Morris Traversal 方法遍历二叉树（非递归，不用栈，O(1)空间）

本文主要解决一个问题，如何实现二叉树的前中后序遍历，有两个要求：

1. O(1)空间复杂度，即只能使用常数空间；

2. 二叉树的形状不能被破坏（中间过程允许改变其形状）。

通常，实现二叉树的前序（preorder）、中序（inorder）、后序（postorder）遍历有两个常用的方法：一是递归(recursive)，二是使用栈实现的迭代版本(stack+iterative)。这两种方法都是 O(n)的空间复杂度（递归本身占用 stack 空间或者用户自定义的 stack），所以不满足要求。（用这两种方法实现的中序遍历实现可以参考这里。）

Morris Traversal 方法可以做到这两点，与前两种方法的不同在于该方法只需要 O(1)空间，而且同样可以在 O(n)时间内完成。

要使用 O(1)空间进行遍历，最大的难点在于，遍历到子节点的时候怎样重新返回到父节点（假设节点中没有指向父节点的 p 指针），由于不能用栈作为辅助空间。为了解决这个问题，Morris 方法用到了线索二叉树（threaded binary tree）的概念。在 Morris 方法中不需要为每个节点额外分配指针指向其前驱（predecessor）和后继节点（successor），只需要利用叶子节点中的左右空指针指向某种顺序遍历下的前驱节点或后继节点就可以了。

Morris 只提供了中序遍历的方法，在中序遍历的基础上稍加修改可以实现前序，而后续就要再费点心思了。所以先从中序开始介绍。

首先定义在这篇文章中使用的二叉树节点结构，即由 val，left 和 right 组成：

```
1 struct TreeNode {
2     int val;
3     TreeNode *left;
4     TreeNode *right;
5     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
6 };
```

Morris traversal:

为了实现 O(1)空间复杂度的遍历，Threaded Binary Tree 对普通二叉树进行了一些改造，将每一个节点在中序遍历时的前驱节点的右子树指向自己。

我们可以每次访问到一棵子树时，找到它的前驱节点，把前驱节点的右儿子变为当前的根节点 root，这样当遍历完前驱节点后，可以顺着这个右儿子回到根节点 root。

Morris 遍历本质上其实就是利用二叉树中 n+1 个指向 NULL 的指针。在遍历的过程中，通过利用叶子节点空的 right 指针，指向中序遍历的后继节点，从而避免了对 stack 的依赖。

一、中序遍历

步骤：

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。

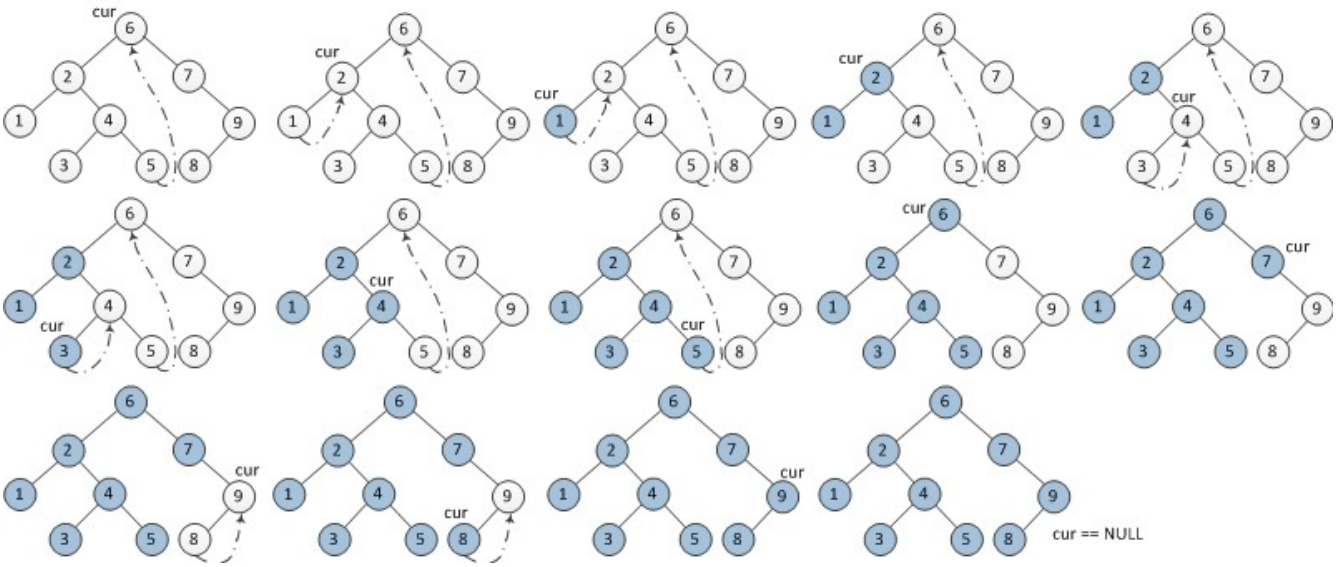
   a) 如果前驱节点的右孩子为空，将它的右孩子设置为当前节点。当前节点更新为当前节点的左孩子。

