Saturday, March 12, 2022 11:48 AM

1932. Merge BSTs to Create Single BST

Hard

21222Add to ListShare

You are given `n` **BST (binary search tree) root nodes** for `n` separate BSTs stored in an array `trees` (**0-indexed**). Each BST in `trees` has **at most 3 nodes**, and no two roots have the same value. In one operation, you can:

- Select two **distinct** indices `i` and `j` such that the value stored at one of the **leaves** of `trees[i]` is equal to the **root value** of `trees[j]`.
- Replace the leaf node in `trees[i]` with `trees[j]`.
- Remove `trees[j]` from `trees`.

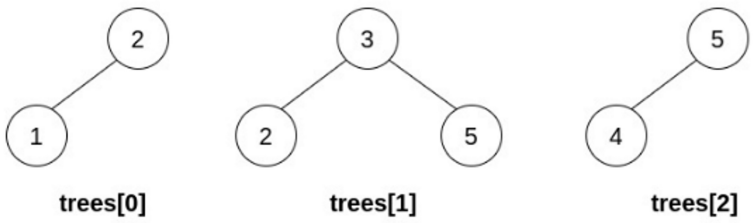
Return *the **root** of the resulting BST if it is possible to form a valid BST after performing* `n - 1` *operations, or* `null` *if it is impossible to create a valid BST*.

A BST (binary search tree) is a binary tree where each node satisfies the following property:

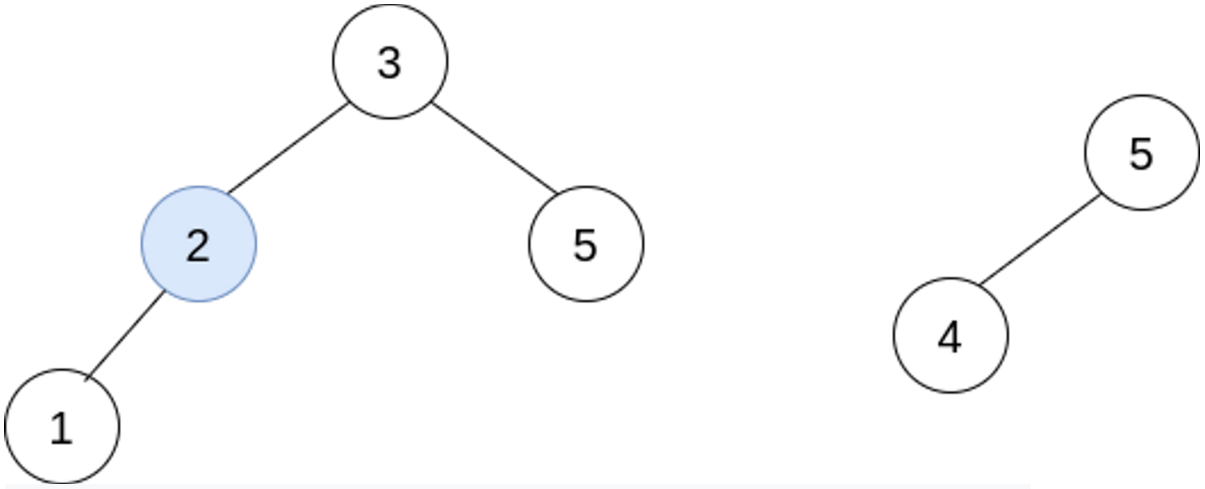
- Every node in the node's left subtree has a value **strictly less** than the node's value.
- Every node in the node's right subtree has a value **strictly greater** than the node's value.

A leaf is a node that has no children.

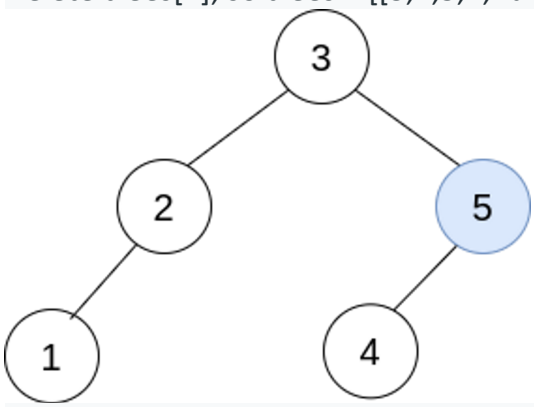
Example 1:



Input: trees = [[2,1],[3,2,5],[5,4]]
Output: [3,2,5,1,null,4]
Explanation:
In the first operation, pick i=1 and j=0, and merge trees[0] into trees[1].
Delete trees[0], so trees = [[3,2,5,1],[5,4]].



