

Lecture 17: 二叉树的遍历

17.1 二叉树的递归遍历

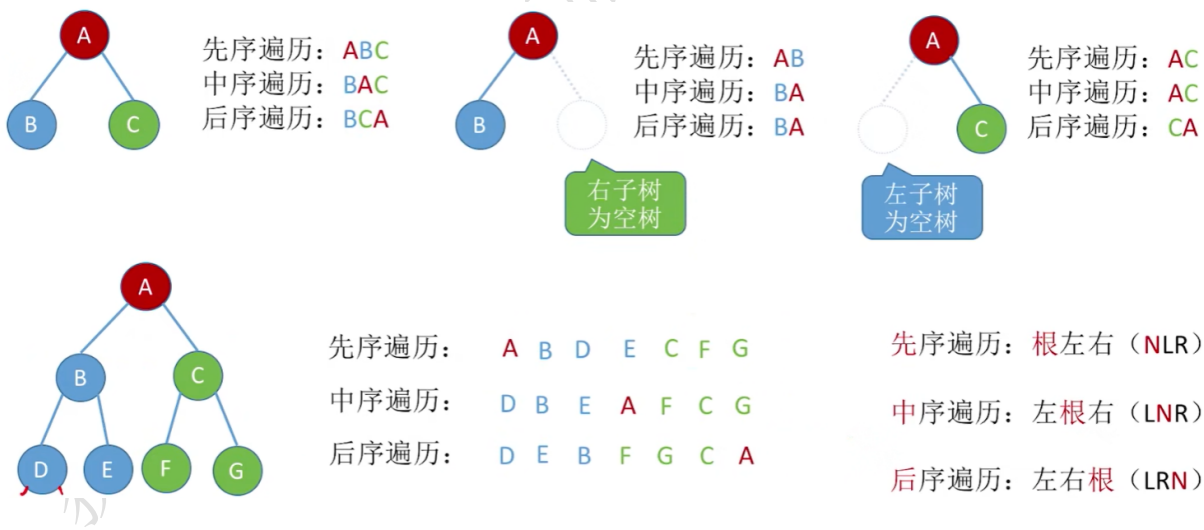
利用二叉树的递归特性遍历

- 等同于利用其左右节点来遍历;
- 使用先中后三种顺序进行遍历;

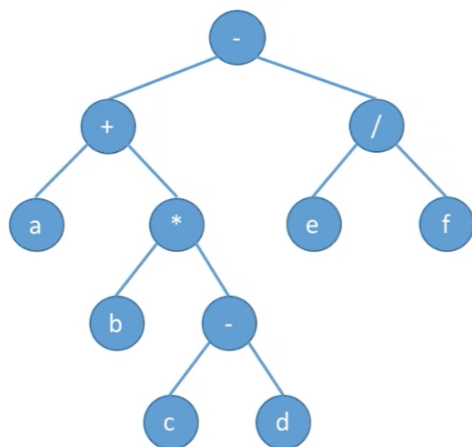
17.1.1 二叉树的遍历

二叉树的遍历

- 二叉树的递归特性
 - 1. 要么是空二叉树
 - 2. 要么就是由**根节点+左子树+右子树**组成的二叉树
- 三种遍历
 - 先序遍历: 根左右(NLR)
 - 中序遍历: 左根右(LNR)
 - 后序遍历: 左右根(LRN)
- 图示:



- 先序遍历: 根左右 (NLR)
- 中序遍历: 左根右 (LNR)
- 后序遍历: 左右根 (LRN)



先序遍历: $-+a*b-cd/ef$

中序遍历: $a+b*c-d-e/f$

后序遍历: $abcd-*+ef/-$

先序遍历 $\rightarrow$ 前缀表达式

中序遍历 $\rightarrow$ 中缀表达式 (需要加界限符)

后序遍历 $\rightarrow$ 后缀表达式

算数表达式的“分析树”

• $a + b * (c - d) - e / f$

17.1.2 先序遍历

先序遍历(PreOrder)

- 操作过程:
 - 1. 若二叉树为空, 则什么也不做
 - 2. 若二叉树非空
 - 1. 访问根节点
 - 2. 先序遍历左子树
 - 3. 先序遍历右子树
- 形象解释:
 - 从根节点出发, 画一条路;
 - 1. 如果左边还有没走的路, 优先往左边走, 走到路的尽头 (空结点)就往回走;
 - 2. 如果左边没路了, 就往右边走;
 - 3. 如果左、右都没路了, 则往上面走;
 - 总结: 先序遍历 $\rightarrow$ 第一次路过时访问结点
- 空间复杂度: $O(h)$
 - 每一个结点都会被路过 3 次

