二叉树的中序遍历

返回二叉树的中序遍历

思路一：递归

思路二：栈迭代

思路三：莫利斯遍历（Morris）

特点：常数级空间+线性时间

思想：修改二叉树，将二叉树展开为链表

方法：

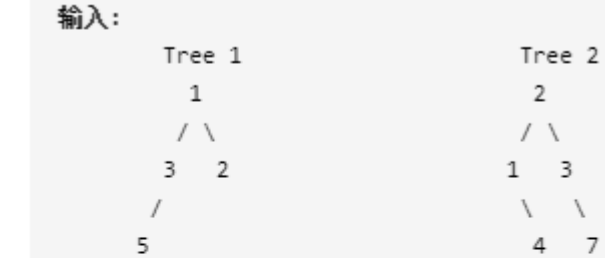
1. 若根节点的左子树存在：将根节点（及其右子树）移动到左子树的最右结点上；让左子树作为新的根节点（断开新旧根节点之间的联系 temp.left = null）
2. 若根节点的左子树不存在：访问根节点，转向右子树遍历

合并二叉树

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。

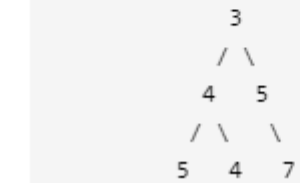
你需要将他们合并为一个新的二叉树。合并的规则是如果两个节点重叠，那么将他们的值相加作为节点合并后的新值，否则不为 NULL 的节点将直接作为新二叉树的节点。

输入：



输出：

合并后的树：



思路一：递归

复杂度分析

- 时间复杂度：O(N)，其中 N 是两棵树中节点个数的较小值。
- 空间复杂度：O(N)，在最坏情况下，会递归 N 层，需要 O(N) 的栈空间。

思路二：迭代 + 栈

> 我们首先把两棵树的根节点入栈，栈中的每个元素都会存放两个根节点，并且栈顶的元素表示当前需要处理的节点。在迭代的每一步中，我们取出栈顶的元素并把它移出栈，并将它们的值相加。随后我们分别考虑这两个节点的左孩子和右孩子，如果两个节点都有左孩子，那么就将左孩子入栈；如果只有一个节点有左孩子，那么将其作为第一个节点的左孩子；如果都没有左孩子，那么不用做任何事情。对于右孩子同理。
>
> 最后我们返回第一棵树的根节点作为答案。

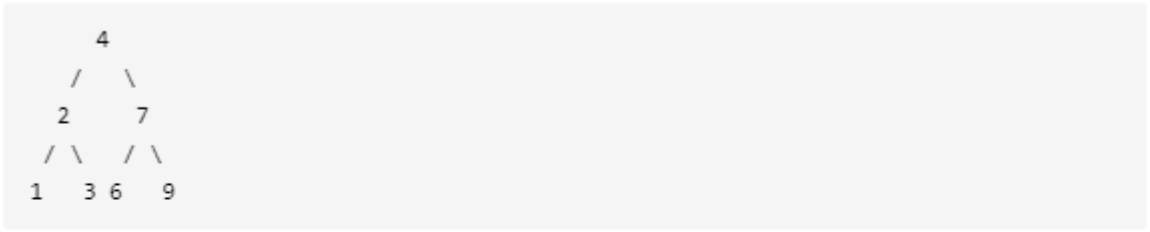
```
public class Solution {
 public TreeNode mergeTrees(TreeNode t1, TreeNode t2) {
     if (t1 == null)
         return t2;
     Stack < TreeNode[] > stack = new Stack < > ();
     stack.push(new TreeNode[] {t1, t2});
     while (!stack.isEmpty()) {
         TreeNode[] t = stack.pop();
         if (t[0] == null || t[1] == null) {
             continue;
         }
         t[0].val += t[1].val;
         if (t[0].left == null) {
             t[0].left = t[1].left;
         } else {
             stack.push(new TreeNode[] {t[0].left, t[1].left});
         }
         if (t[0].right == null) {
             t[0].right = t[1].right;
         } else {
             stack.push(new TreeNode[] {t[0].right, t[1].right});
         }
     }
     return t1;
 }
}
```

