



Data Structure & Algorithm Analysis

Trees & Binary Trees

Zibin Zheng (郑子彬)

School of Data and Computer Science , SYSU

<http://www.inpluslab.com>

课程主页: <http://inpluslab.sysu.edu.cn/dsa2016/>

Traversal of Binary Trees

1. 先序遍历(DLR)

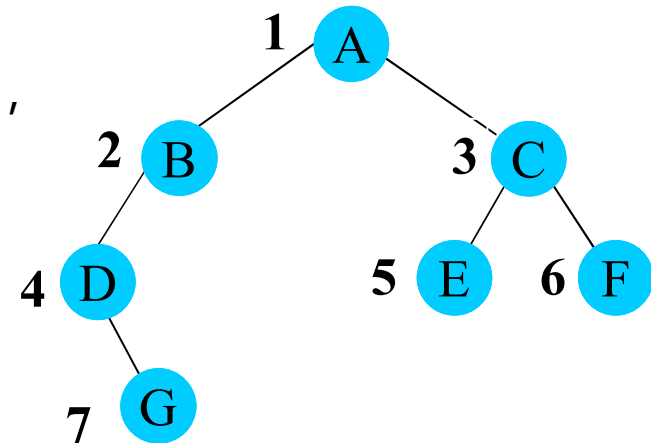
- 递归过程为：若二叉树为空，遍历结束。否则，

- (1)访问根结点；
- (2)先序遍历根结点的左子树；
- (3)先序遍历根结点的右子树。

- 先序遍历二叉树的递归算法：

```
void PreOrder(BiTree bt)           // 先序遍历二叉树bt
{ if (bt==NULL) return; // 递归调用的结束条件
  Visite(bt->data);                // 访问结点的数据域
  PreOrder(bt->lchild); // 先序递归遍历bt的左子树
  PreOrder(bt->rchild); // 先序递归遍历bt的右子树
}
```

对于上图所示的二叉树，按先序遍历所得到的结点序列为：**ABDGCEF**



Traversal of Binary Trees

2. 中序遍历(LDR)

- 递归过程为：若二叉树为空，遍历结束。否则，

(1)中序遍历根结点的左子树；

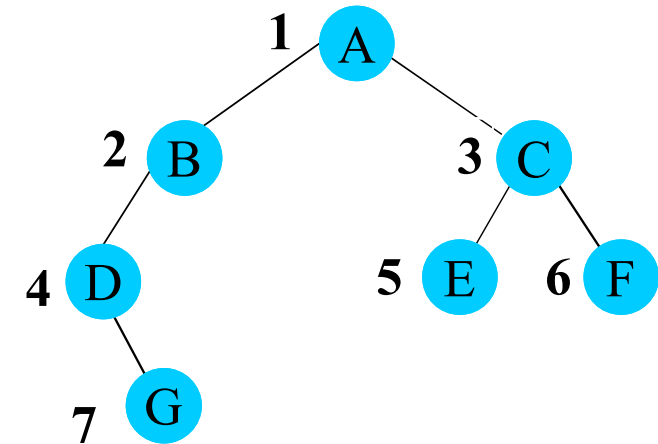
(2)访问根结点；

(3)中序遍历根结点的右子树。

- 中序遍历二叉树的递归算法：

```
void InOrder(BiTree bt)           // 中序遍历二叉树bt
{ if (bt==NULL) return;           // 递归调用的结束条件
  InOrder(bt->lchild);              // 中序递归遍历bt的左子树
  Visite(bt->data);                 // 访问结点的数据域
  InOrder(bt->rchild);              // 中序递归遍历bt的右子树
}
```

对于上图所示的二叉树，按中序遍历所得到的结点序列为：**DGBAECF**



Traversal of Binary Trees

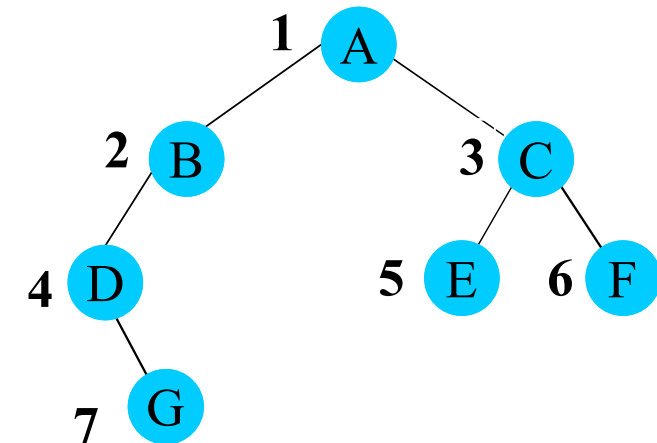
3. 后序遍历(LRD)

- 递归过程为：若二叉树为空，遍历结束。否则，

- (1)后序遍历根结点的左子树；
- (2)后序遍历根结点的右子树。
- (3)访问根结点；

- 后序遍历二叉树的递归算法：

```
void PostOrder(BiTree bt)           // 后序遍历二叉树bt
{ if (bt==NULL) return;             // 递归调用的结束条件
  PostOrder(bt->lchild);              // 后序递归遍历bt的左子树
  PostOrder(bt->rchild);              // 后序递归遍历bt的右子树
  Visite(bt->data);                  // 访问结点的数据域
}
```

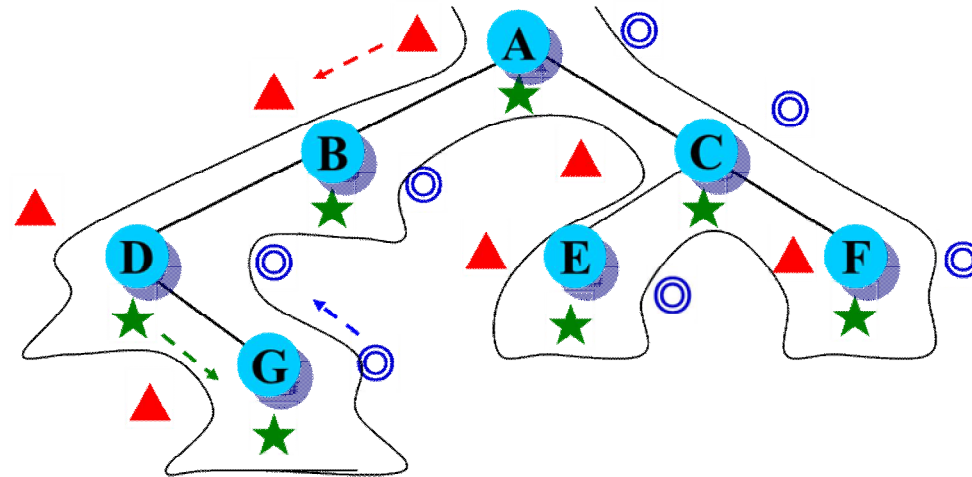


对于上图所示的二叉树，按先序遍历所得到的结点序列为：**GDBEFCA**

Traversal of Binary Trees

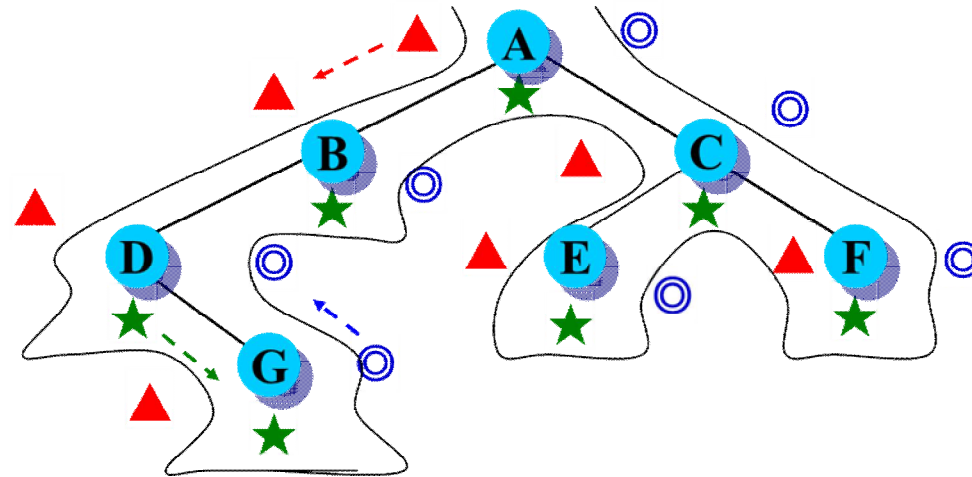
- 二叉树遍历的**非递归实现**
- 前面给出的二叉树先序、中序和后序三种遍历算法都是递归算法。当给出二叉树的链式存储结构以后，用具有递归功能的程序设计语言很方便就能实现上述算法。
- 然而，并非所有程序设计语言都允许递归；另一方面，递归程序虽然简洁，但可读性一般不好，执行效率也不高。
- 因此，就存在如何把一个递归算法转化为非递归算法的问题。解决这个问题的方法可以通过对三种遍历方法的实质过程的分析得到。

Traversal of Binary Trees



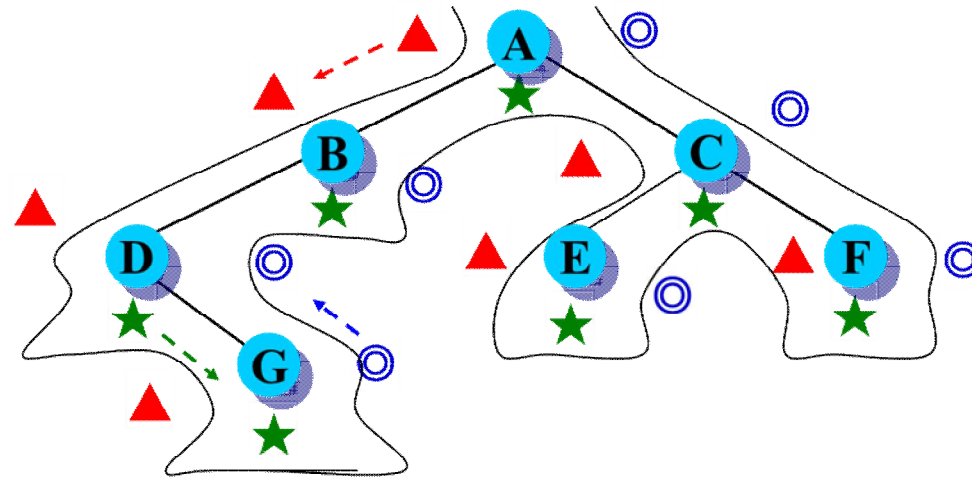
对二叉树进行先序、中序和后序遍历都是从根结点A开始的，且在遍历过程中经过结点的路线是一样的，只是访问的时机不同而已。如上图所示，从根结点左外侧开始，由根结点右外侧结束的曲线，为遍历图的路线。沿着该路线按▲标记的结点读得的序列为**先序序列**，按★标记读得的序列为**中序序列**，按⊙标记读得的序列为**后序序列**。

Traversal of Binary Trees



- 遍历路线正是从根结点开始沿左子树深入下去，当深入到最左端，无法再深入下去时，则返回，再逐一进入刚才深入时遇到结点的右子树，再进行如此的深入和返回，直到最后从根结点的右子树返回到根结点为止。
- **先序遍历**是在深入时遇到结点就访问，**中序遍历**是在从左子树返回时遇到结点访问，**后序遍历**是在从右子树返回时遇到结点访问。

