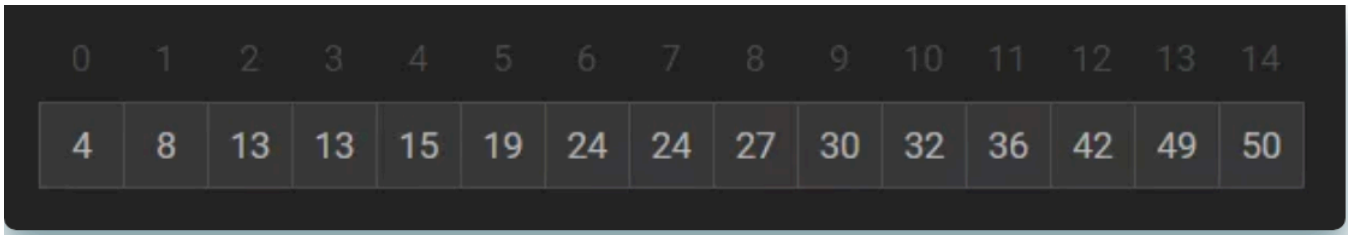
Day45：二叉树的价值和基础概念

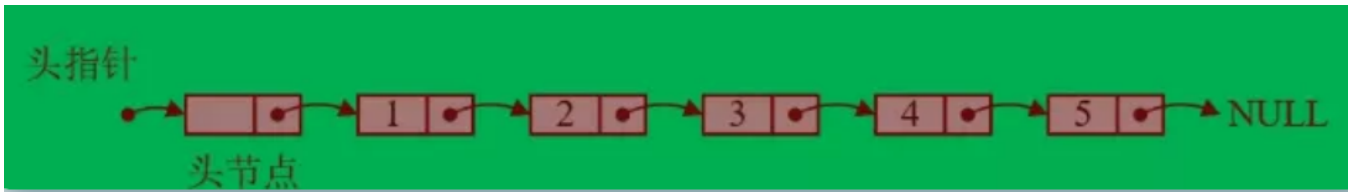
本总结参考星球内星友们的回答。

1. 二叉树与一维的、线性的数组、链表等数据结构有何不同？它的价值是什么？

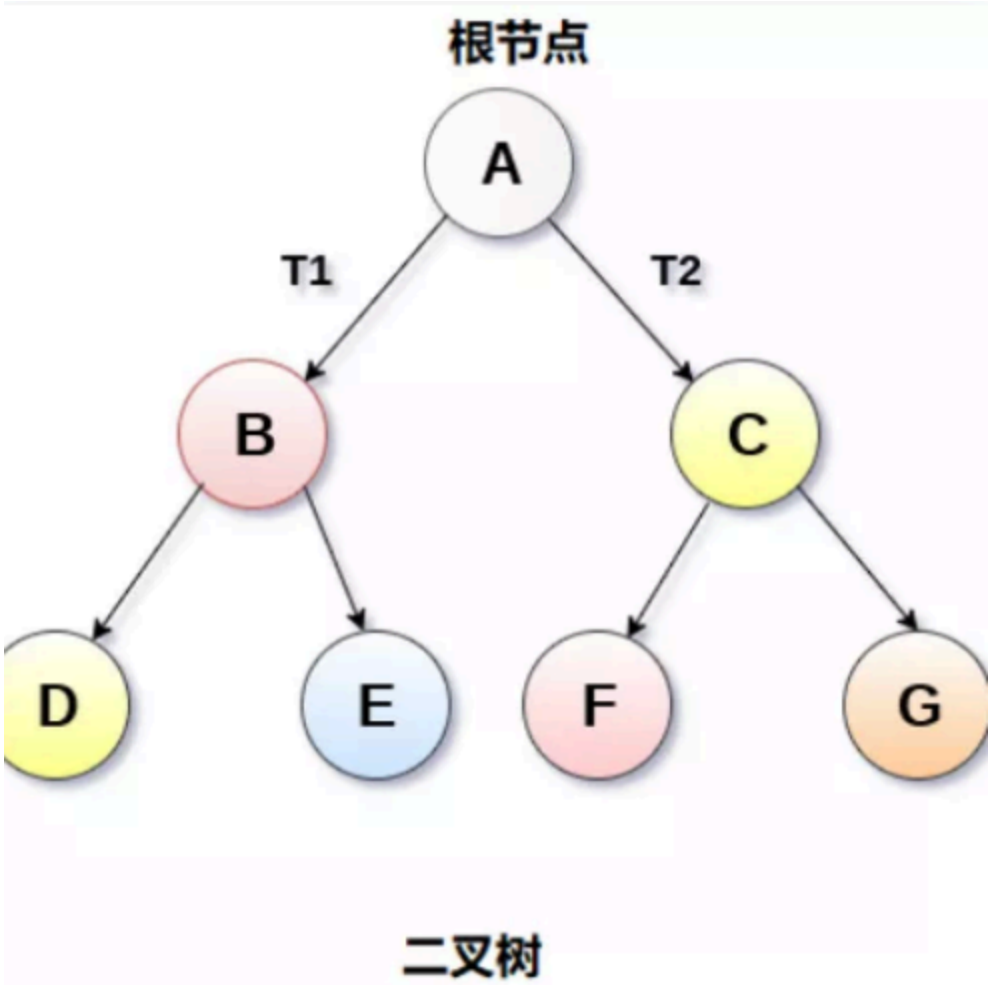
数组是一组连续的内存空间，大小事先指定；每个空间只存储数据。如下是一个数组：



链表中的每个节点除了存储数据，还有指向下一个节点地址的指针，所以它可以是非连续的内存空间。如下是一个链表：



二叉树的节点相比于链表节点，多了一个指针，其两个指针一个指向左子节点，一个指向右子节点。如下为二叉树：



二叉树的价值：

二叉树是典型的**非线性**数据结构，在实际中，有很多逻辑关系并不是简单的线性关系，常常存在一对多，甚至多对多的情况。比如我们家庭成员之间的关系、企业中职级关系。**因此需要像二叉树这种非线性数据结构来表示**。

