Construct Binary Tree using Postorder and Inorder Traversal LeetCode

Example:

The input Inorder Traversal:

9 3 15 20 7

The input Postorder Traversal:

9 15 7 20 3

Output:

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

Level Order Traversal:

Level 0: 3

Level 1: 9, 20

Level 2: 15, 7

Preorder Traversal:

3 9 20 15 7

Approach 1: Function to build a binary tree from inorder and postorder 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 postorder traversal, return nullptr.
    - Extract the current element from the postorder traversal.
    - Create a new node with the current element.
    - Find the index of the current element in the inorder traversal.
    - Recursively build the right and left subtrees (note the order).

- In the **buildTree** function, initialize **postIndex** to **size - 1** 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 inorder and postorder traversals with an unordered map for efficient search

- Define a helper function **buildtreeHelper** that constructs a binary tree recursively using a hashmap for efficient search:
    - Base case: If the inorder index is out of bounds or all nodes have been processed in the postorder traversal, return nullptr.
    - Extract the current element from the postorder traversal.
    - Create a new node with the current element.
    - Find the index of the current element in the inorder traversal using the unordered map.
    - Recursively build the right and left subtrees (note the order).
- In the **buildTreeOptimized** function:
    - Initialize **postIndex** to **size - 1** and **mp**, an unordered map that stores the indices of elements in the inorder traversal for efficient search.
    - Create a mapping of elements to their indices 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 unordered map, and O(N) for the input vectors.**

Conclusion:

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