复杂度分析

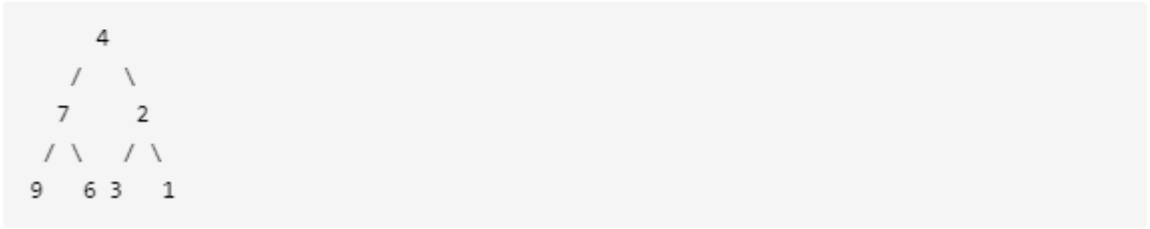
- 时间复杂度：O(N)，其中 N 是两棵树中节点个数的较小值。
- 空间复杂度：O(N)，在最坏情况下，栈中会存放 N 个节点。

翻转二叉树

输入：



输出：



思路一：递归

复杂度分析

- 时间复杂度：O(N)，树中的每个节点都只被访问一次
- 空间复杂度：O(h)，递归函数需要栈空间，而栈空间取决于递归的深度，因此空间复杂度等价于二叉树的高度 h。

思路二：迭代 + 队列

> 这种做法和深度优先搜索（Breadth-fist Search, BFS）很接近？？
>
> 这个方法的思路就是，我们需要交换树中所有节点的左孩子和右孩子。因此可以创一个队列来存储所有左孩子和右孩子还没有被交换过的节点。开始的时候，只有根节点在这个队列里面。只要这个队列不空，就一直从队列中出队节点，然后互换这个节点的左右孩子节点，接着再把孩子节点入队到队列，对于其中的空节点不需要加入队列。最终队列一定会空，这时候所有节点的孩子节点都被互换过了，直接返回最初的根节点就可以了。

```
public TreeNode invertTree(TreeNode root) {
 if (root == null) return null;
 Queue<TreeNode> queue = new LinkedList<TreeNode>();
 queue.add(root);
 while (!queue.isEmpty()) {
     TreeNode current = queue.poll();
     TreeNode temp = current.left;
     current.left = current.right;
     current.right = temp;
     if (current.left != null) queue.add(current.left);
     if (current.right != null) queue.add(current.right);
 }
 return root;
}
```

> ### 复杂度分析
>
> - 时间复杂度：O(N)，树中的每个节点都只被访问/入队一次
> - 空间复杂度：O(N)，队列，在最坏的情况下，也就是队列里包含了树中所有的节点。对于一颗完整二叉树来说，叶子节点那一层拥有 n/2 = O(n)

二叉树的最大深度

思路一：递归

复杂度分析

- 时间复杂度：O(N)，树中的每个节点都只被访问一次
- 空间复杂度：O(h)，递归函数需要栈空间，而栈空间取决于递归的深度，因此空间复杂度等价于二叉树的高度 h。

思路二：DFS

思路三：BFS

把二叉搜索树转换为累加树

给定一个二叉搜索树（Binary Search Tree），把它转换成为累加树（Greater Tree)，使得每个节点的值是原来的节点值加上所有大于它的节点值之和。

```
输入：原始二叉搜索树：
              5
            /   \
           2     13

输出：转换为累加树：
             18
            /   \
          20     13
```

分析：中间结点的值依赖于右子树，而左子树的值依赖于中间结点**（右->中->左）**

思路一：反序中序遍历+递归

1. 累加 => 用一个公共变量存储累加值，且问题的解决与累加是同步的
2. 每一层递归需要解决的问题就转化为：处理右子树的累加问题、中间结点累加、处理左子树的累加问题

复杂度分析

- 时间复杂度：O(N)，树中的每个节点都被累加一次
- 空间复杂度：O(N)，递归函数需要栈空间，而栈空间取决于累加的次数

```
var convertBST = function (root) {
  let sum = 0

  let recursion = function (p) {
    if (!p) return null

    recursion(p.right)  // 处理右子树的累加问题
    sum += p.val
    p.val = sum
    recursion(p.left)

    return p
```

```
  }

  return recursion(root)
};
```

思路二：反序中序遍历+栈迭代（递归思想）+ DFS

> 因为本质上是DFS的思想，所以用栈迭代处理

1. 栈迭代模拟系统递归
2. 反序中序遍历，压入右节点，保证压入的顺序是**降序**的（结点值小的需要依赖于节点值大的，累加）

复杂度分析

- 时间复杂度：O(N)，树中的每个节点都被累加一次
- 空间复杂度：O(N)，每个节点只会被压入栈中恰好一次的结论表明**最坏情况下**栈最多只会包含 n个节点

```
var convertBST = function (root) {
  if (!root) return null

  let sum = 0, cur = root
  const stack = []
  while (cur || stack.length) {
    while (cur) {
      stack.push(cur)
      cur = cur.right
    }
    cur = stack.pop()
    sum += cur.val
    cur.val = sum
    cur = cur.left
  }

  return root
};
```

