原创 数据结构 - 二叉树 - 面试中常见的二叉树算法题

2018-01-17 21:16:09 从零开始的异世界生活 阅读数 17127 ☆ 收藏 更多

数据结构 - 二叉树 - 面试中常见的二叉树算法题

数据结构是面试中必定考查的知识点，面试者需要掌握几种经典的数据结构：**线性表**（数组、链表）、**栈与队列**、**树**（二叉树、二叉查找树、平衡二叉树）、**图**。

本文主要介绍**树**中的常见的**二叉树**数据结构。包括

- 概念简介
- 二叉树中树节点的数据结构（Java）
- 二叉树的遍历（Java）
- 常见的二叉树算法题（Java）

概念简介

如果对二叉树概念已经基本掌握，可以跳过该部分，直接查看常见链表算法题。

二叉树基本概念

二叉树在图论中是这样定义的：二叉树是一个连通的无环图，并且每一个顶点的度不大于3。有根二叉树还要满足根结点的度不大于2。有了根结点之后定义了唯一的父结点，和最多2个子结点。二叉树性质如下：

- 二叉树的每个结点至多只有二棵子树(不存在度大于2的结点)，二叉树的子树有左右之分，次序不能颠倒。
- 二叉树的第 i 层至多有 2^{i-1} 个结点。
- 深度为 k 的二叉树至多有 $2^k - 1$ 个结点。
- 对任何一棵二叉树T，如果其终端结点数为n_0，度为2的结点数为n_2，则$n_0 = n_2 + 1$。
- 一棵深度为k，且有 $2^k - 1$ 个节点称之为**满二叉树**；
- 深度为k，有n个节点的二叉树，当且仅当其每一个节点都与深度为k的满二叉树中，序号为1至n的节点对应时，称之为**完全二叉树。**
- **平衡二叉树**又被称为AVL树（区别于AVL算法），它是一棵二叉排序树，且具有以下性质：它是一棵空树或它的左右两个子树的高度差的绝对值不超过1，并且左右两个子树都是一棵平衡二叉树。

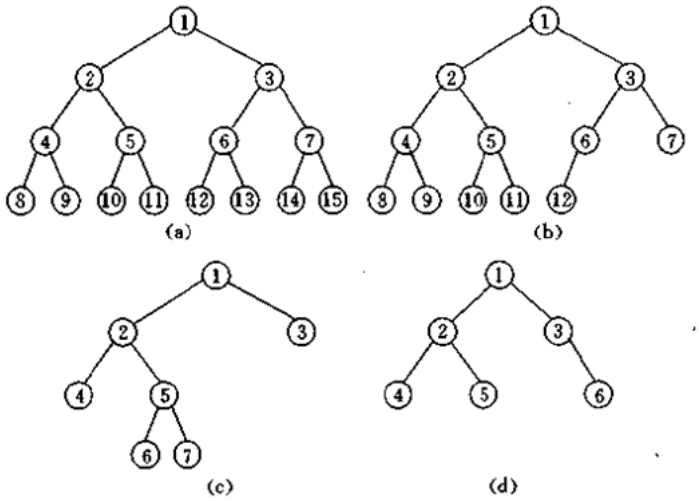

图 6.4 特殊形态的二叉树

(a) 满二叉树；(b) 完全二叉树；(c)和(d)非完全二叉树。

二叉树中树节点的数据结构

二叉树由一系列树结点组成，每个结点包括三个部分：一个是存储**数据元素**的数据域，另一个是存储**左子结点**地址的指针域，另一个是存储**右子结点**地

定义树节点为类：TreeNode。具体实现如下：

```
public class TreeNode {

    public int val; //数据域
    public TreeNode left; //左子树根节点
    public TreeNode right; //右子树根节点

    public TreeNode() {

    }

    public TreeNode(int val) {
        this.val = val;
    }

}
```

二叉树的遍历

1. 前序遍历

递归解法:

