

# Problem 1932: Merge BSTs to Create Single BST

## Problem Information

Difficulty: **Hard**

Acceptance Rate: 0.00%

Paid Only: No

## Problem Description

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.

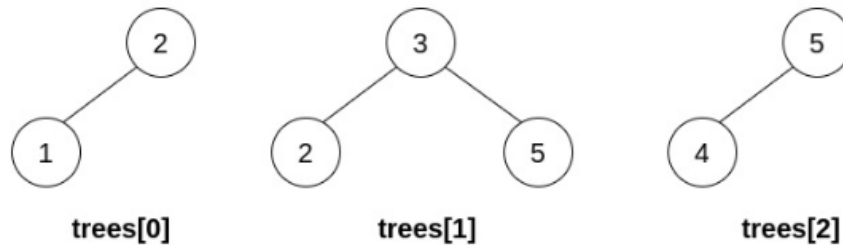
Every node in the node's right subtree has a value

strictly greater

than the node's value.

A leaf is a node that has no children.

Example 1:



Input:

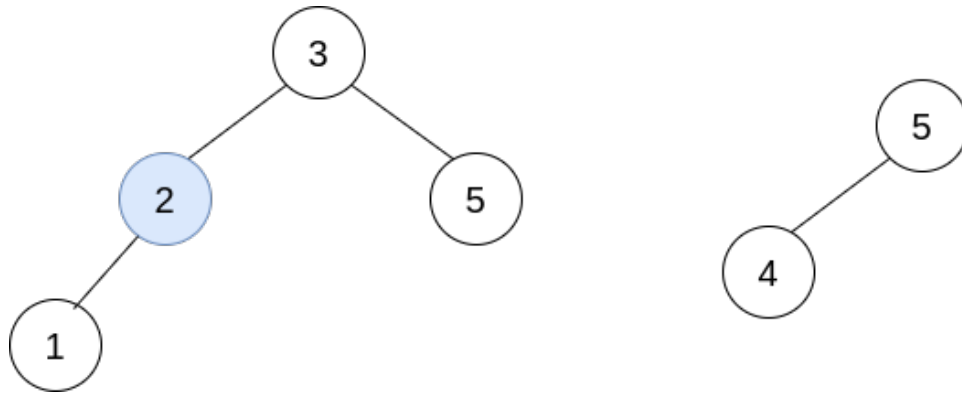
trees = [[2,1],[3,2,5],[5,4]]

Output:

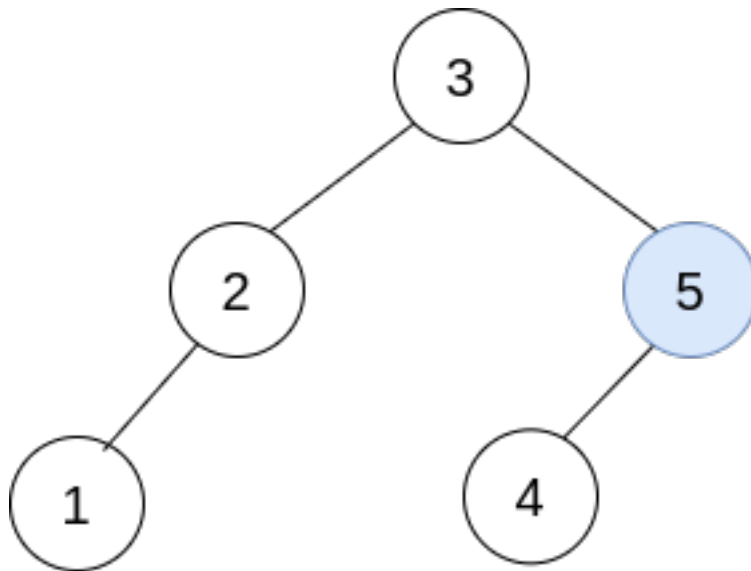
[3,2,5,1,null,4]

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]]`.

