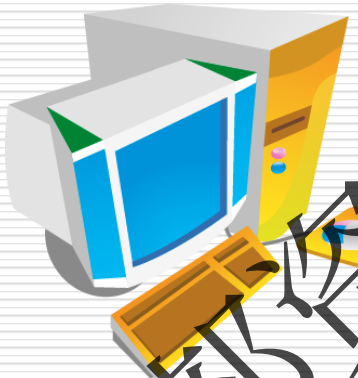
数据结构

第五章　树和二叉树



人工智能学院

刘运

哈夫曼树及哈夫曼编码

在远程通讯中，要将数据文件转换成二进制串，怎样编码才能使它们组成的报文在网络中传得最快？

A	00
B	01
C	10
D	11

ABACCDAAAA

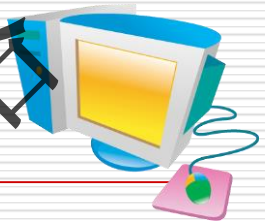
A	0
B	00
C	1
D	01

00010010101100000000

0000110100000

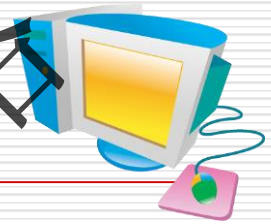
出现次数较多的字符采用尽可能短的编码

主要内容



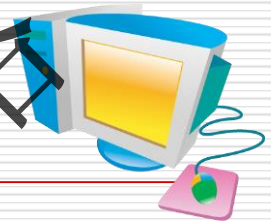
- 5.1 树和二叉树的定义
- 5.2 案例引入
- 5.3 树和二叉树的类型定义
- 5.4 二叉树的性质和顺序结构
- 5.5 遍历二叉树和线索二叉树
- 5.6 树和森林
- 5.7 哈夫曼树及其应用
- 5.8 案例分析与实现

学习要点



- 掌握二叉树的结构特性、各种存储结构和各种遍历算法；
- 理解二叉树线索化的实质，了解二叉树线索化的过程；
- 熟悉树的各种存储结构及特点，掌握树和森林与二叉树的转换方法。
- 学会编写实现二叉树的各种操作的算法；
- 掌握建立哈夫曼树和构造哈夫曼编码的方法。

5.1 树和二叉树的定义



5.1.1 树的定义

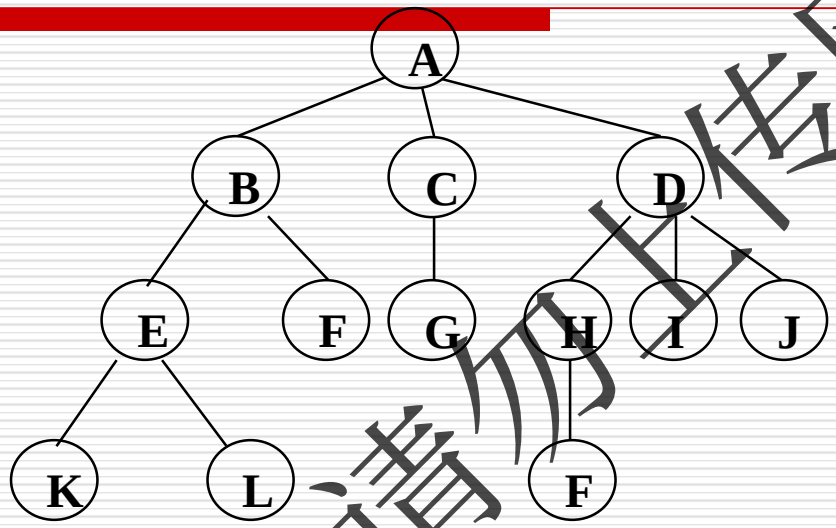
树（Tree）是n(**n≥0**）个结点的有限集，它或为空树（n=0）；或为非空树，对于非空树：

（1）有且仅有一个称之为**根（root）**的结点；

（2）除根结点以外的其余结点可分为m(m≥0)个互不相交的有限集合T1,T2，...，Tm。其中每个集合本身又是一棵树，被称为根的**子树（SubTree）** 。

注：树的定义具有递归性，即“树中还有树”。

树的逻辑结构



从逻辑结构看：

1）树中只有根结点没有前驱；

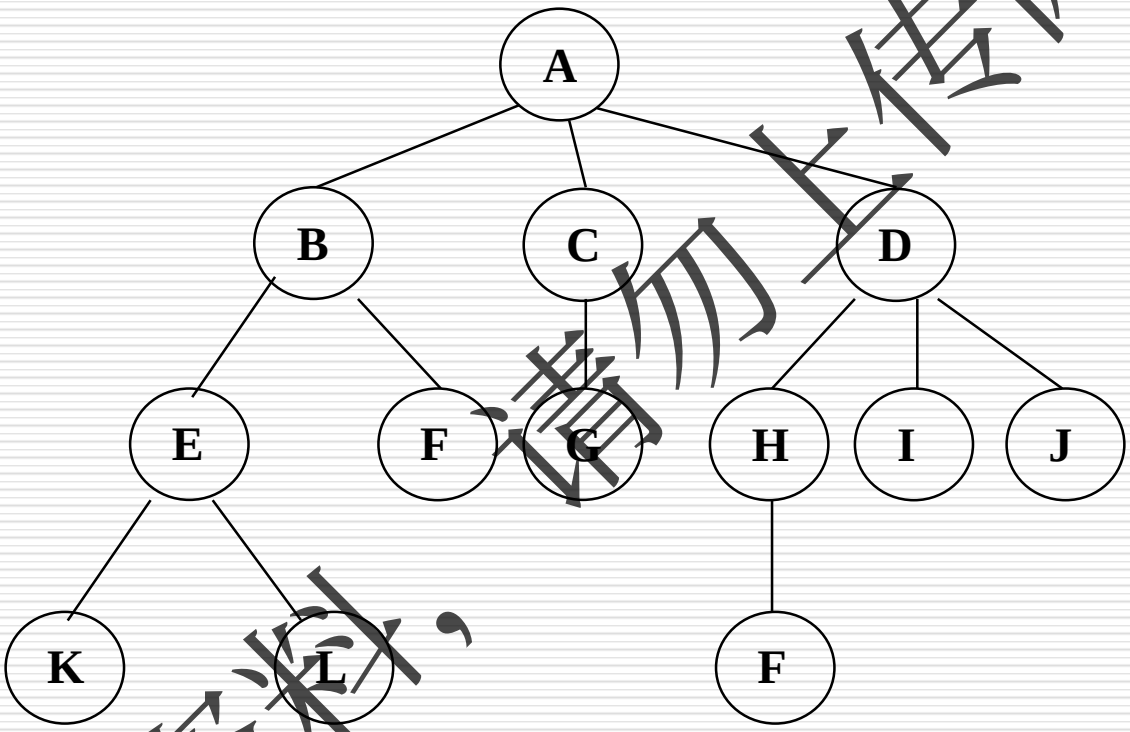
2）除根外，其余结点都有且仅一个前驱；

3）一对多（1:n），树的结点可以有零个或多个后继；

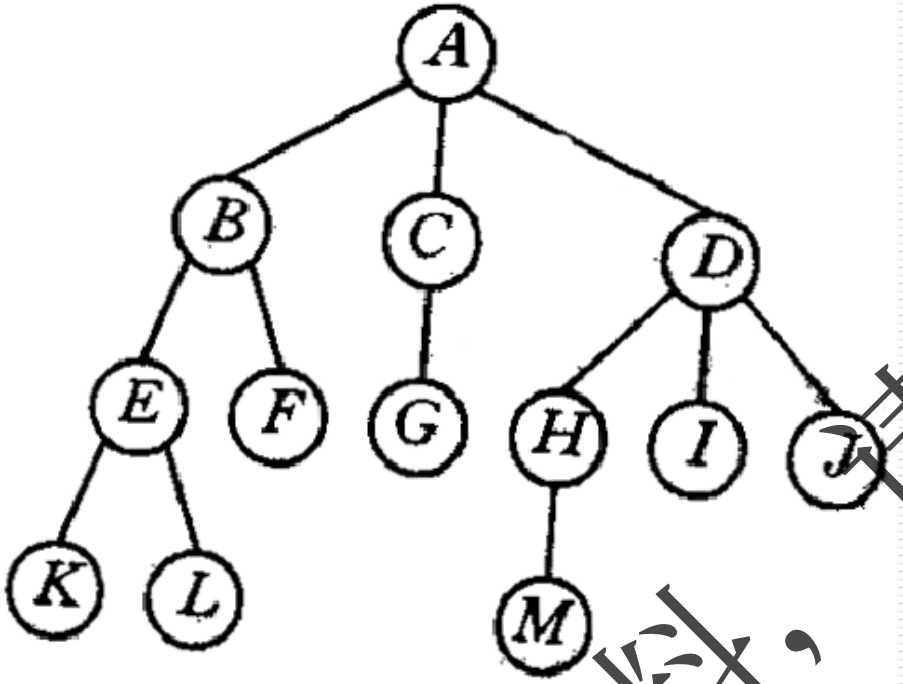
4）除根外的其他结点，都存在唯一条从根到该结点的路径；

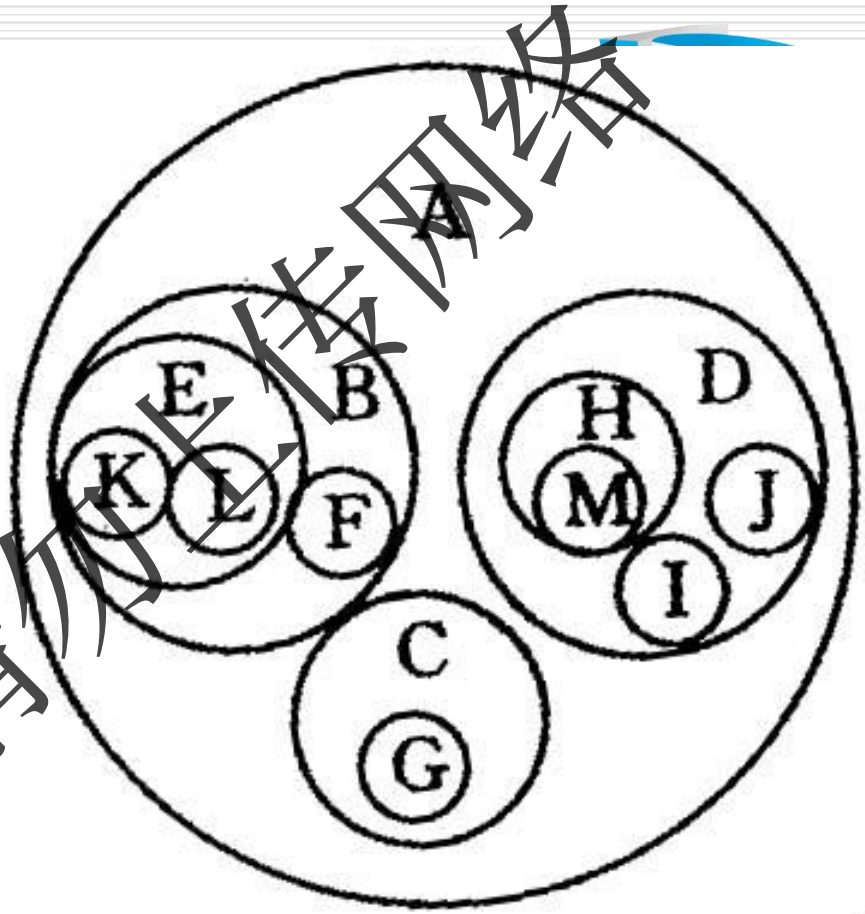
5）树是一种分枝结构

图形表示法

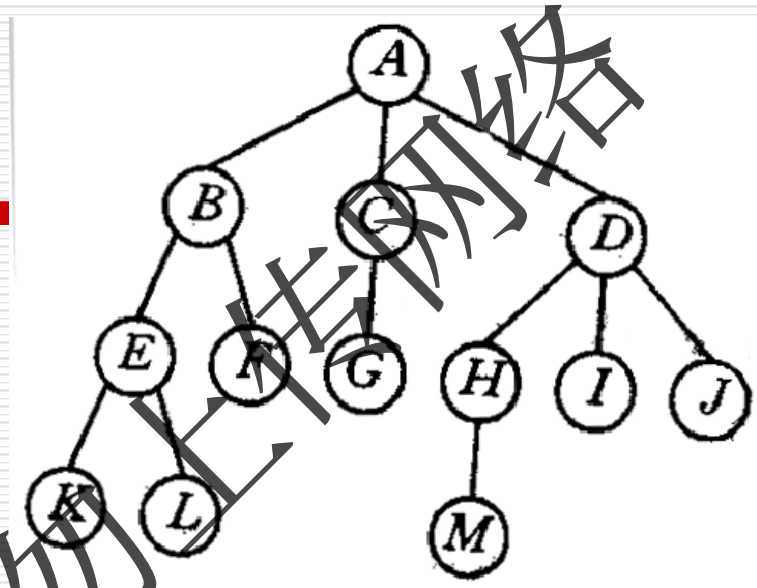


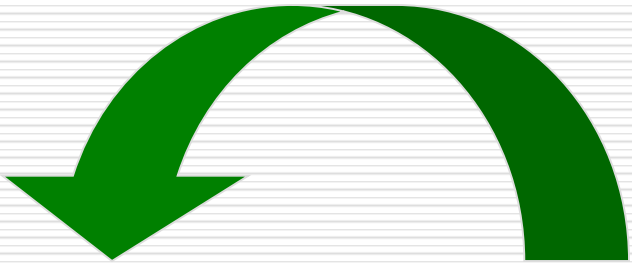
嵌套集合表示法





广义表表示法





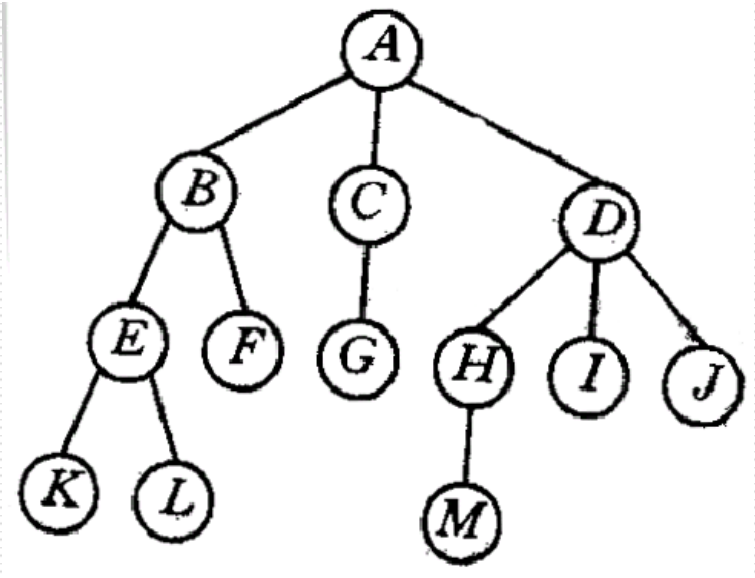
(A (B (E (K, L), F), C (G), D (H (M), I, J)))

