学习目标

1.基础知识

1.1 常见概念

1.2 树的性质

1.3 树的定义与存储方式

1.4 树的遍历方式

1.5 通过序列构造二叉树

1.5.1前中序列复原二叉树

1.5.2 通过中序和后序序列恢复二叉树

2. 树的层次遍历

2.1 层次遍历简介

2.2 基本的层序遍历与变换

2.2.1 二叉树的层序遍历

2.2.2 层序遍历-自底向上

2.2.3 二叉树的锯齿形层序遍历

2.2.4 N 叉树的层序遍历

2.3 几个处理每层元素的题目

2.3.1 在每个树行中找最大值

2.3.2 在每个树行中找平均值

2.3.3 二叉树的右视图

3.树的深度优先遍历

3.1深入理解前中后序遍历

3.2 迭代法实现二叉树的三种遍历

3.2.1 迭代法实现前序遍历

3.2.2 迭代法实现中序遍历

3.2.3 迭代法实现后序遍历

3.3 深度和高度专题

3.3.1 最大深度问题

3.3.2 判断平衡树

3.3.3 最小深度

3.3.4 N叉树的最大深度

3.4 二叉树里的双指针

3.4.1 判断两棵树是否相同

3.4.2 对称二叉树

3.4.3 合并二叉树

3.5 路径专题

3.5.1 二叉树的所有路径

3.5.2 路径总和

3.6 翻转的妙用

3.6.1 翻转二叉树

3.6.2 最底层最左边

4. 中序遍历与搜索树

4.1 中序与搜索树

4.2 二叉搜索树中搜索特定值

4.3 验证二叉搜索树

4.4 有序数组转为二叉搜索树

4.5 公共祖先问题

4.5.1 二叉搜索树的最近公共祖先

4.5.2 普通二叉树的最近公共祖先
5.大厂面试实战

学习目标

从这一章开始，开始学习树的大家族，后面要学习的二分查找算法、几种核心的排序算法以及图算法都与树有非常密切的关系。树在工程中也有非常广泛的应用，我们前面说”没学会反转，链表相当于白学“，现在再加一句”没学会树，算法相当于没学“。

算法中关于树的考察主要有四个方面：层次遍历以及拓展问题，前后序遍历与拓展问题、中序遍历与搜索树问题、堆和平衡树等变换的树。其重点如下：

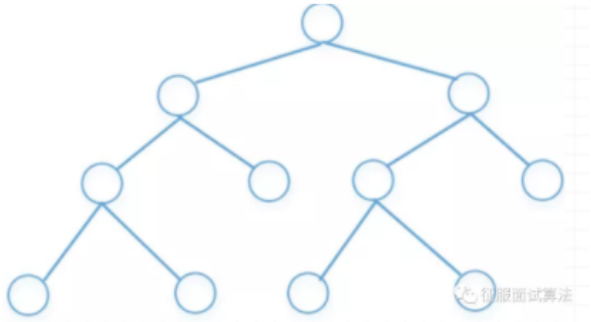
1. 层次遍历，全部的层次遍历几乎就一个模板，本身也不难，之前经常看到有人面试遇到这类问题就直接走人，实在可惜。本部分除了要掌握层次遍历的实现方法，还要掌握相关的常见变换题。
2. 树的前中后序是面试的热点，其根基都是递归。本部分要掌握递归的原理和写法，掌握前中后序遍历的迭代和递归实现方法，掌握前序和后序相关的经典题目。这些内容也是后面回溯、动态规划等的基础。
3. 如果将二分查找的搜索画成图，那与二叉树的中序搜索完全一样，因此二叉树的中序遍历和搜索树本质是一样的。这部分要理解其原理和几个常见的题目。
4. 除此之外还有很多特殊结构的树，例如平衡树是为了提高搜索树效率而产生的，红黑树是为了提高平衡树的效率而产生了，而红黑树本质上就是2-3树。堆也是一种非常重要的结构，有很多经典的题目几乎只能用堆。

本章我们重点研究二叉树相关的算法题目，主要是层次遍历和深度优先遍历方面的问题。因此将平衡树、红黑树、堆等内容移到后面相关章节中。

1.基础知识

1.1 常见概念

树是一个有n个有限节点组成一个具有层次关系的集合，每个节点有0个或者多个子节点，没有父节点的节点称为根节点，也就是说除了根节点以外每个节点都有父节点，并且有且只有一个。树的种类比较多，我们最常见的应该是二叉树了，基本结构如下：



树有一些特有的概念，如下：

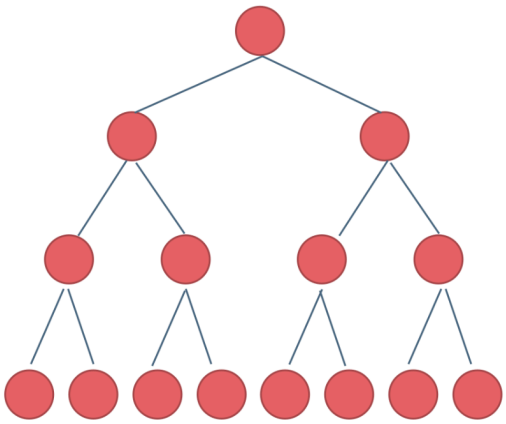
1. 节点的度:一个节点含有的子节点的个数称为该节点的度;
2. 树的度:一棵树中，最大的节点的度称为树的度，注意与节点度的区别;
3. 叶节点或终端节点:度为0的节点称为叶节点;
4. 非终端节点或分支节点:度不为0的节点;
5. 双亲节点或父节点:若一个节点含有子节点，则这个节点称为其子节点的父节点;

6. 孩子节点或子节点:一个节点含有的子树的根节点称为该节点的子节点;
7. 兄弟节点:具有相同父节点的节点互称为兄弟节点;
8. 节点的祖先:从根到该节点所经分支上的所有节点;
9. 子孙:以某节点为根的子树中任一节点都称为该节点的子孙。
10. 森林:由m(m>=0)棵互不相交的树的集合称为森林;
11. 无序树:树中任意节点的子节点之间没有顺序关系，这种树称为无序树，也称为自由树;
12. 有序树:树中任意节点的子节点之间有顺序关系，这种树称为有序树;
13. 二叉树:每个节点最多含有两个子树的树称为二叉树;

1.2 树的性质

性质1: 在二叉树的第i层上至多有2^(i-1)个结点（i>0）
性质2: 深度为k的二叉树至多有2^k - 1个结点（k>0）
性质3: 对于任意一棵二叉树，如果其叶结点数为N0，而度数为2的结点总数为N2，则N0=N2+1;
性质4:具有n个结点的完全二叉树的深度必为 log2(n+1)
性质5:对完全二叉树，若从上至下、从左至右编号，则编号为i 的结点，其左孩子编号必为2i，其右孩子编号必为2i＋1；其双亲的编号必为i/2（i＝1 时为根,除外）

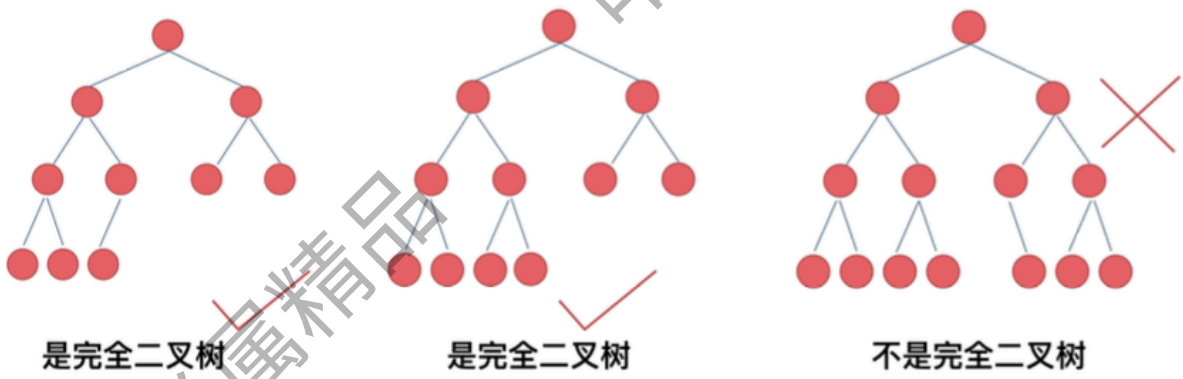
满二叉树和完全二叉树是经常晕的问题，我们有必要单独看一下。满二叉树就是如果一棵二叉树只有度为0的节点和度为2的节点，并且度为0的节点在同一层上，则这棵二叉树为满二叉树。



这棵二叉树为满二叉树，也可以说深度为k=4，有2^k-1=15个节点的二叉树。

完全二叉树的定义如下:在完全二叉树中，除了最底层节点可能没填满外，其余每层节点数都达到最大 值，并且最下面一层的节点都集中在该层最左边的若干位置。

这个定义最邪乎了，估计大部分看了之后还是不懂什么是完全二叉树，看这个图就知道了：



前面两棵树的前n-1层都是满的，最后一层所有节点都集中在左侧区域，而且节点之间不能有空隙。最后一个为什么不是？因为有一节点缺了一个左子节点。

1.3 树的定义与存储方式

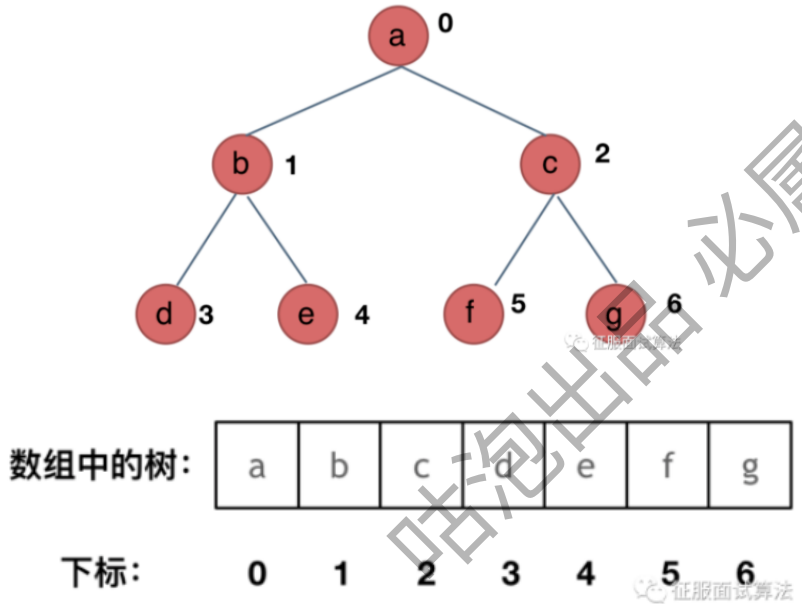
在java中定义树还是比较容易的，如果是二叉树，只要在链表的定义上增加一个指针就可以了：

```
public class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
}
```

这里本质上就是有两个引用，分别指向两个位置，为了便于理解，我们分别命名为左孩子和右孩子。如果是N叉树该如何定义呢？其实就是每个节点最多可以有N个指针指向其他地方，这是不用left和right，使用一个List就可以了，也就是：

```
public class TreeNode {
    int val;
    List<TreeNode> nodes;
}
```

那能否用数组来存储二叉树呢?其实就是用数组来存储二叉树，顺序存储的方式如图:



用数组来存储二叉树如何遍历的呢?如果父节点的数组下表是i，那么它的左孩子就是i * 2 + 1，右孩子就是 i * 2 + 2。但是用链式表示的二叉树，更有利于我们理解，所以一般我们都是用链式存储二叉树。所以大家要了解，用数组依然可以表示二叉树。

使用数组存储的最大不足是可能存在大量的空间浪费。例如上图中如果b分支没有，那么数组种1 3 4 位置都要空着，但是整个数组的大小仍然是7，因此很少使用数组来存储树。

1.4 树的遍历方式

我们现在就来看一下树的常见遍历方法。二叉树的遍历方式有层次遍历和深度优先遍历两种:

- 深度优先遍历:先往深走，遇到叶子节点再往回走。
- 广度优先遍历:一层一层的去遍历，一层访问完再访问下一层。

这两种遍历方式不仅仅是二叉树，N叉树也有这两种方式的，图结构也有，只不过我们更习惯叫广度优先和深度优先，本质是一回事。深度优先又有前中后序三种， 有同学总分不清这三个顺序，问题就在不清楚这里前中后是相对谁来说的。记住一点：**前指的是中间的父节点在遍历中的顺序**，只要大家记住 前中后序指的就是中间节点的位置就可以了。

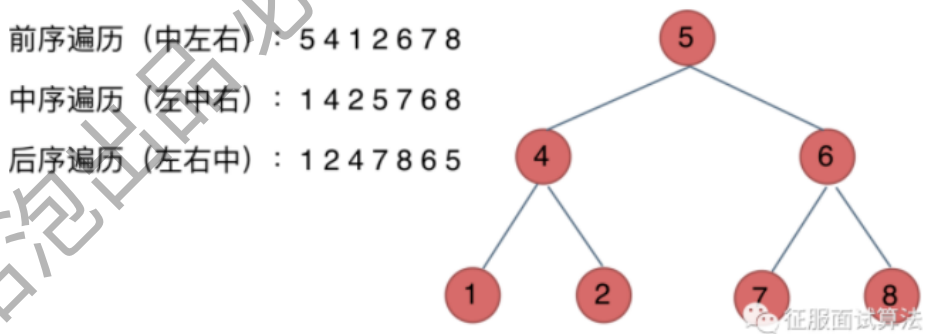
看如下中间节点的顺序，就可以发现，访问中间节点的顺序就是所谓的遍历方式

前序遍历:中左右

中序遍历:左中右

后序遍历:左右中

大家可以对着如下图，看看自己理解的前后中序有没有问题。



后面大量的算法都与这四种遍历方式有关，有的题目根据处理角度不同 ，可以用层次遍历，也可以用一种甚至两种深度优先的方式来实现。

1.5 通过序列构造二叉树

前面我们已经介绍了前中后序遍历的基本过程，现在我们看一下如何通过给出的序列来恢复原始二叉树，看三个序列:

(1) 前序：1 2 3 4 5 6 8 7 9 10 11 12 13 15 14

(2) 中序：3 4 8 6 7 5 2 1 10 9 11 15 13 14 12

(3) 后序：8 7 6 5 4 3 2 10 15 14 13 12 11 9 1

1.5.1前中序列复原二叉树

我们先看如何通过前中序列复原二叉树:

(1) 前序：1 2 3 4 5 6 8 7 9 10 11 12 13 15 14

(2) 中序：3 4 8 6 7 5 2 1 10 9 11 15 13 14 12

- 第一轮:

我们知道前序第一个访问的就是根节点，所以根节点就是1。

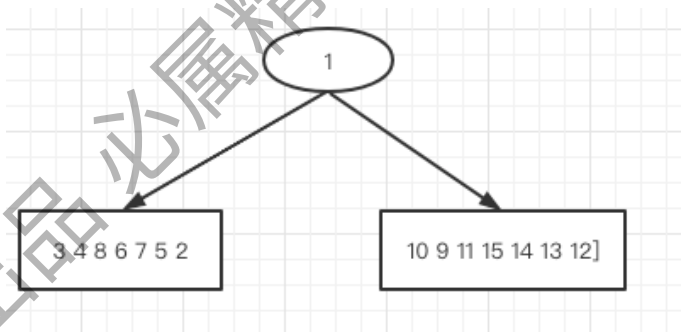
中序遍历的特点是根节点的左子树的元素都在根节点的左侧，右子树的元素都在根节点的右侧，从中序遍历序列我们可以划分成如下结构：

```
中序序列划分：
[3 4 8 6 7 5 2] 1[ 10 9 11 15 13 14 12]
前序序列划分为：
1 [2 3 4 5 6 8 7 ] [9 10 11 12 13 15 14]
```