- 如果二叉树为空，空操作
- 如果二叉树不为空，访问根节点，前序遍历左子树，前序遍历右子树

```
/**
 * 1. 前序遍历
 * 递归
 * @param root 树根节点
 */
public static void preorderTraversalRec(TreeNode root){
    if (root == null) {
        return;
    }
    System.out.print(root.val + "->");
```

```
    preorderTraversalRec(root.left);
    preorderTraversalRec(root.right);
}
```

非递归解法：用一个辅助stack，总是先把右孩子放进栈。

```
/**
 * 1. 前序遍历
 * 非递归
 * @param root 树根节点
 */
public static void preorderTraversal2(TreeNode root) {
    if (root == null) {
        return;
    }
    Stack<TreeNode> stack = new Stack<>(); //辅助栈
    TreeNode cur = root;
    while (cur != null || !stack.isEmpty()) {
        while (cur != null) { //不断将左子节点入栈，直到cur为空
            stack.push(cur);
            System.out.print(cur.val + "->"); //前序遍历，先打印当前节点在打印左子节点，然后再把右子节点加到栈中
            cur = cur.left;
        }
        if (!stack.isEmpty()) { //栈不为空，弹出栈元素
            cur = stack.pop(); //此时弹出最左边的节点
            cur = cur.right; //令当前节点为右子节点
        }
    }
}

/**
 * 1. 前序遍历
 * 非递归解法2
 * @param root 树根节点
 */
public static void preorderTraversal(TreeNode root) {
    if (root == null) {
        return;
    }
    Stack<TreeNode> stack = new Stack<>(); //辅助栈保存树节点
    stack.add(root);
    while (!stack.isEmpty()) { //栈不为空
        TreeNode temp = stack.pop();
        System.out.print(temp.val + "->"); //先根节点，因为是前序遍历
        if (temp.right != null) { //先添加右孩子，因为栈是先进后出
            stack.add(temp.right);
        }
        if (temp.left != null) {
            stack.add(temp.left);
        }
    }
}
```

2. 中序遍历

递归解法：

- 如果二叉树为空，空操作
- 如果二叉树不为空，中序遍历左子树，访问根节点，中序遍历右子树

```
/**
 * 2. 中序遍历
 * 递归
 * @param root 树根节点
 */
public static void inorderTraversalRec(TreeNode root){
    if (root == null) {

```

```
            return;
        }
        inorderTraversalRec(root.left);
        System.out.print(root.val + "->");
        inorderTraversalRec(root.right);
    }
```

非递归解法：用栈先把根节点的所有左孩子都添加到栈内，然后输出栈顶元素，再处理栈顶元素的右子树。

```
/**
 * 2. 中序遍历
 * 非递归
 * @param root 树根节点
 */
public static void inorderTraversal(TreeNode root) {
    if (root == null) {
        return;
    }
    Stack<TreeNode> stack = new Stack<>(); //辅助栈
    TreeNode cur = root;
    while (cur != null || !stack.isEmpty()) {
        while (cur != null) { //不断将左子节点入栈，直到cur为空
            stack.push(cur);
            cur = cur.left;
        }
        if (!stack.isEmpty()) { //栈不为空，弹出栈元素
            cur = stack.pop(); //此时弹出最左边的节点
            System.out.print(cur.val + "->"); //中序遍历，先打印左子节点在打印当前节点，然后再把右子节点加到栈中
            cur = cur.right; //令当前节点为右子节点
        }
    }
}
```

3. 后序遍历

递归解法：

- 如果二叉树为空，空操作
- 如果二叉树不为空，后序遍历左子树，后序遍历右子树，访问根节点

```
/**
 * 3. 后序遍历
 * 递归
 * @param root 树根节点
 */
public static void postorderTraversalRec(TreeNode root){
    if (root == null) {
        return;
    }
    postorderTraversalRec(root.left);
    postorderTraversalRec(root.right);
    System.out.print(root.val + "->");
}
```