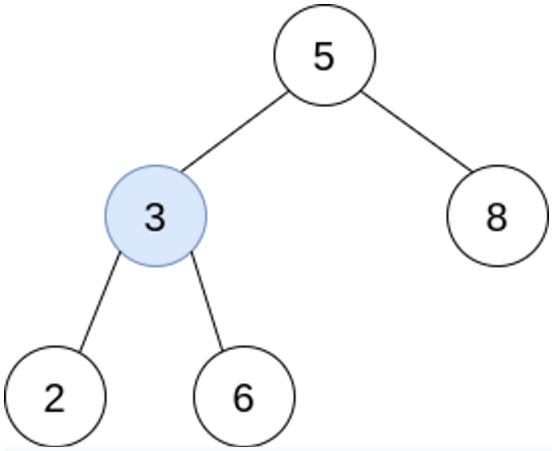
In the second operation, pick i=0 and j=1, and merge trees[1] into trees[0].
Delete trees[1], so trees = [[3,2,5,1,null,4]].



The resulting tree, shown above, is a valid BST, so return its root.

Example 2:

Input: trees = [[5,3,8],[3,2,6]]
Output: []
Explanation:
Pick i=0 and j=1 and merge trees[1] into trees[0].
Delete trees[1], so trees = [[5,3,8,2,6]].



The resulting tree is shown above. This is the only valid operation that can be performed, but the resulting tree is not a valid BST, so return null.

Example 3:

Input: trees = [[5,4],[3]]
Output: []
Explanation: It is impossible to perform any operations.

Constraints:

- `n == trees.length`
- `1 <= n <= 5 * 10^4`
- The number of nodes in each tree is in the range `[1, 3]`.
- Each node in the input may have children but no grandchildren.
- No two roots of `trees` have the same value.
- All the trees in the input are **valid BSTs**.
- `1 <= TreeNode.val <= 5 * 10^4`.

Merge Two Balanced Binary Search Trees

- Difficulty Level : Medium
- Last Updated : 14 Jan, 2022

You are given two balanced binary search trees e.g., AVL or Red-Black Tree. Write a function that merges the two given balanced BSTs into a balanced binary search tree. Let there be m elements in the first tree and n elements in the other tree. Your merge function should take O(m+n) time.

In the following solutions, it is assumed that the sizes of trees are also given as input. If the size is not given, then we can get the size by traversing the tree (See this).

Recommended: Please try your approach on ***{IDE}*** first, before moving on to the solution.

Method 1 (Insert elements of the **first tree to second)**

Take all elements of first BST one by one, and insert them into the second BST. Inserting an element to a self balancing BST takes Logn time (See this) where n is size of the BST. So time complexity of this method is Log(n) + Log(n+1) … Log(m+n-1). The value of this expression will be between mLogn and mLog(m+n-1). As an optimization, we can pick the smaller tree as first tree.

Method 2 (Merge Inorder Traversals)

1) Do inorder traversal of first tree and store the traversal in one temp array arr1[]. This step takes O(m) time.

2) Do inorder traversal of second tree and store the traversal in another temp array arr2[]. This step takes O(n) time.

3) The arrays created in step 1 and 2 are sorted arrays. Merge the two sorted arrays into one array of size m + n. This step takes O(m+n) time.

4) Construct a balanced tree from the merged array using the technique discussed in this post. This step takes O(m+n) time.

Time complexity of this method is O(m+n) which is better than method 1. This method takes O(m+n) time even if the input BSTs are not balanced.

Following is implementation of this method.

