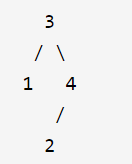
**树的遍历**

树的遍历有前序、中序、后序三种，这里分开介绍三种遍历的方式、递归解法和迭代解法。迭代解法采用栈模拟时会使用O(k)的空间复杂度，而Morris遍历算法可以优化到O(1)的空间复杂度。

假设树为

**前序**

遍历方式为根节点、左子节点、右子节点，上图前序结果为[3,1,4,2]

**递归方式**：

def dfs(root):  
 if not root:

return

ans.append(root)

dfs(root.left)

dfs(root.right)

可以看出，递归方式完全按照遍历方式来做，首先得到根节点，然后进入左子树，最后进入右子树。

而**迭代方式**则是对递归的模拟，从根节点开始，如果当前节点存在，则记录该节点的值，并将该节点压入栈中，进入节点的左子节点，直到不存在。则证明进入到最左的叶子节点，此时弹出栈顶回到该节点的父节点，模拟的就是从叶子节点结束进入后续步骤的过程，退栈后对右子树进行访问。

stack = []

cur = root

while stack or cur:

# 记录当前值后进入左子节点，保存到栈中留作回溯

while cur:

stack.append(cur)

记录当前节点值cur.val

cur = cur.left

cur = stack.pop()

cur = cur.right

此外还有另一种方法，利用出栈顺序和入栈顺序相反的特性，每次添加节点时因为要先访问左子节点再是右子节点，在入栈时先放右子节点再放左子节点，这样在栈弹出时就会先访问左子节点。

stack = [root]

while stack:

cur = stack.pop()

记录当前节点值cur.val

if cur.right:

stack.append(cur.right)

if cur.left:

stack.append(cur.left)

这个方法就是用栈对递归的模拟，记录节点值后，先访问左子节点再访问右子节点。

**Morris前序遍历**：Morris遍历只提供了中序遍历的方法，前序是在他基础上的衍生，所以请先参考中序遍历的方法。前序遍历只是在中序的基础上，修改了访问节点的顺序。

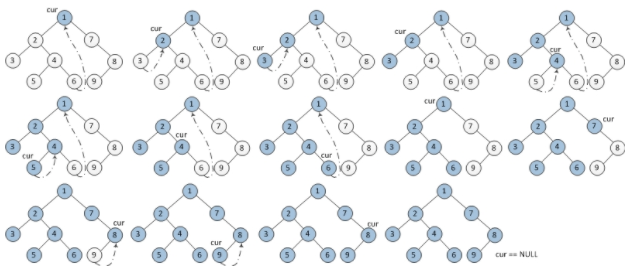
1. 如果当前节点的左孩子为空，则输出当前节点并将其右孩子作为当前节点。

2. 如果当前节点的左孩子不为空，在当前节点的左子树中找到当前节点在中序遍历下的前驱节点。

   a) 如果前驱节点的右孩子为空，将它的右孩子设置为当前节点。**输出当前节点（在这里输出，这是与中序遍历唯一一点不同）。**当前节点更新为当前节点的左孩子。

   b) 如果前驱节点的右孩子为当前节点，将它的右孩子重新设为空。当前节点更新为当前节点的右孩子。

3. 重复以上1、2直到当前节点为空。



cur = root, prev = None

while cur:

# 左子树不存在则不用访问  
 if cur.left == None:

记录当前节点值cur.val

cur = cur.right

else:

# 找前驱节点

prev = cur.left

while prev.right != None and prev.right != cur:  
 prev = prev.right

# 前驱节点为叶子节点，记录后继节点信息，继续遍历

if prev.right == None:

# 注意此处和中序的不同，找到前驱节点之后如果为叶子节点，会修改前驱节点的右子节点，而此时的cur节点为根节点，需要先于left保存

记录当前节点值cur.val

prev.right = cur

cur = cur.left

else:

# 回溯到已经访问过的非叶节点，记录节点并恢复树结构，并向左继续遍历

prev.right = None

cur = cur.right

**中序**

遍历方式为左子节点，根节点、右子节点，上图中的中序结果为[1,3,2,4]。

**递归方式**：

def dfs(root):

if not root:

return

dfs(root.left)

记录当前节点值root.val

dfs(root.right)