非递归解法：双栈法。

```
/**
 * 3. 后序遍历
 * 非递归
 * @param root 树根节点
 */
public static void postorderTraversal(TreeNode root) {
    if(root == null) {
        return;
    }
    Stack<TreeNode> stack1 = new Stack<>(); //保存树节点
    Stack<TreeNode> stack2 = new Stack<>(); //保存后序遍历的结果
```

```
    stack1.add(root);
    while (!stack1.isEmpty()) {
        TreeNode temp = stack1.pop();
        stack2.push(temp); //将弹出的元素加到stack2中
        if (temp.left != null) { //左子节点先入栈
            stack1.push(temp.left);
        }
        if (temp.right != null) { //右子节点后入栈
            stack1.push(temp.right);
        }
    }
    while (!stack2.isEmpty()) {
        System.out.print(stack2.pop().val + "->");
    }
}
```

4. 层次遍历

思路：分层遍历二叉树（按层次从上到下，从左到右）迭代，相当于广度优先搜索，使用队列实现。队列初始化，将根节点压入队列。当 为空，作：弹出一个节点，访问，若左子节点或右子节点不为空，将其压入队列。

```
/**
 * 4. 层次遍历
 * @param root 根节点
 */
public static void levelTraversal(TreeNode root){
    if(root == null) {
        return;
    }
    Queue<TreeNode> queue = new LinkedList<>(); //对列保存树节点
    queue.add(root);
    while (!queue.isEmpty()) {
        TreeNode temp = queue.poll();
        System.out.print(temp.val + "->");
        if (temp.left != null) { //添加左右子节点到对列
            queue.add(temp.left);
        }
        if (temp.right != null) {
            queue.add(temp.right);
        }
    }
}
```

常见的二叉树算法题

1. 求二叉树中的节点个数

递归解法：$O(n)$

- 如果二叉树为空，节点个数为0
- 如果二叉树不为空，二叉树节点个数 = 左子树节点个数 + 右子树节点个数 + 1

```
/**
 * 1. 求二叉树中的节点个数
 * 递归
 * @param root 树根节点
 * @return 节点个数
 */
public static int getNodeNumRec(TreeNode root) {
    if (root == null) {
        return 0;
    }
    return getNodeNumRec(root.left) + getNodeNumRec(root.right) + 1;
}
```

非递归解法：$O(n)$。基本思想同LevelOrderTraversal。即用一个Queue，在Java里面可以用LinkedList来模拟。

```java
/**
 * 1. 求二叉树中的节点个数
 * 非递归
 * @param root 树根节点
 * @return 节点个数
 */
public static int getNodeNum(TreeNode root) {
    if (root == null) {
        return 0;
    }
    Queue<TreeNode> queue =  new LinkedList<>(); // 用队列保存树节点，先进先出
    queue.add(root);
    int count = 1; // 节点数量
    while (!queue.isEmpty()) {
        TreeNode temp = queue.poll(); // 每次从对列中删除节点，并返回该节点信息
        if (temp.left != null) { // 添加左子孩子到对列
            queue.add(temp.left);
            count++;
        }
        if (temp.right != null) { // 添加右子孩子到对列
            queue.add(temp.right);
            count++;
        }
    }
    return count;
}
```

2. 求二叉树的深度（高度）

递归解法：$O(n)$

- 如果二叉树为空，二叉树的深度为0
- 如果二叉树不为空，二叉树的深度 = max(左子树深度， 右子树深度) + 1

```java
/**
* 求二叉树的深度（高度）
* 递归
* @return 树的深度
*/
public static int getDepthRec(TreeNode root) {
   if (root == null) {
       return 0;
   }
   return Math.max(getDepthRec(root.left), getDepthRec(root.right)) + 1;
}
```

非递归解法：$O(n)$。基本思想同LevelOrderTraversal。即用一个Queue，在Java里面可以用LinkedList来模拟。

```java
/**
 * 求二叉树的深度（高度）
 * 非递归
 * @param root 树根节点
 * @return 树的深度
 */
public static int getDepth(TreeNode root) {
    if (root == null) {
        return 0;
    }
    int currentLevelCount = 1; // 当前层的节点数量
    int nextLevelCount = 0; // 下一层节点数量
    int depth = 0; // 树的深度

    Queue<TreeNode> queue = new LinkedList<>(); // 对列保存树节点
    queue.add(root);
    while (!queue.isEmpty()) {
        TreeNode temp = queue.remove(); // 移除节点
        currentLevelCount--; // 当前层节点数减1
```

```
        if (temp.left != null) { // 添加左节点并更新下一层节点个数
            queue.add(temp.left);
            nextLevelCount++;
        }
        if (temp.right != null) { // 添加右节点并更新下一层节点个数
            queue.add(temp.right);
            nextLevelCount++;
        }
        if (currentLevelCount == 0) { // 如果是该层的最后一个节点，树的深度加1
            depth++;
            currentLevelCount = nextLevelCount; // 更新当前层节点数量并且重置下一层节点数量
            nextLevelCount = 0;
        }
    }
    return depth;
}
```

