是否与 t2 当前节点的值一样。比如，题目中的例子，t1 中的节点 2 与 t2 中的节点 2 匹配，然后 t1 跟着 t2 向左，发现 t1 中的节点 4 与 t2 中的节点 4 匹配，t1 跟着 t2 继续向左，发现 t1 中的节点 8 与 t2 中的节点 8 匹配，此时 t2 回到 t2 中的节点 2，t1 也回到 t1 中的节点 2，然后 t1 跟着 t2 向右，发现 t1 中的节点 5 与 t2 中的节点 5 匹配。t2 匹配完毕，结果返回 true。如果匹配的过程中发现有不匹配的情况，直接返回 false，说明 t1 的当前子树从头节点开始，无法与 t2 匹配，那么再去寻找 t1 的下一棵子树。t1 的每棵子树上都有可能匹配出 t2，所以都要检查一遍。

所以，如果 t1 的节点数为 N，t2 的节点数为 M，该方法的时间复杂度为 $O(N\times M)$。

具体过程请参看如下代码中的 contains 方法，

```
public class Node {
        public int value;
        public Node left;
        public Node right;

        public Node(int data) {
                this.value = data;
        }
}

public boolean contains(Node t1, Node t2) {
        return check(t1, t2) || contains(t1.left, t2) || contains(t1.right, t2);
}

public boolean check(Node h, Node t2) {
        if (t2 == null) {
                return true;
        }
        if (h == null || h.value != t2.value) {
                return false;
        }
        return check(h.left, t2.left) && check(h.right, t2.right);
}
```

判断 t1 树中是否有与 t2 树拓扑结构完全相同的子树

【题目】

给定彼此独立的两棵树头节点分别为 t1 和 t2，判断 t1 中是否有与 t2 树拓扑结构完全相同的子树。

例如，图 3-36 所示的 t1 树和图 3-37 所示 t2 树。

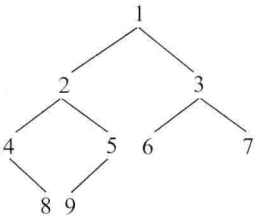

图 3-36

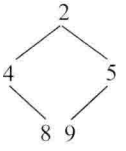

图 3-37

t1 树有与 t2 树拓扑结构完全相同的子树，所以返回 true。但如果 t1 树和 t2 树分别如图 3-38 和图 3-39 所示，则 t1 树就没有与 t2 树拓扑结构完全相同的子树，所以返回 false。

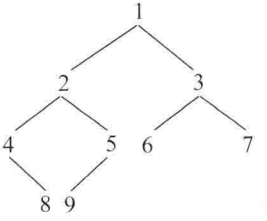

图 3-38

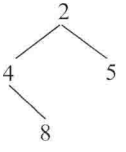

图 3-39

【难度】

校 ★★★☆

【解答】

如果 t1 的节点数为 N，t2 的节点数为 M，本题最优解是时间复杂度为 $O(N+M)$的方法。先简单介绍一个时间复杂度为 $O(N\times M)$的方法，对于 t1 的每棵子树，都去判断是否与 t2 树的拓扑结构完全一样，这个过程的复杂度为 $O(M)$，t1 的子树一共有 N 棵，所以时间复杂度为 $O(N\times M)$，这种方法本书不再详述。

下面重点介绍一下时间复杂度为 $O(N+M)$的方法，首先是把 t1 树和 t2 树按照先序遍历的方式序列化，关于这个内容，请阅读本书“二叉树的序列化和反序列化”问题。以题目的例子来说，t1 树序列化后的结果为“1!2!4!#!8!#!#!5!9!#!#!#!3!6!#!#!7!#!#!”，记为 t1Str。t2 树序列化后的结果为“2!4!#!8!#!#!5!9!#!#!#!”，记为 t2Str。接下来只要验证 t2Str 是否是 t1Str 的子串即可，这个用 KMP 算法可以在线性时间内解决。所以 t1 序列化的过程为 $O(N)$，t2 序列化的过程为 $O(M)$，KMP 解决 t1Str 和 t2Str 的匹配问题 $O(M+N)$，所以时间复杂度为

$O(M+N)$。有关 KMP 算法的内容，请读者阅读本书“KMP 算法”问题，关于这个算法非常清晰的解释，这里不再详述。

本题最优解的全部过程请参看如下代码中的 isSubtree 方法。

```
public boolean isSubtree(Node t1, Node t2) {
        String t1Str = serialByPre(t1);
        String t2Str = serialByPre(t2);
        return getIndexOf(t1Str, t2Str) != -1;
}

public String serialByPre(Node head) {
        if (head == null) {
                return "#!";
        }
        String res = head.value + "!";
        res += serialByPre(head.left);
        res += serialByPre(head.right);
        return res;
}

// KMP
public int getIndexOf(String s, String m) {
        if (s == null || m == null || m.length() < 1 || s.length() < m.length())
        {
                return -1;
        }
        char[] ss = s.toCharArray();
        char[] ms = m.toCharArray();
        int si = 0;
        int mi = 0;
        int[] next = getNextArray(ms);
        while (si < ss.length && mi < ms.length) {
                if (ss[si] == ms[mi]) {
                        si++;
                        mi++;
                } else if (next[mi] == -1) {
                        si++;
                } else {
                        mi = next[mi];
                }
        }
        return mi == ms.length ? si - mi : -1;
}

public int[] getNextArray(char[] ms) {
        if (ms.length == 1) {
                return new int[] { -1 };
        }
        int[] next = new int[ms.length];
        next[0] = -1;
```

```
        next[1] = 0;
        int pos = 2;
        int cn = 0;
        while (pos < next.length) {
                if (ms[pos - 1] == ms[cn]) {
                        next[pos++] = ++cn;
                } else if (cn > 0) {
                        cn = next[cn];
                } else {
                        next[pos++] = 0;
                }
        }
        return next;
}
```

判断二叉树是否为平衡二叉树

【题目】

平衡二叉树的性质为：要么是一棵空树，要么任何一个节点的左右子树高度差的绝对值不超过 1。给定一棵二叉树的头节点 head，判断这棵二叉树是否为平衡二叉树。

【要求】

如果二叉树的节点数为 N，要求时间复杂度为 $O(N)$。

【难度】

士 ★☆☆☆

【解答】

解法的整体过程为二叉树的后序遍历，对任何一个节点 node 来说，先遍历 node 的左子树，遍历过程中收集两个信息，node 的左子树是否为平衡二叉树，node 的左子树最深到哪一层记为 lH。如果发现 node 的左子树不是平衡二叉树，无须进行任何后续过程，此时返回什么已不重要，因为已经发现整棵树不是平衡二叉树，退出遍历过程；如果 node 的左子树是平衡二叉树，再遍历 node 的右子树，遍历过程中再收集两个信息，node 的右子树是否为平衡二叉树，node 的右子树最深到哪一层记为 rH。如果发现 node 的右子树不是平衡二叉树，无须进行任何后续过程，返回什么也不重要，因为已经发现整棵树不是平衡二叉树，退出遍历过程；如果 node 的右子树也是平衡二叉树，就看 lH 和 rH 差的绝对值是否大