上面前序序列第一个括号里的都是左子树的元素，第二个括号一定都是右子树的元素。

那这里怎么知道两个括号从哪里分开呢？是参照中序的两个数组划分的。我们看到前序中7之前的元素都在中序第一个数组中，9之后的所有元素就在第二个数组种，所以我们从7和9之间划分。

由此，画图表示一下此时知道的树的结构为：



- 第二轮：

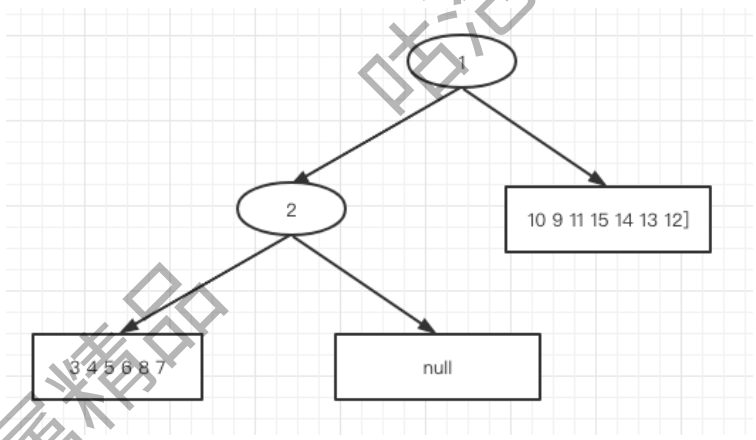
我们先看两个序列的第一个数组：

前序：2 3 4 5 6 8 7　中序：3 4 8 6 7 5 2

此时又可以利用上面的结论划分了：根节点是2，然后根据2在中序中的位置可以划分为：

```
前序：2 [3 4 5 6 8 7 ]
中序：[3 4 8 6 7 5 ] 2
```

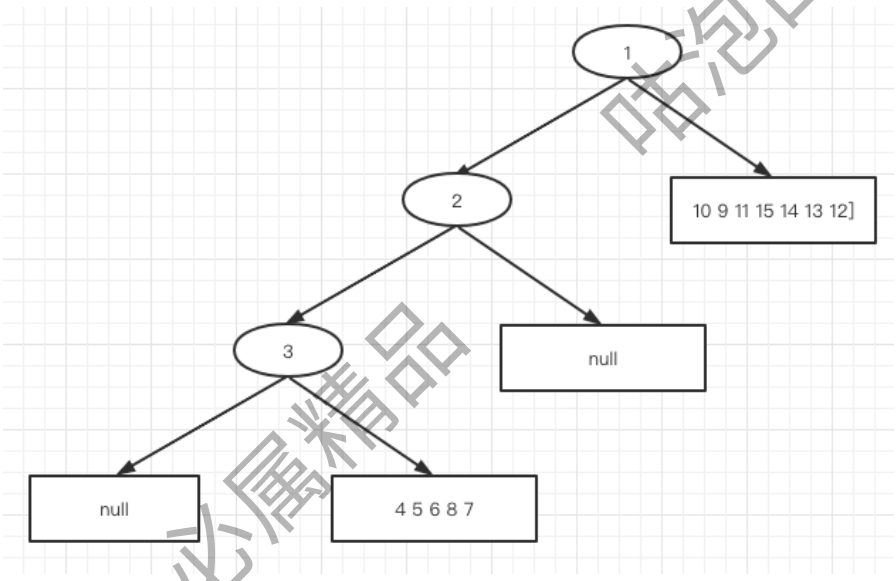
所以此时的树结构为：



- 第三轮：对 3 4 5 6 8 7 继续划分：

```
前序：3 [4 5 6 8 7 ]      中序：3 [4 8 6 7 5 ]
```

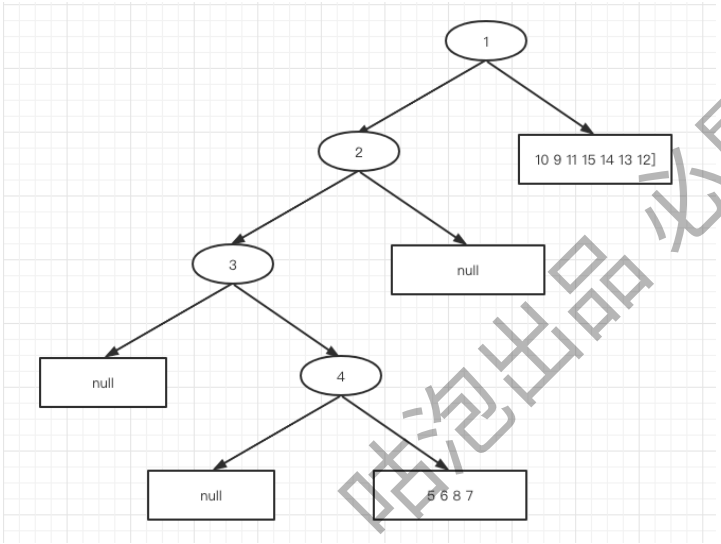
此时结构为：



- 第四轮，对 4 5 6 8 7 继续划分：

```
前序：4 [5 6 8 7 ] 中序：4 [8 6 7 5 ]
```

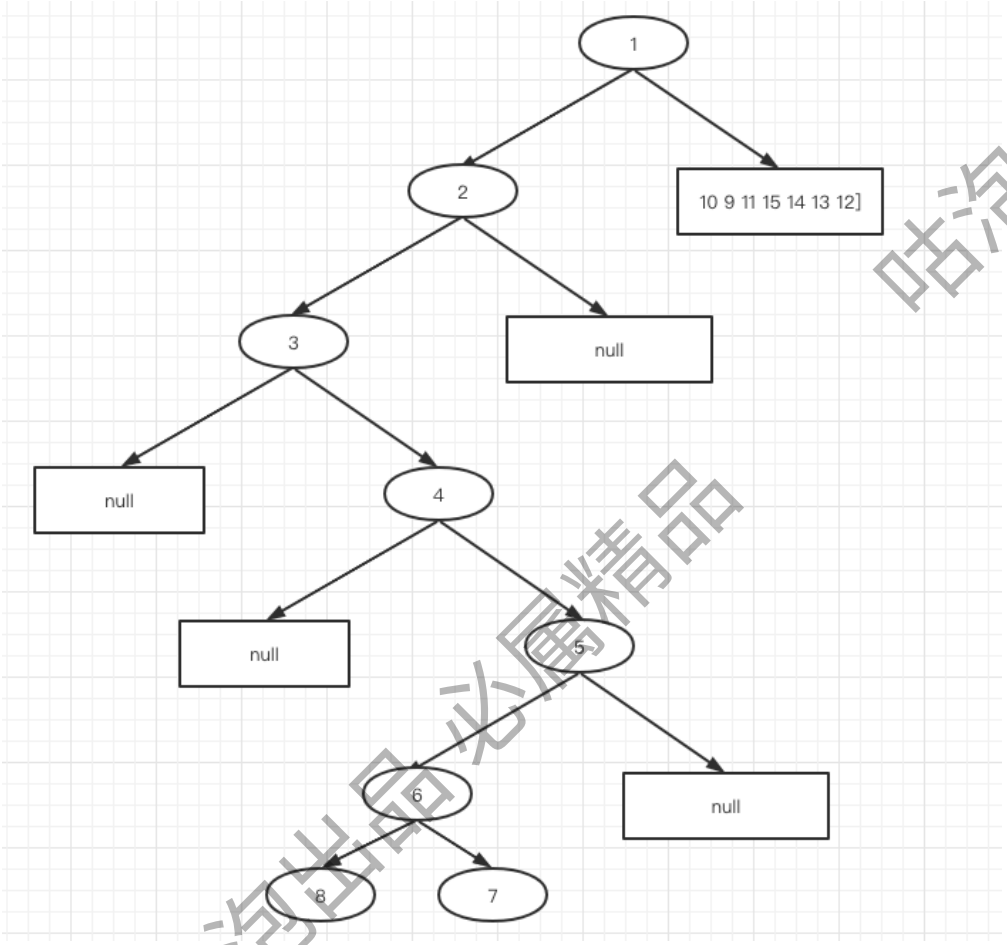
此时树为：



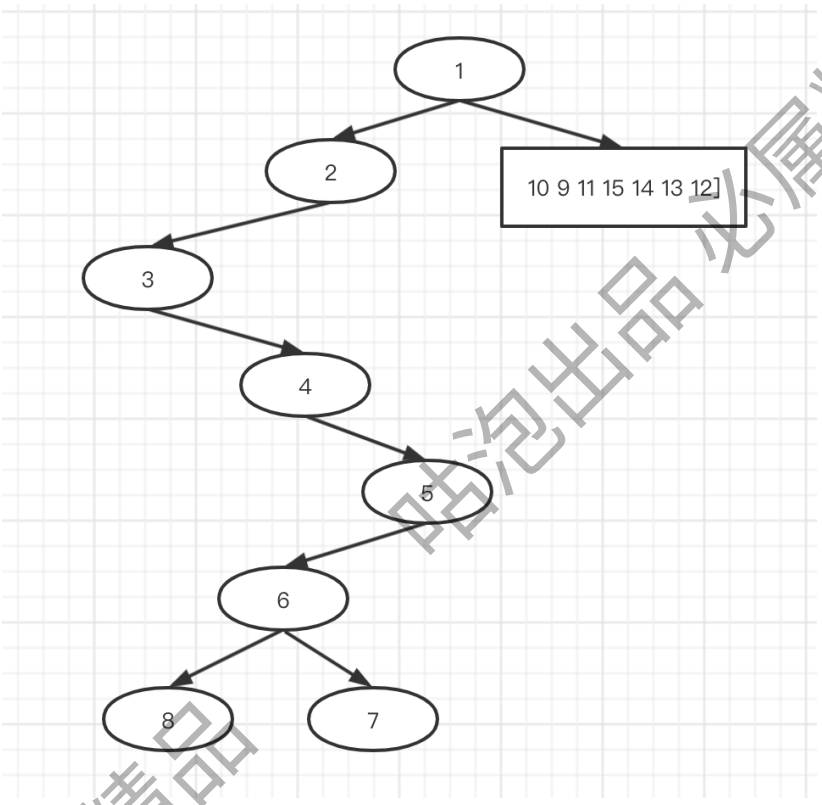
- 第五轮：对 5 6 8 7 继续划分：

```
前序：5 [6 8 7 ] 中序：[8 6 7 ] 5
```

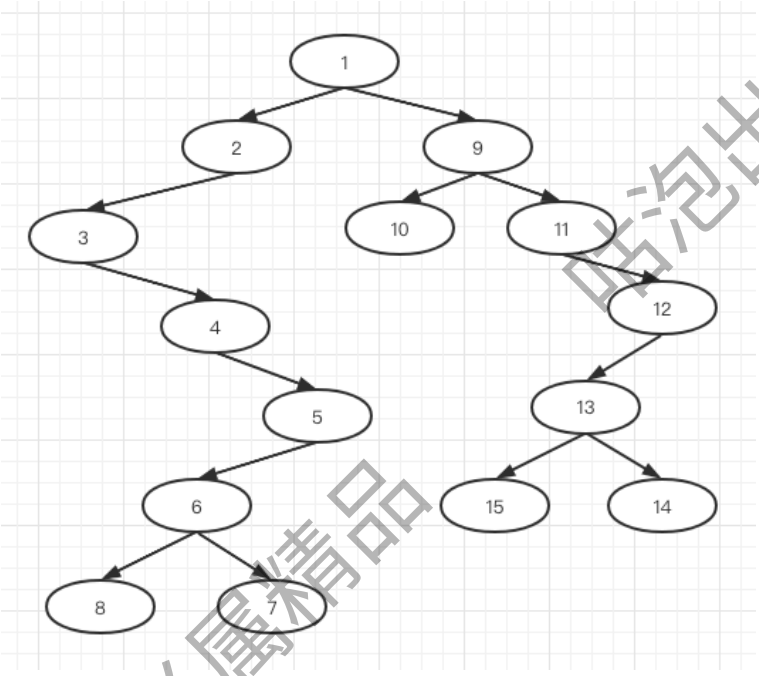
这个题有点恶心，每次只能确定一个元素，道理是这样子，直接画最终结果吧：



这个图有点丑，我们将空节点都去掉就是这样了：



同理，对于序列[10 9 11 15 13 14 12]，我们也可以逐步划分，这个请读同学自己试一试，最终结果为：



1.5.2 通过中序和后序序列恢复二叉树

通过中序和后序也能恢复原始序列的，唯一的不同是后序序列的最后一个是根节点，中序的处理也是上面一样的过程：

前序：1 2 3 4 5 6 8 7 9 10 11 12 13 15 14

中序：3 4 8 6 7 5 2 1 10 9 11 15 13 14 12

后序：8 7 6 5 4 3 2 10 15 14 13 12 11 9 1

读者可以自行试一试，我们就不再赘述。

问题：为什么前序和后序不能恢复二叉树

既然上面两种都行，那为什么前序和后序不行呢？我们看上面的例子：

(1) 前序：1 2 3 4 5 6 8 7 9 10 11 12 13 15 14

(2) 后序：8 7 6 5 4 3 2 10 15 14 13 12 11 9 1

根据上面的说明，我们通过前序可以知道根节点是1，通过后序也能知道根节点是1，但是中间是怎么划分的呢？其他元素哪些属于左子树，哪些属于右子树呢？很明显通过两个序列都不知道，所以前序和后序序列不能恢复二叉树。

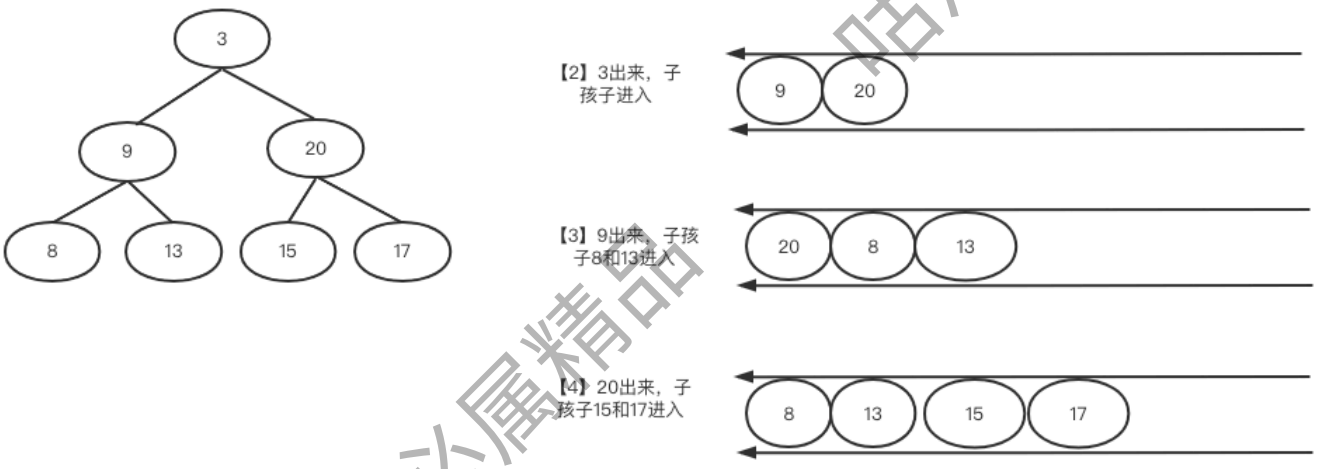
如果将上述过程用代码实现该怎么做呢？通过前序和中序构造树就是LeetCode105题，通过中序和后序构造树就是LeetCode106题，实现过程略微繁琐，感兴趣的同学可以研究一下。

