6.3.2 Threading binary tree

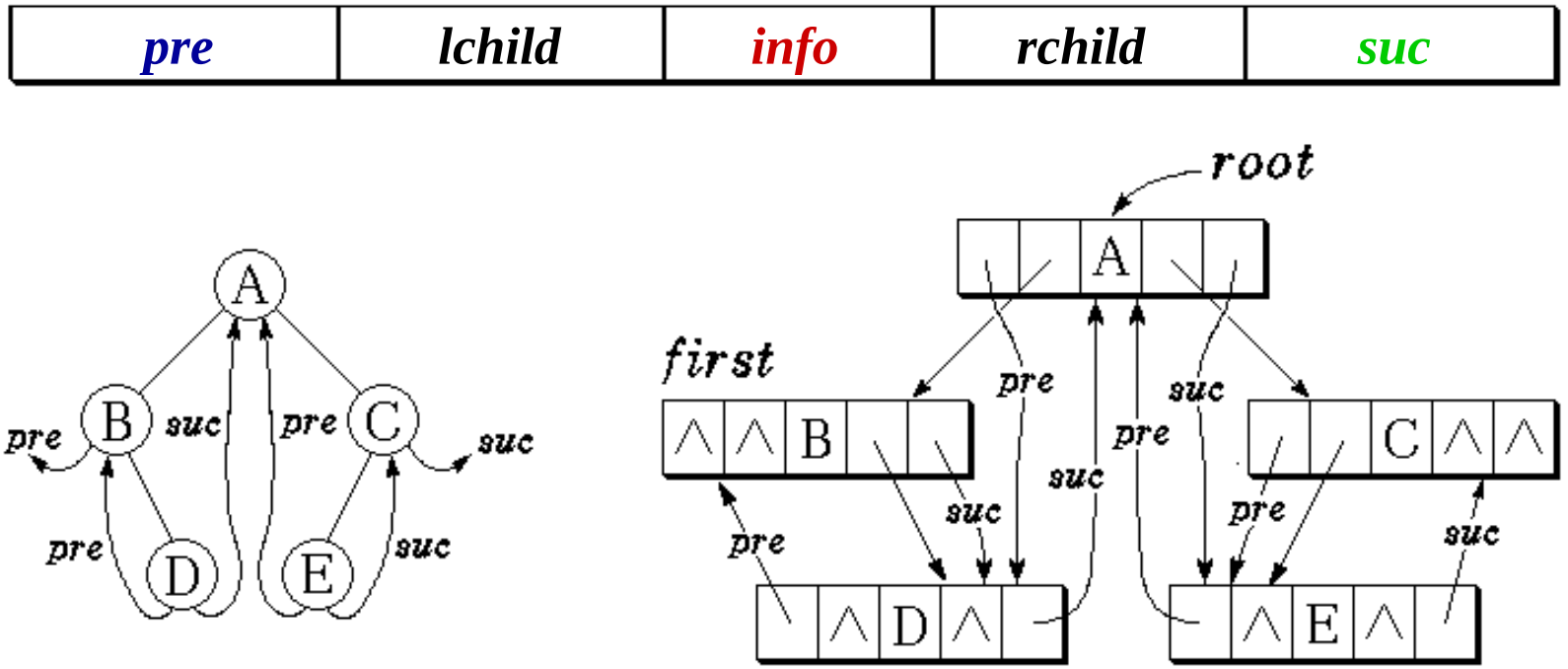
遍历二叉树是按一定的规则将二叉树中结点排列成一个线性序列，这实际上是把一个非线性结构进行线性化的操作。

以二叉链表作为存储结构时，对于某个结点只能找到其左右孩子，而不能直接得到结点在任一序列中的逻辑前驱或后继。要想得到，只能通过遍历的动态过程才行。

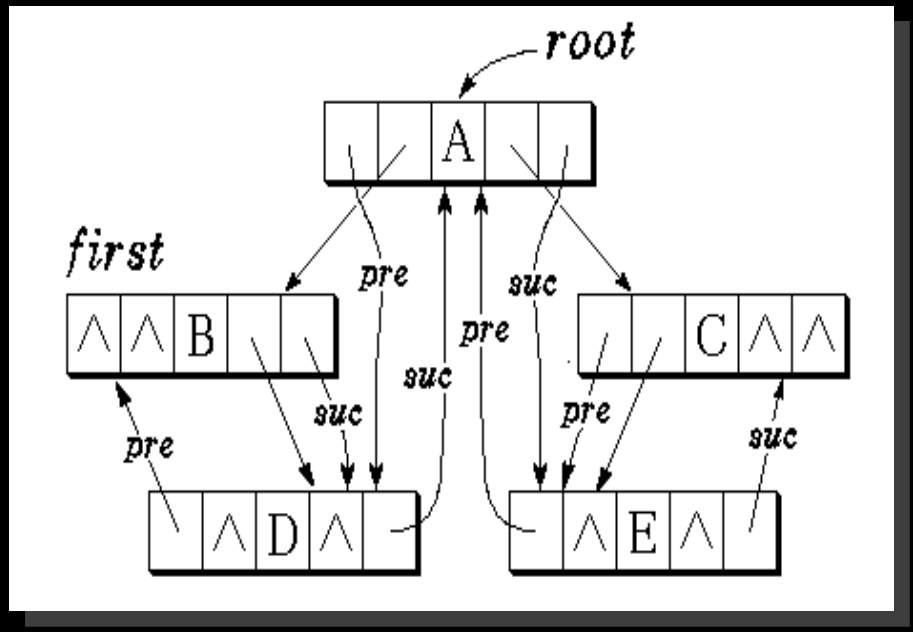
怎样保存遍历过程中得到的信息呢？

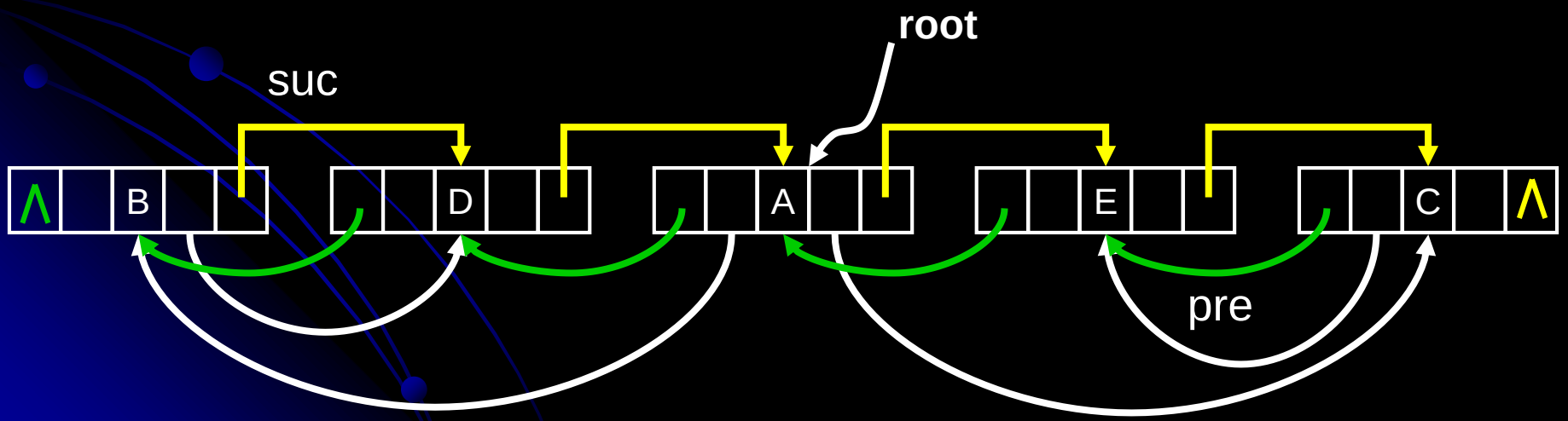
(1) 增加两个指针，分别指示其前驱和后继结点

predecessor **successor**

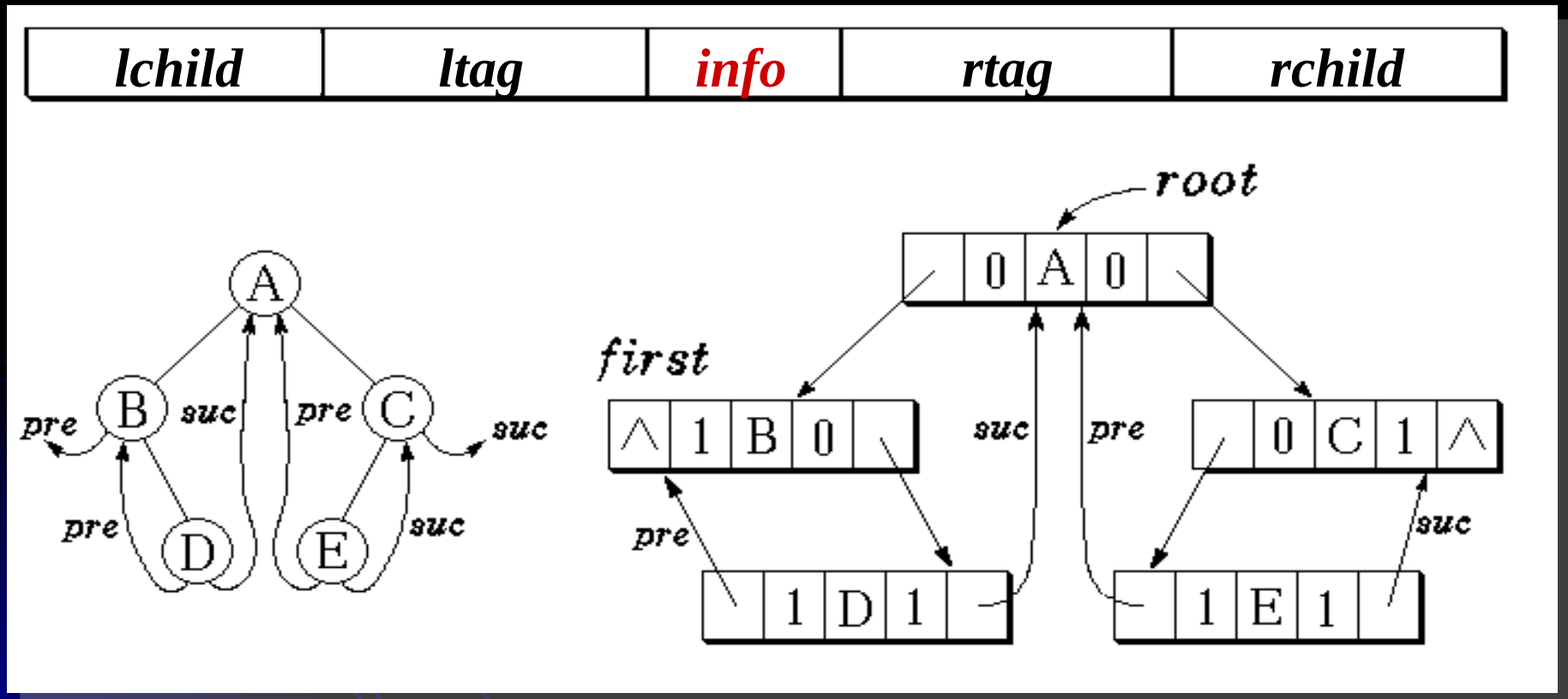


```
struct ThrTreeNode
/* 线索树中每个结点的结构 */
{
    DataType info;
    struct ThrTreeNode  *lchild;
    struct ThrTreeNode  *rchild;
    struct ThrTreeNode  *pre;
    struct ThrTreeNode  *suc;
}ThrTree, *PThrTree, *PThrTreeNode;
```





(2) 利用结构中的空链域，并设立标志，即采用如下的形式



```
struct ThrTreeNode     /* 线索树中每个结点的结构 */
{
    DataType  info;
    struct ThrTreeNode  *lchild, *rchild;
    int  ltag, rtag;
}ThrTree, *PThrTree, *PThrTreeNode;
```

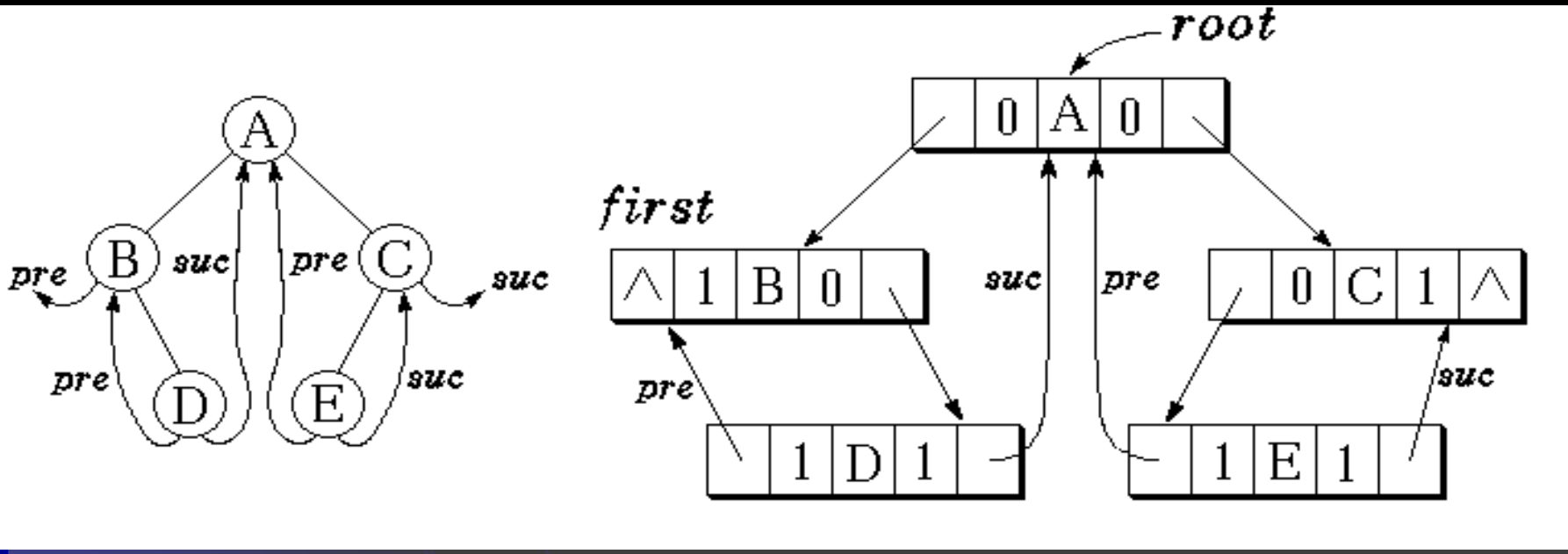
其中当ltag或rtag为0时， llink或rlink为**指针**；否则为1时，llink或rlink表示**线索**。

