二叉树的遍历和线索二叉树

二叉树的遍历

二叉树的遍历是指按某条搜索路径访问树中每个结点，使得每个结点均被访问一次，而且仅被访问一次。由于二叉树是一种非线性结构，每个结点都可能有两棵子树，因此需要寻找一种规律，以便使二叉树上的结点能排列在一个线性队列上，进而便于遍历。

二叉树遍历方式的分析（2009、2011、2012）

（算法题）二叉树遍历的相关应用（2014、2017、2022）+

由二叉树的递归定义可知，遍历一棵二叉树便要决定对根结点N、左子树L和右子树R的访问顺序。按照先遍历左子树再遍历右子树的原则，常见的遍历次序有先序（NLR）、中序（LNR）和后序（LRN）三种遍历算法，其中“序”指的是根结点在何时被访问。

先序遍历（PreOrder）

若二叉树为空，则什么也不做；否则，

1. 访问根结点；
2. 先序遍历左子树；
3. 先序遍历右子树。

下图中的虚线表示对该二叉树进行先序遍历的路径，得到先序遍历序列为124635。

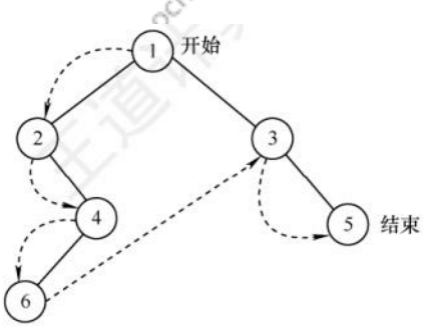

图 5.7　二叉树的先序遍历

对应的递归算法如下：

```
void PreOrder(BiTree T){
    if(T!=NULL){
        visit(T);//访问根结点
        PreOrder(T->lchild);//递归遍历左子树
        PreOrder(T->rchild);//递归遍历右子树
    }
}
```

中序遍历（InOrder）

若二叉树为空，则什么也不做；否则，

1. 中序遍历左子树；
2. 访问根结点；
3. 中序遍历右子树。

中序序列中结点关系的分析（2017）

下图中的虚线表示对二叉树进行中序遍历的路径，得到中序遍历序列为264135.

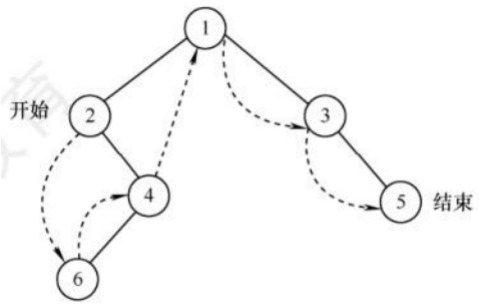

图 5.8　二叉树的中序遍历

对应的递归算法如下：

```
void InOrder(BiTree T){
    if(T!=NULL){
        InOrder(T->lchild);//递归遍历左子树
        visit(T);//访问根结点
        InOrder(T->rchild);//递归遍历右子树
    }
}
```

后序遍历（PostOrder）

若二叉树为空，则什么也不做；否则，

1. 后序遍历左子树；
2. 后序遍历右子树；
3. 访问根结点。

下图中的虚线表示对该二叉树进行后序遍历的路径，得到后序遍历序列为642531。

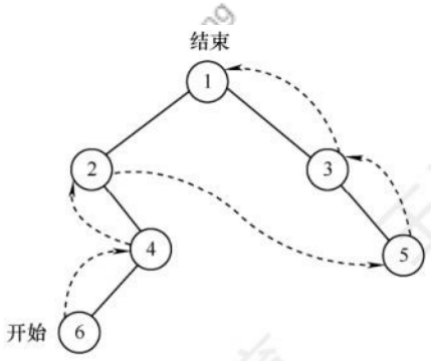

图 5.9　二叉树的后序遍历

对应的递归算法如下：

```
void PostOrder(BiTree T){
    if(T!=NULL){
        PostOrder(T->lchild);//递归遍历左子树
        PostOrder(T->rchild);//递归遍历右子树
        visit(T);//访问根结点
    }
}
```