Traversal of Binary Trees



遍历图的路线示意图

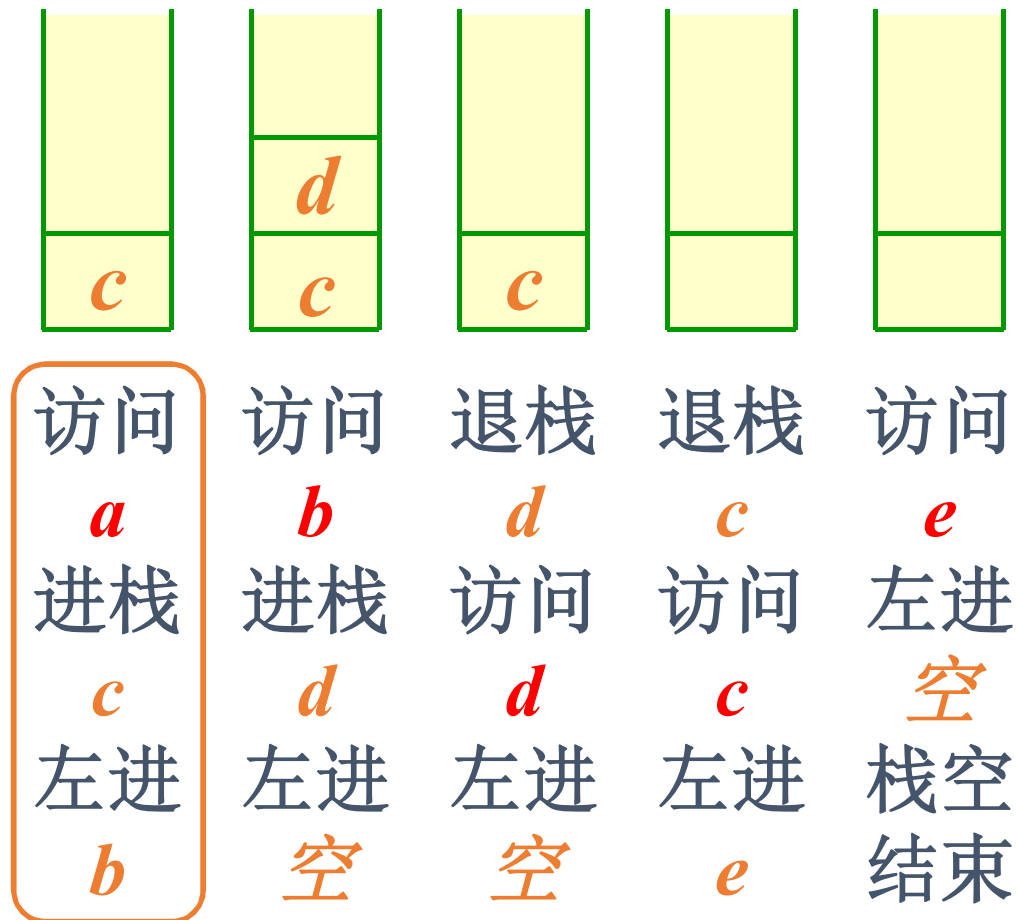
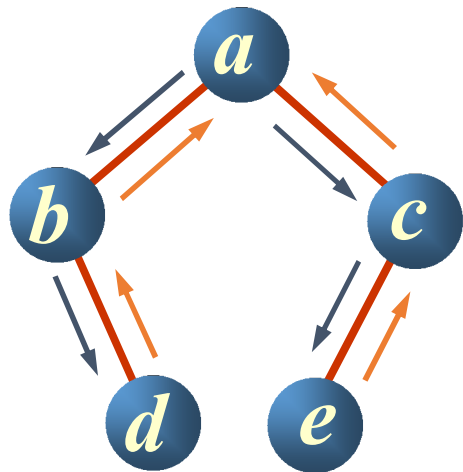
- 在这一过程中，返回结点的顺序与深入结点的顺序相反，即后深入先返回，正好符合栈结构后进先出的特点。因此，可以用栈来帮助实现这一遍历路线。其过程如下。

Traversal of Binary Trees

- 在沿左子树深入时，深入一个结点，入栈一个结点。
- 若为**先序遍历**，则在入栈之前访问之，当沿左分支深入不下去时，则返回，即从堆栈中弹出前面压入的结点；
- 若为**中序遍历**，则此时访问该结点，然后从该结点的右子树继续深入；
- 若为**后序遍历**，则将此结点再次入栈，然后从该结点的右子树继续深入，与前面类同，仍能深入一个结点入栈一个结点，深入不下去再返回，直到第二次从栈里弹出该结点，才访问之。

Traversal of Binary Trees

- 利用栈的先序遍历非递归算法



Traversal of Binary Trees

- 利用栈的前序遍历非递归算法

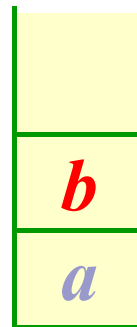
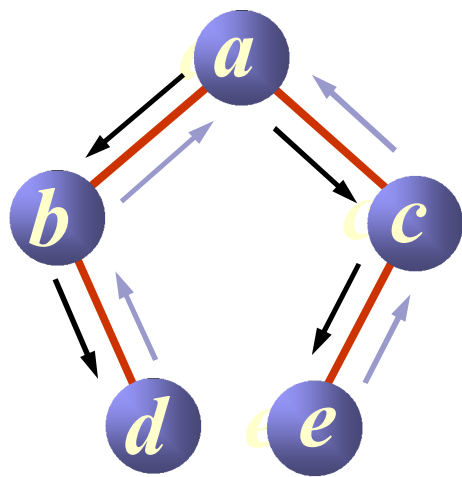
```
template <class T>
void BinaryTree<T>::PreOrder(void (*visit)(BinTreeNode<T> *t))
{
    stack<BinTreeNode<T>*> S;
    BinTreeNode<T> *p = root;
    S.Push (NULL);
    while (p != NULL) {
        {   visit(p);           //访问结点
            if (p->rightChild != NULL)
                S.Push (p->rightChild); //预留右指针在栈中
            if (p->leftChild != NULL)
                p = p->leftChild;       //进左子树
            else S.Pop(p);               //左子树为空
        }
    }
}
```

Traversal of Binary Trees

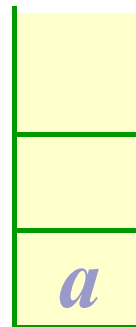
- 在沿左子树深入时，深入一个结点，入栈一个结点。
- 若为**先序遍历**，则在入栈之前访问之，当沿左分支深入不下去时，则返回，即从堆栈中弹出前面压入的结点；
- 若为**中序遍历**，则此时访问该结点，然后从该结点的右子树继续深入；
- 若为**后序遍历**，则将此结点再次入栈，然后从该结点的右子树继续深入，与前面类同，仍能深入一个结点入栈一个结点，深入不下去再返回，直到第二次从栈里弹出该结点，才访问之。

Traversal of Binary Trees

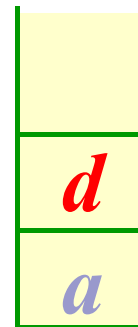
- 利用栈的中序遍历非递归算法



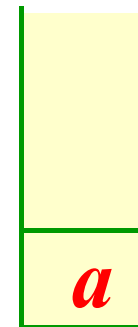
左空



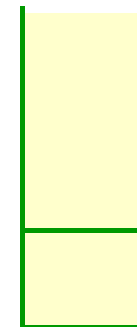
退栈
访问



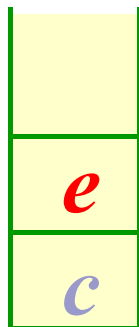
左空



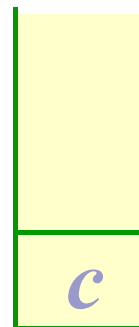
退栈
访问



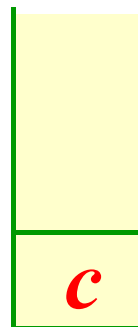
退栈
访问



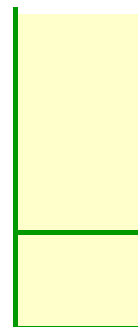
左空



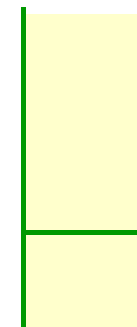
退栈访问



右空



退栈访问



栈空结束

Traversal of Binary Trees

- 利用栈的中序遍历非递归算法

```
template <class T>
void BinaryTree<T>::InOrder(void (*visit) (BinTreeNode<T> *t))
{
    stack<BinTreeNode<T>*> S;
    BinTreeNode<T> *p = root;
    do
    {
        while (p != NULL) //遍历指针向左下移动
        {
            S.Push (p);           //该子树沿途结点进栈
            p = p->leftChild;
        }
        if (!S.IsEmpty()) //栈不空时退栈
        {
            S.Pop (p); visit (p); //退栈, 访问
            p = p->rightChild;    //遍历指针进到右子女
        }
    } while (p != NULL || !S.IsEmpty ());
}
```

Traversal of Binary Trees

- 在沿左子树深入时，深入一个结点，入栈一个结点。
- 若为**先序遍历**，则在入栈之前访问之，当沿左分支深入不下去时，则返回，即从堆栈中弹出前面压入的结点；
- 若为**中序遍历**，则此时访问该结点，然后从该结点的右子树继续深入；
- 若为**后序遍历**，则将此结点再次入栈，然后从该结点的右子树继续深入，与前面类同，仍能深入一个结点入栈一个结点，深入不下去再返回，直到第二次从栈里弹出该结点，才访问之。

Traversal of Binary Trees

- 利用栈的后序遍历非递归算法

- 在后序遍历过程中所用栈的结点定义

```
template <class T>
struct stkNode {
    BinTreeNode<T> *ptr;    //树结点指针
    enum tag {L, R};        //退栈标记
    stkNode (BinTreeNode<T> *N = NULL) :
        ptr(N), tag(L) {}    //构造函数
};
```

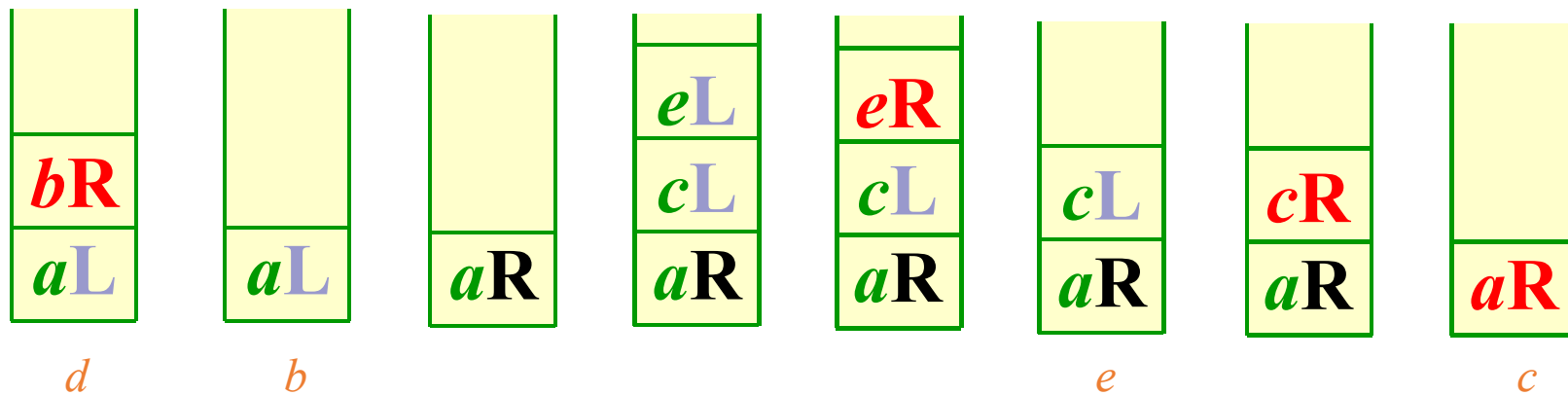
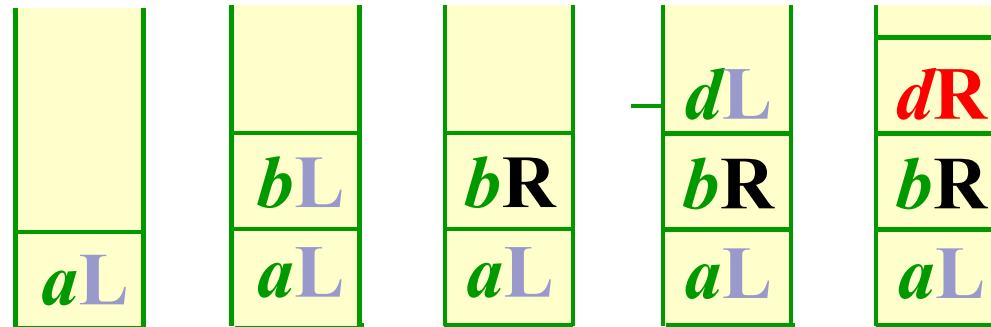
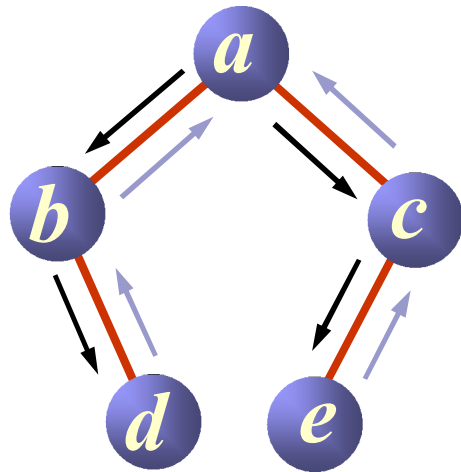
ptr	tag{L,R}
-----	----------

- tag = L, 表示从左子树退回还要遍历右子树;

- tag = R, 表示从右子树退回要访问根结点。

Traversal of Binary Trees

- 利用栈的后序遍历非递归算法



Traversal of Binary Trees

- 后序遍历的非递归算法

```
void BinaryTree<T>::PostOrder (void (*visit) (BinTreeNode<T> *t)
{   Stack< stkNode<T> > S;   stkNode <T> w;
    BinTreeNode<T> * p = root;           //p是遍历指针
    do {
        while (p != NULL) {
            w.ptr = p; w.tag = L; S.Push (w); p = p->leftChild; }
        int continuel = 1;               //继续循环标记, 用于R
        while (continuel && !S.IsEmpty ()) {
            S.Pop (w); p = w.ptr;
            switch (w.tag)                 //判断栈顶的tag标记
            {
                case L: w.tag = R; S.Push (w); continuel = 0;
                    p = p->rightChild; break;
                case R: visit (p); break;
            }
        }
    } while (!S.IsEmpty ());              //继续遍历其他结点
    cout << endl;
};
```

