二叉树的序列化与反序列化

# 对称的二叉树Symmetrical

## 题目要求

请实现一个函数，用来判断一颗二叉树是不是对称的。注意，如果一个二叉树同此二叉树的镜像是同样的，定义其为对称的。

## 思路分析：

通过遍历与对称遍历就可以判断二叉树是否是对称二叉树。但是存在特殊情况：

当二叉树的所有节点的值都相等的时候，无论怎么遍历结果都一样。

**解决办法**：将**空节点**打印出来，就可以唯一确定了。

**前序遍历**：中左右 7,7,7,$,$,7,$,$,7,7,$,$,$

**中右左** 7,7,$,7,$,$,7,7,$,$,7,$,$

对于空节点进行判断就可以唯一确定二叉树的结构。

**当然，真正代码实现的时候，并不需要遍历整个二叉树，而是按照遍历的顺序逐个比较，遇到不对称的节点立即返回false即可，若对称，再继续递归下去。**



**补充说明**：对于中序遍历和后序遍历也可以用来判断是否对称，但是由于它们对中间节点值的判断要放在左节点或右节点之后，需要递归至最左节点，因此不如前序遍历直接判断中间节点效率高，因此以后统一使用前序遍历和前序遍历的对称遍历来判断二叉树的对称性。

## Java代码：

public boolean isSymmetrical(TreeNode root) {

return isSymmetricalHandler(root,root);

}

/\*\*

\* 递归方法

\* 对于root1：中左右；

\* 对于root2：中右左；

\* 每一次判断，对应的两个节点都是对称位置的。

\*/

public boolean isSymmetricalHandler(TreeNode root1,TreeNode root2){

if(root1 == null && root2 == null) return true;

if(root1 == null || root2 == null) return false;

if(root1.val != root2.val) {

return false;

}else{//只有在当前节点与对称节点一致时，才递归下一节点

return isSymmetricalHandler(root1.left,root2.right)&&isSymmetricalHandler(root1.right,root2.left);

}

}

# 二叉树的镜像

## 题目介绍

操作给定的二叉树，将其变换为源二叉树的镜像。

输入描述:

二叉树的镜像定义：

源二叉树

8

/ \

6 10

/ \ / \

5 7 9 11

镜像二叉树

8

/ \

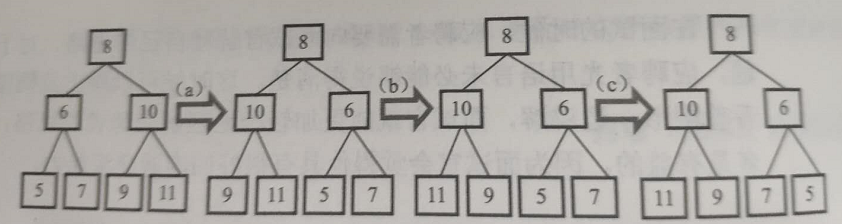
10 6

/ \ / \

11 9 7 5

## 思路分析

不断交换节点的左右节点即可。



## Java代码

/\*\*

\* 获取二叉树镜像

\* 算法：按照前序遍历顺序逐渐交换节点的左右节点

\*/

public void Mirror(TreeNode root) {

if(root == null) return;//递归终止条件

//交换当前节点的左右节点

**TreeNode temp = root.left;**

**root.left = root.right;**

**root.right = temp;**

Mirror(root.left);

Mirror(root.right);

}

# 二叉树的序列化与反序列化—剑指Offer

## 题目要求：

请实现两个函数，分别用来序列化和反序列化二叉树。

<https://www.nowcoder.com/practice/cf7e25aa97c04cc1a68c8f040e71fb84?tpId=13&tqId=11214&tPage=1&rp=1&ru=/ta/coding-interviews&qru=/ta/coding-interviews/question-ranking>

## 分析思路

采用前序遍历的方式序列化和反序列化。

### 序列化—比较简单

**递归实现**与**while循环和栈实现**；

### 反序列化

目前只会递归实现，非递归暂时不能实现。

## Java代码

### 序列化

* 递归实现

/\*\*

\* 序列化递归实现

\*/

String Serialize(TreeNode root) {//前序遍历

StringBuilder sb = new StringBuilder();

helper(root,sb);

return sb.toString().substring(0,sb.length()-1);

}

public void helper(TreeNode root,StringBuilder sb){

if(root == null) {//递归终止条件

sb.append("$").append(",");

return;

}

**//前序遍历递归实现**

**sb.append(root.val).append(",");**

**helper(root.left,sb);**

**helper(root.right,sb);**

}

非递归实现

/\*\*

\* 栈和while循环实现

\*/

String Serialize2(TreeNode root) {//前序遍历

if(root == null) return "";

StringBuilder sb = new StringBuilder();

Deque<TreeNode> stack = new ArrayDeque<TreeNode>();

while(!stack.isEmpty()||root != null){

if(root != null){

stack.push(root);

**sb.append(root.val).append(",");**

root = root.left;

}else{

**sb.append("$").append(",");**

root = stack.pop().right;

}

}

**sb.append("$").append(",");**

return sb.toString().substring(0,sb.length()-1);

}

### 反序列化—递归实现

调用一次deHelper方法，内部会有两次递归调用自身，若连续两个$，正好直接返回回来，如果只有1个$，则只会返回对当前节点的右子树继续生成；如果没有$，则会一直递归进去，直到遇到$逐渐返回。

TreeNode **Deserialize**(String str) {

if(str == null||str.length() == 0) return null;

String[] strs = str.split(",");

int[] index = new int[]{-1};

return deHelper(strs,index);

}

TreeNode **deHelper**(String[] strs,int[] index){

++index[0];

// if( index[0] >= strs.length) return null;//没有必要判断，递归的次数完全可以由$的出现返回

if(**!"$".equals(strs[index[0]])**){ //不要使用!= 判断字符串相等或不相等

/\*将整体看做三部分：根节点、左子树和右子树 \*/

**TreeNode root = new TreeNode(Integer.valueOf(strs[index[0]]));**

**root.left = deHelper(strs,index);**

**root.right = deHelper(strs,index) ;**

return root;

}

return null;

}

二叉树的数据结构：

public class TreeNode {

int val = 0;

TreeNode left = null;

TreeNode right = null;

public TreeNode(int val) {

this.val = val;

}

}