scenarios where simplicity and readability are prioritized over minimizing space usage.

**Style 3: Recursion inorder traversal - Action merge into inorder traversal**

/\*\*

 \* Definition for a binary tree node.

 \* public class TreeNode {

 \*     int val;

 \*     TreeNode left;

 \*     TreeNode right;

 \*     TreeNode() {}

 \*     TreeNode(int val) { this.val = val; }

 \*     TreeNode(int val, TreeNode left, TreeNode right) {

 \*         this.val = val;

 \*         this.left = left;

 \*         this.right = right;

 \*     }

 \* }

 \*/

class Solution {

    TreeNode first = null;

    TreeNode second = null;

    TreeNode prev = null;

    public void recoverTree(TreeNode root) {

        helper(root);

        int tmp = first.val;

        first.val = second.val;

        second.val = tmp;

    }

    // Becareful, the exchanged node here should be 'root',

    // not 'prev', if given inorder traverse as 6, 3, 4, 5, 2

    // corresponding to 2

    private void helper(TreeNode node) {

        if(node == null) {

            return;

        }

        helper(node.left);

        // Start of "do some business"

        // If first element has not been found, assign it to prev

        // (refer to 6 in the example)

        if(first == null && (prev == null || prev.val > node.val)) {

            first = prev;

        }

        // If first element is found, assign the second element

        // to the root (refer to 2 in the example)

        if(first != null && prev.val > node.val) {

            second = node;

        }

        prev = node;

        helper(node.right);

    }

}

Time Complexity: O(n)

Space Complexity: O(n)

**Refer to**

<https://leetcode.com/problems/recover-binary-search-tree/discuss/32535/No-Fancy-Algorithm-just-Simple-and-Powerful-In-Order-Traversal>

<https://leetcode.com/problems/recover-binary-search-tree/discuss/32535/No-Fancy-Algorithm-just-Simple-and-Powerful-In-Order-Traversal/205091>

This question appeared difficult to me but it is really just a simple in-order traversal! I got really

frustrated when other people are showing off Morris Traversal which is totally not necessary here.

Let's start by writing the in order traversal:

private void traverse (TreeNode root) {

if (root == null)

return;

traverse(root.left);

// Do some business

traverse(root.right);

}

So when we need to print the node values in order, we insert System.out.println(root.val) in the place of

"Do some business".

What is the business we are doing here?

We need to find the first and second elements that are not in order right?

How do we find these two elements? For example, we have the following tree that is printed as in order traversal:

6, 3, 4, 5, 2

We compare each node with its next one and we can find out that 6 is the first element to swap because 6 > 3 and 2

is the second element to swap because 2 < 5.

Really, what we are comparing is the current node and its previous node in the "in order traversal".

Let us define three variables, firstElement, secondElement, and prevElement. Now we just need to build the "do

some business" logic as finding the two elements.

class Solution {

TreeNode firstElement = null;

TreeNode secondElement = null;

TreeNode prev = null;

public void recoverTree(TreeNode root) {

// In order traversal to find the two elements

helper(root);

// Swap the values of the two nodes

int temp = firstElement.val;

firstElement.val = secondElement.val;

secondElement.val = temp;

}

// Becareful, the exchanged node here should be 'root',

// not 'prev', if given inorder traverse as 6, 3, 4, 5, 2

// corresponding to 2

private void helper(TreeNode node) {

if(node == null) {

return;

}

helper(node.left);

// Start of "do some business",

// If first element has not been found, assign it to prevElement (refer to 6 in the example above)

if(firstElement == null && (prev == null || prev.val >= node.val)) {

firstElement = prev;

}

// If first element is found, assign the second element to the root (refer to 2 in the example above)

if(firstElement != null && prev.val >= node.val) {

secondElement = node;

}

prev = node;

// End of "do some business"

helper(node.right);

}

}

**Refer to Deepseek**

The two solutions approach the problem differently in terms of space usage and traversal logic. Here's a comparison to determine which is better:

**1. Space Complexity**

Action separate than inorder traversal Solution (List-Based):

Action merge into inorder traversalSolution (Recursive Tracking):

Winner: The Action merge into inorder traversal solution is more space-efficient in most cases.

**2. Time Complexity**

Both solutions perform O(n) in-order traversals and have the same time complexity.

**3. Correctness**

Both solutions correctly identify the swapped nodes:

Action separate than inorder traversal Solution:

Action merge into inorder traversal Solution:

Winner: Both are correct.

**4. Readability**

Action separate than inorder traversal Solution:

Action merge into inorder traversal Solution:

Winner: Action separate than inorder traversal solution is more readable for beginners.

**5. Problem Constraints**

The problem’s follow-up asks for an O(1) space solution (Morris Traversal). Neither of these solutions meets that, but between the two:

The Action merge into inorder traversal solution’s O(h) space is closer to optimal than Action separate than inorder traversal solution’s O(n).

**Final Verdict**

The Action merge into inorder traversal solution is generally better because:

It uses less space (O(h) vs. O(n)).

It avoids storing all nodes explicitly, which is critical for large trees.