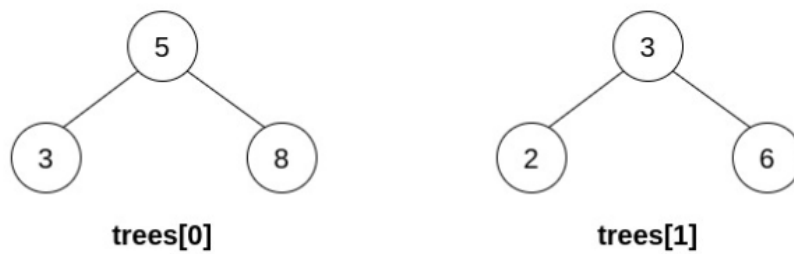


In the second operation, pick  $i=0$  and  $j=1$ , and merge  $\text{trees}[1]$  into  $\text{trees}[0]$ . Delete  $\text{trees}[1]$ , so  $\text{trees} = [[3,2,5,1,\text{null},4]]$ .



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

Example 2:



Input:

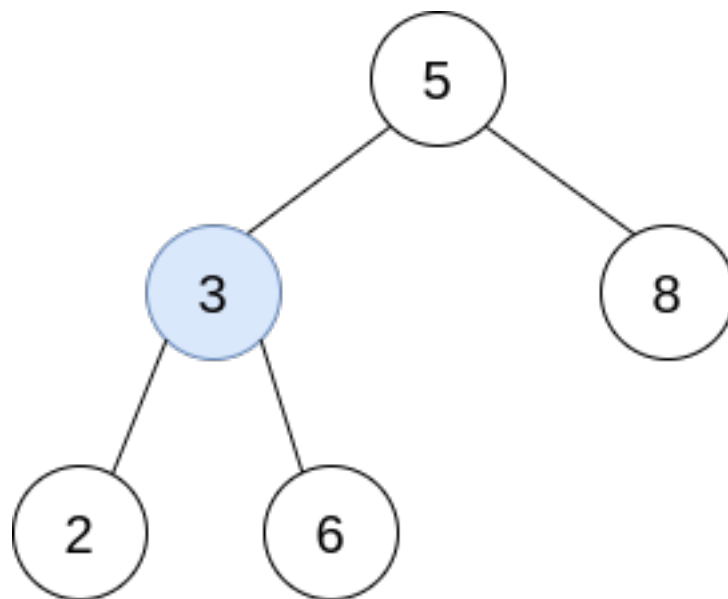
trees = [[5,3,8],[3,2,6]]

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

.

## Code Snippets

### C++:

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *   int val;
 *   TreeNode *left;
 *   TreeNode *right;
 *   TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *   TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *   TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left),
right(right) {}
 * };
 */
class Solution {
public:
    TreeNode* canMerge(vector<TreeNode*>& trees) {

    }
};
```

### Java:

```
/**
 * 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 TreeNode canMerge(List<TreeNode> trees) {

}

}

```

### Python3:

```

# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
def canMerge(self, trees: List[TreeNode]) -> Optional[TreeNode]:

```

### Python:

```

# Definition for a binary tree node.
# class TreeNode(object):
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution(object):
def canMerge(self, trees):
    """
    :type trees: List[TreeNode]
    :rtype: TreeNode
    """

```

### JavaScript:

```

/**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {

```

```

* this.val = (val===undefined ? 0 : val)
* this.left = (left===undefined ? null : left)
* this.right = (right===undefined ? null : right)
* }
*/
/**
* @param {TreeNode[]} trees
* @return {TreeNode}
*/
var canMerge = function(trees) {

};

```

## TypeScript:

```

/**
* Definition for a binary tree node.
* class TreeNode {
*   val: number
*   left: TreeNode | null
*   right: TreeNode | null
*   constructor(val?: number, left?: TreeNode | null, right?: TreeNode | null)
*   {
*     this.val = (val===undefined ? 0 : val)
*     this.left = (left===undefined ? null : left)
*     this.right = (right===undefined ? null : right)
*   }
* }
*/

function canMerge(trees: Array<TreeNode | null>): TreeNode | null {

};

```

## C#:

```

/**
* Definition for a binary tree node.
* public class TreeNode {
*   public int val;
*   public TreeNode left;
*   public TreeNode right;

```