Restore the binary tree by the traversal

- 由遍历序列恢复二叉树
- 从前面讨论的二叉树的遍历知道，任意一棵二叉树结点的先序序列和中序序列都是唯一的。反过来，若已知结点的先序序列和中序序列，能否确定这棵二叉树呢？
- 这样确定的二叉树是否是唯一的呢？

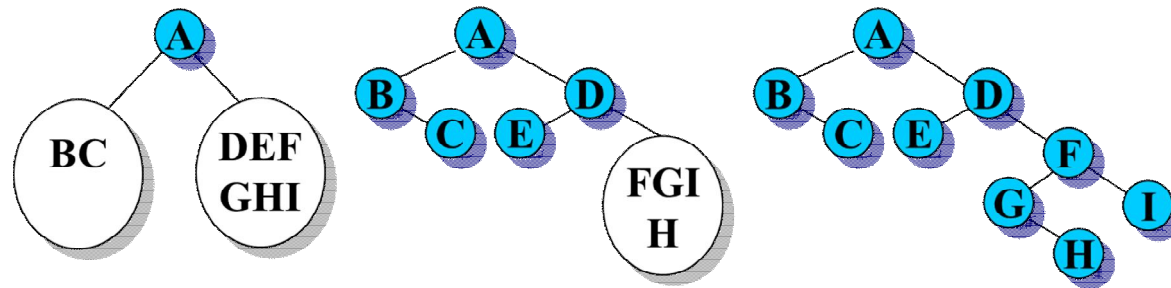
Restore the binary tree by the traversal

- **在先序序列中，第一个结点一定是二叉树的根结点。**另一方面，中序遍历是先遍历左子树，然后访问根结点，最后再遍历右子树。
- 根结点在中序序列中必然将中序序列分割成两个子序列，前一个子序列是根结点的左子树的中序序列，后一个子序列是根结点的右子树的中序序列。
- 在先序序列中找到对应的左子序列和右子序列。在先序序列中，左子序列的第一个结点是左子树的根结点，右子序列的第一个结点是右子树的根结点。这样就确定了二叉树的三个结点。
- 同时，左子树和右子树的根结点又可以分别把左子序列和右子序列划分成两个子序列，如此递归下去，当取尽先序序列中的结点时，便可以得到一棵二叉树。

Restore the binary tree by the traversal

已知一棵二叉树的先序与中序序列分别为**ABCDEFghi**和**BCAEDGHFI**，恢复该二叉树

1. 首先，由先序序列可知，结点A是二叉树的根结点。
2. 其次，根据中序序列，在A之前的所有结点都是根结点左子树的结点，在A之后的所有结点都是根结点右子树的结点，由此得到图6.12(a)所示的状态。
3. 然后，再对左子树进行分解，得知B是左子树的根结点，又从中序序列知道，B的左子树为空，B的右子树只有一个结点C。
4. 接着对A的右子树进行分解，得知A的右子树的根结点为D；而结点D把其余结点分成两部分，即左子树为E，右子树为F、G、H、I，如图6.12(b)所示。接下去的工作就是按上述原则对D的右子树继续分解下去，最后得到如图6.12(c)的整棵二叉树。



Restore the binary tree by the traversal

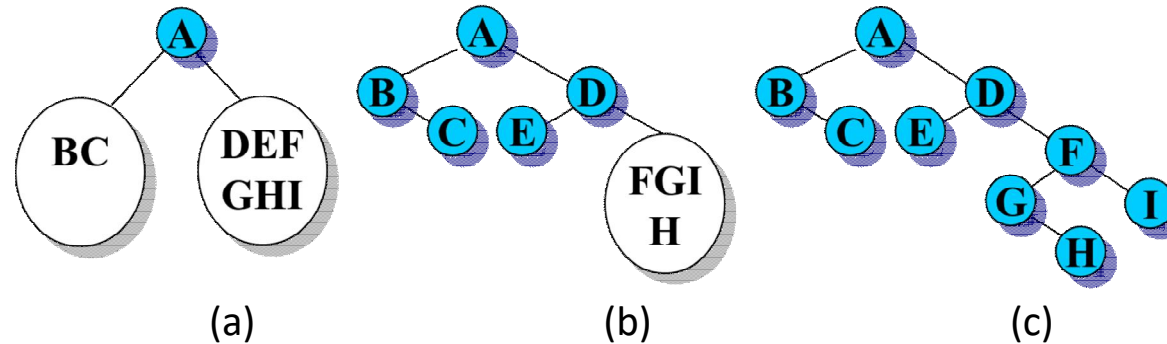


图6.12 一棵二叉树的恢复过程示意

- 这是一个递归过程，其思想是：
 - 先根据先序序列的第一个元素建立根结点；
 - 然后在中序序列中找到该元素，确定根结点的左、右子树的中序序列；
 - 再在先序序列中确定左、右子树的先序序列；
 - 最后由左子树的先序序列与中序序列建立左子树，由右子树的先序序列与中序序列建立右子树。

Restore the binary tree by the traversal

- 同理，由**二叉树的后序序列和中序序列也可唯一地确定一棵二叉树**。
- 因为，依据后序遍历和中序遍历的定义，后序序列的最后一个结点，就如同先序序列的第一个结点一样，可将中序序列分成两个子序列，分别为这个结点的左子树的中序序列和右子树的中序序列。
- 再拿出后序序列的倒数第二个结点，并继续分割中序序列，如此递归下去，当倒着取取尽后序序列中的结点时，便可以得到一棵二叉树。

Restore the binary tree by the traversal

【例】已知二叉树有如下的遍历结果

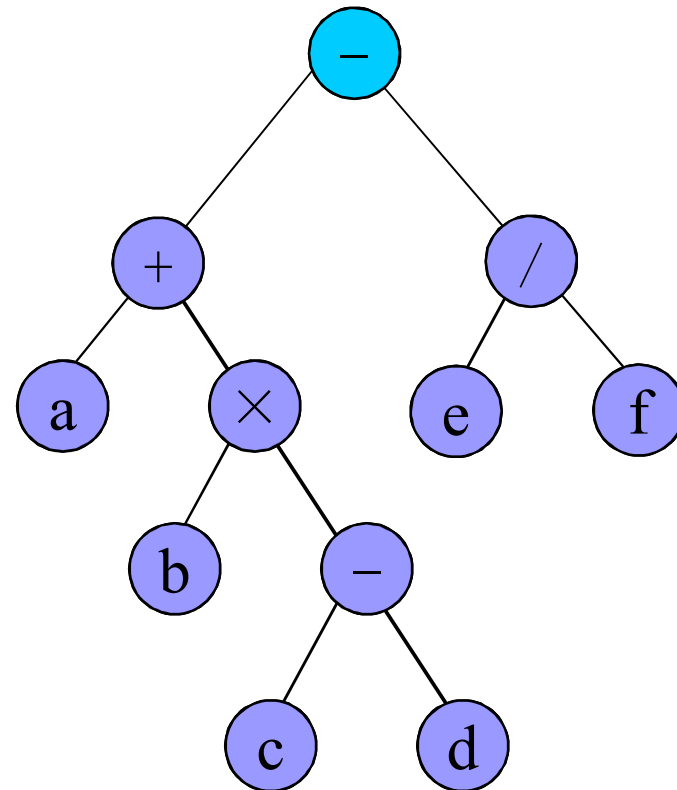
■ 按中序遍历，其中序序列为：

$a+b\times c-d-e/f$

■ 按后序遍历，其后序序列为：

$abcd-\times+ef/-$

请根据中序遍历和后序遍历结果重构二叉树。



Restore the binary tree by the traversal

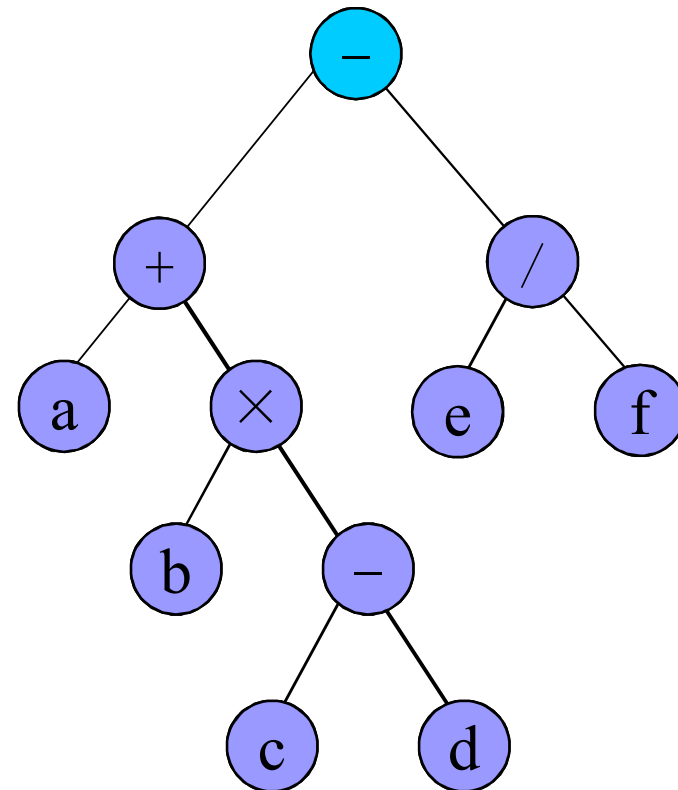
- 另外，如果只知道二叉树的先序序列和后序序列，则不能唯一地确定一棵二叉树。

■ 先序序列为：

$- + a \times b - cd / ef$

■ 后序序列为：

$abcd - \times + ef / -$



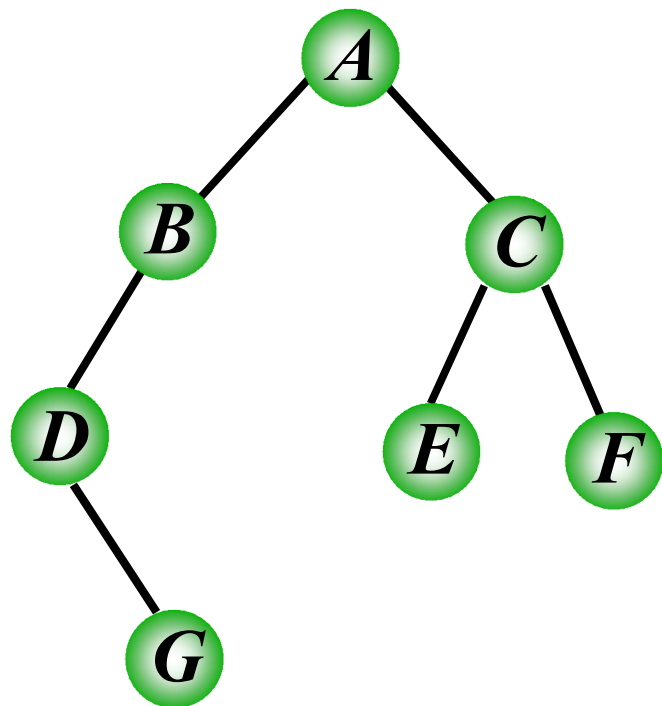
不用栈的二叉树遍历的非递归方法

- 前面介绍的二叉树的遍历算法可分为两类，
 - 一类是依据二叉树结构的递归性，采用递归调用的方式来实现；
 - 另一类则是通过堆栈或队列来辅助实现。
- 采用这两类方法对二叉树进行遍历时，递归调用和栈的使用都带来额外空间增加，递归调用的深度和栈的大小是动态变化的，都与二叉树的高度有关。因此，在最坏的情况下，即二叉树退化为单支树的情况下，递归的深度或栈需要的存储空间等于二叉树中的结点数。
- 还有一类二叉树的遍历算法，就是不用栈也不用递归来实现。利用具有 n 个结点的二叉树中的叶结点和一度结点的 $n + 1$ 个空指针域，来存放线索，然后在这种具有线索的二叉树上遍历时，就可不需要栈，也不需要递归了。

Threaded binary trees

线索链表

① 如何保存二叉树的某种遍历序列？



中序遍历序列: *D G B A F C F*

顺序
存储

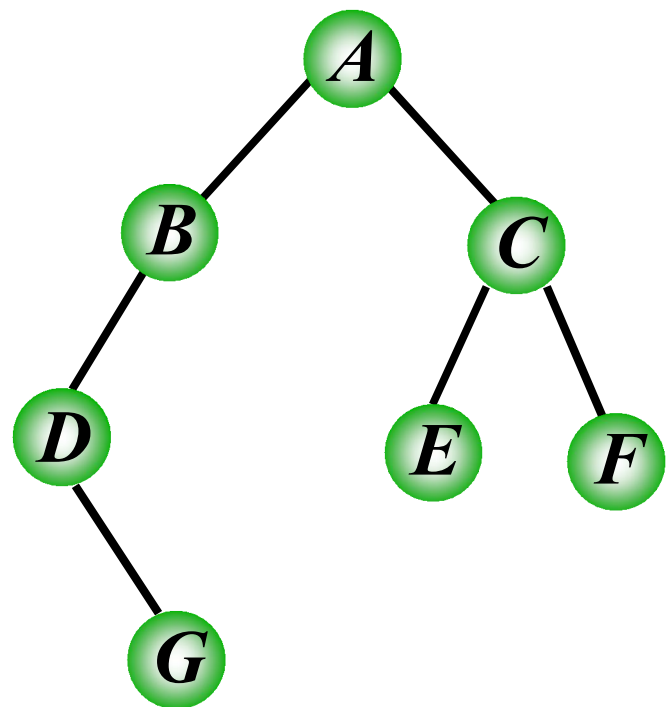
<i>D</i>	<i>G</i>	<i>B</i>	<i>A</i>	<i>F</i>	<i>C</i>	<i>F</i>
----------	----------	----------	----------	----------	----------	----------

