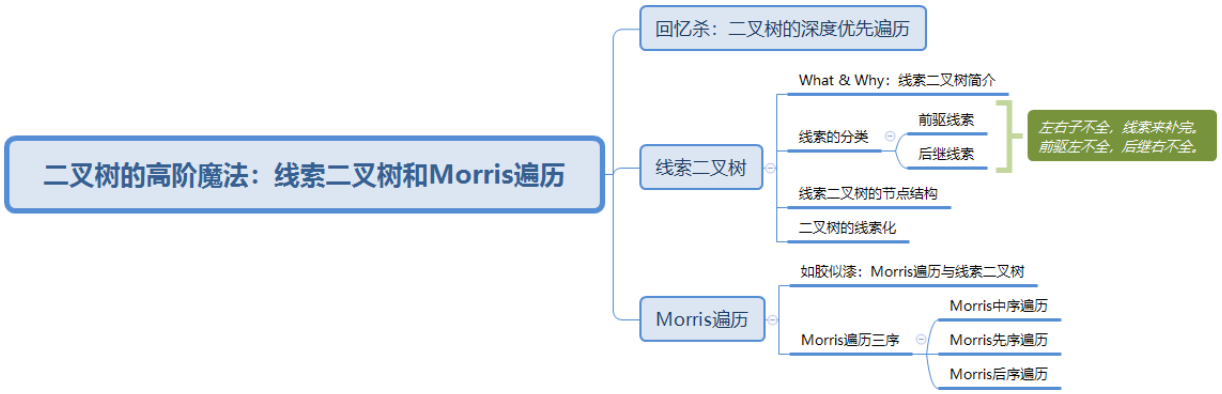


1.回忆杀：二叉树的深度优先遍历

在开始学习线索二叉树和Morris遍历之前，我们先来回顾一下二叉树深度优先遍历部分的内容，

顺便历数一下二叉树深度优先遍历的“罪状”！

早在二叉树基本操作的章节当中，我们就已经研究过二叉树深度优先遍历的内容。

二叉树深度优先遍历总共分为三种顺序：先序遍历、中序遍历和后序遍历。

而在对二叉树进行深度优先遍历的时候，我们往往是借助栈结构或者递归的方式来完成的。

但是就是这两种最传统、最常见的遍历方式，也为二叉树的深度优先遍历带来了不可避免的“原罪”：较高的辅助空间消耗。

情况1：栈结构实现的深度优先遍历

在使用栈结构辅助完成深度优先遍历的时候，因为二叉树结构中的每一个节点都仅仅遍历一次，所以其时间复杂度是O(n)的，

但是因为用到了额外的栈结构，并且这个栈结构的最大容量等于二叉树中节点的数量，所以这种遍历方式的空间复杂度同样是O(n)的。

情况2：递归实现的深度优先遍历

容易理解的是，使用递归方式实现的二叉树深度优先遍历，因为同样是每一个节点仅仅访问一次，所以时间复杂度依然是O(n)的。

此时，有的同学可能会说：在使用递归的时候，我们并没有使用诸如栈结构等额外的辅助空间，所以辅助空间复杂度是不是O(1)甚至O(0)呢？

呵呵！图样图森破……内存中的方法栈，他就不是栈结构了吗？

了解JVM结构的同学应该知道，每当我们调用一个方法的时候，都会向内存中的方法栈结构中添加一个方法栈帧，用来保存这个方法的各种临时变量与运算数等信息。

当方法运行结束之后，这个栈帧才会从方法栈中退出。而方法的递归操作，就是一个方法不断的调用自己本身，

而在每一次自我调用的同时，同样会导致方法栈帧的产生和入栈。也就是说使用递归方式实现的二叉树深度优先遍历操作，

在内存中并没有节省多少空间的开销，所以其空间复杂度同样是O(n)没变。

在回顾了二叉树深度优先遍历的两种实现手段的时间复杂度和空间复杂度之后，加入我说：

还有一种针对二叉树的高阶操作魔法，能够实现时间复杂度O(n)不变，空间复杂度降为O(1)的深度优先遍历操作，你信不信？（内心OS：我信你个鬼！）

对！信就对了！下面就让我们一起学习线索二叉树以及Morris遍历操作。

2.线索二叉树

古人云：工欲善其事，必先利其器！就像一个合格的巫师，必然要有一把（或者n把，n >= 2）优秀的法杖一样，

速度快空间消耗低的Morris遍历，也是依靠一种神奇的二叉树结构来实现的，那就是线索二叉树。

①What & Why：线索二叉树简介

首先我们来明确一个概念：在一个具有n个节点的二叉树当中，一定存在2n个指针域，即每一个节点的左右孩子指针域。