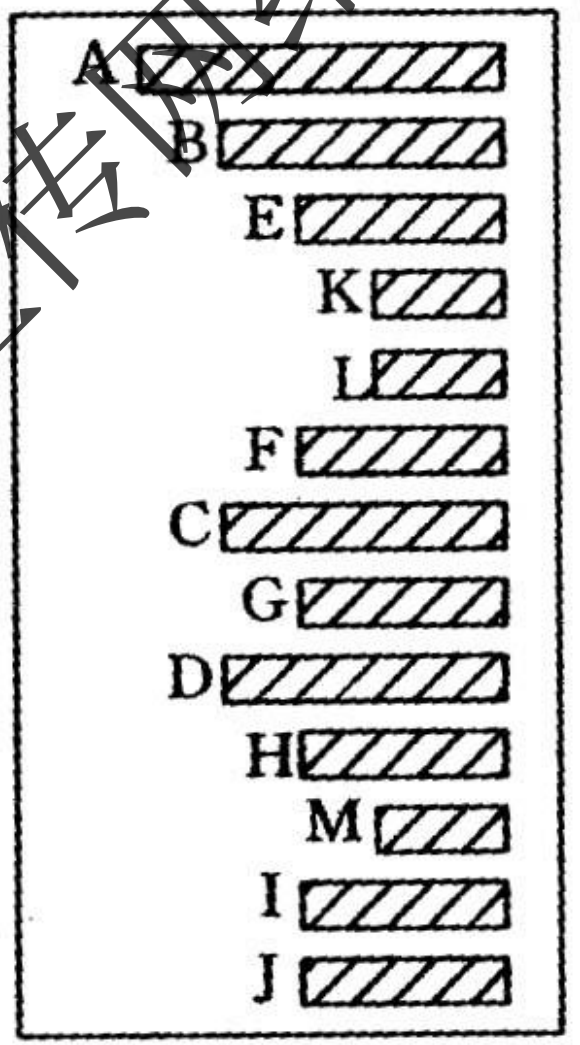
约定：

根作为由子树森林组成的表的名字写在表的左边

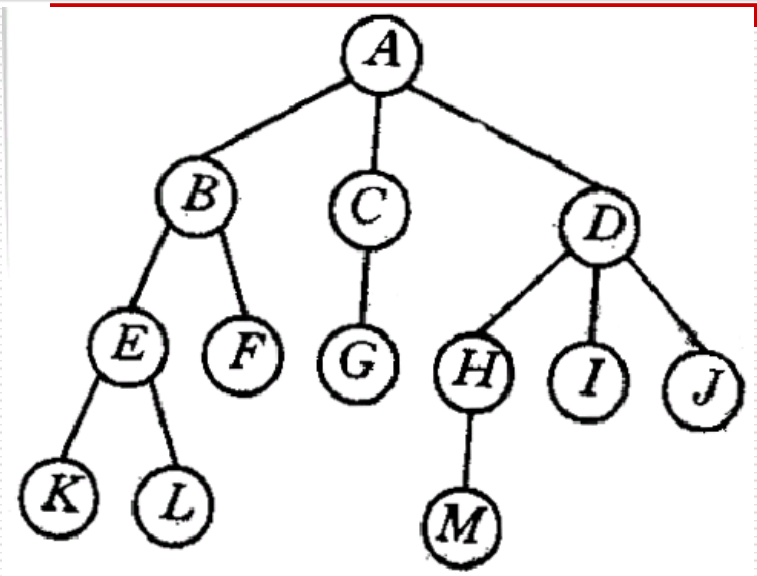
凹入表示法

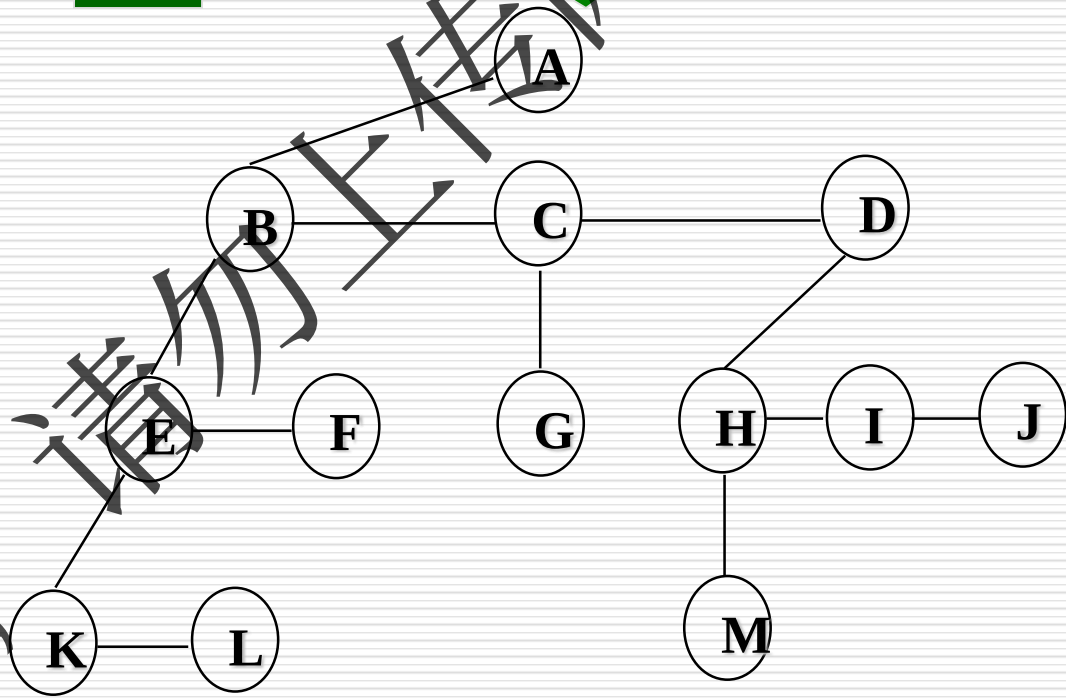
又称目录表示法





左孩子—右兄弟表示法

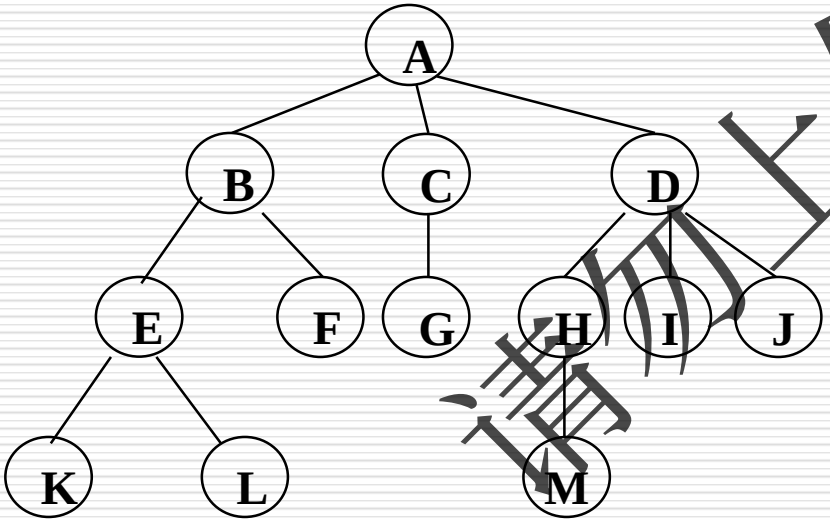




data	
左孩子	右兄弟

多叉树转为了二叉树

5.1.2 树的基本术语



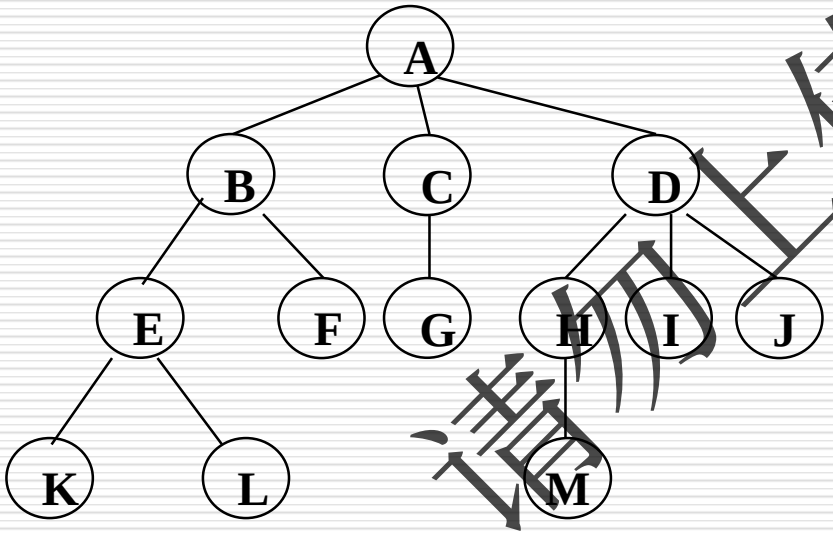
结点——树中的一个独立单元，即树的数据元素。

结点的度——结点拥有的子树数。

树的度——所有结点度中的最大值（Max{各结点的度}）。

叶子——度为0的结点，也称为终端结点。

非终端结点——度不为零的结点，也称为分支结点。



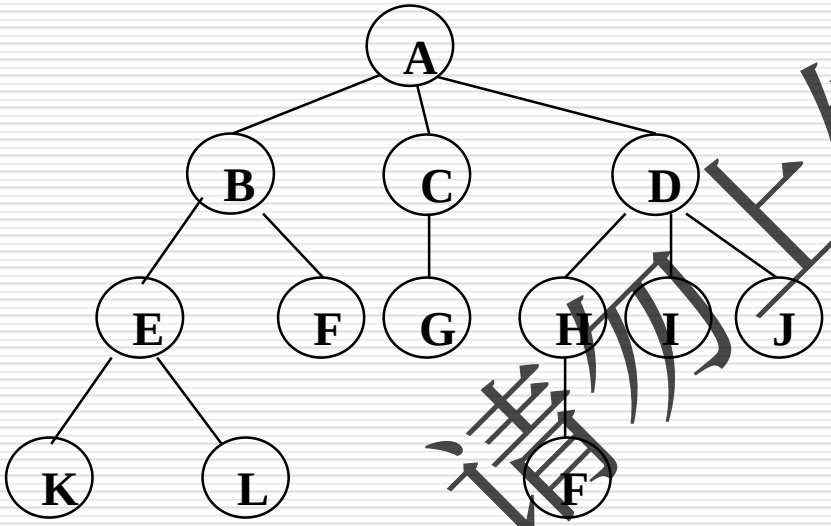
双亲和孩子——结点的子树的根称为该结点的孩子，相应地，该结点称为孩子的双亲。

兄弟——同一双亲的孩子之间互称兄弟。

祖先——即从根到该结点所经分支的所有结点。

子孙——该结点下层子树中的任一结点。

层次——从根到该结点的层数（根结点算第一层）。



堂兄弟——双亲在同一层的结点互为堂兄弟。

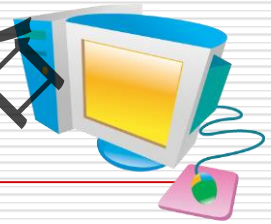
树的深度——指所有结点中最大的层数（Max{各结点的层次}）。

有序树——结点各子树从左至右有序，不能互换（左为第一）

无序树——结点各子树无序，可互换位置。

森林——指m（m>=0）棵不相交的树的集合。

5.1.3 二叉树的定义



二叉树（Binary Tree）是n($n \geq 0$)个结点所构成的集合，它或为空树（n=0）；或为非空树，对于非空树：

（1）有且仅有一个称之为**根（root）**的结点；

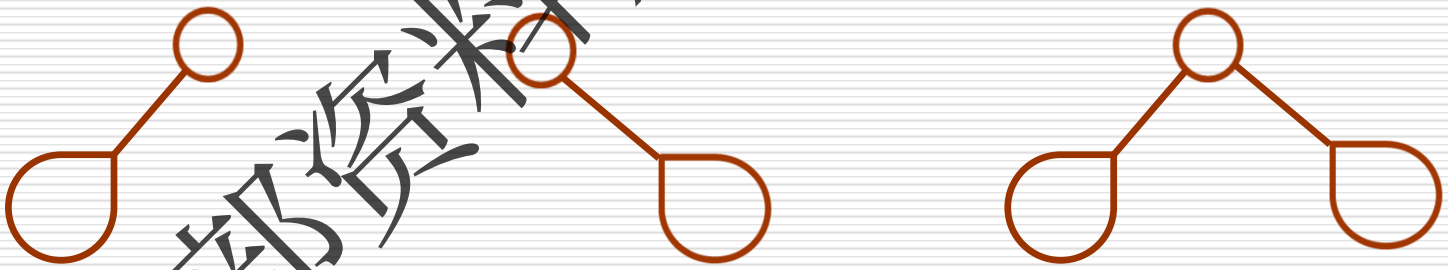
（2）除根结点以外的其余结点可分为两个互不相交的子集T1和T2，分别称为根的左子树和右子树，且T1和T2本身又都是二叉树。

二叉树的基本特征：

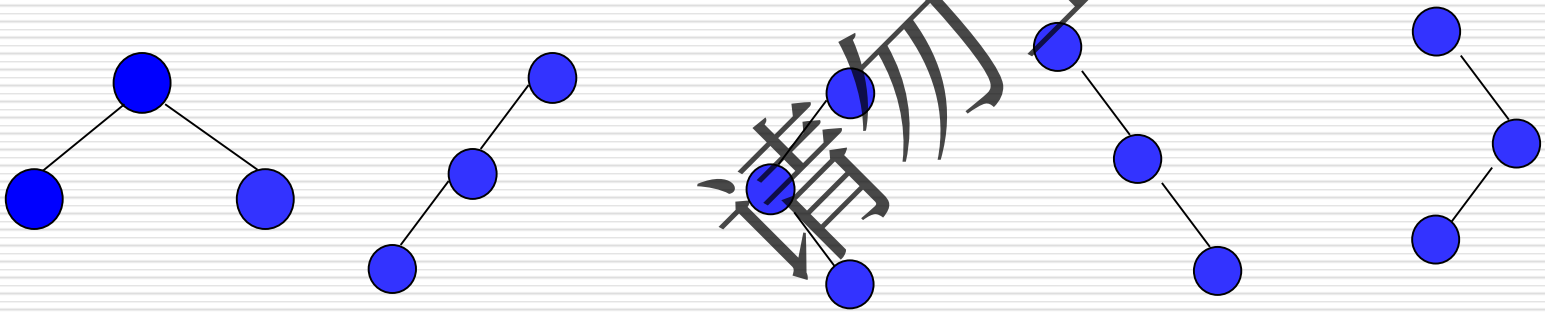
① 每个结点最多只有两棵子树（不存在度大于2的结点）；

② 左子树和右子树次序不能颠倒。

基本形态：



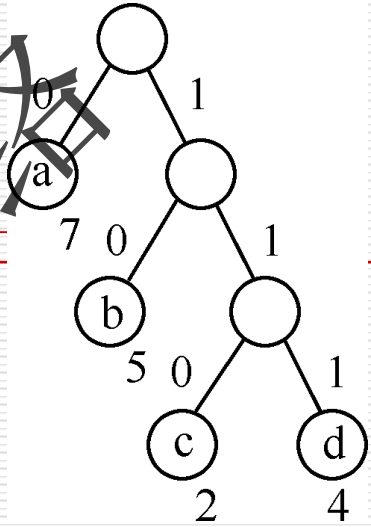
问：具有3个结点的二叉树可能有几种不同形态？



有5种

3个结点的树有
几种不同形态？

5.2 案例引入



案例5.1 ：数据压缩问题

将数据文件转换成由0、1组成的二进制串，称之为编码。

（a）等长编码方案

字符	编码
a	00
b	01
c	10
d	11

（b）不等长编码方案1

字符	编码
a	0
b	10
c	110
d	111

（c）不等长编码方案2

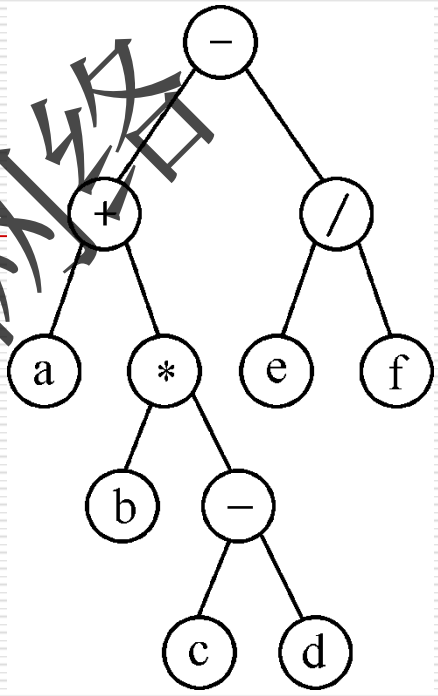
字符	编码
a	0
b	01
c	010
d	111

案例5.2 ：利用二叉树求解表达式的值

以二叉树表示表达式的递归定义如下：

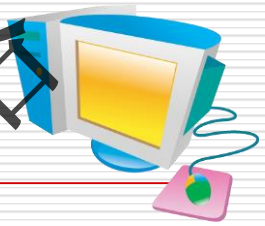
（1）若表达式为数或简单变量，则相应二叉树中仅有一个根结点，其数据域存放该表达式信息；

（2）若表达式为“第一操作数 运算符 第二操作数”的形式，则相应的二叉树中以左子树表示第一操作数，右子树表示第二操作数，根结点的数据域存放运算符（若为一元运算符，则左子树为空）。其中，操作数本身又为表达式。



(a + b *(c-d)-e/f)的二叉树

5.3 树和二叉树的抽象数据类型定义



5.3.1 树的抽象数据类型定义

ADT Tree{

数据对象D: D是具有相同特性的数据元素的集合。

数据关系R: 若D为空集，则称为空树；//允许n=0

若D中仅含一个数据元素，则R为空集；

其他情况下的R存在二元关系：

① root 唯一 //关于根的说明

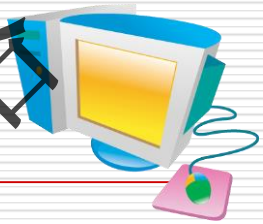
② $D_j \cap D_k = \Phi$ //关于子树不相交的说明

③ …… //关于数据元素的说明

基本操作P: //15个，如求树深，求某结点的双亲

}ADT Tree

5.3.2 二叉树的抽象数据类型定义



ADT BinaryTree{ D是具有相同特性的数据元素的集合。

数据对象D: 若D=Φ，则R= Φ；

数据关系R: 若D≠Φ，则R= {H}；存在二元关系：

① root 唯一 //关于根的说明

② $D_j \cap D_k = \Phi$ //关于子树不相交的说明

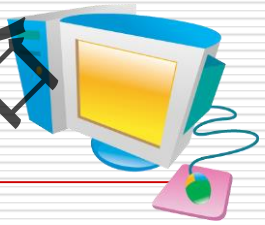
③ …… //关于数据元素的说明

④ …… //关于左子树和右子树的说明

基本操作 P: //至少有20个，如返回某结点的左孩子，或中序遍历，等等

}ADT BinaryTree

二叉树的重要操作



CreateBiTree(&T,definition)

初始条件：definition给出二叉树T的定义。

操作结果：按definition构造二叉树T。

PreOrderTraverse(T)

初始条件：二叉树T存在。

操作结果：先序遍历T，对每个结点访问一次。

InOrderTraverse(T)

初始条件：二叉树T存在。

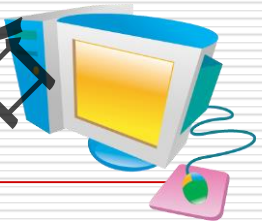
操作结果：中序遍历T，对每个结点访问一次。

PostOrderTraverse(T)

初始条件：二叉树T存在。

操作结果：后序遍历T，对每个结点访问一次。

5.4 二叉树的性质和存储结构



5.4.1 二叉树的性质

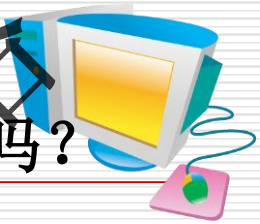
讨论1：第i层的结点数最多是多少？

性质1：在二叉树的第i层上至多有2^{i-1}个结点（i>0）。

讨论2：深度为k的二叉树，最多有多少个结点？

性质2：深度为k的二叉树至多有2^k-1个结点（k>0）。

讨论3：二叉树的叶子数和度为2的结点数之间有关系吗？



性质3：对于任何一棵二叉树，若2度的结点数有n_2个，则叶子数（n_0）必定为n_2+1 （即$n_0=n_2+1$）

物理意义：叶子数=2度结点数＋1

证明：

∵ 二叉树中全部结点数$n=n_0+n_1+n_2$（叶子数＋1度结点数＋2度结点数）

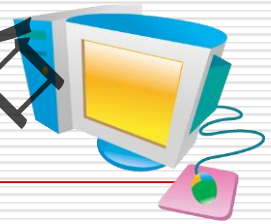
又∵二叉树中全部结点数$n=B+1$ （ 总分支数＋根结点 ）

除根结点外，每个结点必有一个直接前趋，即一个分支）

而 总分支数$B=n_1+2n_2$（1度结点必有1个直接后继，2度结点必有2个）

三式联立可得：$n_0+n_1+n_2=n_1+2n_2+1$，即$n_0=n_2+1$

课堂练习：



1. 树T中各结点的度的最大值称为树T的__D__。

A）高度　B）层次　C）深度　D）度

2.深度为K的二叉树的结点总数，最多为C个。

A)2^{k-1}　B)$\log_2 k$　C)2^k-1　D)2^k

3. 深度为9的二叉树中至少有C个结点。

A)2^9　B)2^8　C)9　D)2^9-1

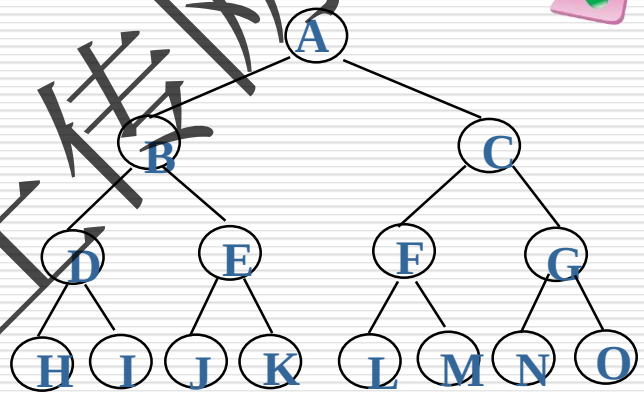
满二叉树： 一棵深度为k 且有2^k -1个结点的二叉树。

特点：每层都“充满”了结点

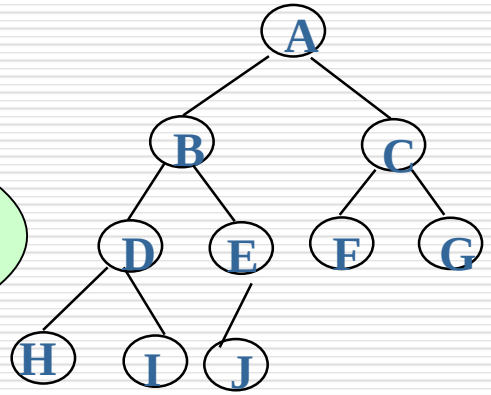
完全二叉树： 深度为k 的、有n个结点的二叉树，当且仅当其每一个结点都与深度为k 的满二叉树中编号从1至n的结点一一对应。

为何要研究这两种特殊形式？

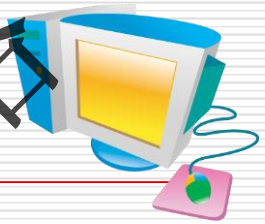
因为它们在顺序存储方式下可以复原！



深度为4的满二叉树



完全二叉树



性质4:

具有n个结点的完全二叉树的深度必为$\lfloor \log_2 n \rfloor + 1$.

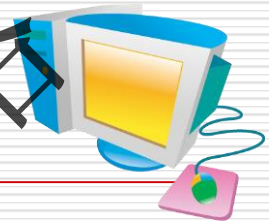
证明：根据性质2，深度为k的二叉树最多只有2^k-1个结点，且完全二叉树的定义是与同深度的满二叉树前面编号相同，即它的总结点数n位于k层和k-1层满二叉树容量之间，

即 $2^{k-1}-1 < n \leqslant 2^k-1$ 或$2^{k-1} \leqslant n < 2^k$

三边同时取对数，于是有：

$$k-1 \leqslant \log_2 n < k$$

因为k是整数，所以$k=\lfloor \log_2 n \rfloor + 1$



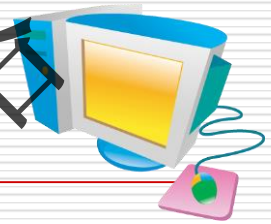
性质5： 对完全二叉树，若从上至下、从左至右编号，则编号为*i* 的结点，其左孩子编号必为*2i*，其右孩子编号为*2i*＋*1*；其双亲的编号必为*i/2*（*i*＝*1* 时为根，除外）。