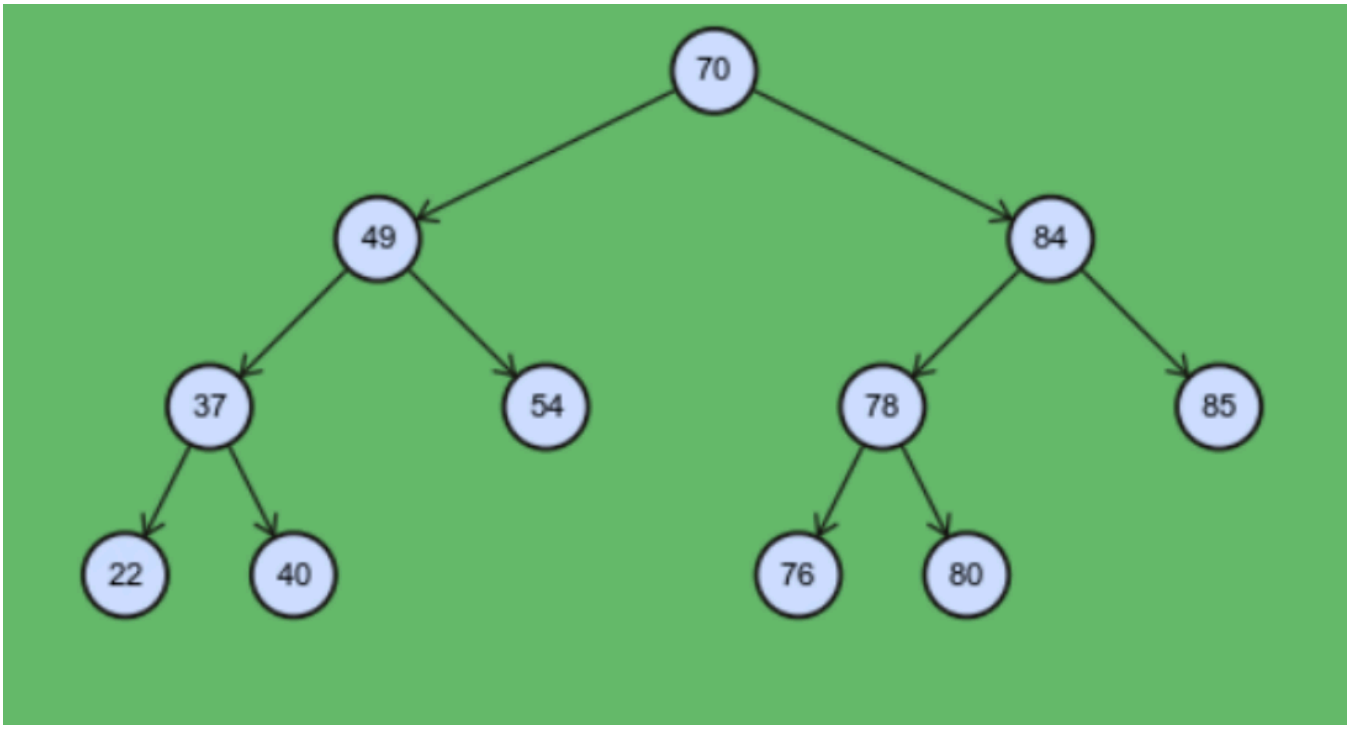
二叉树的优点，也可能最大的价值所在：在平衡的情况下，可以保证对二叉树的查找和插入都是$O(logn)$的时间复杂度。这很了不起。参考下面的例子：

假设要获取关键码在0到5000之间的所有元素。实际上，只有一个这样的元素，10000个其他键不在此范围内的元素。二叉树可以有效地进行范围搜索，因为它不搜索不可能获得答案的子树。如果选用数组和哈希表，他们的时间复杂度都为 $O(n)$

二叉树的缺点：二叉树有可能不平衡，直接退化为一个单链表；

除此之外，如果要使用二叉树存储同样的信息，每个二叉树节点都需要额外的保存两个指针值，从这种角度讲它没有做到高效使用内存。

2. 写出二叉树的以下概念：路径、根、父节点、孩子、叶结点、子树、访问、遍历、层级和关键码。



路径：从根节点到叶节点所经过的节点和边组成一条路径，如上图从70->49->37->22就是一条路径；

根：二叉树的顶点，也就是二叉树顶部的那个节点，70是上图二叉树的根；

父节点：和子节点相对，左右子节点的上一个节点，49是37和54节点的父节点；

孩子：也即左右子节点；37和54是49的孩子节点；

叶节点：无孩子节点的节点；22,40,54都是叶节点；

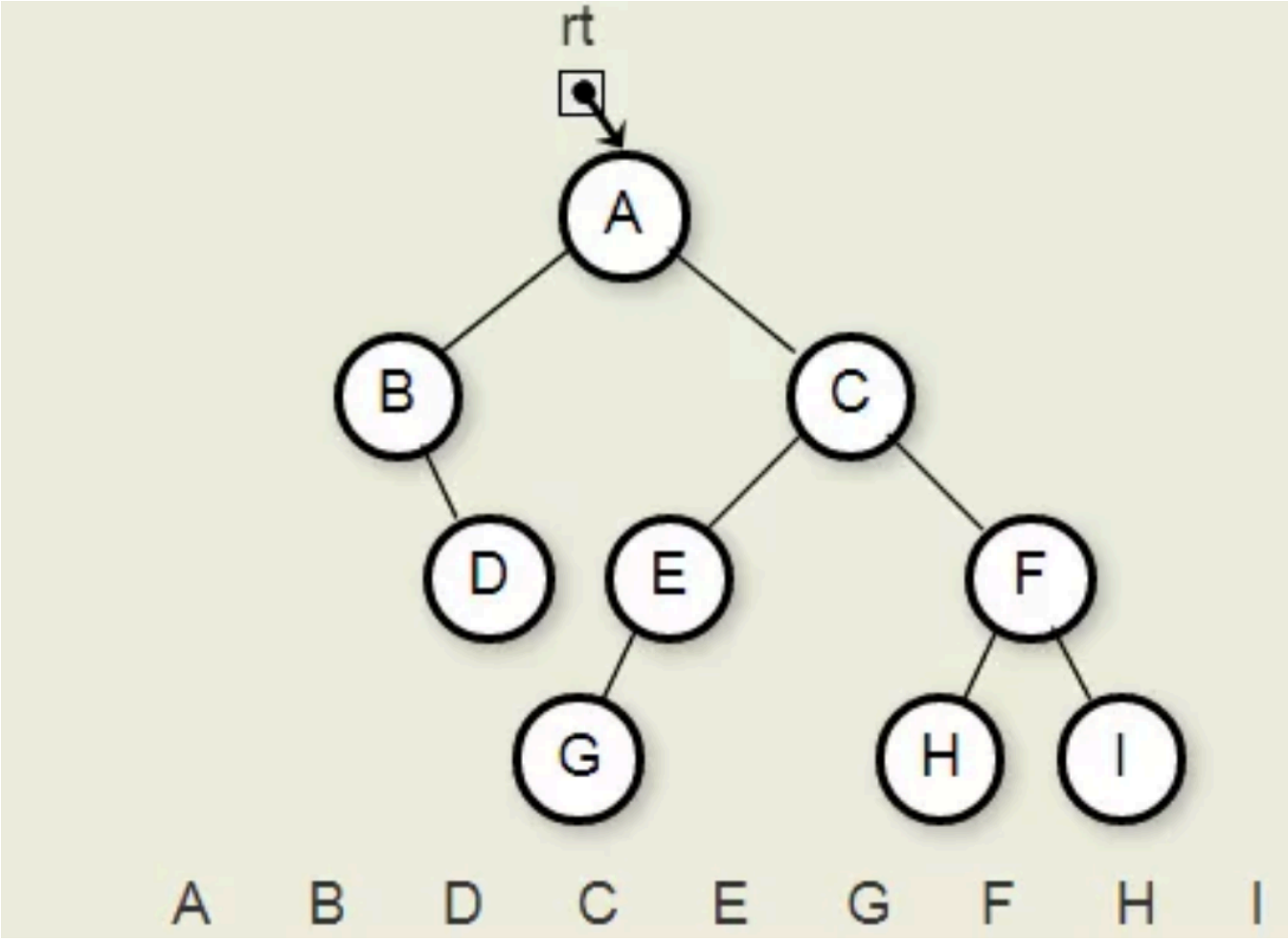
子树：子节点及其包含的所有节点所组成的树；49节点和其包含的所有节点组成一个子树；

访问：读取树节点的值

遍历：读取树所有节点的值，一般有前序、中序、后序、按层遍历。

遍历的动画可参考：

https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/BinaryTreeTraversal.html