```

* public TreeNode(int val=0, TreeNode left=null, TreeNode right=null) {
* this.val = val;
* this.left = left;
* this.right = right;
* }
* }
*/

public class Solution {
public TreeNode CanMerge(IList<TreeNode> trees) {

}

}

```

**C:**

```

/**
 * Definition for a binary tree node.
 * struct TreeNode {
 * int val;
 * struct TreeNode *left;
 * struct TreeNode *right;
 * };
 */

struct TreeNode* canMerge(struct TreeNode** trees, int treesSize){

}

```

**Go:**

```

/**
 * Definition for a binary tree node.
 * type TreeNode struct {
 * Val int
 * Left *TreeNode
 * Right *TreeNode
 * }
 */

func canMerge(trees []*TreeNode) *TreeNode {

}

```

## Kotlin:

```
/**
 * Example:
 * var ti = TreeNode(5)
 * var v = ti.`val`
 * Definition for a binary tree node.
 * class TreeNode(var `val`: Int) {
 *     var left: TreeNode? = null
 *     var right: TreeNode? = null
 * }
 */
class Solution {
    fun canMerge(trees: List<TreeNode?>): TreeNode? {

    }
}
```

## Swift:

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     public var val: Int
 *     public var left: TreeNode?
 *     public var right: TreeNode?
 *     public init() { self.val = 0; self.left = nil; self.right = nil; }
 *     public init(_ val: Int) { self.val = val; self.left = nil; self.right =
nil; }
 *     public init(_ val: Int, _ left: TreeNode?, _ right: TreeNode?) {
 *         self.val = val
 *         self.left = left
 *         self.right = right
 *     }
 * }
 */
class Solution {
    func canMerge(_ trees: [TreeNode?]) -> TreeNode? {

    }
}
```

## Rust:

```
// Definition for a binary tree node.
// #[derive(Debug, PartialEq, Eq)]
// pub struct TreeNode {
//     pub val: i32,
//     pub left: Option<Rc<RefCell<TreeNode>>>,
//     pub right: Option<Rc<RefCell<TreeNode>>>,
// }
//
// impl TreeNode {
//     #[inline]
//     pub fn new(val: i32) -> Self {
//         TreeNode {
//             val,
//             left: None,
//             right: None
//         }
//     }
// }

use std::rc::Rc;
use std::cell::RefCell;

impl Solution {
    pub fn can_merge(trees: Vec<Option<Rc<RefCell<TreeNode>>>>) ->
        Option<Rc<RefCell<TreeNode>>> {

    }
}
```

## Ruby:

```
# Definition for a binary tree node.
# class TreeNode
#   attr_accessor :val, :left, :right
#   def initialize(val = 0, left = nil, right = nil)
#     @val = val
#     @left = left
#     @right = right
#   end
# end

# @param {TreeNode[]} trees
# @return {TreeNode}
def can_merge(trees)
```

```
end
```

## PHP:

```
/**
 * Definition for a binary tree node.
 * class TreeNode {
 * public $val = null;
 * public $left = null;
 * public $right = null;
 * function __construct($val = 0, $left = null, $right = null) {
 * $this->val = $val;
 * $this->left = $left;
 * $this->right = $right;
 * }
 * }
 */
class Solution {

/**
 * @param TreeNode[] $trees
 * @return TreeNode
 */
function canMerge($trees) {

}

}
```

## Scala:

```
/**
 * Definition for a binary tree node.
 * class TreeNode(_value: Int = 0, _left: TreeNode = null, _right: TreeNode =
null) {
 * var value: Int = _value
 * var left: TreeNode = _left
 * var right: TreeNode = _right
 * }
 */
object Solution {
def canMerge(trees: List[TreeNode]): TreeNode = {
```

```
}  
}
```

## Racket:

```
; Definition for a binary tree node.  
#|  
  
; val : integer?  
; left : (or/c tree-node? #f)  
; right : (or/c tree-node? #f)  
(struct tree-node  
  (val left right) #:mutable #:transparent)  
  
; constructor  
(define (make-tree-node [val 0])  
  (tree-node val #f #f))  
  
|#  
  
(define/contract (can-merge trees)  
  (-> (listof (or/c tree-node? #f)) (or/c tree-node? #f))  
  
)
```

