数据结构

北京邮电大学 信息安全中心
武 斌 杨 榆

上次课内容

上次课（树和二叉树(上)）内容：

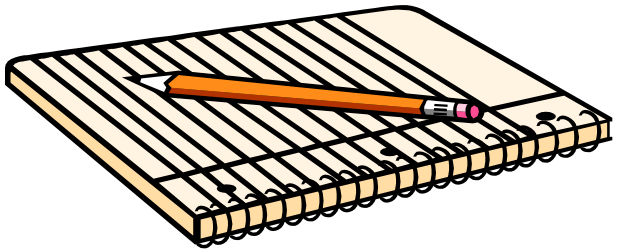
- 领会树和二叉树的类型定义，理解树和二叉树的结构差别
- 熟记二叉树的主要特性，并掌握它们的证明方法
- 熟练掌握二叉树和树的各种存储结构及其建立的算法
- 学会编写实现树的各种操作的算法

本次课程学习目标

学习完本次课程，您应该能够：

- 熟练掌握二叉树的各种遍历算法，并能灵活运用遍历算法实现二叉树的其它操作
- 理解二叉树的线索化过程以及在中序线索化树上找给定结点的前驱和后继的方法
- 学习树和森林的关系及转换方法
- 了解最优树的特性，掌握建立最优树和赫夫曼编码的方法

遍历二叉树和线索二叉树



- 6.1 树的定义和基本术语
- 6.2 二叉树
- 6.3 遍历二叉树和线索二叉树
- 6.4 树和森林
- 6.5 赫夫曼树及其应用

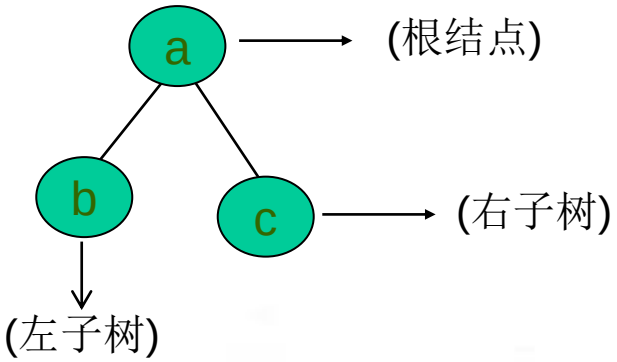
遍历二叉树

●一、遍历二叉树

对线性结构，只有一条搜索路径。而二叉树是非线性结构，每个结点有两个后继，存在按什么样的搜索路径遍历的问题。

二叉树的一些应用常常要求在树中查找具有某种特征的结点，或者对树中全部结点逐一进行某种处理。

由此引入了**遍历二叉树**的问题，即**如何按某条搜索路径巡访树中的每一个结点，使得每一个结点均被访问一次，而且仅被访问一次**。

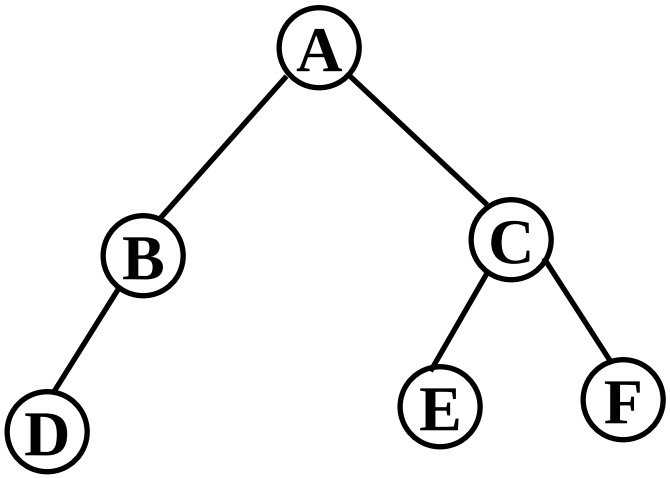


由二叉树的递归定义，二叉树的三个基本组成单元是：根结点、左子树和右子树。

若能依次遍历这三部分，便是遍历整个二叉树。

遍历二叉树

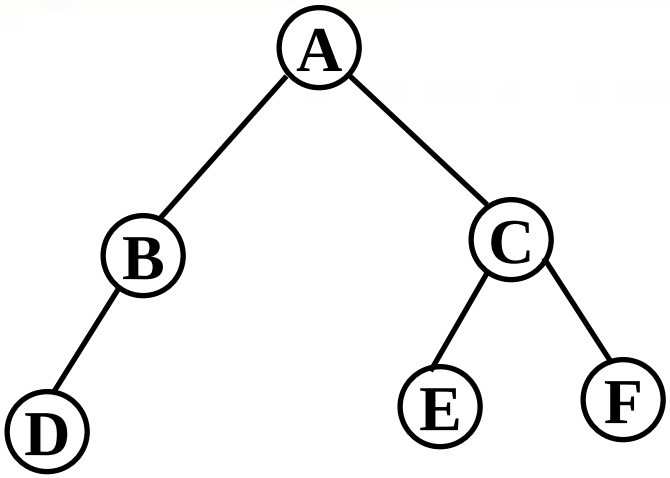
- 假如以L、D、R分别表示遍历左子树、遍历根结点和遍历右子树，遍历整个二叉树则有DLR、LDR、LRD、DRL、RDL、RLD六种遍历方案。若规定先左后右，则只有前三种情况，分别规定为：
  - DLR——先（根）序遍历，
  - LDR——中（根）序遍历，
  - LRD——后（根）序遍历。



- 1、先序遍历二叉树的操作定义为：

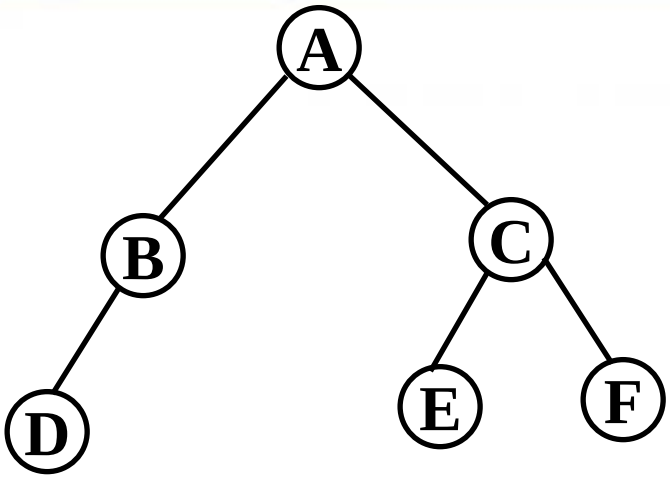
  若二叉树为空，则空操作；否则
  - （1）访问根结点；
  - （2）先序遍历左子树；
  - （3）先序遍历右子树。
  - 演示6-4-2-1.swf

遍历二叉树



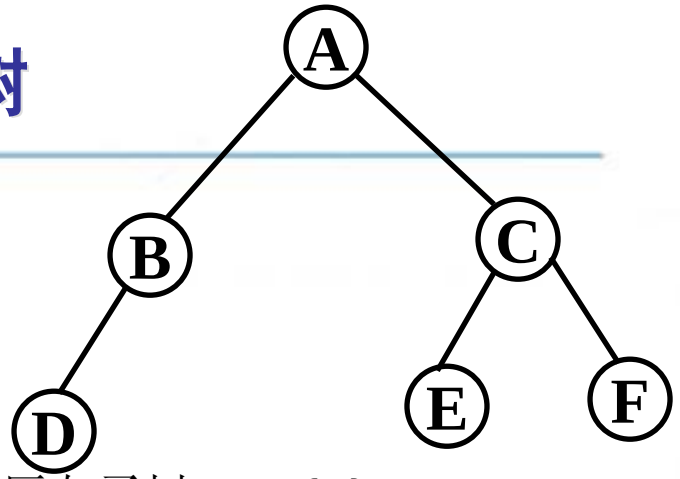
- 例：按先序遍历图示二叉树：
- 1.二叉树T={A,B,C,D,E,F}，根为A，左子树为T1={B,D}，右子树为T2={C,E,F}。按先序定义，首先访问根结点A。
- 2.按先序遍历左子树T1={B,D}。
- 3.左子树T1={B,D}，根为B，其左子树为T11={B}，右子树为T12空树。按先序定义，首先访问根结点B。
- 4.按先序遍历左子树T11={D}。
- 5.左子树T11={D}，根为D，其左子树为T111为空树，右子树为T112空树。按先序定义，首先访问根结点D。
- 6.按先序遍历左子树T111。T111为空树，不做操作。 T11={D}的左子树遍历完毕，按先序遍历其右子树T112。T112也是空树，不做操作。
- 至此，完成对子树T11={D}的先序遍历。T11={D}是T1={B,D}的左子树，按先序，接下来应该先序遍历T1={B,D}的右子树T12。

遍历二叉树



- 例：按先序遍历图示二叉树：
- 7.T12是空树，不需要操作。按先序可知，至此已完成了T1={B,D}遍历，后续应该访问其双亲的右子树T2={C,E,F} 。
- 8.右子树T2={C,E,F} ，根为C，其左子树为T21={E}，右子树为T21={F} 。按先序定义，首先访问根结点C。
- 9.按先序遍历左子树T21={E} 。
- 10.左子树T21={E} ，根为E，其左子树为T211为空树，右子树为T212空树。按先序定义，首先访问根结点E。左右子树均不操作。
- 11.至此已完成了T21={E}遍历，后续应该访问其双亲的右子树T22={F} 。
- 12.右子树T22={F} ，根为F，其左子树为T221为空树，右子树为T222空树。按先序定义，首先访问根结点F。左右子树均不操作。
- 至此，遍历结束。

遍历二叉树



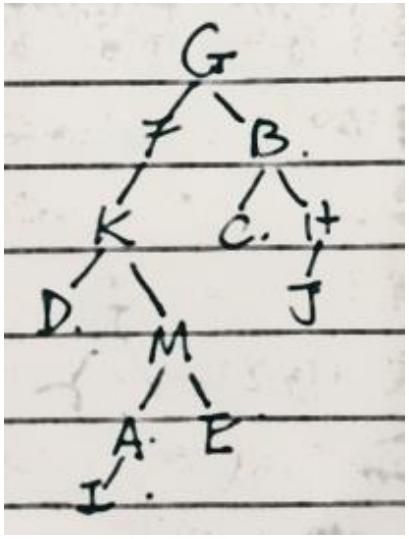
- 例：按**先序遍历**图示二叉树：
- 归纳，先序遍历操作为：
- 1.**访问**树T={A,B,C,D,E,F}的**根结点A**。
- 2.先序遍历左子树T1={B,D} 。
- 3.**访问**树T1={B,D}的**根结点B**。
- 4.先序遍历左子树T11={D} 。
- 5.**访问**树T11={D}的**根结点D**。
- 6.先序遍历左子树T111={} 。
- 7.先序遍历右子树T112={}。
- 8.先序遍历右子树T12={}。
- 9.先序遍历右子树T2={C,E,F}
- 11.**访问**树T2={C,E,F}的**根结点C**。
- 12.先序遍历左子树T21={E} 。
- 13.**访问**树T21={E}的**根结点E**。
- 14.先序遍历左子树T211={} 。
- 15.先序遍历右子树T212={} 。
- 16.先序遍历右子树T22={F} 。
- 17.**访问**树T22={F}的**根结点F**。
- 18.先序遍历左子树T221={} 。
- 19.先序遍历右子树T222={} 。
- 先序遍历访问序列为：A,B,D,C,E,F

遍历二叉树

- 练习

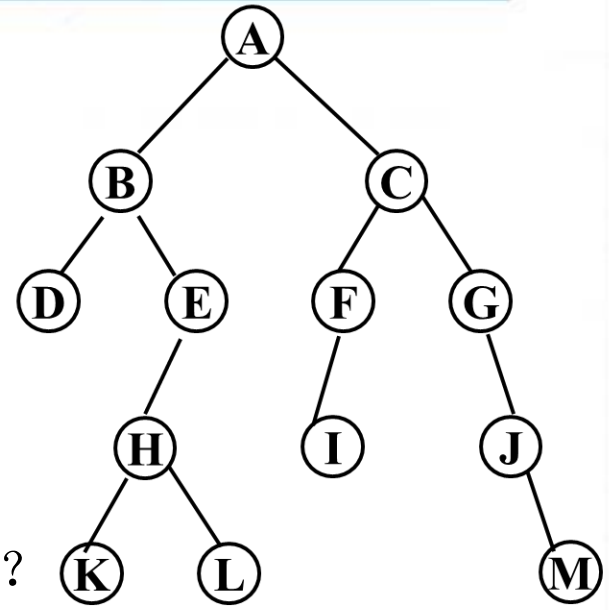
1. 画出和下列已知序列对应的二叉树T：

- 树的先根次序访问序列为：GFKDMAIEBCHJ;
- 树的后根次序访问序列为：DIAEMKFCJHBG。
- 解答如下（注意，二叉树不唯一）



遍历二叉树

- 已知二叉树如下，请回答问题：
- 其后序遍历序列为？
- 中序遍历顺序下，节点E和L的后继结点是？
- 先序遍历顺序下，节点D和G的前驱结点是？
- 后序遍历顺序下，节点K和F的前驱和后继分别是？

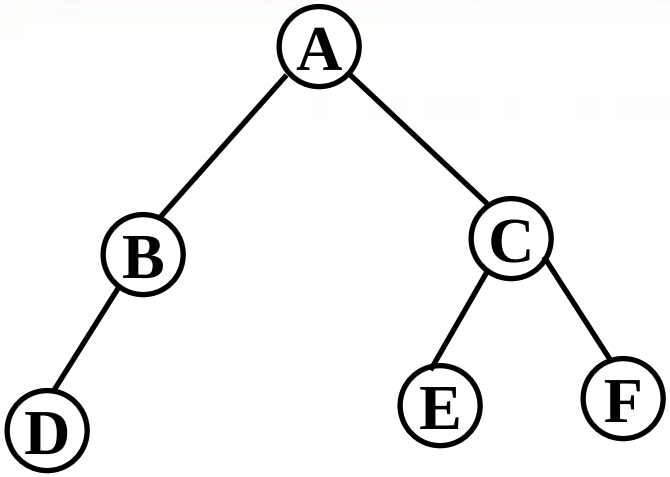


遍历二叉树

●2、**中序遍历二叉树**的操作定义为：

若二叉树为空，则空操作；否则

- （1）中序遍历左子树；
- （2）访问根结点；
- （3）中序遍历右子树。



- 演示6-4-2-2.swf,例图示二叉树中序遍历序列为：D,B,A,E,C,F

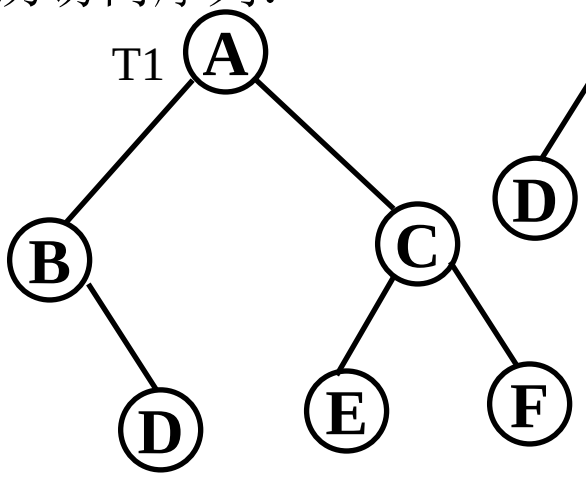
●3、**后序遍历二叉树**的操作定义为：

若二叉树为空，则空操作；否则

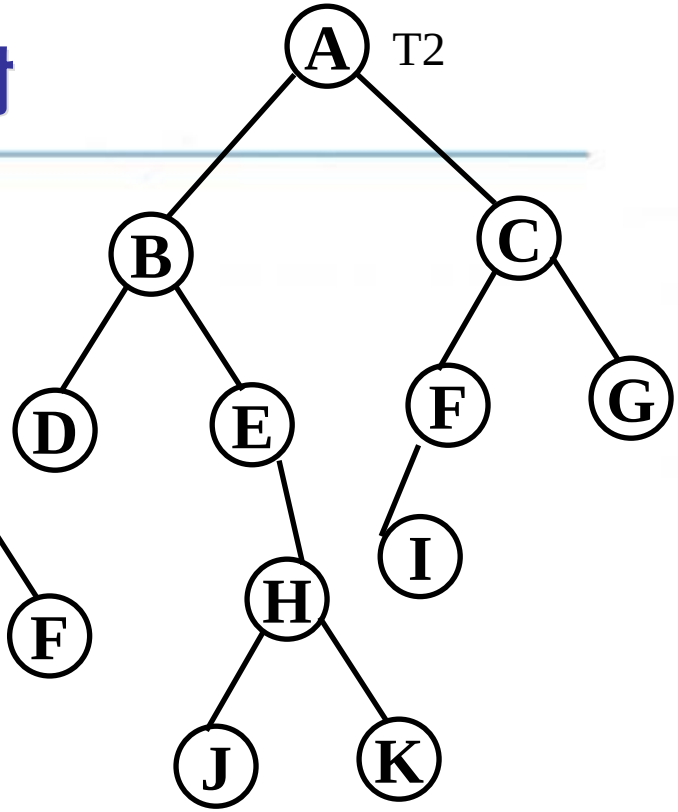
- （1）后序遍历左子树；
- （2）后序遍历右子树；
- （3）访问根结点。
- 演示6-4-2-3.swf，例图示二叉树后序遍历序列为：D,B,E,F,C,A

遍历二叉树

- 练习，给出下列二叉树的遍历访问序列：



- 树T1访问序列：
  - 先序：A,B,D,C,E,F
  - 中序：B,D,A,E,C,F
  - 后序：D,B,E,F,C,A



- 树T2访问序列：
  - 先序：A,B,D,E,H,J,K,C,F,I,G
  - 中序：D,B,E,J,H,K,A,I,F,C,G
  - 后序：D,J,K,H,E,B,I,F,G,C,A