可根据归纳法证明。

满二叉树和完全二叉树的区别

满二叉树是叶子一个也不少的树，而完全二叉树虽然前k-1层是满的，但最底层却允许在右边缺少连续若干个结点。满二叉树是完全二叉树的一个特例。

例：



一棵完全二叉树有1000个结点，则它必有_____个叶子结点，有_____个度为2的结点，有_____个结点只有非空左子树，有_____个结点只有非空右子树。

分析题意：已知n=1000，求n_0和n_2，还要判断末层叶子是挂在左边还是右边？

请注意：叶子结点总数≠末层叶子数！！！

正确答案：

全部叶子数＝⌈1000/2⌉ =500个。

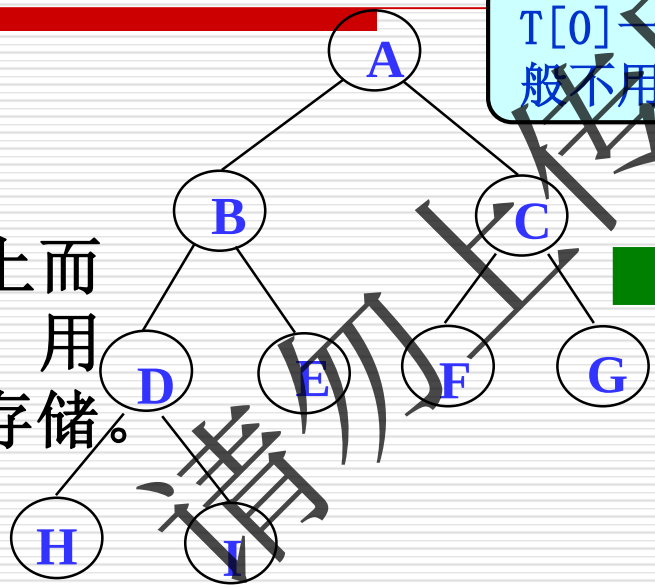
度为2的结点＝叶子总数－1=499个。

因为最后一个结点坐标是偶数，所以必为左子树。

5.4.2 二叉树的存储结构

1 顺序存储结构

按二叉树的结点“自上而下、从左至右”编号，用一组连续的存储单元存储。



T[0]一般不用

[1]	A
[2]	B
[3]	C
[4]	D
[5]	E
[6]	F
[7]	G
[8]	H
[9]	I

问：顺序存储后能否复原成唯一对应的二叉树？

答：若是完全/满二叉树则可以做到唯一复原。

1）若有左孩子，则左孩编号为2i；

2）若有右孩子，则有孩子结点编号为2i+1；

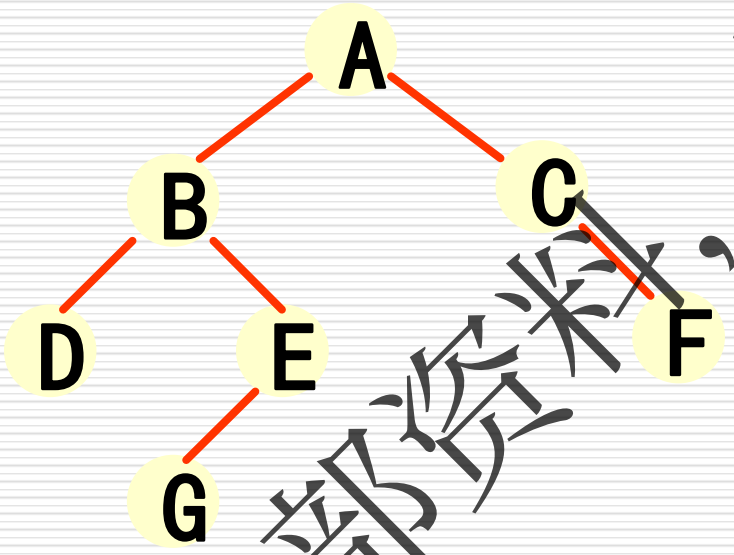
3）若有双亲，则双亲结点编号为i/2；

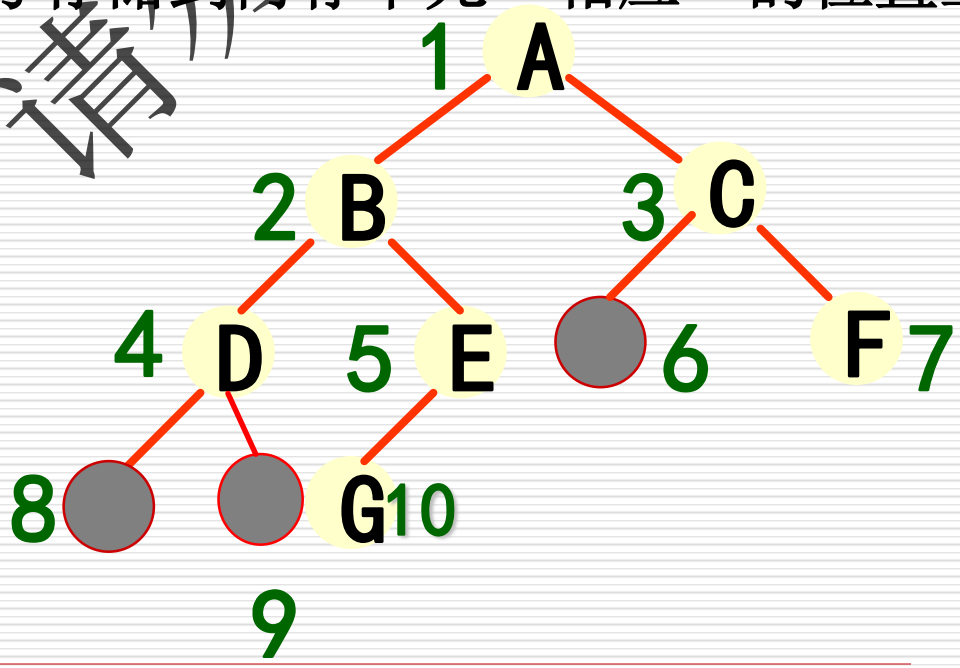
讨论：不是完全二叉树怎么办？

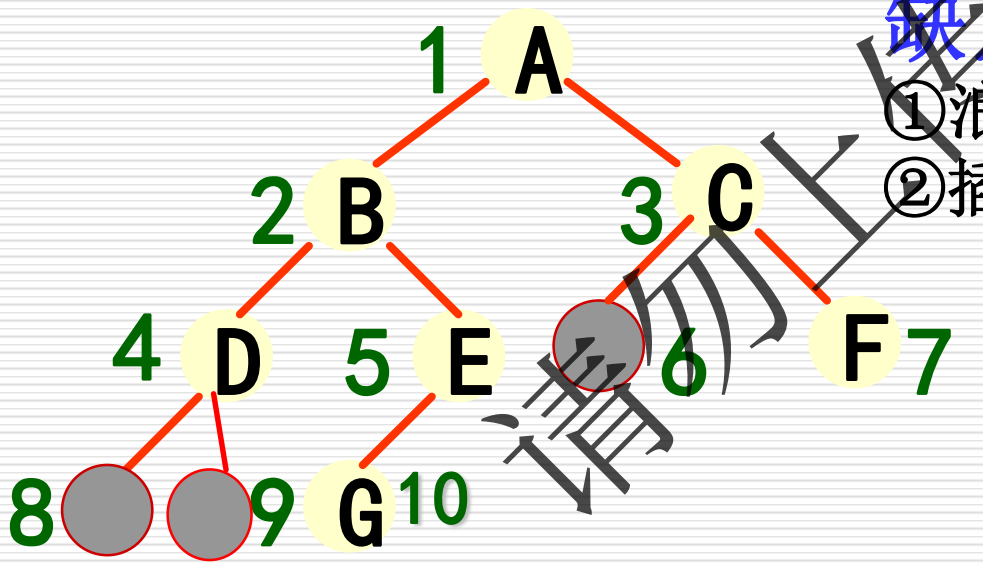
答：一律转为完全二叉树！

方法很简单，将各层空缺处统统补上“虚结点”，其内容为空。

将二叉树原有的结点按编号存储到内存单元“相应”的位置上。







缺点：

①浪费空间；

②插入、删除不方便

1	2	3	4	5	6	7	8	9	10		m-1
A	B	C	D	E		F			G	…	

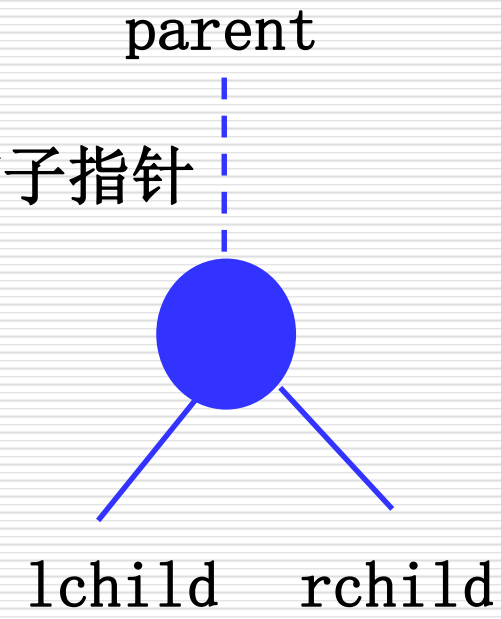
二叉树的顺序结构图示

2 二叉链表

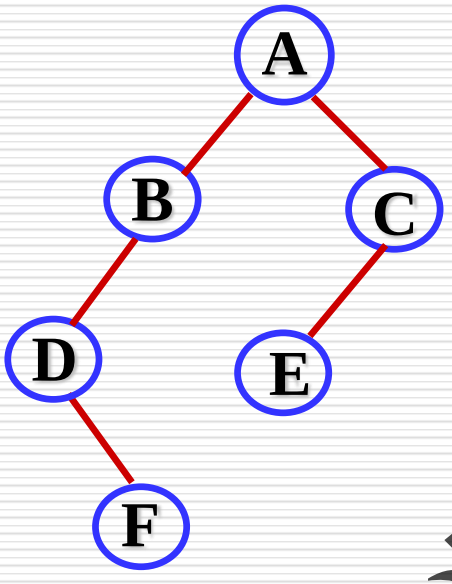
每个结点至少包含三个域：数据域、左指针域、右指针域

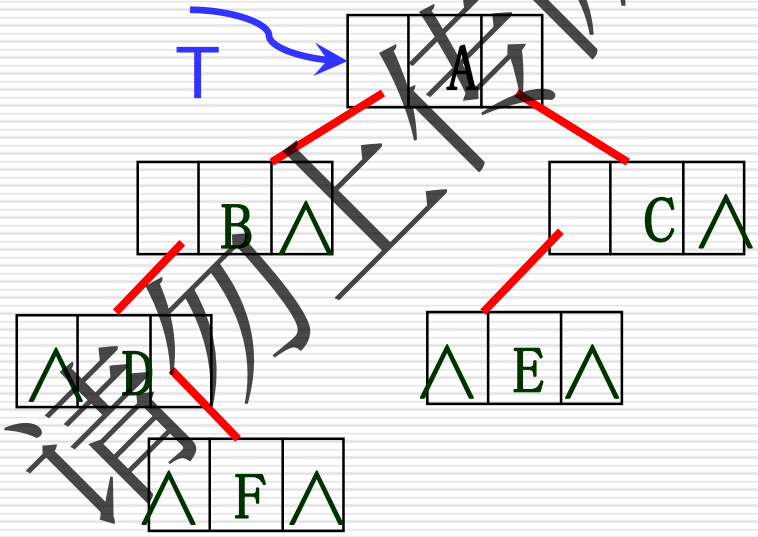
```
typedef struct BiTNode{
  TElemType  data;
  struct BiTNode  *lchild, *rchild; //左右孩子指针
}BiTNode, *BiTree;
```

含有两个指针域的结点结构



二叉链表示例



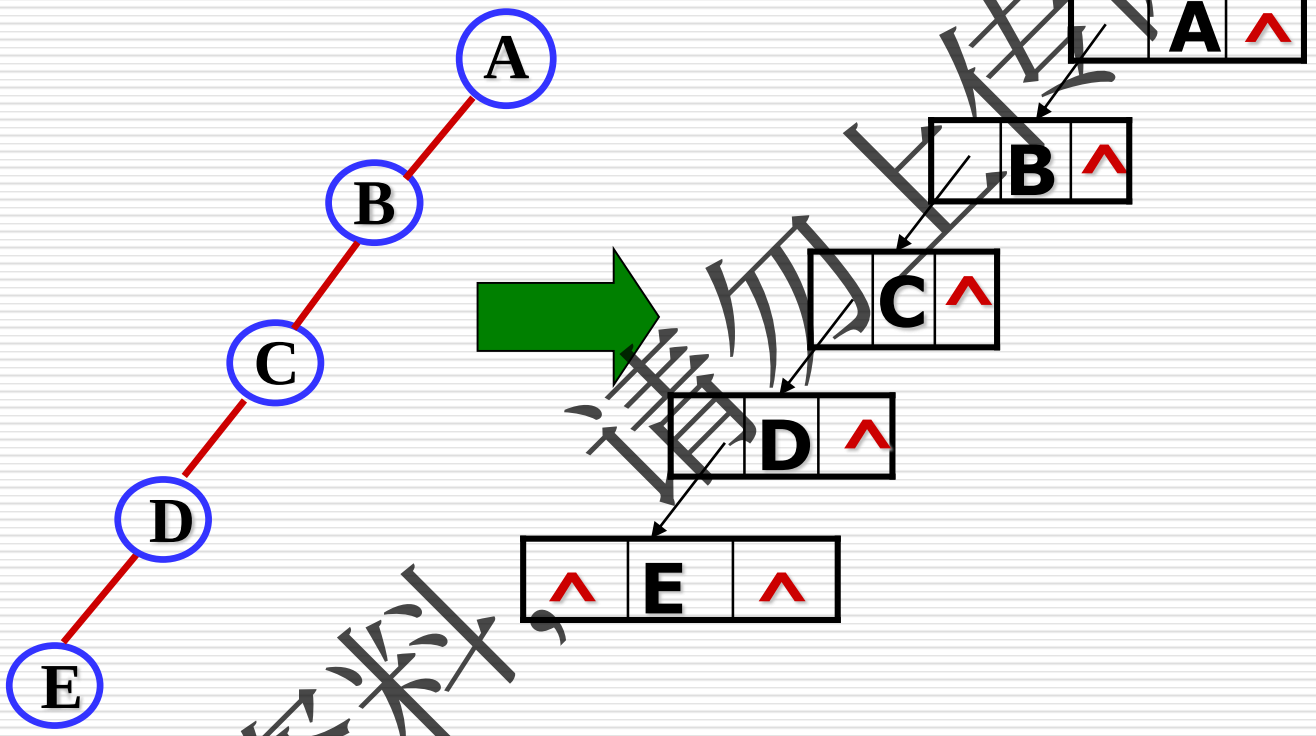


二叉链表图示

在含有n个结点的二叉链表中有n+1个空链域。

（证明留作课后作业）

二叉树链式存储举例：



优点：①不浪费空间；
②插入、删除方便

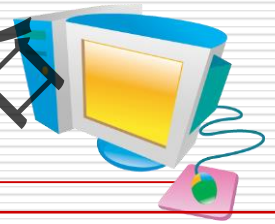
3 三叉链表

三叉链表中每个结点至少包含四个域：

数据域、双亲指针域、左指针域、右指针域

含有三个指针域的结点结构

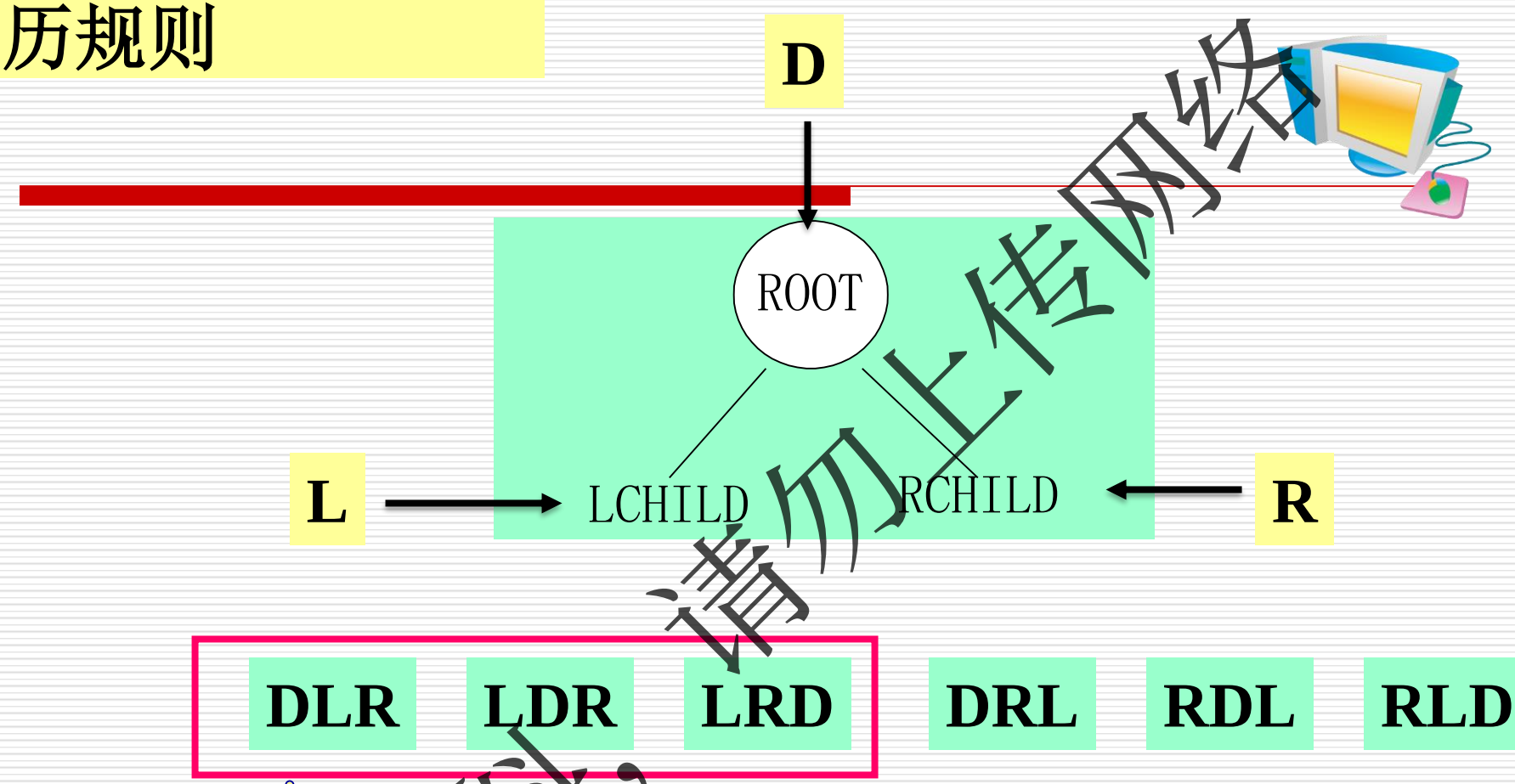
5.5 遍历二叉树和线索二叉树



遍历定义——指按某条搜索路线遍访每个结点且不重复（又称周游）。

遍历用途——它是树结构插入、删除、修改、查找和排序运算的前提，是二叉树一切运算的基础和核心。

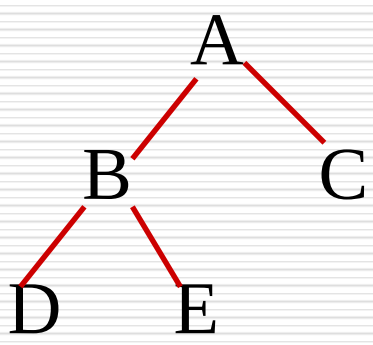
遍历规则



DLR **LDR** **LRD** **DRL** **RDL** **RLD**

先左后右

例1：



先序遍历的结果是：A B D E C

中序遍历的结果是：D B E A C

后序遍历的结果是：D E B C A

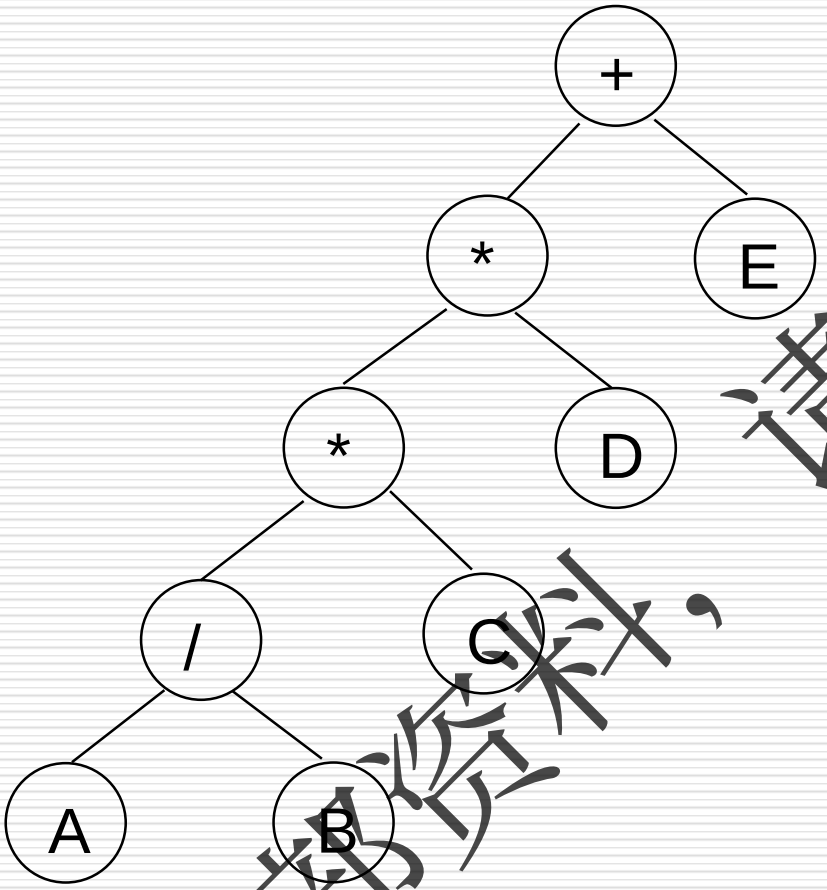
口诀：

DLR—先序遍历，即先根再左再右

LDR—中序遍历，即先左再根再右

LRD—后序遍历，即先左再右再根

用二叉树表示算术表达式



先序遍历

+ * * / A B C D E

前缀表示

中序遍历

A / B * C * D + E

中缀表示

后序遍历

A B / C * D * E +

后缀表示

层序遍历

+ * E * D / C A B

遍历的算法实现—先序遍历

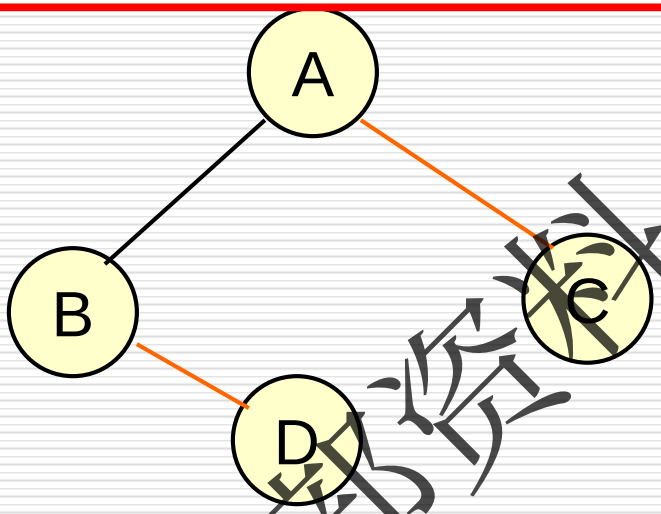
若二叉树为空，则空操作

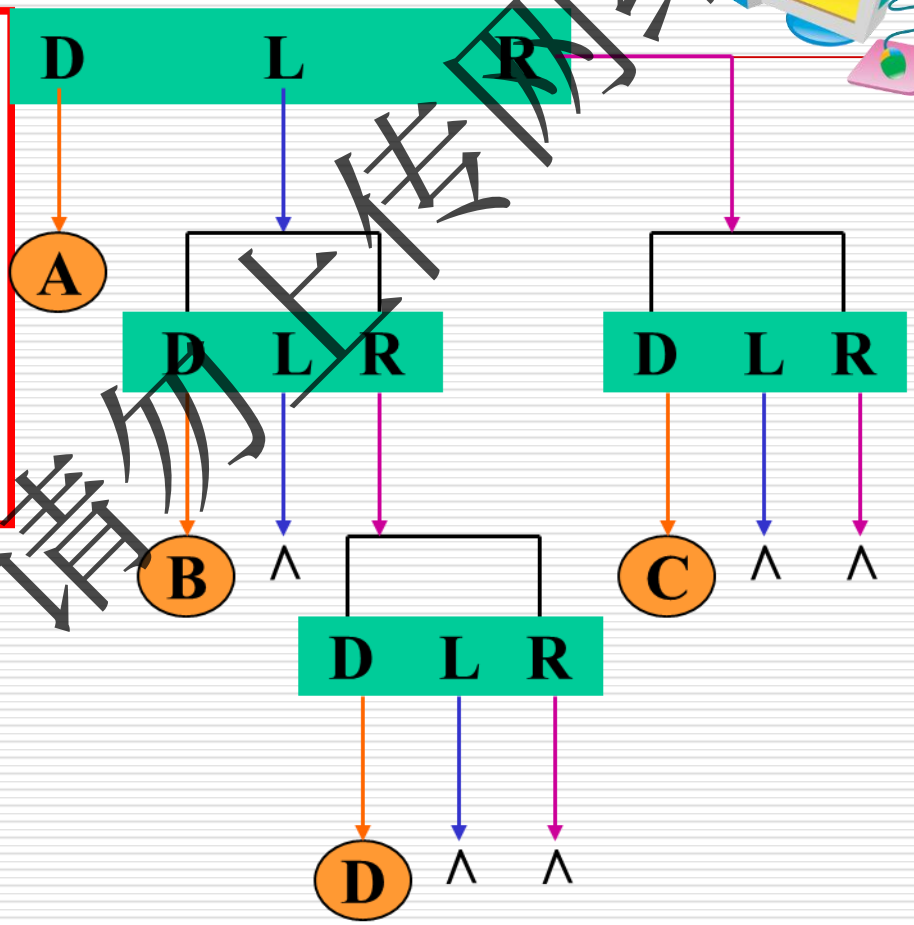
否则

访问根结点（D）

前序遍历左子树（L）

前序遍历右子树（R）





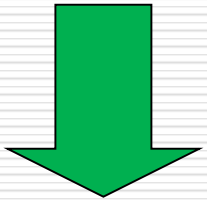
先序遍历序列：A B D C

遍历的算法实现——用递归形式格外简单！

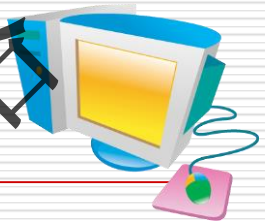
回忆：

```
long Factorial ( long n ) {
    if ( n == 0 ) return 1;//基本项
    else return n * Factorial (n-1); //归纳项}
```