## Solutions

### C++ Solution:

```
/*  
 * Problem: Merge BSTs to Create Single BST  
 * Difficulty: Hard  
 * Tags: array, tree, hash, search  
 *  
 * Approach: Use two pointers or sliding window technique  
 * Time Complexity: O(n) or O(n log n)  
 * Space Complexity: O(h) for recursion stack where h is height  
 */
```

```

/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left),
 *     right(right) {}
 * };
 */
class Solution {
public:
    TreeNode* canMerge(vector<TreeNode*>& trees) {

    }
};

```

## Java Solution:

```

/**
 * Problem: Merge BSTs to Create Single BST
 * Difficulty: Hard
 * Tags: array, tree, hash, search
 *
 * Approach: Use two pointers or sliding window technique
 * Time Complexity: O(n) or O(n log n)
 * Space Complexity: O(h) for recursion stack where h is height
 */

/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode() {
 * // TODO: Implement optimized solution
 *     return 0;
 * }

```

```

* 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 TreeNode canMerge(List<TreeNode> trees) {

}
}

```

### Python3 Solution:

```

"""
Problem: Merge BSTs to Create Single BST
Difficulty: Hard
Tags: array, tree, hash, search

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(h) for recursion stack where h is height
"""

# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
def canMerge(self, trees: List[TreeNode]) -> Optional[TreeNode]:
# TODO: Implement optimized solution
pass

```

### Python Solution:

```

# Definition for a binary tree node.
# class TreeNode(object):

```

```

# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution(object):
def canMerge(self, trees):
    """
    :type trees: List[TreeNode]
    :rtype: TreeNode
    """

```

### JavaScript Solution:

```

/**
 * Problem: Merge BSTs to Create Single BST
 * Difficulty: Hard
 * Tags: array, tree, hash, search
 *
 * Approach: Use two pointers or sliding window technique
 * Time Complexity: O(n) or O(n log n)
 * Space Complexity: O(h) for recursion stack where h is height
 */

/**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
 *   this.val = (val===undefined ? 0 : val)
 *   this.left = (left===undefined ? null : left)
 *   this.right = (right===undefined ? null : right)
 * }
 */

/**
 * @param {TreeNode[]} trees
 * @return {TreeNode}
 */
var canMerge = function(trees) {

};

```

### TypeScript Solution:

```

/**
 * Problem: Merge BSTs to Create Single BST
 * Difficulty: Hard
 * Tags: array, tree, hash, search
 *
 * Approach: Use two pointers or sliding window technique
 * Time Complexity: O(n) or O(n log n)
 * Space Complexity: O(h) for recursion stack where h is height
 */

/**
 * Definition for a binary tree node.
 * class TreeNode {
 *   val: number
 *   left: TreeNode | null
 *   right: TreeNode | null
 *   constructor(val?: number, left?: TreeNode | null, right?: TreeNode | null)
 *   {
 *     this.val = (val===undefined ? 0 : val)
 *     this.left = (left===undefined ? null : left)
 *     this.right = (right===undefined ? null : right)
 *   }
 * }
 */

function canMerge(trees: Array<TreeNode | null>): TreeNode | null {

};

```

## C# Solution:

```

/*
 * Problem: Merge BSTs to Create Single BST
 * Difficulty: Hard
 * Tags: array, tree, hash, search
 *
 * Approach: Use two pointers or sliding window technique
 * Time Complexity: O(n) or O(n log n)
 * Space Complexity: O(h) for recursion stack where h is height
 */

/**

```

```

* Definition for a binary tree node.
* public class TreeNode {
* public int val;
* public TreeNode left;
* public TreeNode right;
* public TreeNode(int val=0, TreeNode left=null, TreeNode right=null) {
* this.val = val;
* this.left = left;
* this.right = right;
* }
* }
*/
public class Solution {
public TreeNode CanMerge(IList<TreeNode> trees) {

}
}

```

## C Solution:

```

/*
* Problem: Merge BSTs to Create Single BST
* Difficulty: Hard
* Tags: array, tree, hash, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

/**
* Definition for a binary tree node.
* struct TreeNode {
* int val;
* struct TreeNode *left;
* struct TreeNode *right;
* };
*/

struct TreeNode* canMerge(struct TreeNode** trees, int treesSize){