而迭代方式同理，首先进入最左叶子节点，并记录所有路径中出现的节点，和前序的区别在于，这里只记录所有节点到栈中，只有在被弹出时才是访问到了非叶节点，记录该值，并进入右子树，因为左子树在此前已经被弹出并记录过，总体的遍历顺序仍保持左子节点、根节点和右子节点。

stack = []

cur = root

while stack or cur:

# 递归首先添加左节点

while cur:

stack.append(cur)

cur = cur.left

# 左节点不存在，弹出当前节点并记录，然后进入右子节点

cur = stack.pop()

记录当前节点值cur.val

cur = cur.right

上述解法的空间复杂度为O(k)，k表示树的最大深度，因为要使用一个栈模拟。

而Morris中序遍历法可以使用O(1)的复杂度完成。

**Morris中序遍历**：解决遍历问题最大的难点是如何回溯，即如何从当前节点返回到父节点，此前可以通过栈保存访问到的节点，而在Morris中不能使用栈作为辅助空间。Morris解决中序遍历利用每个节点的右子节点的空指针指向该节点的后继节点，即对于一个节点，找到他的前驱节点（以左子节点为根的子数中的最右叶子节点），如果这个节点的右子节点为空，说明当前还没有访问过，将右子节点设置为当前节点，当前节点设为左子节点继续遍历，如果右子节点不为空，则证明是此前已经访问过的节点，现在属于回溯阶段，则将前驱节点的右子节点设为null恢复到树之前的状态，并将当前节点设置为当前节点的右子节点，访问右子树。

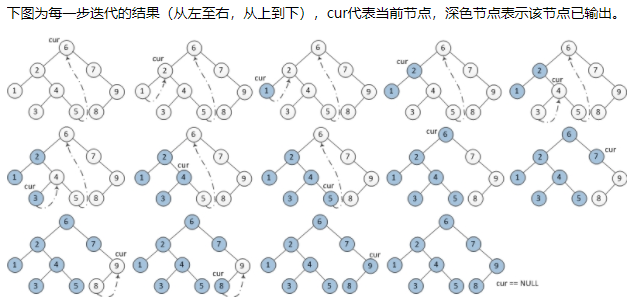
1. 如果当前节点的左孩子为空，则输出当前节点并将其右孩子作为当前节点。

2. 如果当前节点的左孩子不为空，在当前节点的左子树中找到当前节点在中序遍历下的前驱节点。

   a) 如果前驱节点的右孩子为空，将它的右孩子设置为当前节点。当前节点更新为当前节点的左孩子。

   b) 如果前驱节点的右孩子为当前节点，将它的右孩子重新设为空（恢复树的形状）。输出当前节点。当前节点更新为当前节点的右孩子。

3. 重复以上1、2直到当前节点为空



cur = root, prev = None

while cur:

# 左子树不存在则不用访问  
 if cur.left == None:

记录当前节点值cur.val

cur = cur.right

else:

# 找前驱节点

prev = cur.left

while prev.right != None and prev.right != cur:  
 prev = prev.right

# 前驱节点为叶子节点，记录后继节点信息，继续遍历

if prev.right == None:

prev.right = cur

cur = cur.left

else:

# 回溯到已经访问过的非叶节点，记录节点并恢复树结构，并向左继续遍历

记录当前节点值cur.val

prev.right = None

cur = cur.right

**后序**

遍历方式是左子节点、右子节点、根节点。上图中的后序结果是[1,2,4,3]。

**递归**解法：

def dfs(root):

if not root:

return

dfs(root.left)

dfs(root.right)

记录当前节点值root.val

而**迭代**解法有两种，一种是简单易理解的版本，就是类似于前序遍历利用栈特性的方式，既然是左子节点、右子节点、根的形式，那也可以使用根、右子节点、左子节点的方法，这样得到了非常类似于前序遍历的方式，然后再将得到的序列倒序返回即为后序遍历。

stack = [root]

while stack:

cur = stack.pop()

记录当前节点值

if cur.left:

stack.append(cur.left)

if cur.right:

stack.append(cur.right)