2. 树的层次遍历

2.1 层次遍历简介

广度优先在面试里出现的频率非常高，整体属于简单题，但是很多人面试遇到时就直接放弃了，实在可惜。我们本章就集中研究一下到底难在哪里。

广度优先又叫层次遍历，基本过程如下：



层次遍历就是从根节点开始，先访问根节点下面一层全部元素，再访问之后的层次，类似金字塔一样一层层访问。我们可以看到这里就是从左到右一层一层的去遍历二叉树，先访问3，之后访问1的左右子孩子 9 和10，之后分别访问9 和20的左右子孩子 [4,5]和[6,7]，最后得到结果[3,9,20,8,13,15,7]。

这里的问题是怎么将遍历过的元素的子孩子保存一下呢，例如访问9时其左右子孩子8和13应该先存一下，直到处理20之后才会处理。使用队列来存储能完美解决上述问题，例如上面的图中：

```
1．首先3入队。
2．然后3出队，之后将3的左右子孩子9和10 保存到队列中。
3．之后9出队，将9的左右子孩子8和13入队。
4．之后20出队，将20的左右子孩子15和7入队。
5．之后 8，13，15，7分别出队，此时都是叶子结点，只出队就行了。
```

该过程不复杂，如果能将树的每层次分开了，是否可以整点新花样？首先，能否将每层的元素顺序给反转一下呢？能否奇数行不变，只将偶数行反转呢？能否将输出层次从低到root逐层输出呢？再来，既然能拿到每一层的元素了，能否找到当前层最大的元素？最小的元素？最右的元素（右视图）？最左的元素（左视图）？整个层的平均值?

很明显都可以！这么折腾有啥用呢？没啥用！但这就是层次遍历的高频算法题！这就是LeetCode里经典的层次遍历题！

102.二叉树的层序遍历

107.二叉树的层次遍历II

199.二叉树的右视图

637.二叉树的层平均值

429.N叉树的前序遍历

515.在每个树行中找最大值

116.填充每个节点的下一个右侧节点指针

117.填充每个节点的下一个右侧节点指针II

103 锯齿层序遍历

除此之外，在深度优先的题目里，有些仍然会考虑层次遍历的实现方法。

2.2 基本的层序遍历与变换

我们先看最简单的情况，仅仅遍历并输出全部元素，如下图。

```
/**
 *树结构如下
 * 3
   / \
  9  20
    /  \
   15   7
 * 应输出结果 [3, 9, 20, 15, 7]
 */
```

方法上面已经图示了，这里看一下怎么代码实现。先访问根节点，然后将其左右子孩子放到队列里，接着继续出队，出来的元素都将其左右自孩子放到队列里，直到队列为空了就退出就行了：

```
 List<Integer> simpleLevelOrder(TreeNode root) {
    if (root == null) {
        return new ArrayList<Integer>();
    }

    List<Integer> res = new ArrayList<Integer>();
    LinkedList<TreeNode> queue = new LinkedList<TreeNode>();
    //将根节点放入队列中，然后不断遍历队列
    queue.add(root);
    //有多少元素执行多少次
    while (queue.size() > 0) {
        //获取当前队列的长度，这个长度相当于 当前这一层的节点个数
        TreeNode t = queue.remove();
        res.add(t.val);
        if (t.left != null) {
            queue.add(t.left);
        }
        if (t.right != null) {
            queue.add(t.right);
        }
    }
    return res;
}
```

根据树的结构可以看到，一个结点在一层访问之后，其子孩子都是在下层按照FIFO的顺序处理的，因此队列就是一个缓存的作用。

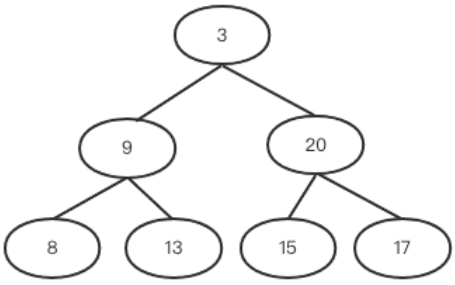
如果要你将每层的元素分开该怎么做呢？请看一下题：

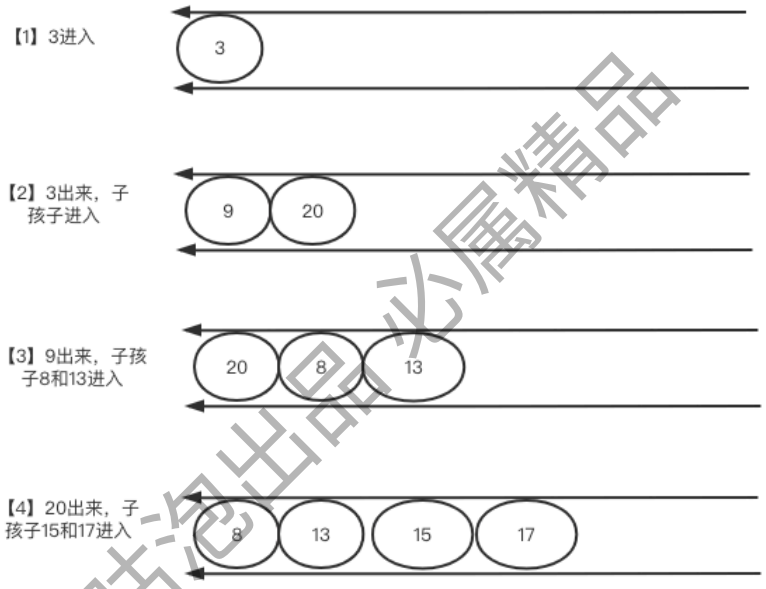
2.2.1 二叉树的层序遍历

LeetCode102 题目要求：给你一个二叉树，请你返回其按层序遍历得到的节点值。(即逐层地，从左到右访问所有节点)。

```
示例1：
二叉树：[3,9,20,null,null,15,7],
   3
   / \
  9  20
    /  \
   15   7
返回其层序遍历结果：
 [
  [3],
  [9,20],
  [15,7]
]
```

我们再观察执行过程图，我们先将根节点放到队列中，然后不断遍历队列。





那如何判断某一层访问完了呢？简单，用一个变量size标记一下就行了，size表示某一层的元素个数，只要出队，就将size减1，减到0就说明该层元素访问完了。当size变成0之后，这时队列中剩余元素的个数恰好就是下一层元素的个数，因此重新将size标记为下一层的元素个数就可以继续处理新的一行了，例如在上面的序列中：

```
1.首先拿根节点3，其左/右子结点都不为空，就将其左右放入队列中，因此此时3已经出队了，剩余元素9和20恰好就是第二层的所有结点，此时size=2。
2.继续，将9从队列中拿走，size--变成1，并将其子孩子8和13入队。之后再将20 出队，并将其子孩子15和7入队，此时再次size--，变成9了。当size=0，说明当前层已经处理完了，此时队列有四个元素，而且恰好就是下一层的元素个数。
```

最后，我们把每层遍历到的节点都放入到一个结果集中，将其返回就可以了：

按层打印经典版代码：

```
public List<List<Integer>> level102Order(TreeNode root) {
    if(root==null) {
      return new ArrayList<List<Integer>>();
    }

    List<List<Integer>> res = new ArrayList<List<Integer>>();
    LinkedList<TreeNode> queue = new LinkedList<TreeNode>();
    //将根节点放入队列中，然后不断遍历队列
    queue.add(root);
    while(queue.size()>0) {
      //获取当前队列的长度，也就是当前这一层的元素个数
      int size = queue.size();
      ArrayList<Integer> tmp = new ArrayList<Integer>();
      //将队列中的元素都拿出来(也就是获取这一层的节点)，放到临时list中
      //如果节点的左/右子树不为空，也放入队列中
      for(int i=0;i<size;++i) {
        TreeNode t = queue.remove();
        tmp.add(t.val);
        if(t.left!=null) {
          queue.add(t.left);
        }
        if(t.right!=null) {
          queue.add(t.right);
        }
      }
      //此时的tmp就是当前层的全部元素，用List类型的tmp保存，加入最终结果集中
      res.add(tmp);
    }
    return res;
  }
}
```

上面的代码是本章最重要的算法之一，也是整个算法体系的核心算法之一，与链表反转、二分查找属于同一个级别，务必认真学习！理解透彻，然后记住！

上面的算法理解了，那接下来一些列的问题就轻松搞定了。

注意

另外一个需要注意的是在java中实现队列的方法基础类不止一个，例如上面使用了`LinkedList<TreeNode> queue`,使用的入队和和出队就是add和remove，除此之外还可以使用以下组合：

```
Queue<TreeNode> queue = new LinkedList<TreeNode>();
入队：queue.offer(left);
出队：queue.poll();
该方法还提供了offerLast和offerFirst等方法，可以更方便的决定在哪一侧处理
```

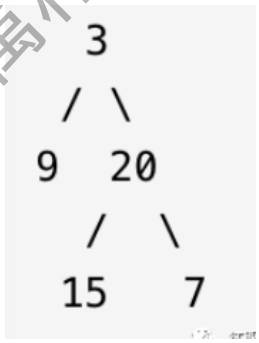
还可以使用如下的组合：

```
Deque<Integer> q = new LinkedList<Integer>();
入队：q.addLast(root); q.addFirst(root);
出队q.pollFirst(); q.pollLast();
```

这些组合在看算法材料时经常会见到，在某些场景下能更轻松处理某写问题，但是笔者不建议使用，因为面试手写代码的时候可能会记不住这些方法，以至于执行时出现问题，笔者建议都使用上面的"按层打印经典版"或者Queue来做，将这个模板训练的特别熟悉，面试能信手拈来。

2.2.2 层序遍历-自底向上

LeetCode 107.给定一个二叉树，返回其节点值自底向上的层序遍历。（即按从叶子节点所在层到根节点所在的层，逐层从左向右遍历）。例如给定的二叉树为：



返回结果为：

```
[
  [15,7],
  [9,20],
  [3]
]
```

如果要求从上到下输出每一层的节点值，做法是很直观的，在遍历完一层节点之后，将存储该层节点值的列表添加到结果列表的尾部。这道题要求从下到上输出每一层的节点值，只要对上述操作稍作修改即可，在遍历完一层节点之后，将存储该层节点值的列表添加到结果列表的头部。

为了降低在结果列表的头部添加一层节点值的列表的时间复杂度，结果列表可以使用链表的结构，在链表头部添加一层节点值的列表的时间复杂度是 O(1)。在 Java 中，由于我们需要返回的 List 是一个接口，这里可以使用链表实现。

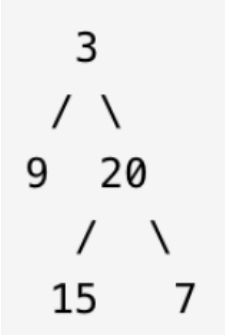
```
    public List<List<Integer>> levelOrderBottom(TreeNode root) {
        List<List<Integer>> levelOrder = new LinkedList<List<Integer>>();
        if (root == null) {
            return levelOrder;
        }
        Queue<TreeNode> queue = new LinkedList<TreeNode>();
        queue.offer(root);
        while (!queue.isEmpty()) {
            List<Integer> level = new ArrayList<Integer>();
```

```
            int size = queue.size();
            for (int i = 0; i < size; i++) {
                TreeNode node = queue.poll();
                level.add(node.val);
                TreeNode left = node.left, right = node.right;
                if (left != null) {
                    queue.offer(left);
                }
                if (right != null) {
                    queue.offer(right);
                }
            }
            levelOrder.add(0, level);//栈
        }
        return levelOrder;
    }
```

2.2.3 二叉树的锯齿形层序遍历

LeetCode103 题，要求是：给定一个二叉树，返回其节点值的锯齿形层序遍历。（即先从左往右，再从右往左进行下一层遍历，以此类推，层与层之间交替进行）。

例如：给定二叉树 [3,9,20,null,null,15,7]



返回结果是：

```
[
  [3],
  [20,9],
  [15,7]
]
```

这个题也是102的变种，只是最后输出的要求有所变化，要求我们按层数的奇偶来决定每一层的输出顺序。如果当前层数是偶数，从左至右输出当前层的节点值，否则，从右至左输出当前层的节点值。这里只要采用以

我们依然可以沿用第 102 题的思想，为了满足题目要求的返回值为「先从左往右，再从右往左」交替输出的锯齿形，可以利用「双端队列」的数据结构来维护当前层节点值输出的顺序。双端队列是一个可以在队列任意一端插入元素的队列。在广度优先搜索遍历当前层节点拓展下一层节点的时候我们仍然从左往右按顺序拓展，但是对当前层节点的存储我们维护一个变量 isOrderLeft 记录是从左至右还是从右至左的：

- 如果从左至右，我们每次将被遍历到的元素插入至双端队列的末尾。

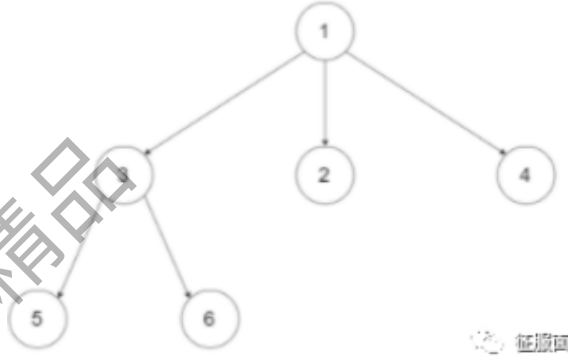
- 从右至左，我们每次将被遍历到的元素插入至双端队列的头部。

```
    public List<List<Integer>> zigzagLevelOrder(TreeNode root) {
      List<List<Integer>> ans = new LinkedList<List<Integer>>();
      if (root == null) {
          return ans;
      }
      Queue<TreeNode> queue = new LinkedList<TreeNode>();
      queue.offer(root);
      boolean isOrderLeft = true;
      while (!queue.isEmpty()) {
          Deque<Integer> levelList = new LinkedList<Integer>();
          int size = queue.size();
          for (int i = 0; i < size; ++i) {
              TreeNode curNode = queue.poll();
              if (isOrderLeft) {
                  levelList.offerLast(curNode.val);
              } else {
                  levelList.offerFirst(curNode.val);
              }
              if (curNode.left != null) {
                  queue.offer(curNode.left);
              }
              if (curNode.right != null) {
                  queue.offer(curNode.right);
              }
          }
          ans.add(new LinkedList<Integer>(levelList));
          isOrderLeft = !isOrderLeft;
      }
      return ans;
  }
```