那么在这2n个指针域当中，只有n-1个指针域是指向节点左右孩子的，也就说一定存在n+1个指针域是空的，并且这一特点，与二叉树的形状无关！

于是乎，有人就动起了这n+1个空指针域的歪脑筋：反正这些指针域空着也是空着，如果我们利用这n+1个空的指针域，指向每一个节点的来源与去处，

那么是不是对二叉树这种非线性结构的遍历，就会变得像遍历链表这种线性存储结构一般简单呢？

答案是：还真是真么回事！

我们将二叉树节点中的那些没有指向子节点的指针域利用起来，或者指向当前节点的来源，或者指向当前节点的去向，

我们把指针域中存储的这种引用，称之为线索（Thread），而这种利用空闲指针域保存线索的二叉树，即为线索二叉树（Threaded Binary Tree）。

而后面我们要去介绍的神奇的Morris遍历，就是依赖线索二叉树中的条条线索来实现的。

②线索的分类

在前文中我们提到：线索二叉树的得名，就是依赖于这个二叉树结构中，使用空闲指针域保存当前节点来龙去脉的做法。

于是乎我们将这些线索按照指向方向与当前节点之间的关系，分为了两种：前驱线索和后继线索。

并且，我们在正式开始介绍这两种线索之前，必须先来确定线索二叉树的一个重要前提条件：

在线索二叉树中，所有线索的确定，均是以当前二叉树节点的中序遍历序列为基础定义的。

1.前驱线索

线索指向：

当前节点的前驱节点

确定线索前提：

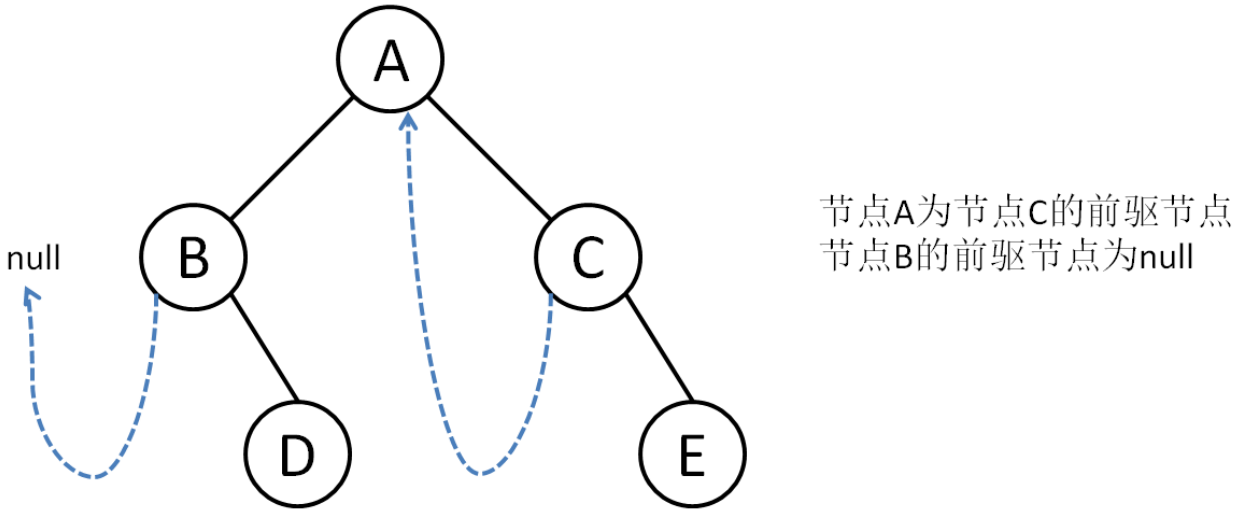
当前节点左孩子指针域为空

确定线索方式：

使用当前节点在中序遍历序列中的前序节点，作为其前驱线索指向，由左孩子指针域指向前驱节点；

若当前节点为整个二叉树结构中序遍历的起始节点，则其左孩子指针域指向空。

图例：



注意：实际上还有一种更加简洁的用来确定前驱线索指向的方式：

在线索二叉树中，如果任一节点左孩子不为空，在以这一节点左子树中最右侧的节点为其前驱节点。

值得注意的是，通过这一方法确定的，并不是当前节点的前驱线索指向，而是当前节点左子树中最右侧节点的前驱线索指向。

2.后继线索

线索指向：

当前节点的后继节点

确定线索前提：

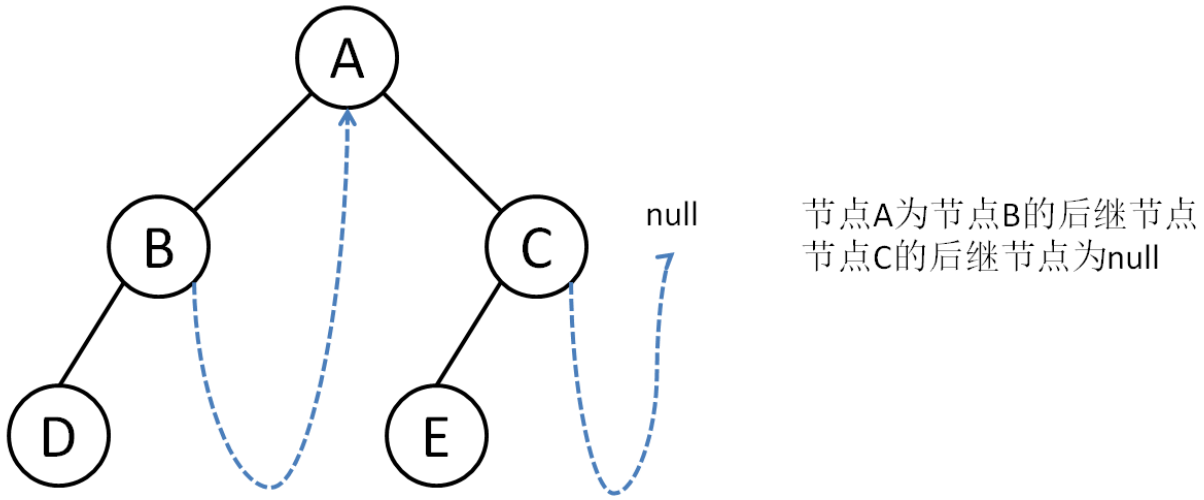
当前节点右孩子指针域为空

确定线索方式：

使用当前节点在中序遍历序列中的后序节点，作为其后继指针指向，由右孩子指针域指向后继节点；

若当前节点为整个二叉树结构中序遍历的终点节点，则其右孩子指针域指向空。

图例：



总结

在上面我们介绍了线索二叉树中，前驱线索和后继线索的定义方式以及相关图例。

虽然在图中没有指出，但是因为二叉树的叶子节点的左右孩子指针域都是空，所以每一个叶子节点都可以同时具有前驱线索和后继线索。