```

```
}
```

## Go Solution:

```
// Problem: Merge BSTs to Create Single BST
// Difficulty: Hard
// Tags: array, tree, hash, search
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(h) for recursion stack where h is height

/**
 * Definition for a binary tree node.
 * type TreeNode struct {
 *     Val int
 *     Left *TreeNode
 *     Right *TreeNode
 * }
 */
func canMerge(trees []*TreeNode) *TreeNode {

}
```

## Kotlin Solution:

```
/**
 * Example:
 * var ti = TreeNode(5)
 * var v = ti.`val`
 * Definition for a binary tree node.
 * class TreeNode(var `val`: Int) {
 *     var left: TreeNode? = null
 *     var right: TreeNode? = null
 * }
 */
class Solution {
fun canMerge(trees: List<TreeNode?>): TreeNode? {

}
```

```
}
```

### Swift Solution:

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 * public var val: Int
 * public var left: TreeNode?
 * public var right: TreeNode?
 * public init() { self.val = 0; self.left = nil; self.right = nil; }
 * public init(_ val: Int) { self.val = val; self.left = nil; self.right =
 nil; }
 * public init(_ val: Int, _ left: TreeNode?, _ right: TreeNode?) {
 * self.val = val
 * self.left = left
 * self.right = right
 * }
 * }
 */
class Solution {
func canMerge(_ trees: [TreeNode?]) -> TreeNode? {

}

}
```

### Rust Solution:

```
// Problem: Merge BSTs to Create Single BST
// Difficulty: Hard
// Tags: array, tree, hash, search
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(h) for recursion stack where h is height

// Definition for a binary tree node.
// #[derive(Debug, PartialEq, Eq)]
// pub struct TreeNode {
// pub val: i32,
// pub left: Option<Rc<RefCell<TreeNode>>>,

```

```

// pub right: Option<Rc<RefCell<TreeNode>>>,
// }
//
// impl TreeNode {
// #[inline]
// pub fn new(val: i32) -> Self {
//   TreeNode {
//     val,
//     left: None,
//     right: None
//   }
// }
// }
// }

use std::rc::Rc;
use std::cell::RefCell;

impl Solution {
  pub fn can_merge(trees: Vec<Option<Rc<RefCell<TreeNode>>>>) ->
    Option<Rc<RefCell<TreeNode>>> {

  }
}

```

### Ruby Solution:

```

# Definition for a binary tree node.
# class TreeNode
#   attr_accessor :val, :left, :right
#   def initialize(val = 0, left = nil, right = nil)
#     @val = val
#     @left = left
#     @right = right
#   end
# end

# @param {TreeNode[]} trees
# @return {TreeNode}
def can_merge(trees)

end

```

### PHP Solution:

```

/**
 * Definition for a binary tree node.
 * class TreeNode {
 * public $val = null;
 * public $left = null;
 * public $right = null;
 * function __construct($val = 0, $left = null, $right = null) {
 * $this->val = $val;
 * $this->left = $left;
 * $this->right = $right;
 * }
 * }
 */
class Solution {

/**
 * @param TreeNode[] $trees
 * @return TreeNode
 */
function canMerge($trees) {

}

}

```

### Scala Solution:

```

/**
 * Definition for a binary tree node.
 * class TreeNode(_value: Int = 0, _left: TreeNode = null, _right: TreeNode =
null) {
 * var value: Int = _value
 * var left: TreeNode = _left
 * var right: TreeNode = _right
 * }
 */
object Solution {
def canMerge(trees: List[TreeNode]): TreeNode = {

}

}

```

### Racket Solution:

```
; Definition for a binary tree node.
#|

; val : integer?
; left : (or/c tree-node? #f)
; right : (or/c tree-node? #f)
(struct tree-node
  (val left right) #:mutable #:transparent)

; constructor
(define (make-tree-node [val 0])
  (tree-node val #f #f))

|#

(define/contract (can-merge trees)
  (-> (listof (or/c tree-node? #f)) (or/c tree-node? #f))

)
```