层级：高度加1等于层数，也就是从根节点到该节点经过的节点数

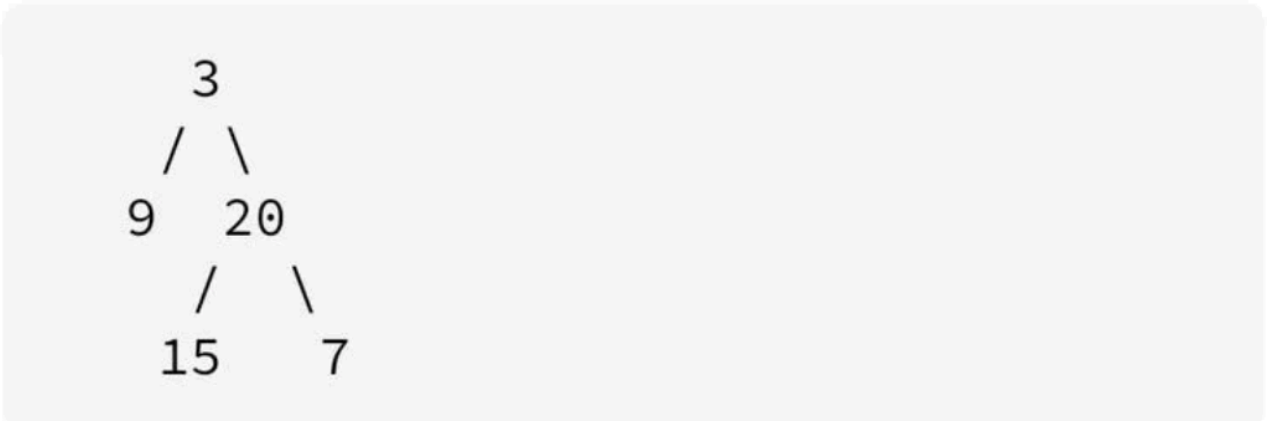
关键码：节点的值被称为关键码

Day46: 列表转化为二叉树

已知列表nums，将其转化为二叉树。举例：

nums = [3,9,20,None,None,15,7]，转化为二叉树后:

节点3的左子节点9，右子节点20，9的左右子节点都为None，20的左子节点15，右子节点7，参考下面：



二叉树定义：

```
class TreeNode:
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None
```

请补全下面函数：

```
def list_to_binarytree(nums):
    pass
```

构建分析

构建满足以上结构的二叉树，可以观察到：树的父节点和左右子节点的关系：

$$left_{index} = 2 \times parent_{index} + 1$$

$$right_{index} = 2 \times parent_{index} + 2$$

基于以上公式，再使用递归构建二叉树。

递归基情况：

```
if index >= len(nums) or nums[index] is None:
    return None
```

递归方程：

$$left_{node} = f(2 \times parent_{index} + 1)$$

$$right_{node} = f(2 \times parent_{index} + 2)$$

根据以上得到如下代码：

代码

```
def list_to_binarytree(nums):
    def level(index):
        if index >= len(nums) or nums[index] is None:
            return None

        root = TreeNode(nums[index])
        root.left = level(2 * index + 1)
        root.right = level(2 * index + 2)
        return root

    return level(0)

binary_tree = list_to_binarytree([3,9,20,None,None,15,7])
```

以上递归版本，代码比较简洁。除此之外，为了训练算法思维，我们还可以使用迭代构建二叉树，使用队列数据结构。但是代码相对复杂一点，这个等再过几天布置这个作业。

Day47: 求二叉树最小深度

Day47: 题目

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。

说明: 叶子节点是指没有子节点的节点。

示例:

给定二叉树 [3,9,20,null,null,15,7],

返回它的最小深度 2.

补全下面代码：

```
# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution(object):
    def minDepth(self, root):
        """
        :type root: TreeNode
        :rtype: int
        """
```

2 分析

分析过程的开始，我们先看一个错误的求解，并说明为什么它是错误的：

```
class Solution(object):
    def minDepth(self, root):
        """
        :type root: TreeNode
        :rtype: int
        """
        if not root:
            return 0
        if not root.left and not root.right:
            return 1
        return 1 + min(self.minDepth(root.left),self.minDepth(root.right))
```

考虑下面二叉树：

使用以上代码返回最小深度为 1，其实最小深度为 2，因为最小深度的定义为：从根节点到最近叶子节点的最短路径上的节点数量。

为什么上面的解有问题呢？

原因在于递归基选取有问题，只考虑了下面两种情况：

1. 二叉树为 None
2. 二叉树只有一个节点

递归基未考虑下面两种情况，所以导致出错：

3 代码

正确的求解，需要把上面遗漏的两种递归基考虑进去:

```
        # 递归基的下面两种情况必须考虑进去:
        if not root.left:
            return 1 + self.minDepth(root.right)
        if not root.right:
            return 1 + self.minDepth(root.left)
```

正确的完整代码如下：

```
class Solution(object):
    def minDepth(self, root):
        if not root:
            return 0
        if not root.left and not root.right:
            return 1

        # 递归基的下面两种情况必须考虑进去:
        if not root.left:
            return 1 + self.minDepth(root.right)
        if not root.right:
            return 1 + self.minDepth(root.left)

        return 1 + min(self.minDepth(root.left),self.minDepth(root.right))
```

Day48：是否为单值二叉树

如果二叉树每个节点都具有相同的值，那么该二叉树就是单值二叉树。

只有给定的树是单值二叉树时，才返回 true；否则返回 false。

```
示例 1：

输入：[1,1,1,1,1,null,1]
输出：true

示例 2：

输入：[2,2,2,5,2]
输出：false
```

出处：

https://leetcode-cn.com/problems/univalued-binary-tree/submissions/

补充下面代码

```
# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution(object):
    def isUnivalTree(self, root):
        """
        :type root: TreeNode
        :rtype: bool
        """
```

分析

检测二叉树是否所有节点取值仅有一个，此题的递归基也比较容易找到，分为以下三种情况：

1. 空树，返回True
2. 只有1个节点，返回True
3. 父节点与左或右子节点取值不等，返回False

如果以上都不满足，表明根节点与其左右子节点都相等，这样求解规模减少1。

剩下的就是递归判断：

```
isUniverse(root.left) and isUniverse(root.right)
```

是否满足即可

代码

有了上面的分析，递归求解代码如下：

```
class Solution(object):
    def isUnivalTree(self, root):
        if not root: # 1
            return True

        if not root.left and not root.right: #2
            return True

        if root.left and root.left.val!=root.val: #3
            return False
        if root.right and root.right.val!=root.val:
            return False

        return self.isUnivalTree(root.left) and self.isUnivalTree(root.right)
```

Day49: 对称二叉树

题目

给定一个二叉树，检查它是否是镜像对称的。

例如，二叉树 [1,2,2,3,4,4,3] 是对称的。

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

但是下面这个 [1,2,2,null,3,null,3] 则不是镜像对称的:

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

来源：力扣（LeetCode）
链接：https://leetcode-cn.com/problems/symmetric-tree/submissions/

```
# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution(object):
    def isSymmetric(self, root):
        """
        :type root: TreeNode
        :rtype: bool
        """
```

