Construct Binary Tree from Inorder and Postorder Traversal

Given inorder and postorder traversal of a tree, construct the binary tree.

Note:
You may assume that duplicates do not exist in the tree.

Solution 1

Below is the O(n) solution from @hongzhi but that discuss is closed now 'cause @hongzhi says little about his code.

https://oj.leetcode.com/discuss/6334/here-is-my-o-n-solution-is-it-neat

I've modified some of and tried this code and got AC. Just share about some comprehension about his code.

I've modified vtn(vector) to stn(stack) in that **stack** is probably what this algs means and needs.

What matters most is the meaning of *stn*.

Only nodes whoes left side **hasn't been** handled will be pushed into *stn*.

And inorder is organized as (inorder of left) root (inorder of right),

And postorder is as (postorder of left) (postorder of right) root.

So at the very begin, we only have root in stn and we check if *inorder.back() == root->val* and in most cases it's **false**(see Note 1). Then we make this node root's right sub-node and push it into stn.

Note 1: this is actually *(inorder of right).back() == (postorder of right).back()*, so if only there's no right subtree or the answer will always be false.

Note 2: we delete one node from *postorder* as we push one into stn.

Now we have [root, root's right] as stn and we check *inorder.back() == stn.top()->val* again.

- **true** means *inorder.back()* is the root node and needs handled left case.
- **false** means *inorder.back()* is the next right sub-node

So when we encounter a true, we will cache *stn.top()* as p and **delete both nodes from inorder and stn**.

Then we check inorder.size(), if there's no nodes left, it means p has no left node.

Else the next node in inorder could be *p's left node* or *p's father* which equals to the now *stn.top()* (remember we popped *p* from *stn* above).

If the latter happens, it means *p* has **no left node** and we need to move on to *p's father(stn.top())*.

If the former happens, it means *p* has one left node and it's *postorder.back()*, so we put it to p's left and delete it from the *postorder* and push the left node into *stn* 'cause **it** should be the next check node as the *postorder* is organized as above.

That's all of it. The algs just build a binary tree. :)

Inform me if there's anything vague or wrong, I'm open to any suggestions.

```
class Solution {
public:
    TreeNode *buildTree(vector<int> &inorder, vector<int> &postorder) {
        if(inorder.size() == 0)return NULL;
        TreeNode *p;
        TreeNode *root;
        stack<TreeNode *> stn;

        root = new TreeNode(postorder.back());
        stn.push(root);
        postorder.pop_back();

        while(true)
        {
            if(inorder.back() == stn.top()->val)
            {
                p = stn.top();
                stn.pop();
                inorder.pop_back();
                if(inorder.size() == 0) break;
                if(stn.size() && inorder.back() == stn.top()->val)
                    continue;
                p->left = new TreeNode(postorder.back());
                postorder.pop_back();
                stn.push(p->left);
            }
            else
            {
                p = new TreeNode(postorder.back());
                postorder.pop_back();
                stn.top()->right = p;
                stn.push(p);
            }
        }
        return root;
    }
};
```

written by Lancelod_Liu original link here

Solution 2

The the basic idea is to take the last element in postorder array as the root, find the position of the root in the inorder array; then locate the range for left sub-tree and right sub-tree and do recursion. Use a HashMap to record the index of root in the inorder array.

```
public TreeNode buildTreePostIn(int[] inorder, int[] postorder) {
    if (inorder == null || postorder == null || inorder.length != postorder.length
)
        return null;
    HashMap<Integer, Integer> hm = new HashMap<Integer,Integer>();
    for (int i=0;i<inorder.length;++i)
        hm.put(inorder[i], i);
    return buildTreePostIn(inorder, 0, inorder.length-1, postorder, 0,
                           postorder.length-1,hm);
}

private TreeNode buildTreePostIn(int[] inorder, int is, int ie, int[] postorder,
int ps, int pe,
                                 HashMap<Integer,Integer> hm){
    if (ps>pe || is>ie) return null;
    TreeNode root = new TreeNode(postorder[pe]);
    int ri = hm.get(postorder[pe]);
    TreeNode leftchild = buildTreePostIn(inorder, is, ri-1, postorder, ps, ps+ri-
is-1, hm);
    TreeNode rightchild = buildTreePostIn(inorder,ri+1, ie, postorder, ps+ri-is,
pe-1, hm);
    root.left = leftchild;
    root.right = rightchild;
    return root;
}
```

written by lurklurk original link here

Solution 3

```
class Solution {
public:
    TreeNode *buildTree(vector<int> &inorder, vector<int> &postorder) {
        if(inorder.size() == 0)return NULL;
        TreeNode* p;
        TreeNode* root;
        vector<int> vint;
        vector<TreeNode*> vtn;
        root = new TreeNode(postorder.back());
        vtn.push_back(root);
        postorder.pop_back();
        while(true)
        {
            if(inorder.back() == vtn.back()->val)
            {
                p = vtn.back();
                vtn.pop_back();
                inorder.pop_back();
                if(inorder.size() == 0) break;
                if(vtn.size())
                    if(inorder.back() == vtn.back()->val)continue;
                p->left = new TreeNode(postorder.back());
                postorder.pop_back();
                vtn.push_back(p->left);
            }
            else
            {
                p = new TreeNode(postorder.back());
                postorder.pop_back();
                vtn.back()->right = p;
                vtn.push_back(p);
            }
        }
        return root;
    }
};
```

written by hongzhi original link here