值得注意的是，我们在介绍前驱线索和后继线索的时候，特别强调了：

只有缺失左孩子的节点才能够具有前驱线索；只有缺失右孩子的节点才能够具有后继线索。

反过来说：如果一个节点的左右孩子节点均存在，那么这个节点将无法记录自己的前驱线索和后继线索。

总结起来就是：左右子不全，线索来补完。前驱左不全，后继右不全。

③线索二叉树的节点结构

在学习了前面的内容之后，有的同学可能会产生这样一个问题：

同样是指针（引用），那么我应该如何区分一个节点左后孩子指针域指向的是真正的左后孩子节点，还是前驱后继节点呢？

这个问题就需要我们在传统的二叉树节点基础上，做一些手脚才能够解决了！

传统的二叉树节点有3个域，分别是：

存储数据的数据域data；存储左孩子节点内存地址的左孩子域leftChild；存储右孩子节点内存地址的右孩子域rightChild。

但是在线索二叉树当中，每一个节点分别多出了两个标记域，即leftTag域与rightTag域，

分别用来标记左右孩子域中保存的到底是左右孩子节点的内存地址，还是前驱后继节点的内存地址：

在真正的编程过程当中，leftTag域和rightTag域的定义，可以使用尽可能少占用内存空间的数据类型，

例如：使用boolean类型的true表示存储的是孩子节点的内存地址，而用false表示存储的是线索指向节点的内存地址；

亦或是和作者类似的，使用一个byte类型的变量，并约定两种不同的取值，分别代表线索和对孩子节点的引用：

```
/**
 * 线索二叉树节点数据类型
 */
public class Node {

    //用来表示左右孩子域存储索引类型的常量
    public static final byte LINK = 0x0;  //表示孩子域中保存的是真正的左右孩子节
    public static final byte THREAD = 0x1;  //表示孩子域中保存的是前驱/后继线索

    public String data;  //数据域

```

```
    public Node leftChild;  //左孩子域，同为前驱线索域
    public byte leftTag;  //左孩子标记域

    public Node rightChild;  //右孩子域，同为后继线索域
    public byte rightTag;  //右孩子标记域

    /*
    为了后面偷懒，就没有按照规范定义属性的get/set方法，
    而是直接使用了public权限的对象属性声明
    小盆友们可不要这么做呦~~~
    */

}
```