错误代码

判断二叉树是否对称，先看看下面代码是否正确，它实现的什么功能？

首先看递归基，分三种情况：

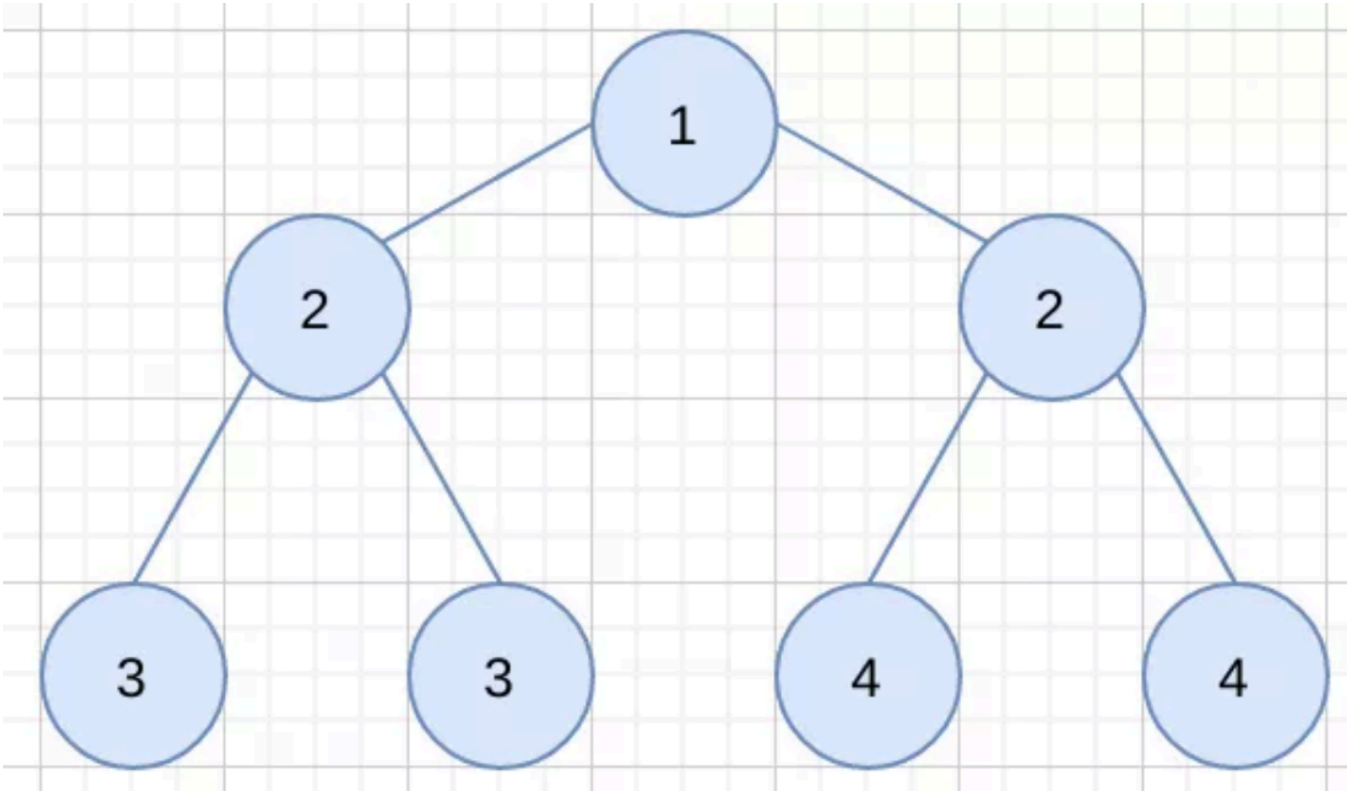
1. 没有根，就是空树，返回True
2. 没有左右子树，返回True
3. 左右子节点`val`不相等，返回False

递归方程如下，判断左、右子树都对称。

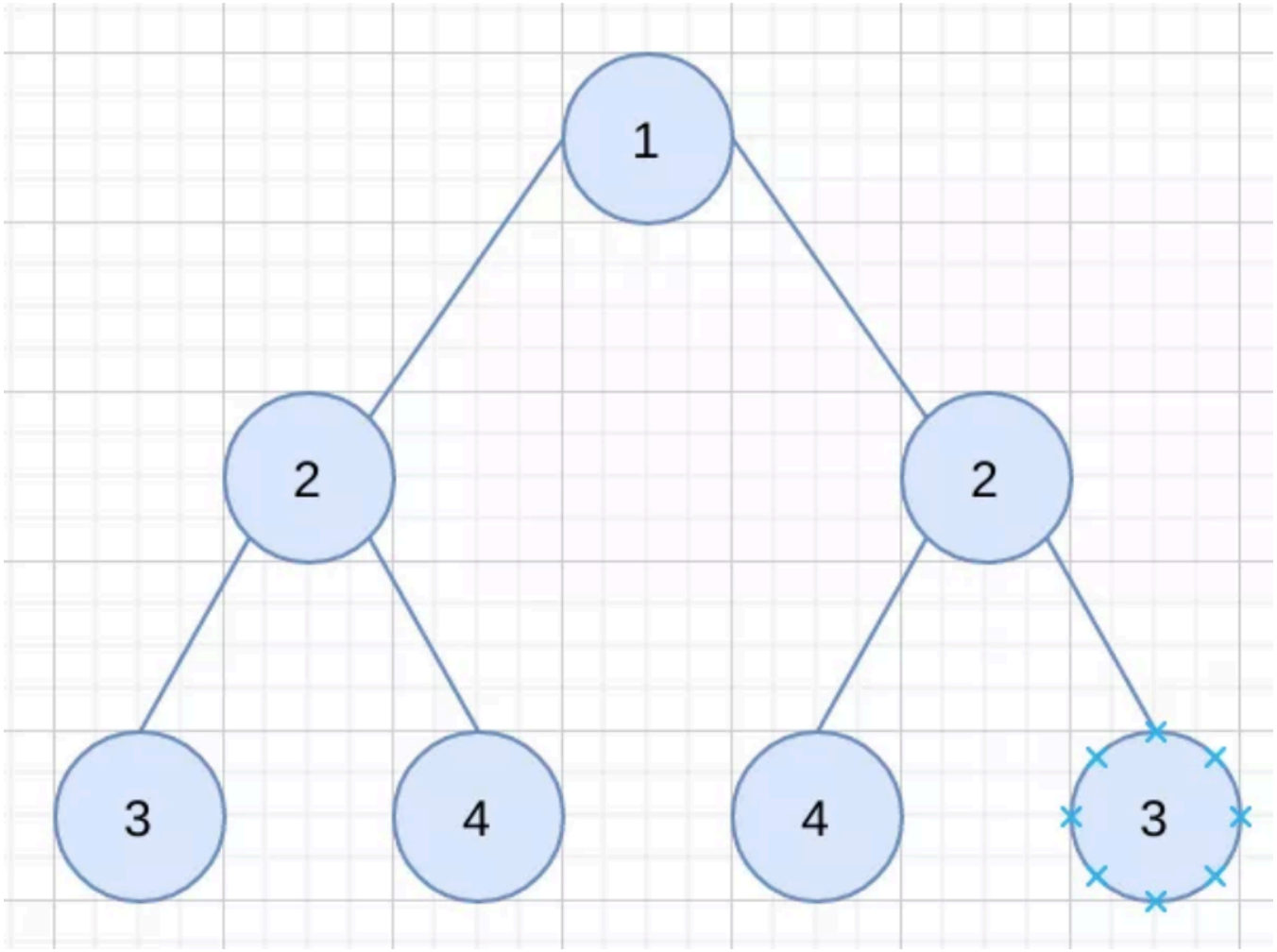
```
self.isSymmetric(root.left) and self.isSymmetric(root.right)
```

```
    def isSymmetric(self, root):
        """
        :type root: TreeNode
        :rtype: bool
        """
        if not root:
            return True
        if not root.left and not root.right:
            return True
        return root.left == root.right and self.isSymmetric(root.left) and self.isSymmetri
```