则三种遍历算法可写出：



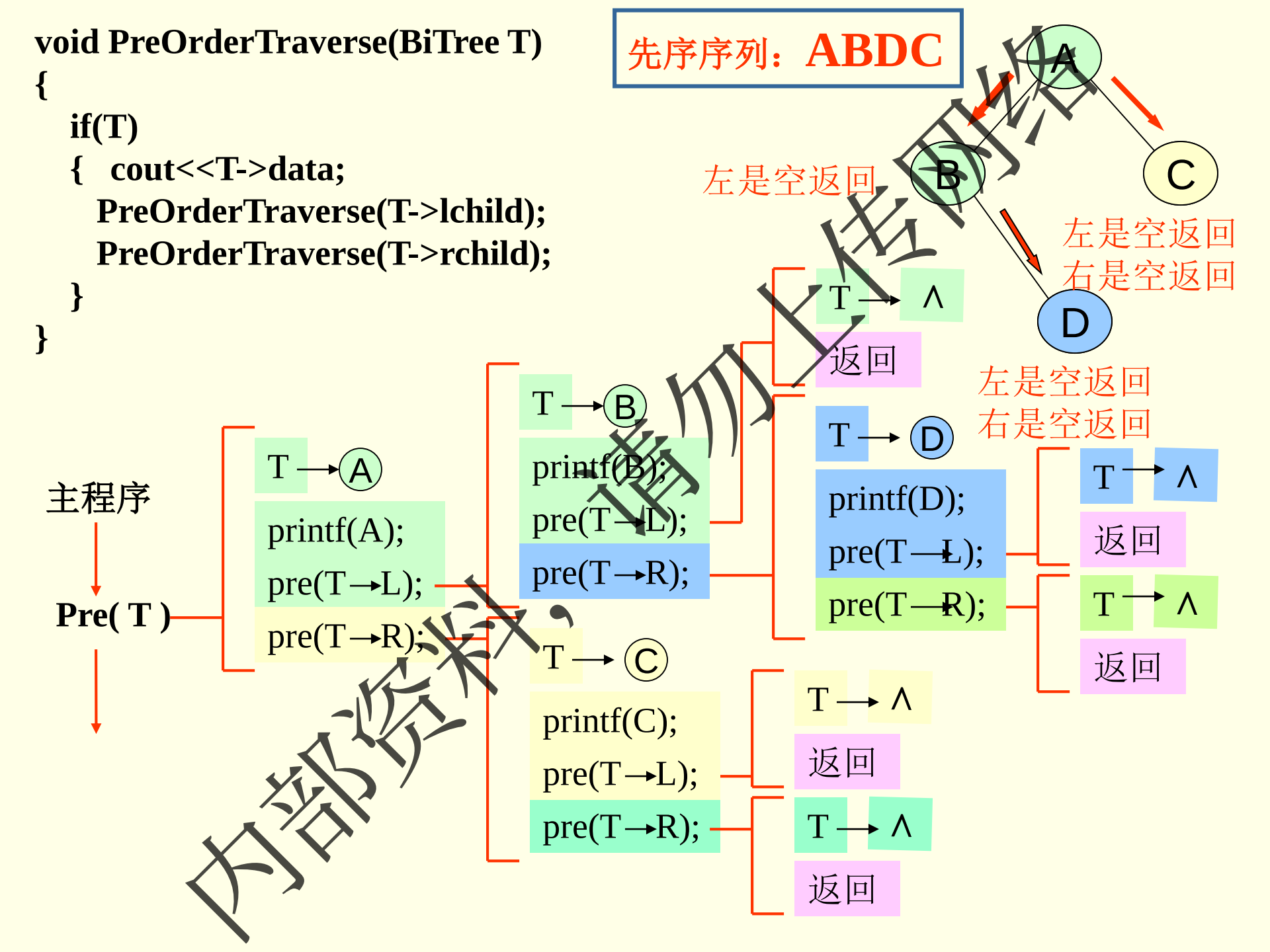
先序遍历算法



```
void PreOrderTraverse(BiTree T)
{//先序遍历二叉树T的递归算法
   if(T)
   { cout<<T->data; //访问根结点
     PreOrderTraverse(T->lchild); //递归遍历左子树
     PreOrderTraverse(T->rchild); //递归遍历右子树
   }
}
```

```
void PreOrderTraverse(BiTree T)
{
   if(T)
   {  cout<<T->data;
      PreOrderTraverse(T->lchild);
      PreOrderTraverse(T->rchild);
   }
}
```

先序序列：ABDC



遍历的算法实现—中序遍历

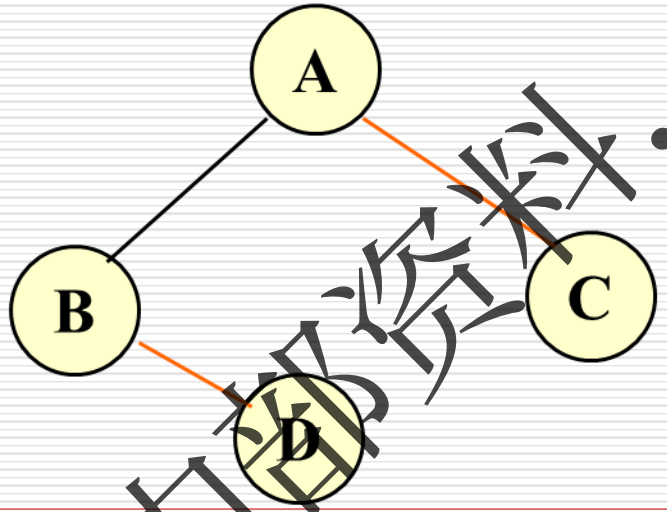
若二叉树为空，则空操作

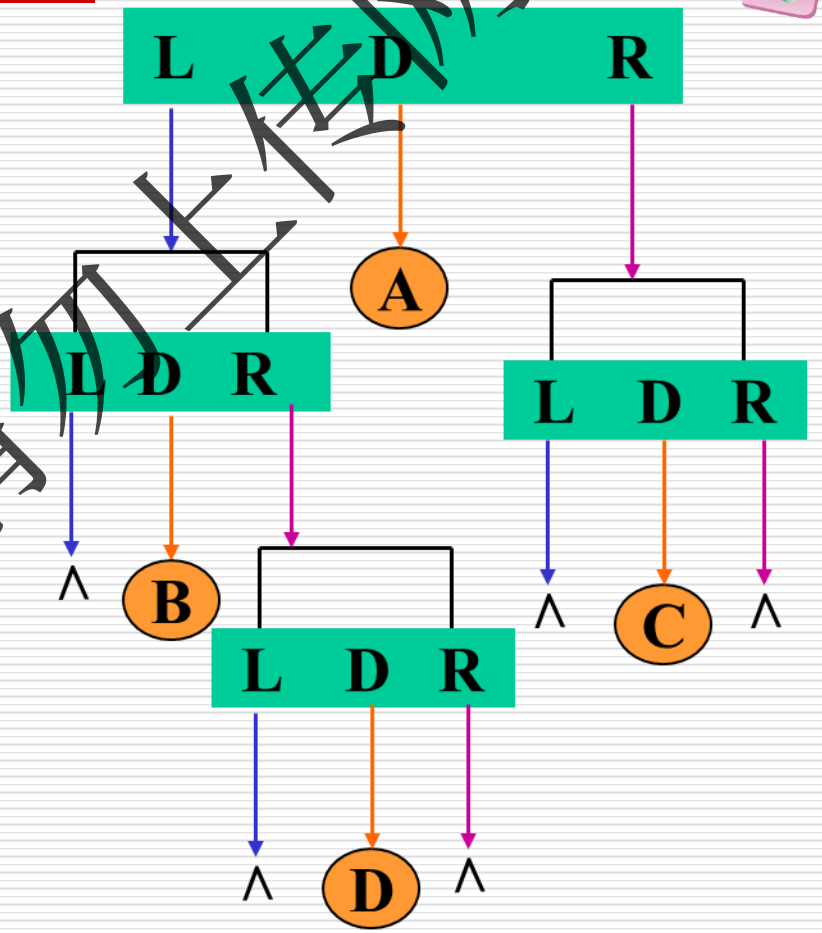
否则：

中序遍历左子树 (L)

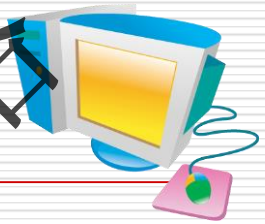
访问根结点 (D)

中序遍历右子树 (R)





中序遍历算法



```
void InOrderTraverse(BiTree T)
{//中序遍历二叉树T的递归算法
   if(T)
   { InOrderTraverse(T->lchild); //递归遍历左子树
     cout<<T->data; //访问根结点
     InPreOrderTraverse(T->rchild); //递归遍历右子
   树
   }
}
```

遍历的算法实现—后序遍历

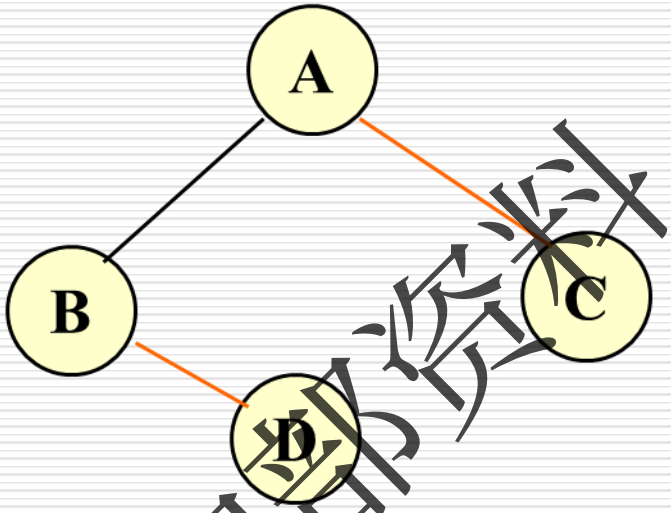
若二叉树为空，则空操作

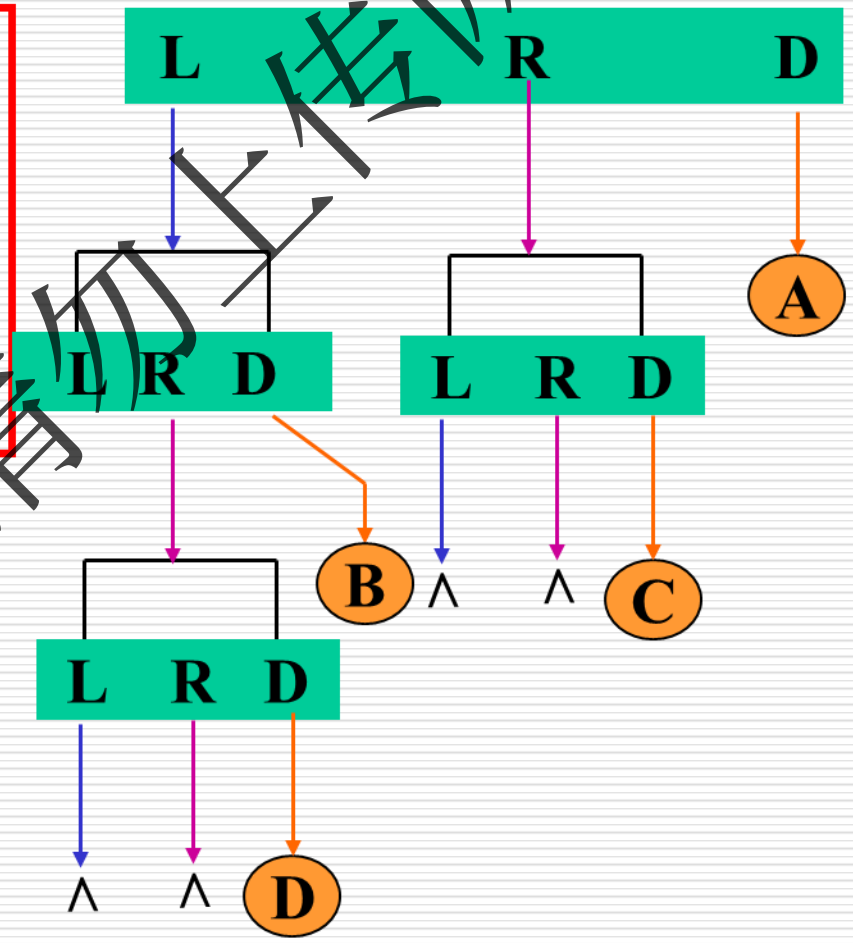
否则

后序遍历左子树（L）

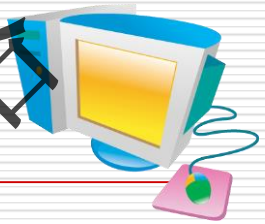
后序遍历右子树（R）

访问根结点（D）



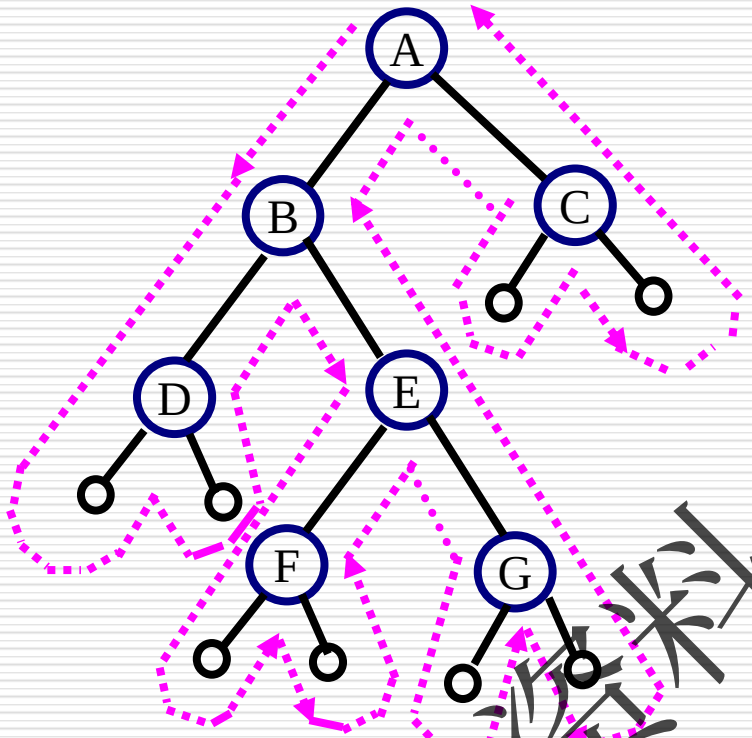


后序遍历算法



```
void PostOrderTraverse(BiTree T)
{//后序遍历二叉树T的递归算法
   if(T)
   { PostOrderTraverse(T->lchild); //递归遍历左子树
     PostOrderTraverse(T->rchild); //递归遍历右子树
     cout<<T->data; //访问根结点
   }
}
```

对遍历的分析

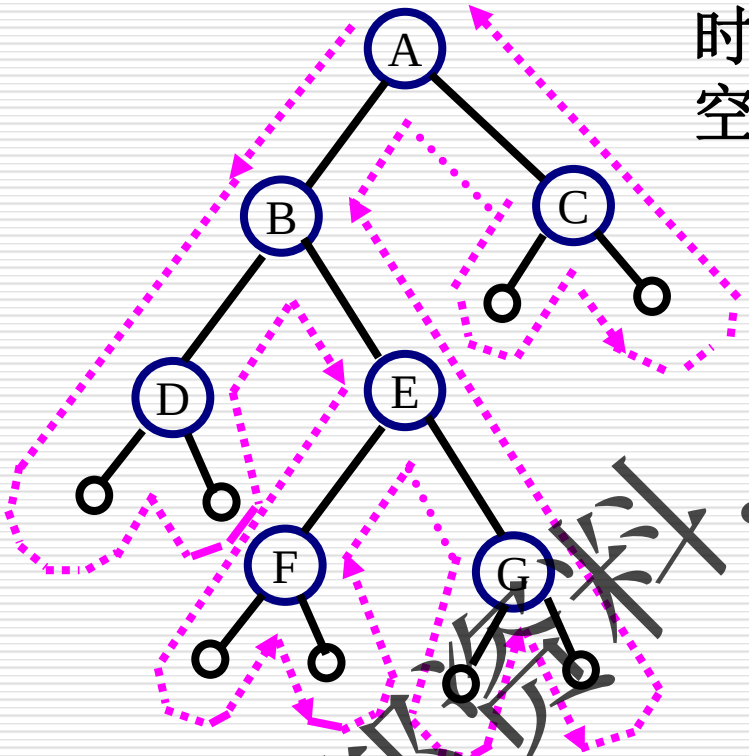


1. 从前面的三种遍历算法可以知道：如果将print语句抹去，从递归的角度看，这三种算法是完全相同的，或者说这三种遍历算法的访问路径是相同的，只是访问结点的时机不同。

从虚线的出发点到终点的路径上，每个结点经过3次。

第1次经过时访问，是先序遍历
第2次经过时访问，是中序遍历
第3次经过时访问，是后序遍历

对遍历的分析



2. 二叉树遍历的时空效率

时间效率：O(n) //每个结点只访问一次

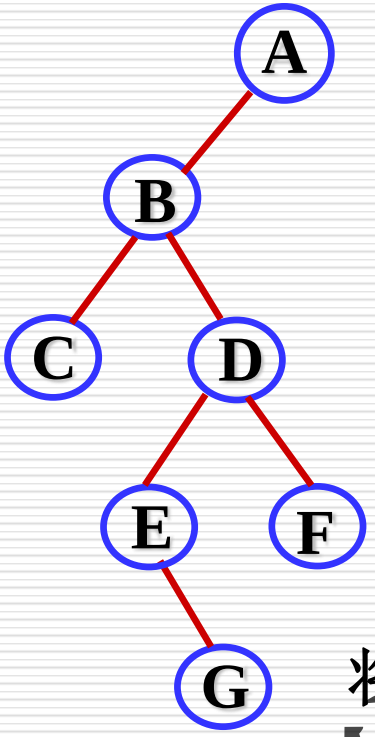
空间效率：O(n) //栈占用的最大辅助空间

中序遍历二叉树的非递归算法（算法6.2）

```
void  InOrderTraverse（BiTree T）
{   InitStack (S);   P=T;
    q=new BiTNode;
     while(p||!StackEmpty(S))  {
    {   if(p)                        //p非空
        {    Push(S,p);              //根指针进栈
             p=p->rchild;            //根指针进栈，遍历左子树
    }
        else                         //p为空
        {    Pop(S,q);               //退栈
             cout<<q->data;          //访问根结点
             p=q->rchild;            //遍历右子树
     }
     }//while
}//InOrderTraverse
```

如何把二叉树存入电脑内？

例：将下面的二叉树以二叉链表形式存入计算机内。



考虑1：输入结点时怎样表示“无孩子”？

考虑2：以何种遍历方式来输入和建树？

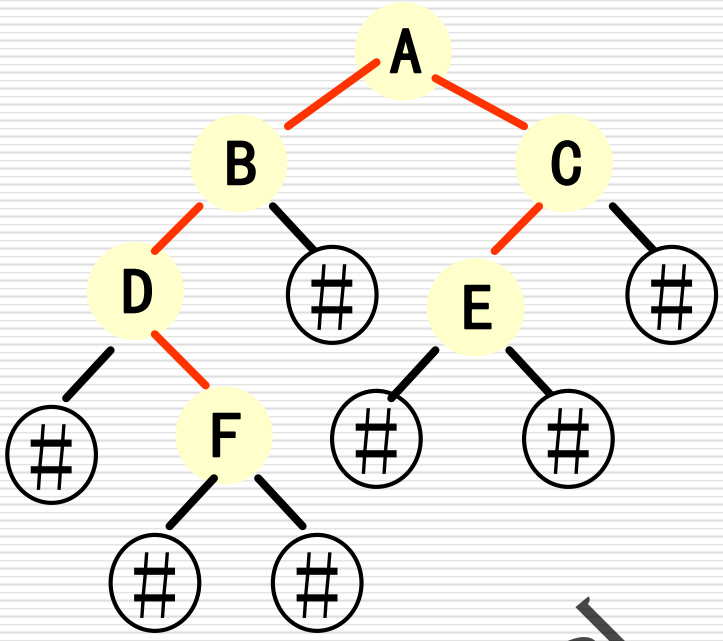
以先序遍历最为合适，让每个结点都能及时被连接到位。

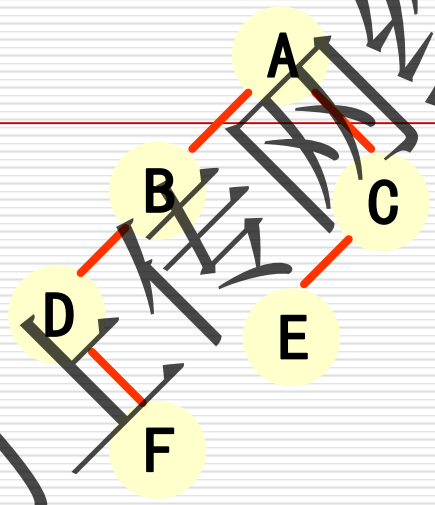
用#字符表示‘无孩子’或指针为空

将二叉树按先序遍历次序输入：

A B C # # D E # G # # F # # # (/n)

