226. 翻转二叉树

问题描述

给你一棵二叉树的根节点 `root`，翻转这棵二叉树，并返回其根节点。

示例

示例 1

输入：

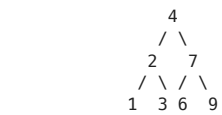
$$\text{root} = [4, 2, 7, 1, 3, 6, 9]$$

输出：

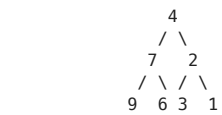
$$[4, 7, 2, 9, 6, 3, 1]$$

树的结构如下：

- 输入：



- 输出：



示例 2

输入：

$$\text{root} = [2, 1, 3]$$

输出：

$$[2, 3, 1]$$

树的结构如下：

- 输入：

- 输出：

示例 3

输入：

$$\text{root} = []$$

输出：

$$[]$$

提示

- 树中节点数目范围在 $[0, 100]$ 内。
- 每个节点的值范围为 $-100 \leq \text{Node.val} \leq 100$

。

In [ ]:

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

这道题目使用前序遍历和后序遍历都可以，唯独中序遍历不方便，因为中序遍历会把某些节点的左右孩子翻转了两次！

In [2]:
```python
# 递归
# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def invertTree(self, node: TreeNode) -> TreeNode:
        if not node:
            return None
        # 交换左右子树
        node.left, node.right = node.right, node.left
        # 递归地翻转子树
        self.invertTree(node.left)
        self.invertTree(node.right)
        return node
```

In [4]:
```python
# 递归. 后序遍历

class Solution:
    def invertTree(self, node: TreeNode) -> TreeNode:
        if not node:
            return None

        self.invertTree(node.left)
        self.invertTree(node.right)

        node.left, node.right = node.right, node.left
        return node
```

In [3]:
```python
# 迭代, 前序遍历

class Solution:
    def invertTree(self, root: TreeNode) -> TreeNode:
        if not root:
            return None
        stack = [root]

        while stack:
            node = stack.pop()
            node.left, node.right = node.right, node.left

            if node.left:
                stack.append(node.left)

            if node.right:
                stack.append(node.right)
        return root
```

In [5]:
```python
# 迭代, 层序遍历
from collections import deque

class Solution:
    def invertTree(self, root: TreeNode) -> TreeNode:
        if not root:
            return None

        queue = deque([root])
        while queue:
            node = queue.popleft()
            node.left, node.right = node.right, node.left
            if node.left: queue.append(node.left)
            if node.right: queue.append(node.right)
        return root
```

101. 对称二叉树

给你一个二叉树的根节点 `root`，检查它是否轴对称。

示例 1：

输入：
`root = [1, 2, 2, 3, 4, 4, 3]`
输出：
`true`

示例 2：

输入：
`root = [1, 2, 2, null, 3, null, 3]`
输出：
`false`

提示：

- 树中节点数目在范围 `[1, 1000]` 内
- `-100 <= Node.val <= 100`

In [ ]:
```python
# 递归

class Solution:
    def isSymmetric(self, root: Optional[TreeNode]) -> bool:
        # base case
        if not root:
            return True

        return self.compare(root.left, root.right)

    def compare(self, left, right):
        # 判断本层
        # 空节点
        if left == None and right != None: return False
        elif left != None and right == None: return False
        elif left == None and right == None: return True
        # 数值不同
        elif left.val != right.val: return False

        # 进一步判断下一层
        result = self.compare(left.left, right.right) and self.compare(left.right, right.left)
        return result
```

In [ ]:
```python
# 迭代: 队列

import collections

class Solution:
    def isSymmetric(self, root: TreeNode) -> bool:

        if not root:
            return True

        queue = collections.deque()
        queue.append(root.left) #将左子树头结点加入队列
        queue.append(root.right) #将右子树头结点加入队列

        while queue: #接下来就要判断这这两个树是否相互翻转

            leftNode = queue.popleft()
            rightNode = queue.popleft()
            if not leftNode and not rightNode: #左节点为空、右节点为空，此时说明是对称的
                continue

            #左右一个节点不为空，或者都不为空但数值不相同，返回false
            if not leftNode or not rightNode or leftNode.val != rightNode.val:
                return False

            queue.append(leftNode.left) #加入左节点左孩子
            queue.append(rightNode.right) #加入右节点右孩子
            queue.append(leftNode.right) #加入左节点右孩子
            queue.append(rightNode.left) #加入右节点左孩子

        return True
```