以上代码认为下面的二叉树才是对称的：

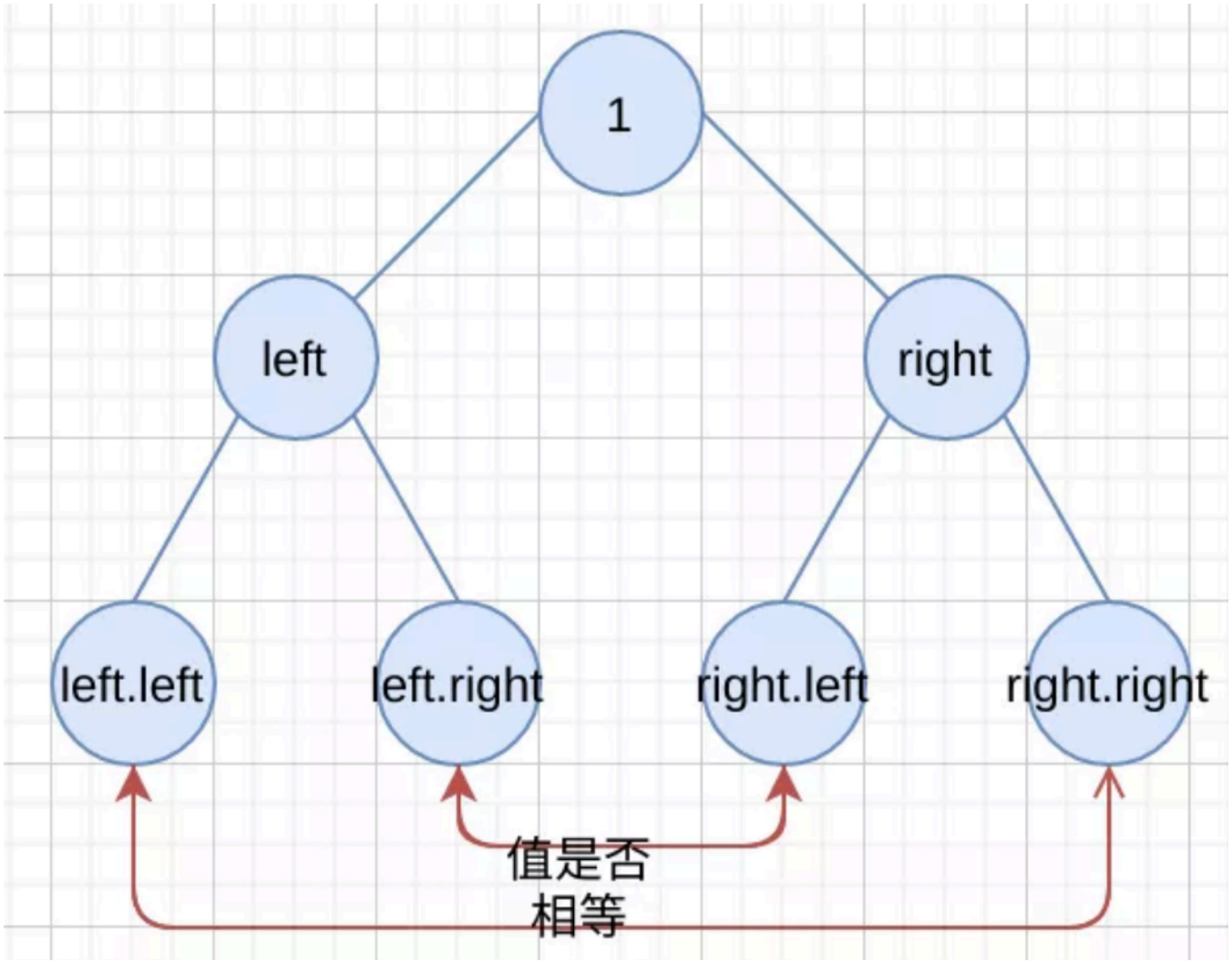


这与题目要求的对称二叉树明显不同，这样才是真的对称二叉树：



正确代码

错误代码错误的原因在于递归方程有问题。请看下图：



因此，得到正确的递归方程：

```
sub(left.left,right.right) and sub(left.right,right.left)
```

完整代码：

```
class Solution(object):
    def isSymmetric(self, root):
        if not root:
            return True
        def sub(left,right):
            # 没有左和右，返回True
            if not left and not right:
                return True
            # 没有左或没有右，返回False
            if not left or not right:
                return False
            return left.val == right.val and sub(left.left,right.right) and sub(left.right
        return sub(root.left,root.right)
```

Day50: 连续数组

Day50 这道题，如果第一次做不看答案，就能想到O(n)时间复杂度的解，说明你有算法天赋。这道题确实有一处很技巧的地方。

先看题目

给定一个二进制数组, 找到含有相同数量的 0 和 1 的最长连续子数组（的长度）。

```
示例 1:
输入: [0,1]
输出: 2
说明: [0, 1] 是具有相同数量0和1的最长连续子数组。
```

```
示例 2:
输入: [0,1,0]
输出: 2
说明: [0, 1] (或 [1, 0]) 是具有相同数量0和1的最长连续子数组。
```

注意: 给定的二进制数组的长度不会超过50000。

补充下面代码

```
class Solution(object):
       def findMaxLength(self, nums):
              pass
```

分析

关注下面的序列：

前四个元素序列[1,0]和[0,1]都是最长的满足题意的序列。

这道题，最难想的就是把0元素看为-1，一旦想到这点，瞬间此题就会变得明朗起来。

遇到1，count加1，遇到0，count加-1：

等遍历到箭头所指元素1时，count又变为2，因为index=1时，count已经为2，现在又为2则说明中间经过的所有元素(0和1)恰好能抵消，也就是找到了一个满足题意的序列：

继续遍历求count，等count再次为2时，说明count上次等于2与本次等于2间的所有元素0和1恰好能抵消，并且能和上步的count等于两个2的区间合并求和：

count等于1合并：

count等于1再合并：

最终满足题意的区间长度等于$len(nums)-1$

代码：

有了以上分析，代码就不难写出来：

```
def findMaxLength(nums):
    count,maxlen=0,0
    d = {0:-1}
    for i,num in enumerate(nums):
        if num==0:
            count += -1
        else:
            count += 1

        if count in d:
            maxlen = max(maxlen,i-d[count]) # d[count] : 第一次等于count的位置
        else:
            d[count]=i
    return maxlen
```

Day51：连续最长子串

1 题目

给你一个字符串 s ，请你返回满足以下条件的最长子字符串的长度：每个元音字母，即 'a'，'e'，'i'，'o'，'u' ，在子字符串中都恰好出现了偶数次。

示例 1：

输入：s = "eleetminicoworoep"
输出：13
解释：最长子字符串是 "leetminicowor" ，它包含 e，i，o 各 2 个，以及 0 个 a，u 。

示例 2：

输入：s = "leetcodeisgreat"
输出：5
解释：最长子字符串是 "leetc" ，其中包含 2 个 e 。

示例 3：

输入：s = "bcbcbc"
输出：6
解释：这个示例中，字符串 "bcbcbc" 本身就是最长的，因为所有的元音 a，e，i，o，u 都出现了 0 次。

提示：

1 <= s.length <= 5 x 10^5
s 只包含小写英文字母。

来源：力扣（LeetCode）
链接：https://leetcode-cn.com/problems/find-the-longest-substring-containing-vowels-in-even-counts

2 分析过程

创建一个状态机：

- 对于非元音字符放置到位置0处，

- 元音字符`'a'`放置到位置1处，
- 元音字符`'e'`放置到位置2处，
- 元音字符`'i'`放置到位置4处，
- 元音字符`'o'`放置到位置8处，
- 元音字符`'u'`放置到位置16处

元音字符之所以放到1，2，4，8，16，是要为位运算创造条件，二进制表示中这些数字都只有1位为1，其他位置都为0. 很明显，为1的位置是标志位。

以处理`leetcode`字符串为例：

状态机有如下6个取值，非元音字符放置到0处：

处理第二个字符`e`时，放置到2处：

第三个字符又是`e`，再次放置到2处：

下面又是两个非元音字符，到字符`c`为止，字符串`leetc`就是满足题意(单个元音字符出现偶数次)的最大子字符串。

判断元音字符出现偶数次的方法：二进制表示下，且6个值(0,1,2,4,8,16)都只有一个位为1，所以使用异或运算，某个元音字符出现偶数次时，此位最终状态必然为0；奇数次时最终值必然为1.