建立二叉链表

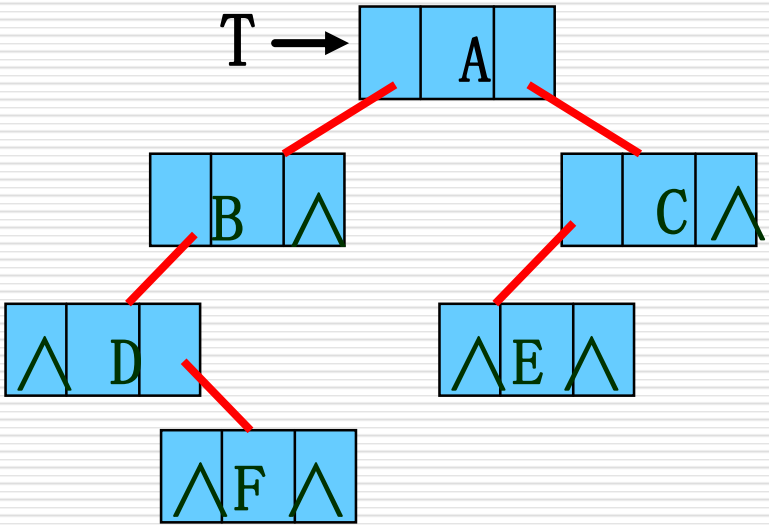




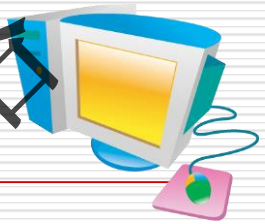
先序序列：A B D F C E

（在空子树处添加#的）先序序列：

A B D # F # # C E # # #

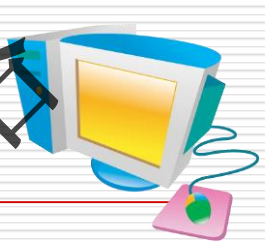


二叉树的建立（算法6.3）



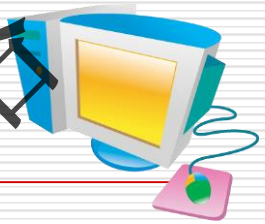
```
void CreateBiTree( BiTree &T ){    //构造二叉树T
    cin>>ch;
    if(ch==’#’)T=NULL;
    else{
            T=new BiTNode;          //生成根结点
            T->data=ch;             //根结点的数据域置为ch
            CreateBiTree(T->lchild);    //构造左子树
            CreateBiTree(T->rchild); //构造右子树
        }
} //CreateBiTree
```

复制二叉树（算法6.4）



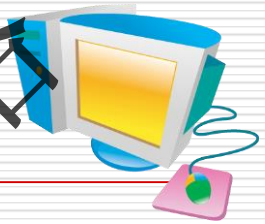
```
void Copy( BiTree T, BiTree &NewT )
{   if ( T == NULL )
        { NewT=NULL;
          return:
        }
     else
        {  NewT=new BiTnode;
           NewT->data=T->data;
          Copy( T->lchild, NewT->lchild );
          Copy(T->rchild , NewT-> rchild );
        }
   }
```

求二叉树深度（算法6.5）



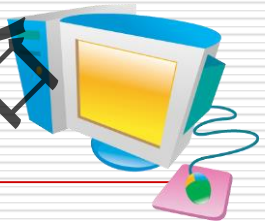
```
int Depth ( BiTree  T )
{  int m, n;
   if ( T == NULL ) return 0;
   else{
         m = Depth ( T->lchild );
         n = Depth ( T->rchild ) );
        return (m > n) ? m+1 : n+1;
        }
}
```

统计结点的数目（算法6.6）



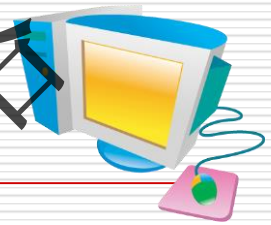
```
int NodeCount(Bitree T)
  //求二叉树中叶子结点的数目
{
  if(!T) return 0; //空树结点数为0
  else return
  NodeCount(T->lchild)+NodeCount(T->rchild)+1;
    //左子树的结点数加上右子树的结点数
}//LeafCount_BiTree
```

统计叶子结点的数目



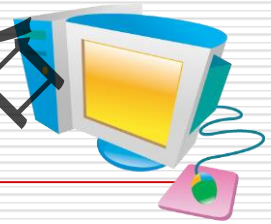
```
int LeafCount(Bitree T)
  //求二叉树中叶子结点的数目
{
  if(!T) return 0; //空树叶子结点数为0
  else if(!T->lchild&&!T->rchild) return 1; //叶子结点为1
  else return
  LeafCount(T->lchild)+LeafCount(T->rchild);
    //左子树的叶子结点个数加上右子树的叶子结点个数
}//LeafCount_BiTree
```

按层次顺序遍历二叉树



```
void LayerOrder(Bitree T)//层序遍历二叉树
{    InitQueue(Q); //建立工作队列
     EnQueue(Q,T);
     while(!QueueEmpty(Q))
    {        DeQueue(Q,p);
             visit(p);
            if(p->lchild) EnQueue(Q,p->lchild);
           if(p->rchild) EnQueue(Q,p->rchild);
     }
}//LayerOrder
```

特别讨论



若已知先序（或后序）遍历结果和中序遍历结果，能否“恢复”出二叉树？

例：已知一棵二叉树的中序序列和后序序列分别是 **BDCEAFHG** 和 **DECBHGFA**，请画出这棵二叉树。

分析：

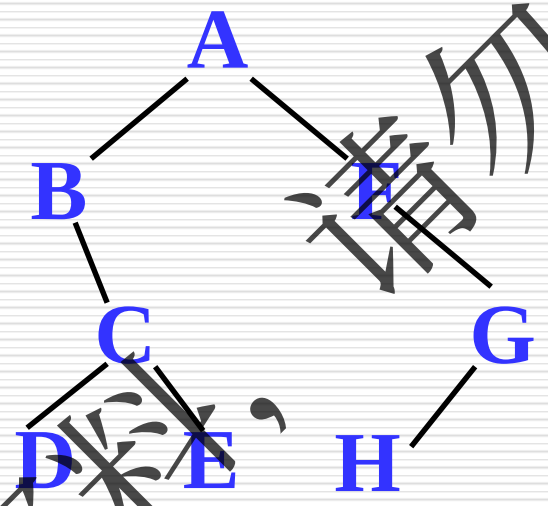
①由后序遍历特征，根结点必在后序序列尾部（即A）；

②由中序遍历特征，根结点必在其中间，而且其左部必全部是左子树的子孙（即BDCE），其右部必全部是右子树的子孙（即FHG）；

③继而，根据后序中的DECB子树可确定B为A的左孩子，根据HGF子串可确定F为A的右孩子；以此类推。

中序遍历序列：B D C E A F H G

后序遍历序列：D E C B H G F A



（D C E）　　（ H GF）

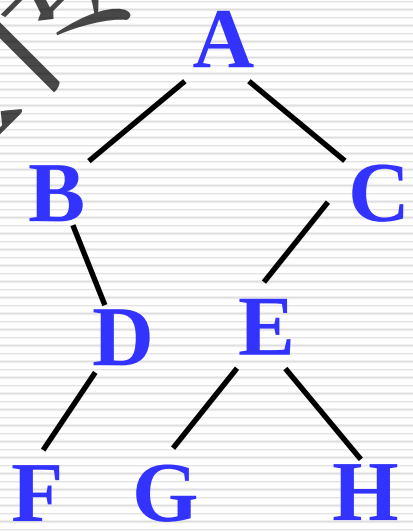
练习

已知：

先序遍历序列：A B D F C E G H

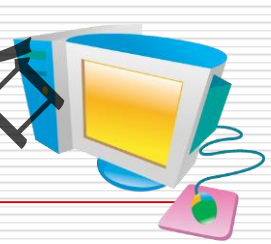
中序遍历序列：B F D A G E H C

求其后序遍历序列。



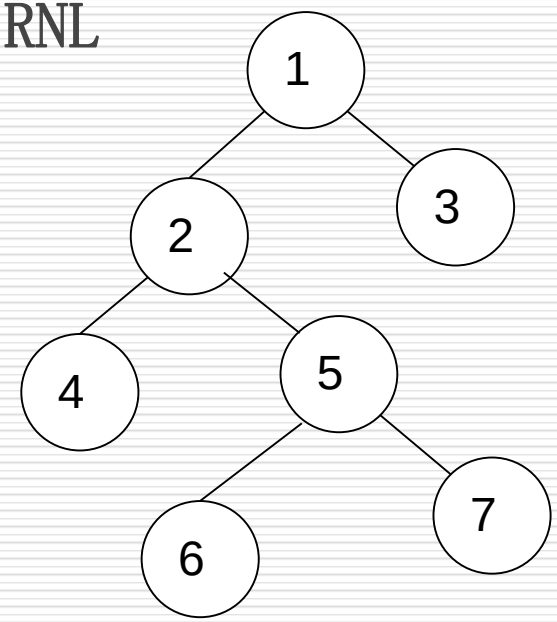
解：后序遍历序列：F D B G H E C A

考研真题

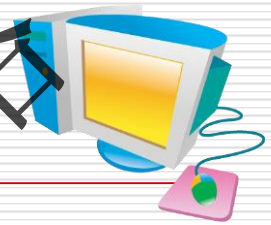


【2009年计算机联考真题】给定二叉树如图所示，设N代表二叉树的根，L代表根结点的左子树，R代表根结点的右子树。若遍历后的结点序列为3，1，7，5，6，2，4，则其遍历方式是（ D ）

A. LRN　　B. NRL　　C. RLN　　D. RNL



考研真题



【2011年计算机联考真题】若一棵二叉树的前序遍历序列和后序遍历序列分别为1，2，3，4和4，3，2，1，则该二叉树的中序遍历序列不会是(C)。

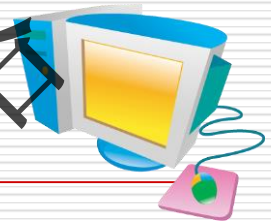
A. 1，2，3，4

B. 2，3，4，1

C. 3，2，4，1

D. 4，3，2，1

考研真题



【2012年计算机联考真题】若一棵二叉树的前序遍历序列为a, e, b, d, c, 后序遍历序列为b, c, d, e, a，则根结点的孩子结点(A)。

A. 只有e

B. 有e, b

C. 有e, c

D. 无法确定

考研真题

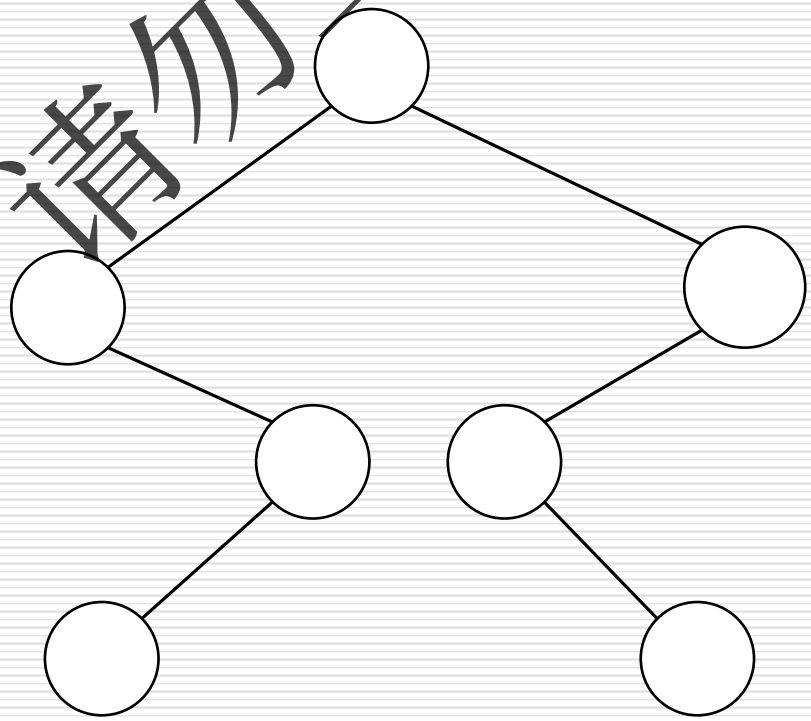
【2017年计算机联考真题】某二叉树的树形如下图所示，其后序序列为e, a, c, b, d, g, f，树中与结点a同层的结点是(B)。

A. c

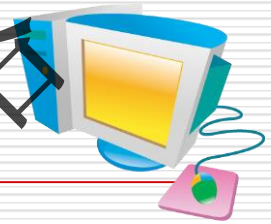
B. d

C. f

D. g



考研真题



【2017年计算机联考真题】要使一棵非空二叉树的先序序列与中序序列相同，其所有非叶结点须满足的条件是（ B ）。

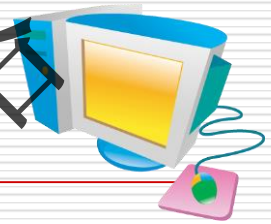
A. 只有左子树

B. 只有右子树

C. 结点的度均为1

D. 结点的度均为2

5.5.2 线索二叉树

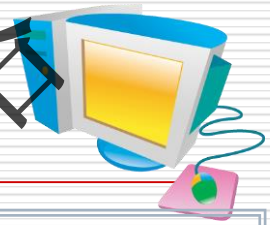


讨论：用二叉链表法存储包含n个结点的二叉树，结点的指针区域中会有多少个空指针？

解：用二叉链表存储包含n个结点的二叉树，结点必有**2n**个链域（见二叉链表数据类型说明）。

除根结点外，二叉树中每一个结点有且仅有一个双亲，意即每个结点地址占用了双亲的一个直接后继，n个结点地址共占用了n-1个双亲的指针域。也就是说，只会有n－1个结点的链域存放指针。

所以，空指针数目＝**2n－(n-1)=n+1**个。



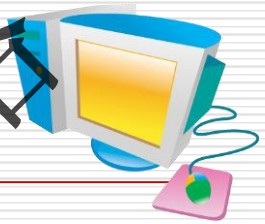
结论：用二叉链表法存储包含n个结点的二叉树，结点的指针区域中会有n+1个空指针。

思考：二叉链表空间效率这么低，能否利用这些空闲区存放有用的信息或线索？

——我们可以用它来存放当前结点的直接前驱和后继等线索，以加快查找速度。

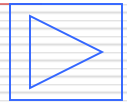
这就是**线索二叉树**（Threaded Binary Tree）

讨论1：

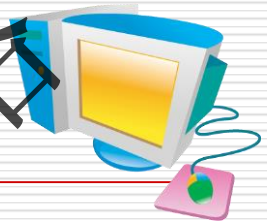


二叉树是1:2的非线性结构，如何定义其直接后继？

对二叉树进行某种遍历之后，将得到一个线性有序的序列。例如对某二叉树的中序遍历结果是B D C E A F H G，意味着已将该树转为线性排列，显然其中结点具有唯一前驱和唯一后继。在此前提下，那n+1个空链域才能装入（也装得下）“线索”。



讨论2：



如何获得这种“直接前驱”或“直接后继”？有何意义？

二叉树中容易找到结点的左右孩子信息，但该结点的直接前驱和直接后继只能在某种遍历过程中动态获得。

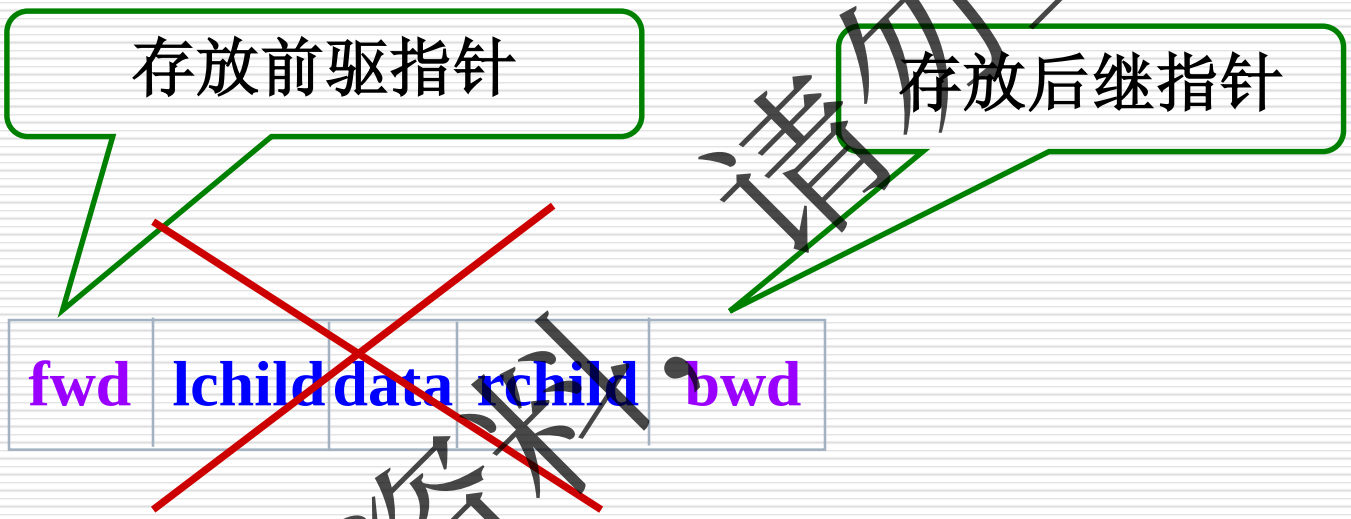
先依遍历规则把每个结点对应的前驱和后继线索预存起来，这叫做“线索化”。

意义：从任一结点出发都能快速找到其前驱和后继，且不必借助堆栈。

如何预存这类信息？

有两种解决方法：

① 每个结点增加两个域：fwd和bwd；



缺点：空间效率太低！

② 原有的左右孩子指针域“复用”，充分利用n+1个空链域。

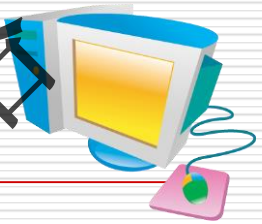
如何判断是孩子指针还是线索指针？

规 定：

1）若结点有左子树，则lchild指向其左孩子；
否则， lchild指向其直接前驱(即线索)；

2）若结点有右子树，则rchild指向其右孩子；
否则，rchild指向其直接后继(即线索) 。

增加两个标志域：



lchild	LTag	data	RTag	rchild

各1bit

左(右)孩子

前驱(后继)

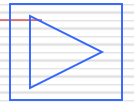
约定：

当Tag域为0时,表示正常情况;

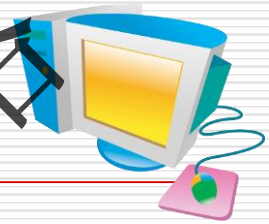
当Tag域为1时,表示线索情况.

