二叉树的序列化问题

# 105. Construct Binary Tree from Preorder and Inorder Traversal

根据**先序和中序遍历**重构二叉树。

## 算法：Simple Java Solution

class Solution {

int preIdx;

int inIdx;

public TreeNode buildTree(int[] preorder, int[] inorder) {

preIdx = 0;

inIdx = 0;

return helper(preorder, inorder, null);

}

private TreeNode helper(int[] preorder, int[] inorder, **TreeNode end**) {

if (inIdx > inorder.length-1 || (end != null && end.val == inorder[inIdx]))

return null;

TreeNode root = new TreeNode(preorder[preIdx++]);

root.left = helper(preorder, inorder, root);

inIdx++;

root.right = helper(preorder, inorder, end);

return root;

}

}

## 算法1：

思路：**迭代算法实现**。先序遍历的特点：肯定以**根节点root**开头；中序遍历的特点：根节点root出现之前的都是左节点。因此：先根据**先序遍历**获取根节点的值root.val，根据该值从中序遍历中找出该根节点下对应的所有的左节点的个数(也就是该值出现之前的元素个数)。重构二叉树的过程也是不断地找出当前节点的左右节点的过程。寻找左节点的方法：从preorder数组中**索引**不断加1，就是当前节点的左节点(迭代实现)。

注意：类似二分法一样，不断的分割成两段，其根节点的左子树和右子树，再次对左子树和右子树进行**迭代实现**。

Hi guys, this is my Java solution. I read this post, which is very helpful.

The basic idea is here:

Say we have 2 arrays, PRE and IN.

Preorder traversing implies that PRE[0] is the root node.

Then we can find this PRE[0] in IN, say it's IN[5].

Now we know that IN[5] is root, so we know that IN[0] - IN[4] is on the left side, IN[6] to the end is on the right side.

Recursively doing this on subarrays, we can build a tree out of it :)

Hope this helps.

public TreeNode buildTree(int[] preorder, int[] inorder) {

return helper(0, 0, inorder.length - 1, preorder, inorder);

}

public TreeNode **helper**(int preStart, int inStart, int inEnd, int[] preorder, int[] inorder) {

*if (preStart > preorder.length - 1 || inStart > inEnd) {//终止条件的寻找很关键*

*return null;*

*}*

**TreeNode root = new TreeNode(preorder[preStart]);**

**//下面的for循环目的是找出当前根节点下对应的所有的子节点。**

int inIndex = 0; // Index of current root in inorder

for (int i = inStart; i <= inEnd; i++) {

if (inorder[i] == root.val) {

inIndex = i;

}

}

**root.left = helper(preStart + 1, inStart, inIndex - 1, preorder, inorder);**

**root.right = helper(preStart + inIndex - inStart + 1, inIndex + 1, inEnd, preorder, inorder);**

return root;

}

## 算法2：

My Java Solution With Hashmap to **avoid** inorder iteration in every recursive call

class Solution {

int pStart = 0;

public TreeNode buildTree(int[] preorder, int[] inorder) {

if(preorder == null ||inorder == null || preorder.length ==0 || inorder.length==0)

return null;

Map<Integer, Integer> indexes = new HashMap();

for(int i =0; i<inorder.length; i++){

indexes.put(inorder[i],i);

}

return helper(preorder,inorder,0, preorder.length-1, indexes);

}

private TreeNode helper(int[] preorder, int[] inorder, int pstart,int pend, Map<Integer, Integer> indexes ){

if(pstart>pend || pStart>= preorder.length)

return null;

int idx = indexes.get(preorder[pStart]);

TreeNode root = new TreeNode(preorder[pStart]);

pStart++;

root.left = helper(preorder, inorder, pstart, idx-1, indexes);

root.right = helper(preorder, inorder, idx+1, pend, indexes);

return root;

}

}

## 算法3：好理解些。建议这种算法。

思路：利用**Map集合**保存inorder的值及其索引。所有节点的值都是一个一个从preorder中获取出来，而inorder主要是用来获取左子树的个数。

class Solution {

// Index of preOrder array

private int preIndex;

Map<Integer, Integer> map;

public TreeNode buildTree(int[] preorder, int[] inorder) {

if (preorder == null || inorder == null) return null;

int len = inorder.length;

if (len == 0) return null;

preIndex = 0;

//存放inorder值及其对应的索引，目的是为了解决算法1的每次循环搜索。

*map = new HashMap<>();*

*for (int i = 0; i < len; i++) {*

*map.put(inorder[i], i);*

*}*

return helper(preorder, inorder, 0, len-1);