接下来，处理下一个字符`o`，但是后面没有字符`o`，只出现1次，不满足题意：

接下来一样方法处理剩余字符，所以整个字符串满足题意的最长子串为：`leetc`

如果字符串修改为：`leetcodoe`，满足题意的最长子串：`leetcodo`.

3 代码

基于以上分析，再看下面代码就不难理解：

```python
class Solution(object):
    def findTheLongestSubstring(self,s):
        state, statedict =  0, {0:-1} #设置此初始值决定下面的代码 i-statedict[state] 这样写
        maxlen = 0
        codedict = {'a':1,'e':2,'i':4,'o':8,'u':16}
        for i,c in enumerate(s):
            if c in codedict:
                state ^= codedict[c]
            if state in statedict:
                maxlen = max(maxlen,i-statedict[state])
            else:
                statedict[state] = i # 记忆新的状态值，二进制位下，可能会出现类似"第1位或第3位为1"的
        return maxlen
```

`statedict` 设置 `{0:-1}` 初始值，也是很有讲究、很巧妙的，决定了下面的代码 `i-statedict[state]` 这样写

`statedict[state] = i` 记忆新的状态值，二进制位下，可能会出现类似**第1位或第3位为1**的32种组合。

4 扩展

今天题目与Day50的思路极为类似，可以归纳为**前缀和**问题，关键是对状态的区分，记忆某种状态，中间经历某种变换或抵消操作后，出现了状态字典里的某个状态，表明找到满足题意的前缀。

比如，字符串 `lee`，第一个状态是 `l`，第二个是 `le`，第三个状态又是 `l`，因为2个 `e` 能抵消。因此，满足题意的最长子串长度为3.

字符串 `oeo`，第一个状态是 `o`，第二个状态 `oe`，第三个状态是 `e`，两个 `o` 抵消，因此没有重复状态。因此，满足题意的最长子串长度为0.

Day52：二叉树坡度

给定一个二叉树，计算整个树的坡度。

一个树的节点的坡度定义即为，该节点左子树的结点之和和右子树结点之和的差的绝对值。空结点的的坡度是0。

整个树的坡度就是其所有节点的坡度之和。

示例：

```
输入：
         1
       /   \
      2     3
输出：1
解释：
结点 2 的坡度: 0
结点 3 的坡度: 0
结点 1 的坡度: |2-3| = 1
树的坡度 : 0 + 0 + 1 = 1
```

```
提示：

任何子树的结点的和不会超过 32 位整数的范围。
坡度的值不会超过 32 位整数的范围。
通过次数14,300提交次数25,580

来源：力扣（LeetCode）
链接：https://leetcode-cn.com/problems/binary-tree-tilt
著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。
```

补充代码：

```python
# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution(object):
    def findTilt(self, root):
        """
        :type root: TreeNode
        :rtype: int
```

分析过程

从二叉树定义出发：

该节点左子树的结点之和和**右子树结点之和**的**差的绝对值**，所以比较容易就能找到递推关系式：

$$tilt(root) = tilt(root.left) + tilt(root.right)

- |\sum{root.left - \sum root.right}| $$

而求左、右子树的节点和，很容易由递推关系式：

$$sum(root) = sum(root.left) + sum(root.right) + root.val$$

求得。所以综合以上两个递推关系式发现，它们的递推关系式是一样的，并且递归基也十分相似：

1. $$|\sum root.left - \sum root.right|$$
2. $$root.val$$

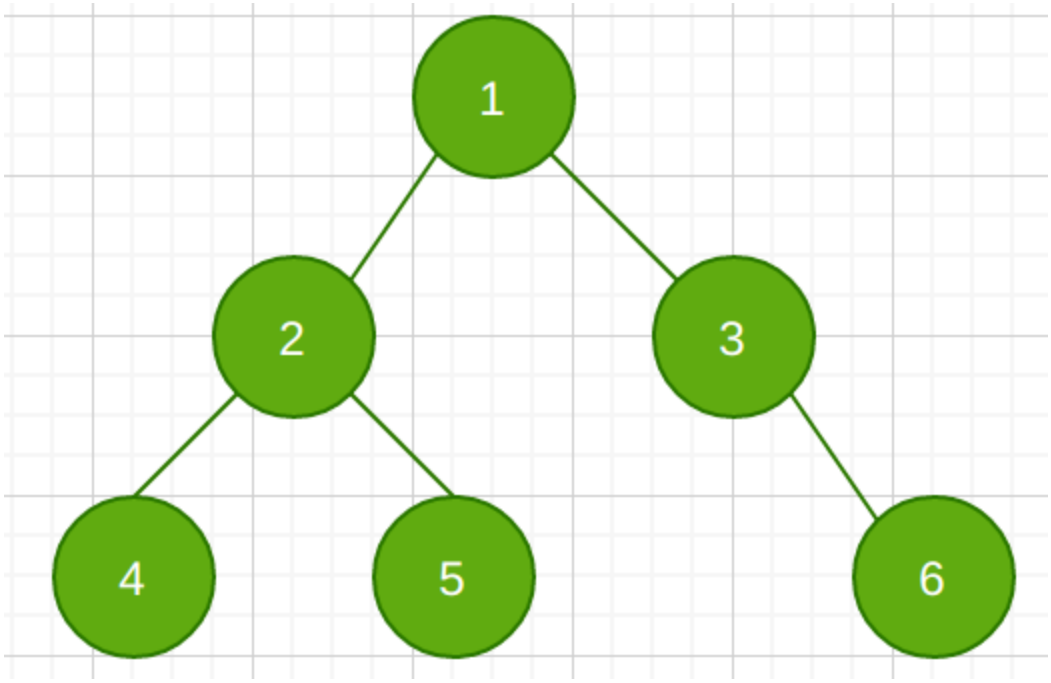
代码

因此，基于以上分析，就不难写出以下代码：

```
class Solution(object):
    def findTilt(self, root):
        self.tilt = 0
        # 求二叉树节点和，再经过一步转化
        def treeSum(root):
            if not root:
                return 0
            leftSum = treeSum(root.left)
            rightSum = treeSum(root.right)
            self.tilt += abs(leftSum-rightSum)
            return leftSum + rightSum + root.val
        treeSum(root)
        return self.tilt
```

Day53：使用迭代，完成对二叉树的层序遍历

再来解释一下二叉树层序遍历，如下二叉树层序遍历的结果为：`[[1],[2,],[4,5,6]]`



分析

为什么说二叉树层序遍历更加重要或者更有实际意义？

相比于前、中、后序遍历，层序遍历更有实际意义。

下面参考星友莱布妮子的总结：

二叉树的层序遍历方法就是图的广度优先搜索方法，所以搞清楚了二叉树层序遍历，再学习图的广度优先就会变得容易，而深度或广度优先遍历实际中使用更多。

二叉树层序遍历用队列实现，实现思路就是广度优先搜索步骤：

1.找出当前顶点的所有邻接点。如果有哪个是没访问过的，就把它标为“已访问”，并且将它入队。（尽管该顶点并未作为“当前顶点”被访问过。）

2.如果当前顶点没有未访问的邻接点，且队列不为空，那就再从队列中移出一个顶点作为当前顶点。

3.如果当前顶点没有未访问的邻接点，且队列里也没有其他顶点，算法完成。

实现代码

代码参考星友莱布妮子，代码写的很漂亮，也很高效：

```
def levelOrder(self, root):
        # 树的广度优先搜索（层次遍历），使用队列或双向队列
      if not root:
          return []

      search_queue = deque()
      search_queue.append(root)
      result = []

      while search_queue:
          visited = []
          # 找出当前顶点的所有邻接点
          for i in range(len(search_queue)):
          # 如果当前顶点没有未访问的邻接点，且队列不为空，那就再从队列中移出一个顶点作为当前顶点
              node = search_queue.popleft()
              visited.append(node.val)
              if node.left:
                  search_queue.append(node.left)
              if node.right:
                  search_queue.append(node.right)

          result.append(visited)

      return result
```

