树的性质篇

LeetCode 中关于树的性质有以下题目：

1. Maximum Depth of Binary Tree
2. Minimum Depth of Binary Tree
3. Balanced Binary Tree
4. Same Tree
5. Symmetric Tree

1. Maximum Depth of Binary Tree，一般都是用递归实现。思路很简单，只需要对走到空结点返回0，然后其他依次按层递增，取左右子树中大的深度加1作为自己的深度即可。非递归解法一般采用层序遍历(相当于图的BFS)，因为如果使用其他遍历方式也需要同样的复杂度O(n)．层序遍历理解上直观一些，维护到最后的level便是树的深度。

代码如下：

```
// Solution 1 - Recursion
public int maxDepth(TreeNode root) {
    if(root == null){
        return 0;
    }
    return Math.max(maxDepth(root.left), maxDepth(root.right)) + 1;
}

// Solution 2 - Level Order Traversal
public int maxDepth(TreeNode root) {
    if(root == null){
        return 0;
    }
    LinkedList<TreeNode> queue = new LinkedList<TreeNode>();
    queue.offer(root);
    int curNum = 1;
    int nextNum = 0;
    int level = 0;
    while(!queue.isEmpty()){
        TreeNode cur = queue.poll();
        curNum--;
```

```
            if(cur.left != null){
                queue.offer(cur.left);
                nextNum++;
            }
            if(cur.right != null){
                queue.offer(cur.right);
                nextNum++;
            }
            if(curNum == 0){
                curNum = nextNum;
                nextNum = 0;
                level++;
            }
        }
        return level;
    }
```

做题时的感悟:

从本质上来说，add()和 offer()操作是没有区别的，只是 add 属于 list 的操作，而 offer()和 poll()则是队列的操作。所以如果规范一些，应该是在把它当成什么数据结构的时候就用那个数据结构对应的操作比较好。

2. Minimum Depth of Binary Tree，这道题因为是判断最小深度，所以必须增加一个叶子的判断（因为如果一个节点如果只有左子树或者右子树，我们不能取它左右子树中小的作为深度，因为那样会是0，我们只有在叶子节点才能判断深度，而在求最大深度的时候，因为一定会取大的那个，所以不会有这个问题）。这道题同样是递归和非递归的解法，递归解法比较常规的思路，比Maximum Depth of Binary Tree多加一个左右子树的判断。非递归解法同样采用层序遍历(相当于图的BFS)，只是在检测到第一个叶子的时候就可以返回了。代码如下:

```
    // Solution 1 - Recursion
    public int minDepth(TreeNode root) {
        if(root == null) {
            return 0;
        }
        if(root.left == null){
            return minDepth(root.right) + 1;
```

```
        }
        if(root.right == null){
            return minDepth(root.left) + 1;
        }
        return Math.min(minDepth(root.right), minDepth(root.left)) + 1;
    }

    // Solution 2 - Level Order Traversal
    public int minDepth(TreeNode root) {
        if(root == null){
            return 0;
        }
        LinkedList<TreeNode> queue = new LinkedList<TreeNode>();
        queue.offer(root);
        int curNum = 1;
        int nextNum = 0;
        int level = 1;
        while(!queue.isEmpty()){
            TreeNode cur = queue.poll();
            curNum--;
            if(cur.left == null && cur.right == null){
                return level;
            }
            if(cur.left != null){
                queue.offer(cur.left);
                nextNum++;
            }
            if(cur.right != null){
                queue.offer(cur.right);
                nextNum++;
            }
            if(curNum == 0){
                curNum = nextNum;
                nextNum = 0;
                level++;
            }
        }
        return level;
    }
```

做题时的感悟:

注意，这里的 level 要从 1 开始或者从 0 开始但 return level + 1。因为在检测到叶子结点的时候就可以返回了，需要加上当前层的高度 1。

3. Balanced Binary Tree，要判断树是否平衡，根据题目的定义，深度是必须的信息，所以我们必须维护深度，另一方面我们又要返回是否为平衡树，那么对于左右子树深度差的判断也是必要的。这里我们用一个整数来做返回值，而0或者正数用来表示树的深度，而-1则用来表示此树已经不平衡了，如果已经不平衡，则递归一直返回-1即可，也没有继续比较的必要了，否则就利用返回的深度信息看看左右子树是不是违反平衡条件，如果违反返回-1，否则返回左右子树深度大的加一作为自己的深度即可。算法的时间是一次树的遍历O(n)，空间是栈高度O(logn)。可以看出树的题目万变不离其宗，都是递归遍历，只是处理保存量，递归条件和结束条件会有一些变化。