上述三种遍历算法中，递归遍历左、右子树的顺序都是固定的，只是访问根结点的顺序不同。不管采用哪种遍历算法，每个结点都访问一次且仅访问一次，所以时间复杂度是O(n)。在递归遍历中，递归工作栈的栈深恰好为树的深度，所以在最坏情况下，二叉树是有n个结点且深度为n的单支树，遍历算法的空间复杂度为O(n)。

递归算法和非递归算法的转换

在上节介绍的三种遍历算法中，暂时抹去和递归无关的visit()语句，则3个遍历算法完全相同，因此，从递归执行过程的角度看先序、中序和后序遍历也是完全相同的。

非递归遍历算法的难度较大，统考对非递归遍历算法的要求通常不高。

下图用带箭头的虚线表示了这三种遍历算法的递归执行过程。其中，向下的箭头表示更深一层的递归调用，向上的箭头表示从递归调用退出返回；虚线旁的三角形、圆形和方形内的字符分别表示在先序、中序和后序遍历的过程中访问根结点时输出的信息。例如，由于中序遍历中访问结点是在遍历左子树之后、遍历右子树之前进行的，则带圆形的字符标在向左递归返回和向右递归调用之间。由此，只要沿虚线从1出发到2结束，将沿途所见的三角形（或圆形或方形）内的字符记下，便得到遍历二叉树的先序（或中序或后序）序列。例如，下图中，沿虚线游走可以分别得到先序序列为ABDEC、中序序列为DBEAC、后序序列为DEBCA。

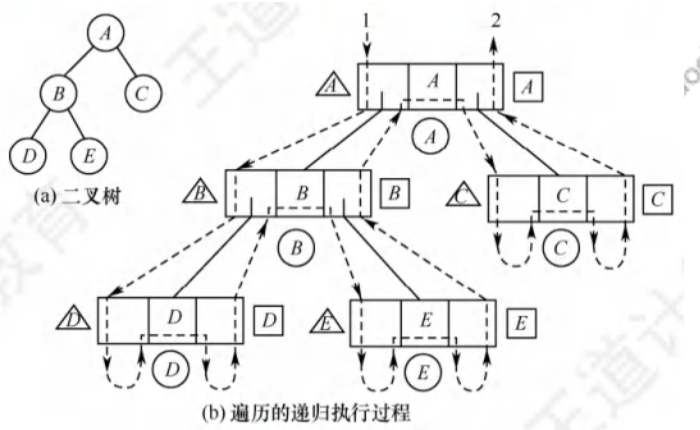

图 5.10　三种遍历过程示意图

借助栈的思路，我们来分析中序遍历的访问过程：

1. 沿着根的左孩子，依次入栈，直到左孩子为空，说明已找到可以输出的结点，此时栈内元素依次为ABD。
2. 栈顶元素出栈并访问：若其右孩子为空，继续执行2；
3. 若其右孩子不空，将右子树转执行1。

栈顶D出栈并访问，它是中序序列的第一个结点；D右孩子为空，栈顶B出栈并访问；B右孩子不空，将其右孩子E入栈，E左孩子为空，栈顶E出栈并访问；E右孩子为空，栈顶A出栈并访问；A右孩子不空，将其右孩子C入栈，C左孩子为空，栈顶C出栈并访问。由此得到中序序列DBEAC。读者可根据上述分析画出遍历过程的出入栈示意图。

根据分析可以写出中序遍历的非递归算法如下：

```
void InOrder2(BiTree T){
    InitStack(S);BiTree p=T;//初始化栈S;p是遍历指针
    while(p||!IsEmpty(S)){//栈不空或p不空时循环
        if(p){//一路向左
            Push(S,p);//当前结点入栈
            p=p->lchild;//左孩子不空,一直向左走
        }
        else{//出栈，并转向出栈结点的右子树
            Pop(S,p);visit(p);//栈顶元素出栈，访问出栈结点
            p=p->rchild;//向右子树走，p赋值为当前结点的右孩子
        }//返回while循环继续进入if-else语句
    }
}
```