Day54: 判断是否为平衡二叉树

1 题目

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。

```
示例 1:

给定二叉树 [3,9,20,null,null,15,7]

    3
   / \
  9  20
    /  \
   15   7
返回 true 。
```

```
示例 2:

给定二叉树 [1,2,2,3,3,null,null,4,4]

       1
      / \
     2   2
    / \
   3   3
  / \
 4   4
返回 false 。
```

来源：https://leetcode-cn.com/problems/balanced-binary-tree/

著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。

2 分析

这道题与求二叉树坡度等都很相似，也是从平衡二叉树的定义开始：

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，每个节点都满足就是平衡二叉树。

根据上方定义，很容易得到下方递推关系式：

```
f(root) = isRootBalanced
          且 f(root.left)
          且 f(root.right)
```

即：根节点左右子树高度差绝对值不超过1，左右子树也满足平衡。

如何求节点高度，前面已经刷过此题，直接给出递推关系式：

```
h(node) = 1 + max(h(node.left),h(node.right))
```

即：节点高度等于1加上节点左右子树的较大高度。

至此，此题分析完毕，下面直接写代码即可：

3 代码

```
class Solution(object):
    def isBalanced(self, root):
        if not root:
            return True
        # 求节点高度，也就是第二个递推关系式
        def getHeight(root):
            if not root:
                return True
            return 1 + max(getHeight(root.left),getHeight(root.right))
        # 第一个递推关系式
        return abs(getHeight(root.left) - getHeight(root.right)) <=1 and self.isBalanced(r
```

以上求法时间复杂度：O(n),空间复杂度主要来自函数栈开销，与树的节点数有线性关系。

Day55: 一维数组的动态和

1 题目

给你一个数组 nums 。数组「动态和」的计算公式为：runningSum[i] = sum(nums[0]…nums[i]) 。

请返回 nums 的动态和。

```
示例 1：

输入：nums = [1,2,3,4]
输出：[1,3,6,10]
解释：动态和计算过程为 [1, 1+2, 1+2+3, 1+2+3+4] 。
```

示例 2：

```
输入：nums = [1,1,1,1,1]
输出：[1,2,3,4,5]
解释：动态和计算过程为 [1, 1+1, 1+1+1, 1+1+1+1, 1+1+1+1+1] 。
```

示例 3：

```
输入：nums = [3,1,2,10,1]
输出：[3,4,6,16,17]
```

提示：

1 <= nums.length <= 1000 -10^6 <= nums[i] <= 10^6

来源：力扣（LeetCode）链接：https://leetcode-cn.com/problems/running-sum-of-1d-array/ 著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。

2 分析

动态和的定义即通项公式：

runningSum[i] = sum(nums[0]…nums[i])
动态和数组的第i项等于前i项的和，所以创建一个临时标量s记忆当前i项的和，动态和数组的第i+1项等于：

runningSum[i+1] = sum(nums[0]…nums[i]+nums[i+1])

= s + nums[i+1]
根据上述分析便能写出代码。

3 代码

```
class Solution(object):
    def runningSum(self, nums):
        if not nums:
            return []
        # r先申请好n个位置，防止append时发生扩容耗费时间
        r,s = [0]*len(nums), 0

        for i,num in enumerate(nums):
            s += num
            r[i] = s
        return r
```

Day56: 判断二叉树是否为二叉搜索树

1 题目

首先给出二叉搜索树的定义：

若对于树中任一节点`r`，左（右）子树中的节点（若存在）均不大于（不小于）`r`，则称之为二叉搜索树（binary search tree），简称为 BST，简而言之，即处处满足这种顺序性的二叉树为二叉搜索树。

如下所示：

2 题目分析

注意此题是我构思的，与leetcode上对二叉搜索树定义稍有区别，所以代码也会有一点区别。

解决此题，还是要从BST的定义出发分析，它的递推关系式如下：

```
f(root) =
 root.left.val <= root.val <= root.right.val
 && f(root.left)
 && f(root.right)
```

根节点不小于左子树且不大于右子树。

3 代码

```
class TreeNode():
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None


def isBinarySortedTree(root):
    if not root:
        return Tree
    return isBST(root)

    def isBST(root,lower=float('-inf'),upper=float('inf')):
        # 二叉树的定义
        if root.val <= lower or upper <= root.val:
            return False

        res = True
        if root.left:
            res = isBST(root.left,lower,root.val) # 左子树不大于root
        if res and root.right:
            res = isBST(root.left,root.val,upper) # 右子树不小于root
        return res
```

可以看到细心的几位星友都发现了此题与leetcode的区别：

类似与Leetcode的98题，只是要求有点不同，就是一个有等号，一个没等号。具体可以看P1和振哥发的pdf。

这道题要注意的就是当前节点的值的取值范围会影响到其子树的节点值的范围，所以用来递归的函数增加了两个参数，来限定取值范围。

Day57 二叉搜索树的查找

1 二叉搜索树定义

首先给出二叉搜索树的定义：

若对于树中任一节点r，左（右）子树中的节点（若存在）均不大于（不小于）r，则称之为二

叉搜索树（binary search tree），简称为 BST ，简而言之，即处处满足这种顺序性的二叉树为二叉搜索树。

如下所示：

2 二叉搜索树查找

根据上文定义，根节点和左右子树可能出现重复元素，所以可能会返回多个关键码值，都返回即可。大家注意这点。

儿查搜索树的查找与二分查找极为相似，大家不妨想一想，每比较一次都能将有效区间长度减半。

3 代码

注意二叉树节点定义为如下：

```
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None
```

如果`target`小于root.val，则一定位于左侧分支，反之位于右分支；如果等于`root.val`，则加入到返回列表中，**但是还要继续查找**，而不是立即返回。

```
def searchNode(root: TreeNode, target: int)->List[TreeNode]:
    """
    root: 二叉搜索树的根节点
    target: 节点的val值
    返回值：等于val的所有节点
    """
    r = []
    def search(root,target):
        if not root:
            return r
        if target == root.val:
            r.append(target)
            search(root.left,target)
            search(root.right,target)
        elif target < root.val:
            search(root.left,target)
        else:
            search(root.right,target)

    search(root,target)
```

Day58: 二叉搜索树第K大节点

1 题目

给定一棵二叉搜索树，请找出其中第k大的节点。

```
示例 1:

输入: root = [3,1,4,null,2], k = 1
   3
  / \
 1   4
  \
   2
输出: 4
```

```
示例 2:

输入: root = [5,3,6,2,4,null,null,1], k = 3
       5
      / \
     3   6
    / \
   2   4
  /
 1
输出: 4
```

限制：

1 ≤ k ≤ 二叉搜索树元素个数

2 题目分析

元素投影到x轴上后，就是按照从小到大排序好的顺序，自然的第K大就是nums[-k]

```
      5
     / \
    3   6
   / \
  2   4
 /
1
```

```
nums = [1 2 3 4 5 6]
```