④二叉树的线索化

在了解了线索二叉树中线索的来龙去脉以及线索二叉树节点的构成之后，我们接下来就来了解一下如何将一个普通的二叉树结构进行线索化。

正如前文中提到的：线索二叉树中所有的线索均是基于对这个二叉树进行中序遍历而得到的。

所以，对一个一般二叉树结构进行线索化的过程，等价于对其进行一次中序遍历的过程。

至于这一次中序遍历是使用栈结构进行辅助还是直接上递归走你，这些都在允许范围内；而真正的关键，是对二叉树进行线索化的步骤。

为了方便理解，笔者将使用递归的方式，就二叉树线索化的流程进行讲解：

步骤1：创建一个全局变量，用于存储当前节点的前驱节点，这个变量的初始值为null；

步骤2：创建一个方法，传递当前被遍历节点为参数，使用递归的方式中序遍历以当前节点为跟的整个二叉树；

步骤3：如果当前节点存在左孩子，优先递归处理左子树；

步骤4：如果当前节点不存在左孩子，则使用当前节点左孩子域指向前驱结点全局变量取值，并标记左孩子保存内容为线索；

步骤5：若当前全局变量指向的前驱节点不为空，且前驱节点的右孩子为空，则使用前驱节点的右孩子域指向当前节点，并标记前驱节点的右孩子域保存内容为线索；

步骤6：更新全局变量的前驱节点为当前节点；

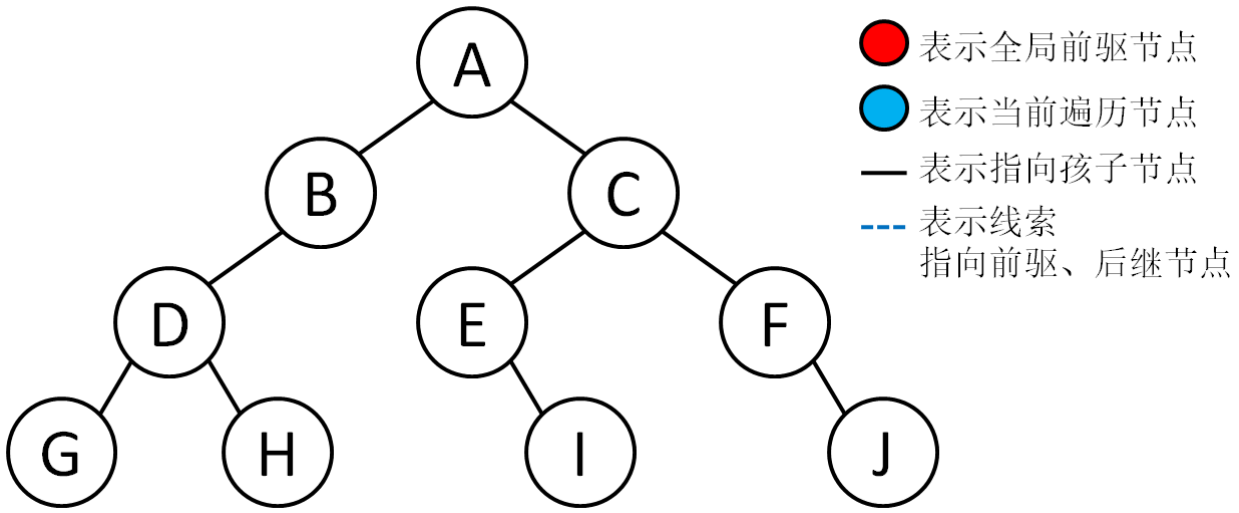
步骤7：如果当前节点存在右孩子，再去递归处理右子树。

从上面的7个步骤来看，二叉树的线索化操作好像挺难的……（内心再次OS：我太难了！），但是如果我们品，我们细细的品……

就不难发现：实际上步骤3~7就是将一个递归实现的中序遍历操作进行了变形，将两次递归之间访问当前节点数据域的步骤，换成了当前节点和前驱节点线索化的过程。（有内味儿了！）