线索链表：以上面结构构成的二叉链表作为二叉树的存储结构，叫做**线索链表**。

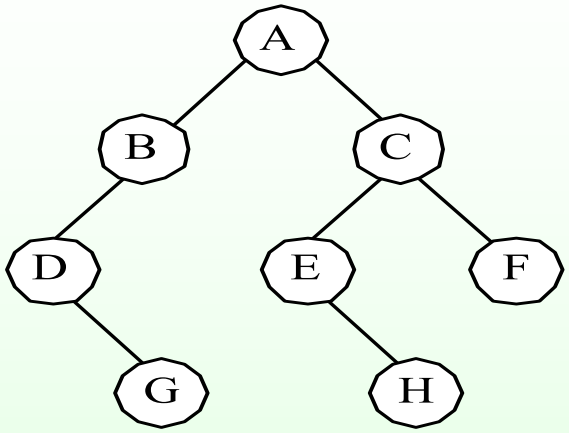
线索：指向结点逻辑前驱或后继的指针叫做**线索**。

线索二叉树：加上线索的二叉树叫**线索二叉树**。

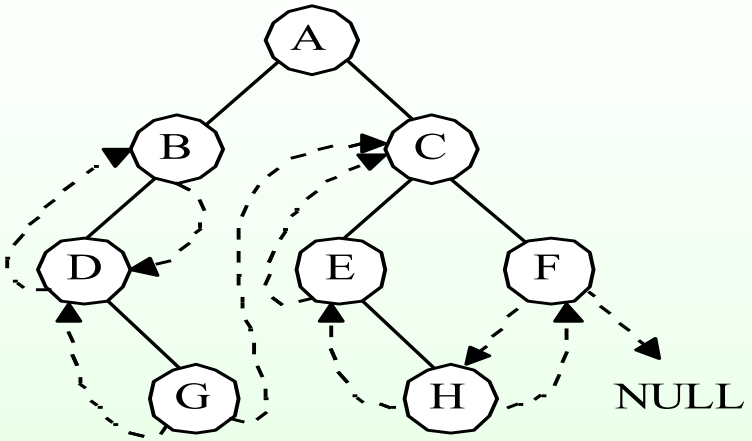
线索化：对二叉树以某种次序遍历使其变为线索二叉树的过程叫做线索化。



ABDGCEHF

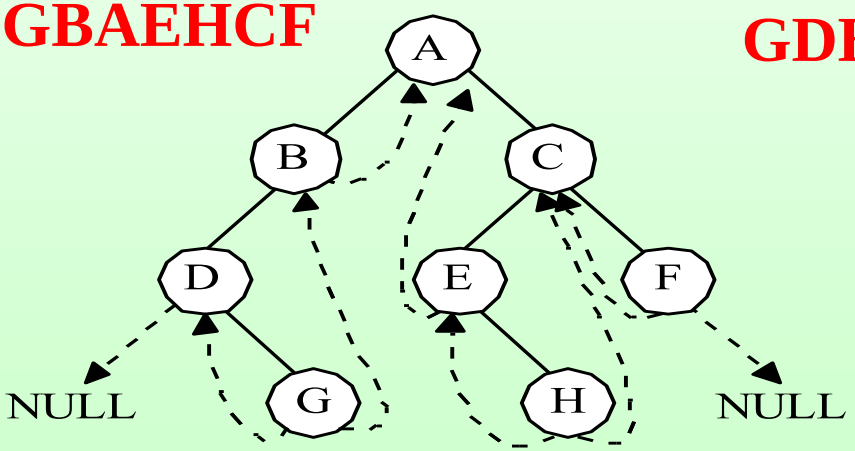


(a) 二叉树



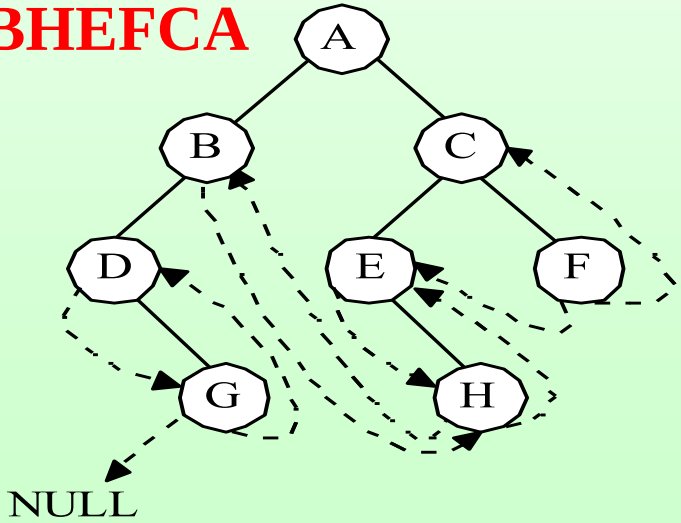
(b)先序线索二叉树

DGBAEHCF



(c) 中序线索二叉树

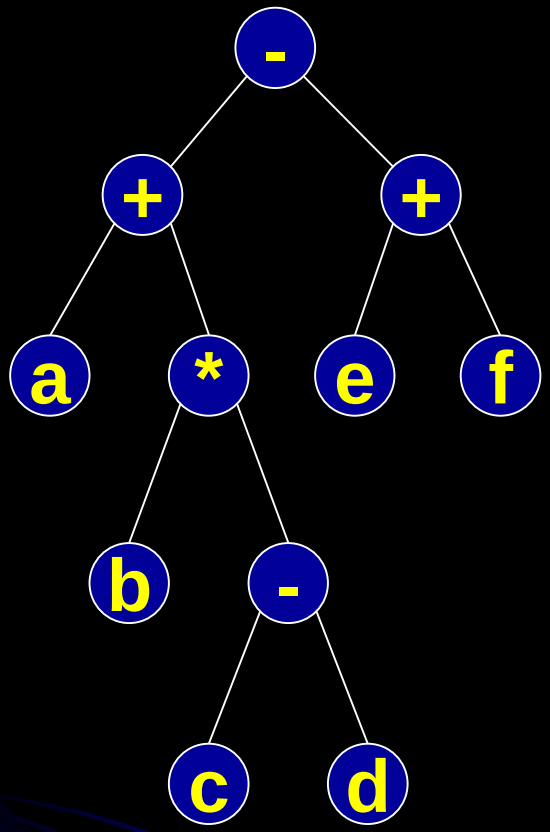
GDBHEFCA

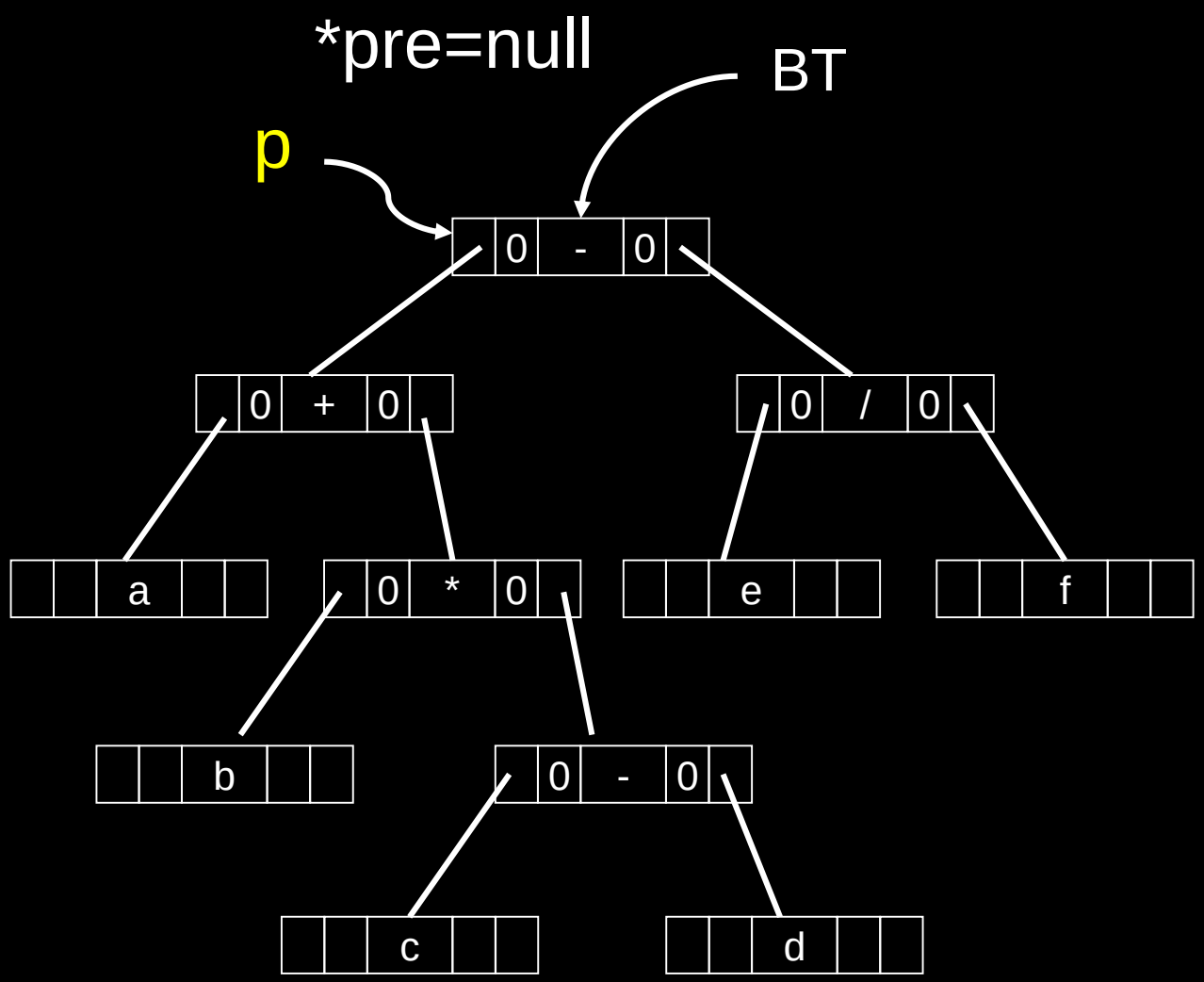


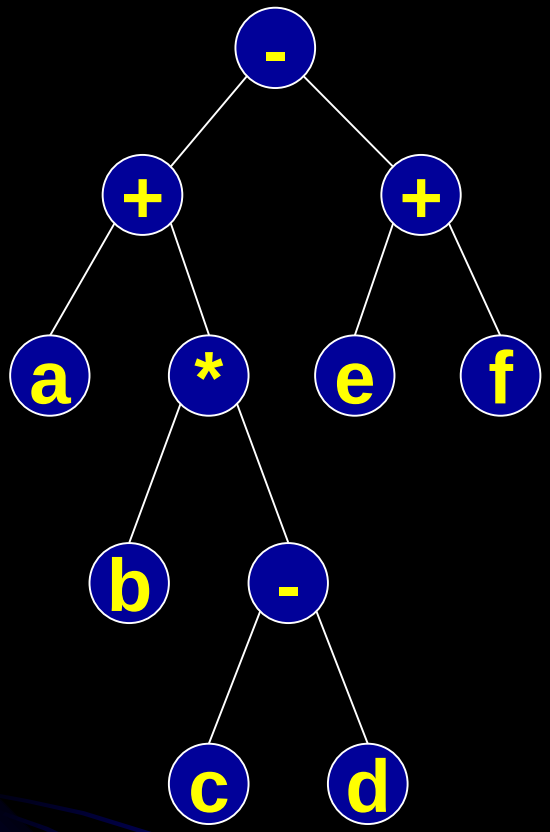
(d) 后序线索二叉树

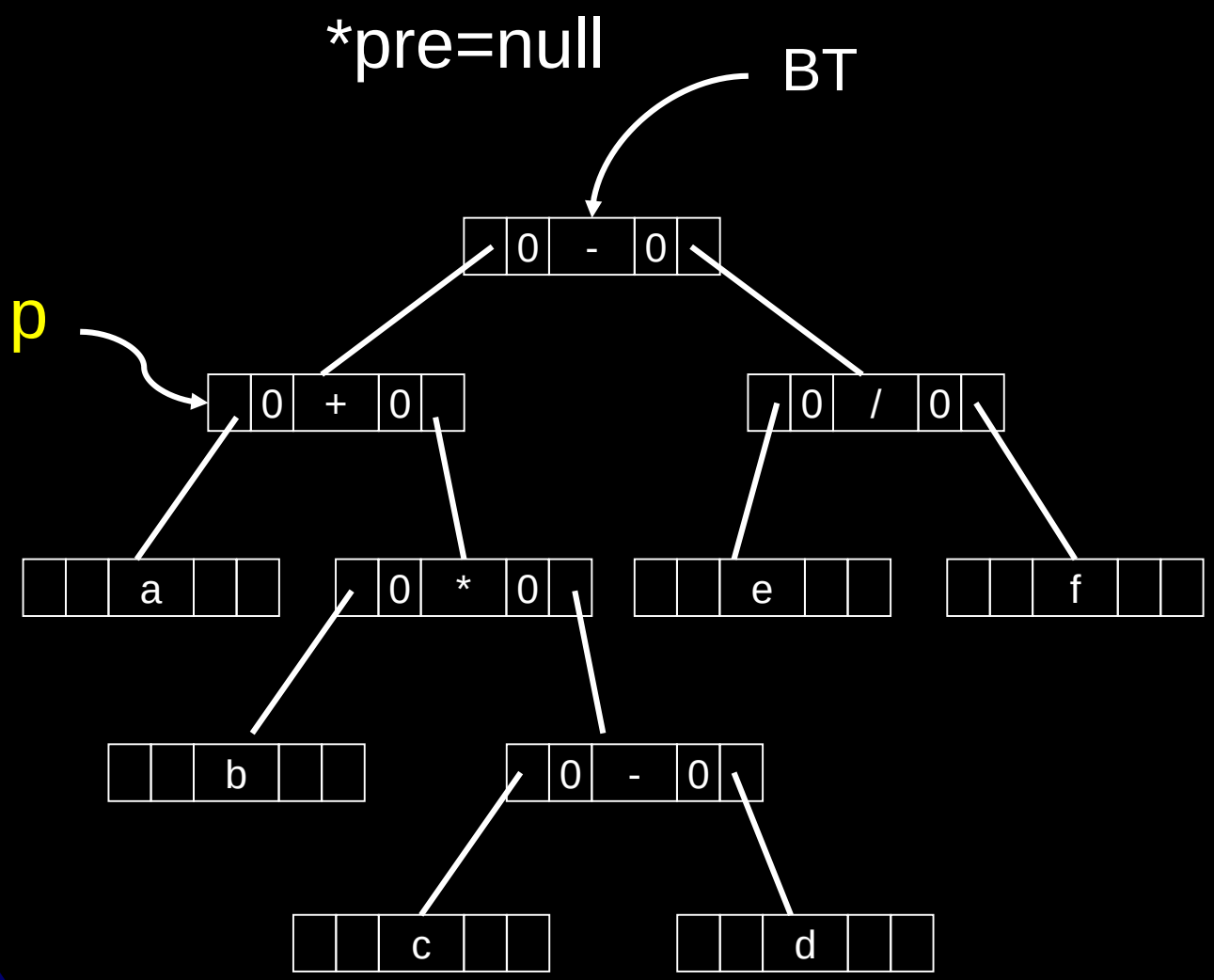
二叉树的中序线索化

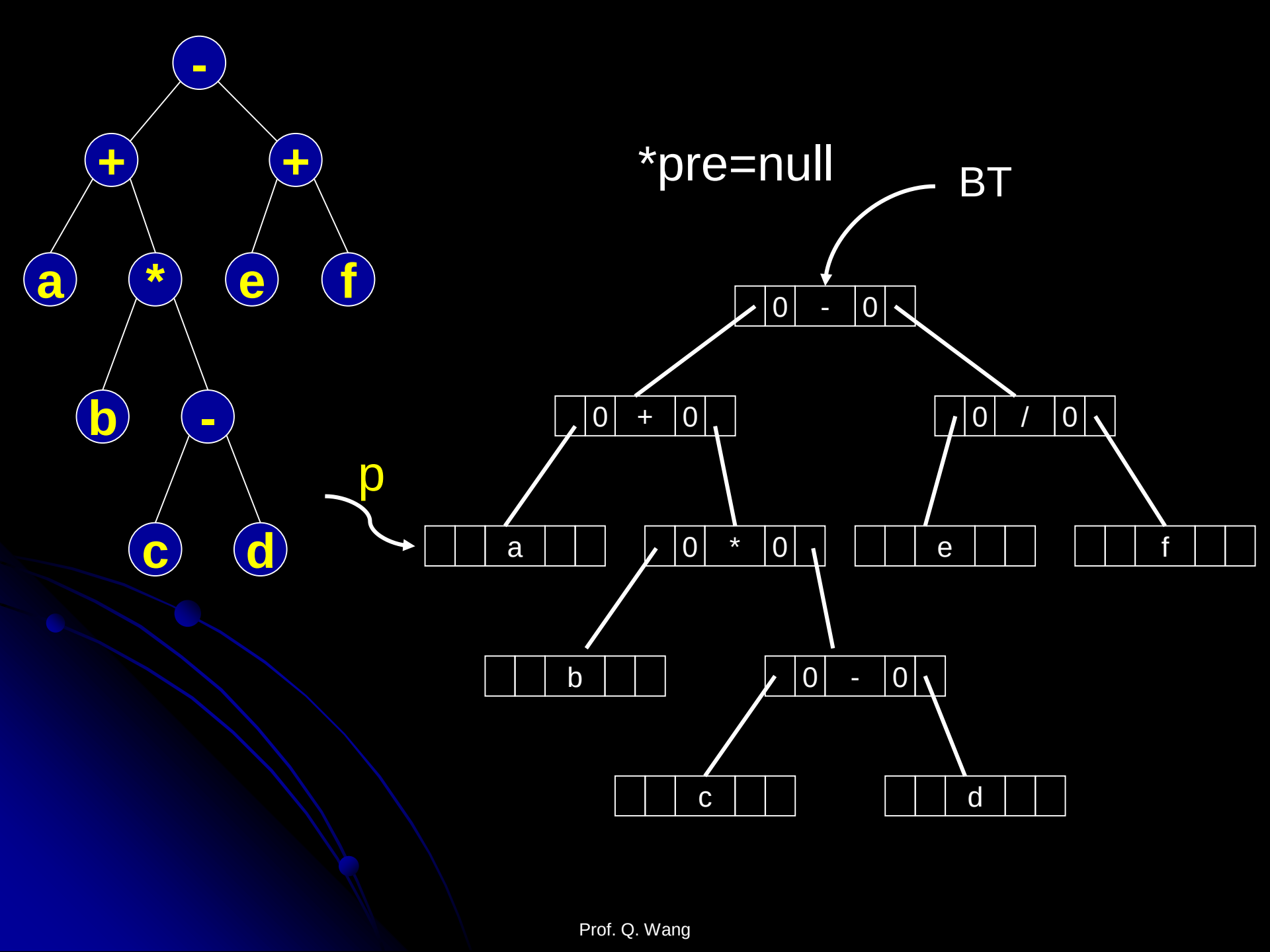
- 原理
  - 在遍历的过程中，修改二叉树中原有的**n+1**个空指针
  - 必备的辅助存储，**设置一个指针*pre始终指向刚刚访问过的结点**
- 算法