先序遍历和中序遍历的基本思想是类似的，只需把访问结点操作放在入栈操作的前面，读者可以参考中序遍历的过程说明自行模拟出入栈示意图。先序遍历的非递归算法如下：

```
void PreOrder2(BiTree T){
    InitStack(S);BiTree p=T;//初始化栈S;p是遍历指针
    while(p||!IsEmpty(S)){
        if(p){
            visit(p);Push(S,p);//访问当前结点，并入栈
            p=p->lchild;//左孩子不空，一直向左走
        }
        else{//出栈,并转向出栈结点的右子树
            Pop(S,p);//栈顶元素出栈
            p=p->rchild;//返回while循环继续进入if-else语句
        }
    }
}
```

后序遍历的非递归实现是三种遍历方法中最难的。因为在后序遍历中，要保证左孩子和右孩子都已被访问并且左孩子在右孩子前访问才能访问根结点，这就为流程的控制带来了难题。

后序非递归遍历算法的思路分析：从根结点开始，将其入栈，然后沿其左子树一直往下搜索，直到搜索到没有左孩子的结点，但是此时不能出栈并访问，因为若其有右子树，则还需按相同的规则对右子树进行处理。直到上述操作进行不下去，若栈顶元素想要出栈被访问，要么右子树为空，要么右子树刚被访问完（此时左子树早已访问完），这样就保证了正确的访问顺序。

后序遍历的非递归算法遍历图a中的二叉树，当访问到E时，ABD都已入过栈，对于后序非递归遍历，当一个结点的左右子树都被访问后才会出栈，图中D已出栈，此时栈内还有A和B，这是E的全部祖先。实际上，访问一个结点p时，栈中结点恰好是结点p的所有祖先，从栈底到栈顶结点再加上结点p，刚好构成从根结点到结点p的一条路径。在很多算法设计中都可以利用这一思路来求解，如求根到某结点的路径、求两个结点的最近公共祖先等。

层次遍历

图中所示为二叉树的层次遍历，即按照箭头所指方向，按照1，2，3，4的层次顺序，自上而下，从左至右，对二叉树中的各个结点进行逐层访问。

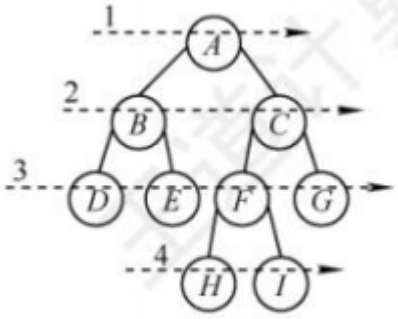

图 5.11　二叉树的层次遍历

进行层次遍历，需要借助一个队列。层次遍历的思想如下：

1. 首先将二叉树的根结点入队。
2. 若队列非空，则队头结点出队，访问该结点，若它有左孩子，则将其左孩子入队；若它有右孩子，则将其右孩子入队。
3. 重复2步，直至队列为空。

二叉树的层次遍历算法如下：

```
void LevelOrder(BiTree T){
    InitQueue(Q);//初始化辅助队列
    BiTree p;
    EnQueue(Q,T);//将根结点入队
    while(!IsEmpty(Q)){//队列不空则循环
        DeQueue(Q,p);//队头结点出队
        visit(p);//访问出队结点
        if(p->lchild!=NULL)
            EnQueue(Q,p->lchild);//若左孩子不空，则左孩子入队
        if(p->rchild!=NULL)
            EnQueue(Q,p->rchild);//若右孩子不空，则右孩子入队
    }
}
```