2.2.4 N 叉树的层序遍历

LeetCode429 给定一个 N 叉树，返回其节点值的层序遍历。（即从左到右，逐层遍历）。

树的序列化输入是用层序遍历，每组子节点都由 null 值分隔（参见示例）。



```
输入：root = [1,null,3,2,4,null,5,6](表述树的元素是这个序列)
输出：[[1],[3,2,4],[5,6]]
```

N叉树的定义为：

```
class Node {
    public int val;
    public List<Node> children;

    public Node() {}

    public Node(int _val) {
        val = _val;
    }

    public Node(int _val, List<Node> _children) {
        val = _val;
        children = _children;
    }
}
```

这个也是102的扩展，很简单的广度优先，与二叉树的层序遍历基本一样，借助队列即可实现。

```
 public List<List<Integer>> nLevelOrder(Node root) {
        List<List<Integer>> value = new ArrayList<>();
        Deque<Node> q = new ArrayDeque<>();
        if (root != null)
            q.addLast(root);
        while (!q.isEmpty()) {
            Deque<Node> next = new ArrayDeque<>();
            List<Integer> nd = new ArrayList<>();
            while (!q.isEmpty()) {
                Node cur = q.pollFirst();
                nd.add(cur.val);
                for (Node chd : cur.children) {
                    if (chd != null)
                        next.add(chd);
                }
            }
            q = next;
            value.add(nd);
        }
        return value;
    }
```

2.3 几个处理每层元素的题目

如果我们拿到了每一层的元素，那是不是可以利用一下造几个题呢？例如每层找最大值、平均值、最右侧的值呢？当然可以。LeetCode里就有三道非常明显的题目。

515.在每个树行中找最大值(最小)

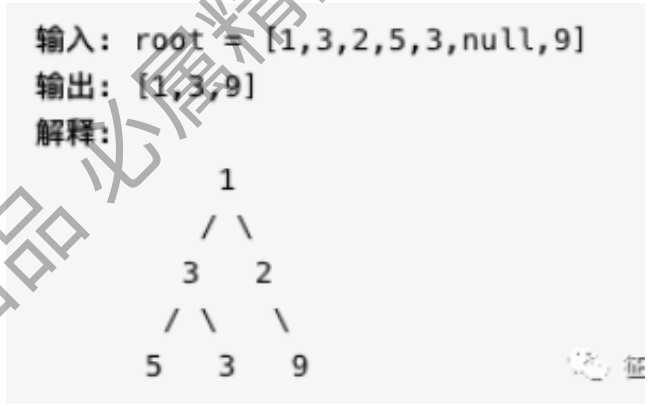
637.二叉树的层平均值

199.二叉树的右视图

既然能这么干，我们能否自己造几个题：求每层最小值可以不？求每层最左侧的可以不？我们是不是可以给LeetCode贡献几道题了？

2.3.1 在每个树行中找最大值

LeetCode 515题目要求：给定一棵二叉树的根节点 root，请找出该二叉树中每一层的最大值。



这里其实就是在得到一层之后使用一个变量来记录当前得到的最大值：

```
public List<Integer> largestValues(TreeNode root) {
        List<Integer> res = new ArrayList<>();
        Deque<TreeNode> deque = new ArrayDeque<>();

        if (root != null) {
            deque.addLast(root);
        }

        while (!deque.isEmpty()) {
            int size = deque.size();
            int levelMaxNum = Integer.MIN_VALUE;
            for (int i = 0; i < size; i++) {
                TreeNode node = deque.poll();
                levelMaxNum = Math.max(node.val,levelMaxNum);
                if (node.left != null) deque.addLast(node.left);
                if (node.right != null) deque.addLast(node.right);
            }
            res.add(levelMaxNum);
        }
        return res;
    }
```

2.3.2 在每个树行中找平均值

LeetCode 637 要求给定一个非空二叉树, 返回一个由每层节点平均值组成的数组。示例

```
输入：
    3
   / \
  9  20
    /  \
   15   7
输出：[3, 14.5, 11]
解释：
第 0 层的平均值是 3 ,  第1层是 14.5 , 第2层是 11 。因此返回 [3, 14.5, 11] 。
```

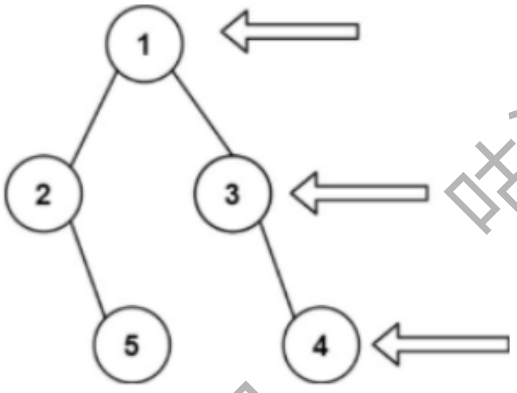
这个题和前面的几个一样，只不过是每层都先将元素保存下来，最后求平均就行了：

```
public List<Double> averageOfLevels(TreeNode root) {
        List<Double> res = new ArrayList<>();
        if (root == null) return res;
        Queue<TreeNode> list = new LinkedList<>();
        list.add(root);
        while (list.size() != 0){
            int len = list.size();
            double sum = 0;
            for (int i = 0; i < len; i++){
                TreeNode node = list.poll();
                sum += node.val;
                if (node.left != null) list.add(node.left);
                if (node.right != null) list.add(node.right);
            }
            res.add(sum/len);
        }
        return res;
    }
```

2.3.3 二叉树的右视图

LeetCode 199题目要求是：给定一个二叉树的根节点 root，想象自己站在它的右侧，按照从顶部到底部的顺序，返回从右侧所能看到的节点值。例如：

示例 1:




```
输入：[1,2,3,null,5,null,4]
输出：[1,3,4]
```

这个题目出现频率还挺高的，如果没有提前思考过，面试现场可能会想不到怎么做。其实也很简单那，利用 BFS 进行层次遍历，记录下每层的最后一个元素。

```
public List<Integer> rightSideView(TreeNode root) {
        List<Integer> res = new ArrayList<>();
        if (root == null) {
            return res;
        }
        Queue<TreeNode> queue = new LinkedList<>();
        queue.offer(root);
        while (!queue.isEmpty()) {
            int size = queue.size();
            for (int i = 0; i < size; i++) {
                TreeNode node = queue.poll();
                if (node.left != null) {
                    queue.offer(node.left);
                }
                if (node.right != null) {
                    queue.offer(node.right);
                }
                if (i == size - 1) {  //将当前层的最后一个节点放入结果列表
                    res.add(node.val);
                }
            }
        }
        return res;
    }
```

是不是很简单，这三题本质都是层次遍历的变形。

我们来造题

如果将右视图换成左视图呢？该问题作为本章的一个作业，请读者自行思考。

再思考，俯视图行不行？答案是不行的，那为什么不行呢，请读者思考。

3.树的深度优先遍历

3.1深入理解前中后序遍历

深度优先遍历有前中后三种情况，大部分人看过之后就能写出来，很遗憾大部分只是背下来的，稍微变换一下就废了。

我们再从二叉树的角度看递归，每次遇到递归，都按照前面说的四步来写，可以更好地写出正确的递归算法。通过二叉树可以非常方便的理解递归，递归只处理当前这一层和下一层之间的关系，并不关心下层和下下层之间的关系，这就像老子只管养好儿子，至于孙子怎么样，那是儿子的事，你也不能瞎掺合。具体来说，我们总结了四条行之有效的方法来分解递归：

- 第一步：从小到大递推
- 第二步：分情况讨论，明确结束条件
- 第三步：组合出完整方法
- 第四步：想验证，则从大到小画图推演

我们接下来就一步步看怎么做：

第一步：从小到大递推，分情况讨论

我们就以这个二叉树为例：

```
    3
   / \
  9  20
    /  \
   15   7
```

我们先选一个最小的子树：

```
     20
    /  \
   15   7
```

假如20为head，则此时前序访问顺序应该是：

```
void visit1(){
   list.add(root);//20被访问
   root.left;//继续访问15
   root.right; //继续访问7
   }
```

然后再向上访问，看node(3)的情况：

```
 void visit2(){
   list.add(root);//3被访问
   root.left;//继续访问，得到9
   root.right; //继续访问，得到20
 }
```

这里的20是一个子树的父节点，访问方式与上面的visit1()一样，所以我们可以直接合并到一起就是：

```
void visit(){
   list.add(root);//20被访问
   visit(root.left);//继续访问15
   visit(root.right); //继续访问7
}
```

这就是我们期待的递归方法。

第二步：分情况讨论，明确结束条件

上面有了递归的主体，但是这个递归什么时候结束呢？很明显应该是root=null的时候。一般来说链表和二叉树问题的终止条件都包含当前访问的元素为null。有些题目结束条件比较复杂，此时最好的方式就是先将所有可能的结束情况列举出来，然后整理一下就行了，这个我们后面在具体题目里再看。

第三步：组合出完整方法

到此为止，我们就能将完整代码写出来了，同时为了方便区分，我们将方法名换成preorder：

```
public void preorder(TreeNode root, List<Integer> res) {
       if (root == null) {
           return;
       }
       res.add(root.val);
       preorder(root.left, res);
       preorder(root.right, res);
   }
```

第四步 从大到小 画图推演

写完之后对不对呢？递归的方法是很难调试的，即使对的，你也可能晕，这里介绍一种简单有效的验证方法——调用过程图法。我们可以画个过程图看一下，因为是两个递归函数，如果比较复杂，我们可以少画几组。

递归的特征是“不撞南墙不回头”，一定是在执行到某个root=null了才开始返回，下图中的序号就是递归的完整过程：

图1 前序遍历的递归过程图

从图中可以看到，当root的一个子树为null的时候还是会执行递归，进入之后发现root==null了，然后就开始返回。这里我们要特别注意res.add()的时机对不对，将其进入顺序依次写出来就是需要的结果。该过程明确之后再debug就容易很多，刚开始学习递归建议多画几次，熟悉之后就不必再画了。

注意

另外注意一个问题，有时候题目给的方法不方便直接递归，我们需要再写一个自己的方法包一下递归过程。例如上面前序遍历的问题，题干给的是这样的结构，但是我们希望在每次递归过程里处理res，此时可以自己创建一个preOrderRecur()方法。因此完整代码如下：

```
class Solution {
    public List<Integer> preorderTraversal(TreeNode root) {
        List<Integer> res = new ArrayList<Integer>();
        preOrderRecur(root, res);
        return res;
    }

public  void preOrderRecur(TreeNode head,List<Integer> res) {
    if (head == null) {
        return;
    }
    res.add(head);
```

```
    preOrderRecur(head.left);
    preOrderRecur(head.right);
 }
}
```

前序遍历写出来之后，中序和后序遍历就不难理解了，中序是左中右，后序是左右中。代码如下：

```
public static void inOrderRecur(TreeNode head) {
    if (head == null) {
        return;
    }
    inOrderRecur(head.left);
    System.out.print(head.value + " ");
    inOrderRecur(head.right);
}
```

后序：

```
public static void postOrderRecur(TreeNode head) {
    if (head == null) {
        return;
    }
    postOrderRecur(head.left);
    postOrderRecur(head.right);
    System.out.print(head.value + " ");
}
```

3.2 迭代法实现二叉树的三种遍历

理论上，递归能做的迭代一定能做，但可能会比较复杂。上面几个递归的遍历方法，背都背下来了，所以有时候面试官要求不使用递归实现三种遍历，

递归就是每次执行方法调用都会先把当前的局部变量、参数值和返回地址等压入栈中，后面在递归返回的时候，从栈顶弹出上一层的各项参数继续执行，这就是递归为什么可以自动返回并执行上一层方法的原因。
我们就用迭代法再次练习三道题：144.二叉树的前序遍历 94.二叉树的中序遍历 145.二叉树的后序遍历。

3.2.1 迭代法实现前序遍历

前序遍历是中左右，如果还有左子树就一直向下找。完了之后再返回从最底层逐步向上向右找。 不难写出如下代码: (注意代码中空节点不入栈)

```
    public List<Integer> preOrderTraversal(TreeNode root) {
        List<Integer> res = new ArrayList<Integer>();
        if (root == null) {
            return res;
        }

        Deque<TreeNode> stack = new LinkedList<TreeNode>();
```

```
        TreeNode node = root;
        while (!stack.isEmpty() || node != null) {
            while (node != null) {
                res.add(node.val);
                stack.push(node);
                node = node.left;
            }
            node = stack.pop();
            node = node.right;
        }
        return res;
    }
```

此时会发现貌似使用迭代法写出前序遍历并不复杂，我们继续看中序遍历：

3.2.2 迭代法实现中序遍历

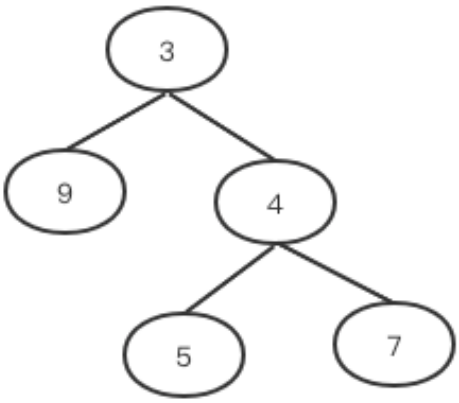
再看中序遍历，中序遍历是左中右，先访问的是二叉树左子树的节点，然后一层一层向下访问，直到到达树左面的最底部，再开始处理节点(也就是在把节点的数值放进res列表中)。在使用迭代法写中序遍历，就需要借用指针的遍历来帮助访问节点，栈则用来处理节点上的元素。 看代码：

```
    public List<Integer> inorderTraversal(TreeNode root) {
        List<Integer> res = new ArrayList<Integer>();
        Deque<TreeNode> stack = new LinkedList<TreeNode>();
        while (root != null || !stack.isEmpty()) {
            while (root != null) {
                stack.push(root);
                root = root.left;
            }
            root = stack.pop();
            res.add(root.val);
            root = root.right;
        }
        return res;
    }
```

3.2.3 迭代法实现后序遍历

后序遍历的非递归实现有三种基本的思路：反转法、访问标记法、和Morris法，可惜三种理解起来都有些难度，如果头发不够，可以先将后面的学完最后再看本小节。其中Morris法是一个老外发明的巧妙思想：不使用栈，而是用好树中的null指针。但是实现后序仍然非常麻烦，我们这里不再展开，感兴趣的同学可以查一下，我们这里只介绍一种：反转法。

如下图，我们先观察后序遍历的结果是seq={9 5 7 4 3}，如果我们将其整体反转的话就是new_seq={3 4 7 5 9}。



后序序列：9 5 7 4 3
将后序反转: 3 4 7 5 9

你有没有发现要得到new_seq的方法和前序遍历几乎一致，前序是先中间，再左边然后右边，而这里是先中间，再后边然后左边。那我们完全可以改造一下前序遍历，得到序列new_seq之后再reverse一下就是想要的结果了，代码如下：

```
public  List<Integer> postOrderTraversal(TreeNode root) {
        List<Integer> res = new ArrayList<>();
        if (root == null) return res;
        Stack<TreeNode> stack = new Stack<>();
        TreeNode node = root;
        while (!stack.isEmpty() || node != null) {
            while (node != null) {
                res.add(node.val);
                stack.push(node);
                node = node.right;
            }
            node = stack.pop();
            node = node.left;
        }
        Collections.reverse(res);
        return res;
    }
```