① 如果二叉树不改变，如何保存？

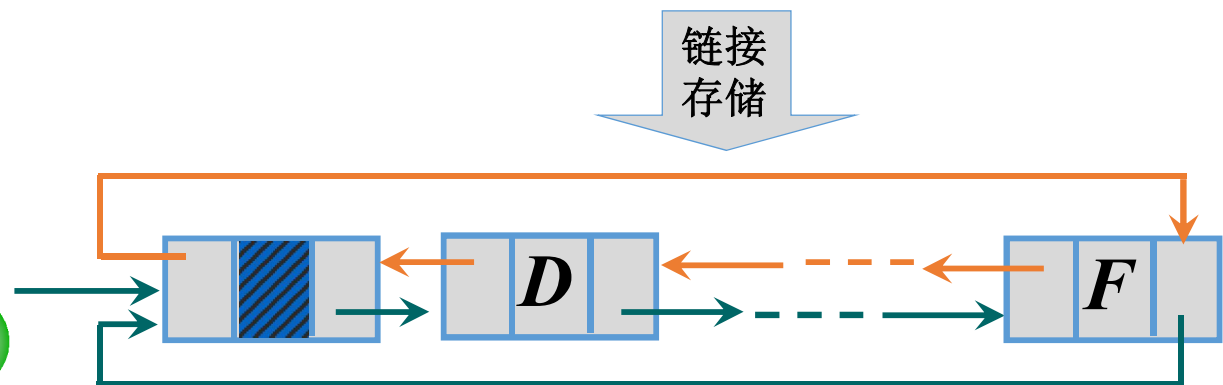
Threaded binary trees

线索链表

① 如何保存二叉树的某种遍历序列？



中序遍历序列: *D G B A F C F*

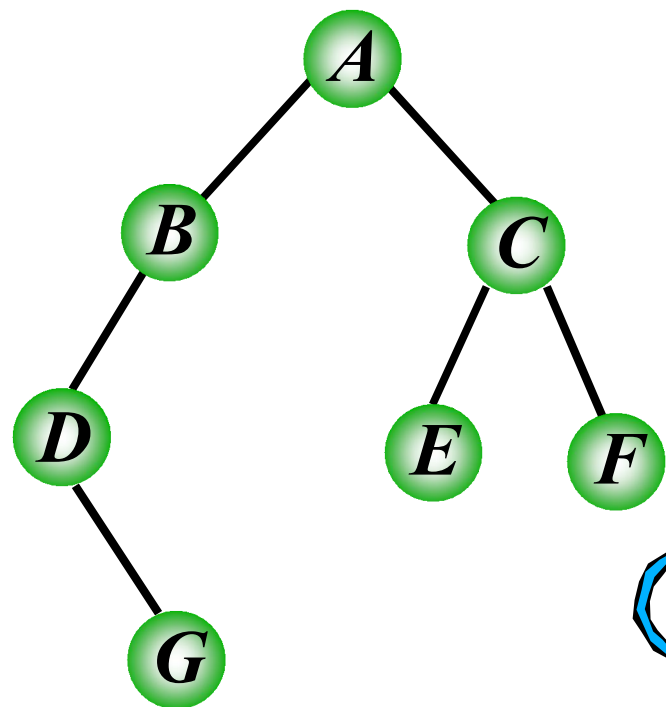


① 如果二叉树改变，如何保存？

Threaded binary trees

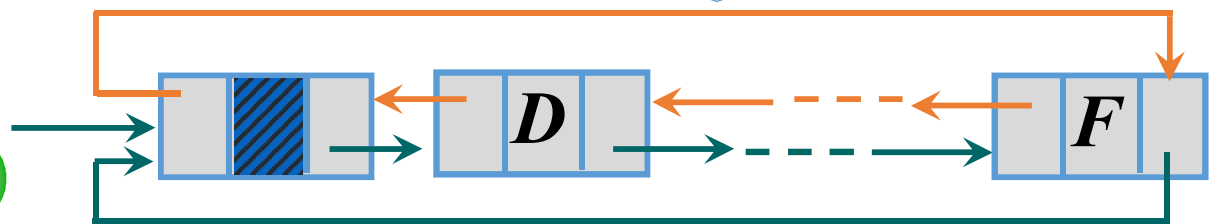
线索链表

① 如何保存二叉树的某种遍历序列？



中序遍历序列: *D G B A F C F*

链接
存储



① 如何将二叉链表与中序链表结合？

线索链表

① 如何保存二叉树的某种遍历序列？

将二叉链表中的空指针域指向其前驱结点和后继结点

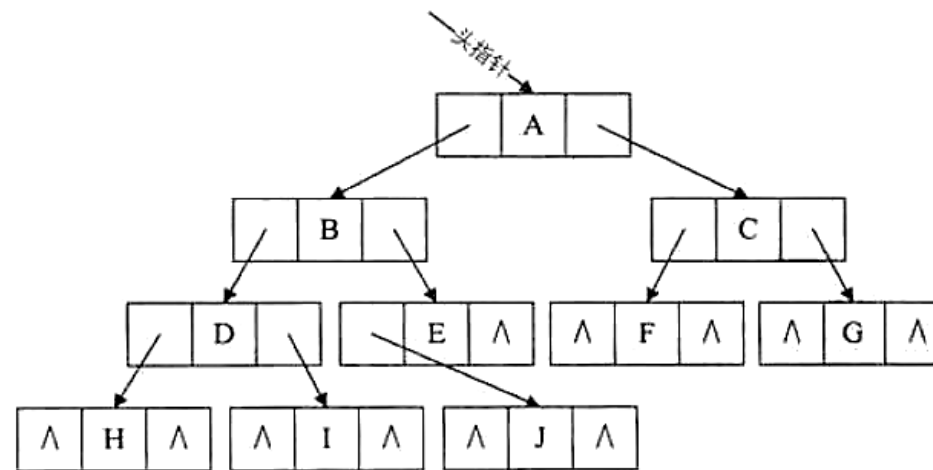
□ **线索**：将二叉链表中的空指针域指向前驱结点和后继结点的指针被称为线索；

□ **线索化**：使二叉链表中结点的空链域存放其前驱或后继信息的过程称为线索化；

□ **线索链表**：加上线索的二叉链表称为线索链表。

Threaded binary trees

- n 个结点的二叉链表，每个结点有两个指针域，一共 $2n$ 个指针域。 n 个结点的二叉树有 $n-1$ 条分支线数，存在 $2n-(n-1)=n+1$ 个空指针域。如下图有10个结点，而带有“ \wedge ”空指针域为11。这些空间不存储任何事物，浪费内存。
- 如下图进行中序遍历，得到HDIBJEAFCG序列。可知结点I的前驱是D，后继是B，结点F的前驱是A，后继是C。每次要知道某结点的前驱和后继结点时，都需要遍历一次。考虑在创建二叉树时，就记录结点的前驱和后继结点，从而节省时间。



Threaded binary trees

线索二叉树的定义

- 按照某种遍历方式对二叉树进行遍历，可以把二叉树中所有结点排列为一个线性序列。在该序列中，除第一个结点外，每个结点有且仅有一个直接前驱结点；除最后一个结点外，每个结点有且仅有一个直接后继结点。
- 但是，二叉树中每个结点在这个序列中的直接前驱结点和直接后继结点是什么，二叉树的存储结构中并没有反映出来，只能在对二叉树遍历的动态过程中得到这些信息。
- 为了保留结点在某种遍历序列中直接前驱和直接后继的位置信息，可以利用二叉树的二叉链表存储结构中的那些空指针域来指示。这些指向直接前驱结点和指向直接后继结点的指针被称为线索(thread)，加了线索的二叉树称为**线索二叉树**。
- 线索二叉树将为二叉树的遍历提供许多遍历方式。

Threaded binary trees

- 区别结点的指针域内存放的是指针还是线索？通常可以采用下面两种方法来实现。

(1) 为每个结点增设两个标志位域ltag和rtag，令：

ltag = {
0 lchild指向结点的左孩子
1 lchild指向结点的前驱结点

rtag = {
0 rchild指向结点的右孩子
1 rchild指向结点的后继结点

- 每个标志位令其只占一个bit，这样就只需增加很少的存储空间。这样结点的结构为：



- (2) 不改变结点结构，仅在作为线索的地址前加一个负号，即负的地址表示线索，正的地址表示指针。

Threaded binary trees

结点结构



```
enum flag {Child, Thread};  
template <class DataType>  
struct ThrNode  
{  
    DataType data;  
    ThrNode<DataType> *lchild, *rchild;  
    flag ltag, rtag;  
};
```