代码实现

```
//先序遍历
typedef struct BiTNode{
    ElemType data;
    struct BiTNode *lchild,*rchild;
}BiTNode,*BiTree;

void PreOrder(BiTree T){
    if(T!=null){
        visit(T);           //访问根节点
        PreOrder(T->lchild); //递归遍历左子树
    }
}
```

```
        PreOrder(T->rchild); //递归遍历右子树
    }
}
```

17.1.3 中序遍历

中序遍历(InOrder)

- 操作过程：
 - 1. 若二叉树为空，则什么也不做
 - 2. 若二叉树非空
 - 1. 先序遍历左子树
 - 2. 访问根节点
 - 3. 先序遍历右子树
- 形象解释：
 - 从根节点出发，一条路如果左边还有没走的路，优先往左边走
 - 走到路的尽头 (空结点)就往回走
 - 如果左边没路了，就往右边走
 - 如果左、右都没路了，则往上面走
 - 中序遍历：第二次路过时访问结点

代码实现

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

17.1.4 后序遍历

后序遍历

- 操作过程如下：
 - 1. 若二叉树为空，则什么也不做
 - 2. 若二叉树非空
 - 1. 先序遍历左子树
 - 2. 先序遍历右子树
 - 3. 访问根节点

代码实现

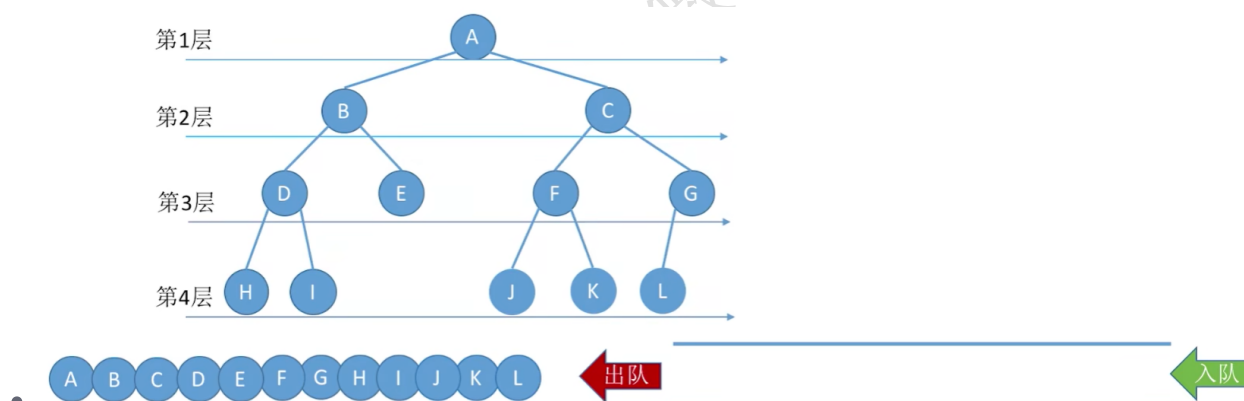
```
//后序遍历
void PostOrder(BiTree T){
    if(T!=NULL){
        PostOrder(T->lchild); //递归遍历左子树
        PostOrder(T->rchild); //递归遍历右子树
        visit(T);             //访问根节点
    }
}
```

17.2 二叉树的层次遍历

17.2.1 基础概念

算法思想

- 思想：
 1. 初始化一个辅助**队列**
 2. 根节点入队
 3. 若队列非空，则对头结点出队，访问该结点，并将其左右孩子插入队尾(如果有的话)
 4. 重复(3)直至队列为空
- 图示：



17.2.2 代码实现

```
//层序遍历
void LevelOrder(BiTree T){
    LinkQueue Q;
    InitQueue(Q); //初始化辅助队列
    BiTree p;
    EnQueue(Q, T); //将根节点入队
    while(!IsEmpty(Q)){ //队列不空则循环
        DeQueue(Q, p); //对头结点出队
        visit(p);      //访问出队结点
        if(p->lchild!=NULL)
            EnQueue(Q, p->lchild); //左孩子入队
        if(p->rchild!=NULL)
            EnQueue(Q, p->rchild); //右孩子入队
    }
}
```

```
}  
}
```

```
//二叉树的结点(链式存储)  
typedef struct BiTNode{  
    char data;  
    struct BiTNode *lchild,*rchild;  
}BiTNode,*BiTree;  
  
//链式队列结点  
typedef struct LinkNode{  
    BiTNode *data; //存指针而不是结点  
    struct LinkNode *next;  
}LinkNode;  
  
typedef struct{  
    LinkNode *front,*rear; //队头队尾  
}LinkQueue;
```