上述二叉树层次遍历的算法，读者在复习过程中应将其作为一个模板，在熟练掌握其执行过程的基础上来记忆，并达到熟练手写的程度，这样才能将模板应用于各种题目之中。

遍历是二叉树各种操作的基础，例如对于一棵给定二叉树求结点的双亲、求结点的孩子、求二叉树的深度、求叶结点的个数、判断两棵二叉树是否相同等。所有这些操作都是在遍历的过程中进行的，因此必须掌握二叉树的各种遍历过程，并能灵活运用以解决各种问题。

由遍历序列构造二叉树

先序序列对应的不同二叉树的分析（2015）

对于一棵给定的二叉树，其先序序列、中序序列、后序序列和层序序列都是确定的。然而，只给出四种遍历序列中的任意一种，却无法唯一地确定一棵二叉树。若已知中序序列，再给出其他三种遍历序列的任意一种，就可以唯一地确定一棵二叉树。

由先序序列和中序序列构造二叉树

先序序列和中序序列相同时确定的二叉树（2017）

由先序序列和中序序列构造一棵二叉树（2020、2021）

在先序序列中，第一个结点一定是二叉树的根结点；而在中序遍历中，根结点必然将中序序列分割成两个子序列，前一个子序列是根的左子树的中序序列，后一个子序列是根的右子树的中序序列。左子树的中序序列和先序序列的长度是相等的，右子树的中序序列和先序序列的长度是相等的。根据这两个子序列，可以在先序序列中找到左子树的先序序列和右子树的先序序列，如下图所示。如此递归地分解下去，便能唯一地确定这棵二叉树。

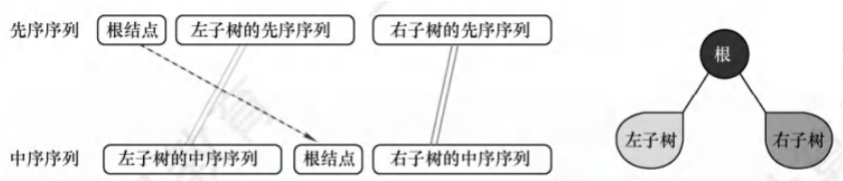

图 5.12　由先序序列和中序序列构造二叉树

例如，求先序序列（ABCDEFGHI）和中序序列（BCAEDGHFI）所确定的二叉树。首先，由先序序列可知A为二叉树的根结点。中序序列A之前的BC为左子树的中序序列，EDGHFI为右子树的中序序列。然后，由先序序列可知B是左子树的根结点，D是右子树的根结点。以此类推，就能将剩下的结点继续分解下去，最后得到的二叉树如图所示。

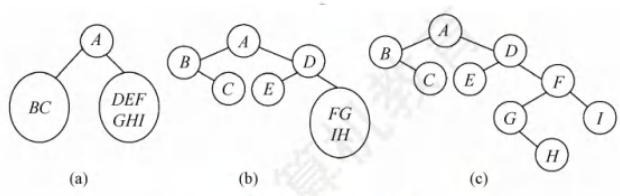

图 5.13　一棵二叉树的构造过程

由后序序列和中序序列构造二叉树

由后序序列和树形构造一棵二叉树（2017、2023）

同理，由二叉树的后序序列和中序序列也可以唯一地确定一棵二叉树。因为后序序列的最后一个结点就如同先序序列的第一个结点，可以将中序序列分割成两个子序列，如图所示，然后采用类似的方法递归地进行分解，进而唯一地确定这棵二叉树。