思路三：反序中序遍历+Morris遍历（线性时间和常数空间）

Morris遍历（线性时间和常数空间 => 中序遍历）

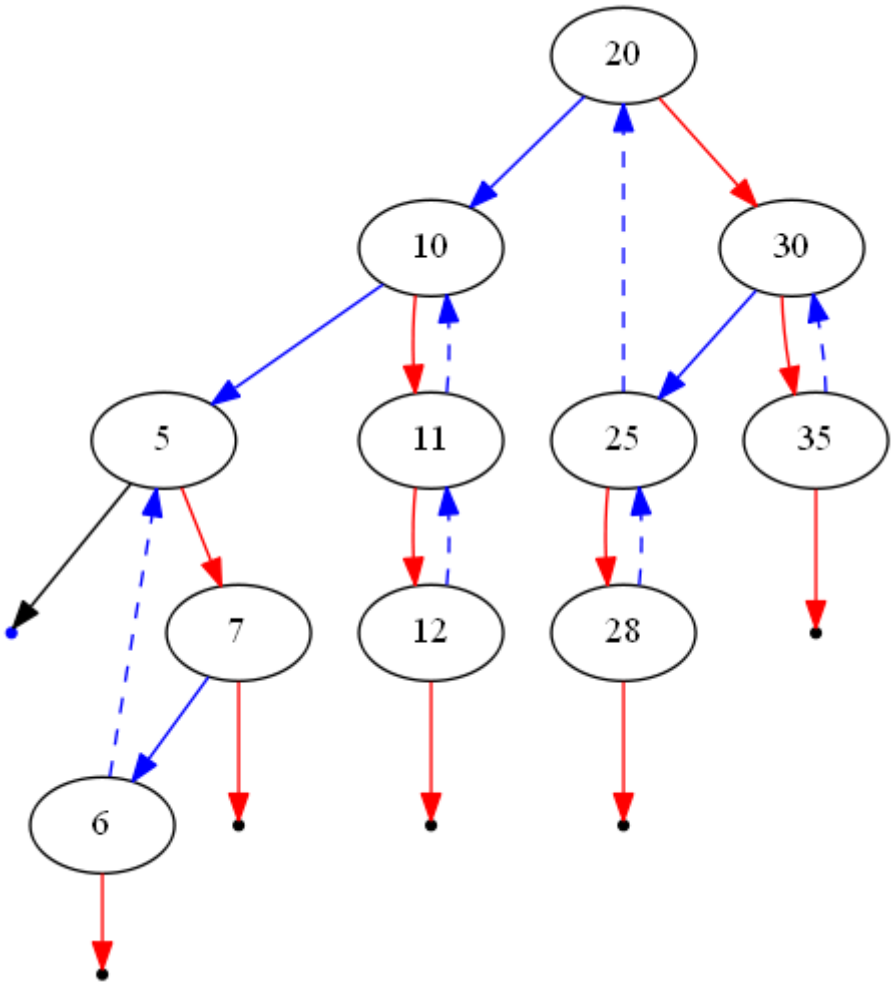
想法

有一种很机智的办法只用线性时间和常数空间来实现中序遍历，这种方法由 J. H. Morris 在他 1979 年的论文 "Traversing Binary Trees Simply and Cheaply" 中首次提出。总的来说，递归和迭代栈的方法在遍历左子树后返回当前节点，都牺牲了线性的空间。 Morris 遍历使用指针 null 指针在左子树以外创建了一个临时指针，使得遍历只需要使用常数额外空间。为了应用于此题，我们只需要交换所有的“左”“右”引用，就可以反转整个遍历过程。

算法

首先，我们初始化 node ，它指向根。然后，在 node 指向 null 之前（具体指树中最小值节点的左孩子 null），我们重复如下过程：先考虑当前节点是否有右子树，如果没有右子树那么所有大于当前节点的值都已遍历，我们就可以更新当前节点并进入左子树。如果还有右子树，那么至少有一个未访问过的更大值，我们必须先访问右子树。我们通过辅助函数 getSuccessor ，得到一个引用来记录当前节点中序遍历的下一个节点（最小大于当前值的节点），这个节点是恰好先于我们访问当前节点之前的节点，所