遍历二叉树

- 例如右图所示的二叉树表达式

  (a+b*(c-d))-e/f

  - 若先序遍历此二叉树,按访问结点的先后次序将结点排列起来,其先序序列为：

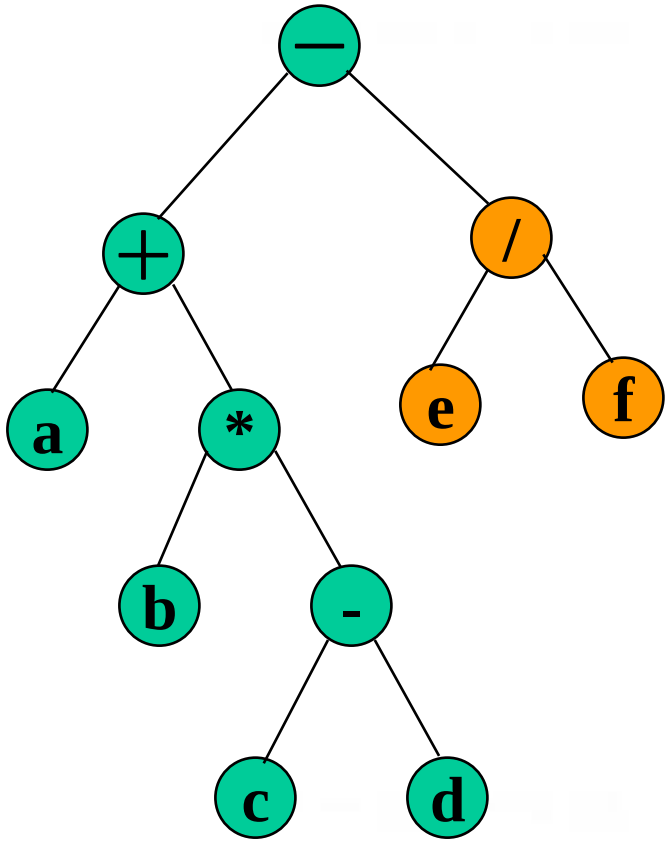
    -+a*b-cd/ef

  - 按中序遍历,其中序序列为：

    a+b*c-d-e/f

  - 按后序遍历,其后序序列为：

    abcd-*+ef/-



遍历二叉树

●**算法6.1先序遍历二叉树的递归实现**

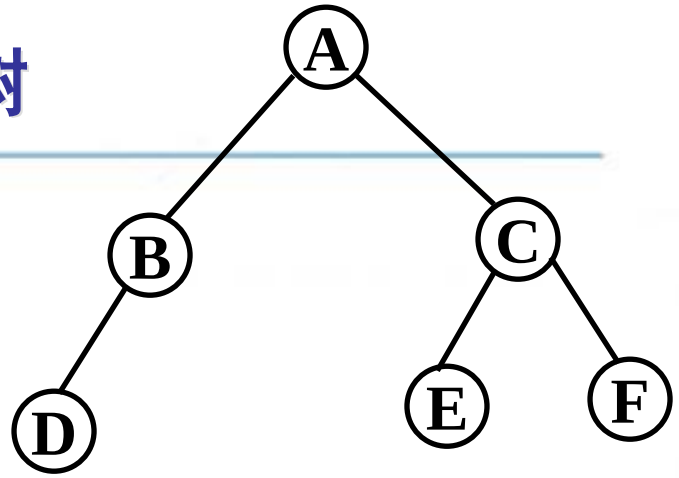
```
Status PreOrderTraverse(BiTree T，Status(* Visit)(TElemType e ))
{// 采用二叉链表存储结构，Visit是对数据元素操作的应用函数，先序遍历
  以T为根指针的二叉树的递归算法，对每个数据元素调用函数Visit。
  if (T) {        // T=NULL时，二叉树为空树，不做任何操作
    if (Visit(T->data))                    // 通过函数指针 *visit 访问根结点
        if(PreOrderTraverse(T->Lchild, Visit))  // 先序遍历左子树
            if(PreOrderTraverse(T->Rchild, Visit)) return OK;  // 先序遍
              历右子树
    return ERROR;
  } else   return OK;
}// PreOrderTraverse
```

遍历二叉树

●算法:中序遍历二叉树的递归实现

```
Status InOrderTraverse(BiTree T，Status(* Visit)(TElemType e ))
{// 采用二叉链表存储结构，Visit是对数据元素操作的应用函数，中序遍历
  以T为根指针的二叉树的递归算法，对每个数据元素调用函数Visit。
   if (T) {        // T=NULL时，二叉树为空树，不做任何操作
     if (InOrderTraverse(T->Lchild, Visit)) // 中序遍历左子树
          if(Visit(T->data)) // 通过函数指针 *visit 访问根结点
               if(InOrderTraverse(T->Rchild, Visit) return OK;  // 中序遍历
                  右子树
     return ERROR;
   } else   return OK;
}// InOrderTraverse
```

遍历二叉树



- 例：分析中序遍历图示二叉树递归过程

```
if (T) { //T:NULL
//…
}else return OK;
```

T为NULL，空树，不操作，应退回到上一层。

```
InOrderTraverse(D->lchild, Visit)//T:D
Visit(D->data)
InOrderTraverse(D->rchild, Visit)
```

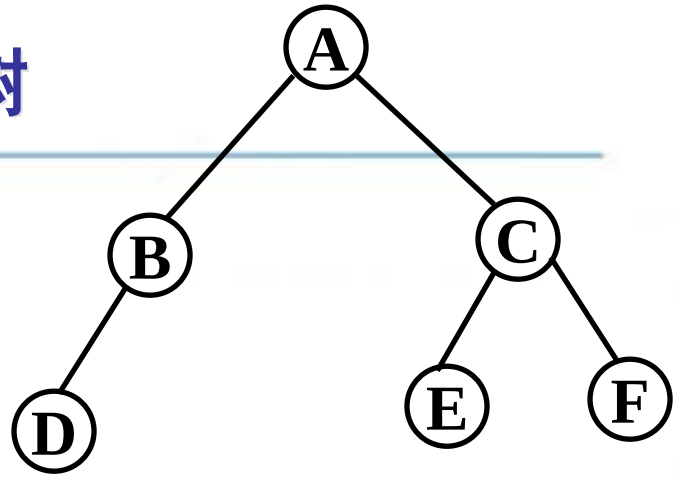
若从左子树返回，应访问当前栈记录指针（T）所指根结点。

```
InOrderTraverse(B->lchild, Visit)//T:B
Visit(B->data)
InOrderTraverse(B->rchild, Visit)
```

```
InOrderTraverse(A->lchild, Visit)//T:A
Visit(A->data)
InOrderTraverse(A->rchild, Visit)
```

```
InOrderTraverse(A, Visit)
```

遍历二叉树



- 例：分析中序遍历图示二叉树递归过程

if (T) { //T:NULL //… }else return OK;

InOrderTraverse(D->lchild, Visit)//T:D
Visit(D->data)
InOrderTraverse(D->rchild, Visit)

InOrderTraverse(B->lchild, Visit)//T:B
Visit(B->data)
InOrderTraverse(B->rchild, Visit)

InOrderTraverse(A->lchild, Visit)//T:A
Visit(A->data)
InOrderTraverse(A->rchild, Visit)

InOrderTraverse(A, Visit)

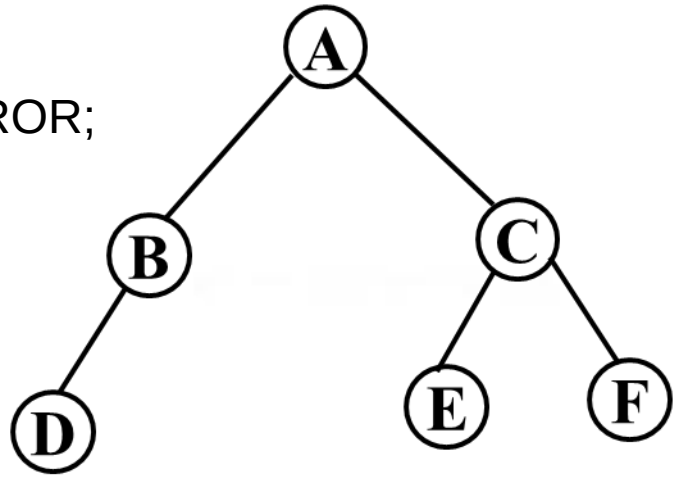
T为NULL，空树，不操作，应退回到上一层。

若从右子树返回，表明当前层遍历结束，应继续退栈。

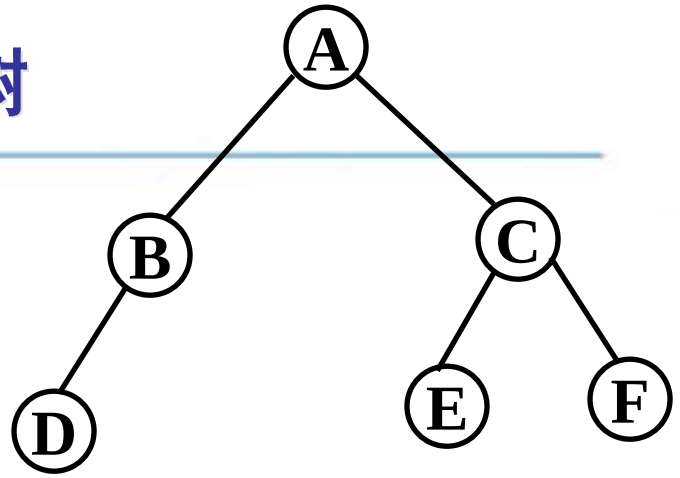
遍历二叉树——中序遍历非递归算法1

- 算法6.2 中序遍历二叉树的非递归实现（方法1）

```
Status InOrderTraverse (BiTree T, Status (* Visit)(TElemType e))
{ // 采用二叉链表存储结构，Visit是对数据元素操作的应用函数。
  // 中序遍历二叉树T的非递归算法，对每个数据元素调用函数Visit。
    InitStack (S); Push(S,T);
    while (!StackEmpty(S)){
        while (GetTop(S,p)&&p)  Push(S, p->lchild);   // 向左走到尽头
        Pop(S,p);                    // 空指针退栈
        if (!StackEmpty(S)){     // 访问结点，向右一步
            Pop(S,p);   if (!Visit (p->data))  return ERROR;
            Push(S,p->rchild);
        }// if
    }// while
    return OK;
}// InOrderTraverse
```



遍历二叉树



- 例：分析**中序遍历**图示二叉树非递归过程

若从左子树返回，应访问当前栈记录指针（T）所指根结点。

Push(S,D->rchild);	S:A,B,NULL
Pop(S,p);Visit (D->data);	S:A,B,
while (GetTop(S,p)&&**NULL**) Push(S, p->lchild); Pop(S,p);	S:A,B,D
while (GetTop(S,p)&&p) Push(S, D->lchild);	S:A,B,D,NULL
while (GetTop(S,p)&&p) Push(S, B->lchild);	S:A,B,D
while (GetTop(S,p)&&p) Push(S, A->lchild);	S:A,B
Push(S,A)	S:A,

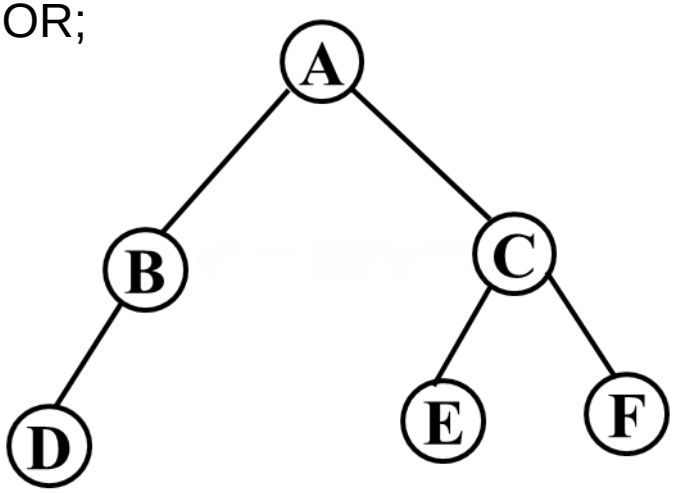
中序遍历右子树。

T为NULL，空树，不操作，应退回到上一层。

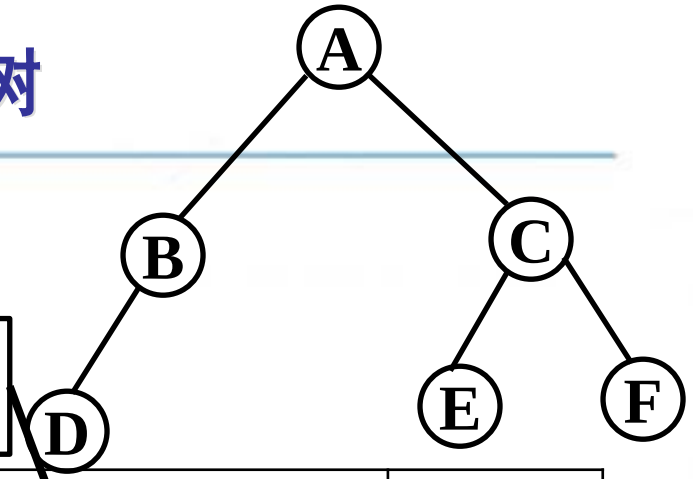
遍历二叉树——中序遍历非递归算法2

- 算法6.3中序遍历二叉树的非递归实现（方法2）

```
Status InOrderTraverse (BiTree T, Status (* Visit)(TElemType e))
{ // 采用二叉链表存储结构，Visit是对数据元素操作的应用函数。
  // 中序遍历二叉树T的非递归算法，对每个数据元素调用函数Visit。
    InitStack (S); p=T;
    while (p || !StackEmpty(S)){
        if (p)  { Push(S,p);  p=p->lchild; }   // 根指针进栈，遍历左子树
        else {                          // 根指针退栈，访问根结点，遍历右子树
            Pop(S,p);   if (!Visit(p->data))  return ERROR;
            p=p->rchild;
        }// else
    }// while
    return OK;
}// InOrderTraverse
```



遍历二叉树



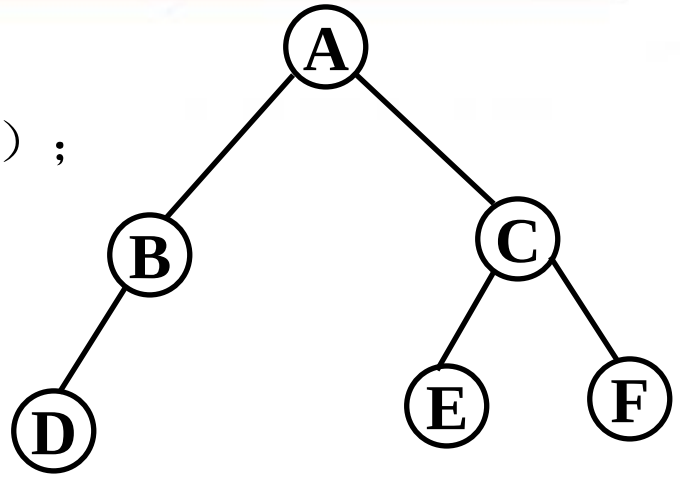
- 例：分析中序遍历图示二叉树非递归过程

左子树访问完毕，应访问当前树根结点（栈顶指针所指结点）。而后访问右子树。

while (p || !StackEmpty(S)){/*p->A,p->C*/ if (p) {/*..*/}**else{Pop(S,p); Visit(p->data);p=p->rchild;}}**	S:
while (p || !StackEmpty(S)){/*p->B,p:NULL*/ if (p) {/*..*/}**else{Pop(S,p); Visit(p->data);p=p->rchild;}}**	S:A
while (p || !StackEmpty(S)){/*p->D,p:NULL*/ if (p) {/*..*/} **else{Pop(S,p); Visit(p->data);p=p->rchild;}}**	S:A,B
while (p || !StackEmpty(S)){/*p->D*/ **if (p) { Push(S,p); p=p->lchild; }**/*..*/}	S:A,B,D
while (p || !StackEmpty(S)){/*p->B*/ **if (p) { Push(S,p); p=p->lchild; }**/*..*/}	S:A,B
while (p || !StackEmpty(S)){/*p->A*/ **if (p) { Push(S,p); p=p->lchild; }** /*..*/}	S:A,

中序遍历左子树。

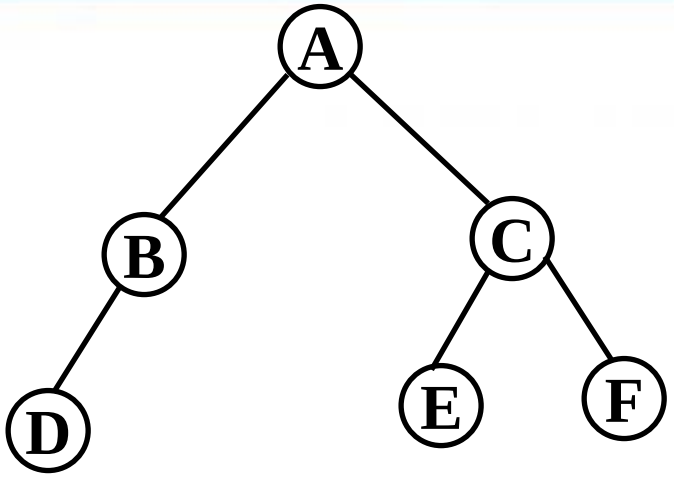
遍历二叉树的应用*



●统计二叉树叶子结点的个数（先序遍历）；

```
void CountLeaf( BiTree T,  int& count)
{
    if(T){
        if( (!T->lchild) && (!T->rchild) )
            count++;
        CountLeaf( T->lchild, count);
        CountLeaf( T->rchild, count);
    }
}
```

遍历二叉树的应用*



- 求二叉树的深度（后序遍历）；

```
int Depth( BiTree T)
{
    if( !T )
        depthval = 0;
    else{
        depthLeft = Depth( T->lchild );
        depthRight = Depth( T->rchild );
        depthval = 1+(depthLeft > depthRight ? depthLeft : depthRight);
    }
    return depthval;
}
```

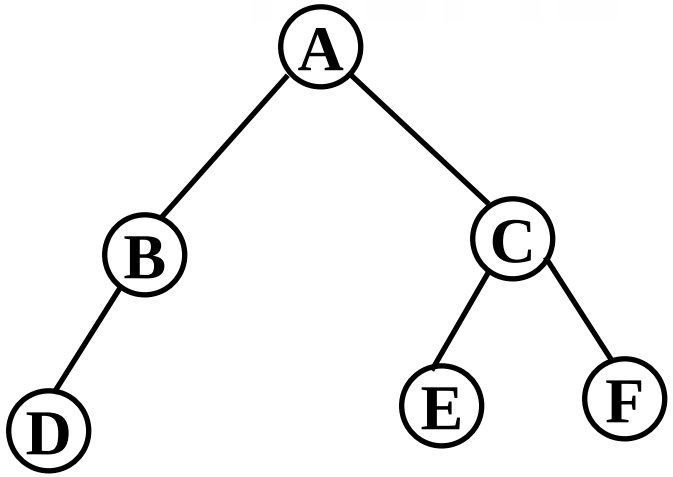
遍历二叉树的应用*

- 复制二叉树（后序遍历）

```
//生成一个二叉树的结点
BiTNode *GetTreeNode(TElemType item, BiTNode *lptr,
                                    BiTNode *rptr)
{
    if( !( T= (BiTNode *) malloc (sizeof(BiTNode *))))  exit(1);
    T->data = item;
    T->lchild = lptr;
    T->rchild = rptr;
    return T;
}
```

遍历二叉树的应用*

- 复制二叉树（后序遍历）



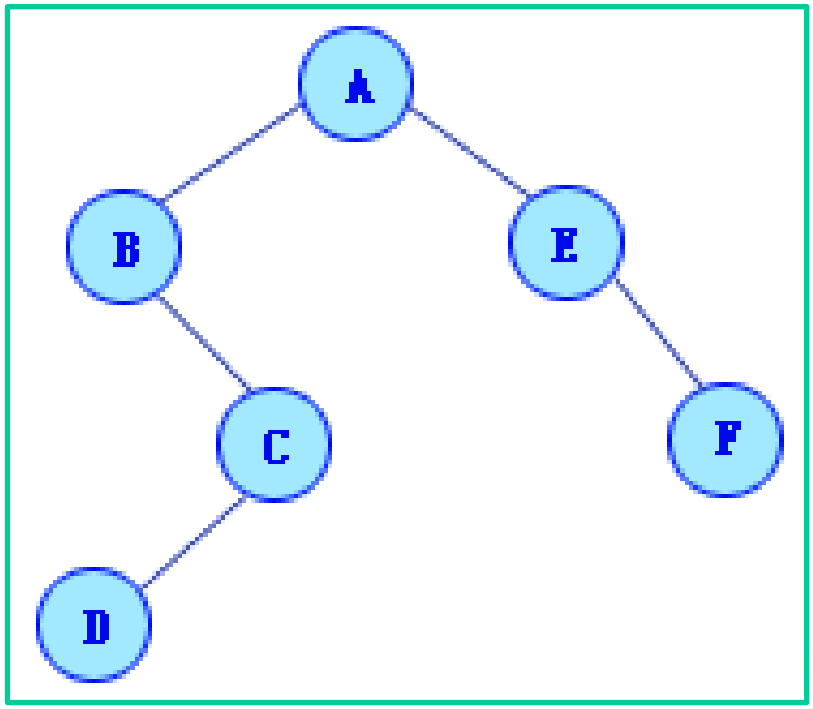

```
BiTNode *CopyTree(BiTNode *T)
{
    if( !T)  return NULL;
    if( T->lchild )
        newlptr = CopyTree(T->lchild);
    else  newlptr = NULL;
    if( T->rchild )
        newrptr = CopyTree(T->rchild);
    else  newrptr = NULL;
    newnode = GetTreeNode(T->data, newlptr, newrptr);
    return newnode;
}
```