3. 求二叉树第k层的节点个数

递归解法：$O(n)$

思路：求以root为根的k层节点数目，等价于求以root左孩子为根的k-1层（因为少了root）节点数目 加上以root右孩子为根的k-1层（因为少了root）节点数目 即：

- 如果二叉树为空或者k<1，返回0
- 如果二叉树不为空并且k==1，返回1
- 如果二叉树不为空且k>1，返回root左子树中k-1层的节点个数与root右子树k-1层节点个数之和

```
/**
 * 求二叉树第k层的节点个数
 * 递归
 * @param root 根节点
 * @param k 第k个节点
 * @return 第k层节点数
 */
public static int getNodeNumKthLevelRec(TreeNode root, int k) {
    if (root == null || k < 1) {
        return 0;
    }
    if (k == 1) {
        return 1;
    }
    return getNodeNumKthLevelRec(root.left, k - 1) + getNodeNumKthLevelRec(root.right, k - 1);
}
```

4. 求二叉树中叶子节点的个数

递归解法：

- 如果二叉树为空，返回0
- 如果二叉树是叶子节点，返回1
- 如果二叉树不是叶子节点，二叉树的叶子节点数 = 左子树叶子节点数 + 右子树叶子节点数

```
/**
 * 4. 求二叉树中叶子节点的个数
 * 递归
 * @param root 根节点
 * @return 叶子节点个数
 */
public static int getNodeNumLeafRec(TreeNode root) {
    if (root == null) {
        return 0;
    }
    if (root.left == null && root.right == null) {
        return 1;
    }
```

```
    return getNodeNumLeafRec(root.left) + getNodeNumLeafRec(root.right);
}
```

非递归解法：基于层次遍历进行求解，利用Queue进行。

```
/**
 * 4. 求二叉树中叶子节点的个数（迭代）
 * 非递归
 * @param root 根节点
 * @return 叶子节点个数
 */
public static int getNodeNumLeaf(TreeNode root){
    if (root == null) {
        return 0;
    }
    int leaf = 0; //叶子节点个数
    Queue<TreeNode> queue = new LinkedList<>();
    queue.add(root);
    while (!queue.isEmpty()) {
        TreeNode temp = queue.poll();
        if (temp.left == null && temp.right == null) { //叶子节点
            leaf++;
        }
        if (temp.left != null) {
            queue.add(temp.left);
        }
        if (temp.right != null) {
            queue.add(temp.right);
        }
    }
    return leaf;
}
```

5. 判断两棵二叉树是否相同的树

递归解法：

- 如果两棵二叉树都为空，返回真
- 如果两棵二叉树一棵为空，另外一棵不为空，返回假
- 如果两棵二叉树都不为空，如果对应的左子树和右子树都同构返回真，其他返回假

```
/**
 * 5. 判断两棵二叉树是否相同的树。
 * 递归
 * @param r1 二叉树1
 * @param r2 二叉树2
 * @return 是否相同
 */
public static boolean isSameRec(TreeNode r1, TreeNode r2) {
    if (r1 == null && r2 == null) { //都是空
        return true;
    } else if (r1 == null || r2 == null) { //有一个为空，一个不为空
        return false;
    }
    if (r1.val != r2.val) { //两个不为空，但是值不相同
        return false;
    }
    return isSameRec(r1.left, r2.left) && isSameRec(r1.right, r2.right); //递归遍历左右子节点
}
```