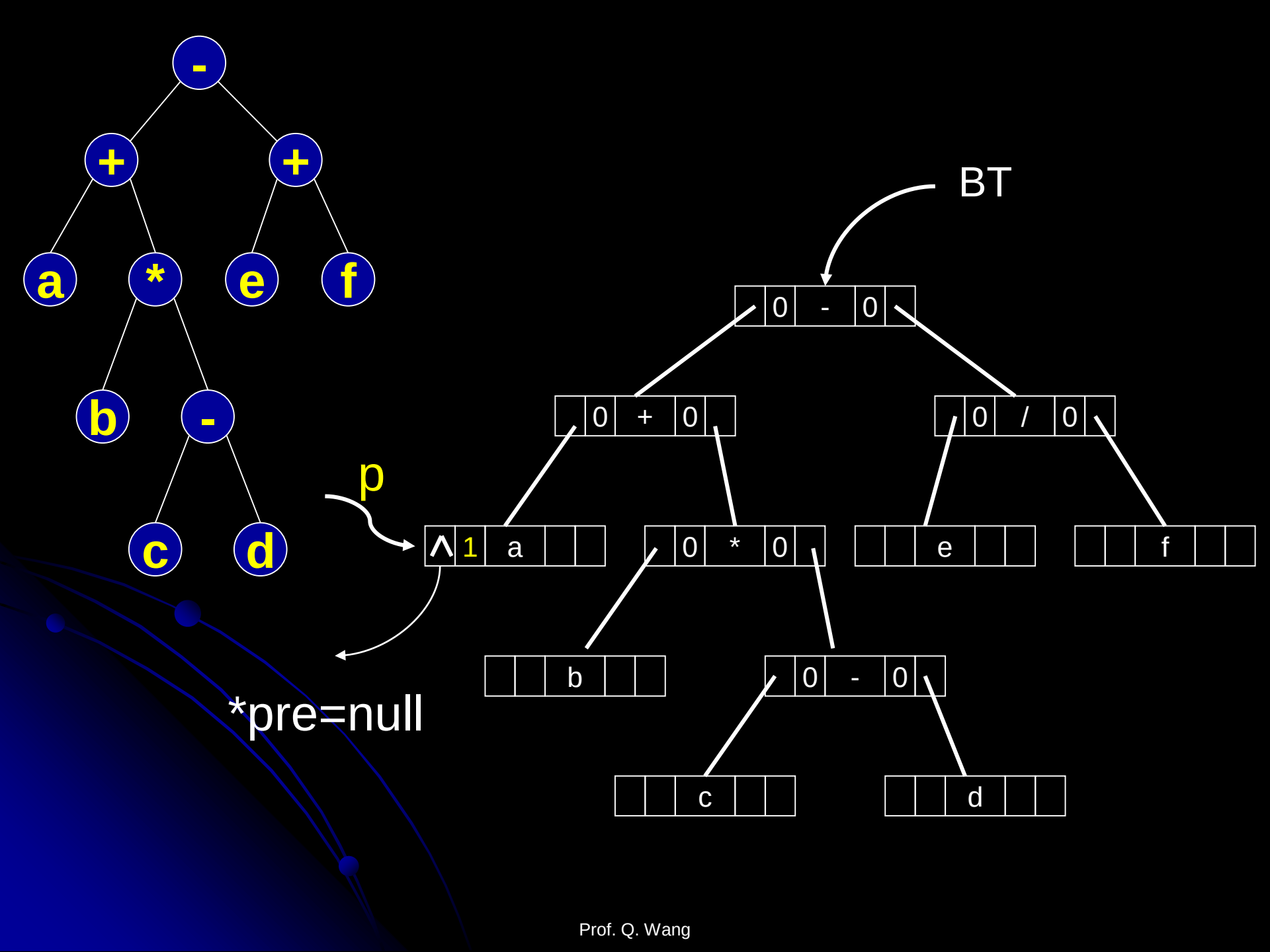

-
+
+
a
*
e
f
b
-
c
d

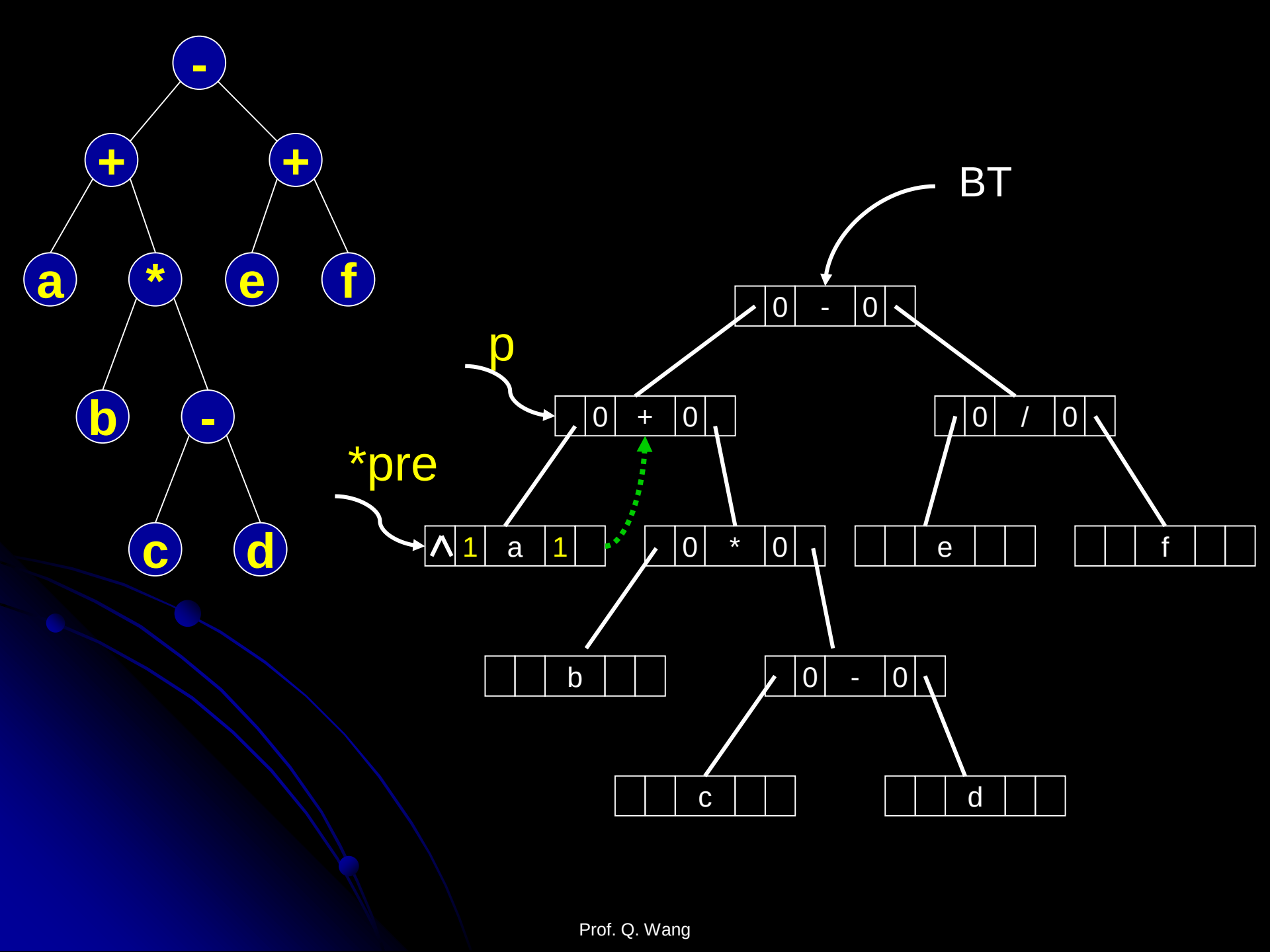

*pre=null
BT
p
0 - 0
0 + 0
0 / 0
a
0 * 0
e
f
b
0 - 0
c
d

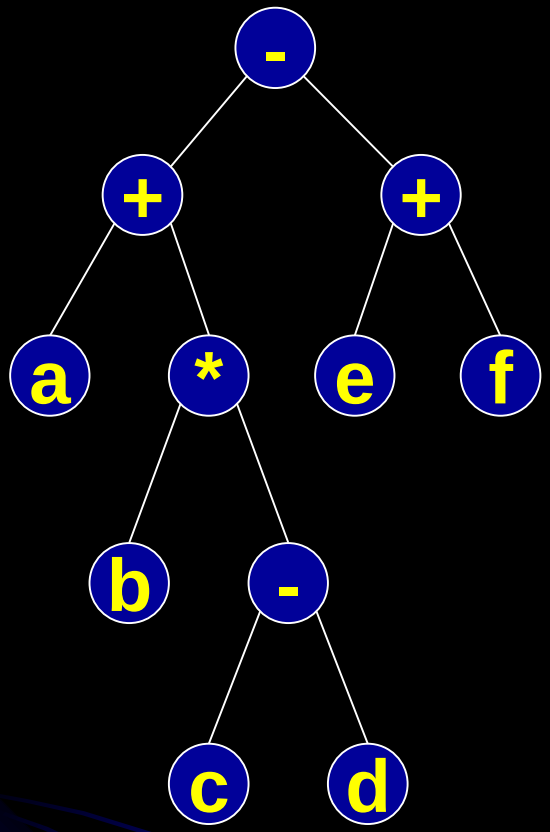

-
+
+
a
*
e
f
b
-
c
d

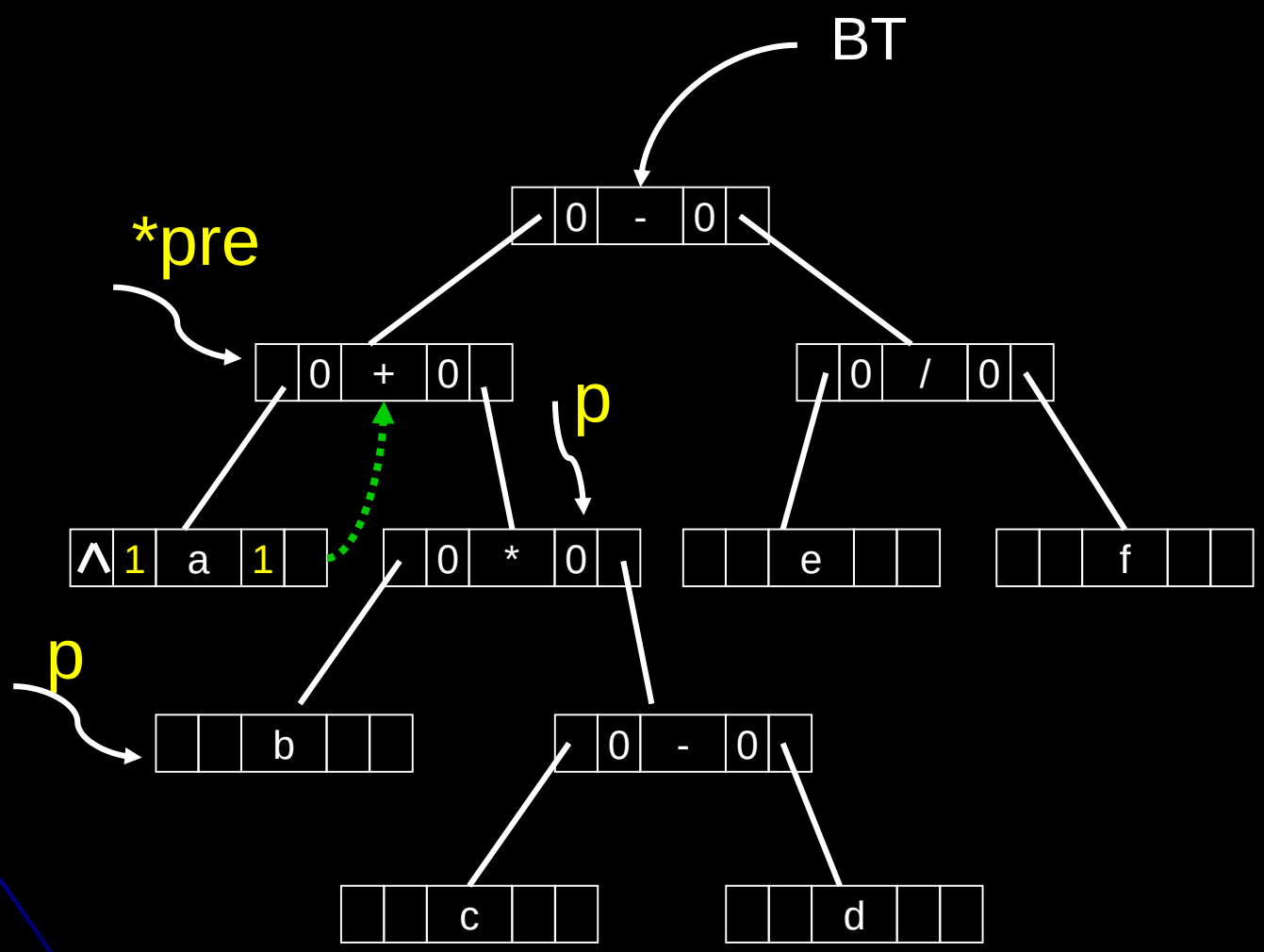

*pre=null
BT
p
0 - 0
0 + 0
0 / 0
a
0 * 0
e
f
b
0 - 0
c
d

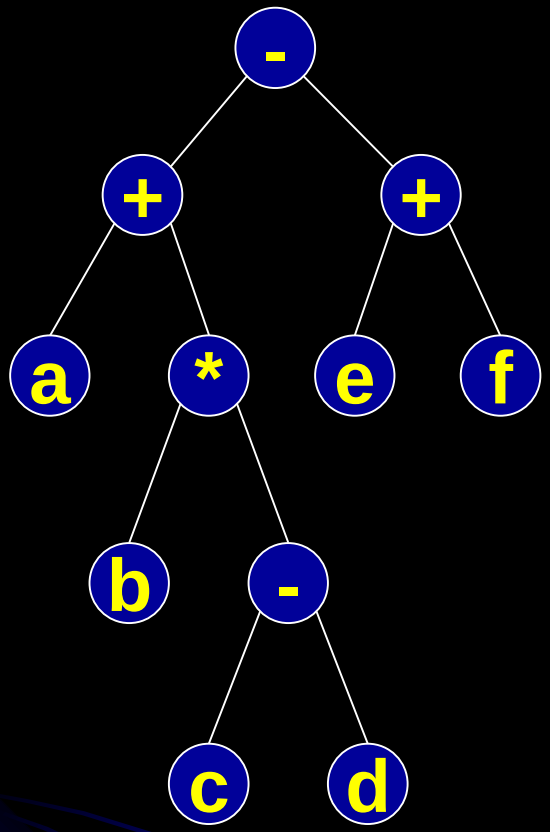

-
+
+
a
*
e
f
b
-
c
d
*pre=null
BT
p
0 - 0
0 + 0
0 / 0
a
0 * 0
e
f
b
0 - 0
c
d

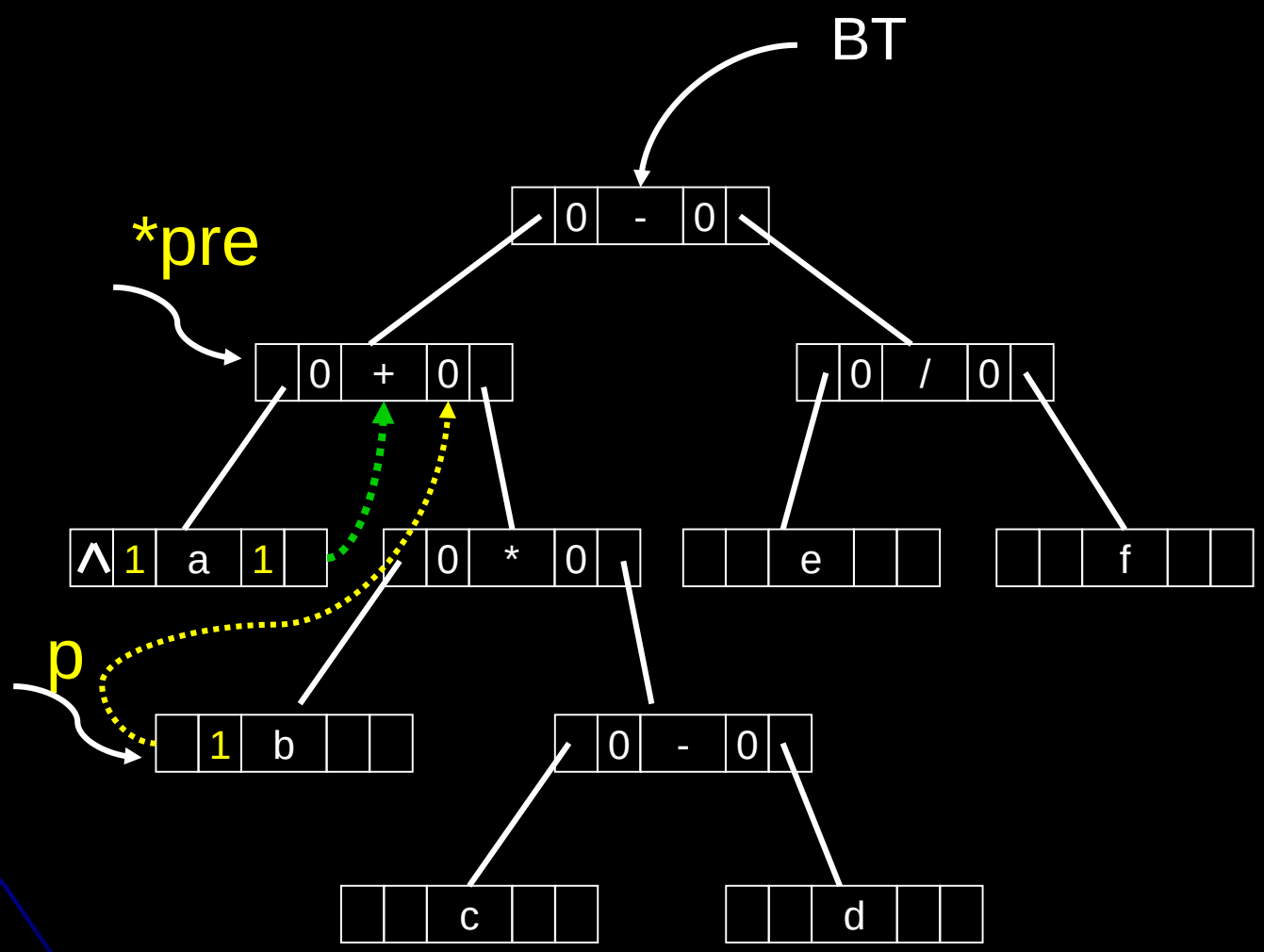

-
+
+
a
*
e
f
b
-
c
d
p
BT
0 - 0
0 + 0
0 / 0
^ 1 a
0 * 0
e
f
b
0 - 0
c
d
*pre=null