遍历二叉树

- 建立二叉树的存储结构—二叉链表
  - 在输入数据正确的前提下，按给定的先序序列建立二叉链表的算法为：
    - 若输入的字符是‘#’，则建立空树；
    - 否则
      - 建立根结点；
      - 递归建立左子树的二叉链表；
      - 递归建立右子树的二叉链表。

遍历二叉树

- 例如，空树以“#”表示，只有一个根结点A的二叉树以“A##”表示，下列二叉树则以“AB#CD###E#F##”表示。

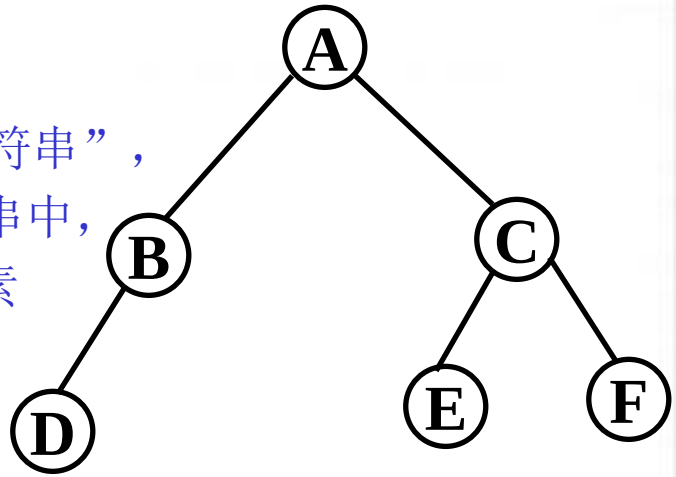


- 以此二叉树为例算法6.4的执行过程如动画所示。演示

遍历二叉树

● 算法6.4

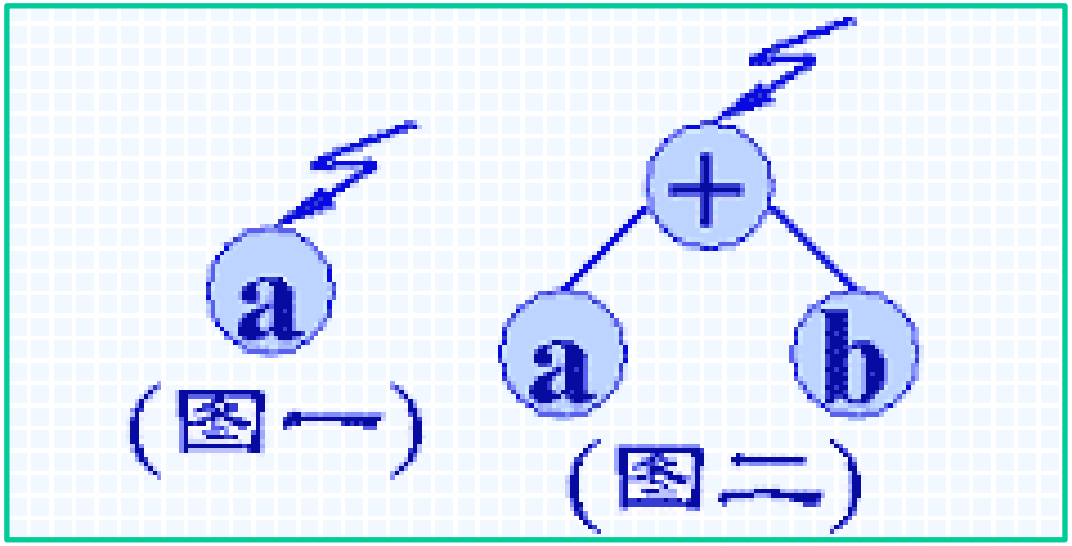
```
Status CreateBiTree(BiTree &T)
{   // 在先序遍历二叉树的过程中输入二叉树的“先序字符串”，
    // 建立根指针为 T的二叉链表存储结构。在先序字符串中，
    // 字符‘#’表示空树，其它字母字符为结点的数据元素
    scanf (&ch) ;
    if (ch==‘#’) T=NULL;                    // 建空树
    else {
        if (!(T = (BiTNode *) malloc(sizeof(BiTNode))))   exit(OVERFLOW);
        T->data = ch;                       // “访问”操作为生成根结点
        CreateBiTree(T->Lchild);            // 递归建(遍历)左子树
        CreateBiTree(T->Rchild);            // 递归建(遍历)右子树
    } // else
} // CreateBiTree
```



例：输入为ABD###CE##F##时，请根据算法建立二叉树，分析执行过程

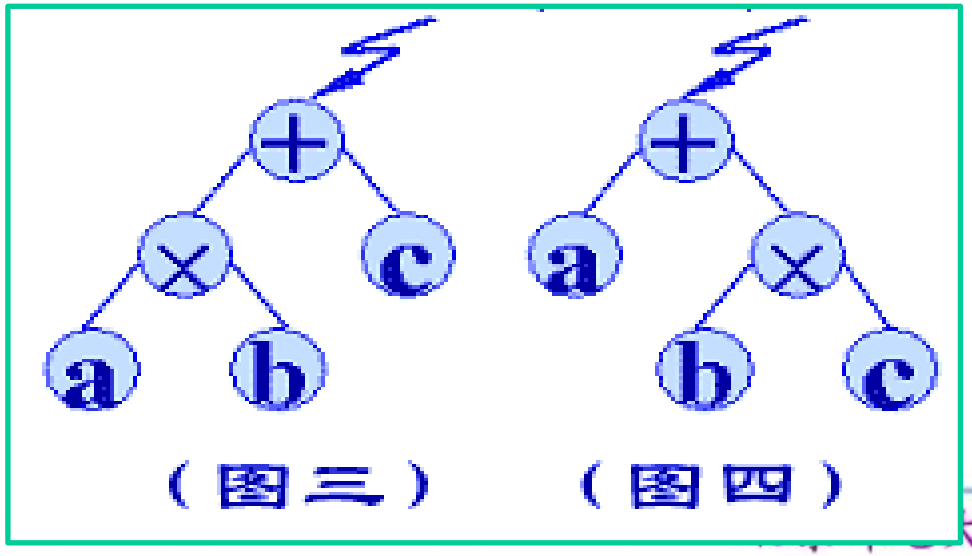
表达式字符串建二叉树*

- 如何根据输入的表达式字符串建二叉树？
  - 从最简表达式起，分析表达式和二叉树的对应关系。
    - 1)只含一个操作数 a 的表达式，对应的二叉树中只含一个叶子结点(图一)。
    - 2)只含一个运算符的表达式 a+b，对应的二叉树的形态是“其左右子树均为叶子结点”(图二)。由于表达式中只有一个运算符，显然它是根，其左右子树分别为它的第一操作数和第二操作数。



表达式字符串建二叉树*

- 3)当表达式中的运算符多于一个时，则存在一个“哪一个是根结点”的问题。
- 例如，表达式 a×b+c 的二叉树如图三所示，表达式 a+b×c 的二叉树如图四所示。它们的差别是显然的。
- 表达式 a×b+c中，因为后面的运算符（+）的优先数“低于”前面的运算符（×）的优先数，则b是在它之前的运算符（×）的第二操作数，a×b 是+的第一操作数。
- 表达式 a+b×c中，由于后面的运算符（×）的优先数"高于"前面的运算符（+）的优先数，则b是在它之后的运算符的第一操作数，b×c 是+的第二操作数。



（图三）　（图四）

大学信息安全中心

表达式字符串建二叉树*

- 由此可见，从原表达式构造二叉树的过程类似于将原表达式转换成后缀式的过程。
- “构建一棵以运算符为根的二叉树”相当于进行运算，因此，对原表达式中出现的每一个运算符是否即刻构建一棵以它为根的子树（即是否即刻进行运算）取决于在它后面出现的运算符。
- 如果它的优先数"高或等于"后面的运算，则它的运算先进行，即和已经建好的左右子树构成一棵以它为根的二叉树，否则就得等待在它的右子树建成（即在它之后出现的所有优先数高于它的"运算"都完成）之后再建以它为根的二叉树。

表达式字符串建二叉树*

- 由此，按原表达式建二叉链表算法的基本思想为：

if (当前识别的字符 ch 是操作数)

{ 建叶子结点; 暂存; **}**

else if (当前识别的字符ch是运算符)

{

和前一个运算符比较优先数;

若当前的优先数"高"，则暂存;

否则以前一运算符为根建立一棵新的子树(最近建立的两棵子树分别为它的左右子树)并暂存;

}

- 显然，在这个算法中需要两个栈，一个是运算符栈，其作用和表达式转换为后缀式的算法中相同，另一个是暂存“已经建好的子树根的指针”栈。

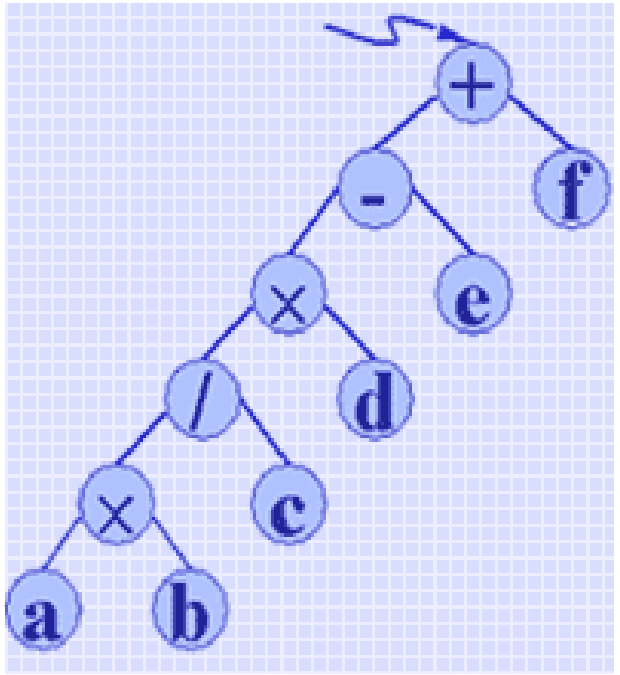
表达式字符串建二叉树*

●例一

原表达式：a × b / c × d - e + f

后 缀 式：a b × c / d × e - f +

➔ 由于‘×’“领先”于‘/’，则‘×’和分别先建好的左右子树(操作数a和b)构成一个子树，又由于‘/’“领先”于在它之后出现的‘×’，则(a×b)就成为‘/’的左子树，而由操作数 c 建成的叶子是它的右子树依次类推，最后建成右图所示二叉树。



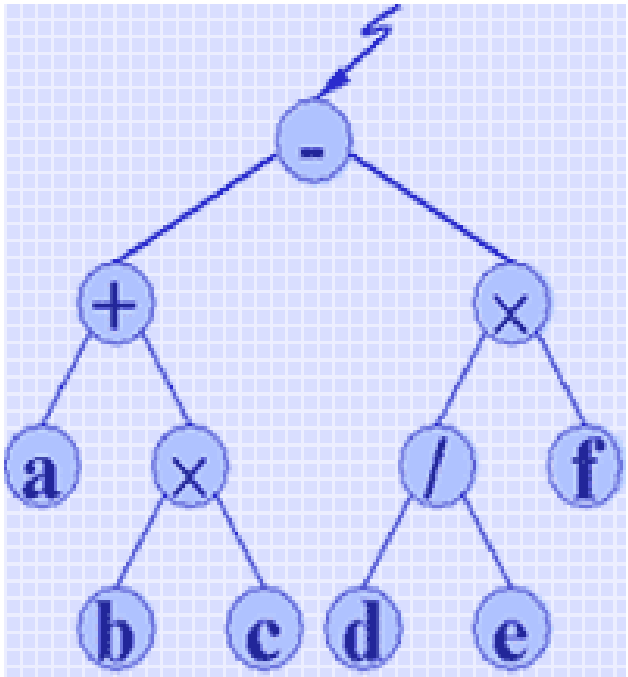
表达式字符串建二叉树*

●例二

原表达式：a + b × c - d / e × f

后 缀 式：a b c × + d e / f × -

➔ 同理，由于‘×’“领先”于‘+’，则‘×’和分别先建好的左右子树(操作数b和c)构成一个子树，又由于‘+’“领先”于在它之后出现的‘-’，则(a×b)就成为‘+’的右子树，而由操作数a建成的叶子是它的左子树，依次类推，最后建成右图所示二叉树。



线索二叉树

●二、线索二叉树

遍历二叉树的结果是求得结点的一个线性序列。

当以二叉链表作为存储结构时，只能找到结点的左右孩子的信息，而不能在结点的任一序列的前驱与后继信息，这种信息只有在遍历的动态过程中才能得到，为了能保存所需的信息，可增加标志域：

lchild	LTag	data	RTag	rchild

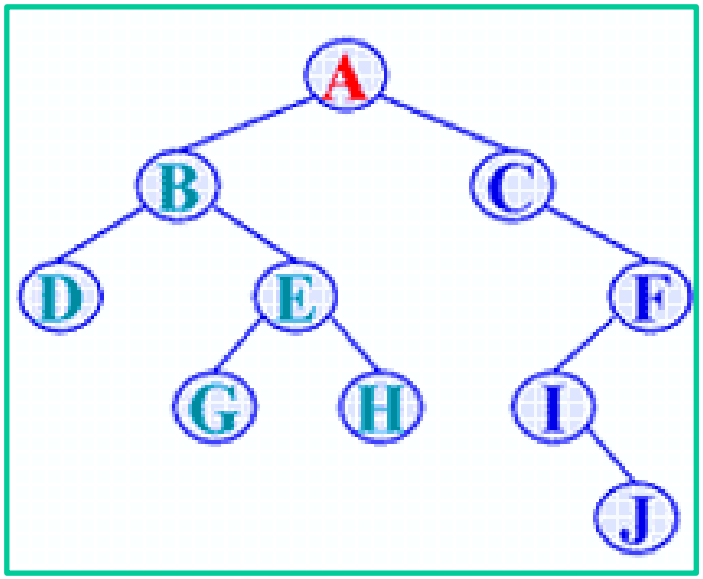
$$LTag=\begin{cases}0 & \text{lchild域指示结点的左孩子}\\ 1 & \text{lchild域指示结点的前驱}\end{cases}$$

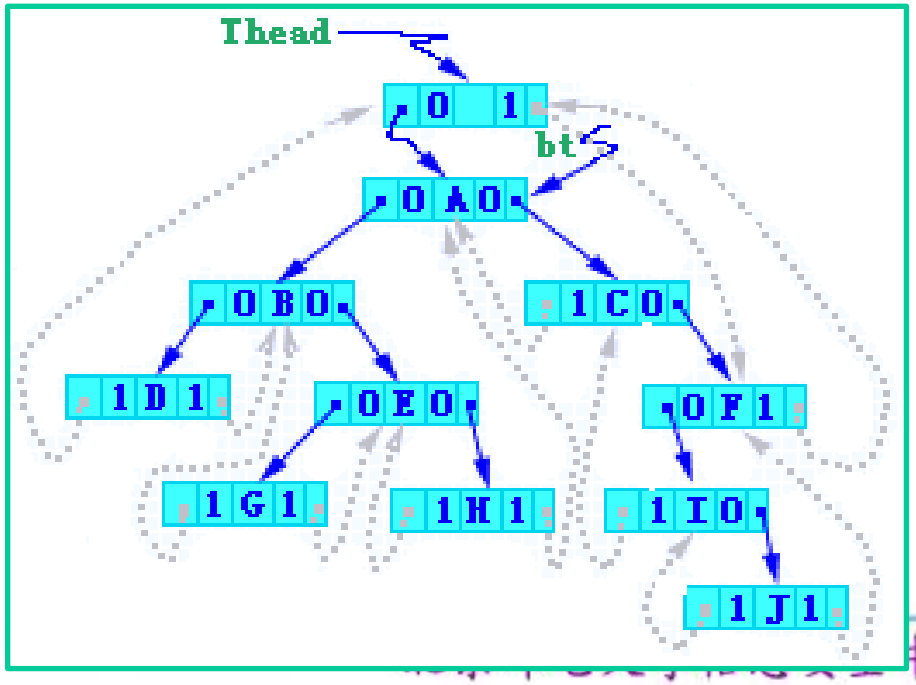
$$RTag=\begin{cases}0 & \text{rchild域指示结点的右孩子}\\ 1 & \text{rchild域指示结点的后继}\end{cases}$$

- 以这种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**，其中指向结点前驱与后继的指针叫做**线索**。加上线索的二叉树称之为**线索二叉树**。

线索二叉树

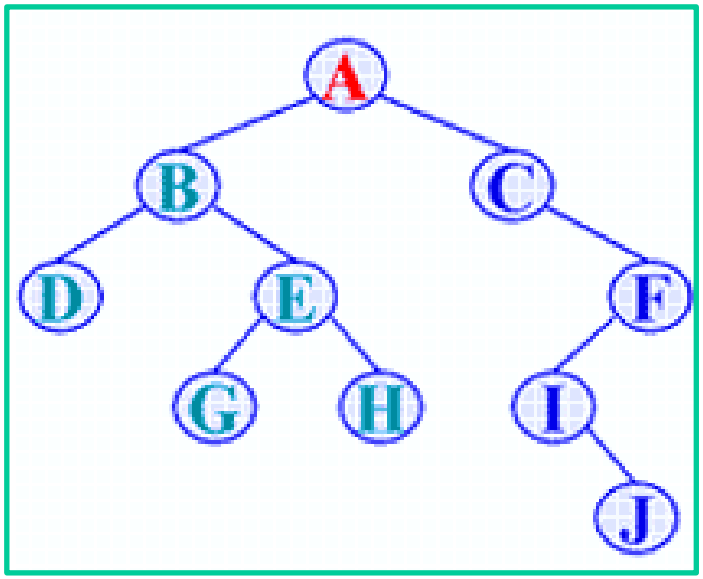
- 若结点的左指针类型标志为“Link”，则 Lchild 指向它的左子树根结点，否则**指向它的“前驱”**；
- 若结点的右指针类型标志为“Link”，则 Rchild指向它的右子树根结点，否则**指向它的“后继”**。
- **例如**，左下图二叉树的**中序线索链表**如右下图所示（图中所有实线表示指针，虚线表示线索）。