非递归解法：利用Stack对两棵树对应位置上的节点进行判断是否相同。

```
/**
 * 5. 判断两棵二叉树是否相同的树（迭代）
 * 非递归
 * @param r1 二叉树1
```

```
 * @param r2 二叉树2
 * @return 是否相同
 */
public static boolean isSame(TreeNode r1, TreeNode r2){
    if (r1 == null && r2 == null) { //都是空
        return true;
    } else if (r1 == null || r2 == null) { //有一个为空，一个不为空
        return false;
    }
    Stack<TreeNode> stack1 = new Stack<>();
    Stack<TreeNode> stack2 = new Stack<>();
    stack1.add(r1);
    stack2.add(r2);
    while (!stack1.isEmpty() && !stack2.isEmpty()) {
        TreeNode temp1 = stack1.pop();
        TreeNode temp2 = stack2.pop();
        if (temp1 == null && temp2 == null) { //两个元素都为空，因为添加的时候没有对空节点做判断
            continue;
        } else if (temp1 != null && temp2 != null && temp1.val == temp2.val) {
            stack1.push(temp1.left); //相等则添加左右子节点判断
            stack1.push(temp1.right);
            stack2.push(temp2.left);
            stack2.push(temp2.right);
        } else {
            return false;
        }
    }
    return true;
}
```

6. 判断二叉树是不是平衡二叉树

递归实现：借助前面实现好的求二叉树高度的函数

- 如果二叉树为空， 返回真
- 如果二叉树不为空，如果左子树和右子树都是AVL树并且左子树和右子树高度相差不大于1，返回真，其他返回假

```
/**
 * 6. 判断二叉树是不是平衡二叉树
 * 递归
 * @param root 根节点
 * @return 是否二叉平衡树（AVL树）
 */
public static boolean isAVLTree(TreeNode root) {
    if (root == null) {
        return true;
    }
    if (Math.abs(getDepth(root.left) - getDepth(root.right)) > 1) { //左右子树高度差大于1
        return false;
    }
    return isAVLTree(root.left) && isAVLTree(root.right); //递归判断左右子树
}
```

7. 求二叉树的镜像

递归实现：破坏原来的树，把原来的树改成其镜像

- 如果二叉树为空，返回空
- 如果二叉树不为空，求左子树和右子树的镜像，然后交换左右子树

```
/**
 * 7. 求二叉树的镜像
 * 递归
 * @param root 根节点
 * @return 镜像二叉树的根节点
 */

```

```
public static TreeNode mirrorRec(TreeNode root) {
    if (root == null) {
        return root;
    }
    TreeNode left = mirrorRec(root.right); //递归镜像左右子树
    TreeNode right = mirrorRec(root.left);
    root.left = left; //更新根节点的左右子树为镜像后的树
    root.right = right;
    return root;
}
```

递归实现：不能破坏原来的树，返回一个新的镜像树

- 如果二叉树为空，返回空
- 如果二叉树不为空，求左子树和右子树的镜像，然后交换左右子树

```
/**
 * 7. 求二叉树的镜像
 * 递归
 * @param root 根节点
 * @return 镜像二叉树的根节点
 */
public static TreeNode mirrorCopyRec(TreeNode root) {
    if (root == null) {
        return root;
    }
    TreeNode newRoot = new TreeNode(root.val); //创建新节点，然后交换左右子树
    newRoot.left = mirrorCopyRec(root.right);
    newRoot.right = mirrorCopyRec(root.left);
    return newRoot;
}
```

非递归实现：破坏原来的树，把原来的树改成其镜像

```
/**
 * 7. 求二叉树的镜像
 * 非递归
 * @param root 根节点
 * @return 镜像二叉树的根节点
 */
public static void mirror(TreeNode root) {
    if (root == null) {
        return ;
    }
    Stack<TreeNode> stack = new Stack<>();
    stack.push(root);
    while (!stack.isEmpty()){
        TreeNode cur = stack.pop();
        //交换左右孩子
        TreeNode tmp = cur.right;
        cur.right = cur.left;
        cur.left = tmp;

        if(cur.right != null) {
            stack.push(cur.right);
        }
        if (cur.left != null) {
            stack.push(cur.left);
        }

    }
}
```