这个方法可以巧妙的避开直接后序时遇到的坑，那直接按照后序的规则写有难写问题呢？感兴趣的同学可以自己研究一下。

3.3 深度和高度专题

给定二叉树 [3,9,20,null,null,15,7]，如下图

```
    3
   / \
  9  20
    /  \
   15   7
```

然后LeetCode给我们造了一堆的题目，现在一起来研究一下104、110和111三个题，这三个题看起来挺像的，都是关于深度、高度的。

3.3.1 最大深度问题

首先看一下104题最大深度:给定一个二叉树，找出其最大深度。二叉树的深度为根节点到最远叶子节点的最长路径上的节点数。说明: 叶子节点是指没有子节点的节点。例如上面的例子返回结果为最大深度为3。

我们先看一个简单情况：

```
    3           3    3
   / \         /      \
  9  20       9        20
```

对于node(3)，最大深度自然是左右子结点+1，左右子节点有的可能为空，只要有一个，树的最大高度就是1+1=2。然后再增加几个结点：

很显然相对于node(20)，最大深度自然是左右子结点+1，左右子节点有的可能为空，只要有一个，树的最大高度就是1+1=2，用代码表示就是：

```
int depth = 1 + max(leftDepth, rightDepth);
```

而对于3，则是左右子树深度最大的那个然后再+1，具体谁更大，则不必关心。所以对于node(3)的判断逻辑就是：

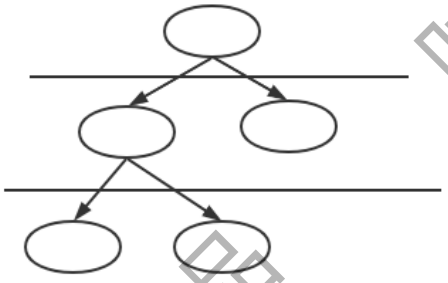
```
int leftDepth = getDepth(node.left); // 左
int rightDepth = getDepth(node.right); // 右
int depth = 1 + max(leftDepth, rightDepth); // 中
```

那什么时候结束呢，这里仍然是root == null返回0就行了。至于入参，自然是要处理的子树的根节点root，而返回值则是当前root所在子树的最大高度。所以合在一起就是：

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

上面代码先拿到左右子树的结果再计算Math.max(left，right) + 1，这与后序遍历本质上一样的，因此可以看做后序遍历的拓展问题。

我们继续分析这个题，如果确定树最大有几层是不是也就知道最大深度了？ 是的，直接套用层序遍历的代码就可以。



具体做法是：我们修改2.2层次遍历中的分层方法，每获得一层增加一下技术就可以了。

```
public int maxDepth(TreeNode root) {
        if (root == null) {
            return 0;
        }
        Queue<TreeNode> queue = new LinkedList<TreeNode>();
        queue.offer(root);
        int ans = 0;
        while (!queue.isEmpty()) {
        //size表示某一层的所有元素数
            int size = queue.size();
            //size=0 表示一层访问完了
            while (size > 0) {
                TreeNode node = queue.poll();
                if (node.left != null) {
                    queue.offer(node.left);
                }
                if (node.right != null) {
                    queue.offer(node.right);
                }
                size--;
            }
            ans++;
        }
        return ans;
    }
```

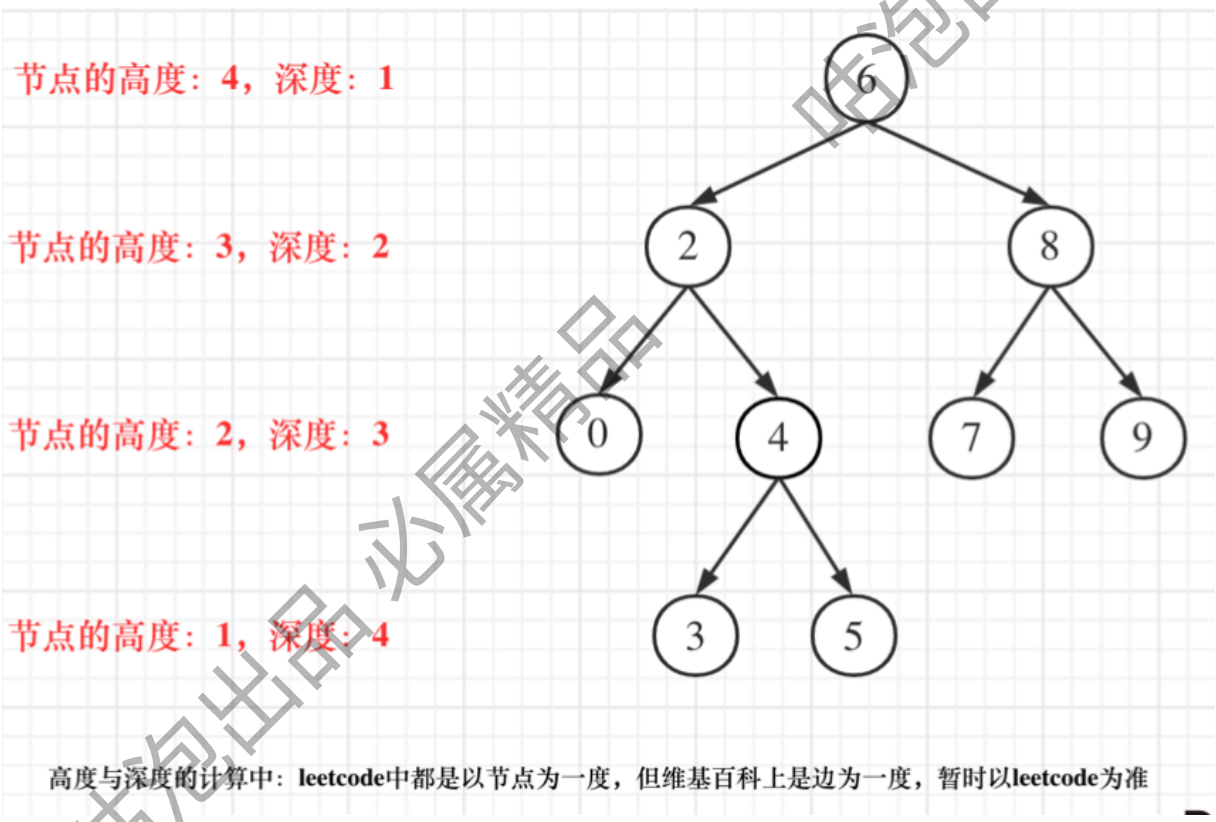
3.3.2 判断平衡树

LeetCode110 判断平衡二叉树：给定一个二叉树，判断它是否是高度平衡的二叉树。本题中，一棵高度平衡二叉树定义为：一个二叉树每个节点 的左右两个子树的高度差的绝对值不超过 1 。

这里的要求是二叉树每个节点的左右两个子树的高度差的绝对值不超过 1。先补充一个问题，高度和深度怎么区分呢?

- 二叉树节点的深度:指从根节点到该节点的最长简单路径边的条数。
- 二叉树节点的高度:指从该节点到叶子节点的最长简单路径边的条数。

直接看图:



关于根节点的深度究竟是1 还是 0，不同的地方有不一样的标准，leetcode的题目中都是以根节点深度是1，其他的就不管了。

言归正传，我们仍然先看一个最简单的情况:

```
  3            3     3
 / \          /       \
9   20       9         20
```

很显然只有两层的时候一定是平衡的，因为对于node(3)，左右孩子如果只有一个，那高度差就是1；如果左右子孩子都有或者都没有，则高度差为0。如果增加一层会怎么样呢？如下图:

```
    3
   / \
  9   20
     /  \
    15   7
```

```
    3
   / \
  9   20
     /
    15
```

```
    3
   / \
  9   20
       \
        7
```

```
    3
     \
      20
     /  \
    15   7
```

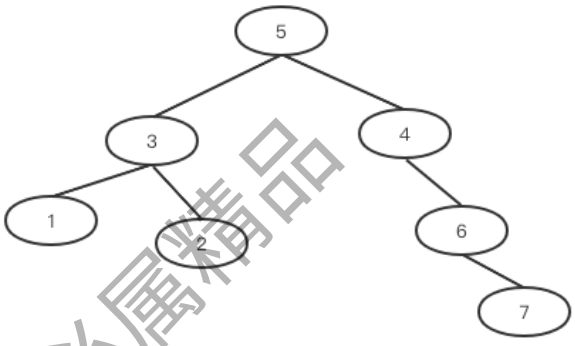
对于node(3)，需要同时知道自己左右子树的最大高度差是否<2。

- 当节点root 左 / 右子树的高度差 < 2，则返回节点 root 的左右子树中最大高度加 1 （ max(left, right) + 1 ）；参考上面的高度和深度的对比图思考一下，这里为什么是最大高度?
- 当节点root 左 / 右子树的高度差 ≥2 ： 则返回 -1 ，代表 此子树不是平衡树 。

也就是：

```
int left = recur(root.left);
int right = recur(root.right);
return Math.abs(left - right) < 2 ? Math.max(left, right) + 1 : -1;
```

我们此时就写出了核心递归。假如子树已经不平衡了，则不需要再递归了直接返回就行，比如这个树中结点4：



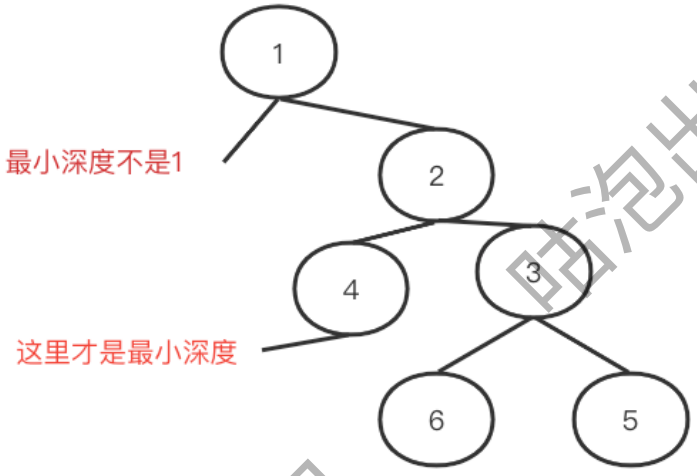
综合考虑几种情况，完整的代码如下：

```
int recur(TreeNode root) {
      if (root == null)
          return 0;
      int left = recur(root.left);
      if(left == -1)
            return -1;
      int right = recur(root.right);
      if(right == -1)
          return -1;
      return Math.abs(left - right) < 2 ? Math.max(left, right) + 1 : -1;
  }
```

3.3.3 最小深度

既然有最大深度，那是否有最小深度呢？LeetCode111就是：给定一个二叉树，找出其最小深度。最小深度是从根节点到最近叶子节点的最短路径上的节点数量，例如下面的例子返回结果为2。**说明：**叶子节点是指没有子节点的节点。

前两个题都涉及最大深度，那将Max改成Min能不能解决本题呢？不行！注意下图:



这里的关键问题是题目中说的:最小深度是从根节点到最近叶子节点的最短路径上的节点数量，也就是最小深度的一层必须要有叶子结点，因此不能直接用。

这里的核心问题仍然是分析终止条件：

- 如果左子树为空，右子树不为空，说明最小深度是 1 + 右子树的深度。
- 反之，右子树为空，左子树不为空，最小深度是 1 + 左子树的深度。

最后如果左右子树都不为空，返回左右子树深 度最小值 + 1 。

```
public int minDepth(TreeNode root) {
        if (root == null) {
            return 0;
        }

        if (root.left == null && root.right == null) {
            return 1;
        }
//这是一种常用的技巧，最小值初始化为最大，最大值初始化为0
        int min_depth = Integer.MAX_VALUE;
        if (root.left != null) {
            min_depth = Math.min(minDepth(root.left), min_depth);
        }
        if (root.right != null) {
            min_depth = Math.min(minDepth(root.right), min_depth);
        }
        return min_depth + 1;
    }
```

除了递归方式 ，我们也可以使用层次遍历，只要遍历时，第一次遇到叶子就直接返回其所在的层次即可，改一下2.2中的方法即可。

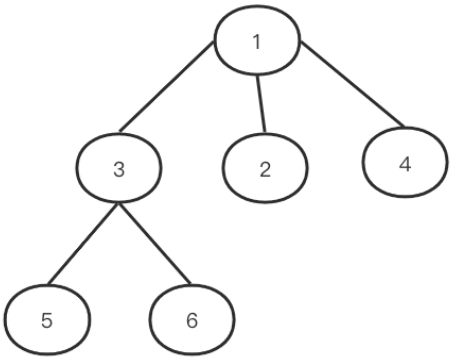
```
    public int minDepth(TreeNode root) {
        if (root == null) {
            return 0;
        }
        int minDepth = 0;
```

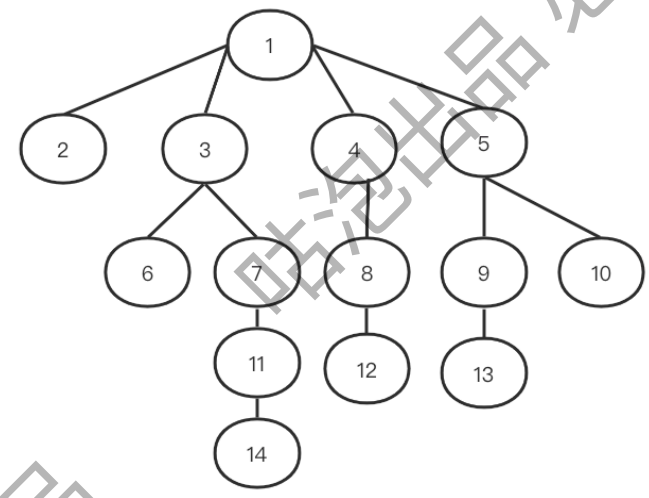
```
        LinkedList<TreeNode> queue = new LinkedList<TreeNode>();
        queue.add(root);
        while (queue.size() > 0) {
            //获取当前队列的长度，这个长度相当于 当前这一层的节点个数
            int size = queue.size();
            minDepth++;
            for (int i = 0; i < size; ++i) {
                TreeNode t = queue.remove();
                if (t.left == null && t.right == null) {
                    return minDepth;
                }
                if (t.left != null) {
                    queue.add(t.left);
                }
                if (t.right != null) {
                    queue.add(t.right);
                }
            }
        }
        return 0;
    }
```

3.3.4 N叉树的最大深度

如果将二叉树换成N叉树又该怎么做呢？这就是LeetCode559.给定一个 N 叉树，找到其最大深度。最大深度是指从根节点到最远叶子节点的最长路径上的节点总数。N 叉树输入按层序遍历序列化表示，每组子节点由空值分隔（请参见示例）。

```
示例1：
输入：root = [1,null,3,2,4,null,5,6]
输出：3
示例2：
输入：root =
[1,null,2,3,4,5,null,null,6,7,null,8,null,9,10,null,null,11,null,12,null,13,null,null,14]
输出：5
```