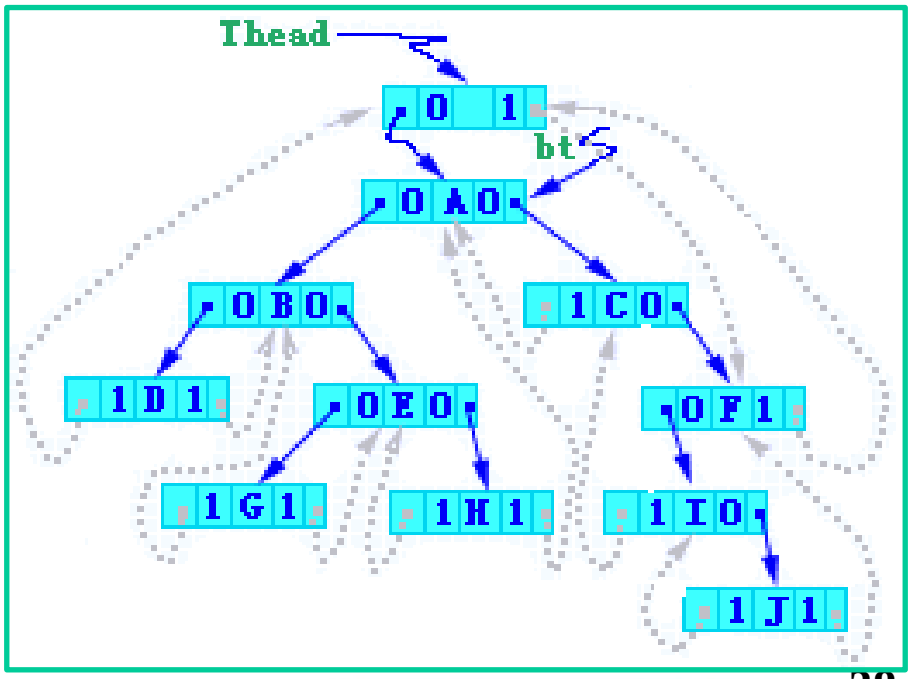




线索二叉树

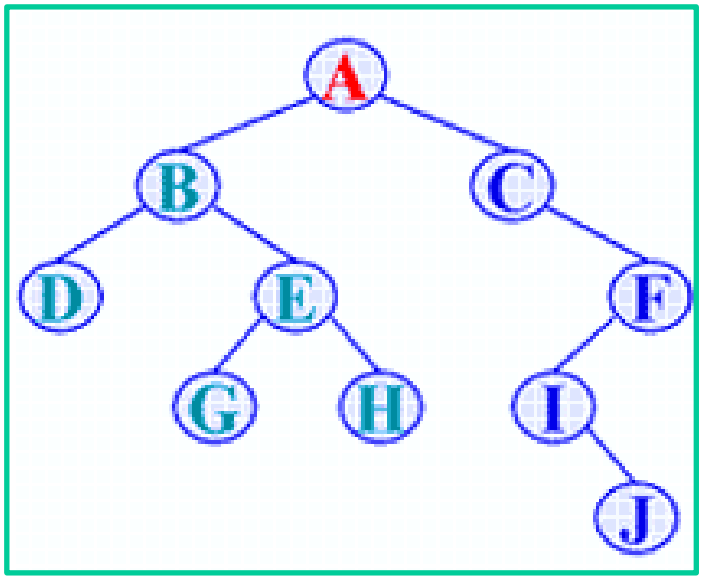
- 从图中可见，在线索链表中添加了一个“头结点”，头结点的左指针指向二叉树的根结点，其右线索指向中序序列中的最后一个结点，中序序列中的第一个结点的左线索和中序序列中的最后一个结点的右线索均指向头结点。空二叉树的线索链表的头结点的左右指针回指到头结点。

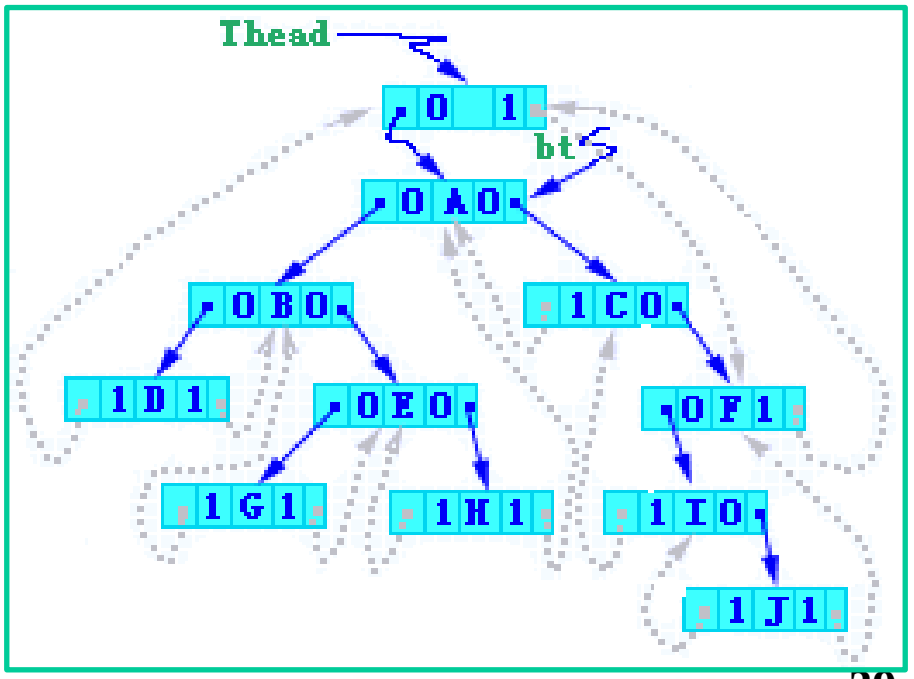




线索二叉树

- 这就好比将二叉树中所有结点置于一个双向循环链表之中，即可以从头结点出发，依照中序遍历的规则对二叉树中的结点依次进行"顺序"（和中序序列相同的次序）访问，或"逆序"（和中序序列相反的次序）访问。





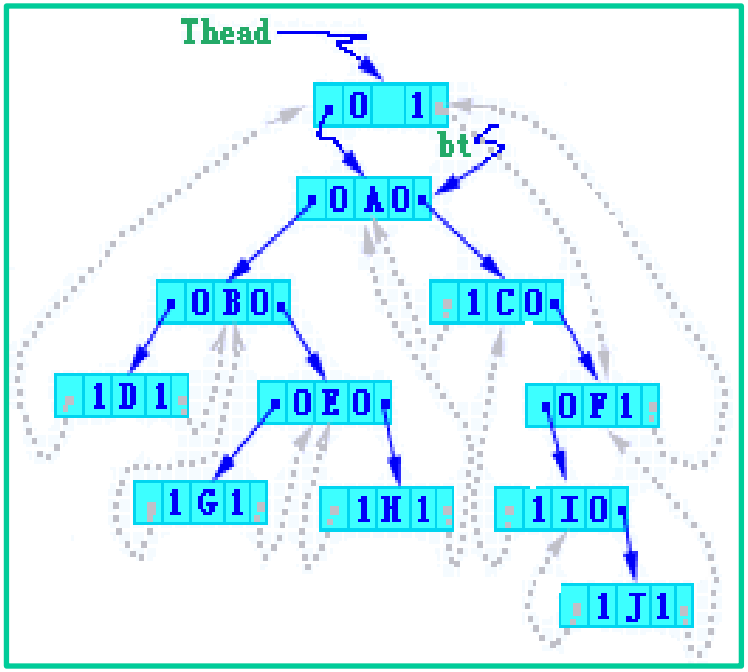
线索二叉树

- 二叉树的二叉线索链表存储表示

```
typedef enum PointerType{ Link=0, Thread=1 };
                    // 定义指针类型，以 Link 表示指针，Thread 表示线索
typedef struct BiThrNode{
    TElemType data;
    struct BiThrNode *Lchild, *Rchild;      // 左右指针
    PointerTag LTag, RTag;                  // 左右指针类型标志
} BiThrNode, *BiThrTree;
```

线索二叉树

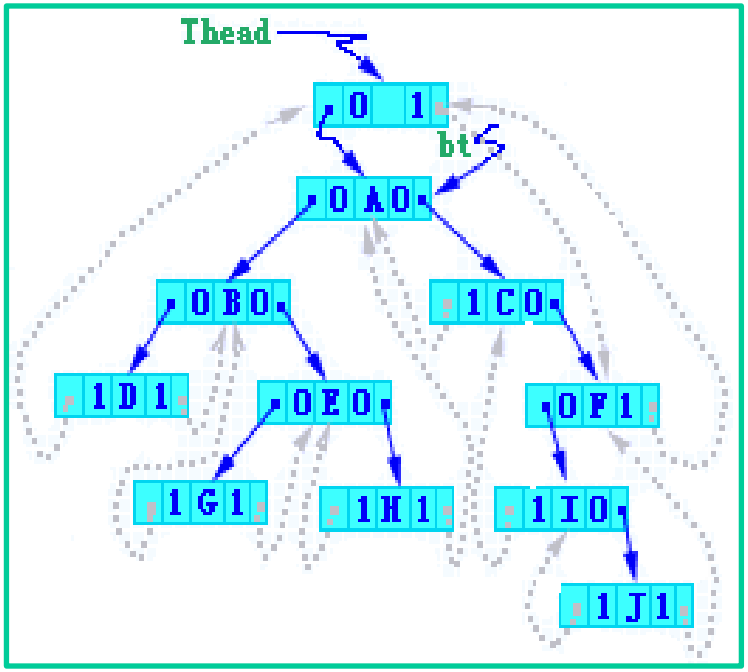
- 如何在中序线索链表上进行遍历，关键问题有二：
  - 一是如何找到访问的第一个结点？
  - 二是如何找到每个结点在中序序列中的后继？
- 首先，在中序线索链表上如何找到中序序列中的第一个结点？
  - 根据对二叉树的中序遍历的定义可知，中序遍历访问的第一个结点必定是“其左子树为空”的结点。
  - 若根结点没有左子树，则根结点即为中序遍历访问的第一个结点；



例：上图线索二叉树中的子树{C,F,I,J}，根结点C没有左子树，则对于子树{C,F,I,J}而言，中序遍历访问的第一个结点是根结点C。

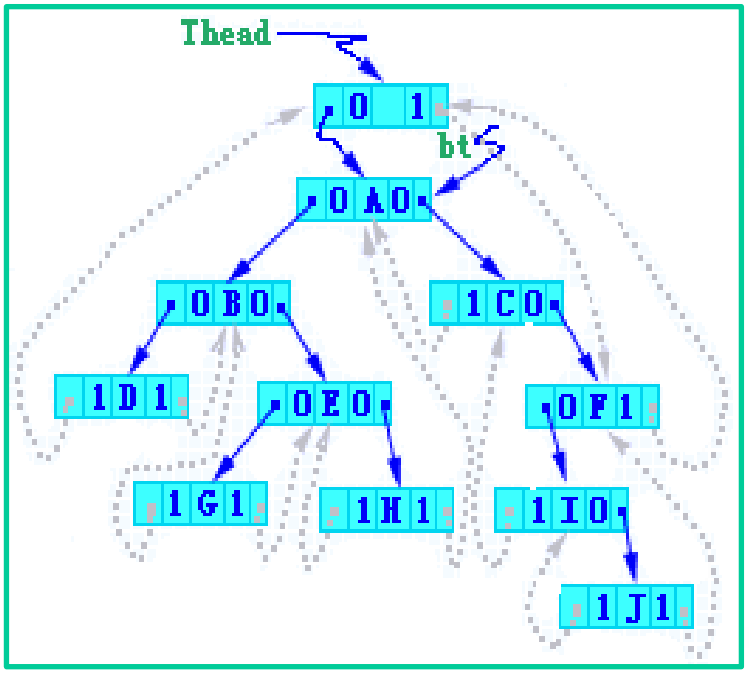
线索二叉树

- 首先，在中序线索链表上如何找到中序序列中的第一个结点？
  - 若根结点的左子树不空，则访问的第一个结点应该是其左子树中“最左下的结点”。
  - 即从根结点出发，顺指针 Lchild 找到其左子树直至某个结点的指针 Lchild 为“线索”止。
  - 该结点必为中序序列中的第一个结点。右图所示例子中的“结点D”。



线索二叉树

- 如何在中序线索链表中找结点的后继？
  - 若结点**没有右子树**，即结点的右指针类型标志 Rtag 为“Thread”，则指针 Rchild 所指即为它的后继，如右图所示例子中的“结点D”、“结点G”等。
  - 若结点**有右子树**，则它的后继应该是其右子树中访问的第一个结点。
  - 和前面所述找二叉树的第一个结点一样，就应该从它的右子树根出发，顺指针 Lchild 直至没有左子树的结点为止，该结点即为它的后继。例如右图中结点B的后继为结点G。



对上图中序线索链表进行遍历的过程演示

线索二叉树

● 算法6.5 基于双向线索链表存储结构的二叉树中序遍历算法

```
Status InOrderTraverse_Thr(BiThrTree T, Status (*Visit)(TElemType e))
{// T 指向中序线索链表中的头结点，头结点的左指针 Lchild 指向二叉树的根结点，
   头结点的右线索 Rchild 指向中序遍历访问的最后一个结点。本算法对此二叉树
   进行中序遍历，对树中每个数据元素调用函数 Visit 进行访问操作
    p = T->Lchild;                              // p 指向二叉树的根结点
    while (p!= T) {                             // 空树或遍历结束时，p==T
      while (p->LTag==Link) p = p->Lchild;
      if(! Visit(p->data))    return ERROR;     // 访问其左子树为空的结点
      while (p->RTag==Thread && p->Rchild!=T) {
        p = p->Rchild; Visit(p->data); // 访问“右线索”所指后继结点
      } // while
      p = p->Rchild;                            // p 进至其右子树根
    } // while
   return OK;
} // InOrderTraverse_Thr
```

1.找到并访问子树第一个结点

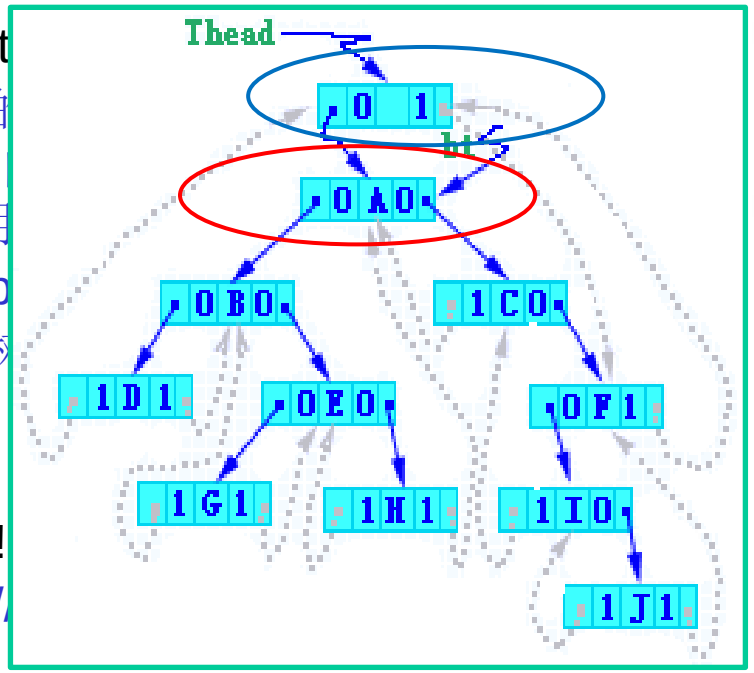
2.右子树为空时，右指针指向后继，直接访问

3.右子树非空，则以右子树为当前子树，迭代上述两步操作。

线索二叉树

- 算法6.5

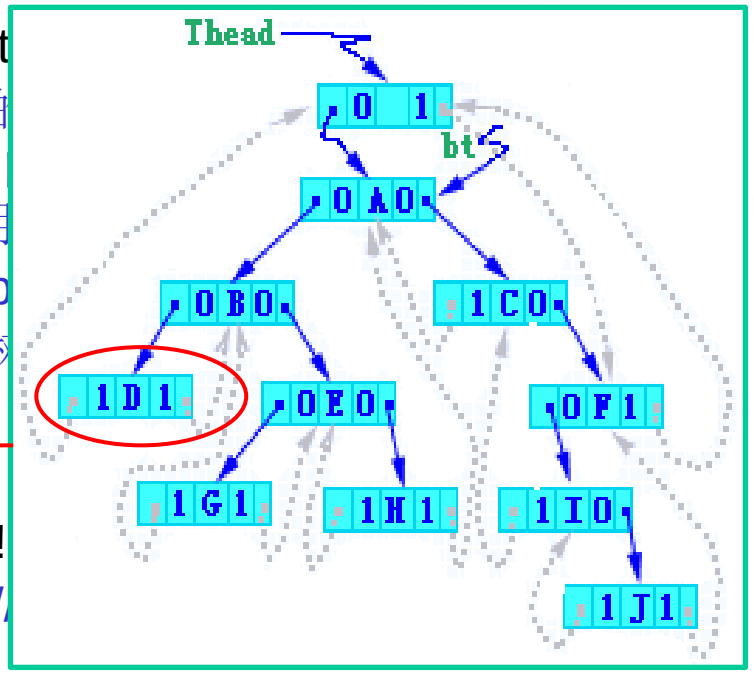
```
Status InOrderTraverse_Thr(BiThrTree T, Stat
{  // T 指向中序线索链表中的头结点，头结点的
   点，头结点的右线索 Rchild 指向中序遍历访
   叉树进行中序遍历，对树中每个数据元素调用
     p = T->Lchild;                          // p
1.从根结点开始操作  while (p!= T) {                        // 空
       while (p->LTag==Link) p = p->Lchild;
       if(! Visit(p->data))    return ERROR;
       while (p->RTag==Thread && p->Rchild!
2.当前结点为T时，代表所有结点都被访问，遍历结束。    p = p->Rchild; Visit(p->data);          //
     } // while
       p = p->Rchild;                          // p 进至其右子树根
     } // while
     return OK;
} // InOrderTraverse_Thr
```



线索二叉树

● 算法6.5

```
Status InOrderTraverse_Thr(BiThrTree T, Stat
{  // T 指向中序线索链表中的头结点，头结点的
   点，头结点的右线索 Rchild 指向中序遍历访
   叉树进行中序遍历，对树中每个数据元素调用
    p = T->Lchild;                                  // p
    while (p!= T) {                                 // 空
      while (p->LTag==Link) p = p->Lchild;
      if(! Visit(p->data))    return ERROR;
      while (p->RTag==Thread && p->Rchild!
        p = p->Rchild; Visit(p->data);              //
      } // while
      p = p->Rchild;                                // p 进至其右子树根
    } // while
   return OK;
} // InOrderTraverse_Thr
```

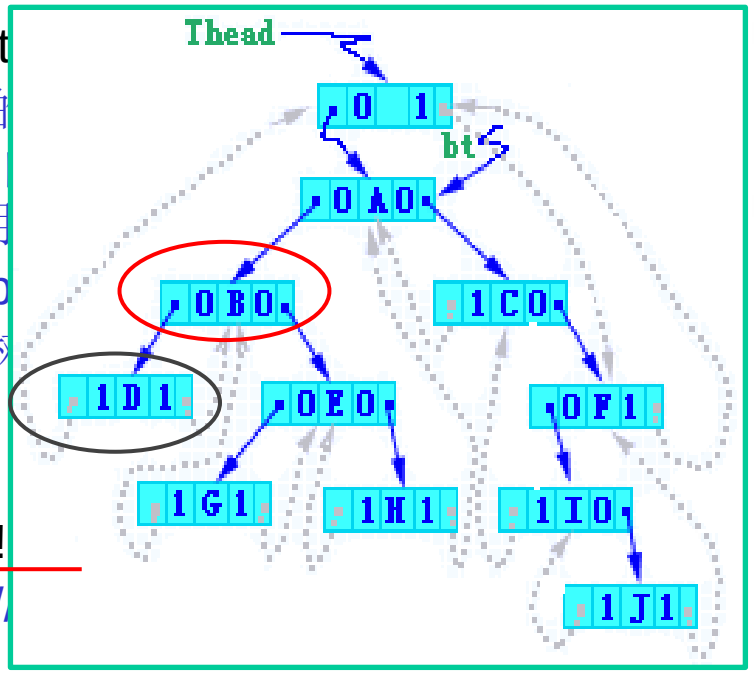


1.首个访问的结点为：当前结点最左侧结点，即左侧首个左标识为线索的结点。

线索二叉树

- 算法6.5

```
Status InOrderTraverse_Thr(BiThrTree T, Stat
{  // T 指向中序线索链表中的头结点，头结点的
   点，头结点的右线索 Rchild 指向中序遍历访
   叉树进行中序遍历，对树中每个数据元素调用
     p = T->Lchild;                                  // p
     while (p!= T) {                                 // 空
       while (p->LTag==Link) p = p->Lchild;
       if(! Visit(p->data))    return ERROR;
       while (p->RTag==Thread && p->Rchild!
         p = p->Rchild; Visit(p->data);              //
       } // while
       p = p->Rchild;                                // p 进至其右子树根
     } // while
    return OK;
} // InOrderTraverse_Thr
```