问：增加了前驱和后继等线索有什么好处？

——能方便找出当前结点的前驱和后继，不用堆栈（递归）也能遍历整个树。



有关线索二叉树的几个术语：



线索链表： 用含Tag的结点样式所构成的二叉链表

线　索： 指向结点前驱和后继的指针

线索二叉树： 加上线索的二叉树

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

线索化过程就是在遍历过程中修改空指针的过程：

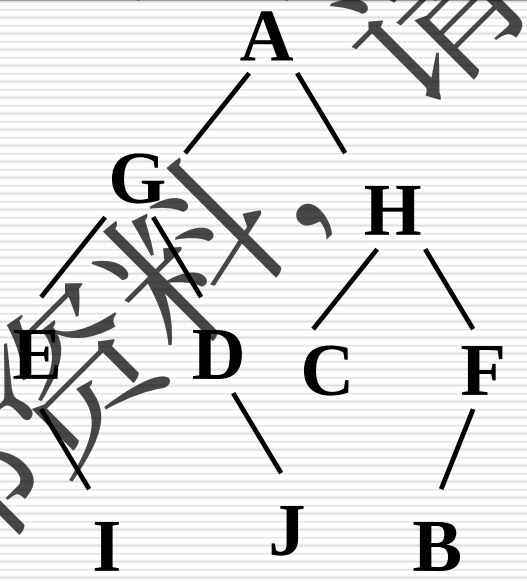
将空的lchild改为结点的直接前驱；

将空的rchild改为结点的直接后继。

非空指针呢？仍然指向孩子结点（称为“正常情况”）

例：带了两个标志的某先序遍历结果如下表所示，请画出对应的二叉树。

Ltag	0	0	1	1	1	1	0	1	0	1
data	A	G	E	I	D	J	H	C	F	B
Rtag	0	0	0	1	0	1	0	1	1	1

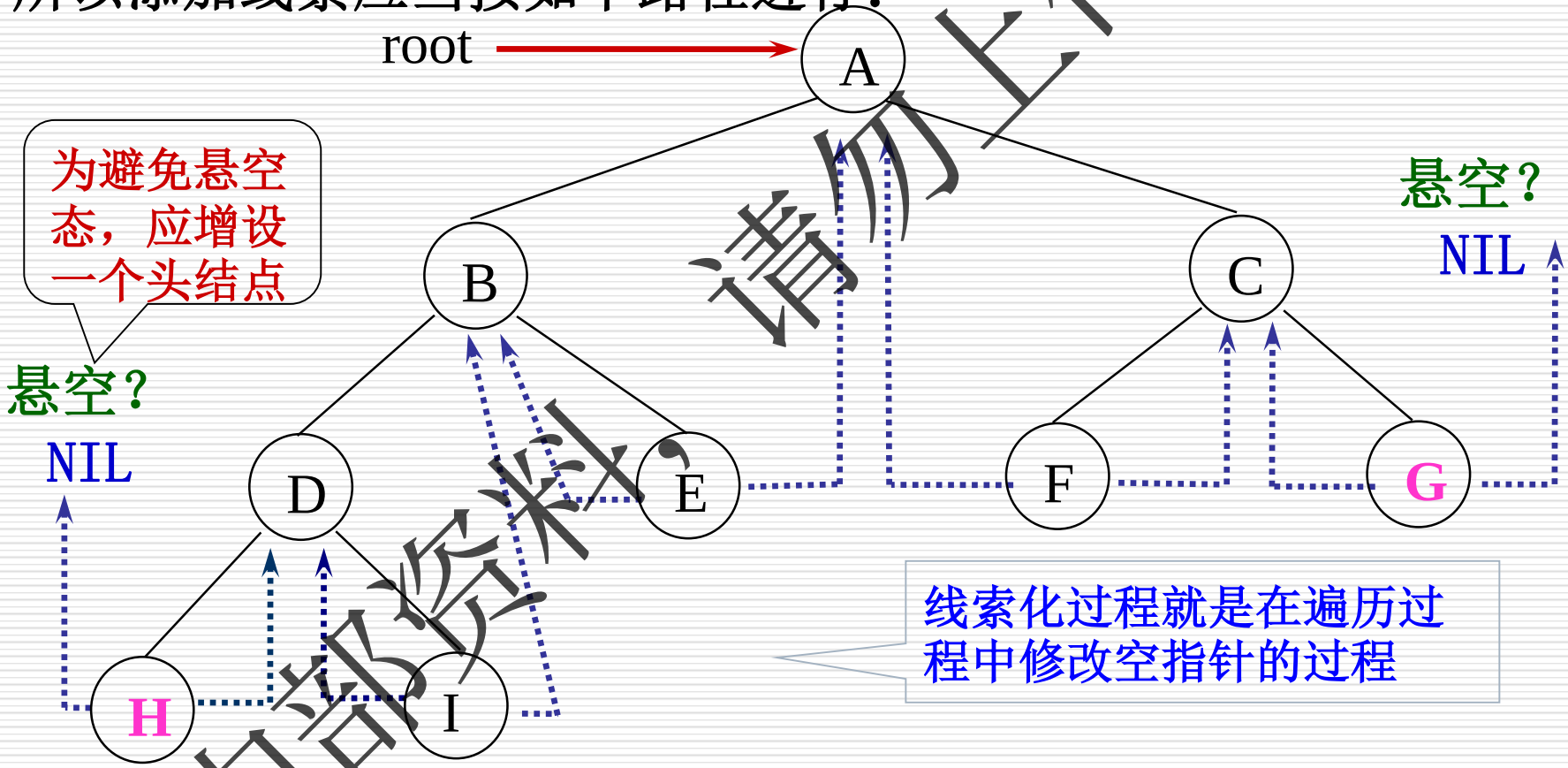


Tag=1表示线索：
Ltag=1表示前驱
Rtag=1表示后继

例：画出以下二叉树对应的中序线索二叉树。

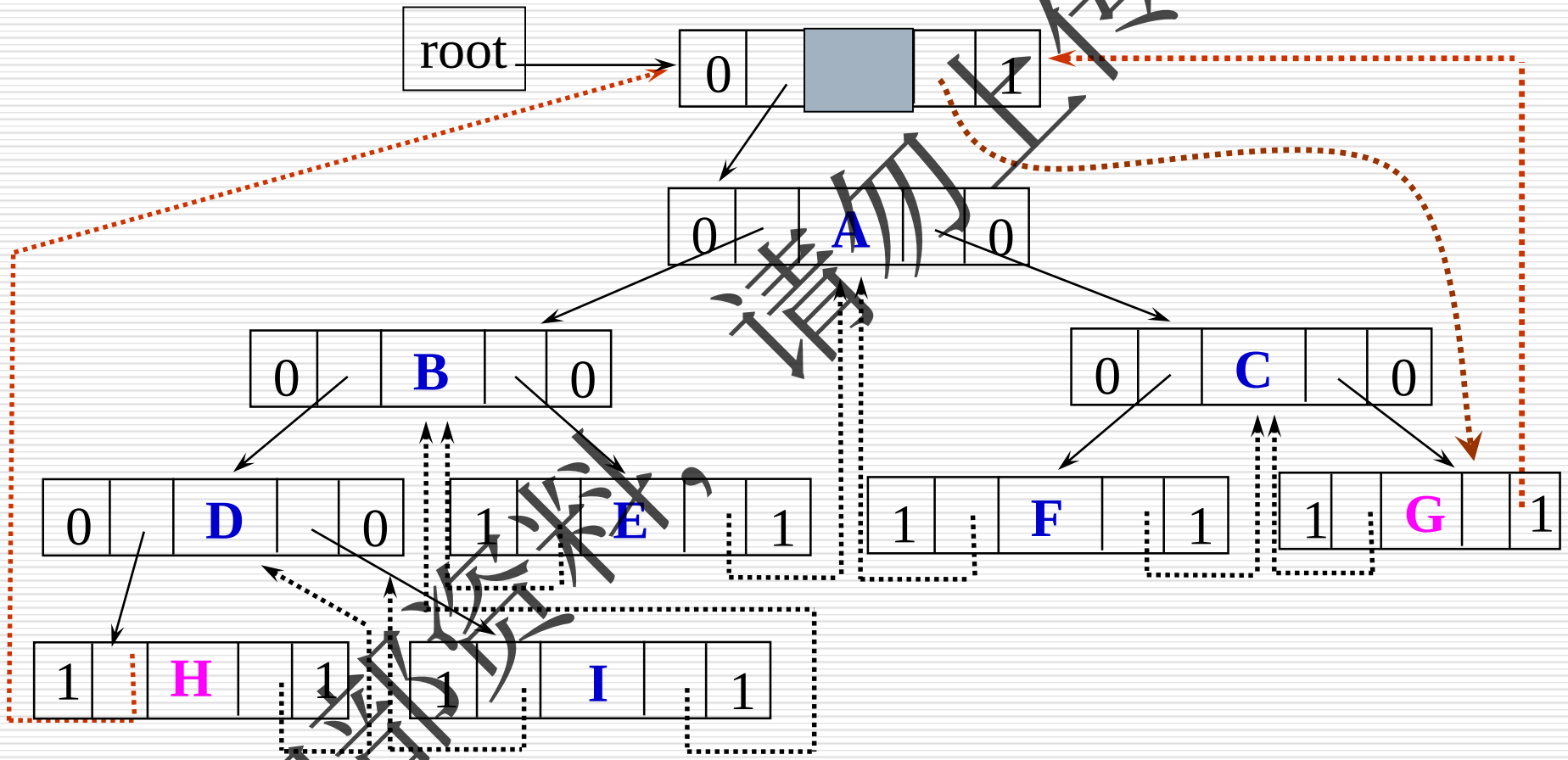
解：对该二叉树中序遍历的结果为：H, D, I, B, E, A, F, C, G

所以添加线索应当按如下路径进行：

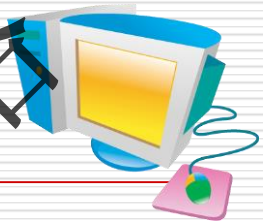


对应的中序线索二叉树存储结构如图所示：

注：中序遍历结果为： H, D, I, B, E, A, F, C, G



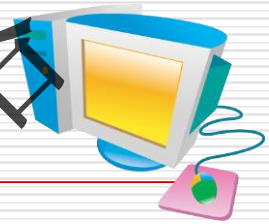
线索链表的类型描述：



```
typedef enum { Link, Thread } PointerTag;
  // Link==0:指针，Thread==1:线索

typedef struct BiThrNode {
   TElemType       data;
   struct BiThrNode  *lchild, *rchild;  // 左右指针
   PointerTag        LTag, RTag;    // 左右标志
} BiThrNode, *BiThrTree;
```

线索二叉树的生成算法



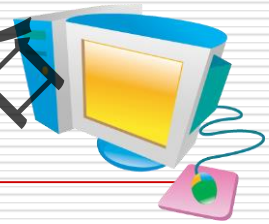
目的：在遍历二叉树的过程中修改空指针，添加前驱或后继的线索，使之成为线索二叉树。

为了记下遍历过程中访问结点的先后次序，需要设置两个指针：

p指针→当前结点之指针；

pre指针→当前结点的前趋结点指针。

设计技巧：



依某种顺序遍历二叉树，对每个结点p，判断其左指针是否为空，以及其前驱结点的右指针是否为空。

每次只修改前驱结点的右指针（后继）和本结点的左指针（前驱），参见算法6.6。

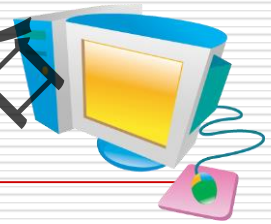
若p->lchild＝NULL, 则{p->Ltag=1;p->lchild＝pre;}

//p的前驱线索应存在p结点的左链域

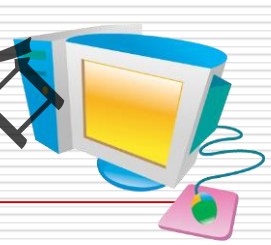
若pre->rchild＝NULL, 则{pre->Rtag＝1;pre->rchild=p;}

//pre的后继线索应存在pre结点的右链域

中序遍历建立中序线索链表

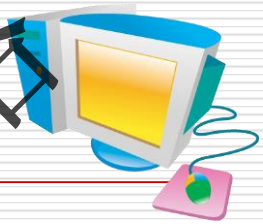


```
Status InOrderThreading(BiThrTree &Thrt, BiThrTree T)
{ // 中序遍历二叉树T，并将其中序线索化，
                          Thrt指向头结点
  if (!(Thrt = (BiThrTree)malloc(sizeof( BiThrNode))))
        exit (OVERFLOW);
  Thrt->LTag = Link; Thrt->RTag =Thread;
  Thrt->rchild = Thrt;      // 添加头结点
        ……
```



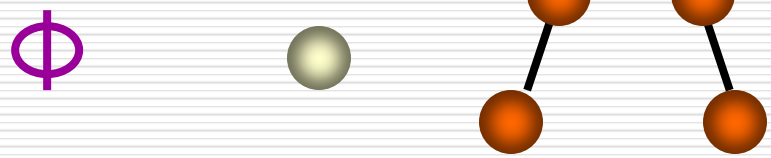
```
if (!T)  Thrt->lchild = Thrt;
else {
   Thrt->lchild = T;   pre = Thrt;
   InThreading(T);
   pre->rchild = Thrt;   // 处理最后一个结点
   pre->RTag = Thread;
   Thrt->rchild = pre;   }
return OK;
} // InOrderThreading
```

对以p为根的非空二叉树进行中序线索化

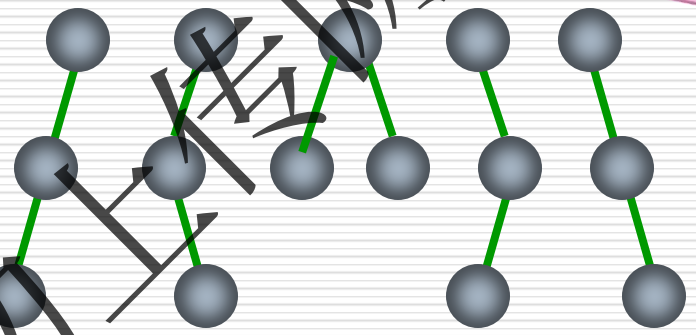


```
void InThreading(BiThrTree p) {
  if (p) {
    InThreading(p->lchild);      // 左子树线索化
    if (!p->lchild)      // 建前驱线索
      { p->LTag = Thread;    p->lchild = pre; }
    if (!pre->rchild)   // 建后继线索
      { pre->RTag = Thread;   pre->rchild = p; }
    pre = p;              // 保持 pre 指向 p 的前驱
    InThreading(p->rchild);      // 右子树线索化
  } // if
} // InThreading
```

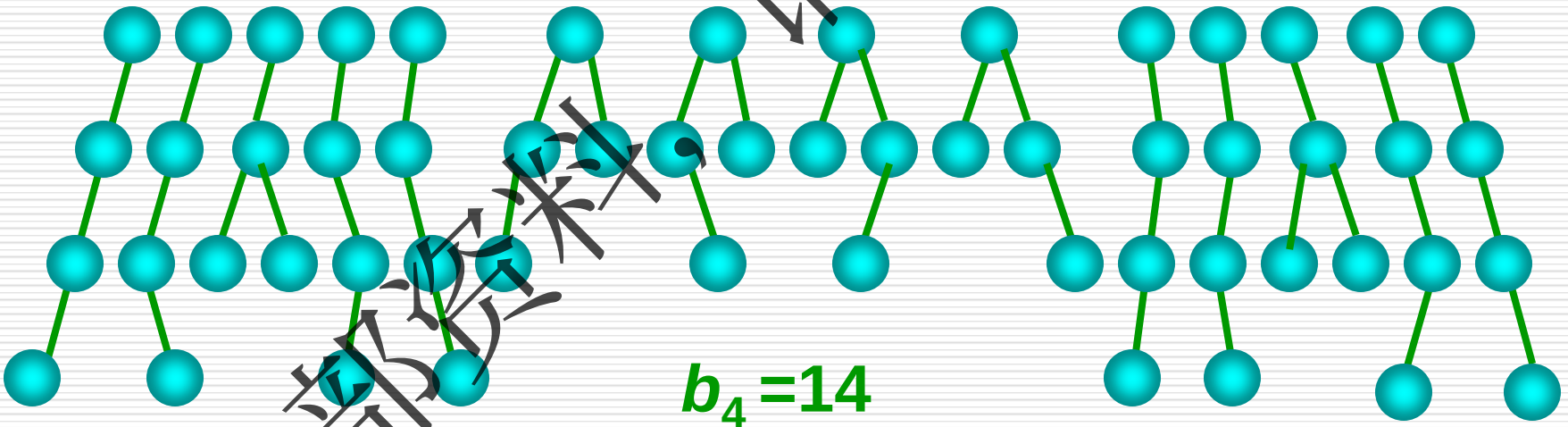
0个, 1个, 2个, 3个,4个结点的不同二叉树



$b_0=1$ $b_1=1$ $b_2=2$



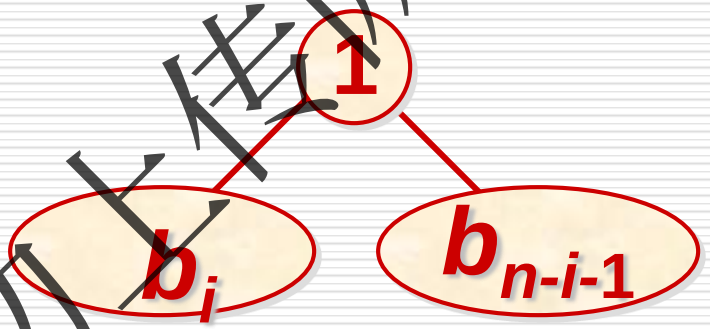
$b_3=5$



$b_4=14$

具有 n 个结点的不同二叉树的棵数

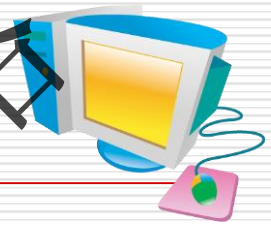
$$b_n = \sum_{i=0}^{n-1} b_i \cdot b_{n-i-1}$$



最终结果：

$$b_n = \frac{1}{n+1} C_{2n}^{n} = \frac{(2n)!}{(n+1)! \cdot n!}$$

2009年计算机考研真题



5. 已知一棵完全二叉树的第6层（设根为第1层）有8个叶结点，则完全二叉树结点个数最多是（C）

A. 39　　B. 52　　C. 111　　D. 119

6. 将森林转换为对应的二叉树，若在二叉树中，结点u是结点v的父结点的父结点，则在原来的森林中，u和v可能具有的关系是（B）

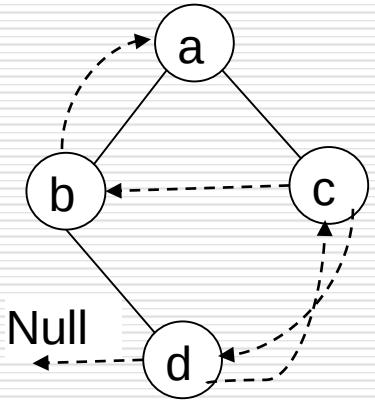
I. 父子关系　　II. 兄弟关系　　III. U的父结点与v的父结点是兄弟关系

A. 只有II　　B. I和II　　C. I和III　　D. I、II和III

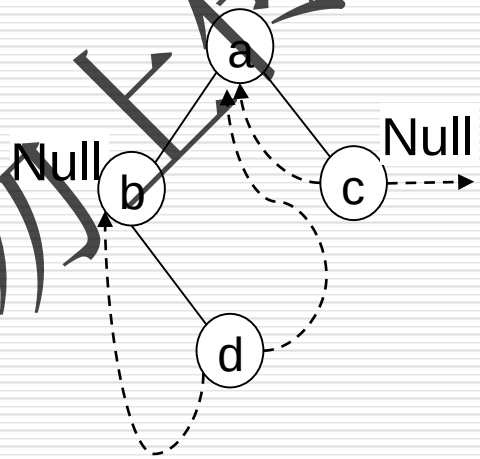
2010年计算机考研真题

3. 下列线索二叉树中（用虚线表示线索），符合后序线索树定义的是（ D）

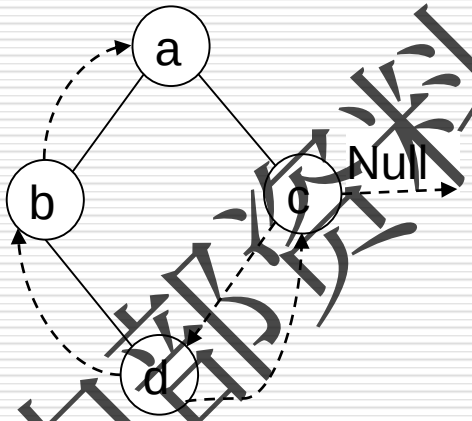
A.



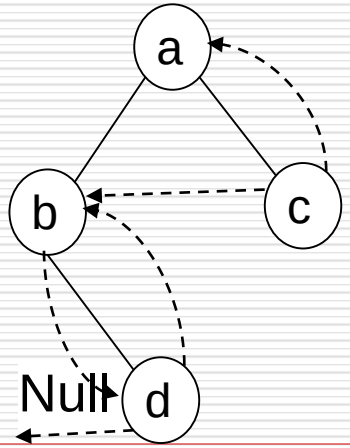
B.



C.

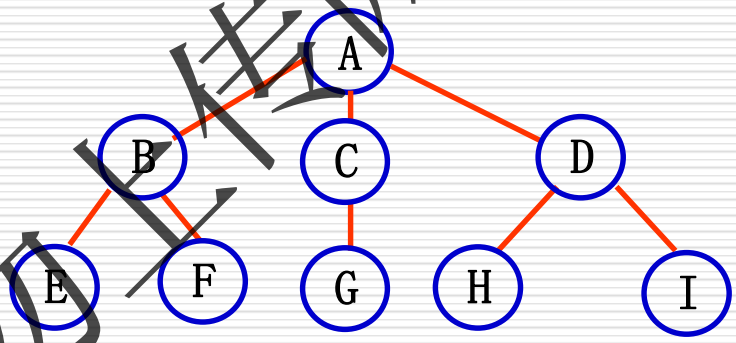


D.