这道题就是将二叉树换成了N叉树，不同点就在于N叉树结点比较多，我们使用List存，遍历时用for即可，我们先看N叉树的定义：

```
class Node {
    public int val;
    public List<Node> children;

    public Node() {}

    public Node(int _val) {
        val = _val;
    }

    public Node(int _val, List<Node> _children) {
        val = _val;
        children = _children;
    }
}
```

这个题的实现和上一个题的最大区别就是处理子树时加了个处理List的for循环

```
public int maxDepth(Node root) {
    if (root == null) {
      return 0;
    } else if (root.children.isEmpty()) {
      return 1;
    } else {
      List<Integer> heights = new LinkedList<>();
      for (Node item : root.children) {
        heights.add(maxDepth(item));
      }
      return Collections.max(heights) + 1;
    }
  }
```

本题也可以使用层次遍历的方法，这个作为拓展问题请感兴趣的同学研究一下。

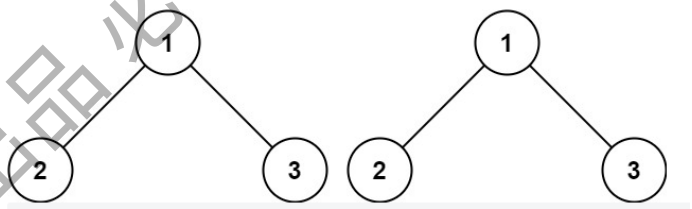
3.4 二叉树里的双指针

所谓的双指针就是定义了两个变量，在二叉树中有时候也需要至少定义两个变量才能解决问题，这两个指针可能针对一棵树，也可能针对两棵树，我们姑且也称之为“双指针”吧。这些问题一般是与对称、反转和合并等类型相关，我们接下来就看一下LeetCode100、101、226和617四道高频问题。

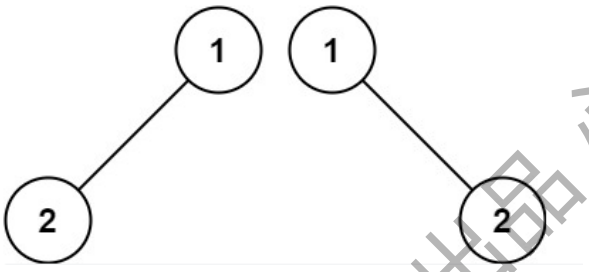
3.4.1 判断两棵树是否相同

LeetCode100：给你两棵二叉树的根节点 `p` 和 `q` ，编写一个函数来检验这两棵树是否相同。如果两个树在结构上相同，并且节点具有相同的值，则认为它们是相同的。

```
示例1：
输入：p = [1,2,3], q = [1,2,3]
输出：true
```




```
示例2：
输入：p = [1,2], q = [1,null,2]
输出：false
```



这个貌似就是两个二叉树同时进行前序遍历，先判断根节点是否相同， 如果相同再分别判断左右子节点是否相同，判断的过程中只要有一个不相同就返回 false，如果全部相同才会返回true。其实就是这么回事。看代码：

```
public boolean isSameTree(TreeNode p, TreeNode q) {
        //如果都为空我们就认为他是相同的
        if (p == null && q == null)
            return true;
        //如果一个为空，一个不为空，很明显不可能是相同的树，直接返回false即可
        if (p == null || q == null)
            return false;
        //如果对应位置的两个结点的值不相等，自然也不是同一个棵树
        if (p.val != q.val)
```

```
            return false;

        //走到这一步说明节点p和q是完全相同的，接下来需要再判断其左左和右右是否满足要求了
        return isSameTree(p.left, q.left) && isSameTree(p.right, q.right);
    }
```

本题除了上述方式，还可以使用广度优先，感兴趣的同学可以试一试。

3.4.2 对称二叉树

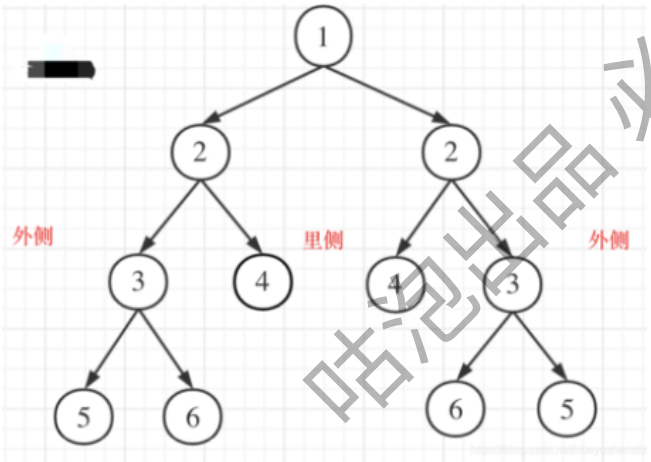
LeetCode101 给定一个二叉树，检查它是否是镜像对称的。例如下面这个就是对称二叉树：

```
    1
   / \
  2   2
 / \ / \
3  4 4  3
```

但是下面这个 `[1,2,2,null,3,null,3]` 则不是对称的:

```
   1
   / \
  2   2
   \   \
   3    3
```

如果树是镜像的，下面这个图更直观一些：



因为我们要通过递归函数的返回值来判断两个子树的内侧节点和外侧节点是否相等，所以准确的来说是一个树的遍历顺序是左右中，一 个树的遍历顺序是右左中。这里的关键还是如何比较和如何处理结束条件。 单层递归的逻辑就是处理左右节点都不为空，且数值相同的情况。

1. 比较二叉树外侧是否对称:传入的是左节点的左孩子，右节点的右孩子。
2. 比较 内侧是否对称，传入左节点的右孩子，右节点的左孩子。
3. 如果左右都对称就返回true ，有一侧不对称就返回false 。

接下来就是合并和进一步简化：

```
class Solution {
     //主方法
    public boolean isSymmetric(TreeNode root) {
      if(root==null){
         return true;
      }
     return check(root.left, root.right);
    }

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

3.4.3 合并二叉树

LeetCode617.给定两个二叉树，想象当你将它们中的一个覆盖到另一个上时，两个二叉树的一些节点便会重叠。你需要将他们合并为一个新的二叉树。合并的规则是如果两个节点重叠，那么将他们的值相加作为节点合并后的新值，否则不为 NULL 的节点将直接作为新二叉树的节点。

```
示例 1:
输入:
  Tree 1                     Tree 2
          1                         2
         / \                       / \
        3   2                     1   3
       /                           \   \
      5                             4   7
输出合并后的树:
        3
       / \
      4   5
     / \   \
    5   4   7
```

两个二叉树的对应节点可能存在以下三种情况，对于每种情况使用不同的合并方式。

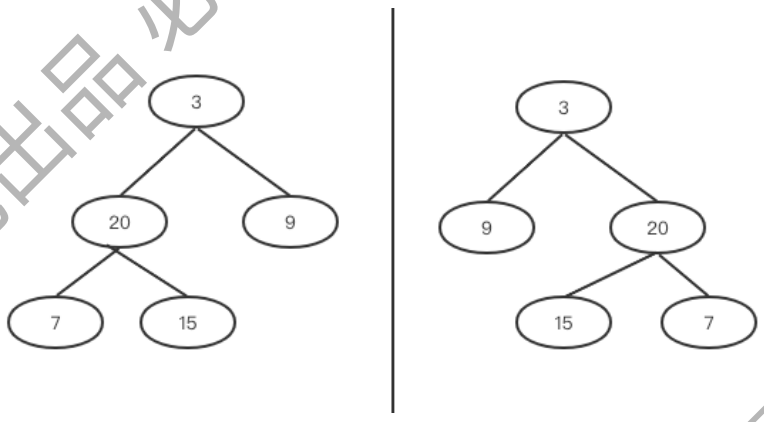
- 如果两个二叉树的对应节点都为空，则合并后的二叉树的对应节点也为空；
- 如果两个二叉树的对应节点只有一个为空，则合并后的二叉树的对应节点为其中的非空节点；
- 如果两个二叉树的对应节点都不为空，则合并后的二叉树的对应节点的值为两个二叉树的对应节点的值之和，此时需要显性合并两个节点。

对一个节点进行合并之后，还要对该节点的左右子树分别进行合并：

```
    public TreeNode mergeTrees(TreeNode t1, TreeNode t2) {
        if (t1 == null) {
            return t2;
        }
        if (t2 == null) {
            return t1;
        }
        TreeNode merged = new TreeNode(t1.val + t2.val);
        merged.left = mergeTrees(t1.left, t2.left);
        merged.right = mergeTrees(t1.right, t2.right);
        return merged;
    }
```

如果这个感觉还是想不明白，可以直接对照题干给的例子来验证一下。

最后我们来造个题：前面我们研究了两棵树相等和一棵树对称的情况，我们可以造一道题，判断两棵树是否对称的。如下就是一个对称的二叉树，那该如何写代码实现呢？请你思考。



3.5 路径专题

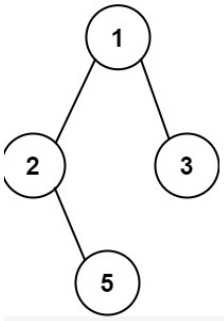
关于二叉树有几道与路径有关的题目，我们统一看一下。初次接触你会感觉有些难，但是这是在为回溯打基础，因为很多回溯问题就是在找路径，甚至要找多条路径。

3.5.1 二叉树的所有路径

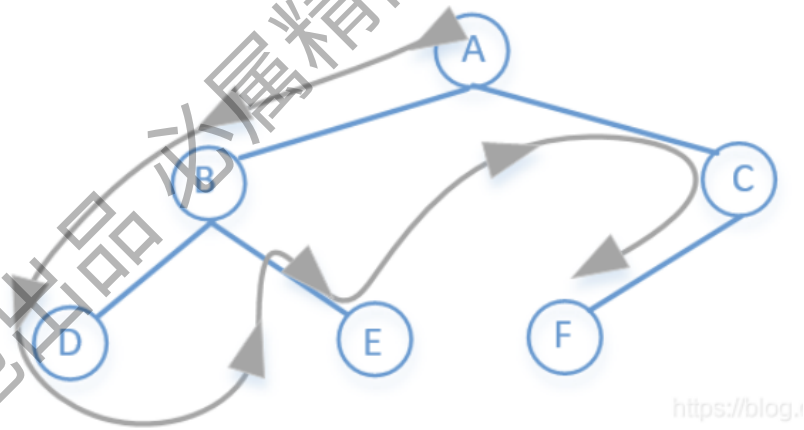
LeetCode257：给你一个二叉树的根节点 `root` ，按 **任意顺序** ，返回所有从根节点到叶子节点的路径。

叶子节点是指没有子节点的节点。

```
示例：
输入：root = [1,2,3,null,5]
输出：["1->2->5","1->3"]
```



我们可以注意到有几个叶子节点，就有几条路径，那如何找叶子节点呢？我们知道深度优先搜索就是从根节点开始一直找到叶子结点，我们这里可以先判断当前节点是不是叶子结点，再决定是不是向下走，如果是叶子结点，我们就增加一条路径，就像下面图中这样：



这里还有个问题，当得到一个叶子结点容易，那这时候怎么知道它所在的完整路径是什么呢？例如上图中得到D之后，怎么知道其前面的A和B呢？简单，增加一个String类型的变量中，访问每个节点访问的时候先存到String中，到叶子节点的时候再添加到集合里：

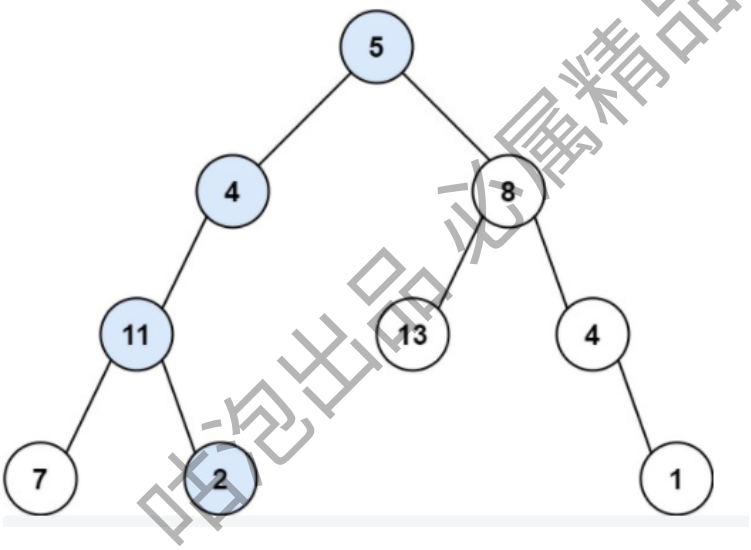
```
public List<String> binaryTreePaths(TreeNode root) {
    List<String> res = new ArrayList<>();
    dfs(root, "", res);
    return res;
}

private void dfs(TreeNode root, String path, List<String> res) {
    //如果为空，直接返回
    if (root == null)
        return;
    //如果是叶子节点，说明找到了一条路径，把它加入到res中
    if (root.left == null && root.right == null) {
        res.add(path + root.val);
        return;
    }
    //如果不是叶子节点，在分别遍历他的左右子节点
    dfs(root.left, path + root.val + "->", res);
    dfs(root.right, path + root.val + "->", res);
}
```

本题还可以使用层次遍历、回溯等方法解决。本题可以作为回溯的入门问题，我们到回溯时再讲解。

3.5.2 路径总和

上面我们讨论的找所有路径的方法，那我们是否可以再找一下哪条路径的和为目标值呢？这就是LeetCode112题：给你二叉树的根节点 root 和一个表示目标和的整数 targetSum ，判断该树中是否存在 根节点到叶子节点 的路径，这条路径上所有节点值相加等于目标和 targetSum。叶子节点 是指没有子节点的节点。

```
示例1：
输入：root = [5,4,8,11,null,13,4,7,2,null,null,null,1], targetSum = 22
输出：true
```