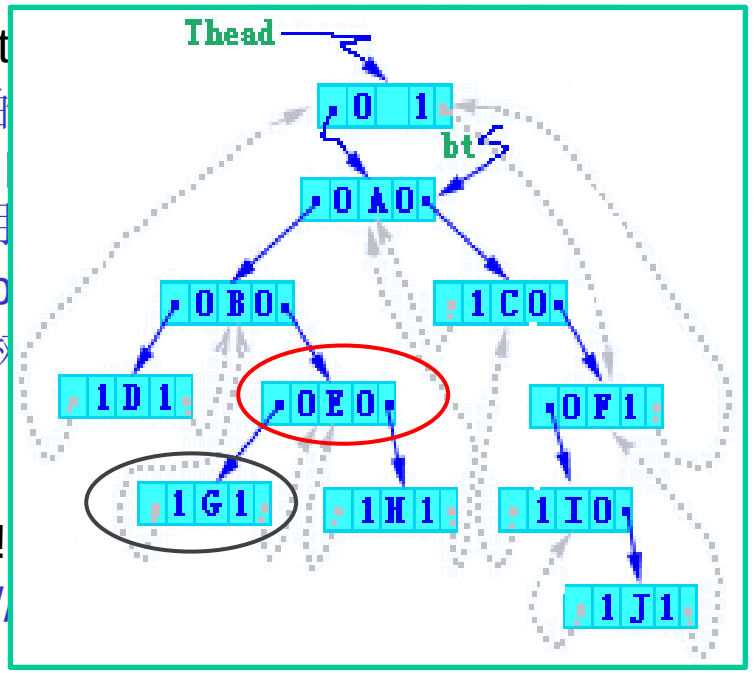
2.1后续访问的结点为：若当前结点右子树为空，即右标识为线索，那么右指针指向后继，可直接访问。

2.2线索二叉链表为循环链表。当前循环终止条件之一为p->Rchild!=T。否则，以上图为例，若循环至结点F还不停止，那么在循环体内，当前指针被赋值指向头结点T，则遍历不会终结。

线索二叉树

● 算法6.5

```
Status InOrderTraverse_Thr(BiThrTree T, Stat
{  // T 指向中序线索链表中的头结点，头结点的
   点，头结点的右线索 Rchild 指向中序遍历访
   叉树进行中序遍历，对树中每个数据元素调用
     p = T->Lchild;                                  // p
     while (p!= T) {                                 // 空
       while (p->LTag==Link) p = p->Lchild;
       if(! Visit(p->data))    return ERROR;
       while (p->RTag==Thread && p->Rchild!
         p = p->Rchild; Visit(p->data);              //
       } // while
       p = p->Rchild;                                // p 进至其右子树根
     } // while
    return OK;
} // InOrderTraverse_Thr
```



3右子树非空时，以右子树为当前结点，重复进行“找子树首个结点”，“找一个结点”操作。

例如：结点G的右孩子为线索，其后续结点为E；结点E的右子树非空，其后继结点应为：右子树的首个结点H。

线索二叉树

- 如何建立二叉树的线索链表？
  - 线索链表上保存的是“遍历”过程中得到的前驱和后继的信息，因此应该在**遍历过程中建立**，即在遍历过程中改变二叉链表中结点的“空指针”以及相应的“指针类型标志”。
    - 若结点没有左子树，则令其左指针指向它的“前驱”,左指针类型标志 “Thread”;
    - 若结点没有右子树，则令它的右指针指向它的“后继”,右指针类型标志 “Thread”。
    - 为了获取“前驱”的信息，需要在遍历过程中添加一个指向其前驱的指针 pre。

线索二叉树

- 如何建立二叉树的线索链表？
  - 建立线索链表的过程即为将遍历过程中对结点进行下列“访问”操作（指针 p 指向当前访问的结点，pre 指向在它之前“刚刚”访问过的结点）：

```
if (!pre->Rchild) {
    pre->RTag = Thread;
    pre->Rchild = p;//上一个访问结点的右线索：pre后继为p
}
if (!p->Lchild) {
    p->LTag = Thread;
    p->Lchild = pre; //当前结点的左线索：p前驱为pre
}
pre = p;
```

线索二叉树

- 算法6.6

```
Status InOrderThreading(BiThrTree &Thrt, BiThrTree T)
{   // T为指向二叉树根结点的指针，由此二叉链表建立二叉树
      // 的中序线索链表，Thrt 指向线索链表中的头结点。
     if (!(Thrt = (BiThrTree)malloc(sizeof(BiThrNode))))  exit (OVERFLOW);
      Thrt->LTag = Link; Thrt->RTag =Thread;   // 建头结点
      Thrt->Rchild = Thrt;                          // 右指针回指
      if (! T) Thrt->Lchild = Thrt;              // 若二叉树空，则左指针回指
      else {
         Thrt->Lchild =T; pre = Thrt;
         InThreading(T);                          // 中序遍历进行中序线索化
         pre->Rchild = Thrt; pre->RTag = Thead;
                                                  // 对中序序列中最后一个结点进行线索化
         Thrt->Rchild = pre;                          // 建非空树的头结点的"右线索“
      } // else
      return OK;
} // InOrderThreading
```

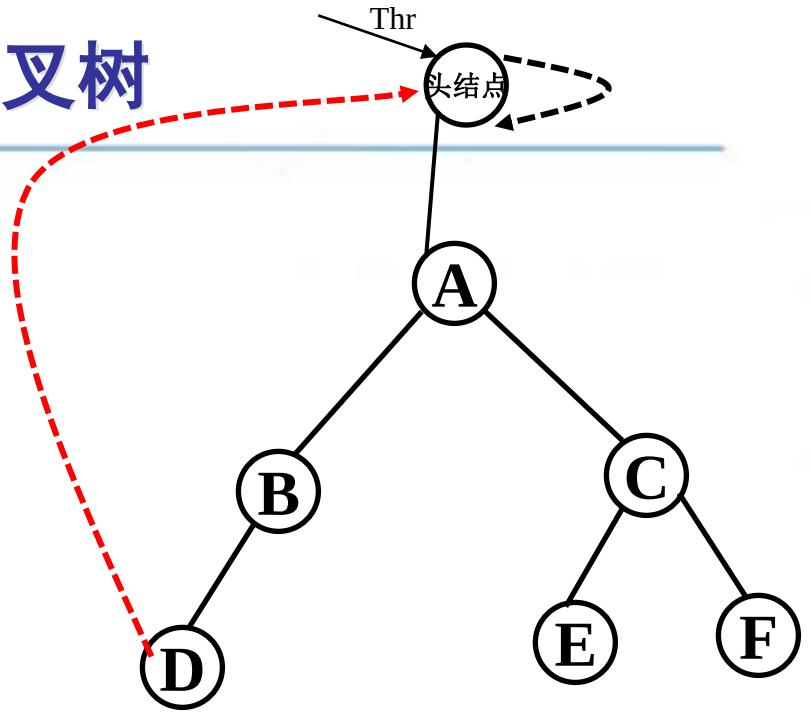
线索二叉树

- 算法6.7

```
void InThreading(BiThrTree p)
{// 对 p 指向根结点的二叉树进行中序遍历，遍历过程中进行“中序线索化”。若p所指结点的左指针为空，则将它改为“左线索”，若 pre 所指结点的右指针为空，则将它改为“右线索”。指针 pre 在遍历过程中紧随其后，即始终指向 p 所指结点在中序序列中的前驱。
    if (p) {
        InThreading(p->Lchild);                    // 对左子树进行线索化
        if (!p->Lchild)     { p->LTag = Thread; p->Lchild = pre; }   // 建前驱线索
        if (!pre->Rchild)   { pre->RTag = Thread; pre->Rchild = p; } // 建后继线索
        pre = p;                                   // 保持 pre 指向 p 的前驱
        InThreading(p->Rchild);                    // 对右子树进行线索化
    } // if
} // InThreading
```

- 结合动画演示6-5-2.swf，理解算法6.6和算法6.7的执行过程。

线索二叉树



InThreading(D->Lchild);
if (!p->Lchild) {
p->LTag = Thread;
p->Lchild = pre; }
if (!pre->Rchild) { pre->RTag = Thread; pre->Rchild = p; }
pre = p;
	pre=Thr,p=D

InThreading(B->Lchild);	pre=Thr,p=B
InThreading(A->Lchild);	pre=Thr,p=A

1.上层函数初始化头结点Thr后，以树根结点A为参数调用递归函数InThreading(A)。pre指向头结点Thr。

2.递归函数InThreading(A)中，前驱pre指向头结点Thr，当前结点p指向A。执行分为三个部份：线索化左子树，线索化当前结点，和线索化右子树。首先执行线索化左子树，通过对函数自身递归调用实现。函数调用栈产生记录：InThreading(A)，当前参数记录为：pre=Thr，p=A。

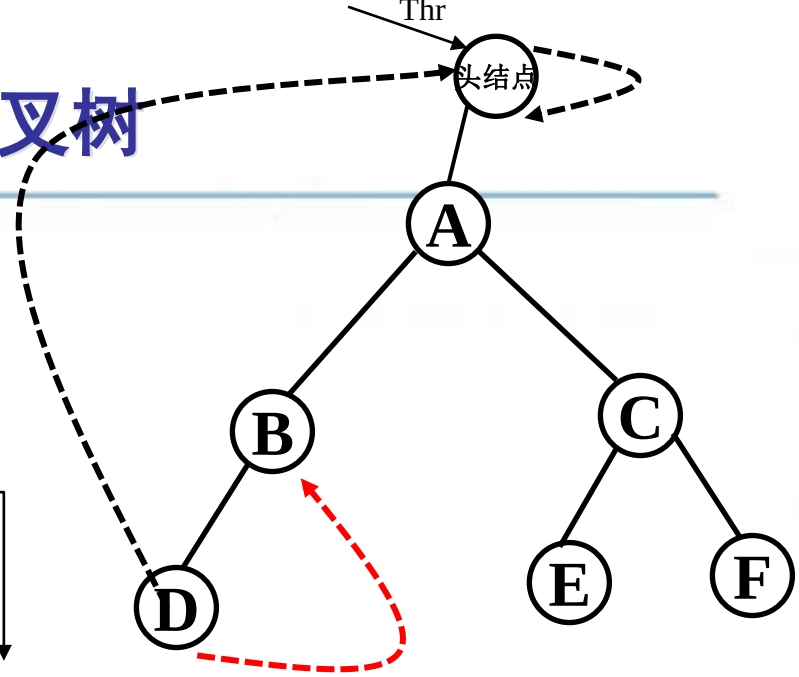
3.对结点B，D的执行过程类似。以D的左子树为参数递归调用时，因左子树为空，不做操作。返回上一次级调用。此时，pre=Thr,p=D，而线索化左子树操作完成，应该线索化当前结点。

线索二叉树

InThreading(D->Lchild);
if (!p->Lchild) {
p->LTag = Thread;
p->Lchild = pre; }
if (!pre->Rchild) { pre->RTag = Thread; pre->Rchild = p; }
pre = p;
InThreading(D->Rchild);
	pre=D,p=D

InThreading(B->Lchild);	pre=Thr,p=B
InThreading(A->Lchild);	pre=Thr,p=A

InThreading(B->Lchild); **if (!p->Lchild) {** **p->LTag = Thread;** **p->Lchild = pre; }** **if (!pre->Rchild) { pre->RTag = Thread; pre->Rchild = p; }**	pre=D,p=B
InThreading(A->Lchild);	pre=B,p=A



4.完成当前结点的线索化后，更新前驱指针pre为指向D，而后线索化D的右子树。

5. D的右子树为空，不做操作。返回上一次级调用。至此，对结点D的递归操作全部完成，返回上一级调用。

6.从结点D返回意味着对结点B的左子树的线索化操作已经完成，应该进行结点B自身的线索化操作，此时，pre=D,p=B。

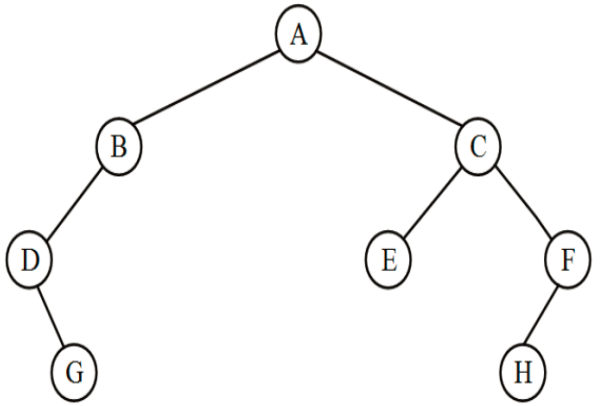
7.其他结点的操作过程类似。

已知二叉树的顺序存储如下，请将该树后序线索化

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	A	B	C	D		E	F		G					H	

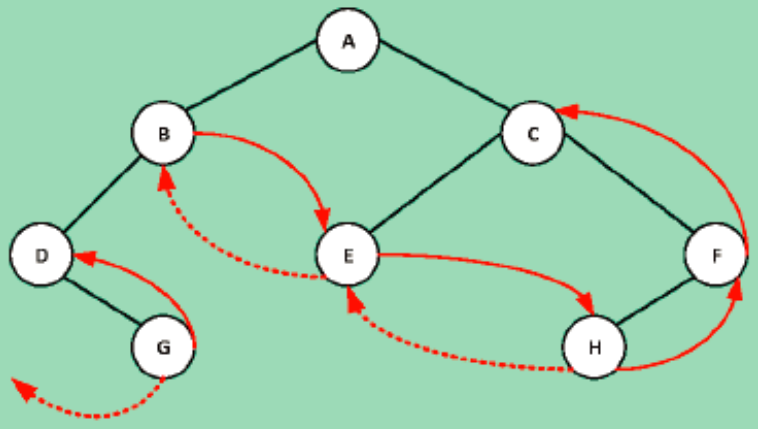
已知二叉树的顺序存储如下，请将该树后序

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	A	B	C	D		E	F		G					H	

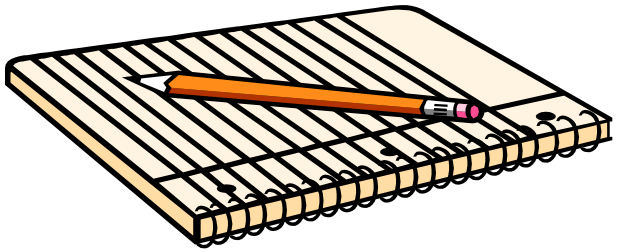


此树的后序遍历序列为：G D B E H F C A

后序线索化之后的树为：



树和森林



- 6.1 树的定义和基本术语
- 6.2 二叉树
- 6.3 遍历二叉树和线索二叉树
- 6.4 树和森林
- 6.5 赫夫曼树及其应用

树和森林—双亲表示法

- 在大量的应用中，人们曾使用多种形式的存储结构来表示树。这里，我们介绍3种常用的链表结构。

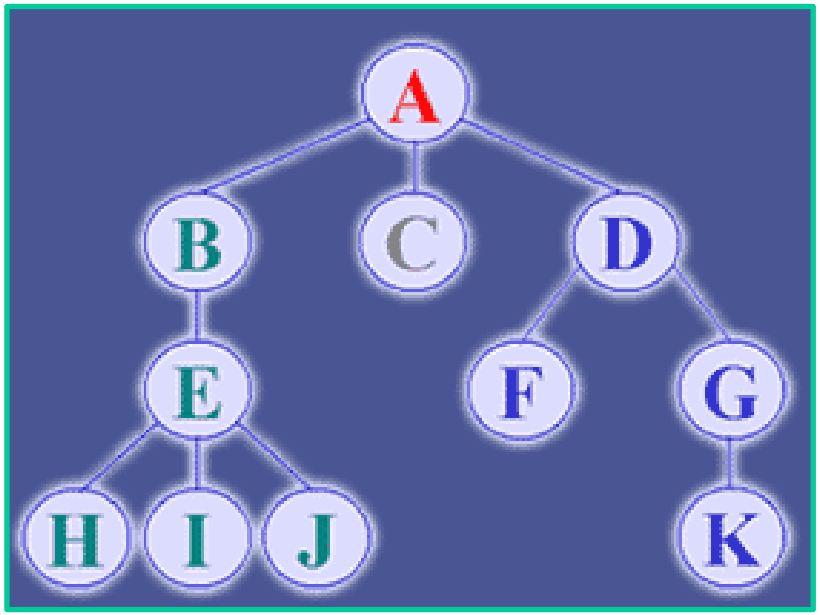
- **1、双亲表示法**

假设以一组连续空间存储树的结点，同时在每个结点中附设一个指示器指示其双亲结点在链表中的位置，其形式说明如下：

```
#define MAX_TREE_SIZE  100
typedef struct  PTNode {                    // 结点结构
        TElemType data;
        int parent;                         // 双亲位置域
} PTNode;
typedef struct {                    // 树结构
        PTNode  nodes[MAX_TREE_SIZE];
        int     r, n;                       // 根的位置和结点数
} PTree;
```

树和森林—双亲表示法

- 例如，下图所示树的双亲链表如右所示。



	data	parent
0	A	-1
1	B	0
2	C	0
3	D	0
4	E	1
5	F	3
6	G	3
7	H	4
8	I	4
9	J	4
10	K	6

r=0

n=11

```
typedef struct{
  TElemType  data;
  int parent;
}PTNode;
```

```
typedef struct{
  PTNode
node[MAX_TREE_SIZE];
  int r,n;
}PTree;
```