线索二叉树

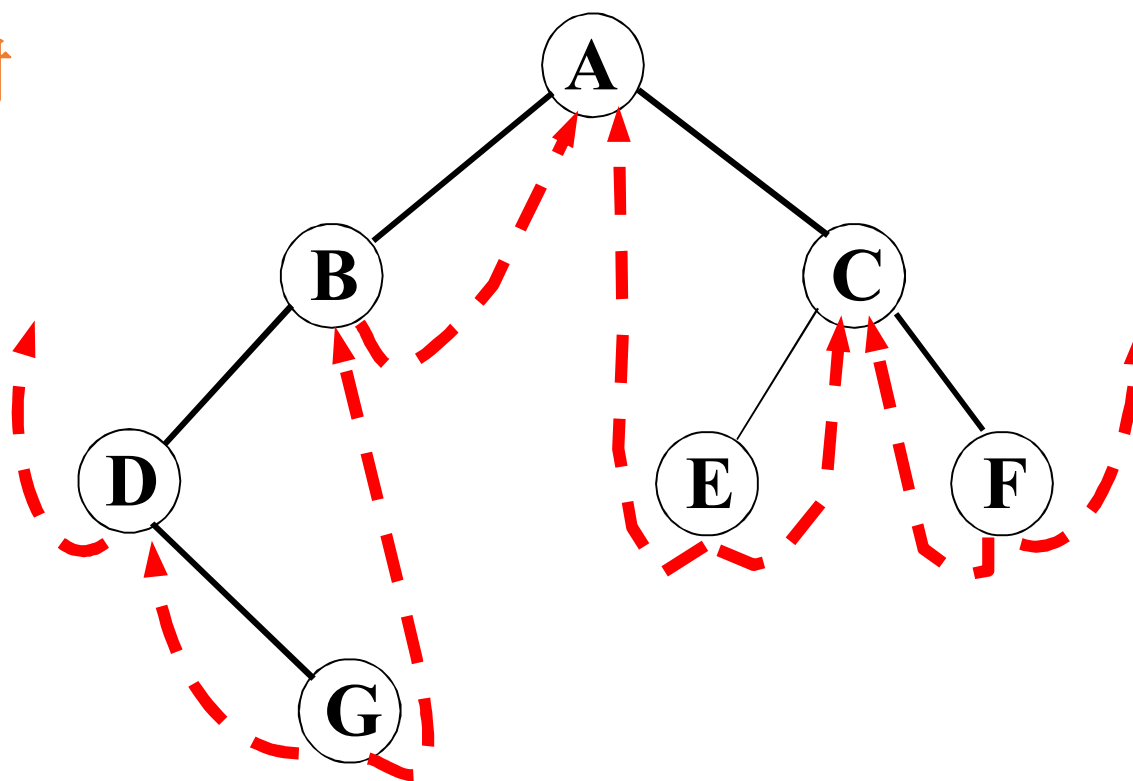
二叉树的遍历方式有4种，故有4种意义下的前驱和后继，相应的有4种线索二叉树：

- (1) 前序线索二叉树
- (2) 中序线索二叉树
- (3) 后序线索二叉树
- (4) 层序线索二叉树

Threaded binary trees

线索二叉树

中序线索二叉树




中序序列: ***D G B A E C F***

中序线索链表的建立——构造函数

分析：建立线索链表，实质上就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该二叉树时才能得到。

建立二叉链表



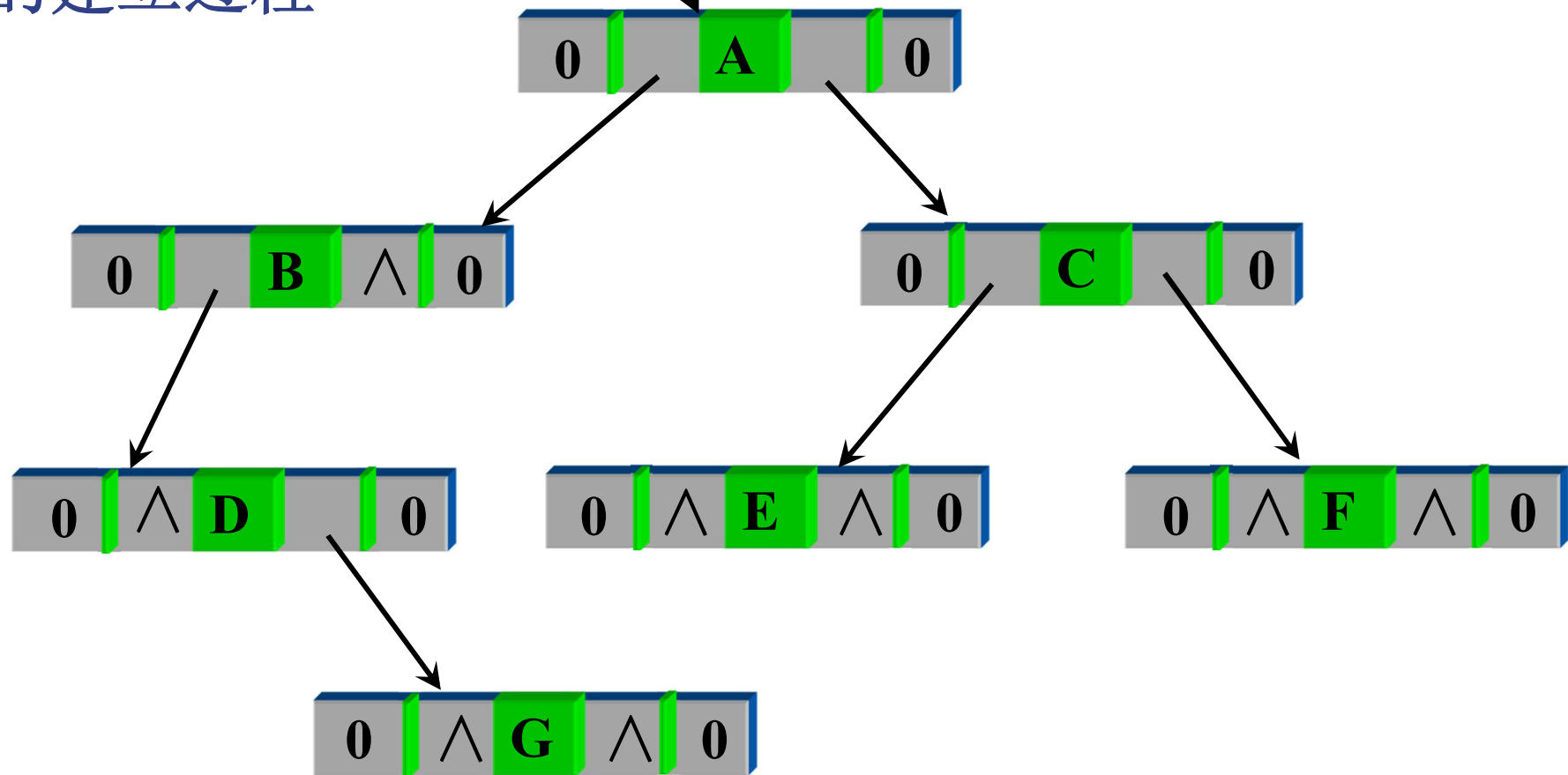
遍历二叉树，将空指针改为线索

Threaded binary trees

中序线索链表的
建立过程

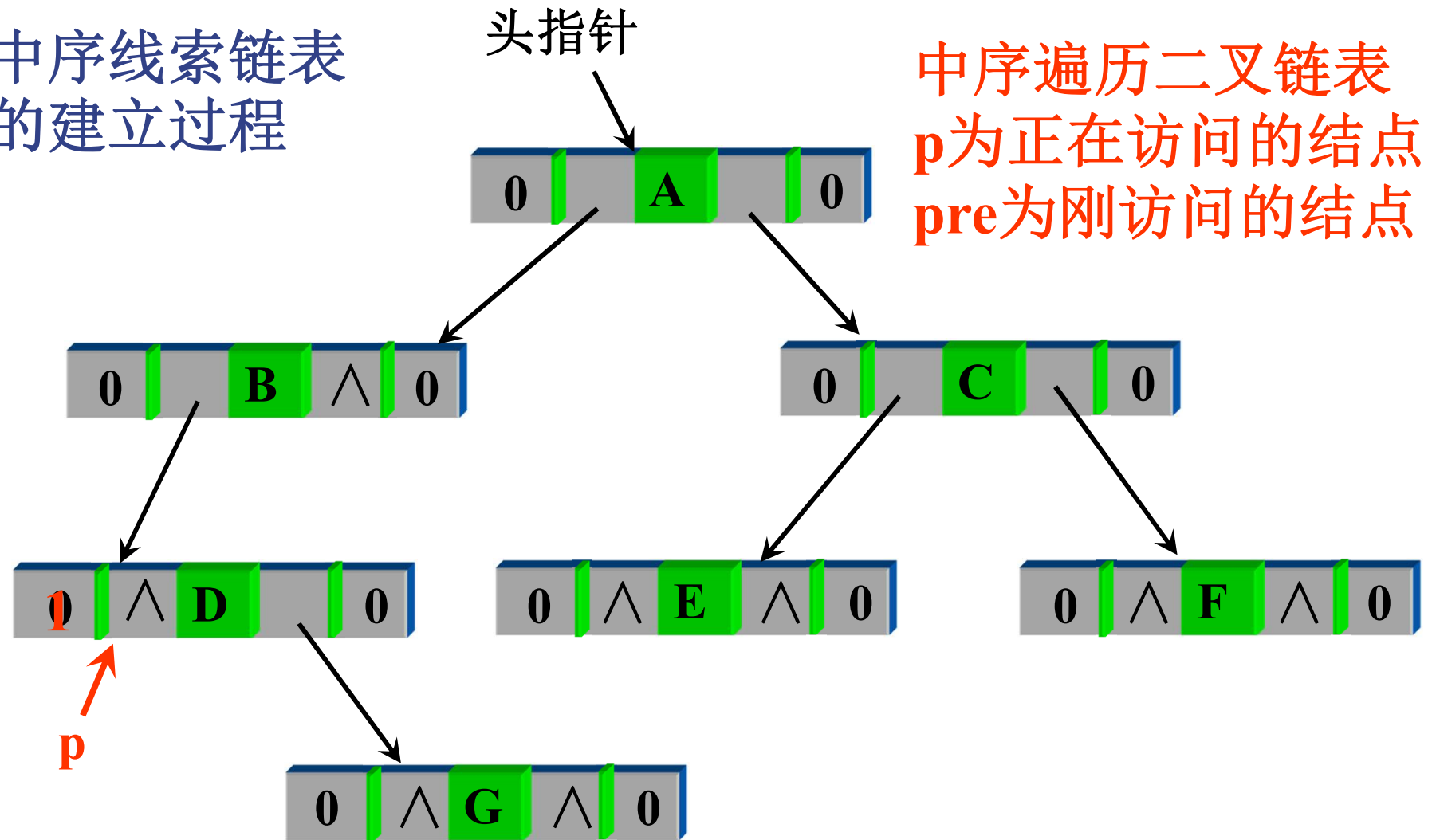
头指针

已经建立起二叉链表



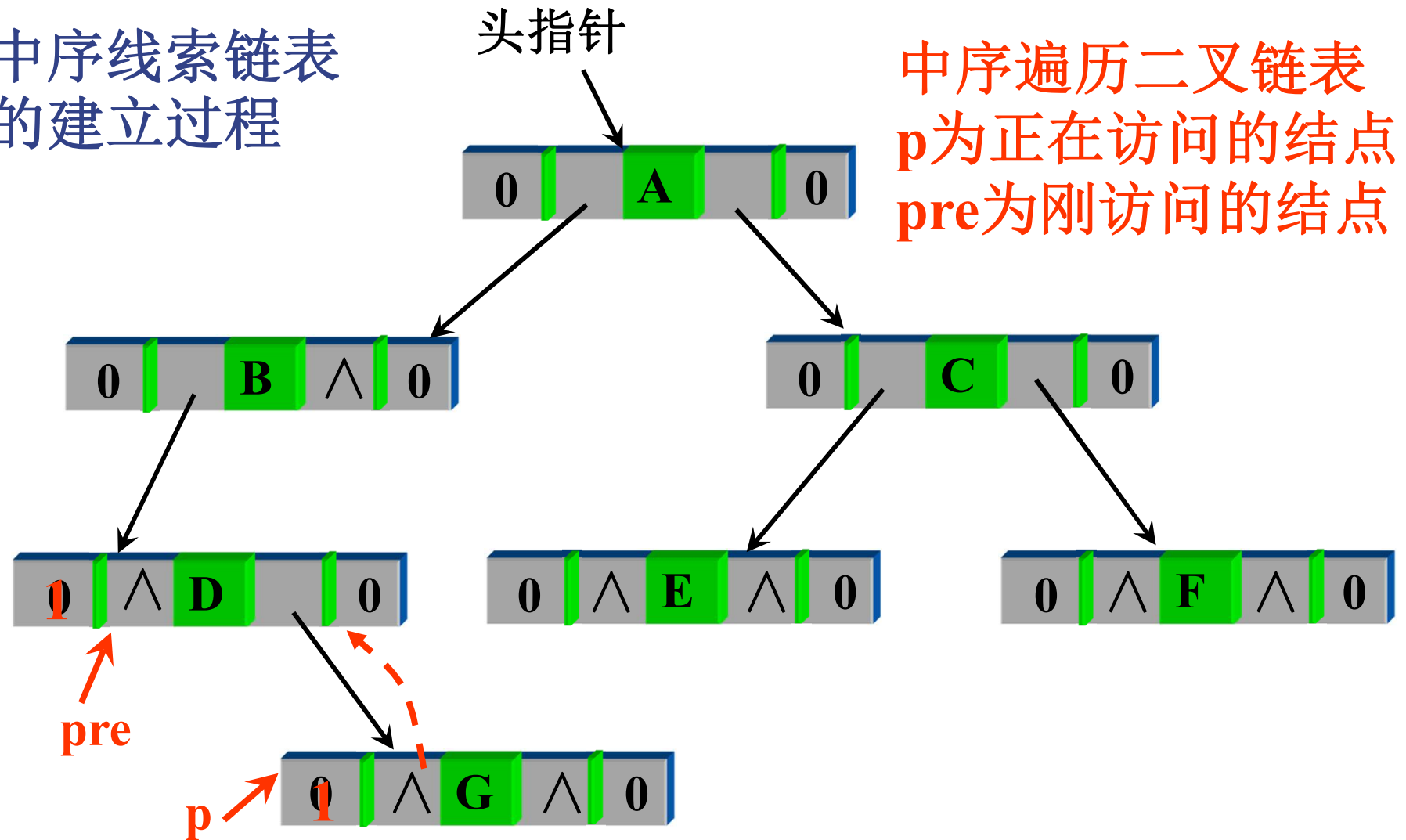
Threaded binary trees

中序线索链表的
建立过程



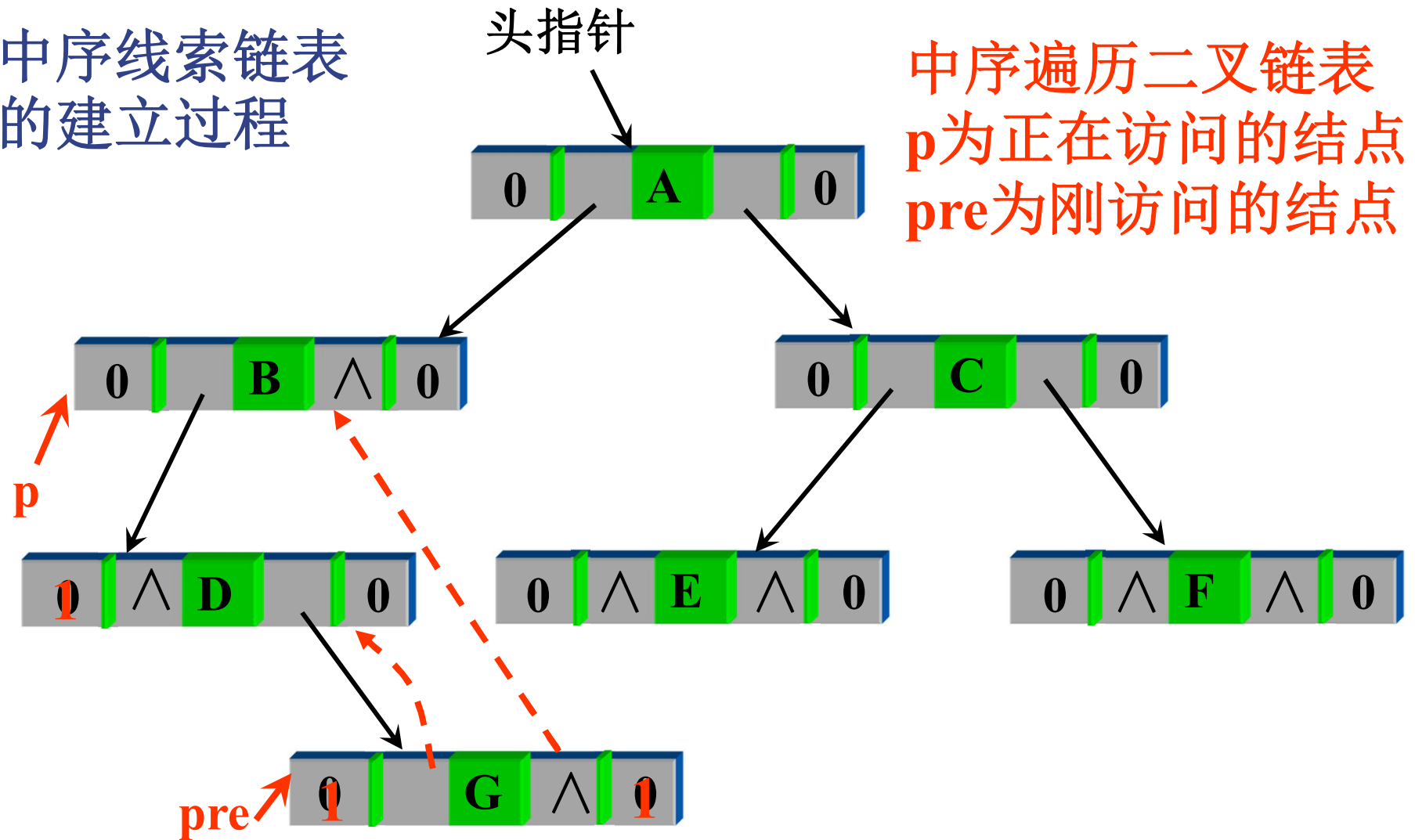
Threaded binary trees

中序线索链表的
建立过程



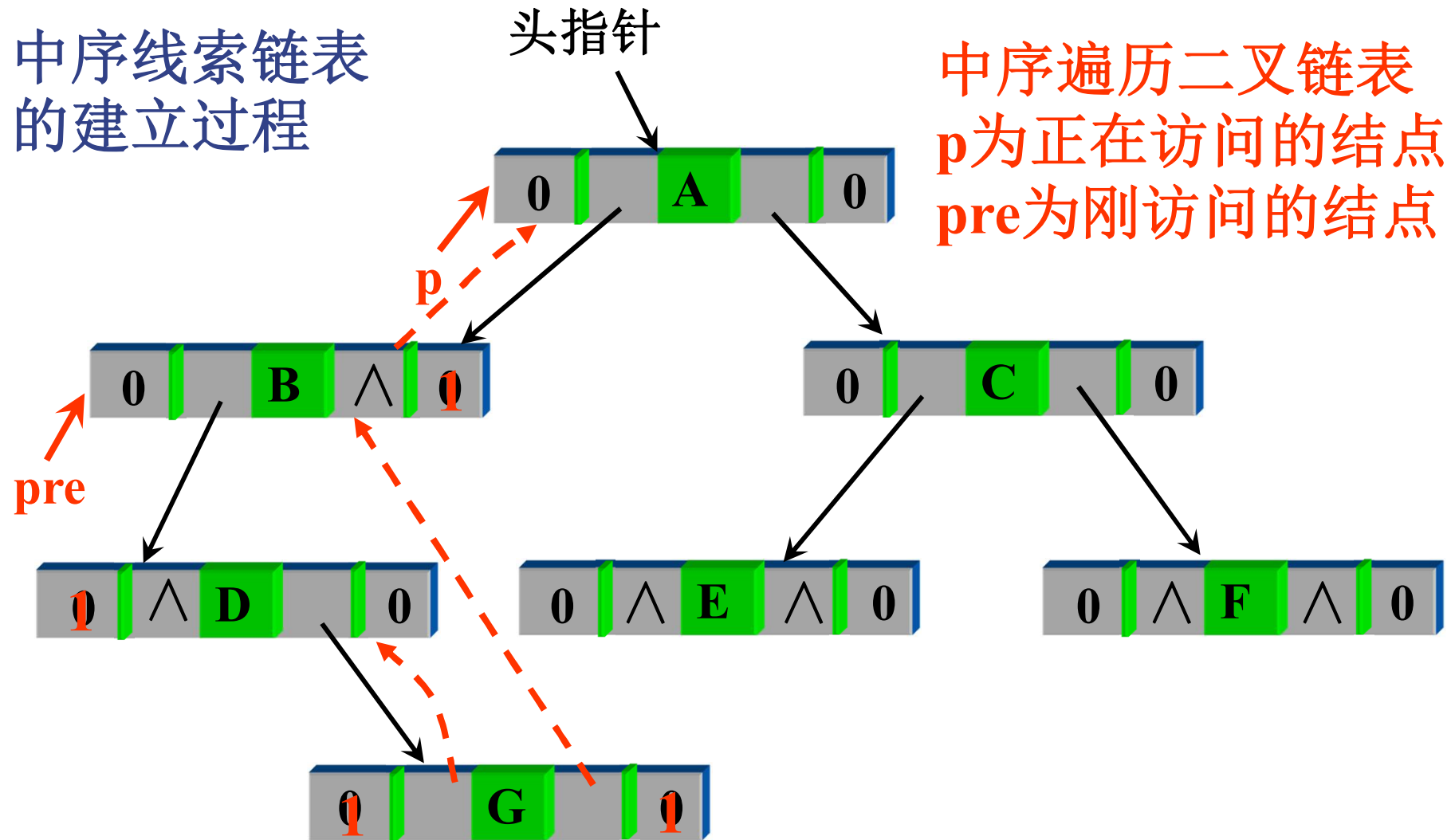
Threaded binary trees

中序线索链表的
建立过程



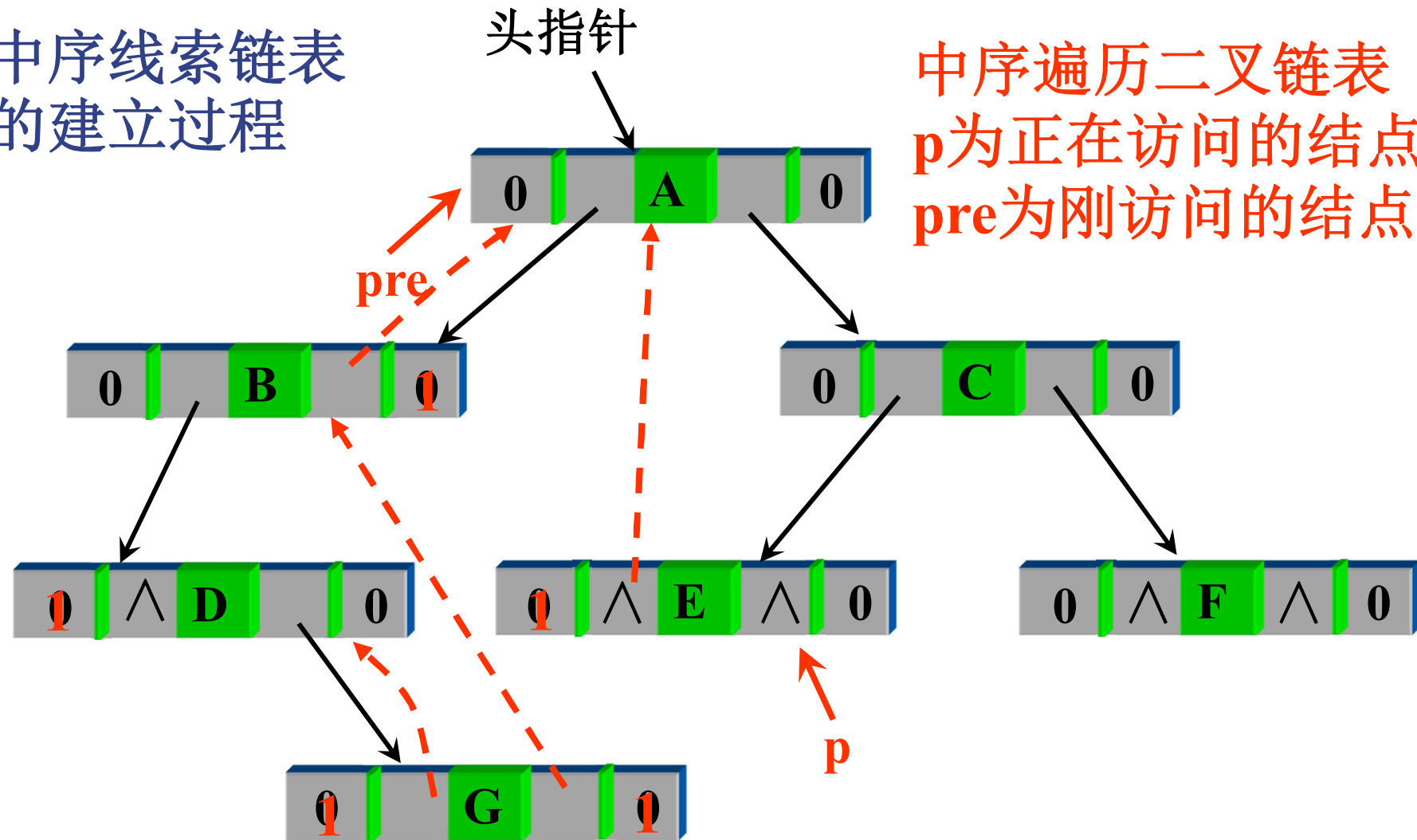
Threaded binary trees

中序线索链表的
建立过程



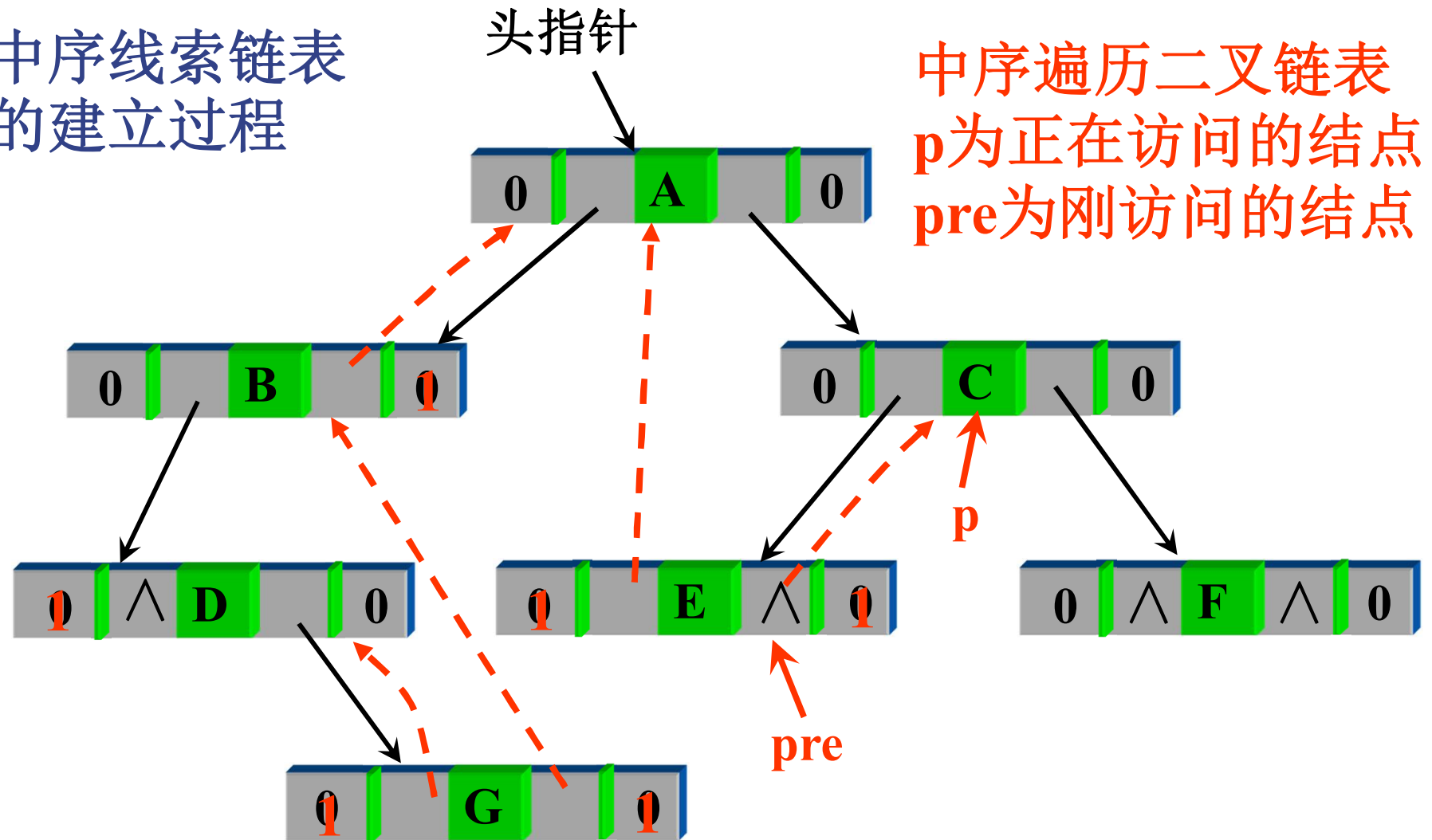
Threaded binary trees

中序线索链表的
建立过程



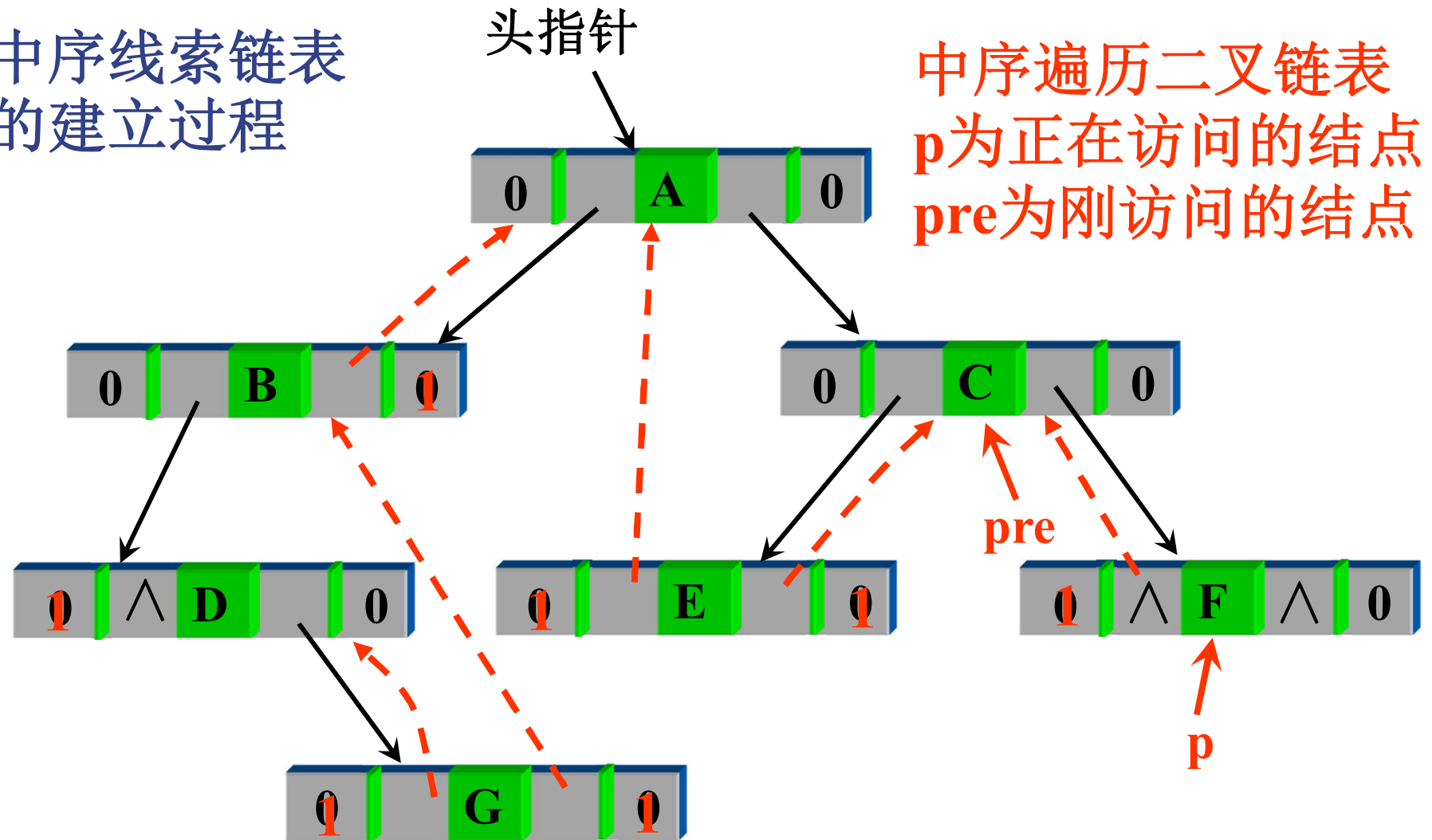