It adheres more closely to the problem’s follow-up intent (even if not fully O(1)).

However, Action separate than inorder traversal solution’s readability makes it a better choice for learning/explaining the concept initially.

**Key Takeaways**

Use the Action merge into inorder traversal solution for optimal space efficiency.

Use Action separate than inorder traversal for clarity during learning or debugging.

For strict O(1) space, implement Morris Traversal (as in Action separate than inorder traversal solution)

**Solution 2: Morris Traversal O(1) Space Complexity Solution (180 min)**

**Refer to Deepseek**

To solve this problem, we need to recover a Binary Search Tree (BST) where exactly two nodes have been swapped by mistake. The goal is to restore the BST without changing its structure.

**Approach**

The key insight here is that an in-order traversal of a BST yields a sorted list of values. If two nodes are swapped, this sorted order will be disrupted. The approach involves identifying the two nodes that are out of order during an in-order traversal and then swapping their values back.

To achieve an optimal solution with O(1) space complexity, we use Morris Traversal. Morris Traversal allows us to perform an in-order traversal without using additional stack or recursion, thus using constant space.

**Explanation**

Morris Traversal: This traversal method allows us to traverse the tree in-order without using a stack or recursion. It uses threading to navigate back up the tree.

Identify Violations: During the traversal, we keep track of the previous node (prev). If we encounter a node where prev.val > current.val, it indicates a violation of the BST property.

Track Misplaced Nodes: The first violation's prev node is stored as first, and the current node as second. If a second violation is found, update second to the current node.

Swap Values: After identifying the two misplaced nodes, swap their values to restore the BST.

This approach efficiently finds and corrects the misplaced nodes with O(1) space complexity, making it optimal for large trees.

**Morris Traversal: In-order Traversal with O(1) Space**

Morris Traversal is an algorithm to traverse a binary tree in-order (left, root, right) without recursion or a stack, achieving O(1) space complexity. It works by \*\*temporarily modifying the tree structure\*\* (creating and removing threads) to keep track of traversal progress.

**Key Idea**

Threaded Binary Tree: Morris Traversal uses "threads" (temporary pointers) to link nodes in a way that allows backtracking without recursion/stack.

Steps:

Find the in-order predecessor (rightmost node in the left subtree).

Link it to the current node (creates a temporary path back).

Traverse left, process nodes, and remove threads when done.

**Algorithm Steps**

Initialize current = root.

While current is not null:

If current.left is null:

Process current (print/store its value).

Move current = current.right (go to the right subtree).

Else (if current.left exists):

Find current's in-order predecessor (pred):

pred = current.left

While pred.right exists and pred.right != current:

pred = pred.right

If pred.right is null:

Set pred.right = current (create a thread).

Move current = current.left (explore left subtree).

Else (pred.right == current):

Remove the thread (pred.right = null).

Process current.

Move current = current.right (done with left subtree).

**Example**

Consider the tree:

4

/ \

2 6

/ \ / \

1 3 5 7

Morris Traversal Order: 1 → 2 → 3 → 4 → 5 → 6 → 7

**Time & Space Complexity**

Time: O(n) (each node is visited at most twice).

Space: O(1) (no stack/recursion, just a few pointers).

**Why Use Morris Traversal?**

When O(1) space is required (e.g., embedded systems, large trees).

Avoids recursion stack overflow for deep trees.

Efficiently solves problems like "Recover BST" where modifying the tree temporarily is acceptable.

**Recover BST Using Morris Traversal (Java)**

class Solution {

public void recoverTree(TreeNode root) {

TreeNode first = null, second = null, prev = null;

TreeNode current = root;

while (current != null) {

if (current.left == null) {

// Check for swapped nodes

if (prev != null && prev.val > current.val) {

if (first == null) first = prev;

second = current;

}

prev = current;

current = current.right;

} else {

// Find in-order predecessor

TreeNode pred = current.left;

while (pred.right != null && pred.right != current) {

pred = pred.right;

}

if (pred.right == null) {

// Create a thread

pred.right = current;

current = current.left;

} else {

// Remove thread and check for swaps

pred.right = null;

if (prev != null && prev.val > current.val) {

if (first == null) first = prev;

second = current;

}

prev = current;

current = current.right;

}

}

}

// Swap the two misplaced nodes

if (first != null && second != null) {

int temp = first.val;

first.val = second.val;

second.val = temp;

}

}

}

**Advantages Over Recursion/Stack**

|  |  |  |  |
| --- | --- | --- | --- |
| Approach | Space Complexity | Modifies Tree? | Best For |
| **Recursion** | O(h) (stack) | ❌ No | Small/balanced trees |
| **Iterative (Stack)** | O(n) (worst case) | ❌ No | General cases |
| **Morris Traversal** | **O(1)** | ✅ Temporarily | Large trees, O(1) space req. |

**Conclusion**

Morris Traversal is optimal for O(1) space but modifies the tree temporarily.

Use stack/recursion if tree modification is not allowed.

Best for problems like "Recover BST" where space efficiency is critical.

Would you like a step-by-step dry run for a specific tree? 🚀