非递归实现：不能破坏原来的树，返回一个新的镜像树

```
/**
 * 7. 求二叉树的镜像
```

```
 * 非递归
 * @param root 根节点
 * @return 镜像二叉树的根节点
 */
public static TreeNode mirrorCopy(TreeNode root) {
    if (root == null) {
        return null;
    }
    Stack<TreeNode> stack = new Stack<TreeNode>();
    Stack<TreeNode> newStack = new Stack<TreeNode>();
    stack.push(root);
    TreeNode newRoot = new TreeNode(root.val);
    newStack.push(newRoot);
    while (!stack.isEmpty()) {
        TreeNode cur = stack.pop();
        TreeNode newCur = newStack.pop();
        if (cur.right != null) {
            stack.push(cur.right);
            newCur.left = new TreeNode(cur.right.val);
            newStack.push(newCur.left);
        }
        if (cur.left != null) {
            stack.push(cur.left);
            newCur.right = new TreeNode(cur.left.val);
            newStack.push(newCur.right);
        }
    }
    return newRoot;
}
```

8. 判断两个二叉树是否互相镜像

递归解法：与比较两棵二叉树是否相同解法一致（题5），非递归解法省略。

- 比较r1的左子树的镜像是不是r2的右子树
- 比较r1的右子树的镜像是不是r2的左子树

```
/**
 * 8. 判断两个树是否互相镜像
 * @param r1 二叉树 1
 * @param r2 二叉树 2
 * @return 是否互相镜像
 */
public static boolean isMirrorRec(TreeNode r1, TreeNode r2) {
    if (r1 == null && r2 == null) {
        return true;
    } else if (r1 == null || r2 == null) {
        return false;
    }
    if (r1.val != r2.val) {
        return false;
    }
    // 递归比较r1的左子树的镜像是不是r2右子树
    // 和r1的右子树的镜像是不是r2的左子树
    return isMirrorRec(r1.left, r2.right) && isMirrorRec(r1.right, r2.left);
}
```

9. 求二叉树中两个节点的最低公共祖先节点

递归解法：

- 如果两个节点分别在根节点的左子树和右子树，则返回根节点
- 如果两个节点都在左子树，则递归处理左子树；如果两个节点都在右子树，则递归处理右子树

```
/**
 * 9. 求二叉树中两个节点的最低公共祖先节点

```

```
 * 递归
 * @param root 树根节点
 * @param n1 第一个节点
 * @param n2 第二个节点
 * @return 最低公共祖先节点
 */
public static TreeNode getLastCommonParentRec(TreeNode root, TreeNode n1, TreeNode n2) {
    if (findNodeRec(root.left, n1)) { //如果n1在左子树
        if (findNodeRec(root.right, n2)) { //如果n2在右子树
            return root; //返回根节点
        } else { //如果n2也在左子树
            return getLastCommonParentRec(root.left, n1, n2); //递归处理
        }
    } else { //如果n1在右子树
        if (findNodeRec(root.left, n2)) { //如果n2在左子树
            return root; //返回根节点
        } else { //如果n2在右子树
            return getLastCommonParentRec(root.right, n1, n2); //递归处理
        }
    }
}

/**
 * 递归判断一个点是否在树里
 * @param root 根节点
 * @param node 查找的节点
 * @return 是否找到该节点
 */
private static boolean findNodeRec(TreeNode root, TreeNode node) {
    if (node == null || root == null) {
        return false;
    }
    if (root == node) {
        return true;
    }
    //先尝试在左子树中查找
    boolean found = findNodeRec(root.left, node);
    if (!found) { //如果查找不到，再在右子树中查找
        found = findNodeRec(root.right, node);
    }
    return found;
}

/**
 * 9. 树中两个节点的最低公共祖先节点
 * 递归解法2（更简单）
 * @param root 树根节点
 * @param n1 第一个节点
 * @param n2 第二个节点
 * @return 最低公共祖先节点
 */
public static TreeNode getLastCommonParentRec2(TreeNode root, TreeNode n1, TreeNode n2) {
    if (root == null) {
        return null;
    }
    //如果有一个match，则说明当前node就是要找的最低公共祖先
    if (root.equals(n1) || root.equals(n2)) {
        return root;
    }
    TreeNode commonLeft = getLastCommonParentRec2(root.left, n1, n2);
    TreeNode commonRight = getLastCommonParentRec2(root.right, n1, n2);
    //如果一个在左子树找到，一个在右子树找到，则说明root是唯一可能得最低公共祖先
    if (commonLeft != null && commonRight != null) {
        return root;
    }
    //其他情况是要不然在左子树要不然在右子树
    if (commonLeft != null) {
        return commonLeft;
```

```
        }
        return commonRight;
    }
```