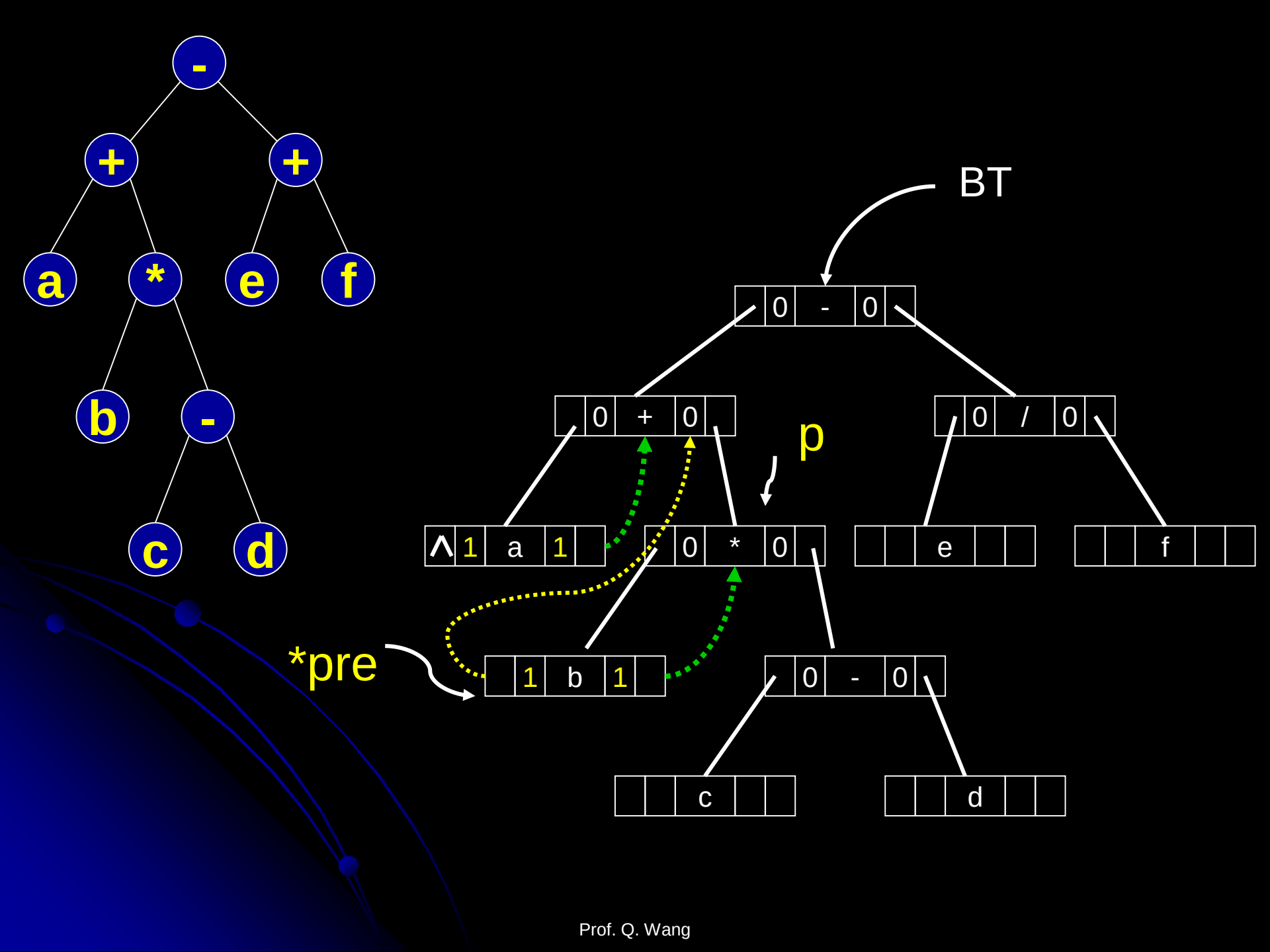

-
+
+
a
*
e
f
b
-
c
d
BT
0 - 0
p
0 + 0
0 / 0
*pre
^ 1 a 1
0 * 0
e
f
b
0 - 0
c
d

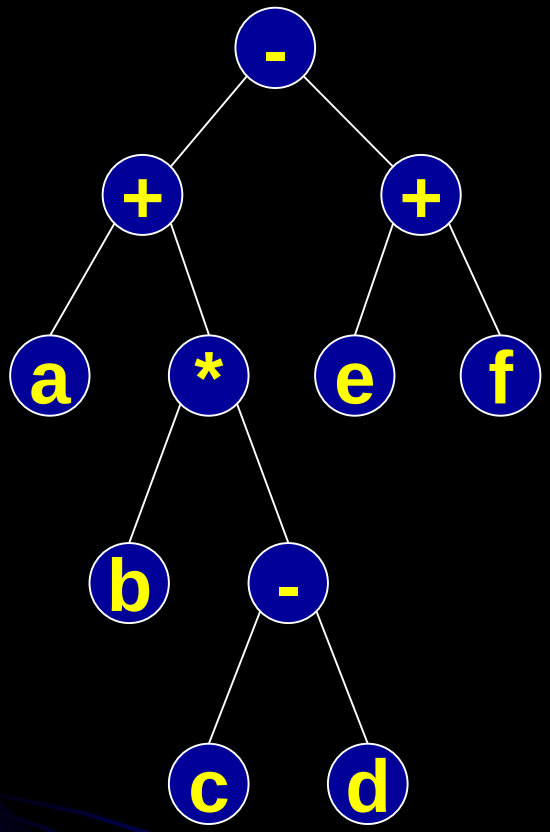

-
+
+
a
*
e
f
b
-
c
d

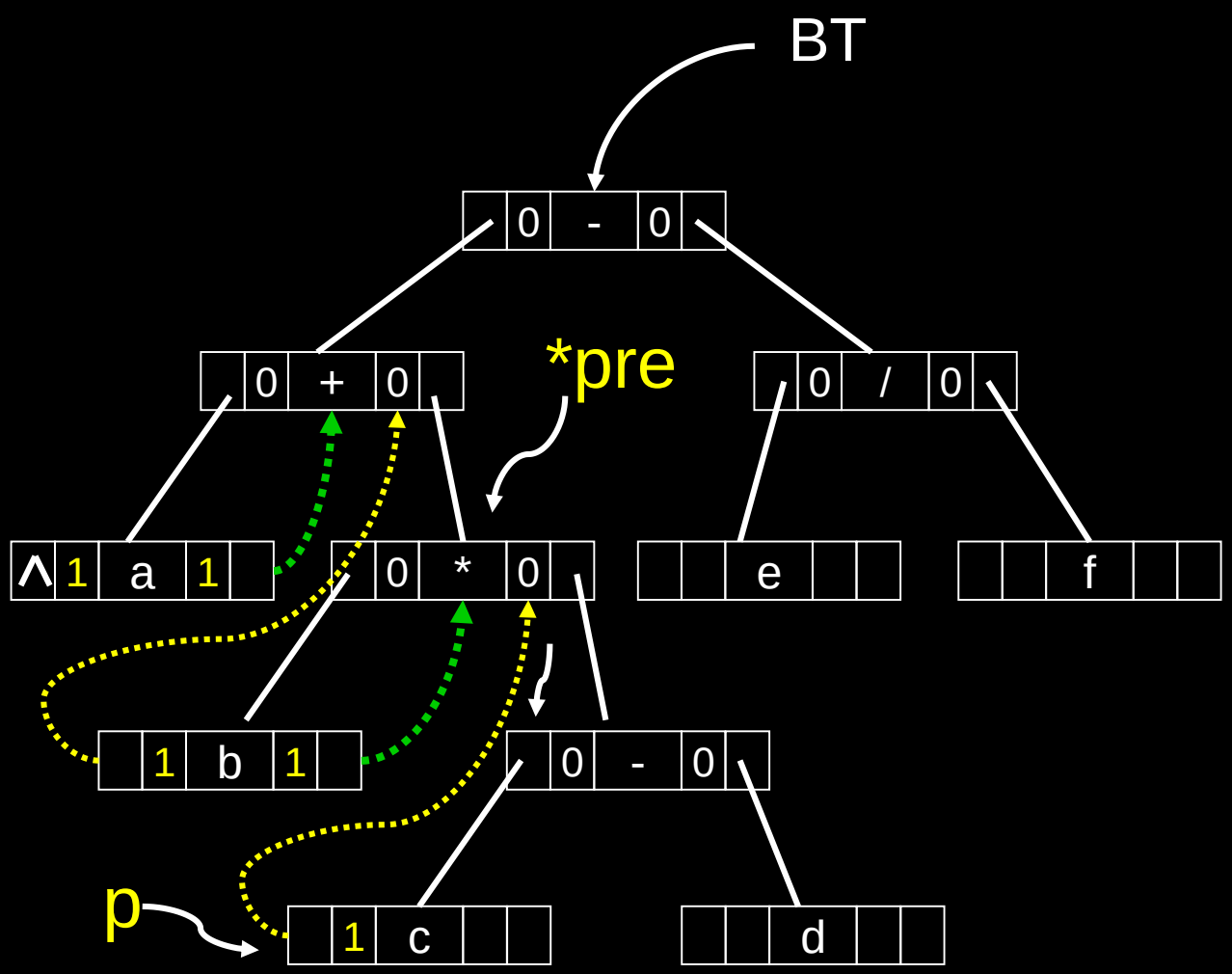

BT
*pre
p
p
0 - 0
0 + 0
0 / 0
^ 1 a 1
0 * 0
e
f
b
0 - 0
c
d

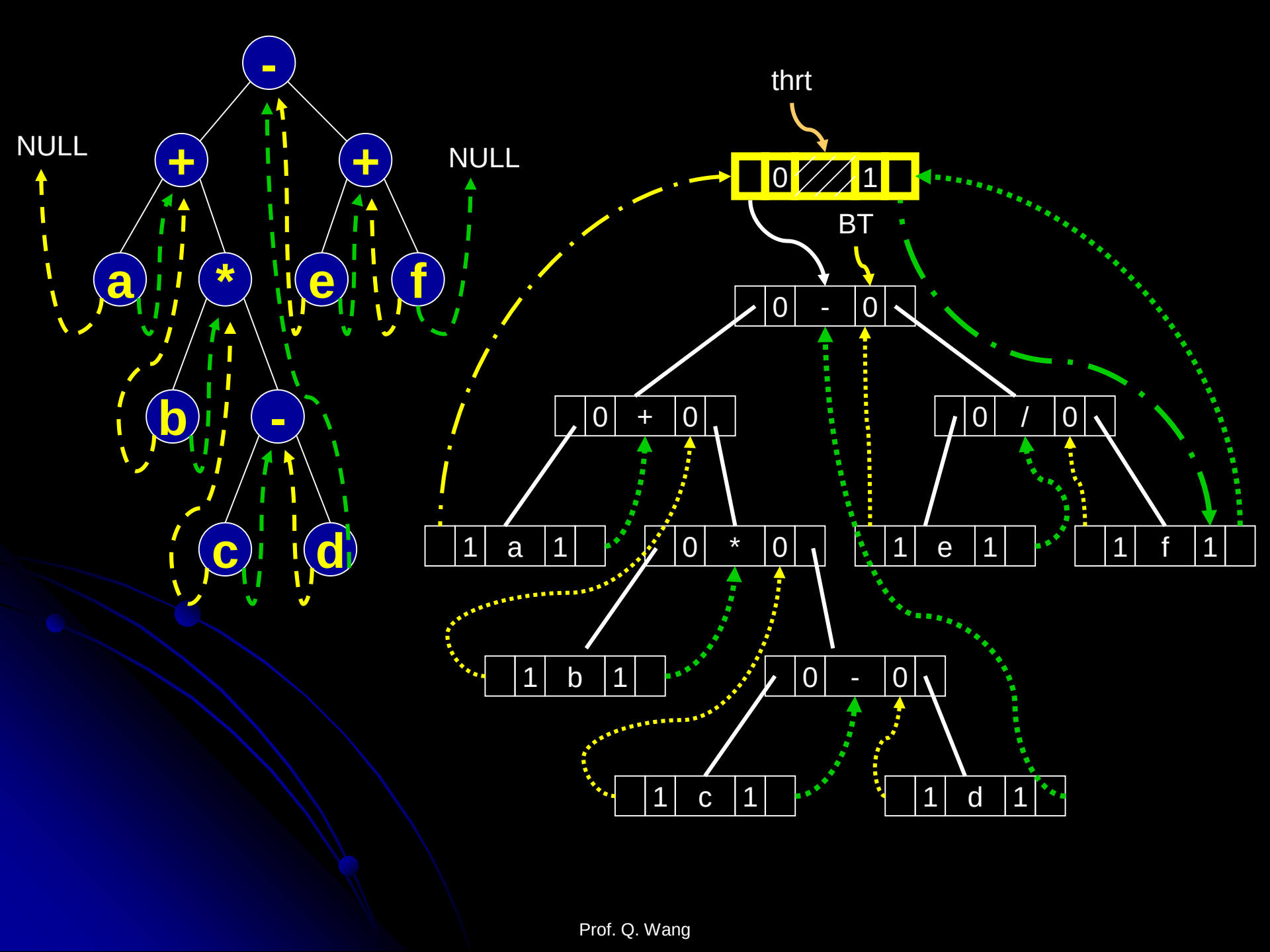

-
+
+
a
*
e
f
b
-
c
d


BT
*pre
0 - 0
0 + 0
0 / 0
^ 1 a 1
0 * 0
e
f
p
1 b
0 - 0
c
d


-
+
+
a
*
e
f
b
-
c
d
BT
0 - 0
0 + 0
0 / 0
p
^ 1 a 1
0 * 0
e
f
*pre
1 b 1
0 - 0
c
d


-
+
+
a
*
e
f
b
-
c
d


BT
0 - 0
0 + 0
*pre
0 / 0
^ 1 a 1
0 * 0
e
f
1 b 1
0 - 0
p
1 c
d


NULL
NULL
thrt
BT
0
1
0
-
0
0
+
0
0
/
0
1
a
1
0
*
0
1
e
1
1
f
1
1
b
1
0
-
0
1
c
1
1
d
1

```
Status InOrderTraverse (PBinTree T, Status (*Visit)(ElemType e))
{
    if (T){
        InOrderTraverse (T->lchild, Visit);
        (*Visit)(T-> info);
        InOrderTraverse (T->rchild, Visit);
    }
}
```

中序遍历

```
void InThreading (PThrTree p, PThrTree *pre)
{
    if (p) {
        InThreading (p->lchild, pre);
        if (!p->lchild) { p->ltag = 1;   p->lchild = *pre;}
        if (!(*pre)->rchild) {(*pre)->rtag = 1;   (*pre)->rchild = p;}
        *pre = p;
        InThreading (p->rchild, pre);
    }
}
```

中序线索化

InOrderThreading (二叉树的中序线索化非递归算法)

```
void thread (PThrTree t)
{
    Stack S;                          /*栈元素的类型为PThrTreeNode*/
    PThrTreeNode p;                   /*指向当前正在访问的结点*/
    PThrTreeNode pre;                 /*指向p的中序前驱结点*/

    if  ( !t ) return ;
    StackEmpty( S );
    p = *t;
    pre = NULL;
    do {
        while ( p )          /* 遇到结点入栈,然后处理其左子树 */
        {
            Push( S, p );
            p = p->lchild;
        }
```

```
while ( !p && !IsEmpty( S ) ) /* 左子树处理完,
{                                从栈顶退出结点并访问 */
   p = getTop( S );
   Pop( S );
   if ( !pre) {
      if ( pre->rchild == NULL)    /* 检查前驱结点的右指针 */
      {     pre->rchild = p;
            pre->rtag = 1;
       }
       if ( !p->lchild)            /* 检查该结点的左指针 */
       {      p->lchild = pre;
              p->ltag = 1;
       }
   }
   pre = p;
   p = p->rchild;                  /* 处理右子树 */
 }
} while ( p || !IsEmpty( S ) );
}
```