二叉树的中序遍历恰好形成上面的序列。

所以本题非常直观的一个思路，中序遍历保存到一个list，遍历完成后l[-k]就是第K大元素。但是这种思路不光多消耗内存，还费时间。

试想，有必要存储吗？有必要全部便利一遍才找出第`k`大吗？

都是没有必要的！

二叉搜索树中，第`1`大到第`len(nums-1)/2`大一定位于右子树中，换句话说，如果`k`值满足：

$$1 \leq k \leq len(nums - 1)/2$$

则只遍历右子树就能找到第`k`大元素；

如果`k`值满足：

$$k > len(nums - 1)/2$$

才有必要去左子树中去查找；

3 代码

```
class Solution(object):

    def kthLargest(self, root, k):
        self.count = k
        self.found = False # 已经找到为True

        def travel(node):
            if not node or self.found:
                return
            # 遍历右子树
            travel(node.right)

            self.count -= 1
            if self.count == 0:
                self.result = node.val
                self.found = True
                return

            # 如果k > (len(nums)-1)/2，遍历左子树
            if not self.found: # 没有找到再去左子树
                travel(node.left)

        travel(root)

        return self.result
```

以上代码加入一个self.found标志位，过滤一些非必要的递归操作。

Day 59: 二叉搜索树的最近公共祖先

1 题目

百度百科中最近公共祖先的定义为：“对于有根树 T 的两个结点 p、q，最近公共祖先表示为一个结点 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，给定如下二叉搜索树: root = [6,2,8,0,4,7,9,null,null,3,5]

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。
```

说明:

- 所有节点的值都是唯一的。
- p、q 为不同节点且均存在于给定的二叉搜索树中。

2 分析

对于二叉搜索树，已知两个节点，寻找最近祖先节点，根据下面的递推关系式：

若 node.val < min(p.val,q.val)，则p和q的最近祖先节点一定在右子树；

若 max(p.val,q.val) < node.val，则p和q的最近祖先节点一定在左子树；

其他情况 node.val位于p.val和q.val间(可能等于node.val)，则node就是p和q的最近祖先。

3 代码

```
class Solution(object):
    def lowestCommonAncestor(self, root, p, q):
        if root.val < min(p.val,q.val):
            return self.lowestCommonAncestor(root.right,p,q)
        elif max(p.val,q.val) < root.val:
            return self.lowestCommonAncestor(root.left,p,q)
        else:
            return root
```

Day60：删除二叉搜索树的某个节点

1 题目

给定一个二叉搜索树的根节点 root 和一个值 key，删除二叉搜索树中的 key 对应的节点，并保证二叉搜索树的性质不变。**返回二叉搜索树（有可能被更新）的根节点的引用**。

一般来说，删除节点可分为两个步骤：

- 首先找到需要删除的节点；
- 如果找到了，删除它。

说明： 要求算法时间复杂度为 O(h)，h 为树的高度。

示例:

root = [5,3,6,2,4,null,7]

key = 3

```
    5
   / \
  3   6
 / \   \
2   4   7
```

给定需要删除的节点值是 3，所以我们首先找到 3 这个节点，然后删除它。

一个正确的答案是 [5,4,6,2,null,null,7], 如下图所示。

```
    5
   / \
  4   6
 /     \
2       7
```

另一个正确答案是 [5,2,6,null,4,null,7]。

```
    5
   / \
  2   6
   \   \
    4   7
```

2 分析过程

这道题被leetcode定为中等难度级别，实话讲确实属于这类级别，虽然思路不难，但是要想一次写出准确无误的代码，仍然不是一件简单的事情。如果你第一次做这道题能很快写出来，说明你对递归的理解、指针的掌控都达到了一定水平。

这道题的思路很straitforward，根据BST的性质，具体说来分为如下几种情况：

1. 如果被删除节点关键码key小于当前根节点nodei.val，则问题规模直接降阶到左子树中
2. 关键码key大于当前根节点nodei.val，直接到右子树中查找
3. key等于当前根节点nodei.val，又分为4种情况：
   (1). nodei 无左右子树，摘除nodei节点，等于直接返回 None
   (2). nodei 仅有左子树，摘除nodei节点，等于直接返回 nodei.left
   (3). nodei 仅有右子树，摘除nodei节点，等于直接返回 nodei.right
   (4). nodei 都有左右子树，麻烦一点，方法之一：选择以nodei.right为根节点的树中最左侧节点，然后替换nodei

最后返回 nodei.

你看，上面的思路应该足够清晰，但是兑现为代码绝对又是另一回事。你首先要对递归有深刻的理解，其次像链表、二叉树等这类具备递归的数据结构，操作它们节点引用问题要时刻保持清醒，很容易出错。

3 如何写出代码

老规矩，这是我们的节点定义：

```
# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None
```

解决方案对象的主方法，调用自定义方法`self.__delNodei(root,key)`，这个方法的构思思路是这样：

第一个参数是BST树中的任意节点，因为BST严格满足递归，所以选取任意一个以节点nodei为根的树，删除里面等于key的节点。因此，这个功能一旦实现后，只需在参数赋值时赋值它为root即可。

```
class Solution(object):
    def deleteNode(self, root, key):
        """
        :type root: TreeNode
        :type key: int
        :rtype: TreeNode
        """
        return self.__delNodei(root,key)
```

所以关键是如何写出`def __delNodei(self,nodei,key)`方法，下面我们一步一步分析。

根据上面的4种情况分析，我们在情况4时会用到找树的最左节点，为此先写出这个方法`def __findMinNode(self,nodei)`：

```
    # 找到以nodei为根节点树的最小值
    def __findMinNode(self,nodei):
        if not nodei.left:
            return nodei
        while nodei.left:
            nodei = nodei.left
        return nodei
```

这个比较简单，是链表、二叉树等递归结构的常规迭代思路，注意与向量表i=i+1迭代思路的区

别。

先写出第一种大情况，比较简单。因为方法`self.__delNodei(nodei.left,key)`实现删除等于key的节点后返回对nodei.left节点的引用，所以将nodei的left域指向它即可。

```
    # 假定已经查询到nodei节点，删除后返回nodei节点的引用
    def __delNodei(self,nodei,key):
        if not nodei:
            return None

        #若满足下面条件，一定在左子树
        if key < nodei.val:
            nodei.left = self.__delNodei(nodei.left,key) # 删除后返回nodei.left节点的引用
```

再写出第二种大情况，与上类似：

```
# 一定在右子树
        elif nodei.val < key:
            nodei.right = self.__delNodei(nodei.right,key)# 删除后返回nodei.right节点的引用
```

再写出第三种大情况，即找到了等于key节点，又分四种小情况：

被删除节点是叶子节点：直接返回None，就是摘除它了：

```
 # nodei.val== key，删除nodei
        else:
            # 情况1：被删除节点是叶子节点
            if not nodei.left and not nodei.right:
                return None  # 置为None
```

第二、三种小情况很相似，跳过被删除节点nodei，直接返回nodei.left 或 node.right 即可：

```
            # 情况2：被删除节点无右子树
            if not nodei.right:
                return nodei.left # 跳过nodei,返回指向nodei.left，相当于删除了nodei
            # 情况3：被删除节点无左子树
            if not nodei.left:
                return nodei.right
```

最后一种小情况，大家直接看注释吧：

```
            # 情况4：被删除节点左右子树都不为空
            # 先找到右子树中最左节点，即右子树最小值节点
            minNodei = self.__findMinNode(nodei.right)
            nodei.val = minNodei.val
            # 删除minNodei，同时返回nodei.right的引用，并赋值给nodei.right
            ## 切记：key已经不是__delNodei的参数key，而是我们找到的minNodei的val值
            nodei.right = self.__delNodei(nodei.right,minNodei.val)

        return nodei
```