}

private TreeNode helper(**int[] preorder, int[] inorder, int start, int end**) {

if (**preIndex** >= preorder.length || start > end) return null;

int inIndex = map.get(preorder[preIndex]);//获取当前根节点在inorder中的索引

TreeNode node = new TreeNode(preorder[preIndex]);

preIndex++;

node.left = helper(preorder, inorder, **start, inIndex-1**);

node.right = helper(preorder, inorder, **inIndex+1, end**);

return node;

}

}

# 106. Construct Binary Tree from Inorder and Postorder Traversal

根据**中序和后序遍历**重构二叉树。

## 算法1：

class Solution {

int inIdx;

int postIdx;

public TreeNode buildTree(int[] inorder, int[] postorder) {

inIdx = inorder.length-1;

postIdx = postorder.length-1;

return helper(inorder, postorder, null);

}

private TreeNode helper(int[] inorder, int[] postorder, TreeNode end) {

if (inIdx < 0 || (end != null && inorder[inIdx] == end.val))

return null;

TreeNode root = new TreeNode(postorder[postIdx--]);

**root.right = helper(inorder, postorder, root);**

inIdx--;

**root.left = helper(inorder, postorder, end);**

return root;

}

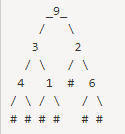
}

# 331. Verify Preorder Serialization of a Binary Tree

验证：二叉树的先序序列化是否正确。

<https://leetcode.com/problems/verify-preorder-serialization-of-a-binary-tree/description/>

One way to serialize a binary tree is to use pre-order traversal. When we encounter a non-null node, we record the node's value. If it is a null node, we record using a sentinel value such as #.



For example, the above binary tree can be serialized to the string "9,3,4,#,#,1,#,#,2,#,6,#,#", where **#** represents a null node.

Given a string of **comma** separated values, verify whether it is a correct preorder traversal serialization of a binary tree. Find an algorithm without reconstructing the tree.

Each comma separated value in the string must be either an integer or a character '#' representing null pointer.

You may assume that the input format is always valid, for example it could never contain two consecutive **commas** such as "**1,,3**".

Example 1:

Input: "9,3,4,#,#,1,#,#,2,#,6,#,#"

Output: true

Example 2:

Input: "1,#"

Output: false

Example 3:

Input: "9,#,#,1"

Output: false

## 算法1：

Some used stack. Some used the depth of a stack. Here I use a different perspective. In a binary tree, if we consider **null** as **leaves**, then

* all non-null node provides 2 outdegree and 1 indegree (2 children and 1 parent), except root
* all null node provides 0 outdegree and 1 indegree (0 child and 1 parent).

Suppose we try to build this tree. During building, we record the difference between out degree and in degree **diff = outdegree - indegree**. When the next node comes, we then decrease diff by 1, because the node provides an in degree. If the node is not null, we increase diff by 2, because it provides two out degrees. ***If a serialization is correct, diff should never be negative and diff will be zero when finished***.

public boolean isValidSerialization(String preorder) {

String[] nodes = preorder.split(",");

int diff = 1;

**for (String node: nodes) {**

**if (--diff < 0) return false;**

**if (!node.equals("#")) diff += 2;**

**}**

return diff == 0;

}

## 算法2：

思路：这种算法与上述算法1有类似之处。

class Solution {

public boolean isValidSerialization(String preorder) {

char[] pOrder = preorder.toCharArray();

int count=1;

for(int i=0; i<pOrder.length; i++){

if(pOrder[i]=='#') count--;

if(count==0) return i==pOrder.length-1;

if(pOrder[i]==',' && pOrder[i-1]!='#')

count++;

}

return false;

}

}

## 算法3：

Just share my solution here. We can just parse the string and count the child a node ought to have. With that count we can decide if input is a valid tree. We are verifying the tree exactly following pre-order traversal. So it's easier for me to understand. Also suppose given input is a post-order tree, we can simply adapt this code to post-order traversal.

public boolean isValidSerialization(String preorder) {

String[] tree = preorder.split(",");

int nodeCount = verify(tree, 0);

return nodeCount == tree.length;

}

// return node count of current subtree

private int verify(String[] str, int start) {

if (start >= str.length) return -1; // return -1 when there is no valid answer

if ("#".equals(str[start])) return 1;

// current node has valid value

int leftLen = verify(str, start + 1);

if (leftLen == -1) return -1;

int rightLen = verify(str, start + 1 + leftLen);

if (rightLen == -1) return -1;

return 1 + leftLen + rightLen;

}