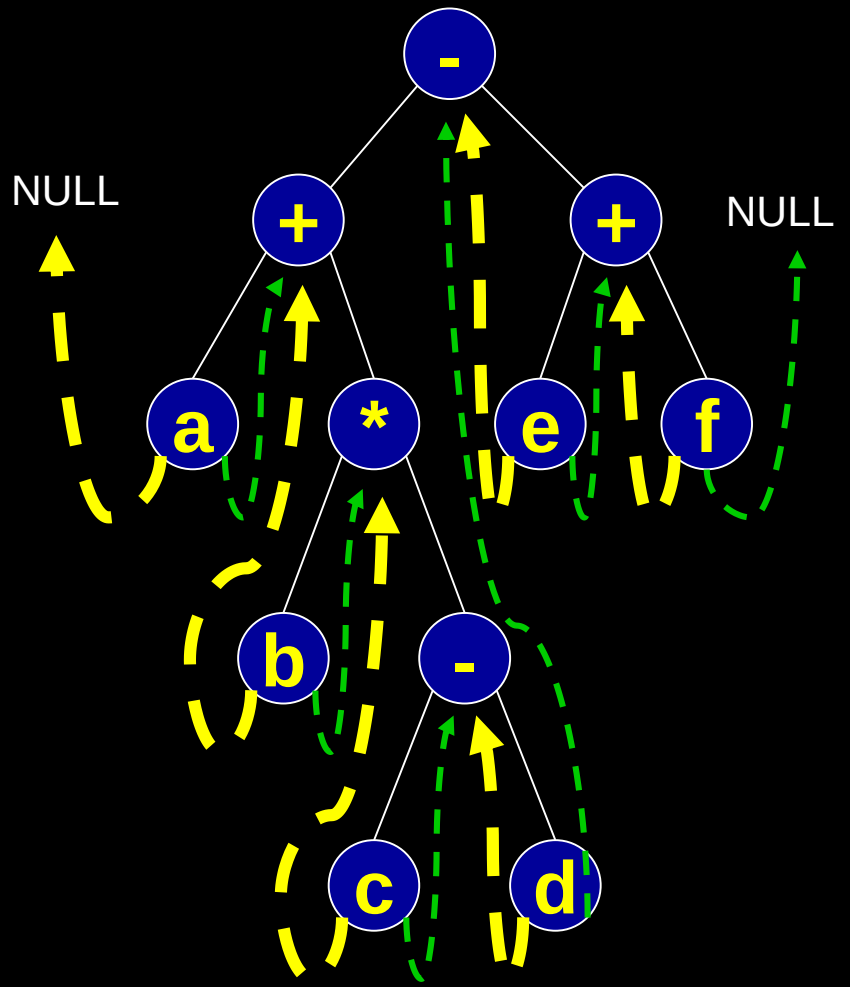
Setup Threading

在中序线索二叉树中找结点的前驱

在中序线索树中找结点前驱的规律是：

1) 若左标志是1，则左链为线索，指示其前驱；

2) 否则，遍历该结点的左子树时最后访问的结点(左子树中最右下的结点)为其前驱。

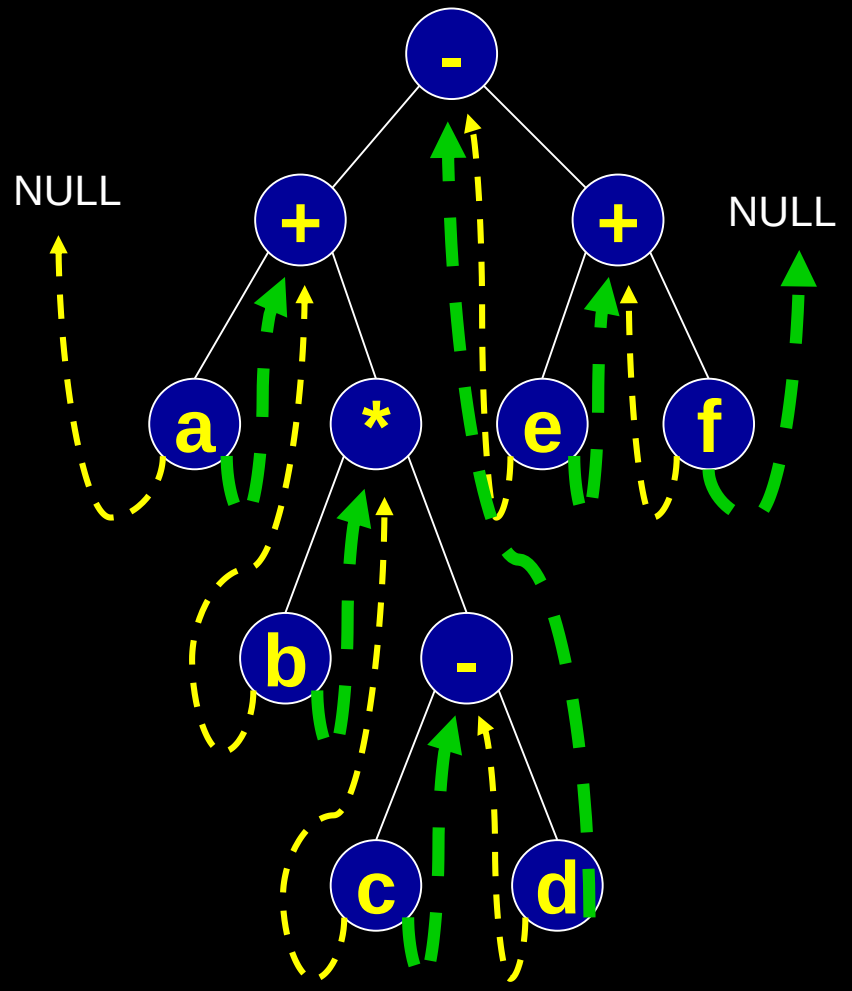


在中序线索二叉树中找结点的后继

在中序线索树中找结点**后继**的规律是：

1) 若右标志是1，则右链为线索，指示其后继；

2) 否则，遍历该结点的右子树时访问的第一个结点(右子树中最左下的结点)为其后继。



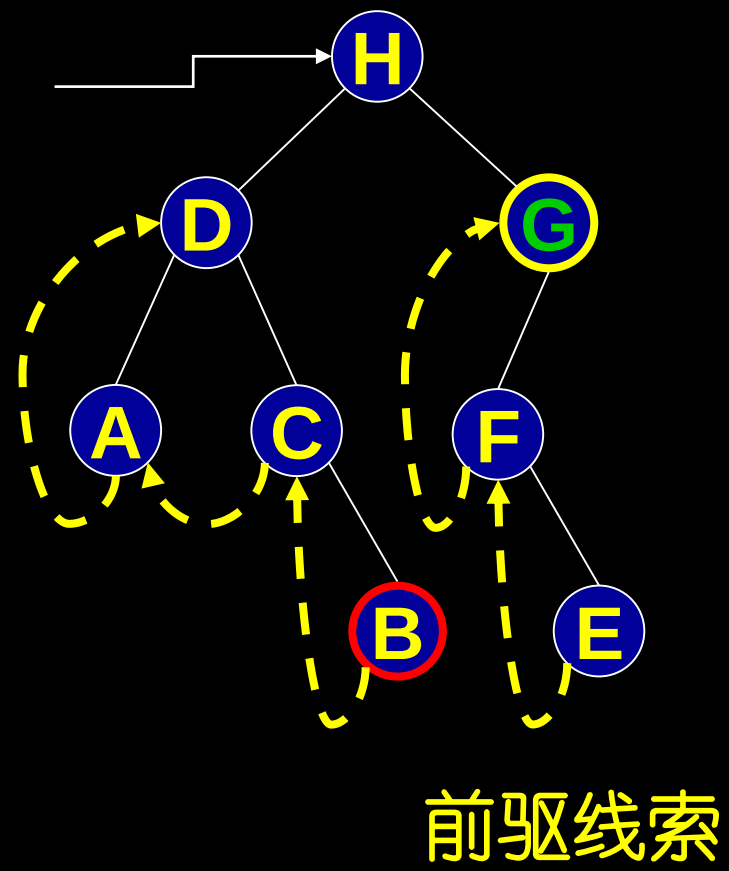
先序线索树中找结点x的前驱，较复杂。

可分三种情况：

(1) 若结点x是二叉树的根，则其前驱为空；

(2) 若结点x是其双亲的左孩子或是右孩子且其双亲没有左子树，则其前驱即为双亲结点。

(3) 若结点x是其双亲的右孩子，且其双亲有左子树，则其前驱为双亲的左子树上按先序遍历出的最后一个结点。



由此可见，在*先序*线索化树上找*前驱*时也需知道结点的双亲，因此需要使用三叉链表。

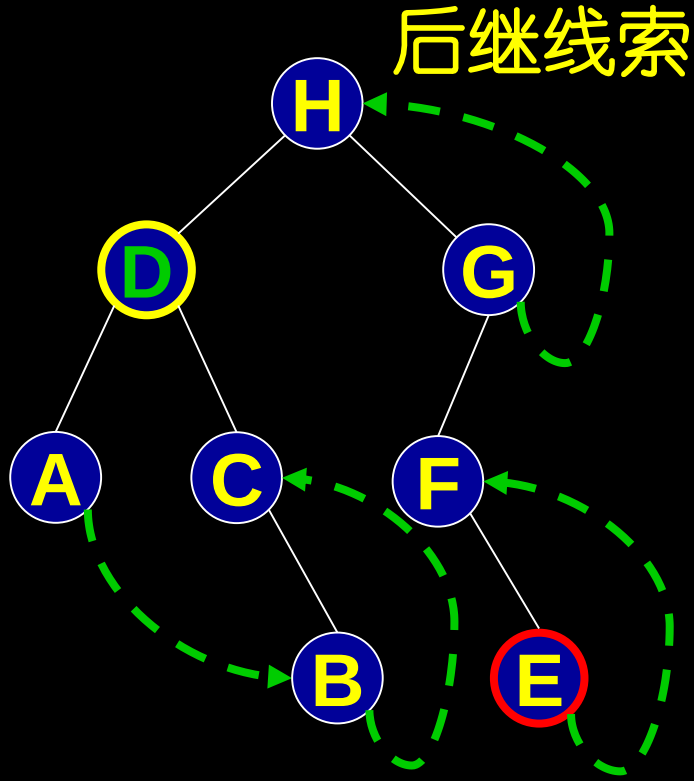
后序线索树中找结点x的后继，较复杂。

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左孩子且其双亲没有右子树，则其后继即为双亲结点。

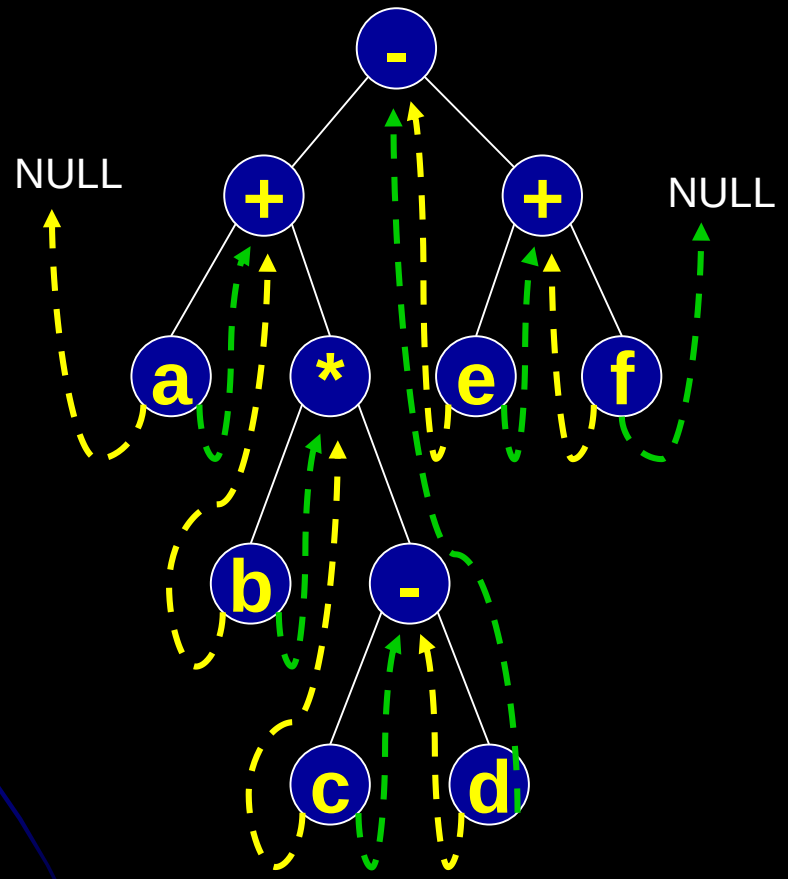
(3) 若结点x是其双亲的左孩子，且其双亲有右子树，则其后继为双亲的右子树上按后序遍历出的第一个结点。



由此可见，在*后序*线索化树上找*后继*时需知道结点的双亲，因此需要使用三叉链表。

如何中序遍历中序线索二叉树？

若在某程序中需要经常遍历或查找结点在遍历所得线性序列中的前驱和后继，则应采用线索链表作存储结构。



```
Status InOrderTraverse (PBinTree T, Status (*Visit)(ElemType e))
{
    if (T){
        InOrderTraverse (T->lchild, Visit);
        (*Visit)(T-> info);
        InOrderTraverse (T->rchild, Visit);
    }
}
```

中序遍历

```
void InThreading (PThrTree p, PThrTree *pre)
{
    if (p) {
        InThreading (p->lchild, pre);
        if (!p->lchild) { p->ltag = 1;   p->lchild = *pre;}
        if (!(*pre)->rchild) {(*pre)->rtag = 1;   (*pre)->rchild = p;}
        *pre = p;
        InThreading (p->rchild, pre);
    }
}
```

中序线索化

中序遍历中序线索二叉树

```
void InOrderTravse_Thr (PThrTree t)
{
    PThrTreeNode  p;
    if ( !t ) return ;
    p = *t;
    while (p->lchild!=NULL && p->ltag==0)       /* 顺左子树一直向下 */
        p = p->lchild;
    while ( !p )  {
        visit(p);                          /* 访问该结点， printf(p->info);*/
        if ( p->rchild && p->rtag==0) {      /* 右子树不是线索时 */
            p = p->rchild;
            while ( p->lchild && p->ltag==0)
                p = p->lchild;              /* 顺右子树根结点的左子树一直向下 */
        }
        else
            p = p->rchild;                 /* 顺线索向下 */
    }
}
```

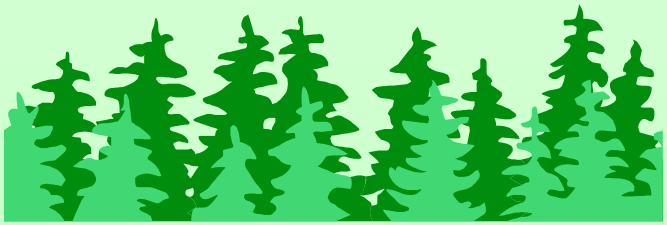
6.4

树、森林和二叉树的关系

1 树的存储结构

2 树、森林与二叉树的相互转换

3 树、森林的遍历



6.4.1 树的三种存储结构

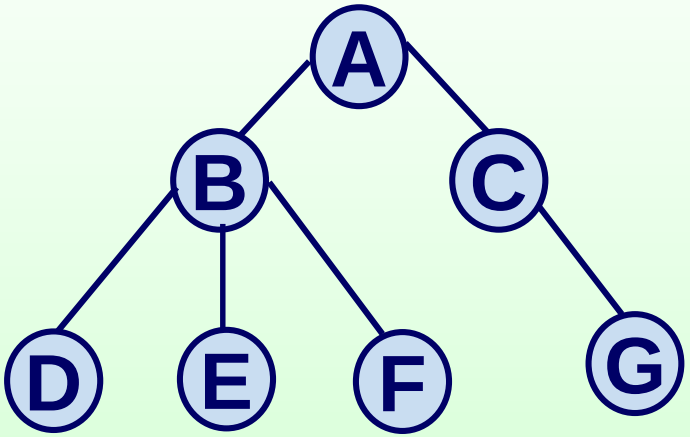
一、双亲表示法

二、孩子表示法

三、孩子兄弟表示法

一、双亲表示法:

以一组连续的空间存放树的结点，同时在每个结点上附设一个指示器指示其双亲结点的位置。



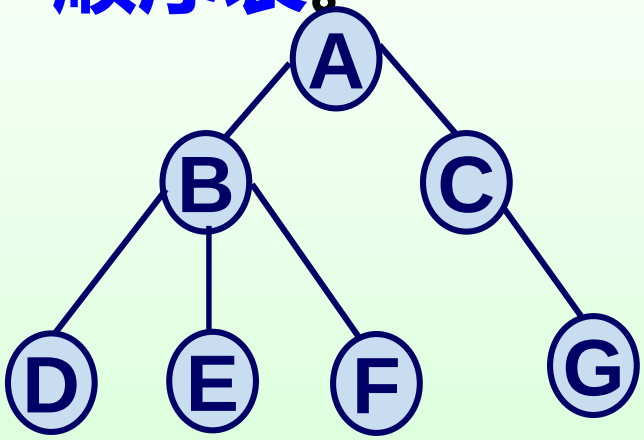
data	parent

结点信息　双亲位置

节点序号	data	parent
0	A	-1
1	B	0
2	C	0
3	D	1
4	E	1
5	F	1
6	G	2

二、孩子表示法:

将每个结点的孩子结点链接构成一个单链表，称为**孩子链表**。将**孩子链表的头指针**又组成了一个**顺序表**。

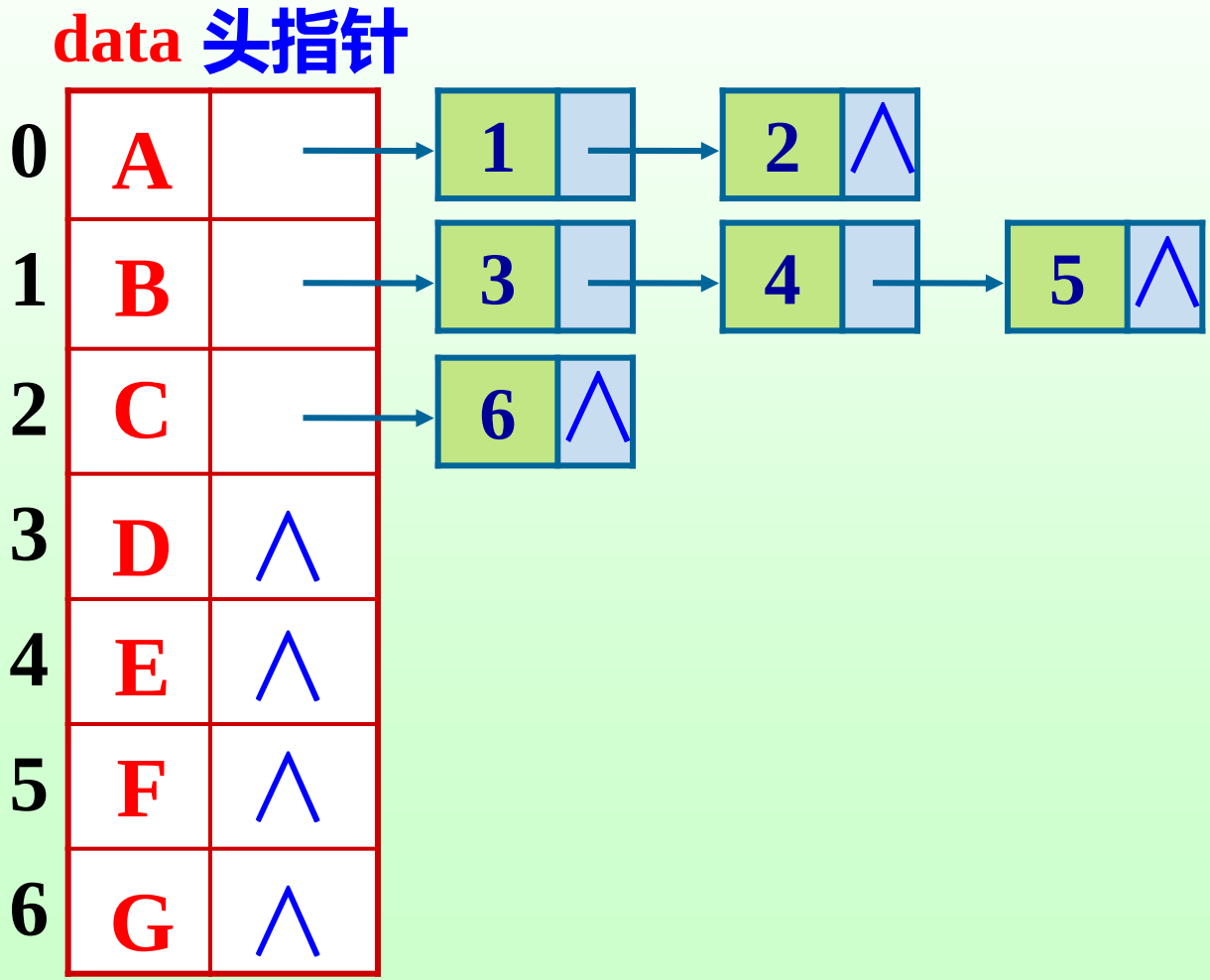


data	FirstChild

孩子链表头指针

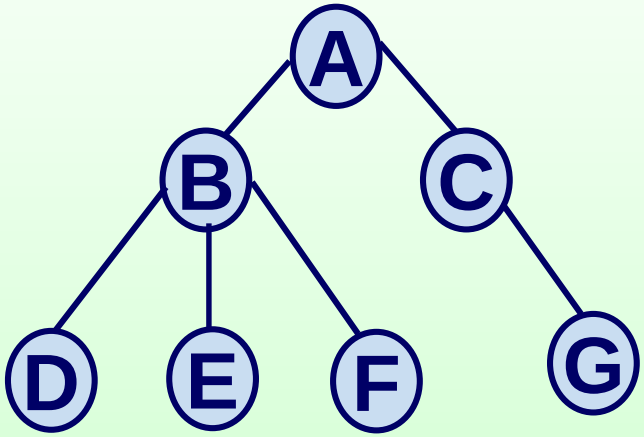
Child	next

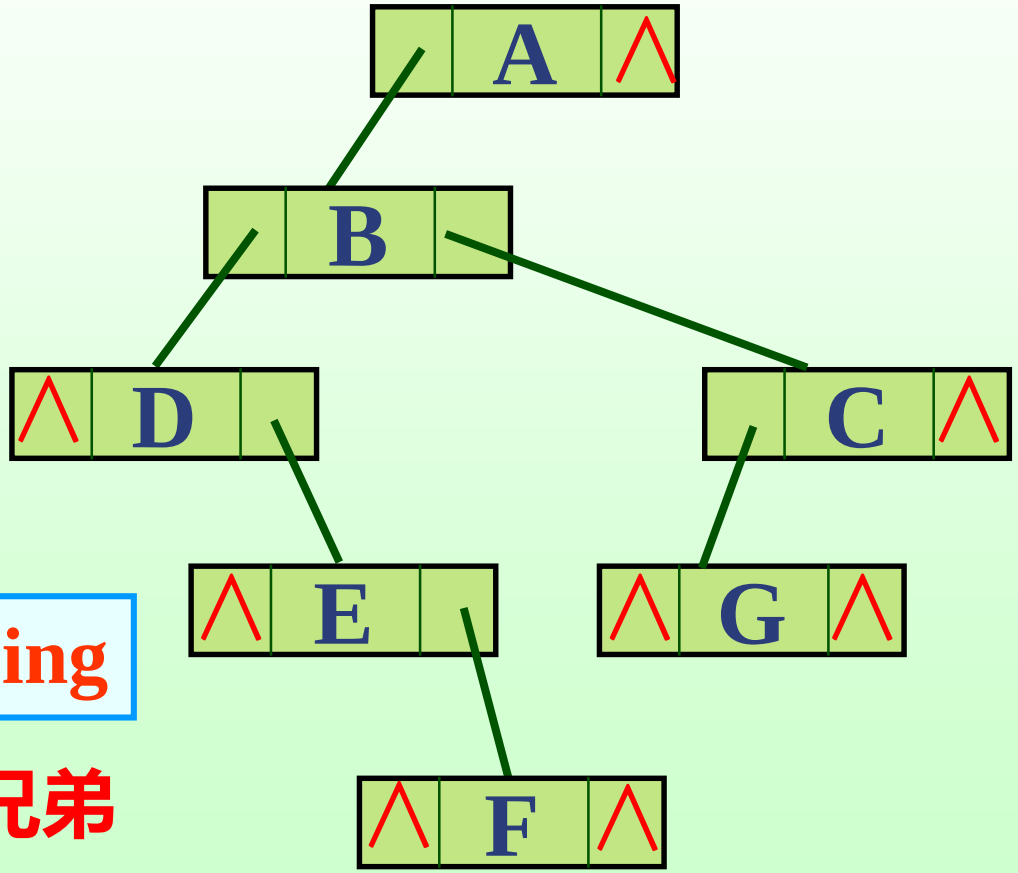
位置 下一个孩子



三、孩子兄弟表示法:

每个结点设有两个指针域，分别指向该结点的第一个孩子和下一个兄弟（右兄弟），又称为树的二叉链表表示法。





FirstChild	data	NextSibing

第一个孩子 结点信息 下一个兄弟

6.4.2 树、森林与二叉树的转换

树与二叉树均可用二叉链表作为存储结构，则以二叉链表为媒介可导出树与二叉树之间的一个对应关系——即给定一棵树，可以找到唯一一棵二叉树与之对应。

【注意】：和树对应的二叉树，其左、右子树的概念已改变为： 左是孩子，右是兄弟。

左孩子 右孩子

第一个孩子 下一个兄弟

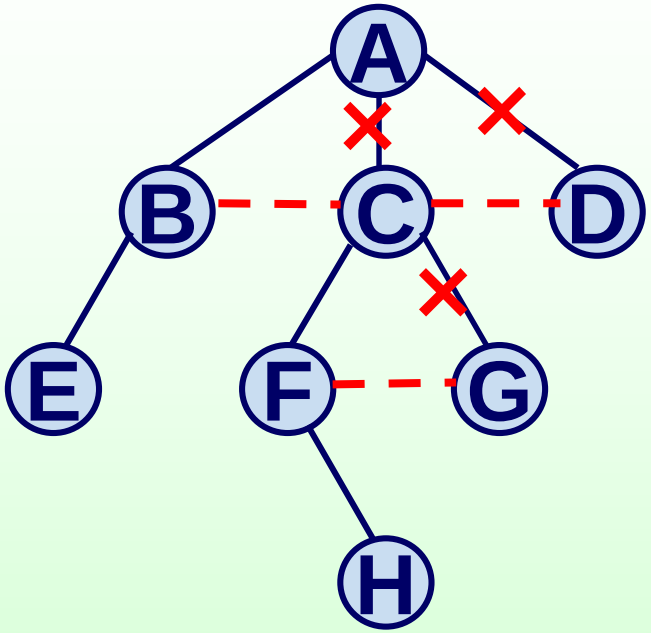
一、树 ⟶ 二叉树

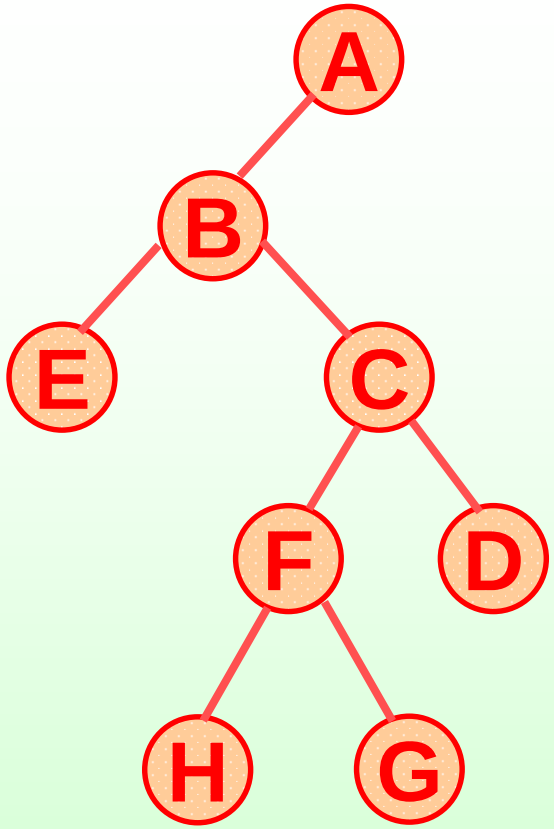
1、**加线**：在各亲兄弟之间加一虚线。

2、**抹线**：抹掉（除第一个孩子外）该结点到其余孩子之间的连线。

3、**旋转**：新加上去的虚线改实线且均向右斜（rchild），原有的连线均向左斜（lchild），使之结构层次分明。

树→二叉树举例：





3、**旋转：** 新加上去的**虚线**改实线且均向右斜（rchild），**原有的连线**均向左斜（lchild），使之结构层次分明。

二、森林 ⟶ 二叉树

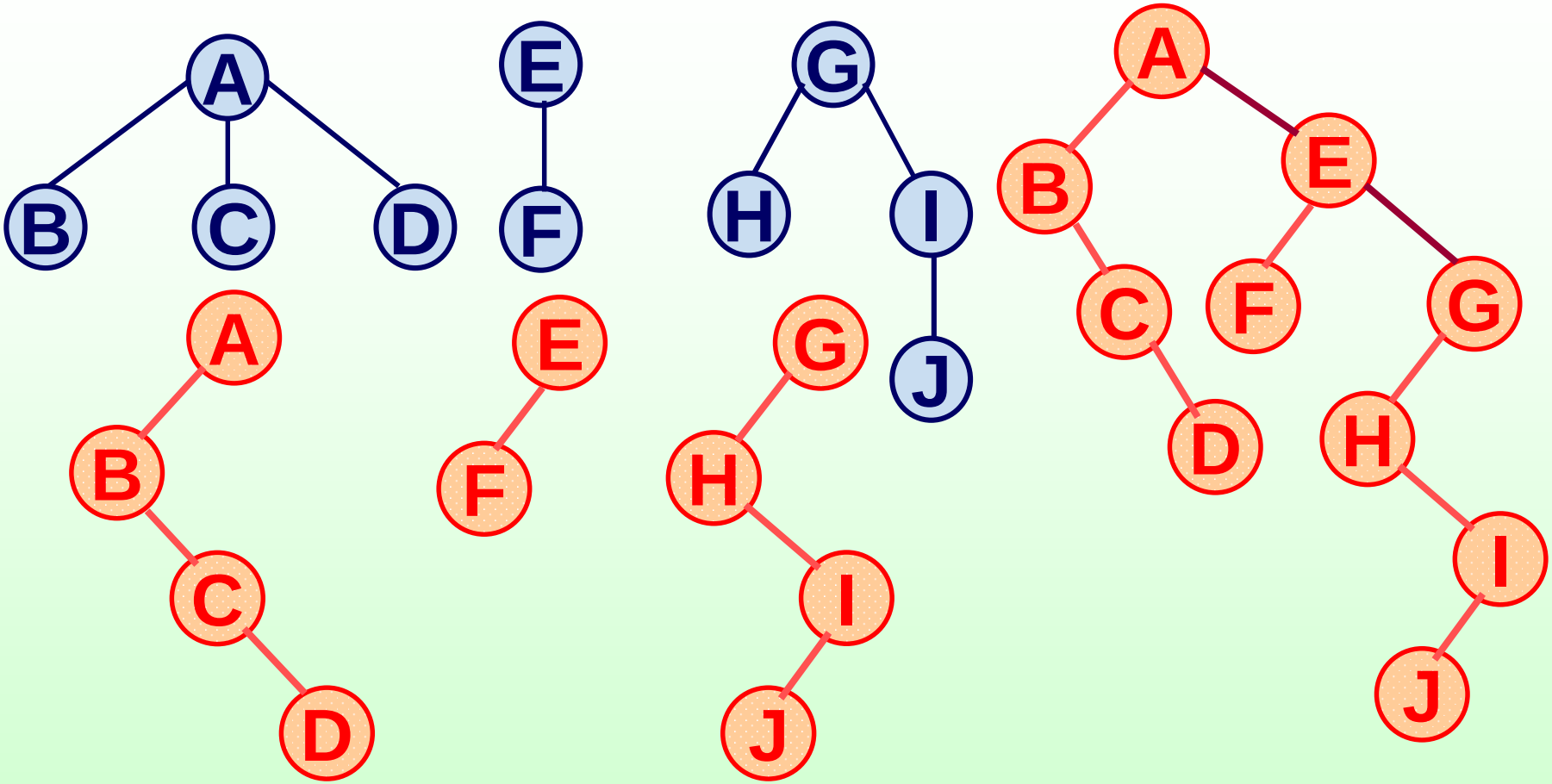
从树的二叉链表表示可知，任何一棵和树对应的二叉树，其右子树必空。若把森林中第二棵树的根结点看成是第一棵树根结点的兄弟，则同样可以导出森林和二叉树的对应关系。

二、森林 ⟶ 二叉树

1、将各棵树分别转换为二叉树。

2、按给出森林中树的次序，依次将后一棵二叉树作为**前一棵二叉树根结点的右子树**，则第一棵树的根结点是转换后二叉树的根。

森林→二叉树举例：



2、按给出森林中树的次序，依次将后一棵二叉树作为**前一棵二叉树根结点的右子树**，则第一棵树的根结点是转换后二叉树的根。

三、二叉树 ⟶ 树

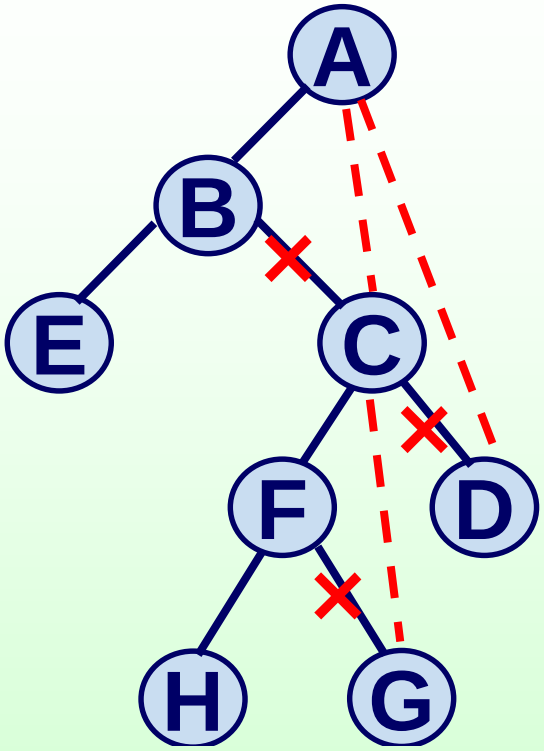
前提：二叉树的根结点无右孩子

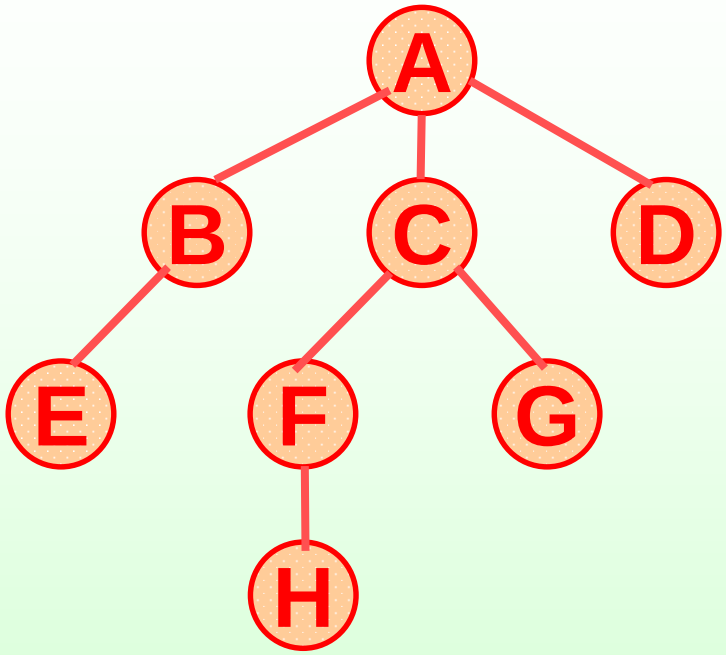