返回节点值记录的倒序

另一种是有些复杂的迭代方式，类似于前序和中序提到的那种方法。但在栈的模拟过程中需要注意的一个点是，由于是先访问完左右子节点最后访问根节点，所以需要获取保存当前节点值之前先判断该节点的右子节点是否为空或者是否已访问过，这样才可以访问这个节点。而判断是否访问过则可以使用一个指针指向该节点的前缀节点（即按照后序遍历的前一个节点），如果前缀节点等于该节点的右子节点则表示该节点该被访问，并且该节点左右均已完成，在保存节点值之后该继续从栈中弹出该节点的父节点。这个前缀节点是用于判断一个根节点是从左子树回到的还是从右子树回到的。

stack = []

cur = root

while stack or cur:

while cur:

stack.append(cur)

cur = cur.left

# 注意这里不能弹出，因为如果right还存在的话还需要继续回溯本节点

cur = stack[-1]

if not cur.right or cur.right == pre:

记录当前节点值cur

# 访问完之后弹出

stack.pop()

pre = cur

cur = None

else:

cur = cur.right

**Morris后序遍历**：后序遍历大体思路相符，也是记录前驱节点，但在此基础上需要加入一个倒序的概念，即回溯到当前节点时，应该倒序输出从当前节点的左孩子到该前驱节点这条路径上的所有节点，即符合先右子节点再根节点的遍历顺序。

当前节点设置为临时节点dump。

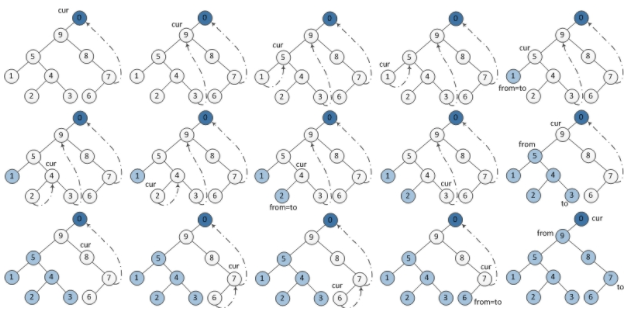
1. 如果当前节点的左孩子为空，则将其右孩子作为当前节点。

2. 如果当前节点的左孩子不为空，在当前节点的左子树中找到当前节点在中序遍历下的前驱节点。

   a) 如果前驱节点的右孩子为空，将它的右孩子设置为当前节点。当前节点更新为当前节点的左孩子。

   b) 如果前驱节点的右孩子为当前节点，将它的右孩子重新设为空。**倒序输出从当前节点的左孩子到该前驱节点这条路径上的所有节点。**当前节点更新为当前节点的右孩子。

3. 重复以上1、2直到当前节点为空



def reverse(from, to):

# 这个函数的作用就是将from到to的路径反转，即from一路right得到to，变成to一路right会得到from。

if from == to:

return

x = from, y = from,right

while True:  
 z = y.right

y.right = x

x = y

y = z

if x == to:

break

def printReverse(from, to):

reverse(from, to)

p = to;

while True:  
 记录当前节点值p.val

if p == from:

break

p = p.right

reverse(to, from)

def morrisPostOrder(root):

dump = TreeNode(0)

dump.left = root

cur = dump, prev = None

while cur:

if cur.left == None:

cur = cur.right

else:

prev = cur.left

while prev.right != None and prev.right != cur:

prev = prev.right

if prev.right == None:

prev.right = cur

cur = cur.left

else:

printReverse(cur.left, prev)

prev.right = None

cur = cur.right

树的层次遍历：

层次遍历指的是按照层数，逐层的从左到右访问所有节点。每一层的结果要分开记录。

使用队列来模拟，并且记录下一层的元素个数。具体做法是：初始队列为[root]，第一层结点个数为1，当前层剩余结点个数不为0时，从队列取队首元素，并将该元素左右子节点放入队列，下一层结点个数对应增加。直到为0时，加到解空间中，并替换为下一层的结点个数开始遍历。

代码：

BFS版本：

if not root:

return []

q = [root]

level = []

while q:

cur\_level = []

num = len(q)

while num:

node = q.pop(0)

cur\_level.append(node.val)

if node.left:

q.append(node.left)

if node.right:

q.append(node.right)

num -= 1

level.append(cur\_level)

return level

DFS版本：

def dfs(node, level):

            if len(ans) == level:

                ans.append([])

            ans[level].append(node.val)

            if node.left:

                dfs(node.left, level + 1)

            if node.right:

                dfs(node.right, level + 1)

        if not root:

            return []

        ans = []

        dfs(root, 0)

        return ans