树和森林—孩子表示法

- **2、孩子表示法—多重链表**

  由于树中每个结点可能有多棵子树，可用多重链表来表示，即每个结点有多个指针域，其中每个指针指向一个子树的根结点。

- 下面两种结点格式

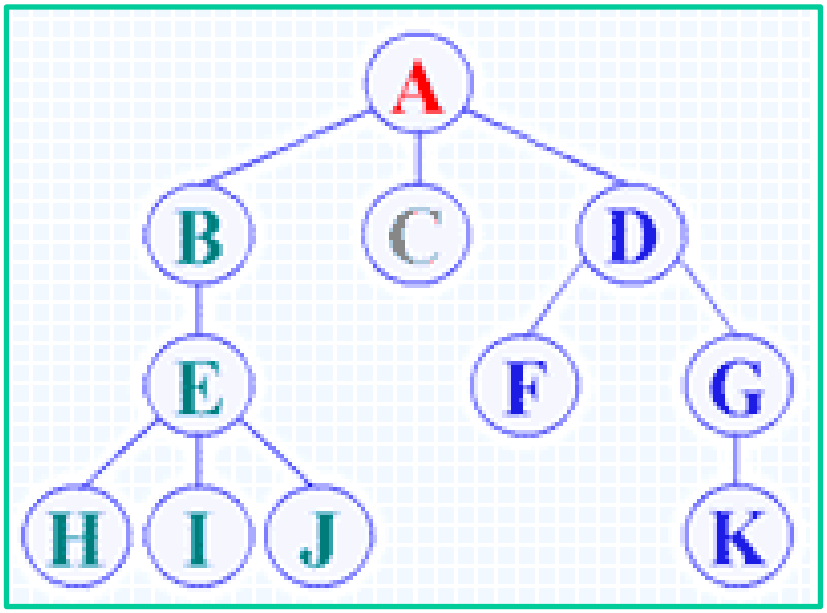
data	child1	child2	。。。	childd

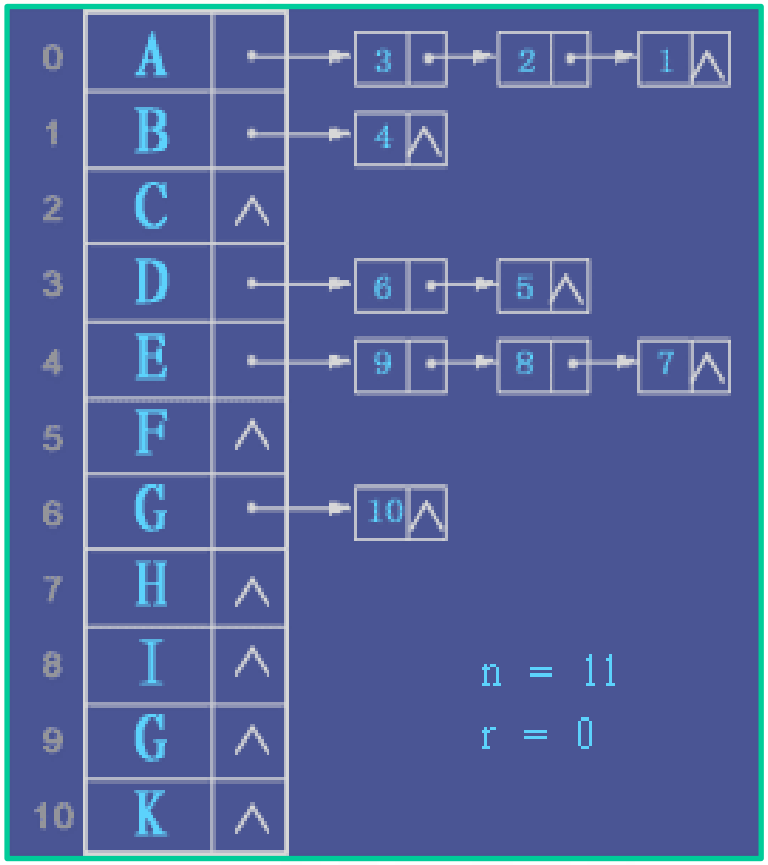
data	degree	child1	child2	。。。	childD

- 第一种：**结点同构**，d为树的度，由于树中很多结点的度小于d，所以造成很多空链域；
- 第二种：**结点异构**，D为结点的度，degree中存放D的值，可以节省空间，但是操作不方便；
- 第三种：见下页

树和森林—孩子表示法

●例如，下列图示树的孩子链表如右图所示。





1.把每个结点的孩子视为一个线性表。结点的度不同，因此采用链式存储结构。
2.N个结点则有N个链表，它们的头指针又组成一个线性表。为方便访问，采用顺序存储结构。

树和森林—孩子表示法

- 让**每个结点的“子树根”构成一个线性表**，以链表作它的存储结构，令该结点的子树根链表的**头指针和结点的数据元素**构成一个结点存储结构，并将所有这样的结点存放在一个地址连续的存储空间里，所构成的树的存储结构称为树的孩子链表。

❊**注意**：“孩子结点”中的“子树根”只存放该结点在一维数组中的下标。由于整个结构无法用一个“指针”表示，所以对整个树结构当然还需要给出结点数目和根的位置。

- 可以将双亲表示法和孩子表示法结合起来，即在结点结构中加上双亲的“地址号”，将双亲表示和孩子链表结合在一起 。

树和森林—孩子表示法

- 树的孩子链表存储表示

孩子结点结构：

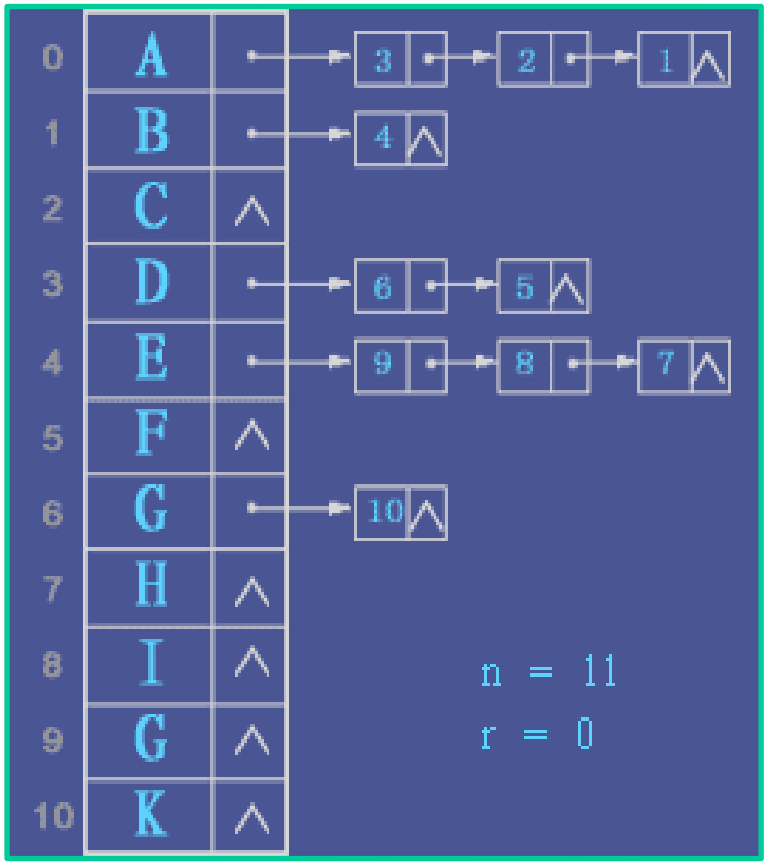
```
typedef struct CTNode {          // 孩子结点
        int child;
        struct CTNode *next;
} *ChildPtr;
```

双亲结点结构：

```
typedef struct {
        ElemType data;           // 结点的数据元素
        ChildPtr firstchild;     // 孩子链表头指针
} CTBox;
```

树结构：

```
typedef struct {
        CTBox  nodes[MAX_TREE_SIZE];
        int n, r;                // 结点数和根结点的位置
} CTree;
```



树和森林—二叉链表（孩子兄弟）表示法

●3、二叉链表表示法（或二叉树表示法，或孩子兄弟表示法）

```
typedef struct CSNode{
        ElemType data;
        struct CSNode *firstchild, *nextsibling;
} CSNode, *CSTree;
```

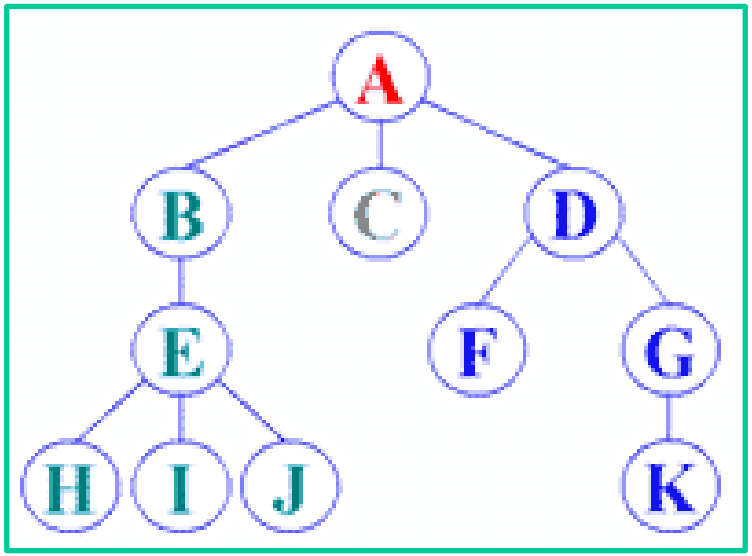
树中每个结点都设有两个指针：

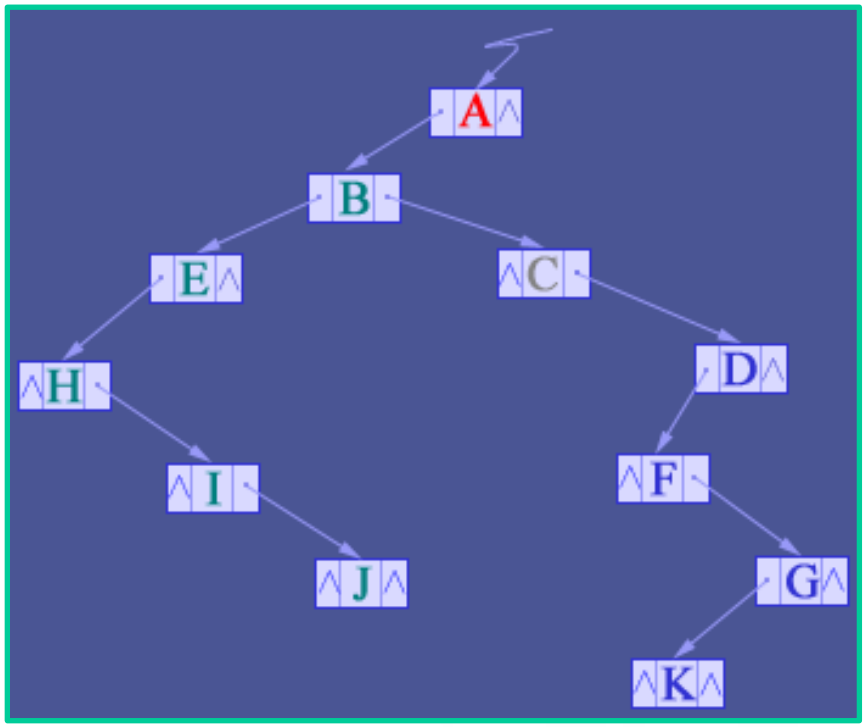
- 其一，firstchild 指向该结点的“第一个”子树根结点；
- 其二，nextsibling 指向它的“下一个”兄弟结点。

这里的“第一个”和“下一个”都没有逻辑上的含义，只是在构造存储结构时自然形成的次序。

树和森林—孩子兄弟表示法

●例如，左下图所示树的孩子-兄弟链表如右下图所示。

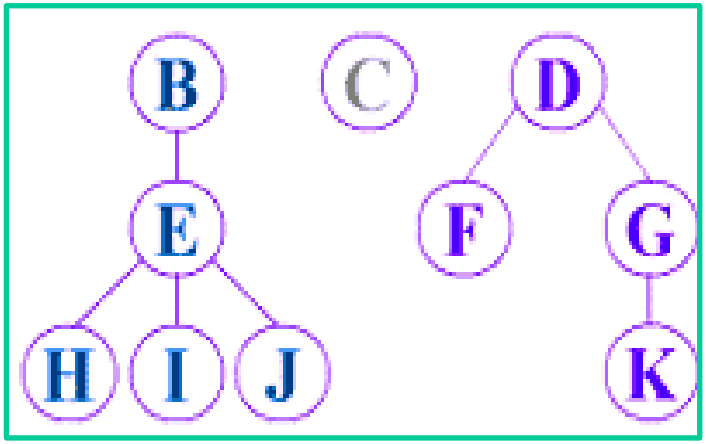


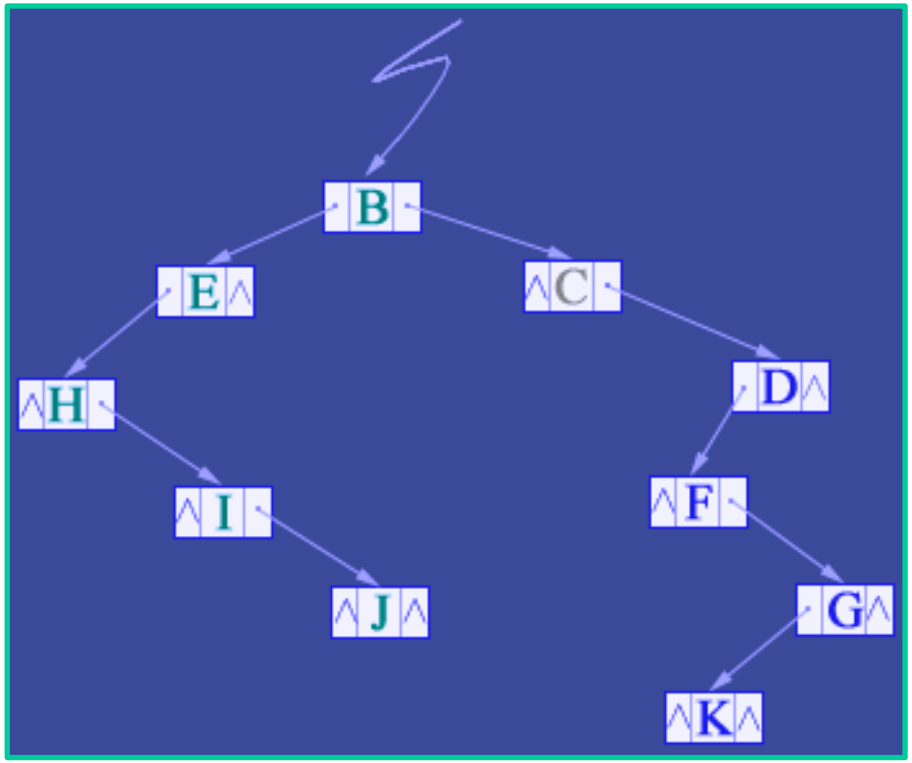



```
typedef struct CSNode{
        ElemType data;
        struct CSNode *firstchild,
*nextsibling;
} CSNode, *CSTree;
```

树和森林—孩子兄弟表示法

- 删除树中的根结点之后，可以得到左下图示的森林，它的孩子-兄弟链表则由指向结点B的指针表示， 见右下图。





- 对森林来说，可认为各棵树的根结点之间是一个“兄弟”关系，因此，无论树和森林都可以用这样的"二叉链表"表示。由于结点中的两个指针指示的分别为 "孩子"和"兄弟"的关系，故称为"孩子-兄弟链表" 。

森林和二叉树的转换

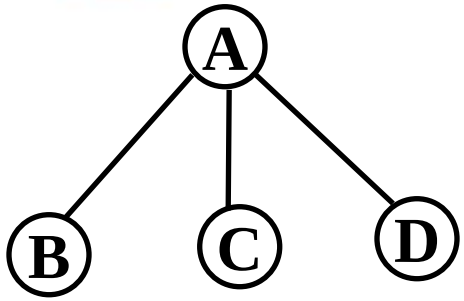
●森林和二叉树的转换

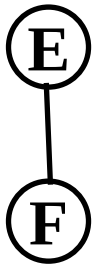
➔ 1、森林转换成二叉树

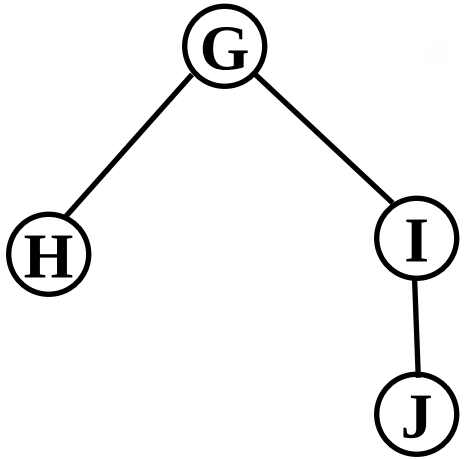
如果F = { T_1 , T_2, …, T_m }是森林，则可按如下规则转换成一棵二叉树B =(root, LB, RB)：

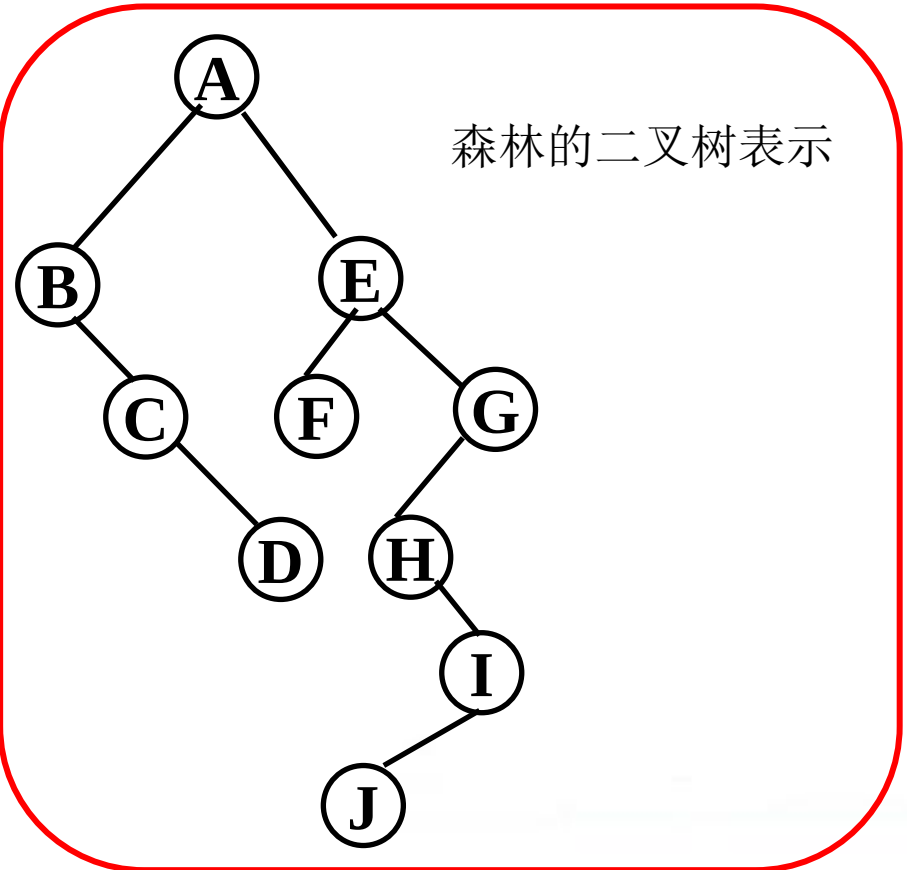
- (1)若森林 F 为空集，即m=0，则二叉树 B 为空树；
- (2)若森林 F 非空，即m≠0，则B的根root即为森林中第一棵树的根结点 ROOT(T_1) ；B的左子树LB是从T_1中根结点的子树森林 {T_{11} , T_{12} , …, T_{1m_1} } 转换而成的二叉树；其右子树RB是从森林中删去第一棵树之后由其余树构成的森林F‘＝{T_2, T_3,…, T_m } 转换而成的二叉树。
- 演示