17.3 遍历序列构造二叉树

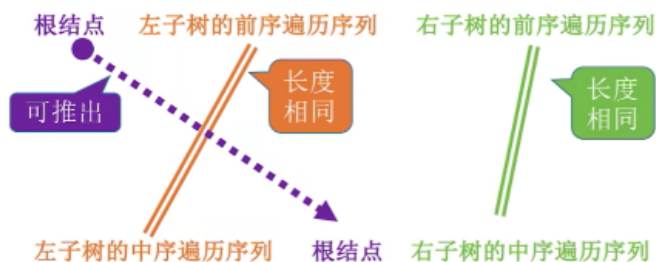
引入

- 概念：
 - 若只给出一颗二叉树的 **前/中/后** 序遍历序列中的一种，不能确定一颗二叉树；
- 原因：
 - **同一个遍历序列，可以对应成不同的二叉树；**
- 方法：
 - 给出先中后序列中的两种，可以唯一确定一颗二叉树；
 - 用前后序确定根节点(层序同理)，中序遍历确定左右子树；
- 构造：
 - 前序 + 中序；
 - 后序 + 中序；
 - 层序 + 中序；
- 方法：
 - 1. 圈根节点：看前序或后序序列，确定当前根节点位置；
 - 2. 划两条线：在中序序列中圈出根节点，其左边部分到前序、后序序列中画一条线，右边部分在前序、后序序列中画一条线；
 - 3. 找父节点：分别在两条划线的序列中，根据先序、后序序列的逻辑，找到父节点；
 - 4. 重复；

前序+中序

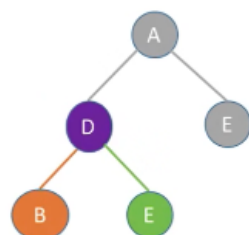
- 概念：
 - 用前序序列结合中序序列，分别交叉判断成分；
 - 前序序列 -> 根节点 -> 中序序列 -> 左右节点；

- 图示:



前序遍历序列: A D B C E

- 中序遍历序列: B D C A E

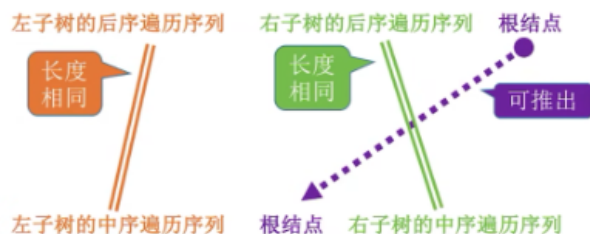


后序+中序

- 概念:

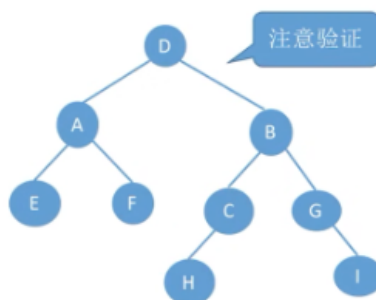
- 后序序列 $\rightarrow$ 根节点 $\rightarrow$ 中序序列 $\rightarrow$ 左右节点;

- 图示:



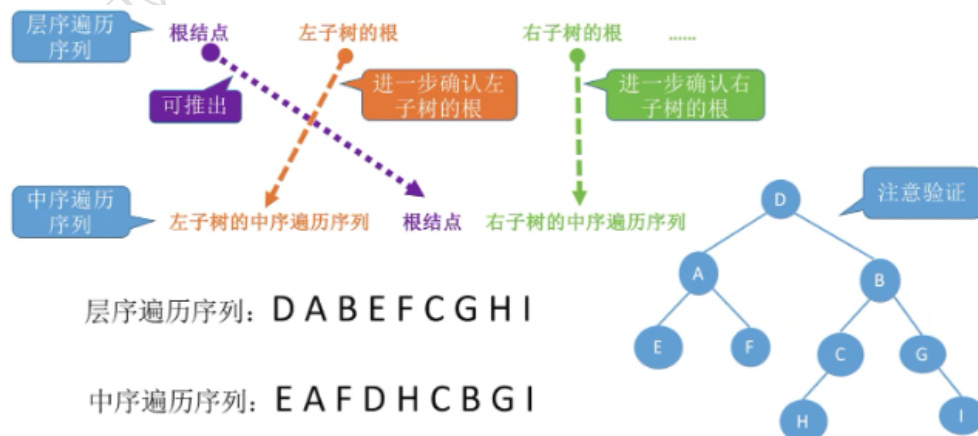
后序遍历序列: E F A H C I G B D

中序遍历序列: E A F D H C B G I



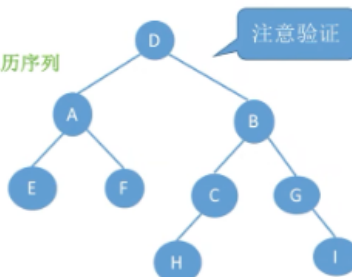
层序+中序

- 图示:



层序遍历序列: D A B E F C G H I

中序遍历序列: E A F D H C B G I



其他方式: 前序、后序、层序序列两两组合 $\rightarrow$ 不能还原出二叉树;

17.4 线索二叉树的概念

17.4.1 中序线索二叉树

一般中序遍历的问题

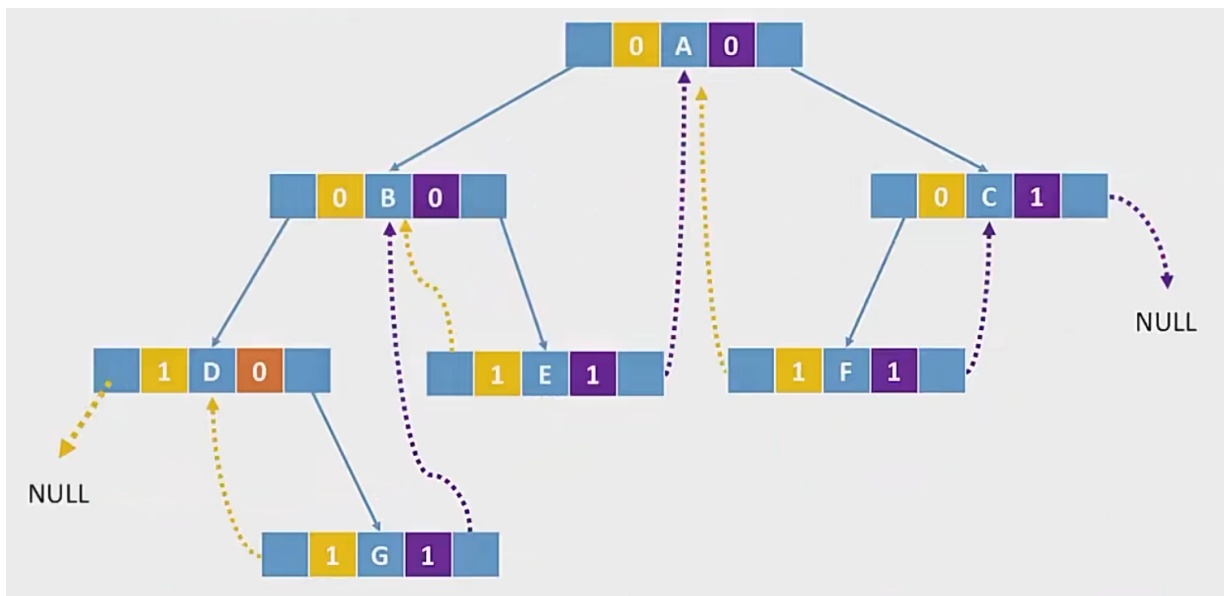
- 实现：
 - 树的中序遍历 $\rightarrow$ 必须总是从根节点开始遍历 $\rightarrow$ 如果只有普通节点、不知道根节点，此时往上找；
- 问题：
 - 传统的遍历，无法找到当前节点的前驱；
 - 所以在线索二叉树中，依据遍历得到的顺序，为结点建立前序结点和后序结点；

中序线索遍历

- 概念：
 - 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一次二叉树；
 - 从根节点出发，重新进行一次中序遍历；
 - 指针 **q** 记录当前访问的结点；
 - 指针 **pre** 记录上一个被访问的结点
- 标志域：
 - $ltag = \begin{cases} 0, & lchild \text{域指示结点的左孩子} \\ 1, & lchild \text{域指示结点的前驱} \end{cases}$
 - $rtag = \begin{cases} 0, & rchild \text{域指示结点的右孩子} \\ 1, & rchild \text{域指示结点的后继} \end{cases}$
- 判断：
 - 1. 当 $q == p$ 时，pre 为前驱；
 - 2. 当 $pre == p$ 时，q 为后继；

