### 所有的树问题，全部可以转化为递归问题（递归都是自底向上的！！！！迭代是从上至下！！！）。

dfs没有返回值就是单纯的向下，“递”，有返回值就是要考虑“归”，有返回值的思考和无返回值的思考是不一样的，在设计时要分开想，只是递吗？还是要归一下？是在递的时候（前序遍历）操作还是归的时候操作（后序遍历）？每次做树首先就是考虑这个

### 词汇：

Lexicographically 按字典排序

### 节点的交换，可以是node的直接交换，而不用val的交换。

### 关于递归时赋值还是不赋值的问题，就看子节点是否发生改变。



### 3.后续遍历查深度，背也要背下来





后续遍历是从下往上的操作。

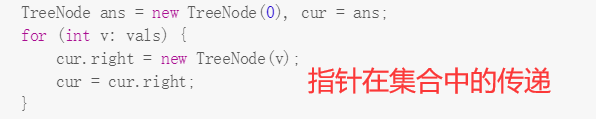
### 4.中序遍历模板:



### 5.在递归中使用全局变量的案列:



### 6.指针的方式传递节点，指针指向该节点的做法：逗号后面直接指向。



### 7.层序遍历（要背，清楚逻辑），也可以不用嵌套两层List，单list记录值也行



这里用记录size的方法来控制每一层是多少个。

还有一种如果是对每一个元素进行操作，而不需要考虑层数，则可以不加size的层数束缚,只管遍历就完了:



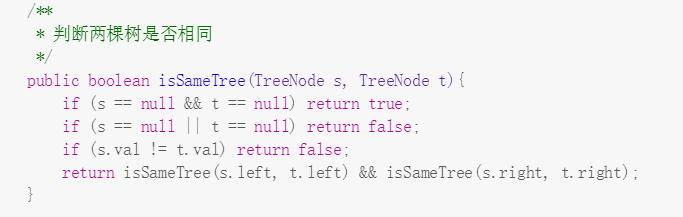
### 8.判断二叉平衡树（深度相差小于2），典中典，使用特殊数字“-1”,起到boolean效果。

### （也叫做剪枝操作）

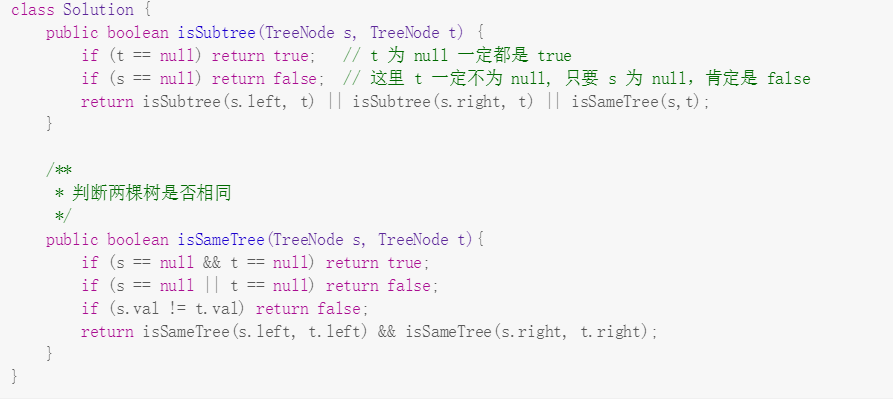


只要再同一个函数体内，当时的left和right值就是该节点的左右深度对应的值（不用怀疑）

### 9.判断两树完全相同的公式



### 10.判断一棵树是否为另一棵树的子树:（重点）



解题思路即，让S的每一个节点都和t进行对比，直到S为空，当比较到S为空时，说明S和T没有一处是相似的。

2：注意第一个函数中的return，意思就是一边往下遍历isSubtree(root.left,subRoot) || isSubtree(root.right,subRoot)

，一边做对比 isSameTree(root,subRoot);

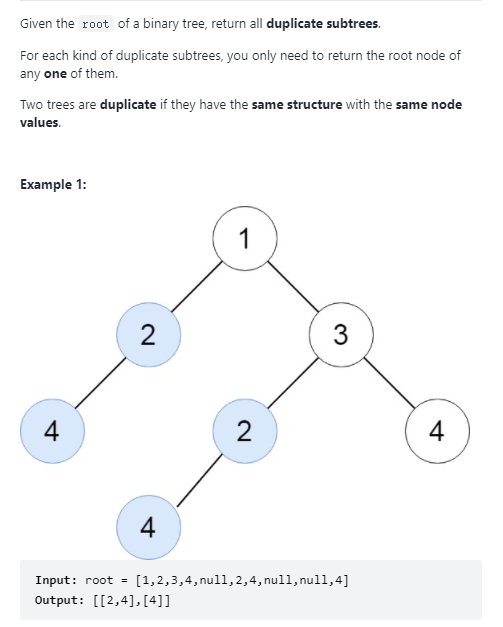
这里使用后续遍历会让效率更高。

### 10.5 如何找到一个树中相同的两个子树（序列化每个节点）

此时是针对该树本身，而不是两棵树了

最直观，也是最简单的方法:

序列化每个节点，进行比较。存在重复的，直接放入结果集。（这种方法其实就是暴力法，应该是第一想到的解法，关于判断树是否相同的题，递归比较往往是两树之间比较。）



图中最右边的4应该也是深色才对。



### 11.修枝操作，即让原节点指向null,怎么指向null?满足条件时返回null即可。

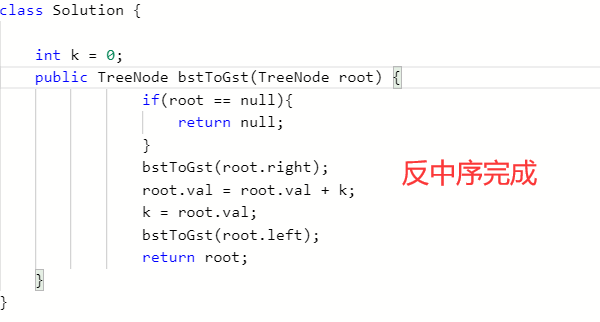


这些削剪子问题全部都可以，先出一个条件函数（这个条件函数也可能包含递归，完成整个子树的判断），然后对其根节点进行循环，意思是两重递归，解决子树条件问题。

### 12.所有递归，返回的是当前，操作的是以后



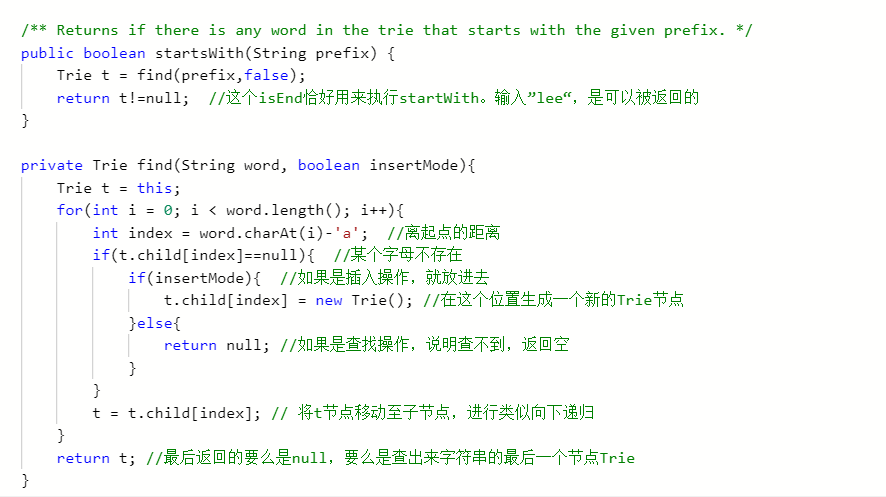
### 13.纪念一下第一次自己做出中等难度:



### 14前缀树 （很经典）



这个find里加入判断是否插入，这种写法很巧妙，反正，搜索只要是查不到，就返回null就完事了。



16.集合元素颠倒:

Collections.reverse(); ，注意这个方法不返回东西，只做操作



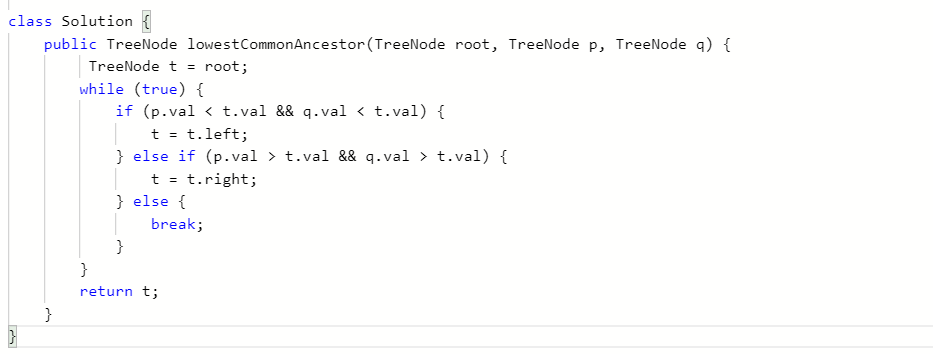
### 16.5 二叉搜索树的公共祖先:（背方法）

典中典问题,关键点在于记住搜索树的性质，左边的节点<中间的节点（祖先）<右边的节点。

所以找公共祖先有两种情况:

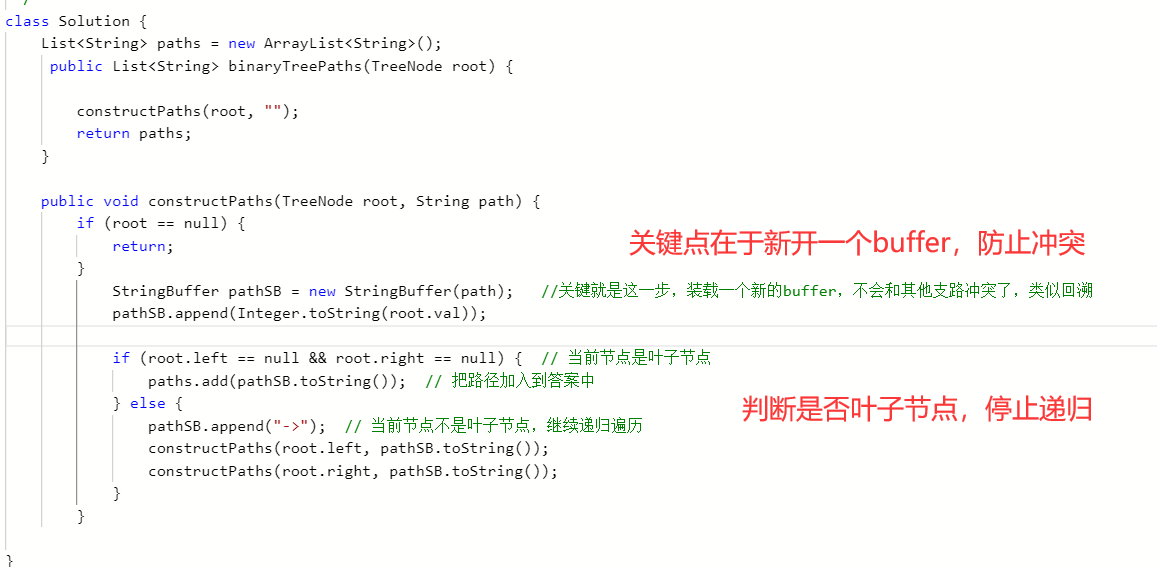
1.祖先节点处于两节点之间(按这个标准去找即可)

2.两节点在同一边，那么两节点中其中一个为祖先（先遍历到的那个即为祖先）。



再补:

所有叶子节点的路径: dfs即可，但要注意是怎么记录路径的，从string到Stringbuffer的用法。



### 16.6 任意二叉树的公共祖先:

见书 265 页，有总体方法。

三种情况 + 后序遍历 即可得到。

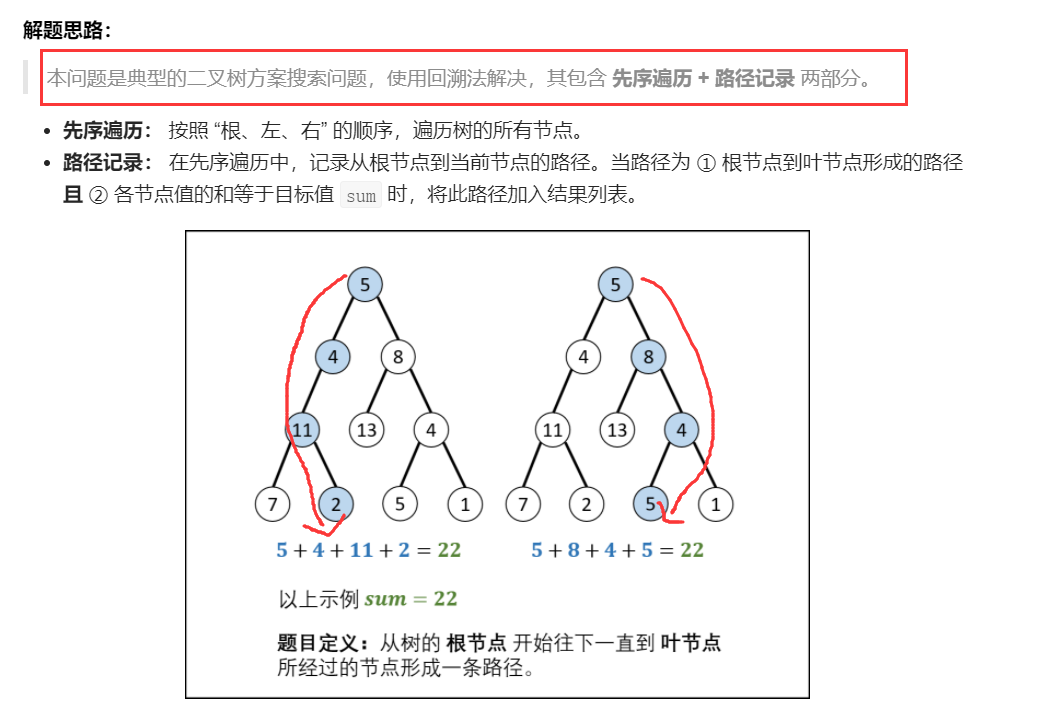
### 17.典中典之树的回溯记录：

这个题太经典了，回溯+树遍历的思想

解题核心:先序遍历+路径记录（记得清除记录，即撤销选择）

一行代码实现回溯，让到达每个节点时的路径都是以这个节点作为终点。

其实很简单，就是来的时候做记录，去的时候删记录罢了。





下面还有个图，二刷的感悟：



这里的回溯清除，是在递归回溯的时候，边回溯边清除记录，在向下递归时记录。

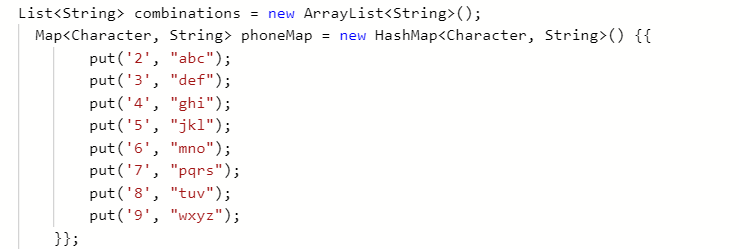
具体的回溯思想，可以去看回溯篇。

这个就是撤销选择，将选择在离开该节点，回到原来的root时做的操作。（P46）

### 18.回溯入门题:

必会题，搞清楚公式的内涵所在:







像这种排列组合不是一个集合内部组合，而是多个集合里面的元素互相组合的

即:{1,2,3}内部组合和{a,b} & {c,d,e} 多个集合组合形式

一般多集合的函数公式中都会增加一个index来跨集合操作。比如这一题的index就是从号码”2”跳到了号码 ”3” ,同样类似的手表问题，也是多集合组合形式，也用到了一个index来跨集合。

而内部组合在循环公式中，会多加一个排除相同项continue的操作，跨集合不同元素则不用。

在backTrack中可以把计数的都放在函数参数当中，随之传递

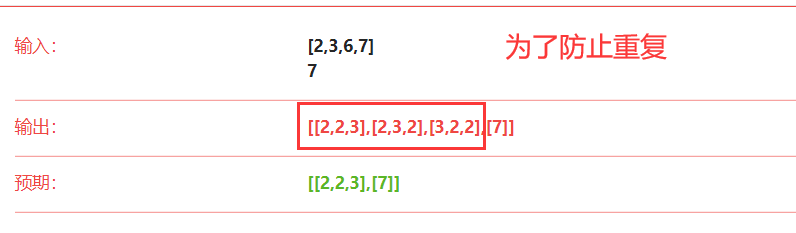
19.经典“（）”问题:

关键点:

1. 只有两个选择，没有for循环
2. 保证“）”之前一定有足够的“（”对应，需要open>close
3. 用来计数的参数可以带着函数走
4. 这里的数字全部要用+1,不能用++,因为在递归中，他会先把递归的东西搞完之后再加1，相当于没有写。
5. 

下面应该是i+1，这里错了





进阶:还有一种特殊的避免，相同元素导致的相同合集，比如一个数组中有两个1,1

后面结果会出来[1,7] 和 [1,7]，两个1是不一样的，这个时候需要先排序 + i > index，

注意后面回溯中还要 i+1；



若不加if判断，结果会被相同项干扰。



就以后要是有相同项会造成干扰结果的，就排序 + 这个if判断

更进阶:使用used表，不用index，实现所有元素的全排列。最正宗的全排列。



神奇的是!used[i-1]和used[i-1]都可以通过。阿这。。。。

重点: 对于使用index，和used表，做一个总结:

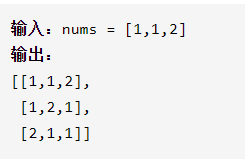
（见书300，有更好的总结，排列组合子集这三种的关系）

1. 使用index是为了避免出现重复结果，如: [2,2,3], [2,3,2] 他们的sum结果都是7

但是不能防止“相同元素被重复选取”，即这里面的2，可能都是同一个2.

1. 对于不含重复数字的全排列，使用track.contain去过滤即可。

那如果有相同数字的全排列，即针对每一个数组位置元素的全排列，应该怎么做？



这个时候用used表去完成

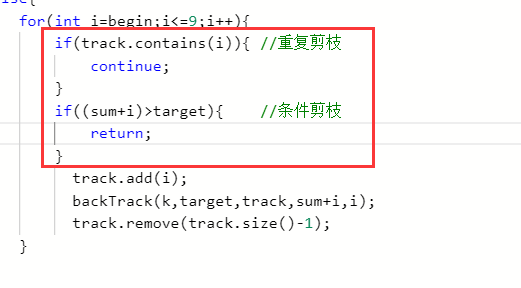
1. 为了达到“相同元素不能重复选取”，需要使用used表，即去记录每一个元素是否被使用。一般用used表的题都是单纯的排列题，即实现一个数组的全排列，不会是条件满足题，如那几个数字合为7这种。

Used表的使用如更进阶所示，主要注意第二个if中的条件。意思是:

1. 排除相同项的干扰 2.元素还未被使用，可以往下跳。

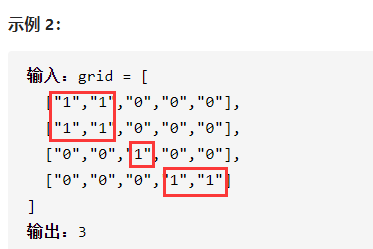
### 20.有一个还原ip地址的题，感觉会常考。

### 21.条件限制的剪枝一般都放在for循环当中:



这么看来，回溯的套路比树简单多了。

### 22.一个BFS里DFS的解法（岛屿问题，深度上下左右）：



即找到“1”区域块。

可以使用经典的dfs“上下左右深度搜索”，通过寻找“1”边界来确定四周。

关键点:走过的路清除，有点像扫雷，将一次扫过的“1”区域全变更为“0”区域。



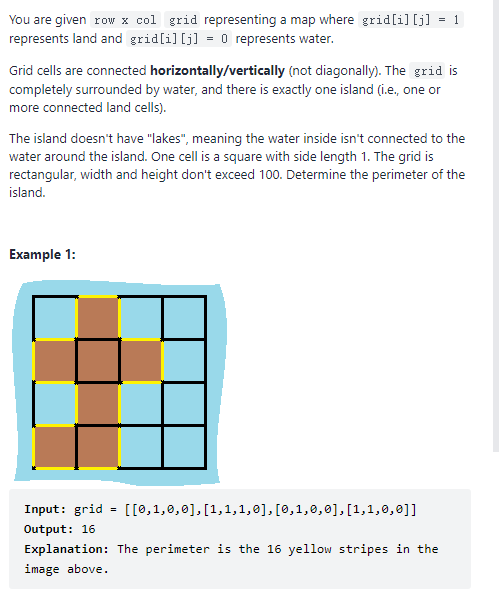
### 22.5再补一个海岛问题:

计算边长，注意答案中是怎么避开重复走的:

设经过的岛屿为grid[i][j] = 2;

判断时即可避开重复走。

其余的都是岛屿dfs公式部分。



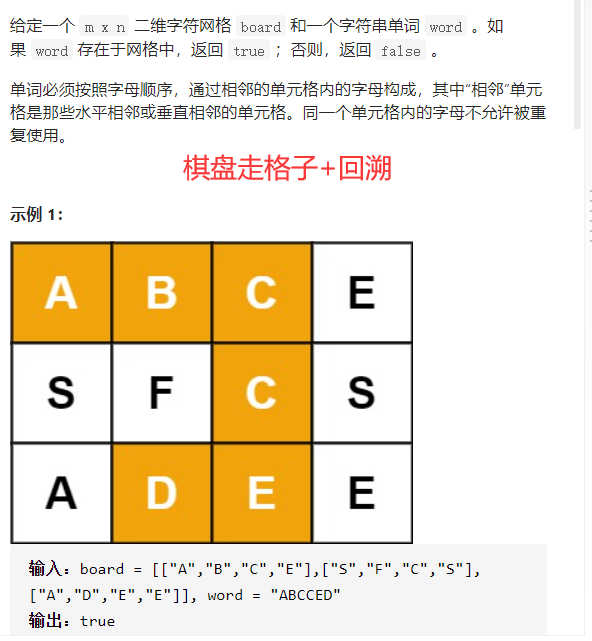


### 23.dfs+回溯：

只要是棋盘问题，大多都可以使用dfs去解题，海岛是不带回溯的，下面是带回溯的写法，

需要在dfs之后，把老路给改回去。

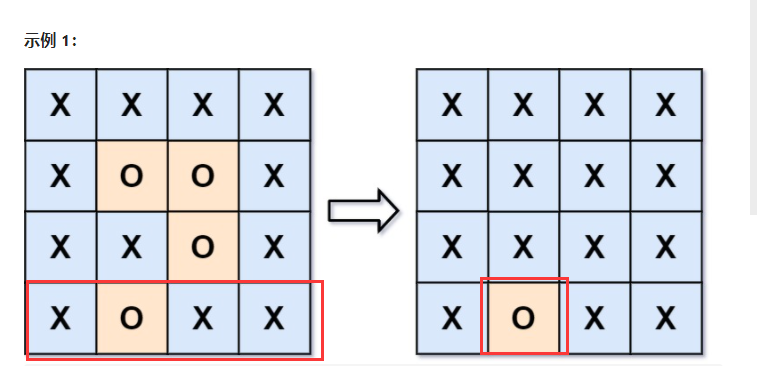
需不需要回溯，主要是看判断的时候要不要数据的连续性存在（这是一种感觉），像这里的字母串，就需要回溯



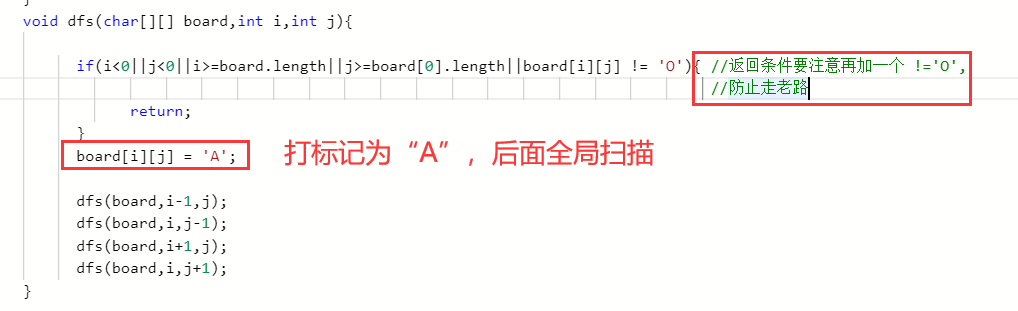


23补充:

算是单独的题解，这种题，以边单位为开始寻找条件的，有“扫描边，打标记”的操作。







后面全局扫描的时候，将A的改回去。

24.修课:A->B问题 (典中典): 这题得背，背思路，写都很好写，主要是思路原理，一般很难想到

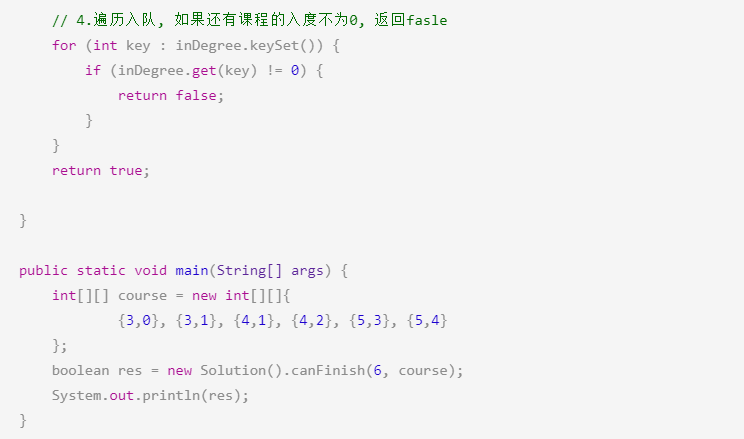


<https://leetcode-cn.com/problems/course-schedule/solution/bao-mu-shi-ti-jie-shou-ba-shou-da-tong-tuo-bu-pai-/>

解答连接如上，是 拓扑学的入度集合 + BFS 结合 来解题，思路很好理解，但得记住





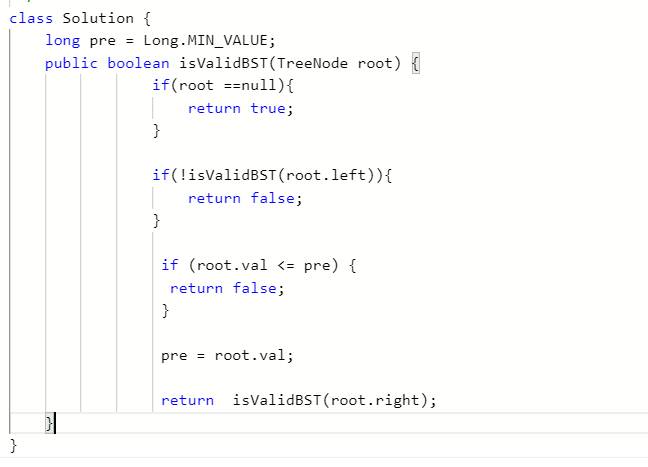


### 典中典之验证二叉搜索树:

二叉搜索树的中序遍历出来的数组，一定是递增数列！

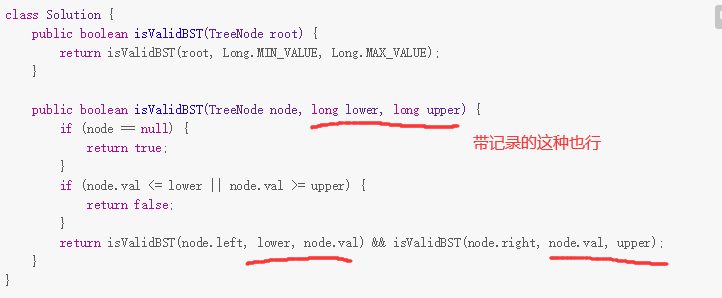
这是一种解题的性质，可以去排列来判断是不是二叉树，或者利用这个性质去打记号解题。

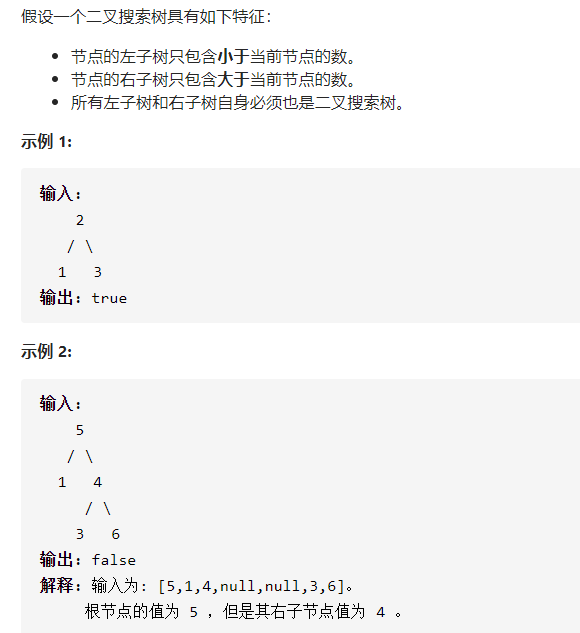
下面这种方法也是中序遍历判断完成的。



中序遍历即可。

带记录的这种方式也行:





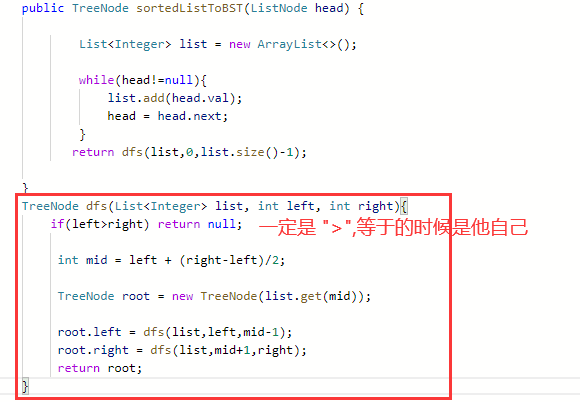
这里有个技巧，怎样在递归中，左边<x,右边>x成立，

是靠！取非来操作的，就是判断不变，一个结果取非即可。

### 25.典中典之升序数组构建搜索树：

还是利用了搜索树 左<中<右的性质，

这个dfs很模板，看一眼肯定能记起来。



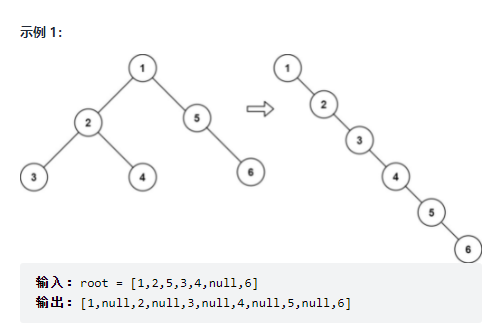
### 26.典中典之恢复二叉搜索树:



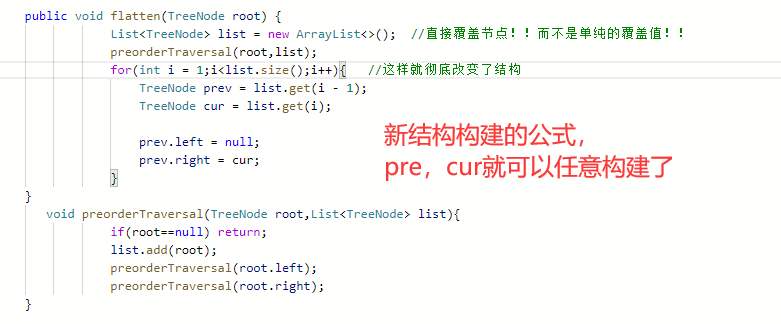
二刷解释: 一共有两个不和谐的元素，首先找到第一个不和谐元素X，确定他，然后再找第二个不和谐元素Y，注意Y是会变的。这个时候就不要改变X了，改变Y即可。

最后替换X和Y。即一共会进入if（不和谐判断）两次，第一次确定X(pre)，第二次确定Y(node)，一共两个节点，反正就这么记吧。

### 典中典之二叉树展开为链表（改变结构一定要存节点！而不是值）



这里明显是树结构直接发生改变，所以要想用记录法的话，一定要存节点！而不是单纯存值，值改变不了树结构。节点替换可以。



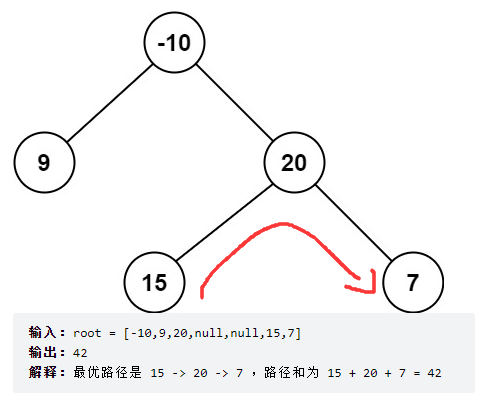
### 典中典之BFS层指向:

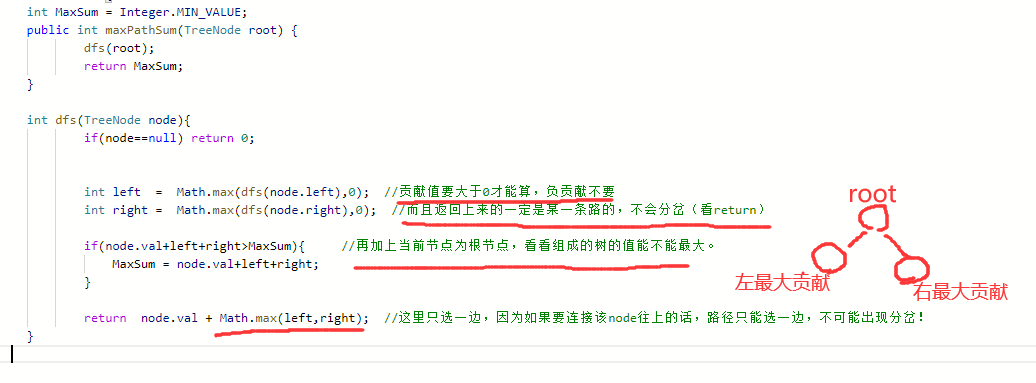
就是要对BFS的运行机制要很熟悉，

才能搞懂为啥要加那个if(i<size-1)的判断。



### 典中典之树的最大路径和:





把注释看完绝对懂。

### 典中典之所有距离为K的节点（from为往上走服务，避免走重复）

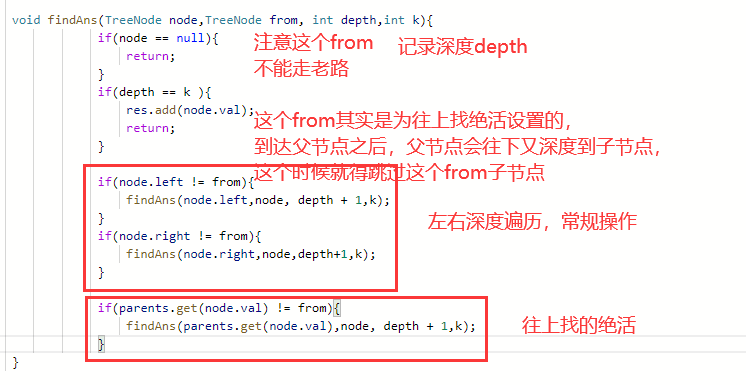
这题入典是因为题型比较好记，很有针对性, 而且技巧十足，可以实现节点向上遍历。（一般都是深度遍历）

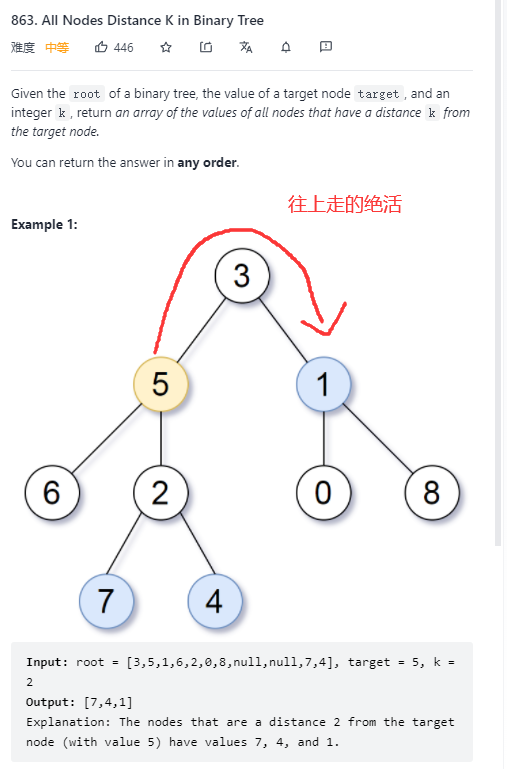
唯一一点要注意一下就是要记录from，为往上找的绝活服务，

往上到达父节点后，为了让父节点不再往下递归而是持续往上，这个时候就得判断，如果是from来的，就不深度递归，继续往上走。

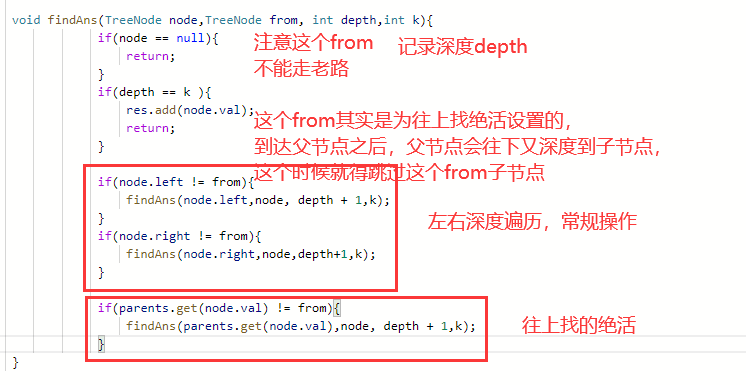
对于绝活自己也得加，在正常的向下深度递归中，得避免重复的向上寻找绝活。

记就完了，这样三个都加也挺整齐的。









典中典之二叉搜索树的插入“递”（dfs没有返回值就是单纯的向下，“递”，有返回值就是要考虑“归”，有返回值的思考和无返回值的思考是不一样的，在设计时要分开想，只是递吗？还是要归一下？是在递的时候操作还是归的时候操作？每次做树首先就是考虑这个

不用dfs，因为不用平衡，所以直接找到那个可以插入的位置即可:

下面就是典型的只是向下“递”，不用考虑“归”



### 二叉树的边界（左分支节点+底层节点+右分支节点倒序）

二叉树的边界:

左边中层节点+叶子节点+右边中层节点

注意rightIndex的用法，实现倒序



