1932. Merge BSTs to Create Single BST

Problem Description

You are given n binary search tree (BST) root nodes for n separate BSTs stored in an array called `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:

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:

1. The value of any node to the left is lesser than the value of the current node.
2. The value of any node to the right is greater than the value of the current node.

A leaf is a node that has no children.

Example

Suppose we have the following BSTs:

```
  2      3      5
 / \      \    / \
1   3      4  3   6
```

One possible sequence of operations to create a valid BST is:

1. Merge the first and second BSTs:

```
  2
 / \
1   3
     \
      4
```

2. Merge the resulting BST with the third BST:

```
    5
   / \
  2   6
 / \
1   3
     \
      4
```

Since we were able to create a valid BST, the function returns the root node of the final BST, in this case, the node with value `5`.

Solution Approach

The solution uses a depth-first search approach to build the final BST. Here are the steps of the algorithm:

1. Create two hash tables: `valToNode` to store each root node indexed by its value and `count` to store the frequencies of each value.
2. Iterate through the input `trees`, updating the hash tables.
3. For each tree in `trees`, check if the count of the tree's root value is 1. If it is, try to build a BST using a helper function `isValidBST`. If the resulting BST is valid and `valToNode` has at most one remaining entry, return the tree's root.
4. If no valid BST can be created, return `null`.

The helper function `isValidBST` performs a depth-first search to build a valid BST. It checks if the current tree node's value is within the specified range (minNode and maxNode) and whether the current node has children. If the current node has no children, it updates the node with the next node from `valToNode` and removes the entry from the hash table. The function continues checking the left and right subtrees and returns true if a valid BST is formed without any remaining entries in `valToNode`.

C++ Solution

```
cpp
#include <unordered_map>
#include <vector>

using namespace std;

// Definition of TreeNode
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) {
    unordered_map<int, TreeNode*> valToNode;  // {val: node}
    unordered_map<int, int> count;            // {val: freq}

    for (TreeNode* tree : trees) {
      valToNode[tree->val] = tree;
      ++count[tree->val];
      if (tree->left)
        ++count[tree->left->val];
      if (tree->right)
        ++count[tree->right->val];
    }

    for (TreeNode* tree : trees)
      if (count[tree->val] == 1) {
        if (isValidBST(tree, nullptr, nullptr, valToNode) &&
            valToNode.size() <= 1)
          return tree;
        return nullptr;
      }

    return nullptr;
  }

 private:
  bool isValidBST(TreeNode* tree, TreeNode* minNode, TreeNode* maxNode,
                  unordered_map<int, TreeNode*>& valToNode) {
    if (tree == nullptr)
      return true;
    if (minNode && tree->val <= minNode->val)
      return false;
    if (maxNode && tree->val >= maxNode->val)
      return false;
    if (!tree->left && !tree->right && valToNode.count(tree->val)) {
      const int val = tree->val;
      tree->left = valToNode[val]->left;
      tree->right = valToNode[val]->right;
      valToNode.erase(val);
    }

    return isValidBST(tree->left, minNode, tree, valToNode) &&
           isValidBST(tree->right, tree, maxNode, valToNode);
  }
};
```

In this C++ solution, two unordered maps are used to store the root nodes and their frequencies. The main function `canMerge` and the helper function `isValidBST` operate on these hash tables and TreeNode objects to build and validate the final BST.

In summary, this solution combines smaller BSTs into a final valid BST by performing a depth-first search and making use of hash tables to keep track of root nodes and their frequencies.### Python Solution

```
python
from collections import defaultdict
from typing import Optional

# Definition of TreeNode
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]:
        valToNode = {t.val: t for t in trees}  # {val: node}
        count = defaultdict(int)               # {val: freq}

        for tree in trees:
            count[tree.val] += 1
            if tree.left:
                count[tree.left.val] += 1
            if tree.right:
                count[tree.right.val] += 1

        for tree in trees:
            if count[tree.val] == 1:
                if self.isValidBST(tree, None, None, valToNode) and len(valToNode) <= 1:
                    return tree
                return None

        return None

    def isValidBST(self, tree: TreeNode, minNode: TreeNode, maxNode: TreeNode, valToNode: dict) -> bool:
        if not tree:
            return True
        if minNode and tree.val <= minNode.val:
            return False
        if maxNode and tree.val >= maxNode.val:
            return False
        if not tree.left and not tree.right and tree.val in valToNode:
            val = tree.val
            tree.left = valToNode[val].left
            tree.right = valToNode[val].right
            del valToNode[val]

        return self.isValidBST(tree.left, minNode, tree, valToNode) and \
            self.isValidBST(tree.right, tree, maxNode, valToNode)
```

The Python solution is very similar to the C++ solution, utilizing a dictionary to store the root nodes and their frequencies, and a `defaultdict` for the count. The main function `canMerge` and the helper function `isValidBST` work together to create and validate the final BST.

Java Solution

```
java
import java.util.HashMap;
import java.util.List;
import java.util.Map;

// Definition of TreeNode
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;
    }
}

public class Solution {
    public TreeNode canMerge(List<TreeNode> trees) {
        Map<Integer, TreeNode> valToNode = new HashMap<>();  // {val: node}
        Map<Integer, Integer> count = new HashMap<>();       // {val: freq}

        for (TreeNode tree : trees) {
            valToNode.put(tree.val, tree);
            count.put(tree.val, count.getOrDefault(tree.val, 0) + 1);
            if (tree.left != null)
                count.put(tree.left.val, count.getOrDefault(tree.left.val, 0) + 1);
            if (tree.right != null)
                count.put(tree.right.val, count.getOrDefault(tree.right.val, 0) + 1);
        }

        for (TreeNode tree : trees)
            if (count.get(tree.val) == 1) {
                if (isValidBST(tree, null, null, valToNode) && valToNode.size() <= 1)
                    return tree;
                return null;
            }

        return null;
    }

    private boolean isValidBST(TreeNode tree, TreeNode minNode, TreeNode maxNode, Map<Integer, TreeNode> valToNode) {
        if (tree == null)
            return true;
        if (minNode != null && tree.val <= minNode.val)
            return false;
        if (maxNode != null && tree.val >= maxNode.val)
            return false;
        if (tree.left == null && tree.right == null && valToNode.containsKey(tree.val)) {
            int val = tree.val;
            tree.left = valToNode.get(val).left;
            tree.right = valToNode.get(val).right;
            valToNode.remove(val);
        }

        return isValidBST(tree.left, minNode, tree, valToNode) &&
               isValidBST(tree.right, tree, maxNode, valToNode);
    }
}
```

The Java solution is also similar to both the C++ and Python solutions, utilizing a `HashMap` to store the root nodes and their frequencies. The main function `canMerge` and the helper function `isValidBST` work on these hash tables and `TreeNode` objects to build and validate the resulting BST.