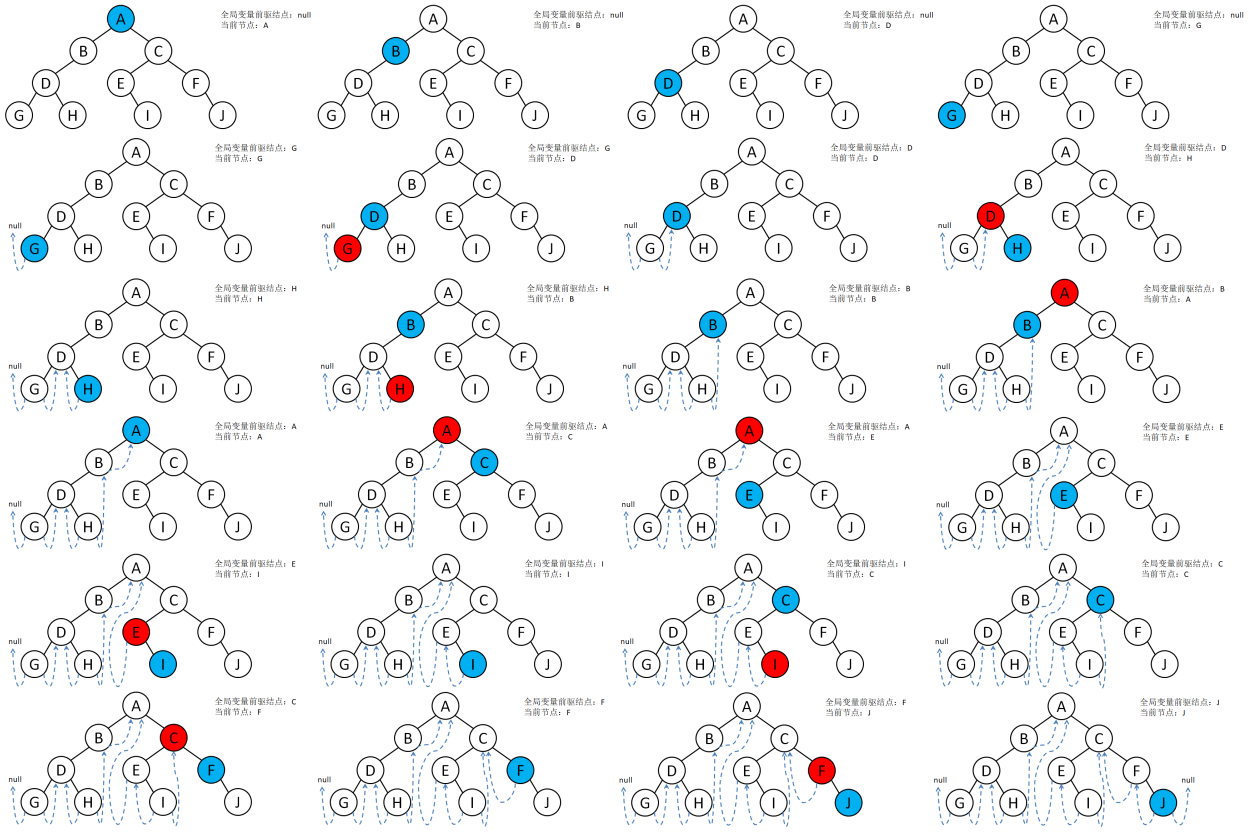
百闻不如一见，接下来就让我们使用一组案例，来观察对二叉树进行线索化的过程。

给出原始的二叉树结构：



怎么样？眼熟不？没错，这就是在我们之前二叉树章节中使用过的原图！

下面我们一起按照本图，完成对二叉树线索化的操作过程（前方多图警告！）：



上面的一组图片，体现的就是将一个一般的二叉树进行线索化的全部过程，从这个过程中我们能够很清晰的看出：

1.二叉树的线索化流程，就是一次对二叉树结构进行中序序列遍历的过程；

2.线索化之后的线索二叉树，除了最左边中序遍历的起始节点的左孩子，以及最右边中序遍历终点节点的右孩子这两个指针域之外，其他空白指针域全部利用了起来。

在了解了线索二叉树的相关内容之后，我们的“法杖”也就准备的差不多了，那么接下来，就是真正开始学习高阶法术——Morris遍历——的时候了！

3.Morris遍历

①如胶似漆：Morris遍历与线索二叉树

Morris遍历的核心思想，就是按照操作双向链表那样来操作一个二叉树，从某种意义上来讲，就是将二叉树这种非线性存储结构向链表这种线性存储结构进行转换的一种思想。

所以在这个层面上来讲，Morris遍历本身就是依赖二叉树的线索性来实现的。

很多介绍Morris遍历的文章当中，都将Morris遍历以线索二叉树分解开来，分别作为独立的知识点进行介绍，这几种做法实际上无伤大雅，

充其量就是在对一个普通的二叉树结构进行Morris遍历的同时，一遍建立线索结构，一遍进行Morris遍历。

但是经过笔者的实践尝试发现：Morris遍历与线索二叉树共同服用，效果最佳！原因有二：

其一：在线索二叉树的基础上进行Morris遍历，可以省去每一次执行遍历之前都对二叉树重新线索化的过程，有利于提升效率；