5.6 树和森林

5.6.1 树的存储结构



讨论1： 树的顺序存储方案应该怎样制定？

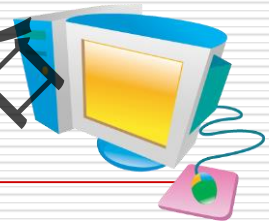
可规定为：

不能唯一复原就没有实用价值！

从上至下、从左至右将树的结点依次存入内存。

缺陷： 复原困难！

1 双亲表示法



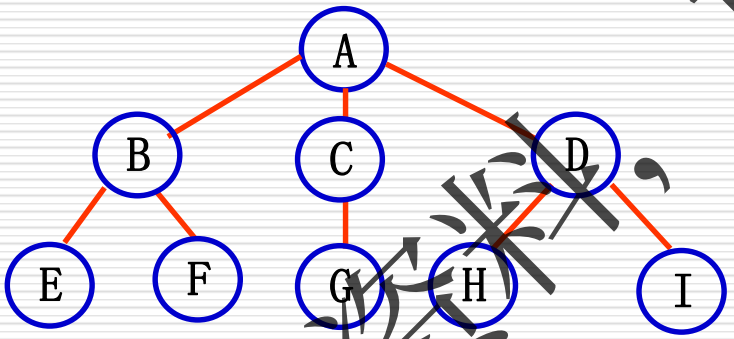
用一组连续的内存单元存储树的结点，每个结点包含两个域：一个数据域，一个“指针域”，用于指示其双亲结点在数组中的位置。

```
#define MAX_TREE_SIZE  100
typedef struct PTNode{
    TelemType  data;
    int   parent;
}PTNode;
```

PTNode

data	parent

```
typedef struct{
    PTNode nodes[MAX_TREE_SIZE];
    int   n;   //结点数
}Ptree;
```



T.nodes

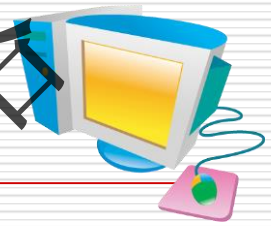
	data	parent
0	A	-1
1	B	0
2	C	0
3	D	0
4	E	1
5	F	1
6	G	2
7	H	3
8	I	3
⋮		

双亲结点在数组中的位置
-1表示无双亲

T.n | 9

结点数

讨论2：树的链式存储方案应该怎样制定？



孩子表示法：通过保存每个结点的孩子结点的位置，表示树中结点之间的结构关系。

方法I ：多重链表

可用多重链表：一个前驱指针，n个后继指针。

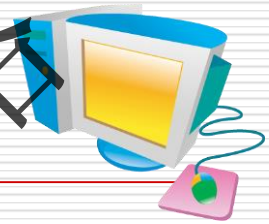
困惑：构造树的结点时应当设多少个链域?

data	link 1	link 2	…		…	link n

细节问题： 树中结点的结构类型样式该如何设计？
即应该设计成“等长”还是“不等长”？

缺点： 等长结构太浪费（每个结点的度不一定相同）；
不等长结构太复杂（要定义好多种结构类型）。

方式II：孩子链表



对树的每个结点用线性链表存储它的孩子结点。

```
typedef struct CTNode{
    int    child;
    struct CTNode * next;
}* ChildPtr;
typedef struct{
    TElemType data;
    ChildPtr firstchild;
}CTBox;
typedef struct{
   CTBox nodes[MAX_TREE_SIZE];
   int   n, r;   //结点数和根的位置；
}CTree;
```

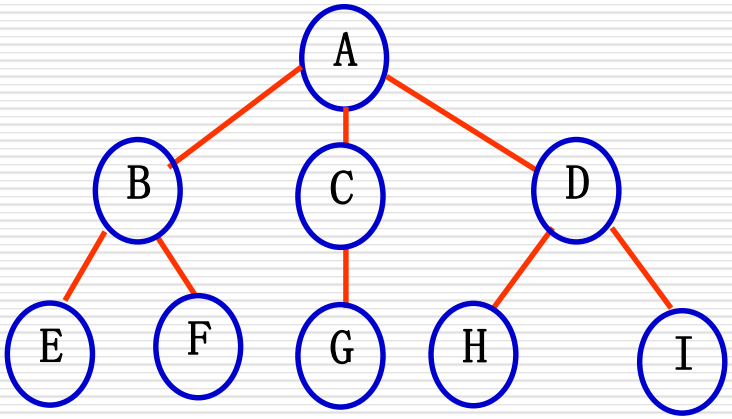
孩子结点

child	next

表头结点

data	firstchild

树的孩子链表图示



T. nodes

data firstchild

	data	firstchild
0	A	→ 1 → 2 → 3 ^
1	B	→ 4 → 5 ^
2	C	→ 6 ^
3	D	→ 7 → 8 ^
4	E	^
5	F	^
6	G	^
7	H	^
8	I	^
…		
99		

T. n: 9

T. r: 0

结点的孩子结点链表

结点数和根的位置

3 孩子兄弟表示法

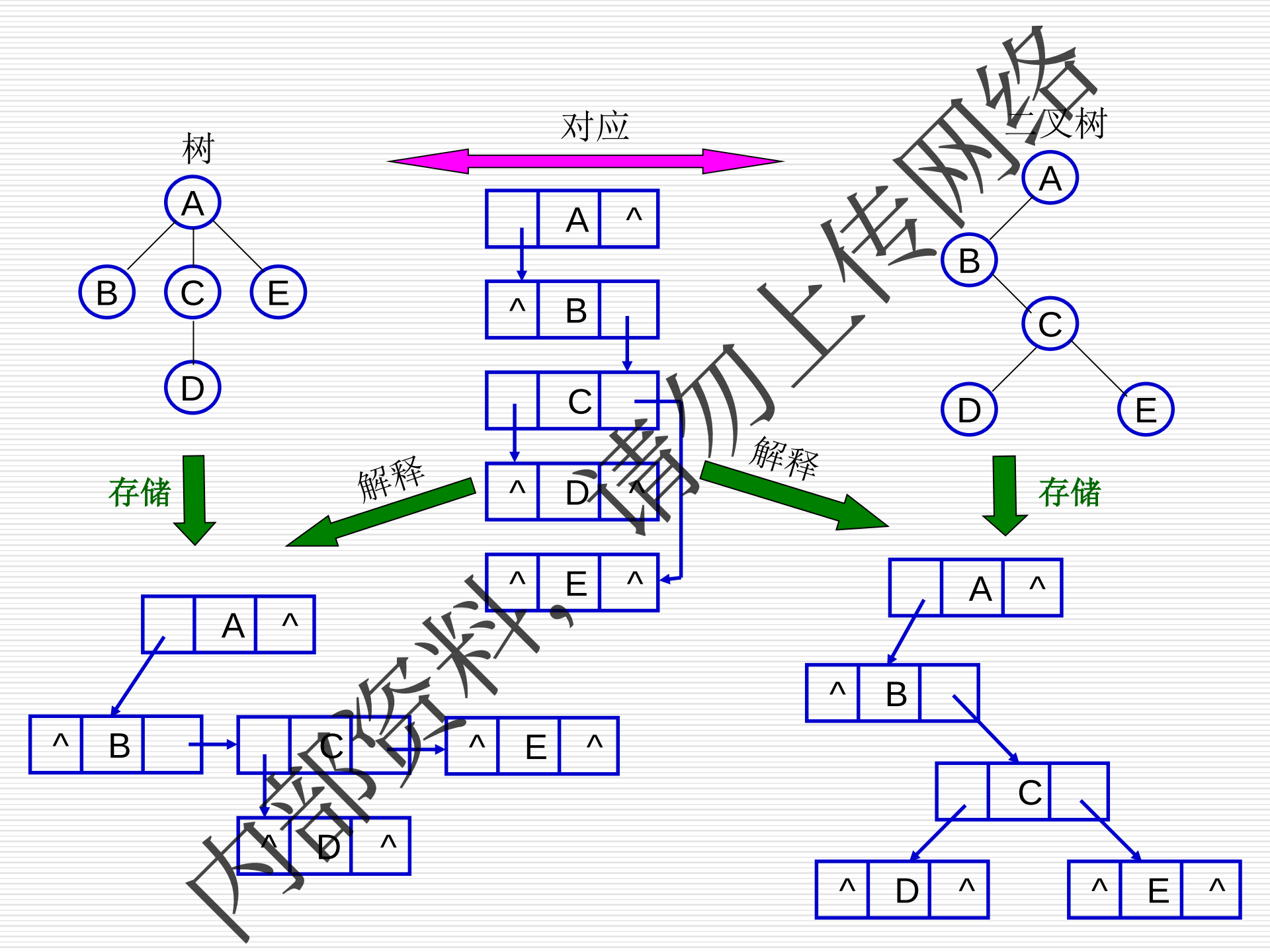
用二叉链表作为树的存储结构

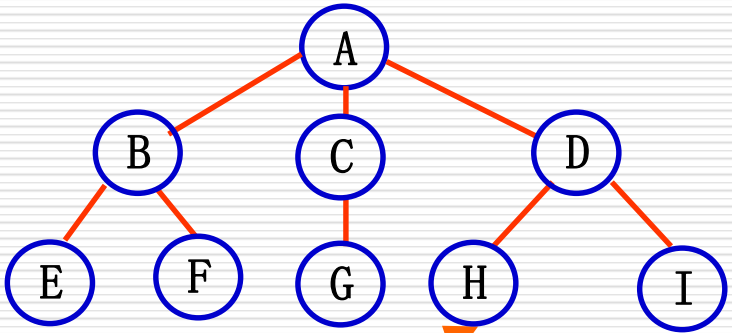
孩子兄弟表示法类型定义

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

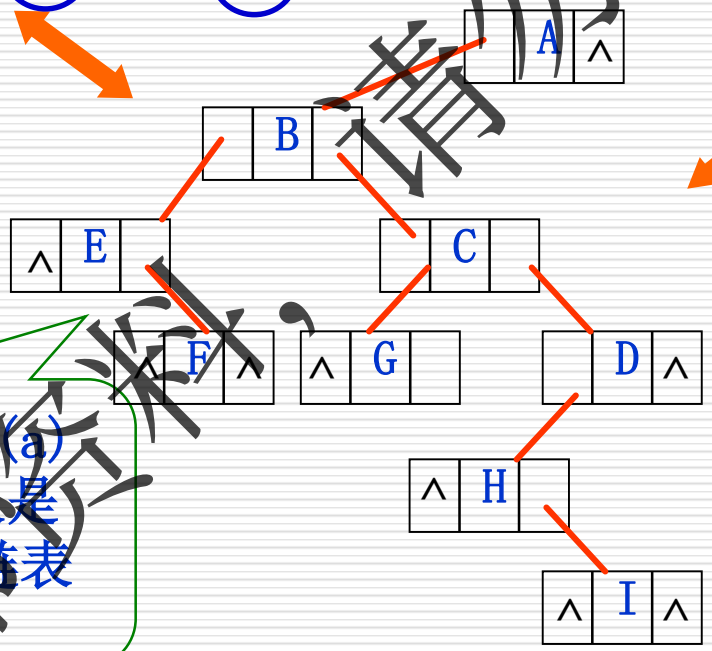
指向左孩子

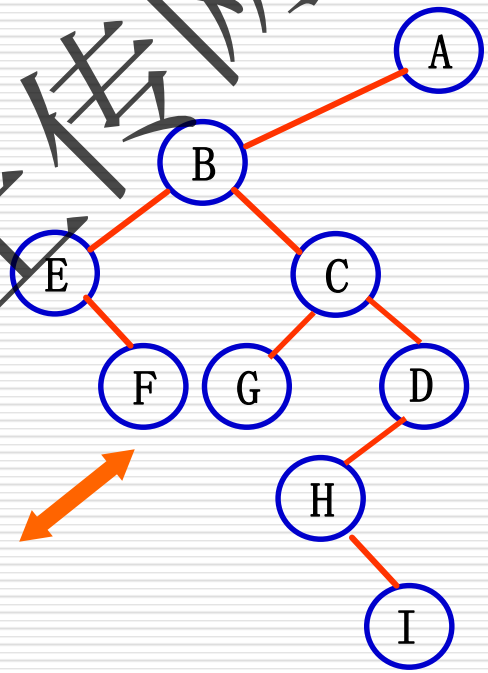
指向右兄弟





(a)

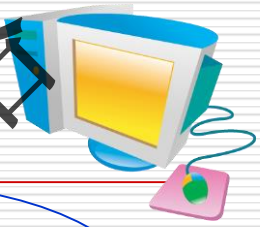




(b)