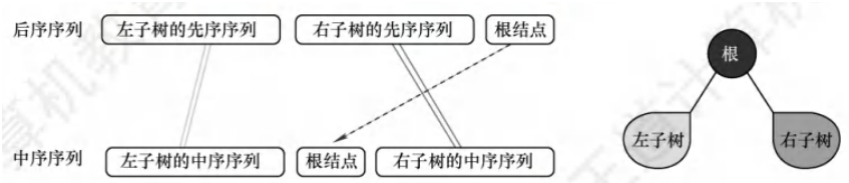

图 5.14　由后序序列和中序序列构造二叉树

请读者分析后序序列（CBEHGIFDA）和中序序列（BCAEDGHFI）所确定的二叉树。

由层序序列和中序序列构造二叉树

在层序遍历中，第一个结点一定是二叉树的根结点，这样就将中序序列分割成了左子树的中序序列和右子树的中序序列。若存在左子树，则层序序列的第二个结点一定是左子树的根，可进一步划分左子树；若存在右子树，则中序序列中紧接着的下一个结点一定是右子树的根，可进一步划分右子树，如图所示。采用这种方法继续分解，就能唯一确定这棵二叉树。

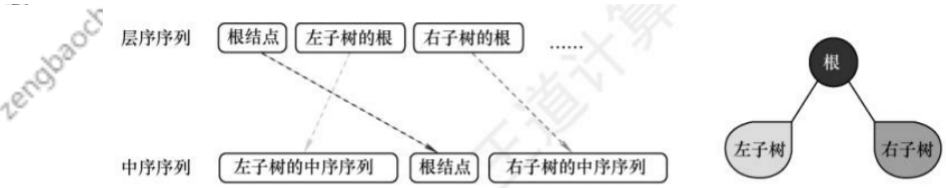

图 5.15　由层序序列和中序序列构造二叉树

请读者分析后序序列（ABDCEFGIH）和中序序列（BCAEDGHFI）所确定的二叉树。

需要注意的是，先序序列、后序序列和层序序列的两两组合，无法唯一确定一棵二叉树。例如，下图所示的两棵二叉树的先序序列都为AB，后序序列都为BA，层序序列都为AB。

图 5.16　两棵不同的二叉树

线索二叉树

线索二叉树的基本概念

遍历二叉树是以一定的规则将二叉树中的结点排列成一个线性序列，从而得到几种遍历序列，使得该序列中的每个结点（第一个和最后一个除外）都有一个直接前驱和直接后继。

后序线索二叉树的定义（2010）

传统的二叉链表存储仅能体现一种父子关系，不能直接得到结点在遍历中的前驱或后继。前面提到，在含n个结点的二叉树中，有n+1个空指针。这是因为每个叶结点都有2个空指针，每个度为1的结点都有1个空指针，空指针总数2n0+n1，又n0=n2+1，所以空指针总数为n0+n1+n2+1=n+1。由此设想能否利用这些空指针来存放指向其前驱或后继的指针？这样就可以像遍历单链表那样方便地遍历二叉树。引入线索二叉树正是为了加快查找结点前驱和后继的速度。

规定：若无左子树，令lchild指向其前驱结点；若无右子树，令rchild指向其后继结点。如图所示，还需增加两个标志域，以表示指针域指向左（右）孩子或前驱（后继）。

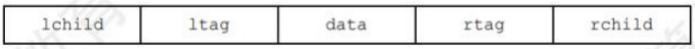

图 5.17　线索二叉树的结点结构

其中，标志域的含义如下：

$$ltag=\begin{cases}0, & \text{lchild域指示结点的左孩子}\\ 1, & \text{lchild域指示结点的前驱}\end{cases}$$

$$rtag=\begin{cases}0, & \text{rchild域指示结点的右孩子}\\ 1, & \text{rchild域指示结点的后继}\end{cases}$$

线索二叉树的存储结构描述如下：

```
typedef struct ThreadNode{
    ElemType data;//数据元素
    struct ThreadNode *lchild, *rchild;//左、右孩子指针
    int ltag, rtag;//左、右线索标志
}ThreadNode, *ThreadTree;
```