其二：通过一些特殊的操作手段，可以在对线索二叉树进行Morris遍历的同时，避免对线索二叉树的线索特性进行破坏。

故而，在上述两点因素的基础上，我们选择让Morris遍历与线索二叉树喜结连理，永结同心！

并在二者共同的加持之下，学习Morris遍历对线索二叉树的三种遍历顺序的操作。

②Morris遍历三序

Morris遍历中使用到的线索二叉树结构，同样是以二叉树中序遍历序列为基础构建的。

虽然线索的构建顺序以中序序列为基础，但是Morris遍历却能够根据这一种顺序，来完成对一颗线索二叉树先序、中序、后序这三种序列的遍历操作。

其中，又以中序序列遍历最为简单（毕竟线索二叉树本身就是按照中序序列构建的），先序序列遍历次之，后序序列遍历最难！

所以我们将按照中序遍历、先序遍历、后序遍历的顺序，对Morris遍历思想进行介绍。

1.Morris中序遍历

遍历步骤：

步骤1：若当前节点左侧记录为左孩子节点，则找到当前节点的前驱节点；

1.1若当前节点的前驱节点尚未在中序遍历序列中出现过，则取当前节点的左孩子节点，继续遍历；

1.2若当前节点的前驱节点已经在中序遍历序列中出现过，则将当前节点加入遍历序列，并取当前节点的右孩子节点，继续遍历；

步骤2：若当前节点左侧记录为前驱线索节点，则直接将当前节点加入遍历序列，并取当前节点的右孩子节点，继续遍历；

步骤3：重复步骤1-2，直到当前节点为null为止，遍历完成。

步骤解释：

解释1：如果当前节点的左侧记录的是一个左孩子节点，考虑到当前节点的两种来源：

1.第一次遍历到这个节点；

2.从其前驱节点通过线索指回这个节点；

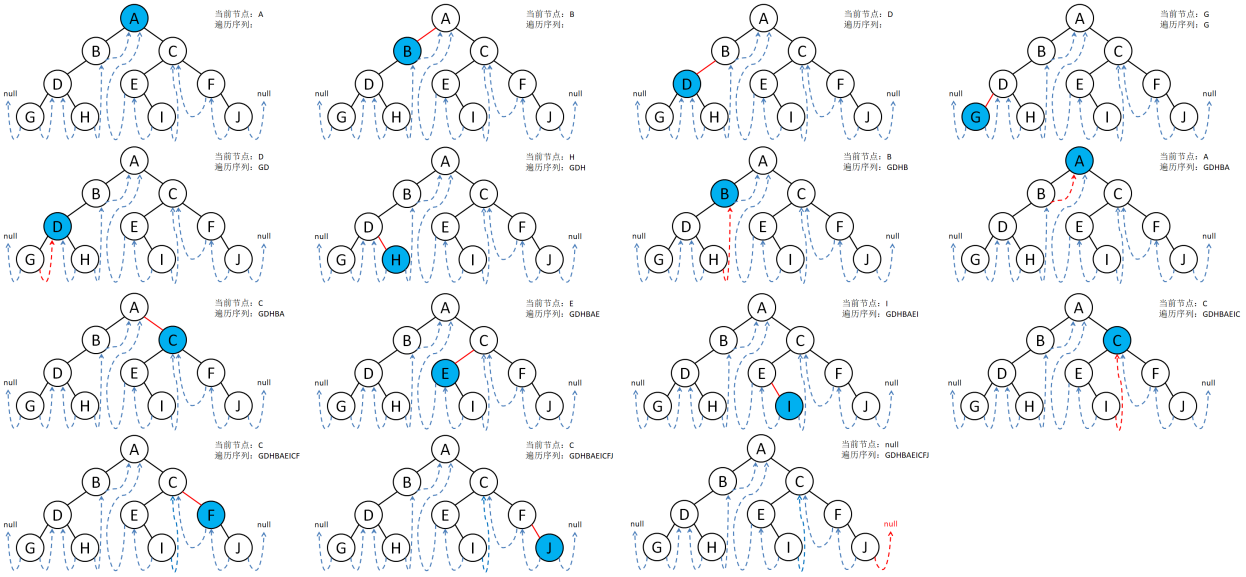
我们必须根据当前遍历序列是否包含其前驱节点来判断目前发生的上述两种情况中的哪一种：