以根据定义这一下一节点有一个为 null 的左孩子指针（否则它不是当前节点的下一个节点）。所以当我们首次找到一个节点的下一节点时，我们将该下一节点的左指针指向当前节点。这样，当我们遍历完当前节点的右子树时，右子树中最左边节点的左指针会是我们的临时指针，它指向当前节点，我们就可以快速回到当前节点中来。如果我们发现从下一节点的左指针回到了当前节点，说明右子树已经遍历完了，我们可以删除这个临时指针并继续递归左子树。



上图说明了 Morris 遍历过程中一棵修改了的树。左指针被标为蓝色，右指针标为红色。虚线边表明遍历过程中的临时指针（在遍历结束之后会被删除）。注意到蓝色的边也可以标为虚线边，就像我们使用的“下一节点”的左空指针一样。除此以外，我们注意到每一个有右子树的节点都有一个“下一节点”的边指向自己。

```
public TreeNode convertBST(TreeNode root) {
    int sum = 0;
    TreeNode node = root;

    while (node != null) {
        /*
         * If there is no right subtree, then we can visit this node and
         * continue traversing left.
         */
        if (node.right == null) {
            sum += node.val;
            node.val = sum;
            node = node.left;
        }
        /*
         * If there is a right subtree, then there is at least one node that
         * has a greater value than the current one. therefore, we must
         * traverse that subtree first.
```

```
             */
            else {
                TreeNode succ = getSuccessor(node);
                /*
                 * If the left subtree is null, then we have never been here before.
                 */
                if (succ.left == null) {
                    succ.left = node;
                    node = node.right;
                }
                /*
                 * If there is a left subtree, it is a link that we created on a
                 * previous pass, so we should unlink it and visit this node.
                 */
                else {
                    succ.left = null;
                    sum += node.val;
                    node.val = sum;
                    node = node.left;
                }
            }
        }

        return root;
    }
```

渣渣鱼 2020-06-09

方法三部分思路，1，修改节点的空左指针使其指向下一个要处理的节点。2，处理节点的时候可以用空的左指针确定下一个处理的节点。2，在完成处理后把空的左指针还原成null。

复杂度分析

- 时间复杂度：O(n)，尽管 Morris遍历比起其他方法来说额外多做了一些工作，但只是一个常数级别的。具体来说，如果我们可以证明一条边被遍历了不超过常数 kk 次，那么这个算法就是一个渐进线性的复杂度。首先注意到 getSuccessor 每个节点最多会调用 2 次，在第一次调用时，会有一个临时指针指向当前节点，第二次调用时，临时指针被删除。然后这个算法会进入左子树且不再会回到该节点。所以每条边最多被遍历 3 次：一次是移动 node 指针，另外两次遍历是调用 getSuccessor 时。
- 空间复杂度：O(1)，由于我们只操作已经存在的指针， Morris 只需要常数的额外空间。

对称二叉树

给定一个二叉树，检查它是否是镜像对称的

分析：自己画一个大的树找规律，可以发现

- 一棵树对称 <=> 左右子树镜像对称
- AB树镜像对称 <=> A左子树、B右子树镜像对称；A右子树、B左子树镜像对称

思路一：递归

思路二：队列迭代

> 本质上是一个 BFS 问题，所以用的是队列迭代

验证二叉搜索树

给定一个二叉树，判断其是否是一个有效的二叉搜索树。

思路一：递归

分析：二叉搜索树的条件

- 左子树只包含**小于**当前节点的数；右子树只包含**大于**当前节点的数 => 树上的结点有**结界限制**
- 所有左子树和右子树自身必须也是二叉搜索树

递归函数参数中传入边界 let isVaildBSTWithRange = function (root, low, high)

思路一：验证中序遍历序列

用一个常数遍历记录上一轮遍历的值，则二叉树满足 pre< cur

二叉树的直径

给定一棵二叉树，你需要计算它的直径长度。一棵二叉树的直径长度是任意两个结点路径长度中的最大值。这条路径可能穿过也可能不穿过根结点。

思路一：递归

1. 直径的情况（取最大值）
    - 穿过根节点：左树深度+右树深度
    - 不穿过根节点：直径出现在左树/右树
2. 优化：在结点缓存树的深度

```
var maxDepth = function (root) {
  if (!root) return 0
  if (root.height) return root.height // 返回缓存
  root.height = Math.max(maxDepth(root.left), maxDepth(root.right)) + 1
  return root.height
}
var diameterOfBinaryTree = function (root) {
  if (!root || (!root.left && !root.right)) return 0
  let lenMid = maxDepth(root.left) + maxDepth(root.right)
  return Math.max(lenMid, diameterOfBinaryTree(root.left),
diameterOfBinaryTree(root.right))
};
```