森林和二叉树的转换



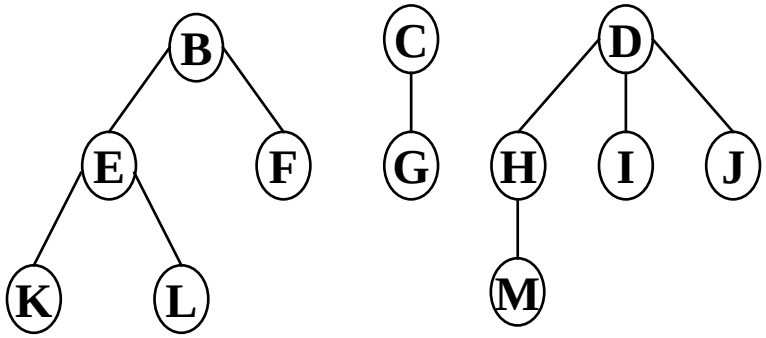


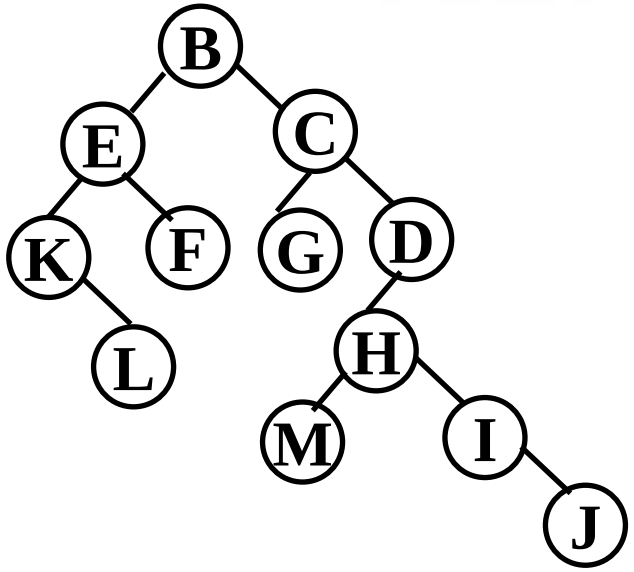




森林的二叉树表示

森林和二叉树的转换





练习：请给出该树的二叉树表示

森林和二叉树的转换

➔ 2、二叉树转换成森林

如果B =(root, LB, RB)是一棵二叉树，则可按如下规则转换成森林F = { T_1 , T_2, …, T_m }：

- (1)若B为空，则F为空；
- (2)若B非空，则F中的第一棵树T_1的根ROOT(T_1)即为二叉树B的根root； T_1 中根结点的子树森林F_1 是由左子树LB转换而成的森林；F中除T_1 之外其余树组成的森林F’ ＝{T_2, T_3,…, T_m }是由B 的右子树RB转换而成的森林。

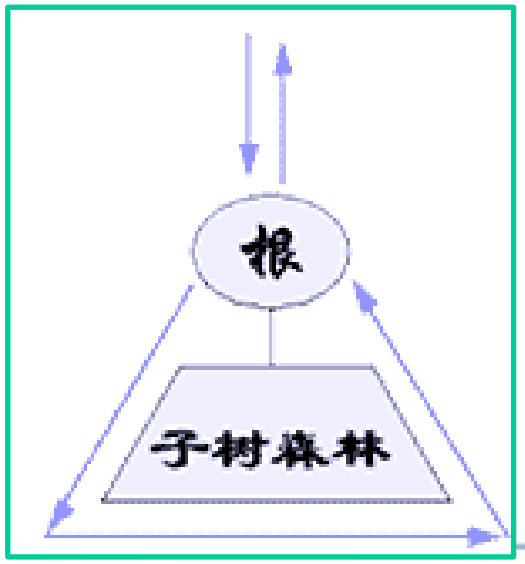
- 从上述递归定义容易写出相互转换的递归算法。同时，森林和树的操作亦可转换成二叉树的操作来实现。

树和森林的遍历

●树的遍历

和二叉树的遍历类似，树的遍历问题亦为，从根结点出发，对树中各个结点进行一次且仅进行一次访问。

对树进行遍历可有两条搜索路径：一条是从左到右（这里的左右指的是在存储结构中自然形成的子树之间的次序），另一条是按层次从上到下。对树进行从左到右遍历的搜索路径如下图所示。



树和森林的遍历

- 类似于二叉树，在这条搜索路径上途经根结点两次，由对根的访问时机不同可得下列两个算法：

  - 一、先根(次序)遍历树

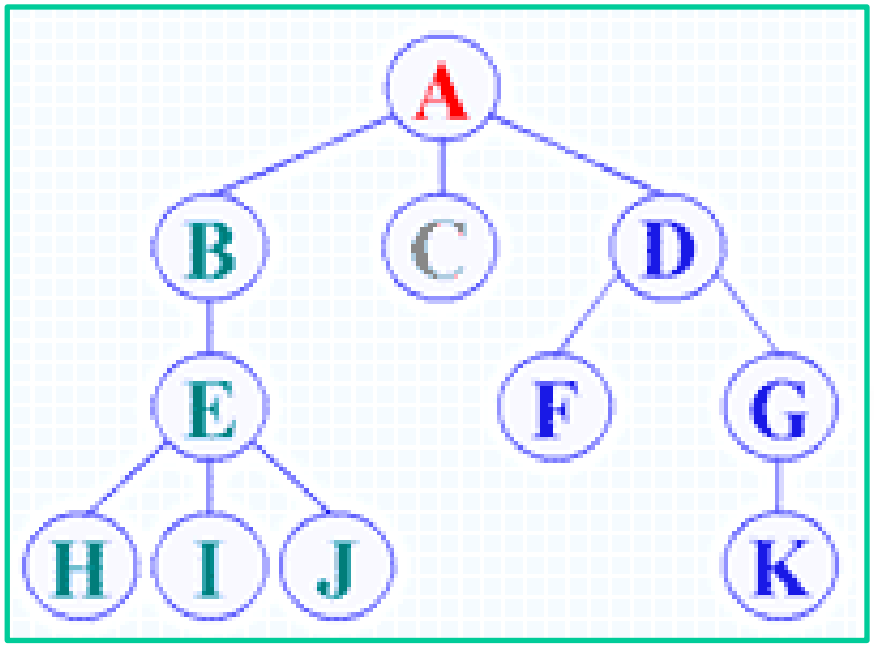
    若树不空，则先访问根结点，然后依次从左到右先根遍历根的各棵子树； 演示6-7-1.swf

  - 二、后根(次序)遍历树

    若树不空，则先依次从左到右后根遍历根的各棵子树，然后访问根结点； 演示6-7-2.swf

树和森林的遍历

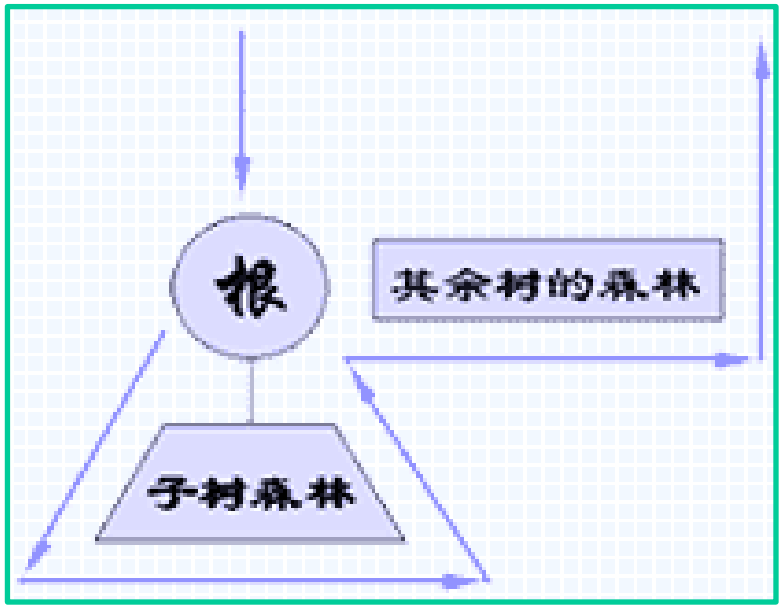
- 例如
  - 对下图进行先根遍历所得结点的访问序列为：ABEHIJCDFGK，
  - 进行后根遍历所得结点的访问序列为：HIJEBCFKGDA。
  - 演示一6-7-3.swf
  - 演示二 6-7-4.swf



树和森林的遍历

●森林的遍历

森林是树的集合，由此可以对森林中的每一棵树依次从左到右（如下图所示）进行先根遍历或者后根遍历。又森林中的（第一棵树的根）、（第一棵树的子树森林）及（其余树构成的森林），分别对应为（二叉树的根）、（二叉树的左子树）和（二叉树的右子树）。



树和森林的遍历

- 如下定义森林的这两种遍历：
  - 一、先序遍历森林

    若森林不空，则可依下列次序进行遍历：
    - (1) 访问森林中第一棵树的根结点；
    - (2) 先序遍历第一棵树中的子树森林；
    - (3) 先序遍历除去第一棵树之后剩余的树构成的森林。
    - 演示6-7-5.swf
  - 二、中序遍历森林

    若森林不空，则可依下列次序进行遍历：
    - (1) 中序遍历第一棵树中的子树森林；
    - (2) 访问森林中第一棵树的根结点；
    - (3) 中序遍历除去第一棵树之后剩余的树构成的森林。
    - 演示6-7-6.swf

树和森林

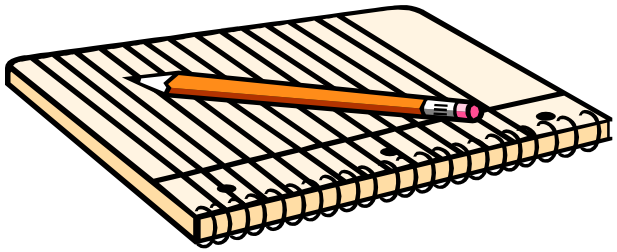
- 容易看出：

  树的先根遍历，即森林的先序遍历可对应到二叉树的先序遍历；

  树的后根遍历，即森林的中序遍历可对应到二叉树的中序遍历。

- 换句话说，若以孩子-兄弟链表作树（或森林）的存储结构，则树的先根遍历（或森林的先序遍历）的算法和二叉树的先序遍历算法类似，而树的后根遍历（或森林的中序遍历）的算法和二叉树的中序遍历算法类似。

赫夫曼树及其应用

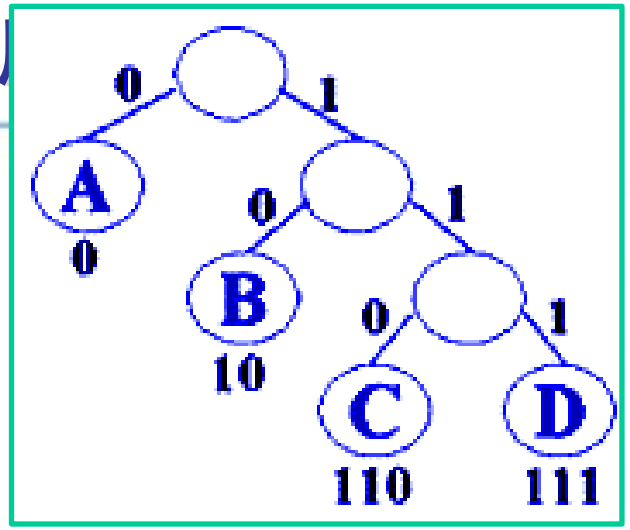


- 6.1 树的定义和基本术语
- 6.2 二叉树
- 6.3 遍历二叉树和线索二叉树
- 6.4 树和森林
- 6.5 赫夫曼树及其应用

赫夫曼树

- 赫夫曼（Huffman）树，又称最优树，是一类带权路径长度最短的树。

赫夫曼树及其应



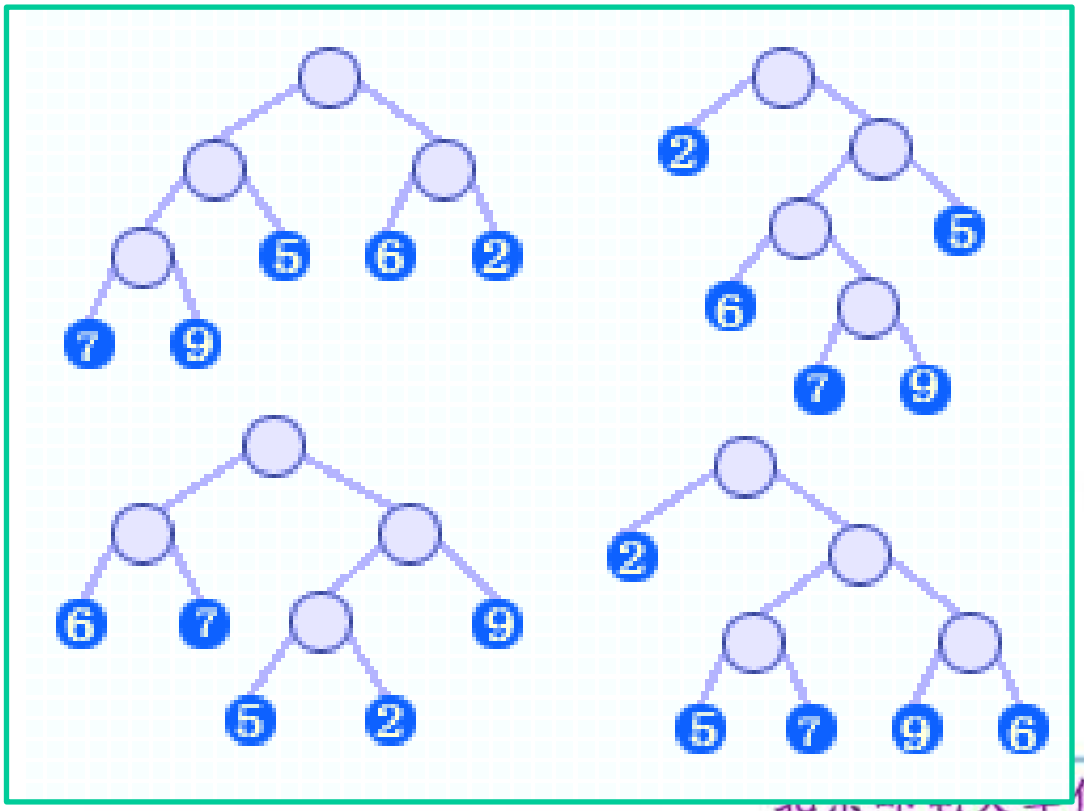
- 最优二叉树（赫夫曼树）
  - 路径：从树中一个结点到另一个结点之间的分支构成这两个结点之间的路径；
  - 路径长度：路径上的分支数目叫路径长度。
  - 结点的路径长度：从根结点到该结点的路径上分支的数目。
  - 树的路径长度指的是从树根到树中其余每个结点的路径长度之和。6.2节中定义的完全二叉树就是这种路径长度最短的二叉树。
  - 结点的带权路径长度则定义为从树根到该结点之间的路径长度与该结点上所带权值的乘积。假设树上有 n 个叶子结点，且每个叶子结点上带有权值为 w_k (k=1,2,…,n)，则树的带权路径长度定义为树中所有叶子结点的带权路径长度之和，通常记作：

$$WPL = \sum_{k=1}^{n} w_k l_k$$

其中：l_k 为带权w_k的叶子结点的路径长度。

赫夫曼树及其应用

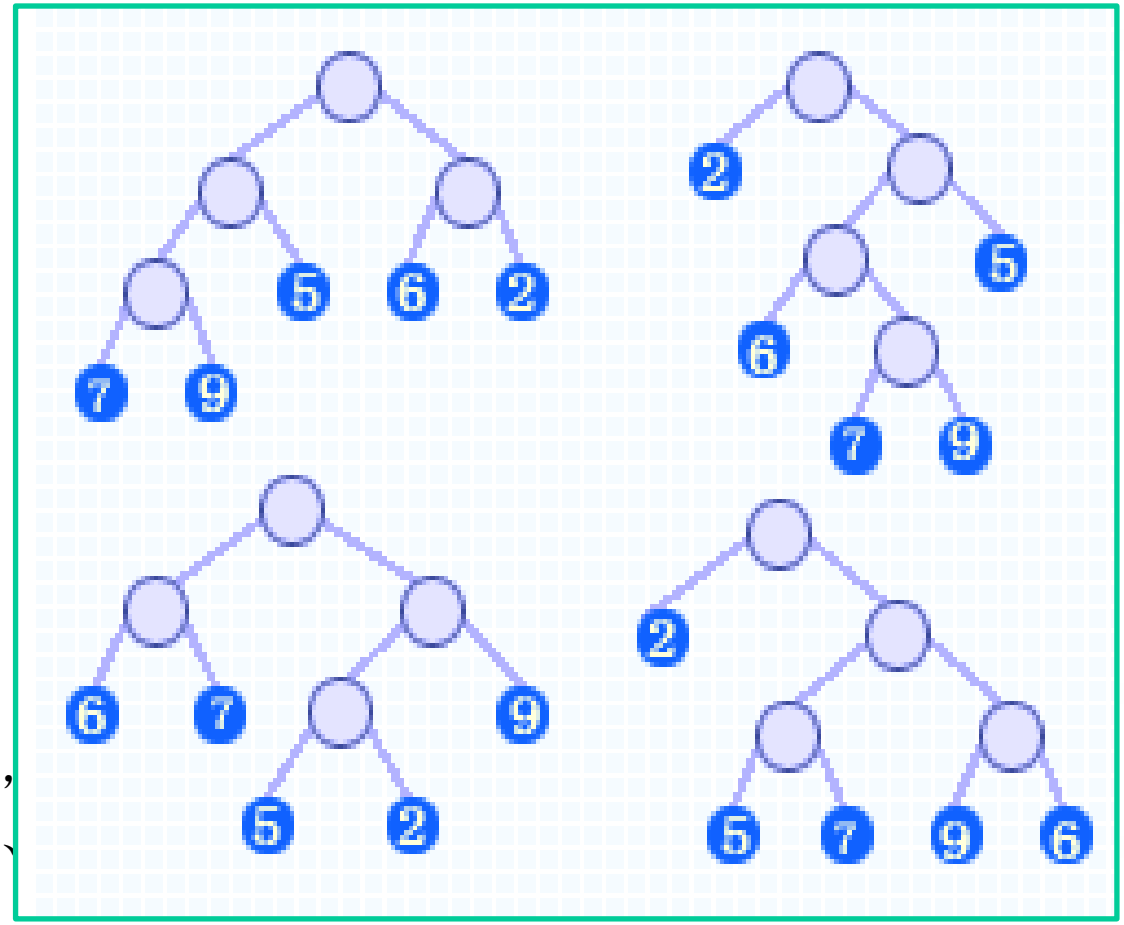
- 假设有 n 个权值{ w_1，w_2 ，… w_n }，试构造一棵有 n 个叶子结点的二叉树，每个叶子结点带权为w_i 。显然，这样的二叉树可以构造出多棵，其中必存在一棵**带权路径长度 WPL 取最小的二叉树**，称该二叉树为**最优二叉树**。
- 例如，右图中的四棵二叉树，都有5个叶子结点且带相同权值5、6、2、9、7。



北京邮电大学信息安全中心

赫夫曼树及其应用

- WPL = 7×3 + 9×3 + 5×2 + 6×2 + 2×2 = 74 (左上图)
- WPL = 2×1 + 6×3 + 7×4 + 9×4 + 5×2 = 94 (右上图)
- WPL = 6×2 + 7×2 + 5×3 + 2×3 + 9×2 = 65 (左下图)
- WPL = 2×1 + 5×3 + 7×3 + 9×3 + 6×3 = 83 (右下图)
- 其中，以左下图中二叉树的带权路径长度为最小。
- 可以验证，它恰为最优二叉树，即在所有叶子结点带权为5、6、2、9、7的二叉树中，带权路径长度的最小值为65。



赫夫曼树及其应用

●最优树的构造方法

赫夫曼最早给出了一个构造最优树的带有一般规律的算法，俗称赫夫曼算法。现以最优二叉树为例叙述如下：

- (1) 根据给定的 n 个权值$\{w_1, w_2, \ldots w_n\}$，构成 n 棵二叉树的集合$F=\{T_1, T_2, \ldots, T_n\}$，其中每棵二叉树$T_i$中只有一个带权为$w_i$的根结点，其左右子树均空。
- (2) 在 F 中选取两棵根结点的权值最小的树作为左右子树，构造一棵新的二叉树，且置新的二叉树的根结点的权值为其左、右子树上根结点的权值之和。
- (3) 在 F 中删除这两棵树，同时将新得到的二叉树加入 F 中。