In [ ]:
```python
# 迭代，栈

class Solution:
    def isSymmetric(self, root: TreeNode) -> bool:

        if not root:
            return True

        stack = [] #这里改成了栈，本质上就是一个list
        stack.append(root.left)
        stack.append(root.right)

        while stack:

            rightNode = stack.pop()
            leftNode = stack.pop()

            if not leftNode and not rightNode:
                continue
            if not leftNode or not rightNode or leftNode.val != rightNode.val:
                return False
```

```
            stack.append(leftNode.left)
            stack.append(rightNode.right)
            stack.append(leftNode.right)
            stack.append(rightNode.left)

        return True
```

In [ ]:
```
# 迭代, 栈

class Solution:
    def isSymmetric(self, root: TreeNode) -> bool:
        if not root:
            return True

        queue = collections.deque([root.left, root.right])

        while queue:
            level_size = len(queue)

            if level_size % 2 != 0:
                return False

            level_vals = []
            for i in range(level_size):
                node = queue.popleft()
                if node:
                    level_vals.append(node.val)
                    queue.append(node.left)
                    queue.append(node.right)
                else:
                    level_vals.append(None)

            if level_vals != level_vals[::-1]:
                return False

        return True
```

222. 完全二叉树的节点个数

给你一棵完全二叉树的根节点 `root`，求出该树的节点个数。

完全二叉树的定义：
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。若最底层为第 h 层（从第 0 层开始），则该层包含 1 到 2^h 个节点。

示例 1

输入：
`root = [1, 2, 3, 4, 5, 6]`
输出：
`6`

示例 2

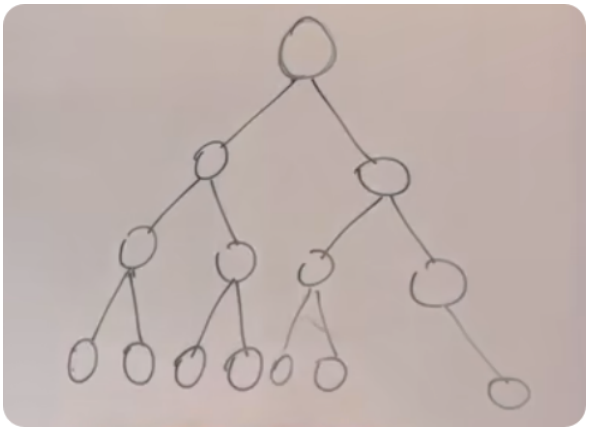
输入：
`root = []`
输出：
`0`

示例 3

输入：
`root = [1]`
输出：
`1`

In [ ]:
```

```



如果只有一个右节点,就不是完全二叉树了

In [ ]:

满二叉树的性质:

节点数量 = 2^层数 - 1

所以通过深度优先搜索,遍历最左边的节点,如果最深的子树是满二叉树, 那么该层的层树就可以直接计算出节点数量

一条线一直往左递归

另一条线一直往右递归

如果两条线的节点数量相等, 那么就是完全二叉树

In [ ]:

```python
class Solution:
    def countNodes(self, root: TreeNode) -> int:
        if not root:
            return 0
        left = root.left
        right = root.right
        leftDepth = 0 #这里初始为0是有目的的，为了下面求指数方便
        rightDepth = 0
        while left: #求左子树深度
            left = left.left
            leftDepth += 1
        while right: #求右子树深度
            right = right.right
            rightDepth += 1
        if leftDepth == rightDepth:
            return (2 << leftDepth) - 1 #注意(2<<1) 相当于2^2，所以leftDepth初始为0
        return self.countNodes(root.left) + self.countNodes(root.right) + 1
```

你的代码并没有漏加节点数，即使遇到非完全二叉子树时，它也会递归处理左子树和右子树并正确累加节点数。

以下是代码的逻辑解释：

1. **完全二叉树的特殊处理**：
   如果左子树的深度 `leftDepth` 等于右子树的深度 `rightDepth`，说明当前子树是一个完全二叉树。可以直接使用公式 `(2 << leftDepth) - 1` 来计算节点总数，而无需进一步递归计算。

2. **非完全二叉树的处理**：
   如果左、右子树的深度不相等，则递归调用 `self.countNodes` 分别处理左子树和右子树，并累加根节点（即 `+1`），确保所有节点都被正确计数。

因此，非完全二叉树的节点数也不会漏加，因为在非完全二叉树的情况下，代码进入递归分支 `self.countNodes(root.left) + self.countNodes(root.right) + 1`，从而遍历所有节点。