中序线索遍历序列

- 中序遍历序列：
 - 按照中序遍历得到的序列顺序，依次建立每个节点的前驱以及后驱序列；
 - D $\leftrightarrow$ G $\leftrightarrow$ B $\leftrightarrow$ E $\leftrightarrow$ A $\leftrightarrow$ F $\leftrightarrow$ C
- 存储图示：



代码实现

- 线索二叉树结点

```
//线索二叉树结点
typedef struct ThreadNode{
    ElemType data;
    struct ThreadNode *lchild,*rchild;
    int ltag,rtag;    //左，右线索标志
}ThreadNode,*ThreadTree;
```

- 中序遍历对二叉树线索化的递归算法

```
//中序遍历对二叉树线索化的递归算法：
void InTread(ThreadTree &p,ThreadTree &pre){
    if(p!=NULL){
        InTread(p->lchild,pre); //递归，先所化左子树
        if(p->lchild==NULL){    //左子树为空，建立前驱线索
            p->lchild = pre;
            p->ltag=1;
        }
        if(pre!=NULL&&pre->rchild==NULL){
            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
        InTread(p->rchild,pre); //递归，线索化右子树
    }//if(p!=NULL)
}
```

- 中序遍历建立中序线索二叉树主过程算法

```
//中序遍历建立中序线索二叉树主过程算法：
void CreateInTread(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;
}
}

```

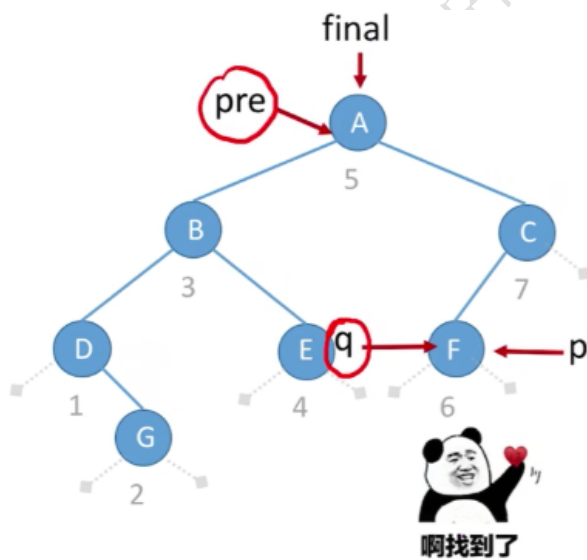
17.4.2 先序与后序线索二叉树

17.5 二叉树的线索化

17.5.1 中序遍历

老方法：通过中序遍历找到节点的前驱节点

- 概念：
 - 定义一个 pre 指针和 q 指针，q 指针递归获得树当中的节点，往下递归一层后，先进行一次判断，判断当前 p 和 q 节点是否一致；
 - 如果不一致，则将当前 pre 节点指向当前 q 节点；
 - 如果一致，则表示找到目标节点。此时不要再将 pre 节点指向当前 q 节点，因为此时 pre 节点已经是 p 节点的前驱节点，如图所示；
- 核心：
 - 多用一个遍历，存储上一步结果；
- 图示：