以这种结点结构构成的二叉链表作为二叉树的存储结构，称为线索链表，其中指向结点前驱和后继的指针称为线索。加上线索的二叉树称为线索二叉树。

中序线索二叉树的构造

二叉树的线索化是将二叉链表中的空指针改为指向前驱或后继的线索。而前驱或后继的信息只有在遍历时才能得到，因此线索化的实质就是遍历依次二叉树。

中序线索二叉树中线索的指向（2014）

以中序线索二叉树的建立为例。附设指针pre指向刚刚访问过的结点，指针p指向正在访问的结点，即pre指向p的前驱。在中序遍历的过程中，检查p的左指针是否为空，若为空就将它指向p，如图所示。

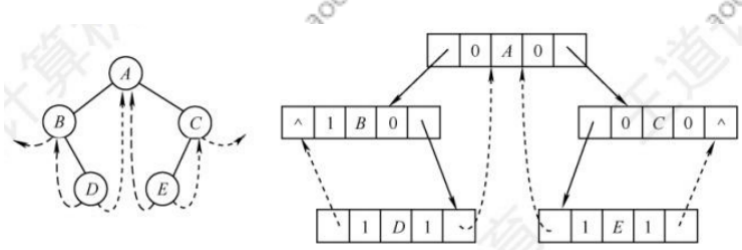

图 5.18　中序线索二叉树及其二叉链表示

通过中序遍历对二叉树线索化的递归算法如下：

```
void InThread(ThreadTree &p, ThreadTree &pre){
    if(p!=NULL){
        InThread(p->lchild, pre);//递归，线索化左子树
        if(p->lchild==NULL){//当前结点的左子树为空
            p->lchild=pre;//建立当前结点的前驱线索
            p->ltag=1;
        }
        if(pre!=NULL&&pre->rchild==NULL){//前驱结点非空且其右子树为空
            pre->rchild=p;//建立前驱结点的后继线索
            pre->rtag=1;
        }
        pre=p;//标记当前结点成为刚刚访问过的结点
        InThread(p->rchild,pre);//递归，线索化右子树
    }
}
```

通过中序遍历建立中序线索二叉树的主过程算法如下：

```
void CreateInThread(ThreadTree T){
    ThreadTree pre=NULL;
    if(T!=NULL){//非空二叉树，线索化
        InThread(T,pre);//线索化二叉树
        pre->rchild=NULL;//处理遍历的最后一个结点
        pre->rtag=1;
    }
}
```

为了方便，可以在二叉树的线索链表上也添加一个头结点，令其lchild域的指针指向二叉树的根结点，其rchild域的指针指向中序遍历时访问的最后一个结点；令二叉树中序序列中的第一个结点lchild域和最后一个结点的rchild域指针均指向头结点。这好比为二叉树建立了一个双向线索链表，方便从前往后或从后往前对线索二叉树进行遍历，如图所示。

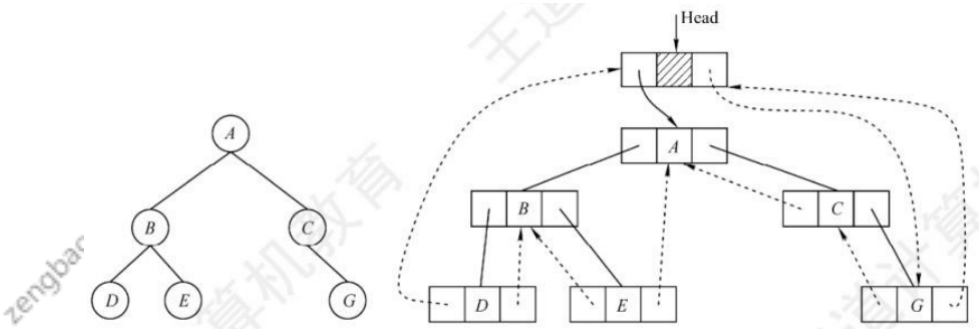

图 5.19　带头结点的中序线索二叉树

中序线索二叉树的遍历

中序线索二叉树的结点中隐含了线索二叉树的前驱和后继信息。在对其进行遍历时，只要先找到序列中的第一个结点，然后依次找结点的后继，直至其后继为空。在中序线索二叉树中找结点后继的规律是：若其右标志为“1”，则右链为线索，指示其后继，否则遍历右子树第一个访问的结点（右子树中最左下的结点）为其后继。不含头结点的线索二叉树的遍历算法如下。