```
示例2：
输入：root = [1,2,3], targetSum = 5
输出：false
```

本题询问是否存在从当前节点 root 到叶子节点的路径，满足其路径和为 sum，假定从根节点到当前节点的值之和为 val，我们可以将这个大问题转化为一个小问题：是否存在从当前节点的子节点到叶子的路径，满足其路径和为 sum - val。

不难发现这满足递归的性质，若当前节点就是叶子节点，那么我们直接判断 sum 是否等于 val 即可（因为路径和已经确定，就是当前节点的值，我们只需要判断该路径和是否满足条件）。若当前节点不是叶子节点，我们只需要递归地询问它的子节点是否能满足条件即可。

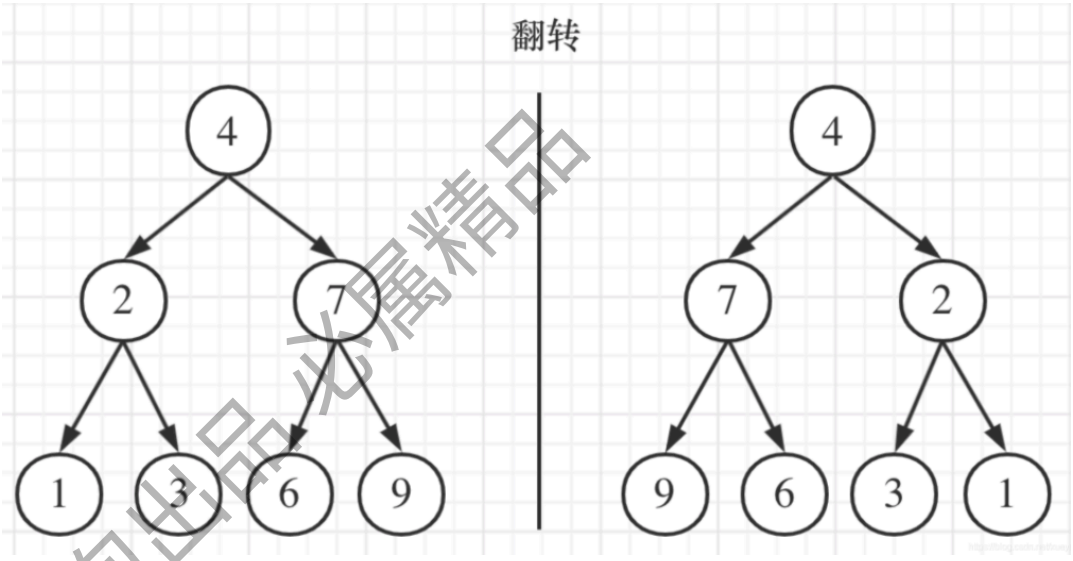
```
public boolean hasPathSum(TreeNode root, int sum) {
       if (root == null) {
           return false;
       }
       if (root.left == null && root.right == null) {
           return sum == root.val;
       }
       return hasPathSum(root.left, sum - root.val) || hasPathSum(root.right, sum -
root.val);
   }
```

本题还有个拓展，既然找到是否存在路径了，那能把找到的路径打印出来吗？当然可以，这就是LeetCode113题，我们将本题做为入门题在回溯部分讲解。

3.6 翻转的妙用

3.6.1 翻转二叉树

首先来看 LeetCode226 翻转二叉树，将二叉树整体反转。如下图所示：



这个题也是剑指offer27题的要求，根据上图，可以发现想要翻转树，就是把每一个节点的左右孩子交换一下。关键在于遍历顺序，前中后序应该选哪一种遍历顺序。遍历的过程中去翻转每一个节点的左右孩子就可以达到整体翻转的效果。注意只要把每一个节点的左右孩子翻转一下，就可以达到整体翻转的效果。

这是一道很经典的二叉树问题。显然，我们从根节点开始，递归地对树进行遍历，并从叶子节点先开始翻转。如果当前遍历到的节点 root 的左右两棵子树都已经翻转，那么我们只需要交换两棵子树的位置，即可完成以 root 为根节点的整棵子树的翻转。

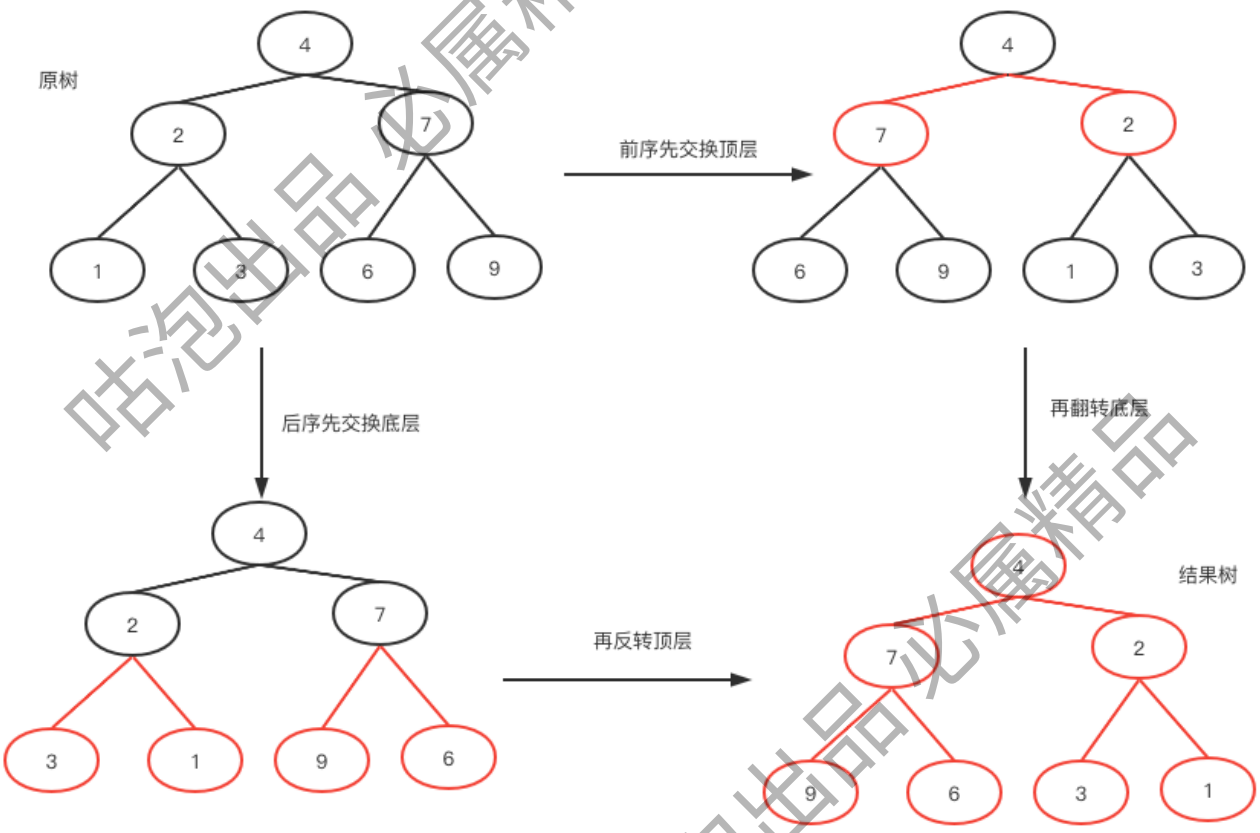
先看前序交换：

```
    public TreeNode invertTree(TreeNode root) {
         if (root == null) {
            return null;
        }
        TreeNode temp=root.left;
        root.left=root.right;
        root.right=temp;

        TreeNode left = invertTree(root.left);
        TreeNode right = invertTree(root.right);
        return root;
    }
```

再看后序：

```
public TreeNode invertTree(TreeNode root) {
        if (root == null) {
            return null;
        }
        TreeNode left = invertTree(root.left);
        TreeNode right = invertTree(root.right);
        root.left = right;
        root.right = left;
        return root;
    }
```

这道题目使用前序遍历和后序遍历都可以，主要区别是是前序是先处理当前节点再处理子节点，是自顶向下，后序是先处理子结点最后处理自己，一个是自下而上的。观察下图就明白了：



本题还可以使用层次遍历实现，核心思想是元素出队时，先将其左右两个孩子不是直接入队，而是先反转再放进去，代码如下：

```
class Solution {
    public TreeNode invertTree(TreeNode root) {
    LinkedList<TreeNode> que = new LinkedList<TreeNode>();
    if (root == null)
        return null;
     que.add(root);
    while (que.size()>0) {
     int size = que.size();
      for (int i = 0; i < size; i++) {
         TreeNode node = que.remove();
```

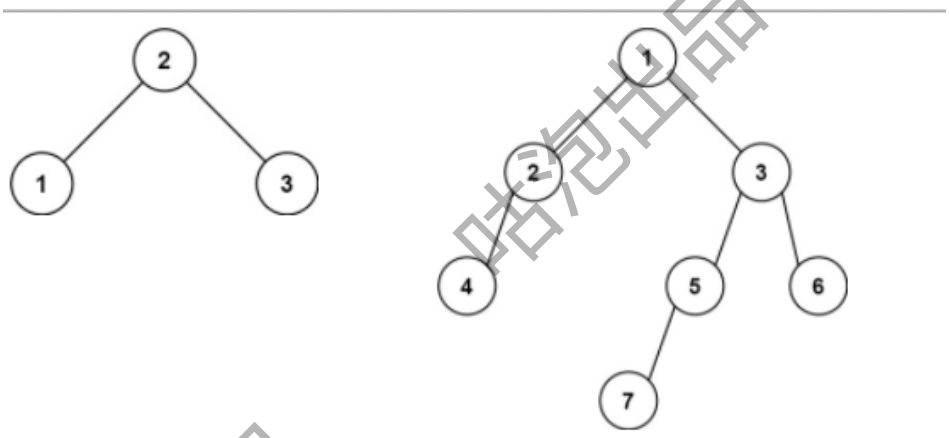
```
            //这一行就是 精华所在
             swap(node.left, node.right); // 节点处理
             if (node.left!=null)
                 que.add(node.left);
             if (node.right!=null)
                 que.add(node.right);
               }
            }
            return root;
        }
        public static void swap(TreeNode left,TreeNode right){
            TreeNode tmp=left;
            left=right;
            right=tmp;
        }
    }
```

3.6.2 最底层最左边

上面这个层次遍历的思想可以方便的解决剑指 Offer II 045. 二叉树最底层最左边的值的问题：给定一个二叉树的 **根节点** `root`，请找出该二叉树的 **最底层 最左边** 节点的值。

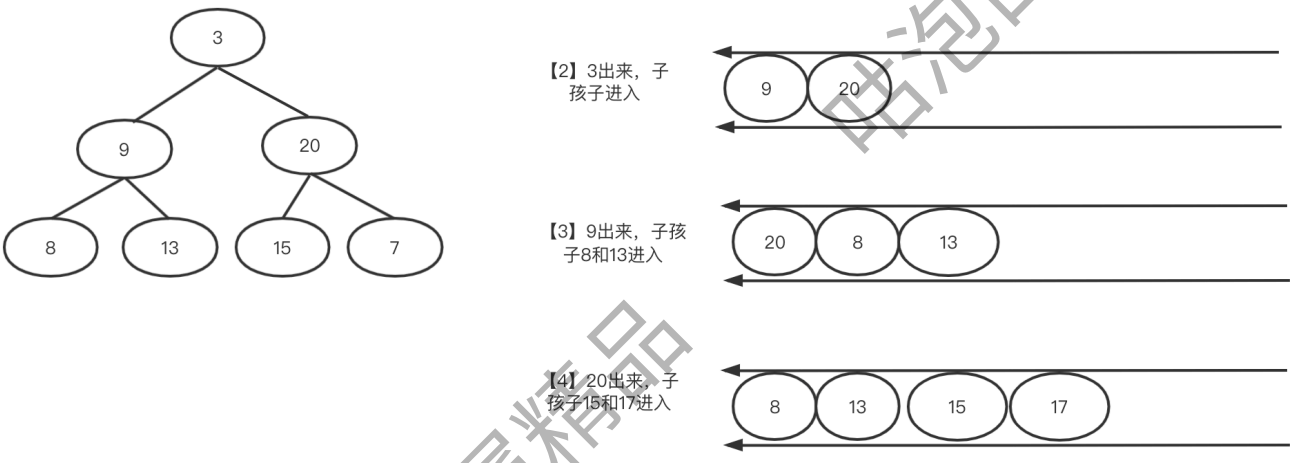
假设二叉树中至少有一个节点。

```
示例1：
输入：root = [2,1,3]
输出：1

示例2：
输入：[1,2,3,4,null,5,6,null,null,7]
输出：7
```



我们在第二章介绍了很多次如何使用层次遍历，这里有两个问题：该怎么知道什么时候到了最底层呢？假如最底层有两个，该怎么知道哪个是最左的呢？

我们继续观察层次遍历的执行过程：



我们可以发现，正常执行层次遍历，不管最底层有几个元素，最后一个输出的一定是是最底层最右的元素7，那这里我们就想了，能否将该处理与上一次题的翻转结合一下，每一层都是先反转再放入队列，就可以让最后一个输出的是最左的呢？是的，这就是解决本题的关键。

```
public int findBottomLeftValue(TreeNode root) {
        if (root.left == null && root.right == null) {
            return root.val;
        }

        Queue<TreeNode> queue = new LinkedList<>();
        queue.offer(root);
        TreeNode temp = new TreeNode(-100);

        while (!queue.isEmpty()) {
            temp = queue.poll();
            if (temp.right != null) {
                // 先把右节点加入 queue
                queue.offer(temp.right);
            }
            if (temp.left != null) {
                // 再把左节点加入 queue
                queue.offer(temp.left);
            }
        }
        return temp.val;
    }
```

本结我们分析了十几道深度优先遍历的题目，研究的时候你会发现所有的题目都是来自几种遍历的拓展，只不过结束条件和处理方式存在区别，除此之外还有一些质量不错的二叉树的题目，例如LeetCode404.计算给定二叉树的所有左叶子之和等等，感兴趣的同学可以研究一下。

4. 中序遍历与搜索树

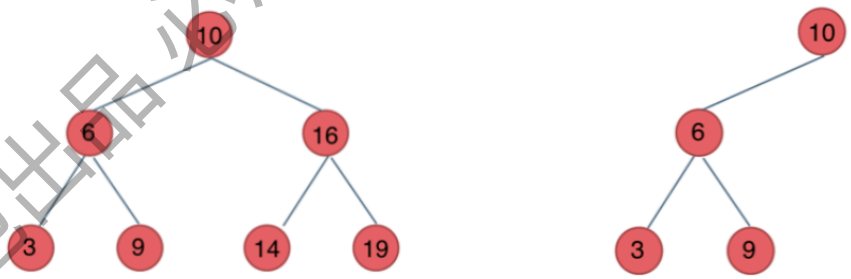
在第三章我们发现很多题使用前序、后序或者层次遍历都可以解决，但几乎没有中序遍历的。这是因为中序与前后序相比有不一样的特征，例如中序可以和搜索树结合在一起，但是前后序则不行。

4.1 中序与搜索树

二叉搜索树是一个很简单的概念，但是想说清楚却不太容易。简单来说就是如果一棵二叉树是搜索树，则按照中序遍历其序列正好是一个递增序列。比较规范的定义是：

- 若它的左子树不空，则左子树上所有节点的值均小于它的根节点的值;
- 若它的右子树不空，则右子树上所有节点的值均大于它的根节点的值;
- 它的左、右子树也分别为二叉排序树。

  下面这两棵树一个序列是{3,6,9,10,14,16,19}，一个是{3,6,9,10}，因此都是搜索树：



搜索树的题目虽然也是用递归，但是与前后序有很大区别，主要是因为搜索树是有序的，就可以根据条件决定某些递归就不必执行了，这也称为“剪枝”。

4.2 二叉搜索树中搜索特定值

LeetCode 700.给定二叉搜索树（BST）的根节点和一个值。 你需要在BST中找到节点值等于给定值的节点。 返回以该节点为根的子树。 如果节点不存在，则返回 NULL。例如：

```
target为2，给定二叉搜索树：
        4
       / \
      2   7
     / \
    1   3
```

你应该返回如下子树:

```
      2
     / \
    1   3
```

本题看起来很复杂，但是实现非常简单，递归：

- 如果根节点为空 root == null 或者根节点的值等于搜索值 val == root.val，返回根节点。

- 如果 val < root.val，进入根节点的左子树查找 searchBST(root.left, val)。
- 如果 val > root.val，进入根节点的右子树查找 searchBST(root.right, val)。