代码如下:

```
public boolean isBalanced(TreeNode root) {
    return getHeightAndCheck(root) != -1;
}

private int getHeightAndCheck(TreeNode root){
    if(root == null){
        return 0;
    }
    int left = getHeightAndCheck(root.left);
    if(left == -1){
        return -1;
    }
    int right = getHeightAndCheck(root.right);
    if(right == -1){
        return -1;
    }
    int diff = Math.abs(left - right);
    if(diff > 1){
        return -1;
    }
    return Math.max(left, right) + 1;
}
```

做题时的感悟：

```
int left = getHeightAndCheck(root.left);
if(left == -1){
    return -1;
}
int right = getHeightAndCheck(root.right);
if(right == -1){
    return -1;
}
```

我们在得到 left 之后，应当立即判断 left 是否为-1，如果是说明左面已经不平衡了，可以直接返回-1，不需要再判断右面了。可以提高算法的时间复杂度。

4. Same Tree，这道题是树的题目，属于最基本的树遍历的问题。这里我们主要考虑一下结束条件，如果两个结点都是null，也就是到头了，那么返回true。如果其中一个是null，说明在一棵树上结点到头，另一棵树结点还没结束，即树不相同，或者两个结点都非空，并且结点值不相同，返回false。最后递归处理两个结点的左右子树，返回左右子树递归的与结果即可。这里使用的是先序遍历，算法的复杂度跟遍历是一致的，如果使用递归，时间复杂度是O(n)，空间复杂度是O(logn)。

树的题目大多都是用递归来完成比较简练，当然也可以如同Binary Tree Inorder Traversal中介绍的那样，用非递归方法甚至线索二叉树来做，只需要根据要求做相应改变即可。

代码如下：

```
public boolean isSameTree(TreeNode p, TreeNode q) {
    if(p == null && q == null){
        return true;
    }
    if(p == null || q == null){
        return false;
    }
    if(p.val != q.val){
        return false;
    }
    return isSameTree(p.left, q.left) && isSameTree(p.right, q.right);
}
```

做题时的感悟:

1. 可以一个方法完成的，尽量不要再多创建一个方法，要先考虑一个方法能不能完成。

2. 做判断的时候，思路一定要清晰。判断的条件范围不能相互重叠，而要像切蛋糕一样，每次切去一部分，而将所有情况整合起来又是整个蛋糕，即包含所以情况。

5. Symmetric Tree，本质上还是树的遍历，也就是里面的程序框架加上对称性质的判断即可。主要说说结束条件，假设到了某一结点，不对称的条件有以下三个：

（1）左边为空而右边不为空；

（2）左边不为空而右边为空；

（3）左边值不等于右边值。

根据这几个条件在遍历时进行判断即可。算法的时间复杂度是树的遍历O(n)，空间复杂度同样与树遍历相同是O(logn)。非递归方法是使用层序遍历来判断对称性质。非递归方法比起递归方法要繁琐一些，因为递归可以根据当前状态（比如两个都为空）直接放回true，而非递归则需要对false的情况一一判断，不能如递归那样简练。

代码如下：

```
// Solution 1 - Recursion
public boolean isSymmetric(TreeNode root) {
    if(root == null){
        return true;
    }
    return helper(root.left, root.right);
}

private boolean helper(TreeNode p, TreeNode q){
    if(p == null && q == null){
        return true;
    }
    if(p == null || q == null){
        return false;
    }
    if(p.val != q.val){
        return false;
    }
    return helper(p.left, q.right) && helper(p.right, q.left);
}
```

```
// Solution 2 - Iteration
public boolean isSymmetric(TreeNode root) {
    if(root == null){
        return true;
    }
    if(root.left == null && root.right == null){
        return true;
    }
    if(root.left == null || root.right == null){
        return false;
    }
    LinkedList<TreeNode> left = new LinkedList<TreeNode>();
    LinkedList<TreeNode> right = new LinkedList<TreeNode>();
    left.offer(root.left);
    right.offer(root.right);
    while(!left.isEmpty() && !right.isEmpty()){
        TreeNode l = left.poll();
        TreeNode r = right.poll();
        if(l.val != r.val){
            return false;
        }
        if(l.left != null && r.right == null || l.left == null && r.right != null){
            return false;
        }
        if(l.right != null && r.left == null || l.right == null && r.left != null){
            return false;
        }
        if(l.left != null && r.right != null){
            left.offer(l.left);
            right.offer(r.right);
        }
        if(l.right != null && r.left != null){
            left.offer(l.right);
            right.offer(r.left);
        }
    }
    return true;
}
```

做题时的感悟:

1. 大神利用!left.isEmpty() && !right.isEmpty()来判断循环的结束条件，是因为在后面的代码中，左右子树是同时加入或者弹出的，所以不会出现1个为空，另一个不为空的情况。否则，需要在判断结果前，判断一下是不是两个queue都为空。

2. 经过验证，是可以向queue中加入空值null的。并且它会使queue的size加1。