1、**加线**：若某结点i是双亲结点的左孩子，则将该结点的右孩子以及当且仅当连续地沿着此右孩子的**右链**不断搜索到的所有右孩子都分别与结点i的**双亲用虚线连起来**。

2、**抹线**：抹掉原二叉树中所有双亲结点与右孩子的连线。

3、**归整化**：将图形归整化，使各结点**按层次**排列且将加上去的虚线变成实线。

二叉树→树举例：





3、**归整化**：将图形归整化，使各结点**按层次**排列且将加上去的虚线变成实线。

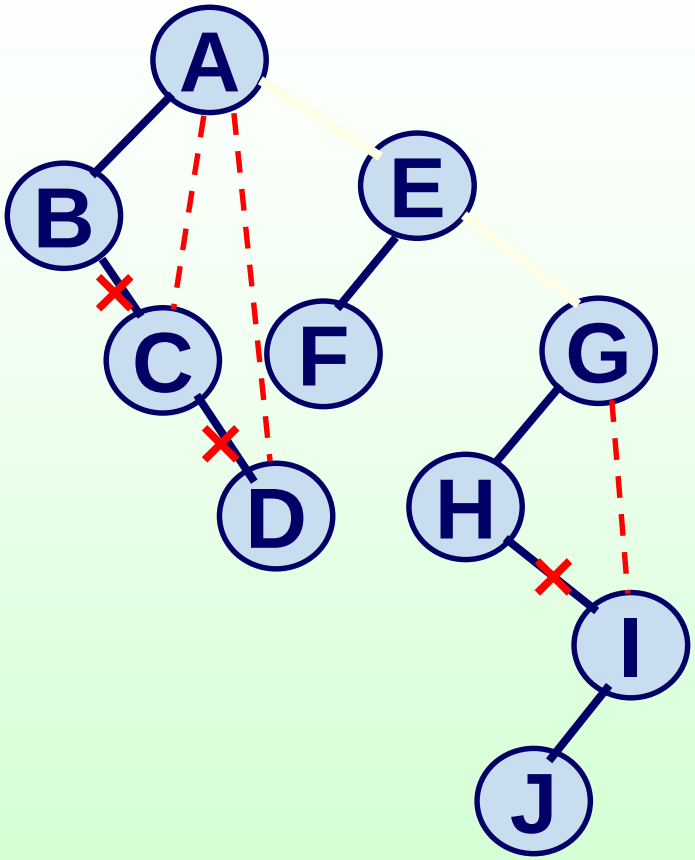
四、二叉树 ⟶ 森林

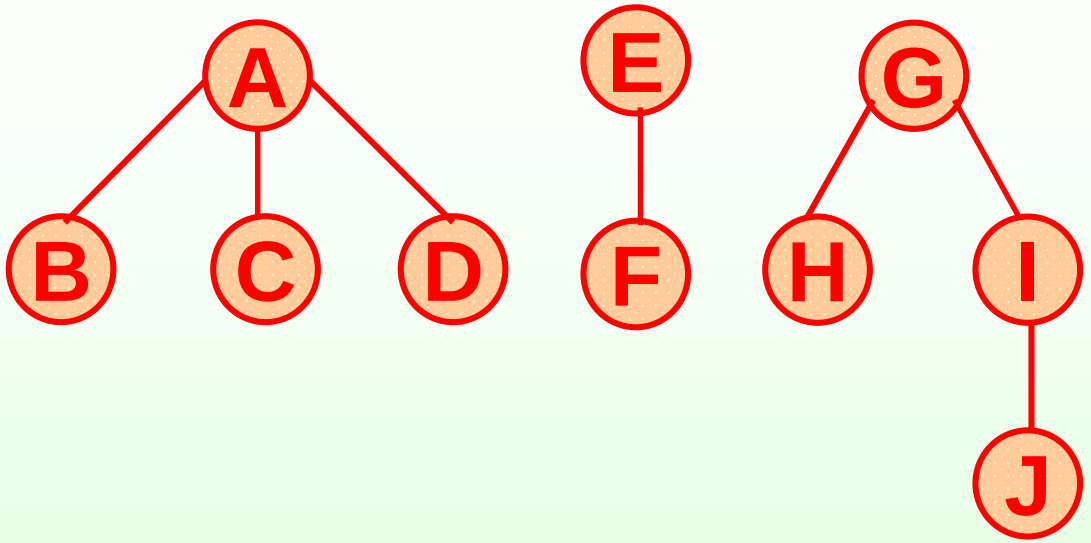
前提：二叉树的根结点必有右孩子

1、**抹线**：将二叉树的根结点与其右孩子 i 的连线以及当且仅当连续地沿着 i 的右链不断搜索到的所有右孩子间的连线全部抹掉，这样得到若干 棵孤立的二叉树。

2、**还原**：将各棵孤立的二叉树按二叉树还原为一般树的方法还原为树。

二叉树→森林举例：





2、**还原**：将各棵孤立的二叉树按二叉树还原为一般树的方法还原为树。

6.4.3 树和森林的遍历

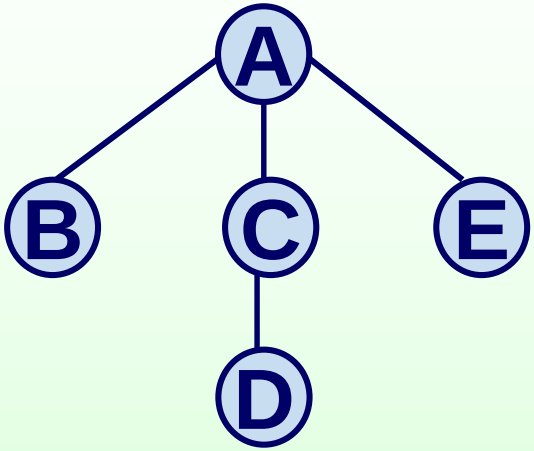
遍历树（Traversal of Tree）的方法：

一、**先根遍历（深度方向）**：先访问树的根结点，然后依次先根遍历根的每棵子树.

二、**后根遍历（深度方向）**：先依次后根遍历每棵子树，然后访问根结点.

三、**层次序列（广度方向）**

树的遍历 举例



先根遍历序列： A B C D E

后根遍历序列： B D C E A

森林的遍历方法

(1) 先根遍历

若森林F为空, 返回；否则 ：

访问F的第一棵树的根结点；

先根次序遍历第一棵树的子树森林；

先根次序遍历其它树组成的森林。

(2) 中根遍历（实际就是后根遍历每棵树）

若森林F为空，返回；否则：

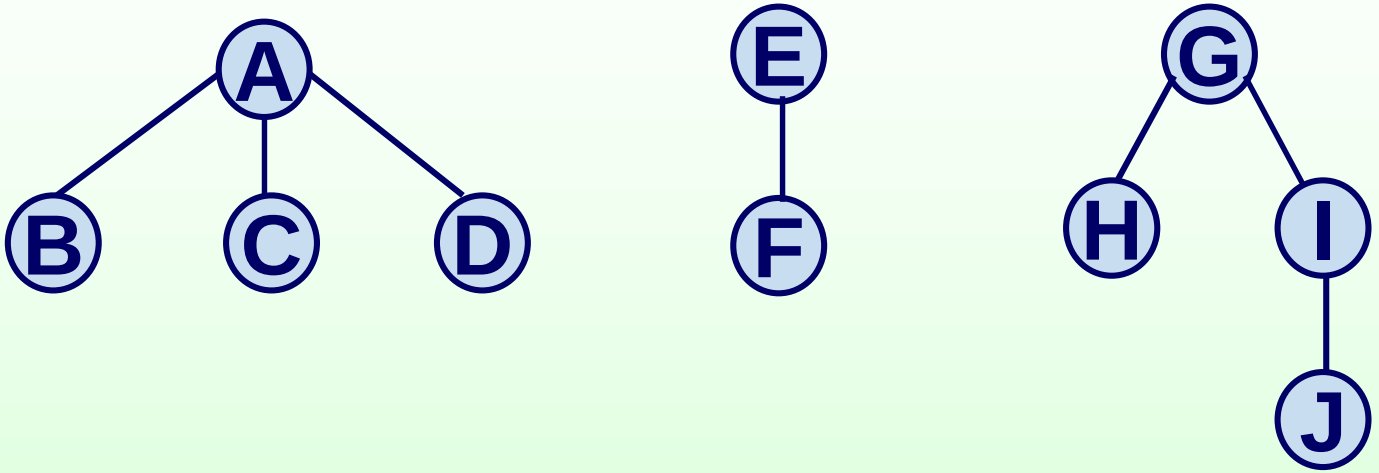
中根次序遍历第一棵树的子树森林；

访问F的第一棵树的根结点；

中根次序遍历其它树组成的森林。

(3) 后根遍历

森林的遍历 举例



先序序列： A B C D E F G H I J

中序序列： B C D A F E H J I G

End of 6.4