举例说明

假设一棵非完全二叉树如下：

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

- 初始调用 `countNodes(root)`， `root` 为节点 1。
  - 左子树深度为 2，右子树深度为 1（不相等）。
  - 递归调用 `countNodes(root.left)` 和 `countNodes(root.right)`。
- 处理左子树（根为节点 2）：
  - 左、右子树深度相等，为 1。
  - 节点 2 的子树是完全二叉树，节点总数为 `(2 << 1) - 1 = 3`。
- 处理右子树（根为节点 3）：
  - 左、右子树深度均为 0（没有子节点）。

    - 节点总数为 `1`。
- 最终结果为 `3 + 1 + 1 = 5`。

因此，代码逻辑是正确的，完全覆盖了完全二叉树和非完全二叉树的情况，没有遗漏节点计数。

110. 平衡二叉树

给定一个二叉树，判断它是否是平衡二叉树。

平衡二叉树的定义：
一棵二叉树每个节点的左右子树高度差的绝对值不超过 1。

示例 1

输入：
`root = [3, 9, 20, null, null, 15, 7]`
输出：
`true`

示例 2

输入：
`root = [1, 2, 2, 3, 3, null, null, 4, 4]`
输出：
`false`

示例 3

输入：
`root = []`
输出：
`true`

提示

- 树中的节点数在范围 `[0, 5000]` 内
- `-10^4 <= Node.val <= 10^4`

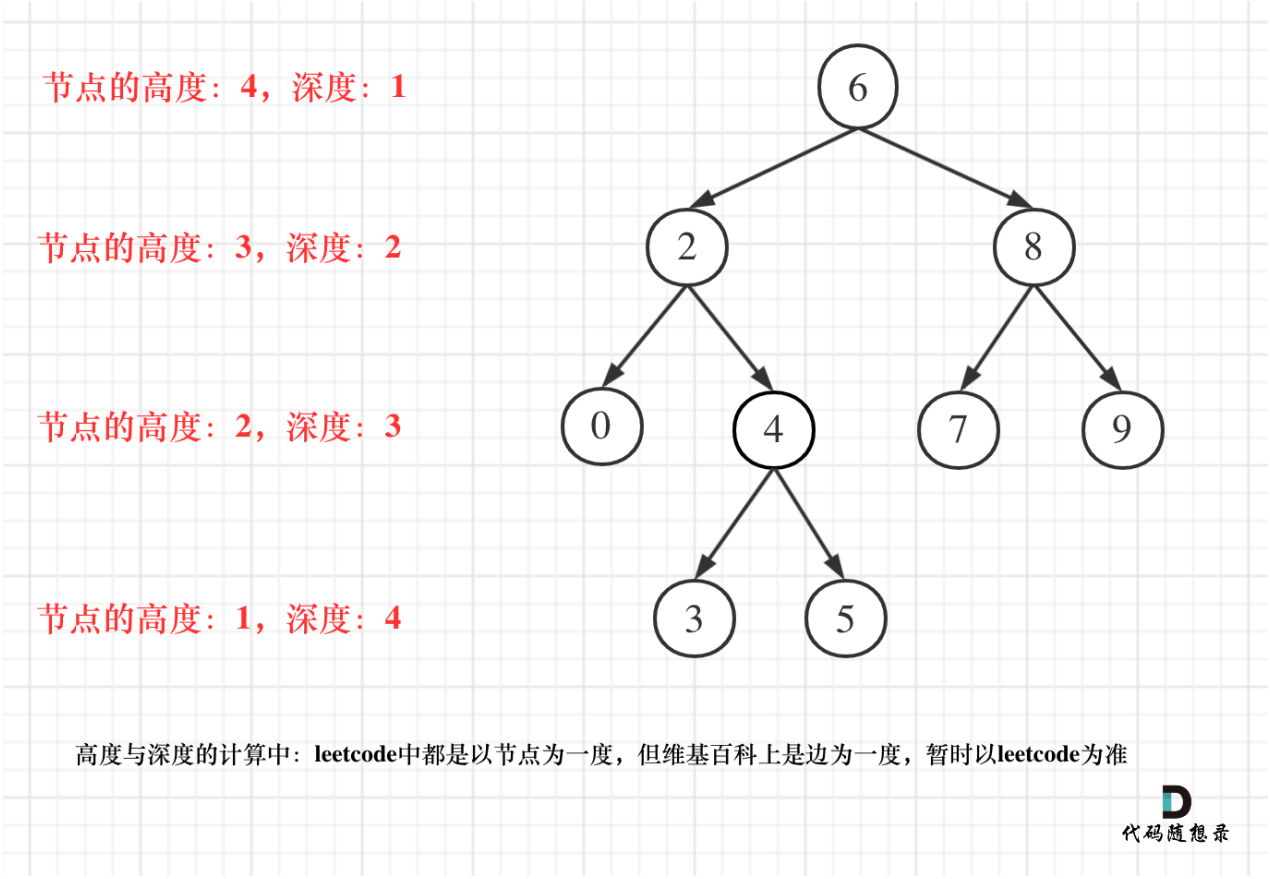
求高度要用 后序遍历

先求左+右的高度 再在 中 进行比较

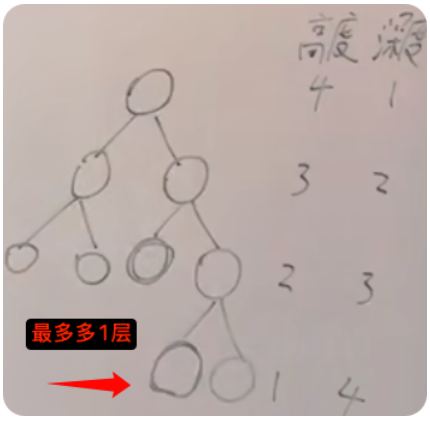
二叉树节点 的 深度：根节点到该节点的最长简单路径边的条数。

二叉树节点 的 高度：该节点到叶子节点的最长简单路径边的条数。

leetcode的题目中根节点深度是1



平衡二叉树概念



递归三部曲

1. 明确递归函数的 参数 和 返回值
   A. 参数: 当前传入节点
   B. 返回值: 以当前传入节点为根节点的树的高度
2. 明确终止条件: 遇到空节点终止
3. 明确单层递归逻辑 : 左子树和右子树高度差值

In [ ]:
```python
# 递归

# Definition for a binary tree node.
from typing import Optional

class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def isBalanced(self, root: Optional[TreeNode]) -> bool:
        if self.get_height(root) != -1:
            return True
```

```
        else:
            return False

    def get_height(self, root: TreeNode) -> int:

        # base case (终止条件)
        if not root:
            return 0 # 空树高度为0