1. 求中序线索二叉树的中序序列下的第一个结点：

```
ThreadNode *Firstnode(ThreadNode *p){
    while(p->ltag==0)p=p->lchild;//最左下结点（不一定是叶结点）
    return p;
}
```

2. 求中序线索二叉树中结点p在中序序列下的后继：

```
ThreadNode *Nextnode(ThreadNode *p){
    if(p->rtag==0)return Firstnode(p->rchild);//右子树中最左下结点
    else return p->rchild;//若rtag==1则直接返回后继线索
}
```

请读者自行分析并完成求中序线索二叉树的最后一个结点和结点p前驱的运算。

3. 利用上面两个算法，可写出不含头结点的中序线索二叉树的中序遍历的算法：

```
void Inorder(ThreadNode *T){
    for(ThreadNode *p=Firstnode(T);p!=NULL;p=Nextnode(p))
        visit(p);
}
```

先序线索二叉树和后序线索二叉树

上面给出了建立中序线索二叉树的代码，建立先序线索二叉树和建立后序线索二叉树的代码类似，只需变动线索化改造的代码段与调用线索化左右子树递归函数的位置。

以下图a的二叉树为例给出手动求先序线索二叉树的过程：先序序列为ABCDF，然后依次判断每个结点的左右链域，若为空，则将其改造为线索。结点A，B均有左右孩子；结点C无左孩子，将左链域指向前驱B，无右孩子，将右链域指向后继D；结点D无左孩子，将左链域指向前驱C，无右孩子，将右链域指向后继F；结点F无左孩子，将左链域指向前驱D，无右孩子，也无后继，所以置空，得到的先序线索二叉树如图b所示。求后序线索二叉树的过程：后序序列为CDBFA，结点C无左孩子，也无前驱，所以置空，无右孩子，将右链域指向后继D；结点D无左孩子，将左链域指向前驱C，无右孩子，将右链域指向后继B；结点F无左孩子，将左链域指向前驱B，无右孩子，将右链域指向后继A，得到的后序线索二叉树如图c所示。

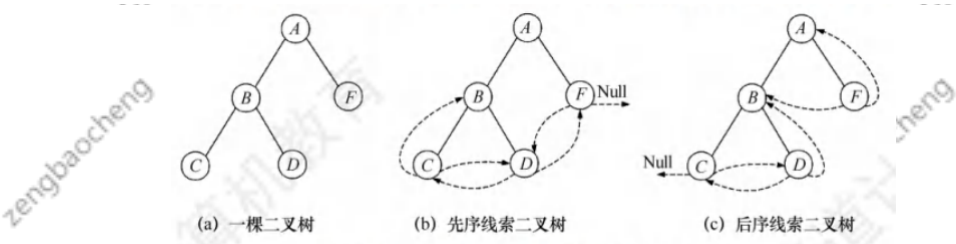

图 5.20 先序线索二叉树和后序线索二叉树

如何在先序线索二叉树中找结点的后继？若有左孩子，则左孩子就是其后继；若无左孩子但有右孩子，则右孩子就是其后继；若为叶结点，则右链域直接指示了结点的后继。

后序线索二叉树中线索的指向（2013）

在后序线索二叉树中找结点的后继较为复杂，可分三种情况：

1. 若结点x是二叉树的根，则其后继为空；
2. 若结点x是其双亲的右孩子，或是其双亲的左孩子且其双亲没有右子树，则其后继即为双亲；
3. 若结点x是其双亲的左孩子，且其双亲有右子树，则其后继为双亲的右子树上按后序遍历列出的第一个结点。c中找结点B的后继无法通过链域找到，可见在后序线索二叉树上找后继时需知道结点双亲，即需采用带标志域的三叉链表作为存储结构。