非递归算法：得到从二叉树根节点到两个节点的路径，路径从头开始的最后一个公共节点就是它们的最低公共祖先节点

```
/**
 * 9. 树中两个节点的最低公共祖先节点
 * 非递归
 * @param root 树根节点
 * @param n1 第一个节点
 * @param n2 第二个节点
 * @return 第一个公共祖先节点
 */
public static TreeNode getLastCommonParent(TreeNode root, TreeNode n1, TreeNode n2) {
    if (root == null || n1 == null || n2 == null) {
        return null;
    }
    ArrayList<TreeNode> p1 = new ArrayList<>();
    boolean res1 = getNodePath(root, n1, p1);
    ArrayList<TreeNode> p2 = new ArrayList<>();
    boolean res2 = getNodePath(root, n2, p2);
    if (!res1 || !res2) {
        return null;
    }
    TreeNode last = null;
    Iterator<TreeNode> iter1 = p1.iterator();
    Iterator<TreeNode> iter2 = p2.iterator();
    while (iter1.hasNext() && iter2.hasNext()) {
        TreeNode tmp1 = iter1.next();
        TreeNode tmp2 = iter2.next();
        if (tmp1 == tmp2) {
            last = tmp1;
        } else { //直到遇到非公共节点
            break;
        }
    }
    return last;
}

/**
 * 把从根节点到node路径上所有的点都添加到path中
 * @param root 树根节点
 * @param node 终点节点
 * @param path 路径
 * @return 是否是目标节点
 */
public static boolean getNodePath(TreeNode root, TreeNode node, ArrayList<TreeNode> path) {
    if (root == null) {
        return false;
    }
    path.add(root); //把这个节点添加到路径中
    if (root == node) {
        return true;
    }
    boolean found = false;
    found = getNodePath(root.left, node, path); //先在左子树中找
    if (!found) {
        found = getNodePath(root.right, node, path);
    }
    if (!found) { //如果实在没找到证明这个节点不在路径中，删除刚刚那个节点
        path.remove(root);
    }
    return found;
}
```

10. 判断是否为二分查找树BST

递归解法：中序遍历的结果应该是递增的。

```
/**
 * 10. 判断是否为二分查找树BST
 * @param root 根节点
 * @param pre 上一个保存的节点
 * @return 是否为BST树
 */
public static boolean isValidBST(TreeNode root, int pre){
    if (root == null) {
        return true;
    }
    boolean left = isValidBST(root.left, pre);
    if (!left) {
        return false;
    }
    if(root.val <= pre) {
        return false;
    }
    pre = root.val;
    boolean right = isValidBST(root.right, pre);
    if(!right) {
        return false;
    }
    return true;
}
```

非递归解法：参考非递归中序遍历。

```
/**
 * 10. 判断是否为二分查找树BST
 * 非递归
 * @param root 根节点
 */
public boolean isValidBST2(TreeNode root){
    Stack<TreeNode> stack = new Stack<>();
    //设置前驱节点
    TreeNode pre = null;
    while(root != null || !stack.isEmpty()){
        while (root != null) { // 将当前节点，以及左子树一直入栈，循环结束时，root==null
            stack.push(root);
            root = root.left;
        }
        root = stack.pop();
        //比较并更新前驱，与普通遍历的区别就在下面四行
        if(pre != null && root.val <= pre.val){
            return false;
        }
        pre = root;
        root = root.right;  //访问右子树
    }
    return true;
}
```

文章最后发布于: 2018-01-1