        left_height = self.get_height(root.left)
        right_height = self.get_height(root.right)

        # return -1 if not balanced

        # 左
        if left_height == -1:
            return -1

        # 右
        if right_height == -1:
            return -1

        # 中
        if abs(left_height - right_height) > 1:
            return -1

        # return height if balanced
        return max(left_height, right_height) + 1
```

上面修复了一个错误

```
# base case (终止条件)
if not root:
    return -1 # 空树高度为-1, 这不符合树的高度定义

# base case (终止条件)
if not root:
    return 0 # 空树高度为0
```

代码结构的「左、右、中」含义

1. **左**（处理左子树）

   ```
   left_height = self.get_height(root.left)
   ```
   这部分代码通过递归调用 `self.get_height` 处理当前节点的左子树，计算左子树的高度。如果左子树已经失衡（即返回 `-1`），后续判断会直接返回 `-1`，不再继续。

2. **右**（处理右子树）

   ```
   right_height = self.get_height(root.right)
   ```
   类似于左子树的处理，递归调用 `self.get_height` 计算右子树的高度。如果右子树已经失衡（即返回 `-1`），也会直接返回 `-1`。

3. **中**（处理当前节点）

   ```
   if abs(left_height - right_height) > 1:
       return -1
   ```
   这部分代码处理当前节点，判断左右子树的高度差是否超过 1。如果高度差大于 1，说明当前节点不平衡，返回 `-1`。
   如果当前节点平衡，则返回当前节点的高度：

   ```
   return max(left_height, right_height) + 1
   ```

总结

- **左**：递归计算左子树的高度，检查左子树是否平衡。
- **右**：递归计算右子树的高度，检查右子树是否平衡。
- **中**：检查当前节点的左右子树高度差是否超过 1，并返回节点高度或标记为不平衡。

这种「左、右、中」的遍历模式是后序遍历（post-order traversal），因为在处理当前节点之前，先处理了左子树和右子树。

为什么是后序遍历

之所以称这段代码的遍历为**后序遍历**（post-order traversal），是因为在递归过程中，对每个节点的逻辑顺序是：

1. **先处理左子树**（递归计算左子树高度）。
2. **再处理右子树**（递归计算右子树高度）。
3. **最后处理当前节点**（检查是否平衡，并计算当前节点高度）。

为什么是后序遍历?