Threaded binary trees

中序线索链表的
建立过程



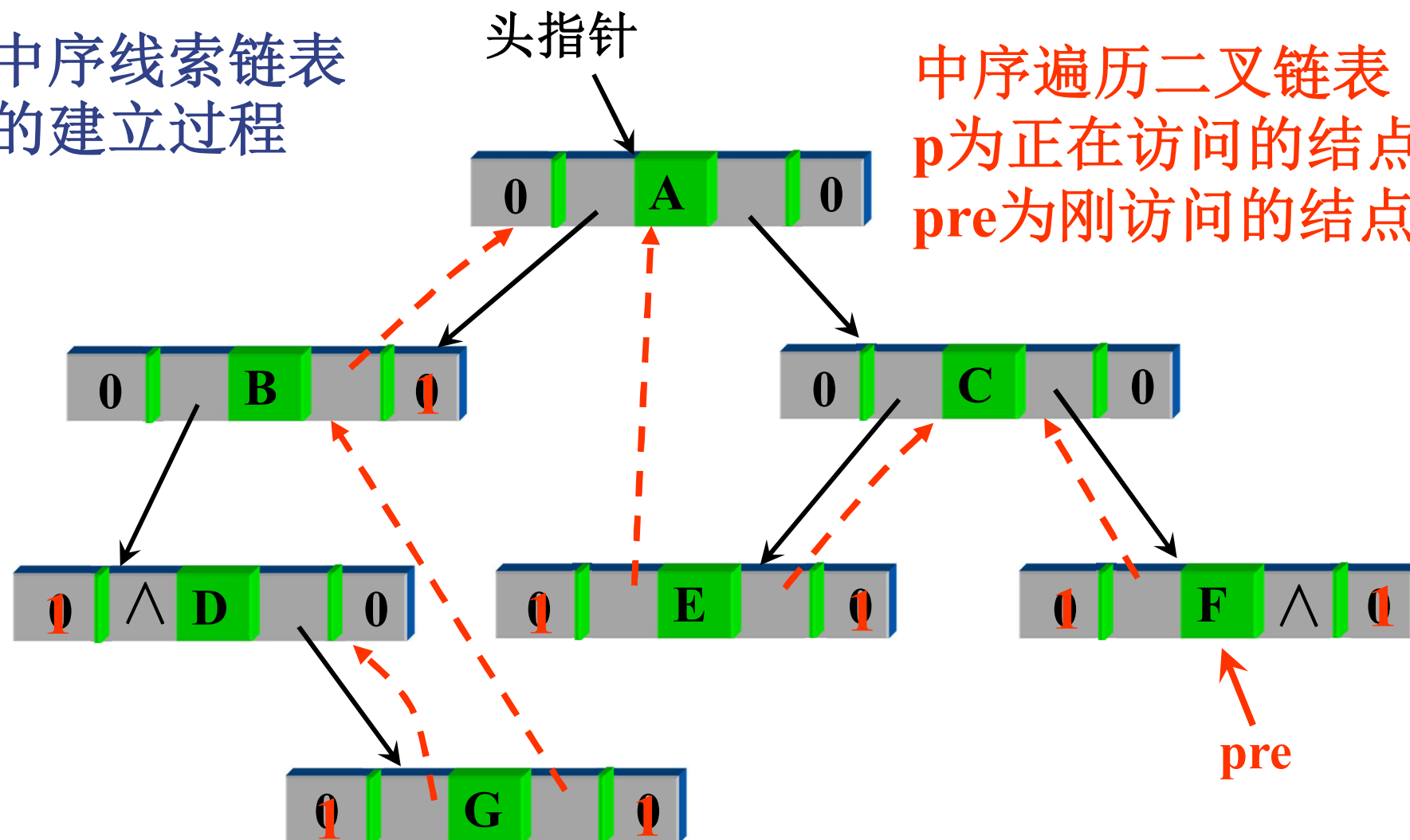
Threaded binary trees

中序线索链表的
建立过程



Threaded binary trees

中序线索链表的
建立过程



中序线索链表的建立

在遍历过程中，访问当前结点`root`的操作为：

- (1) 如果`root`的左、右指针域为空，则将相应标志置1；
- (2) 若`root`的左指针域为空，则令其指向它的前驱，这需要设指针`pre`始终指向刚刚访问过的结点，显然`pre`的初值为`NULL`；若`pre`的右指针域为空，则令其指向它的后继，即当前访问的结点`root`；
- (3) 令`pre`指向刚刚访问过的结点`root`；

Threaded binary trees

1. 建立二叉链表，将每个结点的左右标志置为0；
2. 遍历二叉链表，建立线索；
 - 2.1 如果二叉链表root为空，则空操作返回；
 - 2.2 对root的左子树建立线索；
 - 2.3 对根结点root建立线索；
 - 2.3.1 若root没有左孩子，则为root加上前驱线索；
 - 2.3.2 若root没有右孩子，则将root右标志置为1；
 - 2.3.3 若结点pre右标志为1，则为pre加上后继线索；
 - 2.3.4 令pre指向刚刚访问的结点root；
 - 2.4 对root的右子树建立线索。

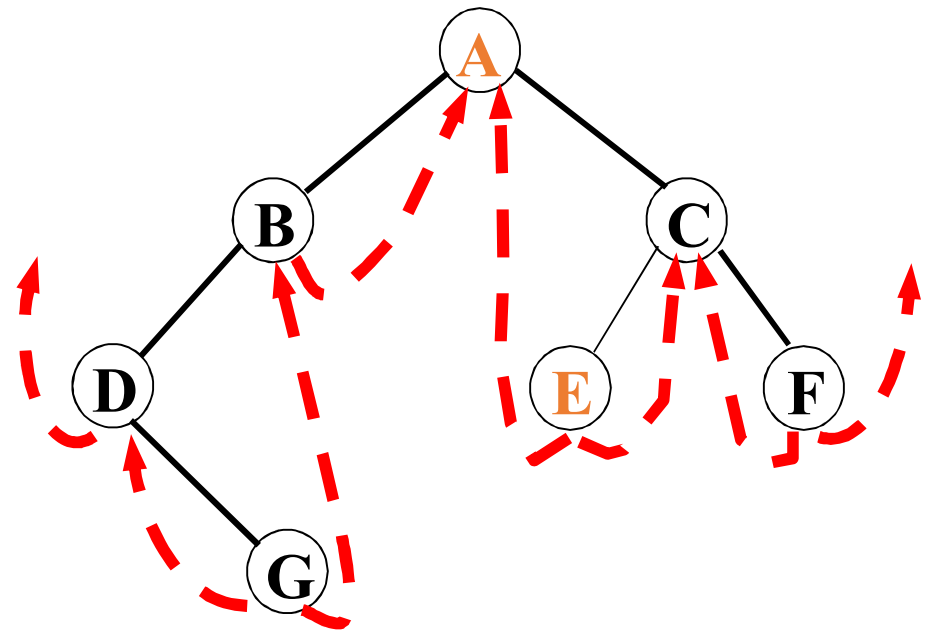
Threaded binary trees

```
template <class DataType>
void InThrBiTree<DataType> ::ThrBiTree(ThrNode<DataType> *bt,
                                         ThrNode<DataType> *pre)
{
    if (bt == NULL) return;