- C++
- C
- Java
- Python3
- C#
- Javascript

```
// Java program to Merge Two Balanced Binary Search Trees
import java.io.*;
import java.util.ArrayList;

// A binary tree node
```

```
class Node {

    int data;
    Node left, right;

    Node(int d) {
        data = d;
        left = right = null;
    }
}

class BinarySearchTree
{

    // Root of BST
    Node root;

    // Constructor
    BinarySearchTree() {
        root = null;
    }

    // Inorder traversal of the tree
    void inorder()
    {
        inorderUtil(this.root);
    }

// Utility function for inorder traversal of the tree
void inorderUtil(Node node)
{
    if(node==null)
        return;

    inorderUtil(node.left);
    System.out.print(node.data + " ");
    inorderUtil(node.right);
}


    // A Utility Method that stores inorder traversal of a tree
    public ArrayList<Integer> storeInorderUtil(Node node,
ArrayList<Integer> list)
    {
        if(node == null)
            return list;

        //recur on the left child
        storeInorderUtil(node.left, list);

        // Adds data to the list
        list.add(node.data);

        //recur on the right child
        storeInorderUtil(node.right, list);

        return list;
    }
```

```
    // Method that stores inorder traversal of a tree
    ArrayList<Integer> storeInorder(Node node)
    {
        ArrayList<Integer> list1 = new ArrayList<>();
        ArrayList<Integer> list2 = storeInorderUtil(node,list1);
        return list2;
    }

    // Method that merges two ArrayLists into one.
    ArrayList<Integer> merge(ArrayList<Integer>list1, ArrayList<Integer>
list2, int m, int n)
    {
        // list3 will contain the merge of list1 and list2
        ArrayList<Integer> list3 = new ArrayList<>();
        int i=0;
        int j=0;

        //Traversing through both ArrayLists
        while( i<m && j<n)
        {
            // Smaller one goes into list3
            if(list1.get(i)<list2.get(j))
            {
                list3.add(list1.get(i));
                i++;
            }
            else
            {
                list3.add(list2.get(j));
                j++;
            }
        }

        // Adds the remaining elements of list1 into list3
        while(i<m)
        {
            list3.add(list1.get(i));
            i++;
        }

        // Adds the remaining elements of list2 into list3
        while(j<n)
        {
            list3.add(list2.get(j));
            j++;
        }
        return list3;
    }

    // Method that converts an ArrayList to a BST
    Node ALtoBST(ArrayList<Integer>list, int start, int end)
    {
        // Base case
        if(start > end)
            return null;

        // Get the middle element and make it root
        int mid = (start+end)/2;
        Node node = new Node(list.get(mid));
```

```
        /* Recursively construct the left subtree and make it
        left child of root */
        node.left = ALtoBST(list, start, mid-1);

        /* Recursively construct the right subtree and make it
        right child of root */
        node.right = ALtoBST(list, mid+1, end);

    return node;
    }

    // Method that merges two trees into a single one.
    Node mergeTrees(Node node1, Node node2)
    {
        //Stores Inorder of tree1 to list1
        ArrayList<Integer>list1 = storeInorder(node1);

        //Stores Inorder of tree2 to list2
        ArrayList<Integer>list2 = storeInorder(node2);

        // Merges both list1 and list2 into list3
        ArrayList<Integer>list3 = merge(list1, list2, list1.size(),
list2.size());

        //Eventually converts the merged list into resultant BST
        Node node = ALtoBST(list3, 0, list3.size()-1);
        return node;
    }

    // Driver function
    public static void main (String[] args)
    {

        /* Creating following tree as First balanced BST
                100
                / \
                50 300
                / \
               20 70
        */

        BinarySearchTree tree1 = new BinarySearchTree();
        tree1.root = new Node(100);
        tree1.root.left = new Node(50);
        tree1.root.right = new Node(300);
        tree1.root.left.left = new Node(20);
        tree1.root.left.right = new Node(70);

        /* Creating following tree as second balanced BST
                80
                / \
             40 120
        */

        BinarySearchTree tree2 = new BinarySearchTree();
        tree2.root = new Node(80);
        tree2.root.left = new Node(40);
        tree2.root.right = new Node(120);
```

```
        BinarySearchTree tree = new BinarySearchTree();
        tree.root = tree.mergeTrees(tree1.root, tree2.root);
        System.out.println("The Inorder traversal of the merged BST is:
");
        tree.inorder();
    }
}
// This code has been contributed by Kamal Rawal
```

Output:

```
Following is Inorder traversal of the merged tree
20 40 50 70 80 100 120 300
```

Method 3 (In-Place Merge using DLL)

We can use a Doubly Linked List to merge trees in place. Following are the steps.

1) Convert the given two Binary Search Trees into doubly linked list in place (Refer this post for this step).

2) Merge the two sorted Linked Lists (Refer this post for this step).

3) Build a Balanced Binary Search Tree from the merged list created in step 2. (Refer this post for this step)

Time complexity of this method is also O(m+n) and this method does conversion in place.

Thanks to Dheeraj and Ronzii for suggesting this method.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

From <https://www.geeksforgeeks.org/merge-two-balanced-binary-search-trees/>