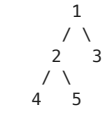
后序遍历的核心特点是：**当前节点的逻辑处理依赖于左右子树的结果**。

- 在这里，只有左子树和右子树都返回了各自的高度（或者确定是否平衡），才能对当前节点进行处理。

- 所以，这段代码的执行顺序是典型的后序遍历：**左 → 右 → 当前节点**。

示例

假设树如下：



遍历顺序：

1. 递归到节点 `4`，处理完左右子树后，返回高度。
2. 回到节点 `5`，处理完左右子树后，返回高度。
3. 处理节点 `2`（此时左右子树都已处理完）。
4. 转到右子树，递归处理节点 `3`。
5. 最后处理根节点 `1`。

遍历顺序为：`4 → 5 → 2 → 3 → 1`，符合后序遍历的特性。

对比前序和中序

前序遍历（Pre-order Traversal）

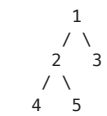
顺序：中 → 左 → 右
处理顺序：先处理当前节点，然后递归处理左子树，最后递归处理右子树。

伪代码

```python
def pre_order_traversal(node):
    if not node:
        return
    # 中：处理当前节点
    print(node.val)  # 或进行其他操作
    # 左：递归处理左子树
    pre_order_traversal(node.left)
    # 右：递归处理右子树
    pre_order_traversal(node.right)
```

执行示例

给定二叉树：



遍历顺序：`1 → 2 → 4 → 5 → 3`

- 先处理根节点 `1`，然后依次处理左子树（`2 → 4 → 5`），最后处理右子树（`3`）。

中序遍历（In-order Traversal）

顺序：左 → 中 → 右
处理顺序：先递归处理左子树，然后处理当前节点，最后递归处理右子树。

伪代码

```python
def in_order_traversal(node):
    if not node:
        return
    # 左：递归处理左子树
    in_order_traversal(node.left)
    # 中：处理当前节点
    print(node.val)  # 或进行其他操作
    # 右：递归处理右子树
    in_order_traversal(node.right)
```

执行示例

给定二叉树：

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

遍历顺序：`4 → 2 → 5 → 1 → 3`

- 先处理左子树，得到 `4 → 2 → 5`；然后处理根节点 `1`；最后处理右子树 `3`。

对比总结

遍历类型	处理顺序	示例顺序（树结构见上）	关键点
前序遍历	中 → 左 → 右	`1 → 2 → 4 → 5 → 3`	先处理当前节点，再递归处理子树。
中序遍历	左 → 中 → 右	`4 → 2 → 5 → 1 → 3`	左子树完全处理后才处理当前节点。
后序遍历	左 → 右 → 中	`4 → 5 → 2 → 3 → 1`	左右子树都处理完才处理当前节点。

In [ ]:
```python
# 迭代法, 用stack,层序遍历
# 这个不好记, 就先放弃

class Solution:
    def getDepth(self, cur):
        stack = []
        if cur is not None:
            stack.append(cur)
        depth = 0
        result = 0
        while stack:
            node = stack[-1]
            if node is not None:
                stack.pop()
                stack.append(node)                  # 中
                stack.append(None)
                depth += 1
                if node.right:
                    stack.append(node.right)        # 右
                if node.left:
                    stack.append(node.left)         # 左

            else:
                node = stack.pop()
                stack.pop()
                depth -= 1
            result = max(result, depth)

        return result

    def isBalanced(self, root):
        stack = []
        if root is None:
            return True
        stack.append(root)
        while stack:
            node = stack.pop()                      # 中
            if abs(self.getDepth(node.left) - self.getDepth(node.right)) > 1:
                return False
            if node.right:
                stack.append(node.right)            # 右（空节点不入栈）
            if node.left:
                stack.append(node.left)             # 左（空节点不入栈）
        return True
```

In [ ]:
```python

```

257. 二叉树的所有路径

题目描述

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 1

输入:
`root = [1,2,3,null,5]`

输出:
`["1->2->5","1->3"]`

示例 2

输入:

```
root = [1]
```

输出:

```
["1"]
```

In [ ]:

```python
# 递归 + 回溯
# 这道题有点难, 需要想象力
# 之后再复习

class Solution:
    def binaryTreePaths(self, root: Optional[TreeNode]) -> List[str]:
        result = []
        path = []
        if not root:
            return result
        self.traversal(root, path, result)
        return result

    def traversal(self, cur, path, result):
        path.append(cur.val)  # 中  # 这里用一个path在每次递归的时候都记录下当前节点

        if not cur.left and not cur.right:  # 到达叶子节点
            sPath = '->'.join(map(str, path))
            result.append(sPath) # 到达叶子节点, 就把当前路径加入到result里面
            return
        if cur.left:  # 左
            self.traversal(cur.left, path, result)
            path.pop()  # 回溯 # 这里巧妙. 在归的过程中, 将path的最后一位踢出去, 最后回到root节点的时候, path就是空的了
        if cur.right:  # 右
            self.traversal(cur.right, path, result)
            path.pop()  # 回溯
```