思路二：递归+公共变量存储

思路一进阶：结果路径必然会经过某一个结点 => DFS遍历，用一个公共变量存储最大值

- 计算路径时需要涉及到计算树的最大深度（DFS），所以可以将比较的思想嵌入其中

```
var diameterOfBinaryTree = function (root) {
  let maxPath = 0

  let maxDepth = function (root) {
    if (!root) return 0
    let L = maxDepth(root.left)
```

```
    let R = maxDepth(root.right)
    maxPath = Math.max(L + R, maxPath)
    return Math.max(L, R) + 1
  }

  maxDepth(root)
  return maxPath
}
```

二叉树展开为链表

给定一个二叉树，原地将它展开为一个单链表。

思路一：类似莫利斯遍历展开

1. 将根节点的右子树移动到左子树的最右上
2. 将根节点的左子树移动到根节点的右子树上
3. 向右子树执行同一步骤

复杂度分析

- 时间复杂度：O(n)，其中 nn 是二叉树的节点数。展开为单链表的过程中，需要对每个节点访问一次，在寻找前驱节点的过程中，每个节点最多被额外访问一次。
- 空间复杂度：O(1)。

思路二：递归

单层逻辑：左子树展开；右子树展开；左子树右移；根节点寻最右；原右子树右接

二叉树的层序遍历

思路一：传统层序遍历+标记'#'

思路二：传统层序遍历升级，一次取n个出来处理

不同的二叉搜索树

思路一：动态规划+找规律

分析可得表达式（递归表达式）：$G(n) = \sum_{i=1}^{n} G(i-1) \cdot G(n-i)$

思路二：卡特兰数

上述公式G(n)函数的值在数学上被称为卡塔兰数，$C_0 = 1, \quad C_{n+1} = \frac{2(2n+1)}{n+2} C_n$

打家劫舍 III

在上次打劫完一条街道之后和一圈房屋后，小偷又发现了一个新的可行窃的地区。这个地区只有一个入口，我们称之为“根”。 除了“根”之外，每栋房子有且只有一个“父”房子与之相连。一番侦察之后，聪明的小偷意识到“这个地方的所有房屋的排列类似于一棵二叉树”。 如果两个直接相连的房子在同一天晚上被打劫，房屋将自动报警。

计算在不触动警报的情况下，小偷一晚能够盗取的最高金额。

示例 1:

```
输入: [3,2,3,null,3,null,1]

     3
    / \
   2   3
    \   \
     3   1

输出: 7
解释: 小偷一晚能够盗取的最高金额 = 3 + 3 + 1 = 7.
```

示例 2:

```
输入: [3,4,5,1,3,null,1]

     3
    / \
   4   5
  / \   \
 1   3   1

输出: 9
解释: 小偷一晚能够盗取的最高金额 = 4 + 5 = 9.
```

思路一：暴力递归 - 最优子结构

分析：用爷父孙来描述，最大的钱数一定出现在以下两种情况内，比较取最大值

- 爷爷结点被偷（可以去偷四个孙子结点）
- 爷爷结点不偷（可以去偷两个父亲结点）

存在**重复子问题**：爷爷在计算自己能偷多少钱的时候，同时计算了 4 个孙子能偷多少钱，也计算了 2 个儿子能偷多少钱。**当儿子作为爷爷时**，就会产生**重复计算一遍孙子节点**

思路二：递归 + 记忆化（解决重复子问题）

分析：针对子问题进一步优化，可以将计算的结果缓存起来（大部分问题可以用数组缓存）。因二叉树的特殊性，**选用哈希表来缓存结果**

处理方法：在递归函数开头先检测是否有缓存值，有则返回；无则正常计算，将结果缓存

思路三：递归 + 直接消除重复子问题

思考：为什么计算爷爷的时候还要去计算孙子？

解决方法：让递归函数返回两种结果（是否取根）：递归内部可指定取，递归外部选最大 => 无需解决重复子问题

二叉树的最近公共祖先

祖先的定义： 若**节点 p** 在节点 root 的**左（右）子树中**，或 **p = root** ，则称 root是 p 的祖先。

最近公共祖先的定义： 设节点 root 为节点 p, q 的某公共祖先，若其左子节点 root.left 和右子节点 root.right 都不是 p,q 的公共祖先，则称 root 是 “最近的公共祖先” 。