1.如果遍历序列中不包含前驱节点，则说明是第一次遍历到这个节点，因为这个节点还有左子树，所以优先遍历左子树；

2.如果遍历序列中包含前驱节点，则说明是根据当前节点的前驱节点到达这个节点的，其左子树已经遍历完成，直接将当前节点纳入遍历序列即可；

解释2：如果当前节点左侧记录的是前驱线索，则说明在原有二叉树结构当中，当前节点的左孩子为空，在中序遍历中，直接将当前节点纳入遍历序列即可。

遍历图例：



2.Morris先序遍历

遍历步骤：

步骤1：若当前节点左侧记录为左孩子节点，则找到当前节点的前驱节点；

1.1若当前节点的前驱节点尚未在先序遍历序列中出现过，则将当前节点加入遍历序列，并取当前节点的左孩子节点，继续遍历；

1.2若当前节点的前驱节点已经在先序遍历序列中出现过，则取当前节点的右孩子节点，继续遍历；

步骤2：若当前节点左侧记录为前驱线索节点，则直接将当前节点加入遍历序列，并取当前节点的右孩子节点，继续遍历；

步骤3：重复步骤1-2，直到当前节点为null为止，遍历完成。

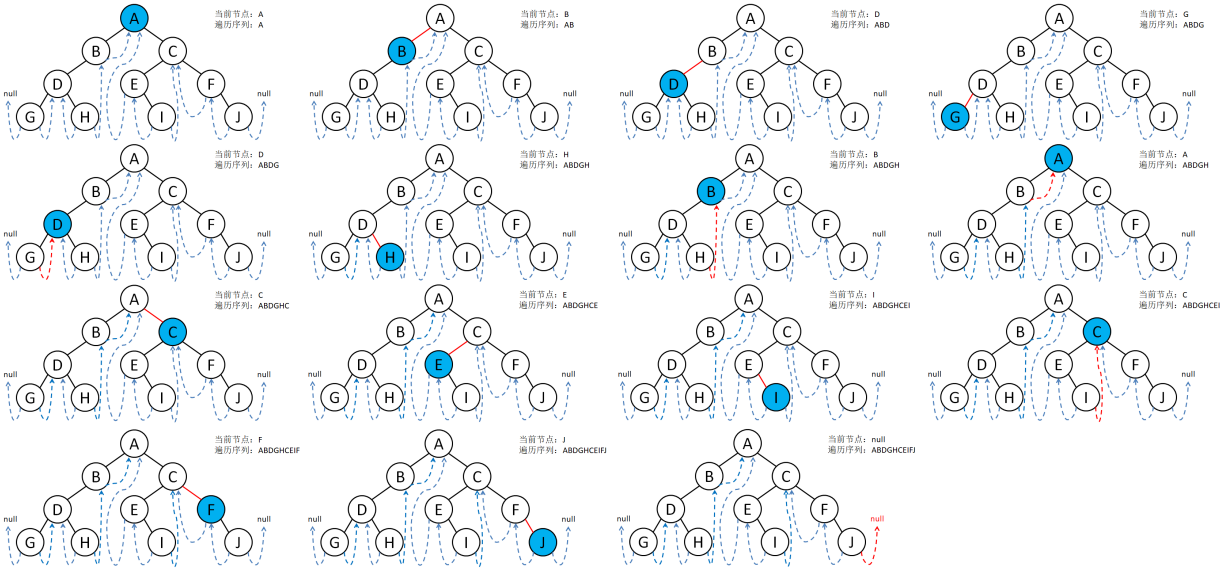
步骤解释：

解释1：Morris先序遍历的整体流程与Morris中序遍历的整体流程相似，只是当前节点加入遍历序列的时机有所变化；

解释2：如果当前节点的左侧保存的是左孩子节点，那么在对左孩子节点进行访问之前，首先要将当前节点加入遍历序列，才能够得到正确的先序序列，

解释3：与Morris中序遍历相似的是，我们同样需要通过判断当前节点的前驱节点是否存在于遍历序列中，来判断是第一次到达当前节点还是从前驱节点通过线索返回当前节点。