   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。

图示：

下图为每一步迭代的结果（从左至右，从上到下），cur 代表当前节点，深色节点表示该节点已输出。

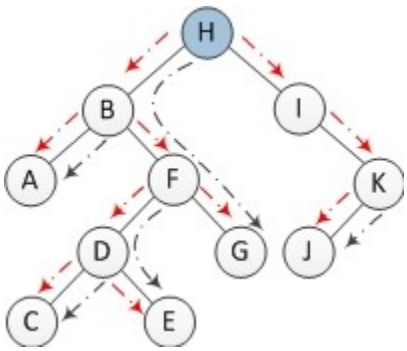


代码：

```
1 void inorderMorrisTraversal(TreeNode *root) {
2     TreeNode *cur = root, *prev = NULL;
3     while (cur != NULL)
4     {
5         if (cur->left == NULL)          // 1.
6         {
7             printf("%d ", cur->val);
8             cur = cur->right;
9         }
10        else
11        {
12            // find predecessor
13            prev = cur->left;
14            while (prev->right != NULL && prev->right != cur)
15                prev = prev->right;
16
17            if (prev->right == NULL)    // 2.a)
18            {
19                prev->right = cur;
20                cur = cur->left;
21            }
22            else                        // 2.b)
23            {
24                prev->right = NULL;
25                printf("%d ", cur->val);
26                cur = cur->right;
27            }
28        }
29    }
30 }
```

复制代码

复杂度分析：



空间复杂度：O(1)，因为只用了两个辅助指针。

时间复杂度：O(n)。证明时间复杂度为 O(n)，最大的疑惑在于寻找中序遍历下二叉树中所有节点的前驱节点的时间复杂度是多少，即以下两行代码：

```
1 while (prev->right != NULL && prev->right != cur)
2     prev = prev->right;
```

直觉上，认为它的复杂度是 O(nlgn)，因为找单个节点的前驱节点与树的高度有关。但事实上，寻找所有节点的前驱节点只需要 O(n)时间。n 个节点的二叉树中一共有 n-1 条边，整个过程中每条边最多只走 2 次，一次是为了定位到某个节点，另一次是为了寻找上面某个节点的前驱节点，如上图所示，其中红色是为了定位到某个节点，黑色线是为了找到前驱节点。所以复杂度为 O(n)。