    ThrBiTree(bt->lchild, pre);
    if (bt->lchild == NULL) {           //对bt的左指针进行处理
        bt->ltag = 1;
        bt->lchild = pre;              //设置pre的前驱线索
    }
    if (bt->rchild == NULL) bt->rtag = 1; //对bt的右指针进行处理
    if (pre->rtag == 1) pre->rchild = bt; //设置pre的后继线索
    pre = bt;
    ThrBiTree(bt->rchild, pre);
}
```

Threaded binary trees

中序线索链表查找后继

- (1) 如果结点p的右标志为1，则表明该结点的右指针是线索；
- (2) 如果结点p的右标志为0，则表明该结点有右孩子。根据中序遍历的操作定义，它的后继结点应该是遍历其右子树时第一个访问的结点，即右子树中的最左下结点。

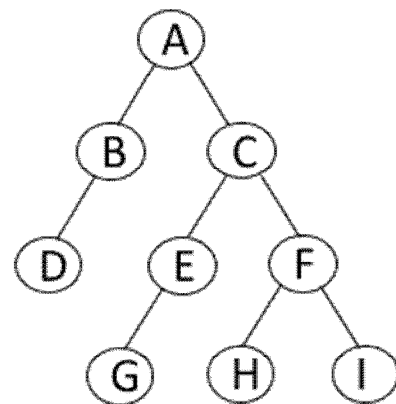


Threaded binary trees

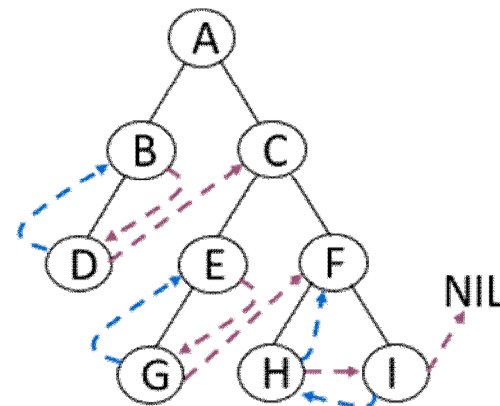
中序线索链表查找后继

```
template <class DataType>
ThrNode<DataType> *InThrBiTree<DataType> :: Next(
    ThrNode<DataType> *p)
{
    if (p->rtag == 1)
        q = p->rchild;           //右标志为1, 可直接得到后继结点
    else {
        q = p->rchild;           //工作指针q指向结点p的右孩子
        while (q->ltag == 0)     //查找最左下结点
            q = q->lchild;
    }
    return q;
}
```