遍历图例：



3.Morris后序遍历

注意：相比起Morris中序遍历和Morris先序遍历，Morris后续遍历相对比较麻烦，请注意观察执行步骤！

遍历步骤：

步骤1：为整个线索二叉树创建一个临时根节点，临时根节点的左孩子域指向线索二叉树的树根，并标记临时根节点的左侧标记域取值为子节点类型；

步骤2：调整线索二叉树中中序遍历终点节点的右孩子域（原本指向为null）指向步骤1中定义的临时节点，并标记根节点的右侧标记域取值为线索类型；

步骤3：从步骤1中定义的临时根节点开始，对线索二叉树进行后续遍历；

步骤4：若当前节点左侧记录为左孩子节点，则进行如下判断：

4.1若当前节点的左孩子节点尚未在后序遍历序列中出现过，则取当前节点的左孩子节点，继续遍历；

4.2若当前节点的左孩子节点已经在后序遍历序列中出现过，则判断当前节点右侧保存的是否为后继线索：

4.2.1若当前节点右侧保存的是后继线索，则将当前节点加入遍历序列当中，并取当前节点的右孩子节点取值（后继线索），继续遍历

4.2.2若当前节点右侧保存的是右孩子节点，则直接取当前节点的右孩子节点取值（右孩子节点），继续遍历

步骤5：若当前节点左侧记录为前驱线索节点，则进行如下判断：

5.1若当前节点的右侧记录为后继线索节点，则将当前节点改变为当前节点的后继节点（当前节点的右孩子线索指向节点），计算其前驱节点，

并计算当前节点左孩子节点到达前驱节点之间所经历所有节点的路径，逆序加入到遍历序列当中；

5.2若当前节点的右侧记录为右孩子节点，则直接取得当前节点的右孩子节点，遍历继续。

步骤6：重复步骤4-5，直到当前节点为null为止，遍历完成。

步骤解释：

解释1：Morris后续遍历算法的整体思想，可以看做是将一个线索二叉树中所有的节点，按照从左到右、从上至下的逆序进行遍历并输出的过程；

解释2：Morris后续遍历的实现，依赖的是找到一个节点之后，获取其左孩子节点到其前驱节点（左子树中最右节点）过程中全部节点的序列，并进行逆序输出，

但是在这个过程中，并不包含当前节点。也就是说，我们想要打印任意连续从左上至右下路径上连续节点的逆序序列，我们必须首先找到这个路径的后继节点才行。

所以为了保证线索二叉树的根节点，以及根节点所在最左上至右下的一条路径上，所有节点能够遍历到并逆序输出，我们必须为这条路径安排一个后继节点，

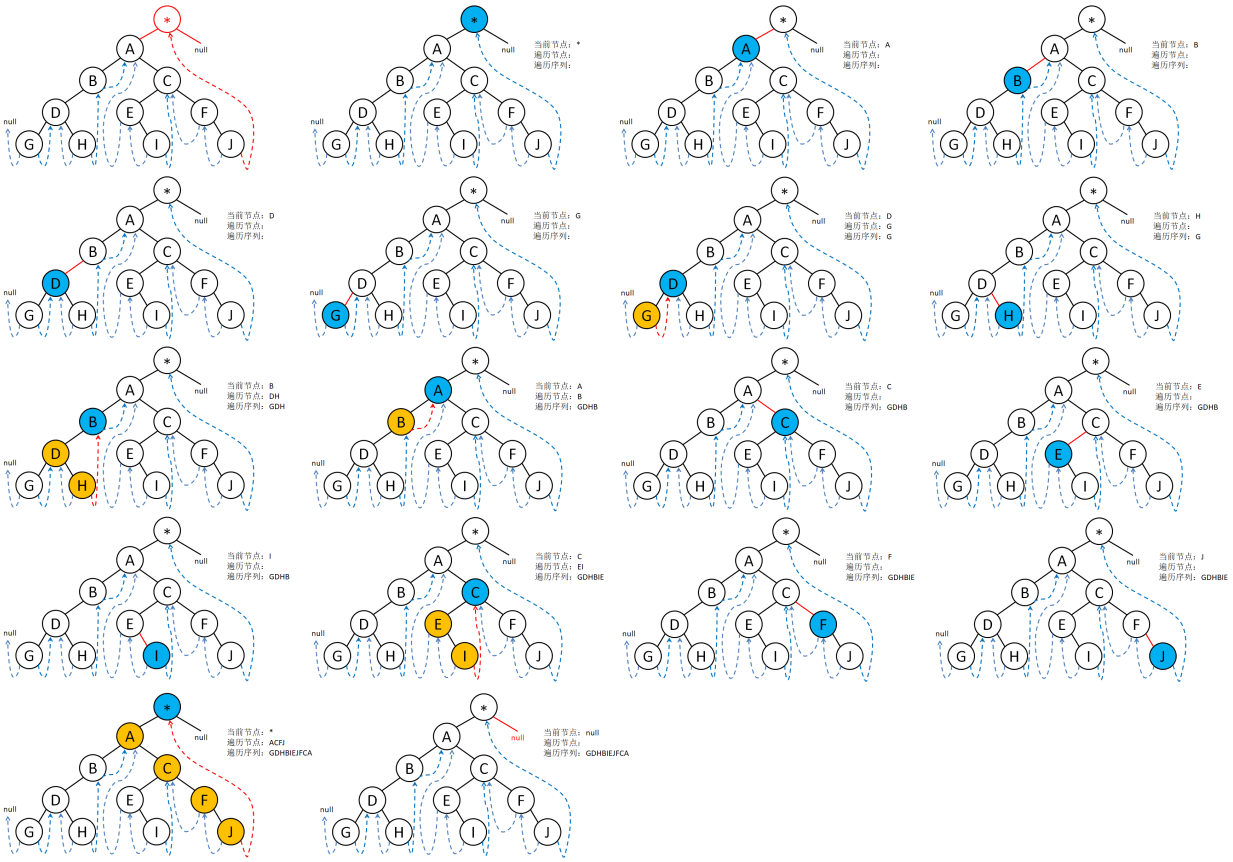
而这个后继节点也就是在步骤1-3中创建的临时根节点。也就是说，只有依赖这个临时根节点，才能够将线索二叉树正常根节点所在的最左上至右下一列上的全部节点遍历到，

并逆序输出到后续遍历序列当中去；

解释3：任意节点，如果左侧指针域存储的是线索，或者存储是孩子节点，但是左孩子已经在遍历序列中出现过，并且这个节点的右侧指针域保存的是线索的话，

则说明当前节点在线索化之前其右侧指针域指向null，而在当前的条件之下，相当于这个节点的左右子树已经遍历完成，按照后序遍历的顺序，当前节点应该将自己加入遍历序列当中。

遍历图例：



注意事项

在上述三种遍历方式当中，我们经常会用这样一段代码：判断某一节点（一般是当前节点的前驱节点，或者当前节点的左孩子节点），是否在遍历序列中出现过。

这一功能的作用，是用来判断我们是如何来到当前节点的，进而用来确定我们是第几次访问到当前节点：

如果当前节点的前驱节点没有在遍历序列中出现过，那么说明我们是第一次访问当前节点；

如果当前节点的前驱节点已经在遍历序列中出现过，那么说明我们是通过当前节点的前驱节点跳转到当前节点的。

值得一说的是，这种判断方式，仅仅适用于那些元素内容不重复的线索二叉树结构。

如果在某一线索二叉树结构当中，允许出现重复元素，那么上述方式将不再适用，此时我们可以在通过某一节点找到其后继节点之后，断开后继i线索的方式，

来替代上述功能。在后继线索断开后，我们可以判断当前节点的前驱节点的右孩子是否指向当前节点的方式，来判断其前驱节点是否已经遍历过，

进而进一步的确定，当前节点是第一次遍历，还是通过前驱节点的后继线索，重新回到当前节点上的：

如果当前节点的前驱节点的右孩子依然指向当前节点，说明前驱节点的后继线索尚未断开，前驱节点没有访问过，我们是第一次访问当前节点；

如果当前节点的前驱节点的右孩子不再指向当前节点，说明前驱节点的后继线索已经断开，已经访问过前驱节点，我们不是第一次访问当前节点了。

也就是说，上述解决方案是以牺牲线索二叉树结构的后继线索性为代价的。