二、前序遍历

前序遍历与中序遍历相似，代码上只有一行不同，不同就在于输出的顺序。

步骤：

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。

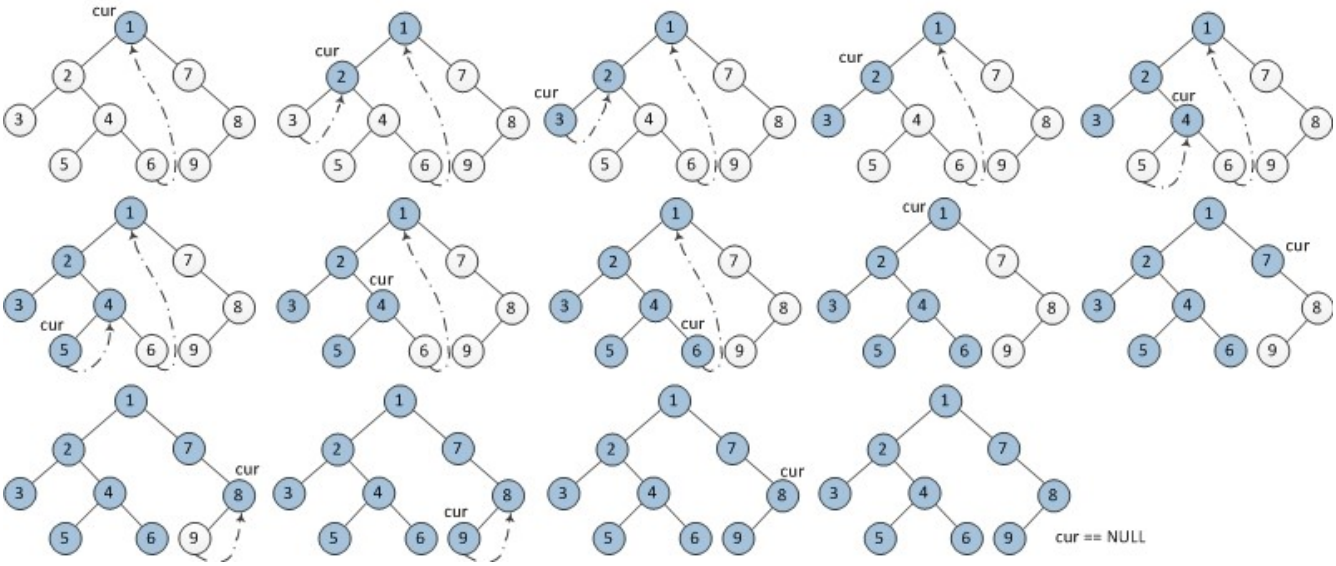
2. 如果当前节点的左孩子不为空，在当前节点的左子树中找到当前节点在中序遍历下的前驱节点。

   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。

图示：



代码：

```
 1 void preorderMorrisTraversal(TreeNode *root) {
 2     TreeNode *cur = root, *prev = NULL;
 3     while (cur != NULL)
 4     {
 5         if (cur->left == NULL)
 6         {
 7             printf("%d ", cur->val);
 8             cur = cur->right;
 9         }
10         else
11         {
12             prev = cur->left;
13             while (prev->right != NULL && prev->right != cur)
14                 prev = prev->right;
15 
16             if (prev->right == NULL)
17             {
18                 printf("%d ", cur->val);  // the only difference with inorder-traversal
19                 prev->right = cur;
20                 cur = cur->left;
21             }
22             else
23             {
24                 prev->right = NULL;
25                 cur = cur->right;
26             }
27         }
28     }
29 }
```