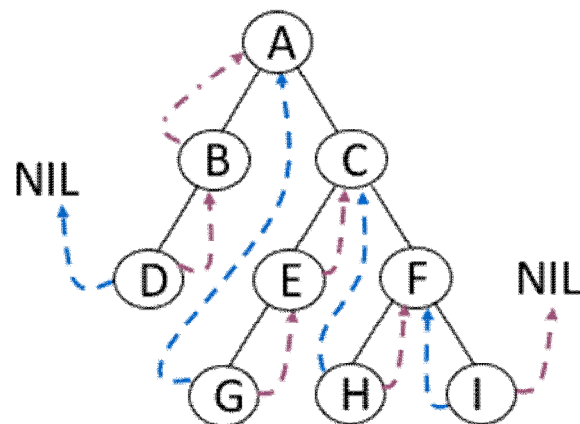
Threaded binary trees



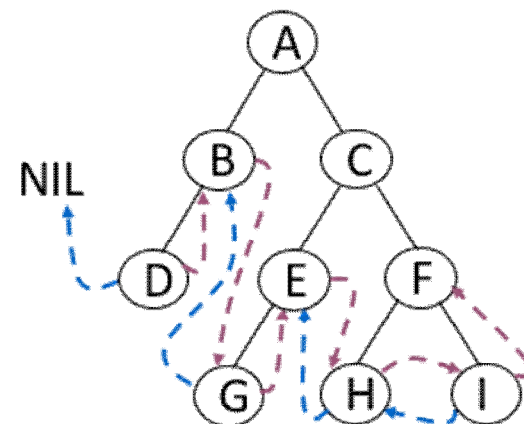
(a) 二叉树



(b) 先序线索树的逻辑形式
结点序列: **ABDCEGFHI**



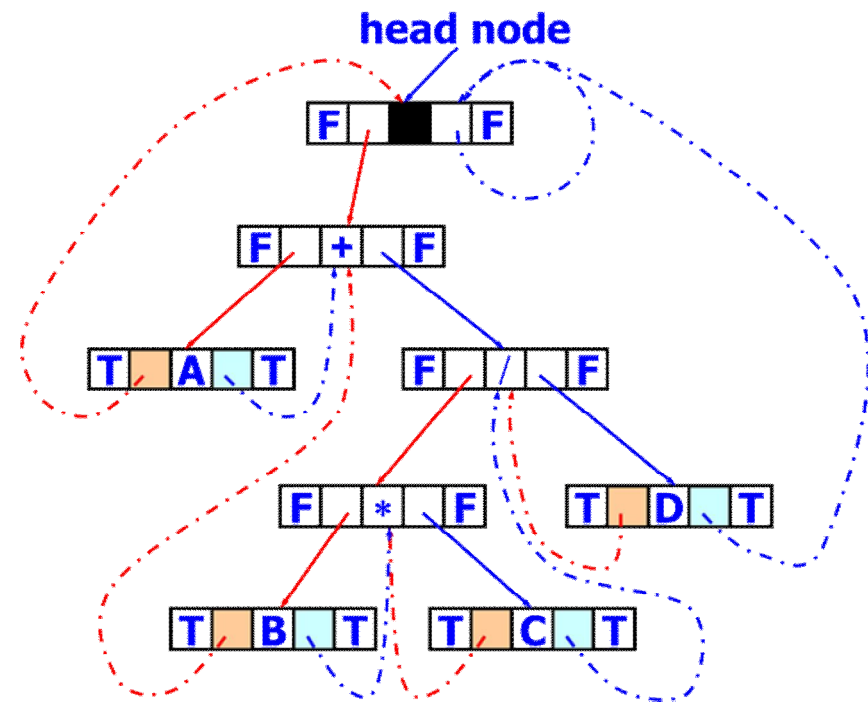
(c) 中序线索树的逻辑形式
结点序列: **DBAGECHFI**



(d) 后序线索树的逻辑形式
结点序列: **DBG E H I F C A**

Threaded binary trees

- 为了将二叉树中所有空指针域都利用上，以及操作便利的需要，在存储线索二叉树时往往增设一**头结点**，其结构与其它线索二叉树的结点结构一样，只是其数据域不存放信息，其左指针域指向二叉树的根结点，右指针域指向自己。
- 而原二叉树在某序遍历下的第一个结点的前驱线索和最后一个结点的后继线索都指向该头结点。



Applications: Haffman Tree

Haffman Tree

- The optimal binary tree 最优二叉树，也称哈夫曼(Haffman)树，是指对于一组带有确定权值的叶结点，构造的具有最小带权路径长度的二叉树。
- 那么什么是二叉树的带权路径长度呢？
- 在前面介绍过路径和结点的路径长度的概念，而二叉树的路径长度则是指由根结点到所有叶结点的路径长度之和。如果二叉树中的叶结点都具有一定的权值，则可将这一概念加以推广。

Applications: Haffman Tree

- 设二叉树具有n个带权值的叶结点，那么从根结点到各个叶结点的路径长度与相应结点权值的乘积之和叫做二叉树的带权路径长度，记为：

$$WPL = \sum_{k=1}^n W_k \times L_k$$

- 其中 W_k 为第k个叶结点的权值， L_k 为第k个叶结点的路径长度。

Applications: Haffman Tree

- 【例】如图6.16所示的二叉树，它的带权路径长度值

$$WPL = 2 \times 2 + 4 \times 2 + 5 \times 2 + 3 \times 2 = 28。$$

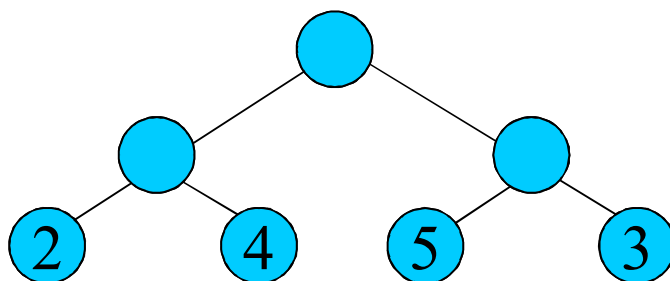


图6.16 一个带权二叉树

- 在给定一组具有确定权值的叶结点，可以构造出不同的带权二叉树。例如，给出4个叶结点，设其权值分别为1,3,5,7，我们可以构造出形状不同的多个二叉树。这些形状不同的二叉树的带权路径长度将各不相同。图6.17给出了其中5个不同形状的二叉树。

Applications: Haffman Tree

- 这五棵树的带权路径长度分别为：

(a) $WPL = 1 \times 2 + 3 \times 2 + 5 \times 2 + 7 \times 2 = 32$

(b) $WPL = 1 \times 3 + 3 \times 3 + 5 \times 2 + 7 \times 1 = 29$

(c) $WPL = 1 \times 2 + 3 \times 3 + 5 \times 3 + 7 \times 1 = 33$

(d) $WPL = 7 \times 3 + 5 \times 3 + 3 \times 2 + 1 \times 1 = 43$

(e) $WPL = 7 \times 1 + 5 \times 2 + 3 \times 3 + 1 \times 3 = 29$

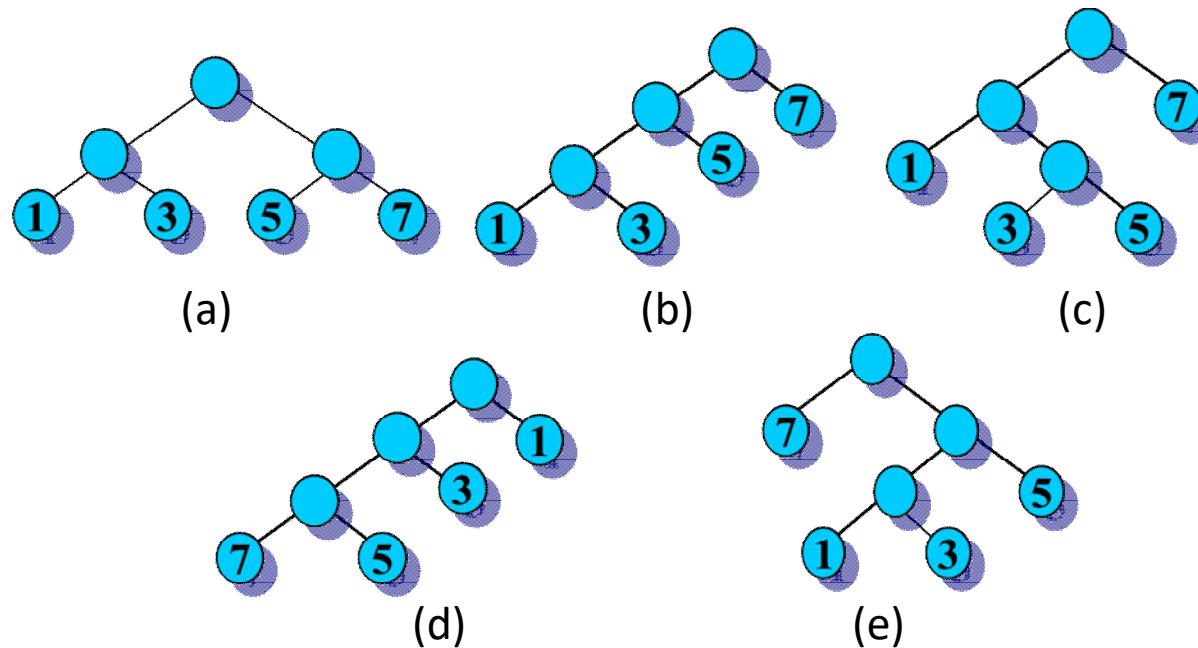


图6.17 具有相同叶结点和不同带权路径长度的二叉树

Applications: Haffman Tree

- 由相同权值的一组叶结点所构成的二叉树有不同的形态和不同的带权路径长度，那么如何找到带权路径长度最小的二叉树呢？
- 根据哈夫曼树的定义，一棵二叉树要使其WPL值最小，必须使权值越大的叶结点越靠近根结点，而权值越小的叶结点越远离根结点。
- 哈夫曼(Haffman)依据这一特点提出了一种方法。

Applications: Haffman Tree

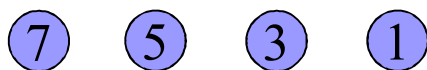
哈夫曼树的基本思想

- (1) 由给定的 n 个权值 $\{W_1, W_2, \dots, W_n\}$ 构造 n 棵只有一个叶结点的二叉树，从而得到一个二叉树的集合 $F = \{T_1, T_2, \dots, T_n\}$ ；
- (2) 在 F 中选取根结点的权值最小和次小的两棵二叉树作为左、右子树构造一棵新的二叉树，这棵新的二叉树根结点的权值为其左、右子树根结点权值之和；
- (3) 在集合 F 中删除作为左、右子树的两棵二叉树，并将新建立的二叉树加入到集合 F 中；
- (4) 重复(2)、(3)两步，当 F 中只剩下一棵二叉树时，这棵二叉树便是所要建立的哈夫曼树。

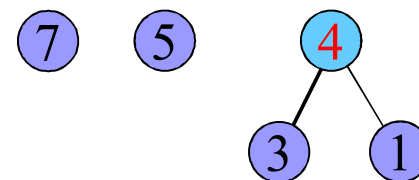
Applications: Haffman Tree

- 图6.18给出了前面提到的叶结点权值集合为 $W = \{1, 3, 5, 7\}$ 的哈夫曼树的构造过程。可以计算出其带权路径长度为29，由此可见，对于同一组给定叶结点所构造的哈夫曼树，树的形状可能不同，但带权路径长度值是相同的，一定是最小的。

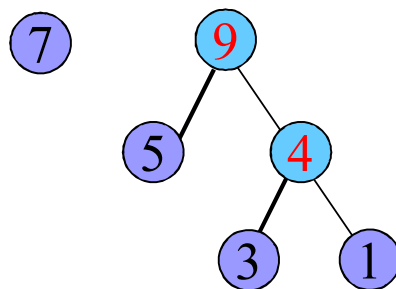
第一步



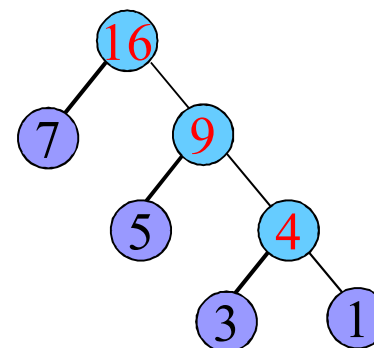
第二步



第三步



第四步



哈夫曼树的建立过程

谢谢！