```
public TreeNode searchBST(TreeNode root, int val) {
  if (root == null || val == root.val) return root;
  return val < root.val ? searchBST(root.left, val) : searchBST(root.right, val);
}
```

如果采用迭代方式，也不复杂：

- 如果根节点不空 root != null 且根节点不是目的节点 val != root.val：
  - 如果 val < root.val，进入根节点的左子树查找 root = root.left。
  - 如果 val > root.val，进入根节点的右子树查找 root = root.right。

```
public TreeNode searchBST(TreeNode root, int val) {
   while (root != null && val != root.val)
     root = val < root.val ? root.left : root.right;
   return root;
}
```

4.3 验证二叉搜索树

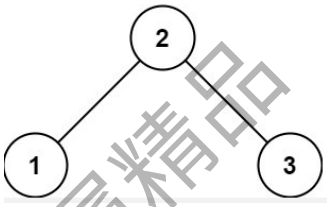
LeetCode98.给你一个二叉树的根节点 root ， 判断其是否是一个有效的二叉搜索树。

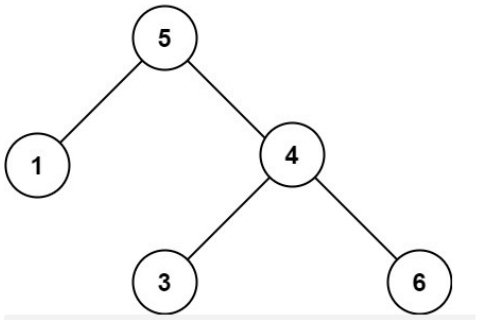
有效 二叉搜索树定义如下：

- 节点的左子树只包含 小于 当前节点的数。
- 节点的右子树只包含 大于 当前节点的数。
- 所有左子树和右子树自身必须也是二叉搜索树。

示例：

```
示例1：
输入：root = [2,1,3]
输出：true

示例2：
输入：root = [5,1,4,null,null,3,6]
输出：false
解释：根节点的值是 5 ，但是右子节点的值是 4 。
```





根据题目给出的性质，我们可以进一步知道二叉搜索树「中序遍历」得到的值构成的序列一定是升序的，在中序遍历的时候实时检查当前节点的值是否大于前一个中序遍历到的节点的值即可。

```
 public boolean isValidBST(TreeNode root) {
        Deque<TreeNode> stack = new LinkedList<TreeNode>();
        double inorder = 0;
        while (!stack.isEmpty() || root != null) {
            while (root != null) {
                stack.push(root);
                root = root.left;
            }
            root = stack.pop();
              // 如果中序遍历得到的节点的值小于等于前一个 inorder，说明不是二叉搜索树
            if (root.val <= inorder) {
                return false;
            }
            inorder = root.val;
            root = root.right;
        }
        return true;
    }
```

如果使用递归，实现方式也类似：

```
class Solution {
    long pre = Long.MIN_VALUE;
    public boolean isValidBST(TreeNode root) {
        if (root == null) {
            return true;
        }
        // 如果左子树下某个元素不满足要求，则退出
        if (!isValidBST(root.left)) {
            return false;
        }
        // 访问当前节点：如果当前节点小于等于中序遍历的前一个节点，说明不满足BST，返回 false；否则继续遍历。
        if (root.val <= pre) {
            return false;
        }
        pre = root.val;
        // 访问右子树
        return isValidBST(root.right);
    }
}
```

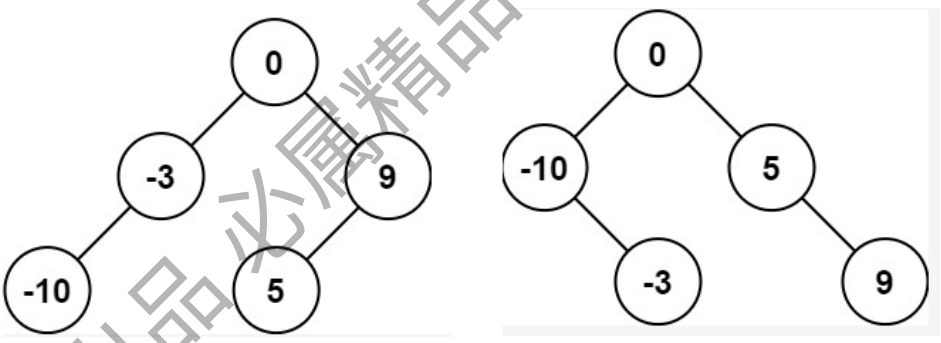
如果这个题理解了，可以继续研究LeetCode530.二叉搜索树的最小绝对差和LeetCode501.二叉搜索树中的众数两个题。

4.4 有序数组转为二叉搜索树

LeetCode108 给你一个整数数组 nums ，其中元素已经按 升序 排列，请你将其转换为一棵 高度平衡 二叉搜索树。

高度平衡 二叉树是一棵满足「每个节点的左右两个子树的高度差的绝对值不超过 1 」的二叉树。

```
示例1：
输入：nums = [-10,-3,0,5,9]
输出：[0,-3,9,-10,null,5]
解释：[0,-10,5,null,-3,null,9] 也将被视为正确答案：
```



理论上如果要构造二叉搜索树，可以以升序序列中的任一个元素作为根节点，以该元素左边的升序序列构建左子树，以该元素右边的升序序列构建右子树，这样得到的树就是一棵二叉搜索树。 本题要求高度平衡，因此我们需要选择升序序列的中间元素作为根节点，这本质上就是二分查找的过程：

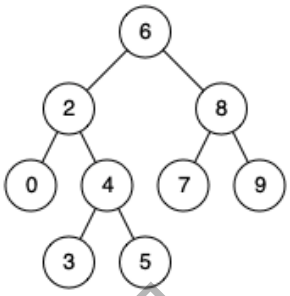
```
class Solution {
    public TreeNode sortedArrayToBST(int[] nums) {
        return dfs(nums, 0, nums.length - 1);
    }

    private TreeNode dfs(int[] nums, int lo, int hi) {
        if (lo > hi) {
            return null;
        }
        // 以升序数组的中间元素作为根节点 root。
        int mid = lo + (hi - lo) / 2;
        TreeNode root = new TreeNode(nums[mid]);
        // 递归的构建 root 的左子树与右子树。
        root.left = dfs(nums, lo, mid - 1);
        root.right = dfs(nums, mid + 1, hi);
        return root;
    }
}
```

除了通过数组构造，是否可以通过一个个插入的方式来实现呢？当然可以，这就是LeetCode701题，如果要从中删除一个元素呢？这就是LeetCode405题， 感兴趣的同学可以自己研究一下。

4.5 公共祖先问题

4.5.1 二叉搜索树的最近公共祖先

LeetCode235.给定一个二叉搜索树, 找到该树中两个指定节点的最近公共祖先。



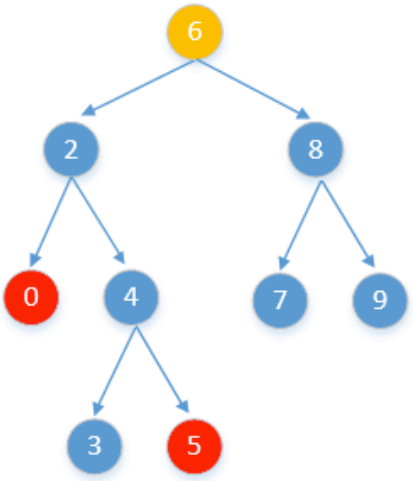

```
示例1:
输入: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 8
输出: 6
解释: 节点 2 和节点 8 的最近公共祖先是 6。

示例2:
输入: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 4
输出: 2
解释: 节点 2 和节点 4 的最近公共祖先是 2, 因为根据定义最近公共祖先节点可以为节点本身。
```

这个题虽然在LeetCode里标记是简单，但是笔者感觉不是很容易想明白，这题让求二叉搜索树的最近公共祖先，而二叉搜索树的特点就是 左子树的所有节点都小于当前节点，右子树的所有节点都大于当前节点，并且每棵子树都具有上述特点，所以这题应该从根节点开始遍历：

- 如果两个节点值都小于根节点，说明他们都在根节点的左子树上，我们往左子树上找
- 如果两个节点值都大于根节点，说明他们都在根节点的右子树上，我们往右子树上找
- 如果一个节点值大于根节点，一个节点值小于根节点，说明他们他们一个在根节点的左子树上一个在根节点的右子树上，那么根节点就是他们的最近公共祖先节点。

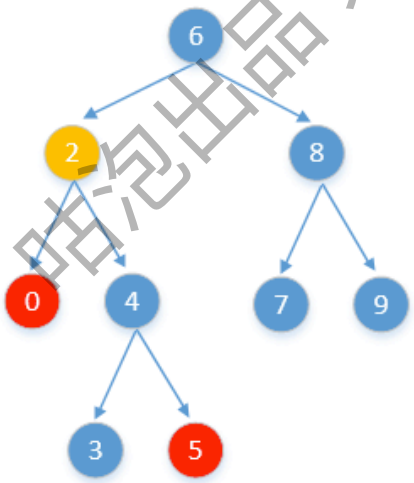
画个图看一下，比如要找0和5的最近公共祖先节点，如下图所示：

第一步



因为6比0和5都大，所以0和5是在节点6的左子树上

第二步



节点6的左子树的节点值是2，2比0大，并且小于5，所以0和5一个是在节点2的左子树，一个在节点2的右子树，那么节点2就是0和5的最近祖先节点

```
//迭代法
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
        while(root!=null) {
            if (root.val > p.val && root.val > q.val) {
                root = root.left;
            }else if (root.val < p.val && root.val < q.val){
                root = root.right;
            }else { //当无路可走，便是答案
                return root;
            }
        }
        return null;
    }
```

上面的代码可以直接改为递归的方式：

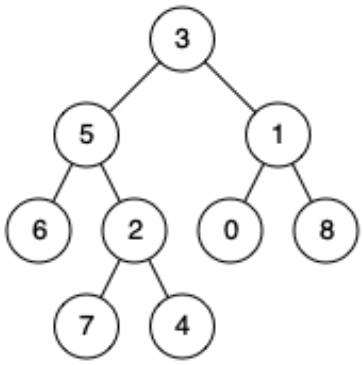
```
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
        //如果小于等于0，说明p和q位于root的两侧，直接返回即可
        if ((root.val - p.val) * (root.val - q.val) <= 0)
            return root;
        //否则，p和q位于root的同一侧，就继续往下找
        return lowestCommonAncestor(p.val < root.val ? root.left : root.right, p, q);
    }
```

4.5.2 普通二叉树的最近公共祖先

如果将搜索二叉树换成普通的二叉树该怎么做呢？这就是LeetCode236.给定一个二叉树, 找到该树中两个指定节点的最近公共祖先。

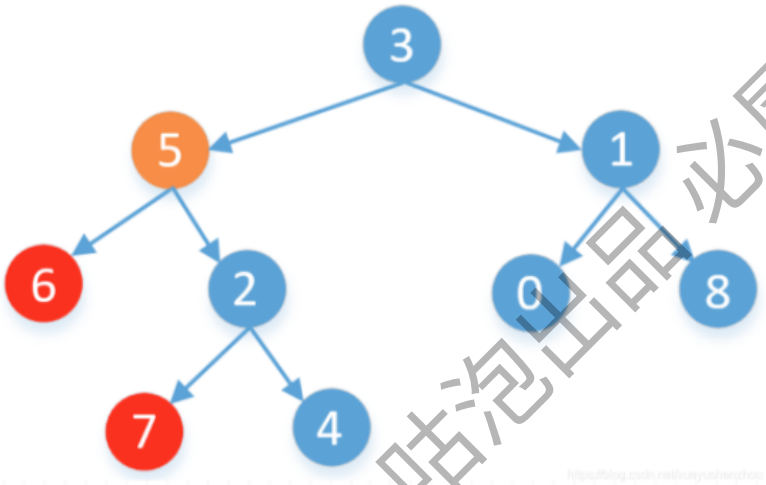
最近公共祖先的定义为：“对于有根树 T 的两个节点 p、q，最近公共祖先表示为一个节点 x，满足 x 是 p、q 的祖先且 x 的深度尽可能大（一个节点也可以是它自己的祖先）。”

例如，对于下面的二叉树：




```
示例1：
输入：root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 1
输出：3
解释：节点 5 和节点 1 的最近公共祖先是节点 3 。

示例2：
输入：root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 4
输出：5
解释：节点 5 和节点 4 的最近公共祖先是节点 5 。因为根据定义最近公共祖先节点可以为节点本身。
```

要想找到两个节点的最近公共祖先节点，我们可以从两个节点往上找，每个节点都往上走，一直走到根节点，那么根节点到这两个节点的连线肯定有相交的地方，如果是从上往下走，那么最后一次相交的节点就是他们的最近公共祖先节点。我们就以找6和7的最近公共节点来画个图看一下：



6的祖先结点有3 和5，7的是3，5，2，所以6和7的最近公共祖先是5。如果要用代码实现，需要考虑好几种情况。根据以上定义，若 root是 p, q 的 最近公共祖先 ，则只可能为以下情况之一：

(1)p和 q 在 root 的子树中，且分列 root 的 异侧（即分别在左、右子树中）；
(2)p = root，且 q 在 root 的左或右子树中；
(3)q = root，且 p在 root 的左或右子树中；

而具体在执行递归时，我们要判断的情况稍微复杂一些：例如我们在上面的树中查找6和7的公共祖先，遍历的时候从树的根节点开始逐步向下，假如某个时刻访问的结点为root，我们通过后序递归的查找其左右子树，则此时的判断逻辑是：

- 1.如果left和right都为null，说明在该子树root里p和q一个都没找到，直接返回null即可。例如上图中递归到了root为1的子树时。
- 2.如果left和right都不为null，说明p和q分别在root的两侧，例如root为5时，此时6和7就分别在其两侧，直接返回5即可。
- 3.当right为空，left不为空时，此时情况略复杂，要考虑两种情况：

  (1)先判断一下root是不是p或者q，如果是说明q和p一个是另一个的祖先，直接返回就好了，否则：

  （2）说明right子树里什么都没查到，而6和7是在left子树里，此时需要递归的去左子树查即可。例如root为3时，此时递归的结果必然是right为null而left不为空。
- 4.如果left为空，而right不为空，说明是与情况3相反的情况。

总结看递归的代码：

```
public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
    if (root == null) {
        return null;
    }
    if (root.val == p.val || root.val == q.val) {
        return root;
    }
    TreeNode left = lowestCommonAncestor(root.left, p, q);
    TreeNode right = lowestCommonAncestor(root.right, p, q);

    if (left != null && right != null) {
        return root;
    }

    if (left == null) {
        return right;
    }

    if (right == null) {
        return left;
    }

    return null;
}
```

5.大厂面试实战

二叉树是我们算法面试的绝对重点，涉及的题目多， 有些难度还挺大，如果不准备，面试时甚至都不知道怎么做。

经过上面的学习，我们也可以看到，很多题目都是有规律可循的，最大的规律就是二叉树的特征，以及其遍历方法。我们按照层次遍历、前序后序遍历、中序与搜索树三个章节分析了30多道高频题目的解决方法。很多题目我们给出了不止一种方法。本人认为这些方法大部分都应该掌握，至少理解用别的方法结解题的思路是什么。只有这样才能融会贯通，刷清楚一道题，10道题就不用做了。