- 中序遍历序列：D G B E A F C

代码实现

```

//中序遍历
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
    }
}

```

```

}
}

//访问结点q
void visit(BiTNode *q){
    if(q==p)        //当前访问结点刚好是结点p
        final = pre; //找到p的前驱
    else
        pre = q;     //pre指向当前访问的结点
}

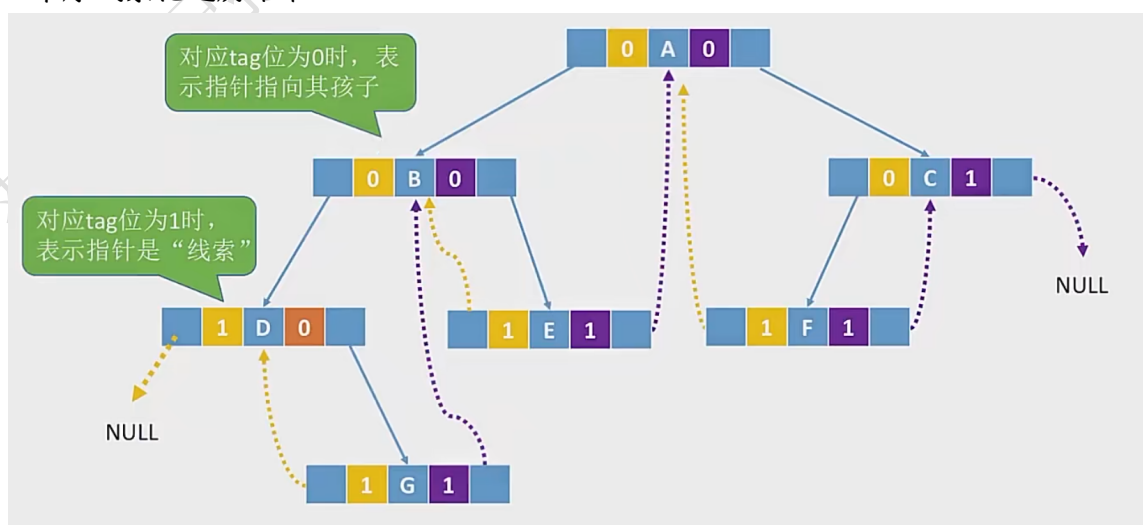
//辅助全局变量，用于查找结点p的前驱
BiTNode *p;          //p指向目标结点
BiTNode *pre = NULL; //指向当前访问结点的前驱
BiTNode *final = NULL; //用于记录最终结果

```

17.5.2 中序线索化

基本概念

- 概念：
 - 1. 中序/先序/后序遍历算法的改造，当访问一个结点时，连接该结点与前驱结点的线索信息；
 - 2. 用一个指针 pre 记录当前访问结点的前驱结点；
- 注意：
 - 1. 最后一个结点的 rchild、rtag 的处理
 - 2. 先序线索化中，注意处理爱滴魔力转圈圈问题，当 ltag== 0 时，才能对左子树先序线索化；
- 图示：
 - 节点：中序线索化遍历结果



代码实现

```

//线索二叉树结点
typedef struct ThreadNode{

```

```

ElemType data;
struct ThreadNode *lchild,*rchild;
int ltag,rtag;    //左,右线索标志
}ThreadNode,*ThreadTree;

```

```

//访问根节点
void visit(ThreadNode *q){
    if(q->lchild == NULL){ //左子树为空,建立前驱线索
        q->lchild = pre;
        q->ltag = 1;
    }
    if(pre != NULL&&pre->rchild==NULL){
        pre->rchild=q; //建立前驱结点的后继线索
        pre->rtag=1;
    }
    pre=q;
}

//全局变量pre 指向当前访问结点的前驱
ThreadNode *pre = NULL;

```

17.5.3 先序遍历线索化

代码实现

```

//先序遍历二叉树,一边遍历一边线索化
void PreThread(ThreadTree T){
    if(T!=NULL){
        visit(T);    //先处理根结点
        PreThread(T->lchild);
        PreThread(T->rchild);
    }
}

//先序遍历二叉树,一边遍历一边线索化
void PreThread(ThreadTree T){
    if(T!=NULL){
        visit(T);    //先处理根结点
        if(T->ltag==0) //lchild不是前驱线索
            //对左指针指向的树线索化
            PreThread(T->lchild);
        PreThread(T->rchild);
    }
}

```

17.5.4 后序遍历线索化

代码实现

```
//后序线索化
void PostThread(ThreadTree p, ThreadTree &pre){
    if(p!=NULL){
        PostThread(p->lchild, pre); //递归，线索化左子树
        PostThread(p->rchild, pre); //递归，线索化右子树
        if(p->lchild == NULL){ //左子树为空，建立前驱线索
            pre->lchild=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
    } //if(p!=NULL)
}
```

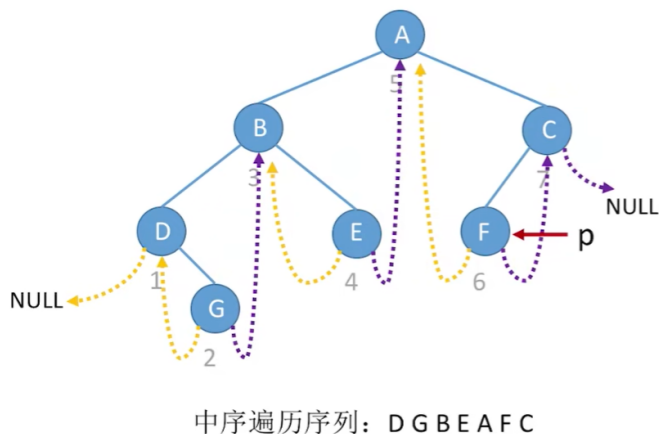
```
//后序线索化二叉树T
void CreatePostThread(ThreadTree T){
    ThreadTree pre = NULL;
    if(T!=NULL){ //非空二叉树，线索化
        PostThread(T.pre); //线索化二叉树
        if(pre->rchild==NULL) //处理遍历的最后一个结点
            pre->rtag=1;
    }
}
```