重复(2)和(3)，直到 F 只含一棵树为止。这棵树便是所求的赫夫曼树。

●例如，对5个权值 {5,6,2,9,7} 构造最优二叉树的过程如动画所示。演示6-8-1.swf

赫夫曼树及其应用

●最优前缀编码

赫夫曼二叉树在通讯编码中的一个应用是利用它构造一组最优前缀编码。在某些通讯场合，需将传送的文字转换成由二进制字符组成的字符串。

通常有两类二进制编码：

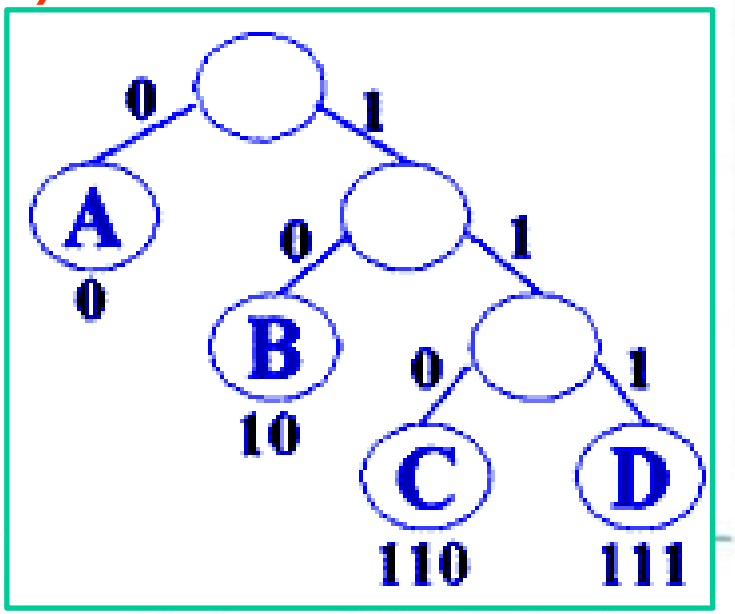
- 一类为等长编码，这类编码的二进制串的长度取决于电文中不同的字符个数，假设需传送的电文中只有四种字符，只需两位字符的串便可分辨，但如果电文中可能出现26种不同字符，则等长编码串的长度为5。
- 例如，假设需传送的电文为“ABACCDA”，它只有A、B、C和D四种字符，可设它们的编码分别为00、01 、10和11，则上述七个字符的电文便为 “000100101100”，总长14位。显然，这样的等长编码便于译码，对方接收时，可按两位一分进行译码。

赫夫曼树及其应用

- 另一类是**不等长编码**，即各个字符的编码长度不等。
  - 不等长编码的好处是，可以使传送电文的字符串的总长度尽可能地短。
  - 在实用的不等长编码中，任意一个字符的编码都不能是另一个字符的编码的前缀，这种编码称为**前缀编码**。
  - 例如根据上述电文中的四个字符A、B、C和D在电文中出现的多少为它们设计的编码分别为0、00、1和01，则上述七个字符“ABACCDA”的电文可转换成总长为9的字符串“000011010”。
  - 但是，这样的编码产生一个新的问题，即如何反译成原文，除非在每个字符之间加上空格符，否则将产生多义性。例如上述字符串中前四个字符的子串“0000”就可有多种译法，或“AAAA”，或是“ABA”，也可以是“BB”等。

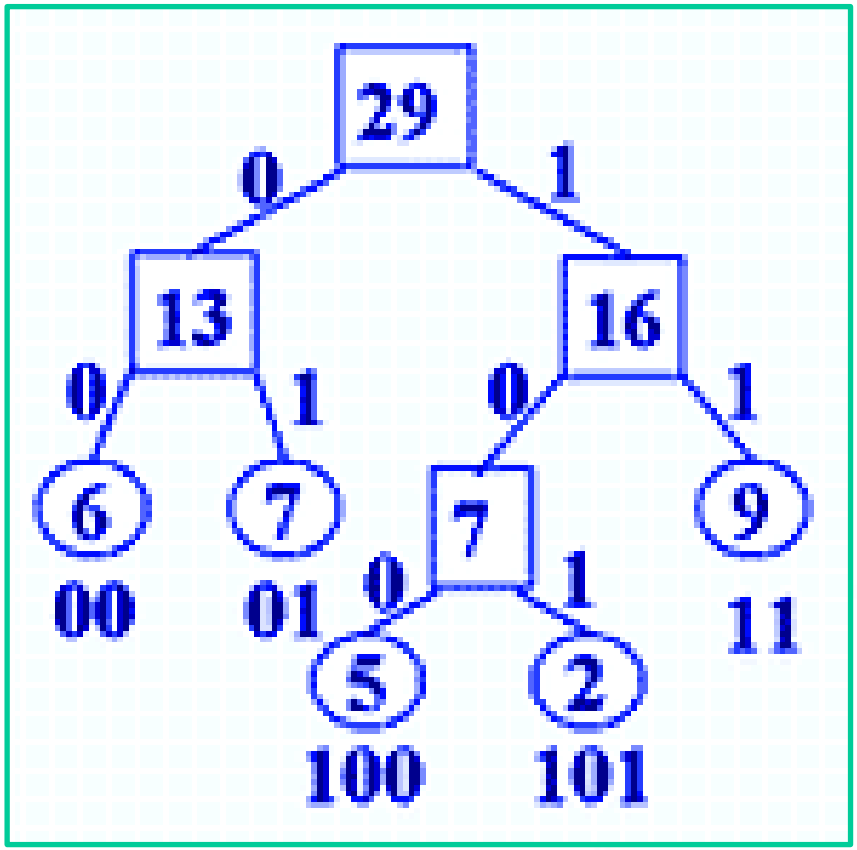
赫夫曼树及其应用

- 可以**利用二叉树来设计二进制的前缀编码**。
  - 下图所示二叉树，四个叶子结点分别表示A、B、C和D四个字符，约定左分支表示字符‘0’，右分支表示字符‘1’，则以由从根到叶子的路径上的分支表示的字符组成的字符串作为该叶子结点字符的编码。
  - 如下图中A、B、C和D的二进制前缀编码分别为0、10、110和111。
  - 并且，若以字符出现的次数为权，构造一棵赫夫曼树，由此得到的二进制前缀编码便为**“最优前缀编码”(赫夫曼编码)**。
  - **即以这组编码传送电文可使电文总长最短。**



赫夫曼树及其应用

- 假设电文总只有5个字符，且在电文中出现的频率分别为：5/29，6/29，2/29，9/29，7/29。则所构造的最优前缀编码如下图所示。



赫夫曼编码实例

- /* 赫夫曼树和赫夫曼编码的存储表示 */

```
typedef struct
{
  unsigned int weight;
  unsigned int parent,lchild,rchild;
}HTNode,*HuffmanTree; /* 动态分配数组存储赫夫曼树 */
typedef char **HuffmanCode; /* 动态分配数组存储赫夫曼编码表 */
```

初始化：有n个叶子结点的赫夫曼树共有2n-1个结点。
证：分支数=结点总数-1=2*度为2的结点数，
（N0+N1+N2）-1=2*N2可得：N0-1=N2

● /* 算法6.12 */

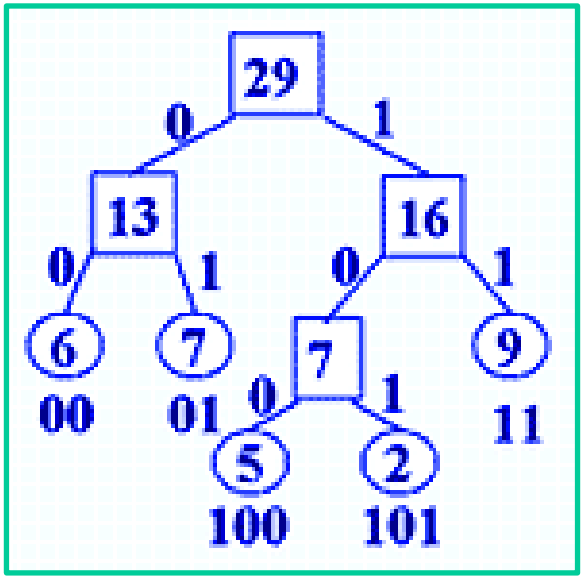
```
void HuffmanCoding(HuffmanTree *HT,HuffmanCode *HC,int *w,int n)
{ //w存放n个字符的权值(均>0)，构造赫夫曼树HT，并求出n个字符的赫夫曼编码HC
    int m,i,s1,s2,start;  unsigned c,f;  HuffmanTree p;  char *cd;
     if(n<=1)   return;  m=2*n-1;
    *HT=(HuffmanTree)malloc((m+1)*sizeof(HTNode)); /* 0号单元未用 */
    for(p=*HT+1,i=1;i<=n;++i,++p,++w)                          叶子结点初始化
    {   (*p).weight=*w;   (*p).parent=0;   (*p).lchild=0;   (*p).rchild=0;  }
    for(;i<=m;++i,++p)    (*p).parent=0;
                                                  非终端结点初始化
    for(i=n+1;i<=m;++i) /* 建赫夫曼树 */
    { //在HT[1～i-1]中选择parent为0且weight最小的两个结点，其序号分别为s1和s2
      select(*HT,i-1,&s1,&s2);
      (*HT)[s1].parent=(*HT)[s2].parent=i;
      (*HT)[i].lchild=s1;   (*HT)[i].rchild=s2;
      (*HT)[i].weight=(*HT)[s1].weight+(*HT)[s2].weight;
    }
```

赫夫曼编码实例

- /* 从叶子到根逆向求每个字符的赫夫曼编码 */

```
*HC=(HuffmanCode)malloc((n+1)*sizeof(char*));//分配n个字符编码的头指针向量([0]不用)
cd=(char*)malloc(n*sizeof(char)); /* 分配求编码的工作空间 */
cd[n-1]='\0'; /* 编码结束符 */
for(i=1;i<=n;i++)
{ /* 逐个字符求赫夫曼编码 */
  start=n-1; /* 编码结束符位置 */
  for(c=i,f=(*HT)[i].parent;f!=0;c=f,f=(*HT)[f].parent)     /* 从叶子到根逆向求编码 */
    if((*HT)[f].lchild==c)       cd[--start]='0';
          else     cd[--start]='1';
  (*HC)[i]=(char*)malloc((n-start)*sizeof(char));     /* 为第i个字符编码分配空间 */
  strcpy((*HC)[i],&cd[start]); /* 从cd复制编码(串)到HC */
}
free(cd); /* 释放工作空间 */
}
```

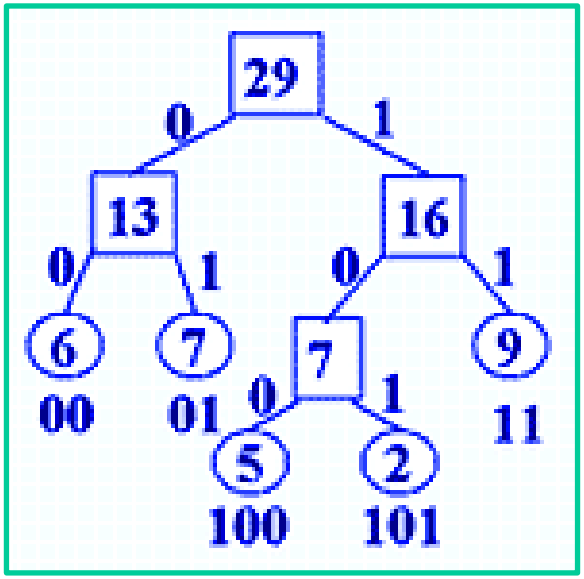
赫夫曼编码实例



序号	左孩子	右孩子	双亲	值
0				
1	0	0	0	2
2	0	0	0	5
3	0	0	0	6
4	0	0	0	7
5	0	0	0	9
6			0	
7			0	
8			0	
9			0	

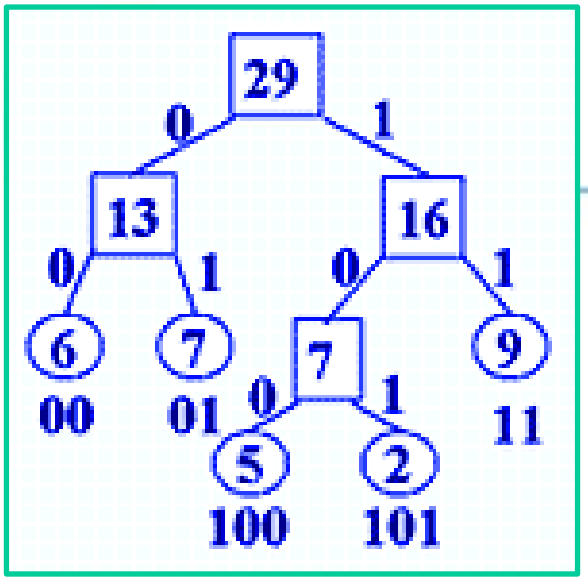
```
m=2*n-1;
for(p=*HT+1,i=1;i<=n;++i,++p,++w)
    {    (*p).weight=*w;    (*p).parent=0;    (*p).lchild=0;    (*p).rchild=0;   }
for(;i<=m;++i,++p)    (*p).parent=0;
```

赫夫曼编码实例



序号	左孩子	右孩子	双亲	值
0				
1	0	0	6	2
2	0	0	6	5
3	0	0	0	6
4	0	0	0	7
5	0	0	0	9
6	2	1	0	7
7			0	
8			0	
9			0	

```
for(i=n+1;i<=m;++i) /* 建赫夫曼树 */
    { //在HT[1～i-1]中选择parent为0且weight最小的两个结点，其序号分别为s1和s2
      select(*HT,i-1,&s1,&s2);
      (*HT)[s1].parent=(*HT)[s2].parent=i;
      (*HT)[i].lchild=s1;    (*HT)[i].rchild=s2;
      (*HT)[i].weight=(*HT)[s1].weight+(*HT)[s2].weight;
   }
```



序号	左孩子	右孩子	双亲	值
0				
1	0	0	6	2
2	0	0	6	5
3	0	0	7	6
4	0	0	7	7
5	0	0	8	9
6	2	1	8	7
7	3	4	9	13
8	6	5	9	16
9	7	8	0	29

以叶结点**2**为例分析编码过程

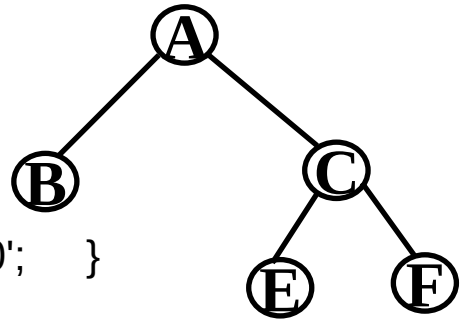
序号	0	1	2	3	4
编码		‘1’	‘0’	‘1’	0

```
start=n-1; /* 编码结束符位置 */
for(c=i,f=(*HT)[i].parent;f!=0;c=f,f=(*HT)[f].parent)     /* 从叶子到根逆向求编码 */
    if((*HT)[f].lchild==c)       cd[--start]='0';
                          else     cd[--start]='1';
  (*HC)[i]=(char*)malloc((n-start)*sizeof(char));     /* 为第i个字符编码分配空间 */
  strcpy((*HC)[i],&cd[start]); /* 从cd复制编码(串)到HC */
```

赫夫曼编码实例

- /* 算法6.13，无栈非递归遍历赫夫曼树，求赫夫曼编码*/

```
*HC=(HuffmanCode)malloc((n+1)*sizeof(char*));   /* 分配n个字符编码的头指针向量([0]不用) */
c=m;   cdlen=0;
for(i=1;i<=m;++i)     (*HT)[i].weight=0; /* 遍历赫夫曼树时用作结点状态标志 */
while(c)   {
  if((*HT)[c].weight==0)  { /* 向左 */
       (*HT)[c].weight=1;
       if((*HT)[c].lchild!=0)       {    c=(*HT)[c].lchild;        cd[cdlen++]='0';     }
      else if((*HT)[c].rchild==0)     { /* 登记叶子结点的字符的编码 */
           (*HC)[c]=(char *)malloc((cdlen+1)*sizeof(char)); cd[cdlen]='\0';
           strcpy((*HC)[c],cd); /* 复制编码*/  } }
  else if((*HT)[c].weight==1)  { /* 向右 */       (*HT)[c].weight=2;
       if((*HT)[c].rchild!=0)       {      c=(*HT)[c].rchild;      cd[cdlen++]='1';     } }
       else    { (*HT)[c].weight=0;                /* HT[c].weight==2，退回 */
                 c=(*HT)[c].parent;  --cdlen; /* 退到父结点，编码长度减1 */     }
}
```



本章小结

- 熟练掌握**二叉树的结构特性**。
- 熟悉二叉树的各种**存储结构**的特点及适用范围。
- **树和二叉树的遍历算法**是实现各种操作的基础。对非线性结构的遍历需要选择合适的搜索路径，以确保在这条路径上可以访问到结构中的所有数据元素，并使每一个数据元素只被访问一次。掌握各种遍历策略的递归算法和非递归算法，灵活运用遍历递归算法。

本章小结

- 遍历的实质是按某种规则将二叉树中的数据元素排列成一个线性序列，而**线索链表是通过遍历生成**的，即在遍历过程中保存结点之间的“前驱”和“后继”的关系，并为方便起见，在线索链表中添加一个“头结点”，并由此构成一个“双向循环链表”。
- **最优树和最优前缀编码的构造方法。**最优树是一种"带权路径长度最短"的树，最优前缀编码是最优二叉树的一种应用。

本章知识点与重点

- **知识点**

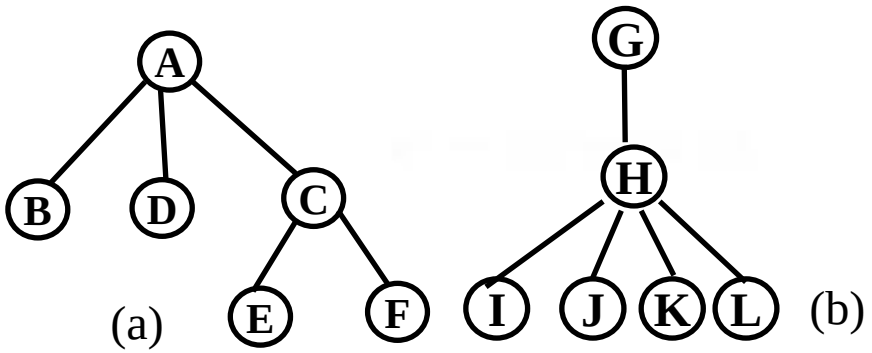
二叉树、线索二叉树、树和森林的存储表示、树和森林的遍历以及其它操作的实现、最优树和赫夫曼编码

- **重点和难点**

二叉树和树的遍历及其应用是本章的学习重点，而编写实现二叉树和树的各种操作的递归算法也恰是本章的难点所在 。

本章练习

- 假设一棵二叉树的先序序列为EBADCFHGIKJ，中序序列为ABCDEFGHIJK。请画出此二叉树。
- 假设用于通信的电文仅有8个字母组成，出现频率分别为0.07, 0.19, 0.02, 0.06, 0,32, 0.03, 0.21, 0.10。试为这8个字母设计赫夫曼编码。
- 编写递归算法，将二叉树中所有结点的左、右子树相互交换。
- 如下图所示的森林：
- (1) 求树(a)的先根序列和后根序列；
- (2) 求森林先序序列和中序序列；
- (3)将此森林转换为相应的二叉树。



本章练习

- 深度为k的二叉树至多有_____个结点。
- 将二叉树T转换为对应的森林F，已知二叉树T中叶结点、没有左孩子的结点和没有右孩子的结点数分别为m，n，和k，请问是否能根据这些信息指明F中叶结点的个数？若能，F中叶结点的个数是？若不能，还需要知道二叉树的什么信息？
- 设有n个结点的完全二叉树的深度为_____；如果按照从自上到下、从左到右从1开始顺序编号，则第i个结点的双亲结点编号为_____，右孩子结点的编号为_____。
- 已知一棵树的广义表表示为（A（C，D（E，F，G），H（I，J））），请画出这课树。