📌 Building a Binary Tree from Preorder & Inorder Traversal

🔹 Problem Statement:

Given preorder and inorder traversal arrays, construct the corresponding binary tree.

🔥 Approach & Logic

1️⃣Identifying the Root (Preorder Property)

- The **first element** in preorder is always the **root**.
- Example:
- preorder = [3, 9, 20, 15, 7]
- inorder = [9, 3, 15, 20, 7]

Here, 3 is the first element, so it becomes the **root**.

2️⃣Finding Left and Right Subtrees (Inorder Property)

- **Elements before the root in inorder belong to the left subtree**.
- **Elements after the root in inorder belong to the right subtree**.
- Example:
- inorder = [9, 3, 15, 20, 7]
-             ↑
-             (Root at index 1)
    - Left Subtree: [9]
    - Right Subtree: [15, 20, 7]

3️⃣Recursively Building the Subtrees

- Recursively pass the relevant parts of preorder and inorder for left and right subtree construction.
- LeftCount and RightCount track subtree sizes.

🔄 Understanding the Variables Used in Code

- pre & in: Vectors storing the **preorder** and **inorder** traversal.
- prelo, prehi: Indices marking the **current segment** of preorder being processed.
- inlo, inhi: Indices marking the **current segment** of inorder being processed.
- i: Iterator used to **find the index of root** in inorder.
- LeftCount: Number of elements in the **left subtree** (i - inlo).

- RightCount: Number of elements in the **right subtree** (inhi - i).
- root: The **current root node** being created.

🔄 Dry Run Example 1

Input:

preorder = [1, 2, 4, 5, 3, 6]

inorder = [4, 2, 5, 1, 3, 6]

Step 1: Identify Root

- preorder[0] = 1 → Root
- inorder index of 1 = 3
- Left Subtree: [4, 2, 5]
- Right Subtree: [3, 6]

```
    1
   / \
  ?   ?
```

Step 2: Left Subtree

- preorder[1] = 2
- inorder index of 2 = 1
- Left: [4], Right: [5]

```
     1
    / \
   2   ?
  / \
 4   5
```

Step 3: Right Subtree

- preorder[4] = 3
- inorder index of 3 = 4
- Left: [], Right: [6]

```
    1
   / \
  2   3
```

```
 / \   \
4   5    6
```

📝 Code Implementation

```
class Solution {
public:
    TreeNode* build(vector<int>& pre, int prelo, int prehi,
                    vector<int>& in, int inlo, int inhi) {
        // Base Case 1: If the inorder range is invalid, return NULL
        if (inlo > inhi) return NULL;

        // Create the root node from the preorder traversal
        TreeNode* root = new TreeNode(pre[prelo]);

        // Base Case 2: If there is only one element, return that node
        if (prelo == prehi) return root;

        // Find the index of the root in inorder
        int i = inlo;
        while (i <= inhi) {
            if (in[i] == pre[prelo]) break;
            i++;
        }

        int LeftCount = i - inlo;
        int RightCount = inhi - i;

        root->left = build(pre, prelo + 1, prelo + LeftCount, in, inlo, i - 1);
        root->right = build(pre, prelo + LeftCount + 1, prehi, in, i + 1, inhi);

        return root;
```

```
    }

    TreeNode* buildTree(vector<int>& pre, vector<int>& in) {
        int n = pre.size();
        return build(pre, 0, n - 1, in, 0, n - 1);
    }
};
```

🛡️ Key Takeaways

1. **First element of preorder is always the root.**
2. **inorder helps identify left and right subtrees.**
3. **Recursion is used to construct the subtrees.**
4. **Index tracking helps manage subtree sizes.**
5. **Each recursive call processes a smaller portion of preorder and inorder.**
6. **Using a hashmap for inorder lookup reduces complexity to O(N).**
7. **Base Case 1 ensures recursion stops when there are no elements left to process.**
8. **Base Case 2 ensures a single node is returned when no children exist.**

This structured approach ensures an efficient and clear understanding of binary tree construction. 🚀