17.6 线索二叉树找前驱后继

17.6.1 中序线索二叉树

中序遍历找中序后继

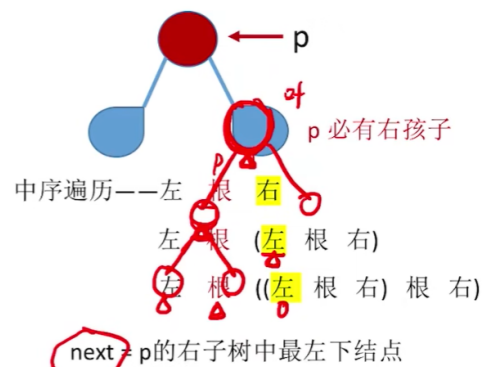
- 概念：
 - 如果 `p->rtag==1`，则当前后继节点即为 `next = p->rchild`；
 - 如果 `p->rtag==0`，则当前节点有右节点，并且在右子树的最左下节点为其后继节点；
- 代码：
 - `ThreadNode *Nextnode(ThreadNode *p) { //右子树中最左下结点 if (p->rtag==0) return Firstnode (p->rchild); else return p->rchild; }`
- 图示：



在中序线索二叉树中找到指定结点*p的中序后继 next

①若 $p \rightarrow rtag == 1$, 则 $next = p \rightarrow rchild$

②若 $p \rightarrow rtag == 0$



代码实现

//找到以P为根的子树中，第一个被中序遍历的结点

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

//在中序线索二叉树中找到结点p的后继结点

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

代码：使用中序线索二叉树进行中序遍历 -> 使用每个节点的后继节点，进行遍历 -> 空间复杂度为 $O(1)$

//对中序线索二叉树进行中序遍历(利用线索实现的非递归算法)

```
void Inorder(ThreadNode *T){
    for(ThreadNode *p=Firstnode(T);p!=NULL;p=Nextnode(p))
        visit(p);
}
//空间复杂度 $O(1)$ :因为不需要递归调用函数
```

//找到以P为根的子树中，最后一个被中序遍历的结点

```
ThreadNode *Lastnode(ThreadNode *p){
    //循环找到最右下结点(不一定是叶结点)
    while(p->rtag==0)
```

```

    p=p->rchild;
    return p;
}

```

代码：找前驱节点

```

//在中序线索二叉树中找到结点p的前驱结点
ThreadNode *Prenode(ThreadNode *p){
    //左子树中最右下结点
    if(p->ltag==0)
        return Lastnode(p->lchild);
    else
        return p->lchild; //ltag==1直接返回前驱线索
}

```

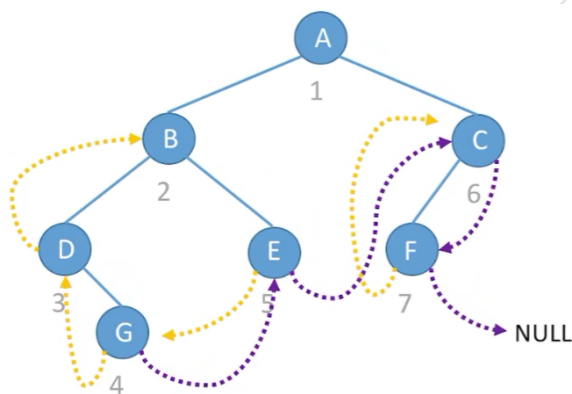
```

//对中序线索二叉树进行逆向中序遍历
void RevInorder(ThreadNode *T){
    for(ThreadNode *p=Lastnode(T);p!=NULL;p=Prenode(p))
        visit(p);
}

```

17.6.2 先序线索二叉树

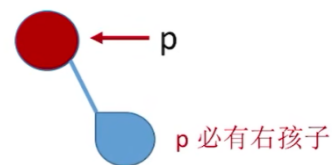
找先序后继



先序遍历序列：A B D G E C F

在先序线索二叉树中找到指定结点*p的先序后继 next

- ①若 $p \rightarrow rtag == 1$ ，则 $next = p \rightarrow rchild$
- ②若 $p \rightarrow rtag == 0$



