Construct Binary Tree using Preorder and Inorder Traversal LeetCode

Given two integer arrays preorder and inorder where preorder is the preorder traversal of a binary tree and inorder is the inorder traversal of the same tree, construct and return *the binary tree*.

Example:

The input Inorder Traversal:

9 3 15 20 7

The input Preorder Traversal:

3 9 20 15 7

Output:

```
          3
        /   \
      9       20
            /   \
          15      7
```

Level Order Traversal:

Level 0: 3

Level 1: 9, 20

Level 2: 15, 7

Postorder Traversal:

9 15 7 20 3

Approach 1: Function to build a binary tree from preorder and inorder traversals

- Define a helper function **solve** that constructs a binary tree recursively:
    - Base case: If the inorder index is out of bounds or all nodes have been processed in the preorder traversal, return nullptr.
    - Extract the current element from the preorder traversal.
    - Create a new node with the current element.
    - Find the index of the current element in the inorder traversal.

    - Recursively build the left and right subtrees.
- In the **buildTree** function, initialize **preIndex** to 0 and call the **solve** function to create the binary tree.
- **Time Complexity: O(N) as it visits each node exactly once.**
- **Space Complexity: O(N) for the function call stack and O(N) for the input vectors.**

Approach 2: Function to build a binary tree from preorder and inorder traversals using a hashmap for optimized lookup

- Define a helper function **buildTreeHelper** that constructs a binary tree recursively using a hashmap for optimized lookup:
    - Base case: If the inorder index is out of bounds or all nodes have been processed in the preorder traversal, return nullptr.
    - Extract the current element from the preorder traversal.
    - Create a new node with the current element.
    - Find the index of the current element in the hashmap.
    - Recursively build the left and right subtrees.
- In the **buildTreeOptimized** function:
    - Initialize **preIndex** to 0 and **mp**, an unordered map that stores the indices of elements in the inorder traversal for efficient lookup.
    - Populate the hashmap with the indices of elements in the inorder traversal.
    - Call the **buildTreeHelper** function to create the binary tree.
- **Time Complexity: O(N) as it visits each node exactly once.**
- **Space Complexity: O(N) for the function call stack, O(N) for the hashmap, and O(N) for the input vectors.**

Conclusion:

- Both approaches effectively construct a binary tree from the given inorder and preorder traversals.
- The output demonstrates the level-order traversal of the binary tree and its postorder traversal. Both methods yield the same tree structure.
- The optimized approach using a hashmap for efficient lookup may offer better performance in terms of time complexity, especially for larger input trees.