复制代码

复杂度分析：

时间复杂度与空间复杂度都与中序遍历时的情况相同。

三、后序遍历

后续遍历稍显复杂，需要建立一个临时节点 dump，令其左孩子是 root。并且还需要一个子过程，就是倒序输出某两个节点之间路径上的各个节点。

步骤：

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。

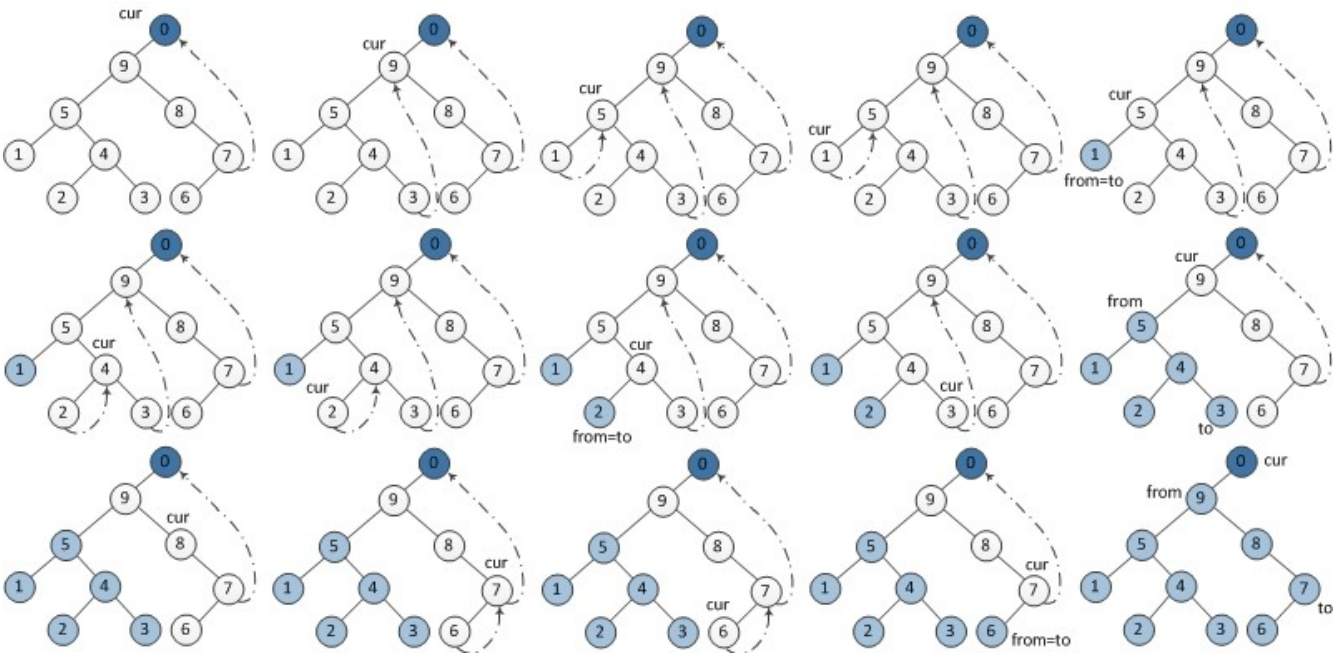
2. 如果当前节点的左孩子不为空，在当前节点的左子树中找到当前节点在中序遍历下的前驱节点。

   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。

图示：



代码：

```
 1 void reverse(TreeNode *from, TreeNode *to) // reverse the tree nodes 'from' -> 'to'.
 2 {
 3     if (from == to)
 4         return;
 5     TreeNode *x = from, *y = from->right, *z;
 6     while (true)
 7     {
 8         z = y->right;
 9         y->right = x;
10         x = y;
11         y = z;
12         if (x == to)
13             break;
14     }
15 }
16
17 void printReverse(TreeNode* from, TreeNode *to) // print the reversed tree nodes 'from' -> 'to'.
18 {
19     reverse(from, to);
20
21     TreeNode *p = to;
22     while (true)
23     {
24         printf("%d ", p->val);
25         if (p == from)
26             break;
27         p = p->right;
28     }
29
30     reverse(to, from);
31 }
32
33 void postorderMorrisTraversal(TreeNode *root) {
34     TreeNode dump(0);
35     dump.left = root;
36     TreeNode *cur = &dump, *prev = NULL;
37     while (cur)
38     {
39         if (cur->left == NULL)
40         {
41             cur = cur->right;
42         }
43         else
44         {
45             prev = cur->left;
46             while (prev->right != NULL && prev->right != cur)
47                 prev = prev->right;
```

```
48
49            if (prev->right == NULL)
50            {
51                prev->right = cur;
52                cur = cur->left;
53            }
54            else
55            {
56                printReverse(cur->left, prev);  // call print
57                prev->right = NULL;
58                cur = cur->right;
59            }
60        }
61    }
62 }
```

复杂度分析：

空间复杂度同样是 O(1)；时间复杂度也是 O(n)，倒序输出过程只不过是加大了常数系数。

注：

以上所有的代码以及测试代码可以在我的 Github 里获取。

参考：

http://www.geeksforgeeks.org/inorder-tree-traversal-without-recursion-and-without-stack/
http://www.geeksforgeeks.org/morris-traversal-for-preorder/
http://stackoverflow.com/questions/6478063/how-is-the-complexity-of-morris-traversal-on
http://blog.csdn.net/wdq347/article/details/8853371
Data Structures and Algorithms in C++ by Adam Drozdek

以前我只知道递归和栈+迭代实现二叉树遍历的方法，昨天才了解到有使用 O(1)空间复杂度的方法。以上都是我参考了网上的资料加上个人的理解来总结，如果有什么不对的地方非常欢迎大家的指正。