若p有左孩子，则先序后继为左孩子

先序遍历——根 左 右
根 (根 左 右) 右

假设有左孩子

若p没有左孩子，则先序后继为右孩子

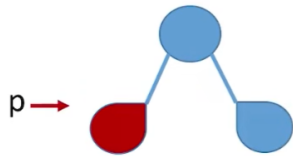
先序遍历——根 右
根 (根 左 右)

假设没有左孩子

先序线索二叉树找到先序前驱

- 注意：如果 p 是根节点，则 p 没有先序前驱

①如果能找到 p 的父节点，
且 p 是左孩子

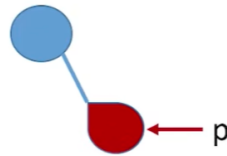


先序遍历——根 左 右

根 (根 左 右) 右

p 的父节点即为其前驱

②如果能找到 p 的父节点，且
 p 是右孩子，其左兄弟为空

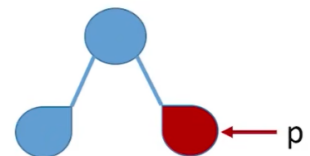


先序遍历——根 右

根 (根 左 右)

p 的父节点即为其前驱

③如果能找到 p 的父节点，且
 p 是右孩子，其左兄弟非空



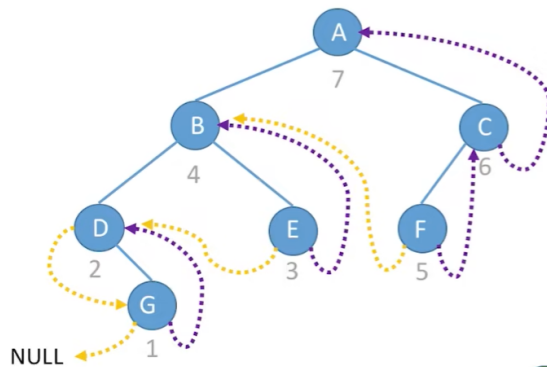
根 左 右

p 的前驱为左兄弟子树中最后一个被先序遍历的结点

17.6.3 后序线索二叉树

找后序前驱

• 图示：

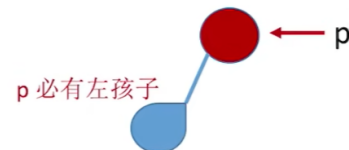


后序遍历序列：G D E B F C A

在后序线索二叉树中找到指定结点 $*p$ 的后序前驱 pre

①若 $p \rightarrow ltag == 1$ ，则 $pre = p \rightarrow lchild$

②若 $p \rightarrow ltag == 0$



p 必有左孩子

若 p 有右孩子，
则后序前驱为右孩子

后序遍历——左 右 根

假设没有右孩子

左 (左 右 根) 根

若 p 没有右孩子，
则后序前驱为左孩子

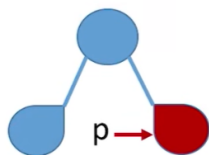
后序遍历——左 根

(左 右 根) 根

找后续后继(三叉链表)

• 图示：

①如果能找到 p 的父节点，
且 p 是右孩子

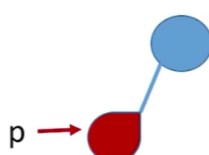


后序遍历——左 右 根

左 (左 右 根) 根

p 的父节点即为其后继

②如果能找到 p 的父节点，且
 p 是左孩子，其右兄弟为空

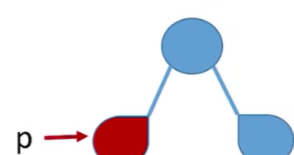


后序遍历——左 根

(左 右 根) 根

p 的父节点即为其后继

③如果能找到 p 的父节点，且
 p 是左孩子，其右兄弟非空



左 右 根

p 的后继为右兄弟子树中第一个被后序遍历的结点

④如果 p 是根节点，则 p 没有后序后继

总结

- 对于先序以及后序：除非用三叉链表，或者用土办法从根开始遍历寻找，不然就无法直接（可以从头开始找）找前驱或者后继节点；

	中序线索二叉树	先序线索二叉树	后序线索二叉树
找前驱	✓	✗	✓
找后继	✓	✓	✗

获取笔记持续更新可加微信：WandD2277