此二叉链表既是树(a)的孩子兄弟表示又是二叉树(b)的二叉链表

5.6.2 树和森林与二叉树的转换



讨论1：树如何转为二叉树？

孩子—兄弟表示法

转换步骤：

step1：将树中同一结点的兄弟相连；

加线

step2：保留结点的最左孩子连线，删除其它孩子连线；

抹线

step3：将同一孩子的连线绕左孩子顺时针旋转45度角。

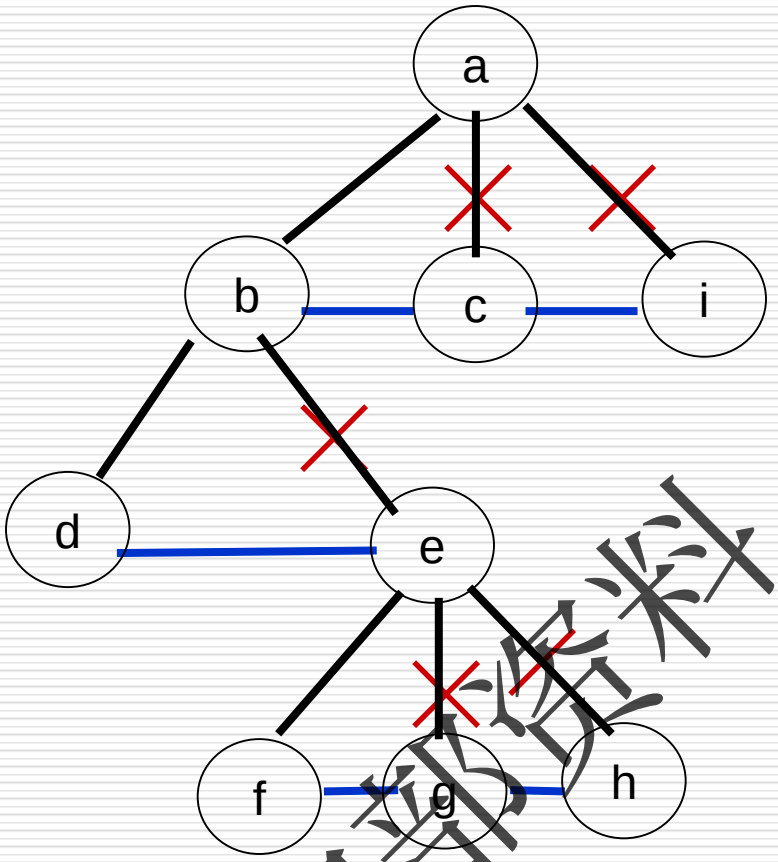
旋转

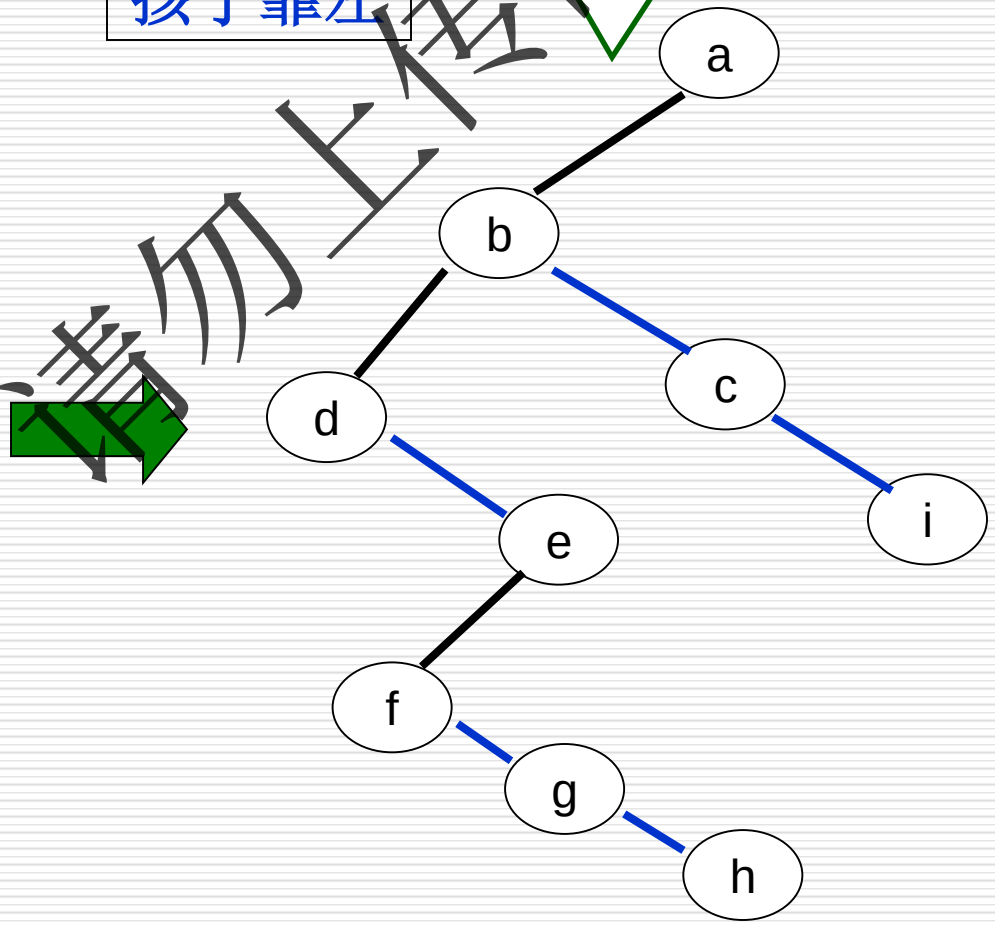
树转二叉树举例：

方法：加线—抹线—旋转

兄弟相连　　长兄为父　　孩子靠左

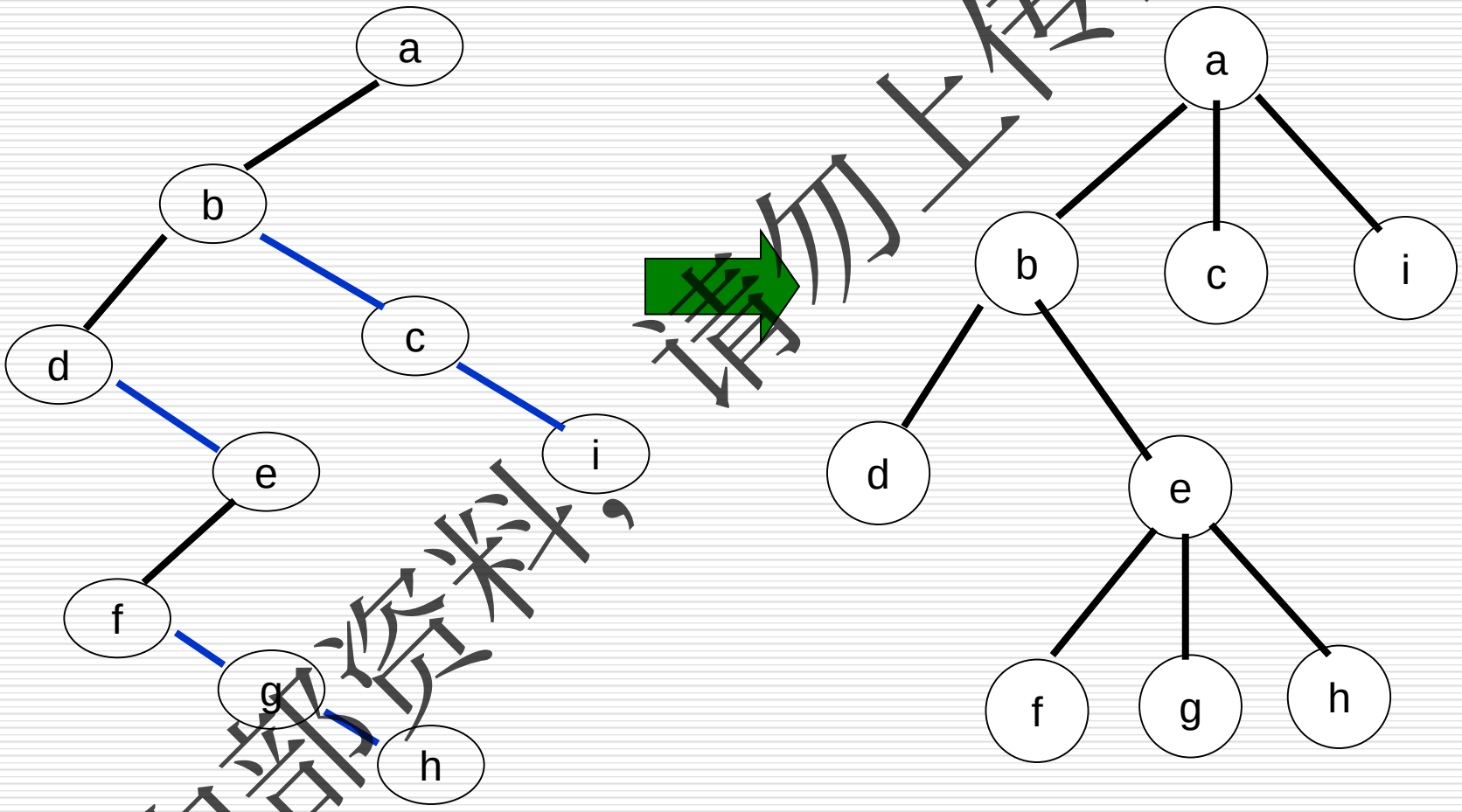
特点是？
根结点没有右孩子！



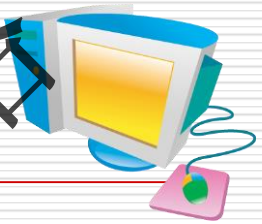


讨论2：二叉树怎样还原为树？

要点：逆操作，把所有右孩子变为兄弟！



讨论3：森林如何转为二叉树？



即$F=\{T_1, T_2, \cdots, T_m\}$ ⟺ $B=\{root, LB, RB\}$

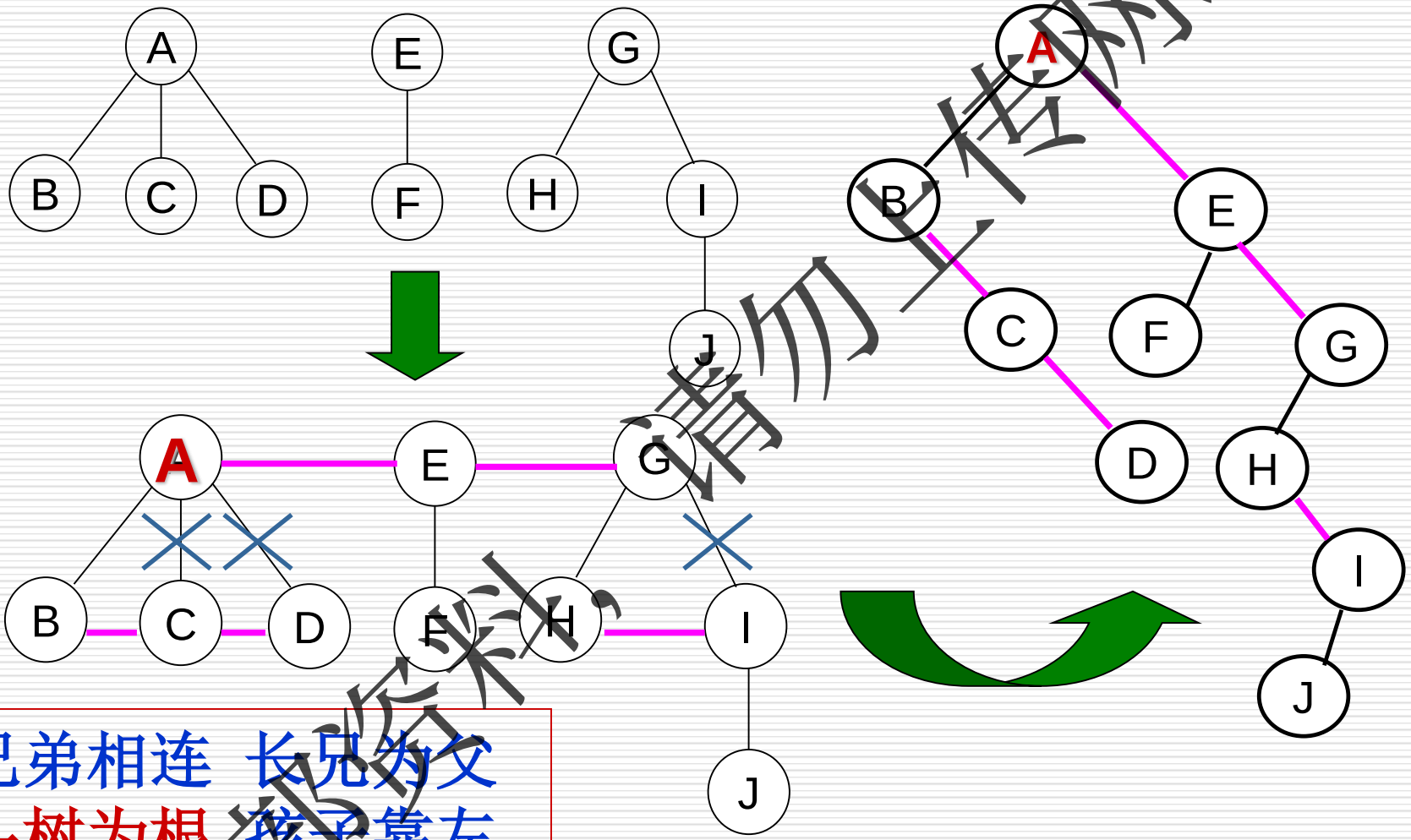
法一：① 树先各自转为二叉树；

② 依次连到前一个二叉树的右子树上。

法二：各树直接变兄弟，再转为二叉树

两种方法得到的二叉树是完全相同的、惟一的。

森林转二叉树举例：（用法二，森林直接变兄弟，再转为二叉树）

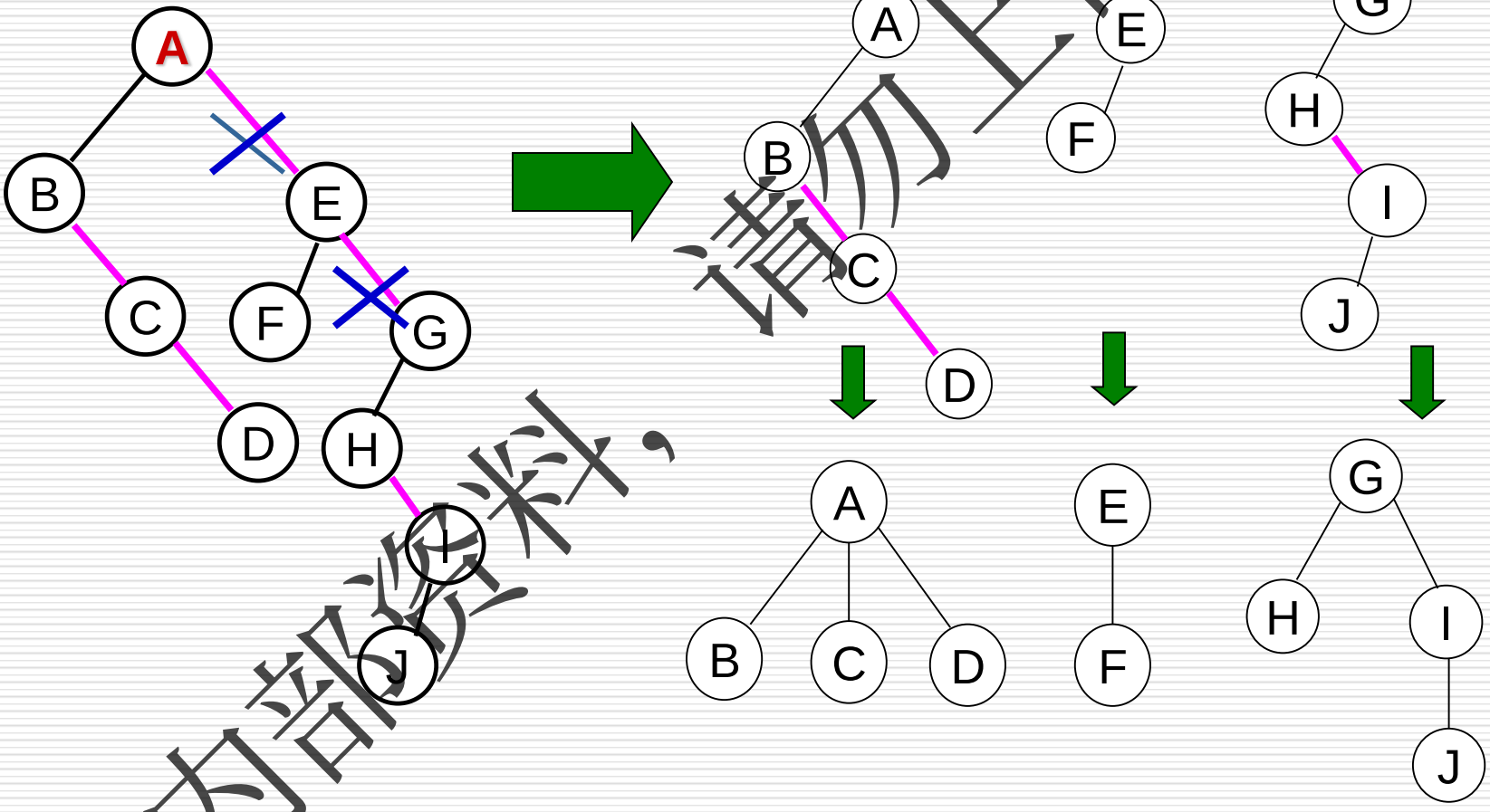


兄弟相连 长兄为父
头树为根 孩子靠左

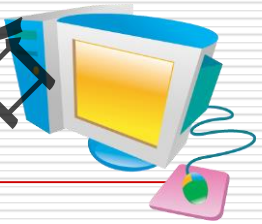
讨论4：二叉树如何还原为森林？

即B={root，LB，RB} ⟺ F={T_1，T_2，…，T_m}

要点：先变为子二叉树森林，每棵二叉树再分别变为树



5.6.3 树和森林的遍历



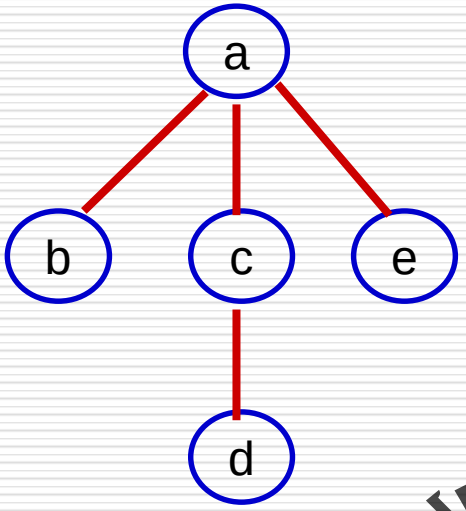
树的遍历
- 先根遍历
- 后根遍历
- 层序遍历

树没有中序遍历
（因为子树不分左右）

- 先根遍历
  - 访问根结点；
  - 依次先根遍历根结点的每棵子树。

- 后根遍历
  - 依次后根遍历根结点的每棵子树；
  - 访问根结点。

例：

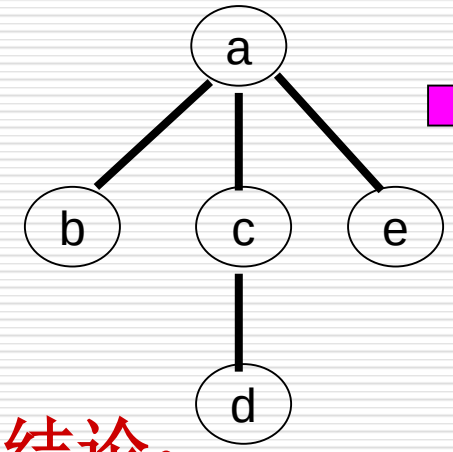


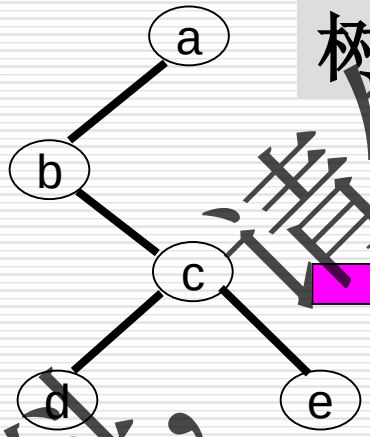
先根序列： a b c d e

后根序列： b d c e a

讨论：

树若采用“先转换，后遍历”方式，结果如何？





树的先根序列：a b c d e

树的后根序列：b d c e a

先序遍历：a b c d e

中序遍历：b d c e a

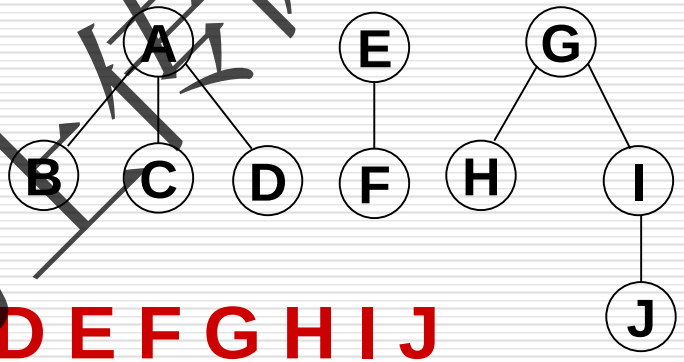
后序遍历：d e c b a

结论：

1. 树的先根遍历与对应二叉树的先序遍历相同；
2. 树的后根遍历相当于对应二叉树的中序遍历；
3. 树没有中序遍历，因为子树无左右之分。

森林的遍历

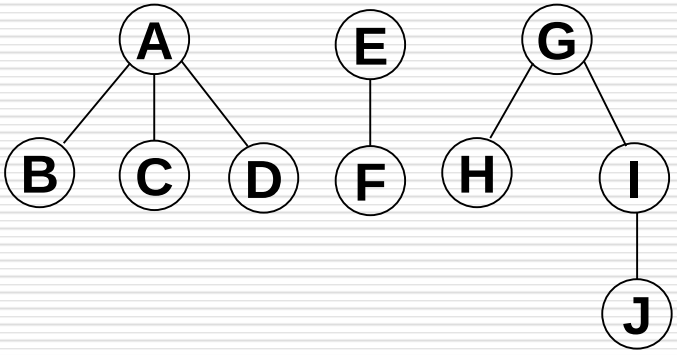
- 先根遍历
- 中根遍历



先根序列：A B C D E F G H I J

先根遍历

- 若森林为空，返回；
- 访问森林中第一棵树的根结点；
- 先根遍历第一棵树的根结点的子树森林；
- 先根遍历除去第一棵树之后剩余的树构成的森林。



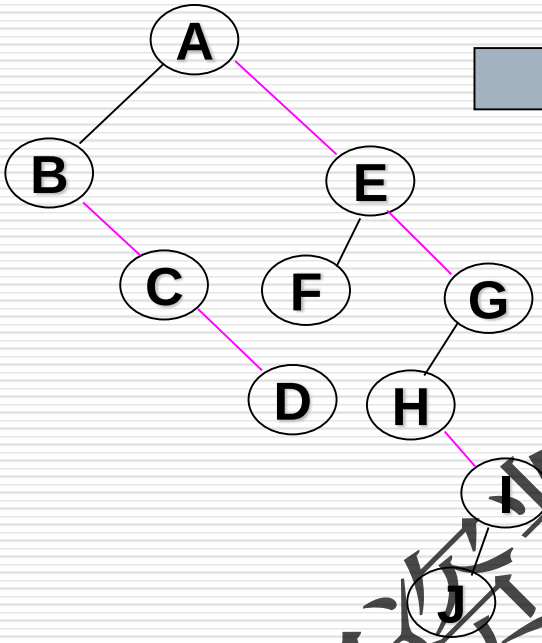
中根序列：B C D A F E H J I G

■中根遍历

- 若森林为空，返回；
- 中根遍历森林中第一棵树的根结点的子树森林；
- 访问第一棵树的根结点；
- 中根遍历除去第一棵树之后剩余的树构成的森林。

讨论：

若采用“先转换，后遍历”方式，结果是否相同？



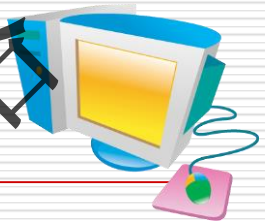
先序序列：A B C D E F G H I J

中序序列：B C D A F E H J I G

结论：森林的先根和中根遍历在两种方式下的结果相同。

但森林的后根遍历则不一定，因而少用

练习

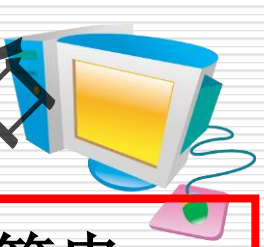


1.在下列存储形式中，哪一个不是树的存储形式？
（ D ）【北方交通大学 2001】
A．双亲表示法　B．孩子链表表示法
C．孩子兄弟表示法 D．顺序存储表示法

2. 某二叉树中序序列为A,B,C,D,E,F,G，后序序列为B,D,C,A,F,G,E 则前序序列是（ B）【南京理工大学2000 】
A．E,G,F,A,C,D,B　　B．E,A,C,B,D,G,F
C．E,A,G,C,F,B,D　　D．上面的都不对

3.上题的二叉树对应的森林包括多少棵树（ B ）【南京理工大学 2000 】
A．1　　B．2　　C．3　　D．概念上是错误的

5.7 哈夫曼树及其应用



在远程通讯中，要将待传字符转换成二进制的字符串，怎样编码才能使它们组成的报文在网络中传得最快？

A	00
B	01
C	10
D	11

ABACCDAAAA

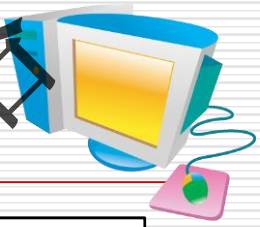
A	0
B	00
C	1
D	01

00010010101100000000

000011010000

出现次数较多的字符采用尽可能短的编码

哈夫曼树应用实例——哈夫曼编码



ABACCDAAAA

A	0
B	00
C	1
D	01

0000

AAAA ABA BB

重码

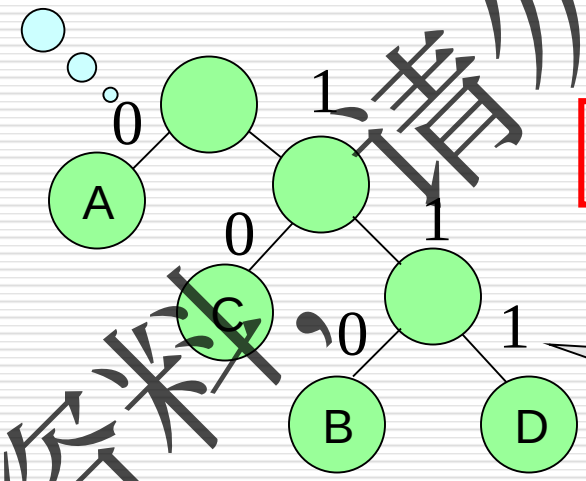
000011010000

关键：要设计长度不等的编码，则必须使任一字符的编码都不是另一个字符的编码的前缀——前缀编码

ABACCDAAAAA

A—0
B—110
C—10
D—111

采用二叉树设计
前缀编码

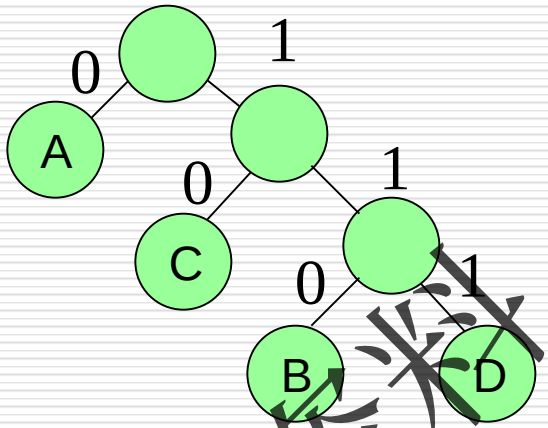


011001010111000 0

左分支用“0”
右分支用“1”

哈夫曼编码的译码过程

分解接收字符串：遇“0”向左，遇“1”向右；一旦到达叶子结点，则译出一个字符，反复由根出发，直到译码完成。

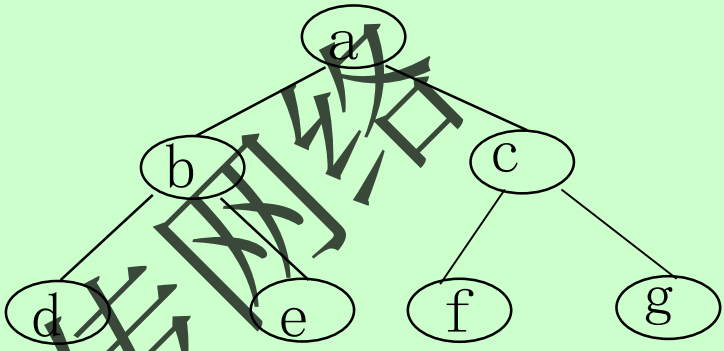


0110010101110000

ABACCDAAAA

特点：每一码都不是另一码的前缀，绝不会错译！称为前缀码

1. 若干术语



路　　径： 由一结点到另一结点间的分支所构成

路径长度： 路径上的分支数目　　a→e的路径长度＝2

带权路径长度： 结点到根的路径长度与结点上权的乘积

树的带权路径长度： 树中所有叶子结点的带权路径长度之和

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

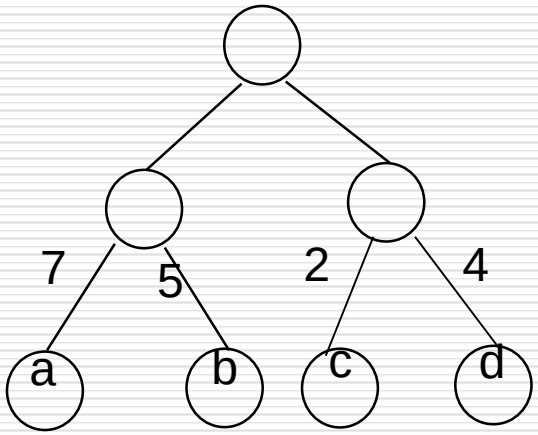
哈 夫 曼 树： 带权路径长度最小的树

树的带权路径长度如何计算？

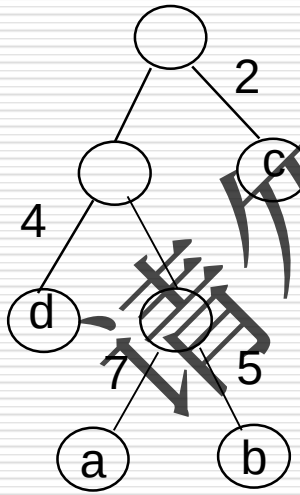
$$WPL = \sum_{k=1}^{n} w_k l_k$$

树中所有叶子结点的带权路径长度之和

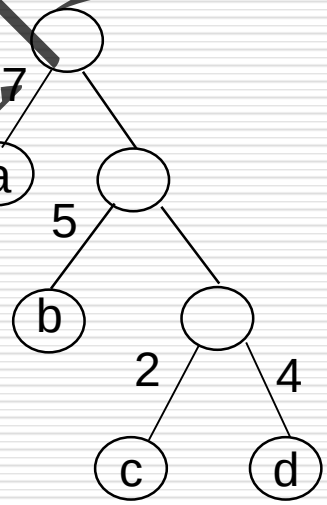
例：



(a)



(b)



(c)

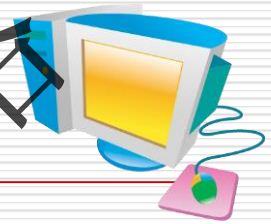
WPL= 36

WPL= 46

WPL= 35

Huffman树是*WPL* 最小的树

2. 哈夫曼树的构造算法



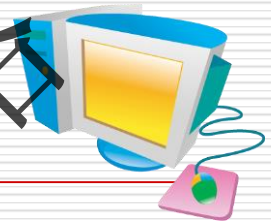
（1）根据给定的n个权值$\{w_1, w_2, \cdots\cdots w_n\}$，构造n棵只有根结点的二叉树，这n棵二叉树构成一个森林F。

（2）在森林F中选取两棵根结点r的权值最小的树作左右子树，构造一棵新的二叉树，置新二叉树根结点权值为其左右子树根结点权值之和。

（3）在森林中删除这两棵树，同时将新得到的二叉树加入F中。

（4）重复（2）和（3），直到F只含一棵树为止，这棵树便是哈夫曼树。

哈夫曼树的特点：



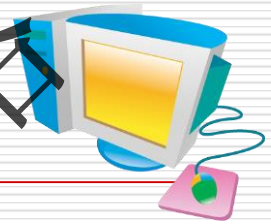
（1）哈夫曼树的*WPL*最小；

（2） 哈夫曼树肯定没有度为1的结点；

（3）一棵有n 个叶子结点的哈夫曼树，共有2n-1个结点；

（4） n个叶子结点的哈夫曼树最多有n层。

例



1）下面几个符号串编码集合中，不是前缀编码的是（ B ）。
【西安电子科技大学2001】

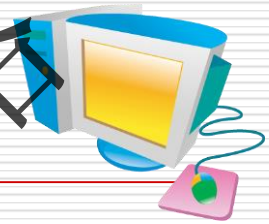
A．{0,10,110,1111}　　B．{11,10,001,101,0001}

C．{00,010,0110,1000}　　D．{b,c,aa,ac,aba,abb,abc}

2）一棵二叉树高度为h,所有结点的度或为0，或为2，则这棵二叉树最少有(B)结点。【南京理工大学2001】

A．2h　B．2h-1　C．2h+1　D．h+1

如何编程实现Huffman编码？



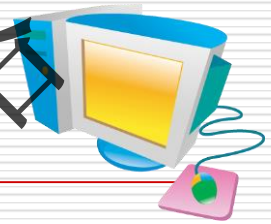
Huffman树中结点的结构可设计成4分量形式：

weight	parent	lchild	rchild

Huffman树的存储结构可采用顺序存储结构：

```
typedef struct{
int weight；  //权值分量（可放大取整）
int parent，lchild，rchild；  //双亲和孩子分量
}HTNode，*HuffmanTree；//用动态数组存储Huffman树
```

哈夫曼树构造算法的实现



1)初始化HT[1..2n-1]：lchild=rchild=parent=0

2)输入初始n个叶子结点：置HT[1..n]的weight值

3)进行以下n-1次合并，依次产生HT[i]，i=n+1..2n-1：
①在HT[1..i-1]中选两个未被选过的weight最小的两个结点HT[s1]和HT[s2]（从parent = 0 的结点中选）
②修改HT[s1]和HT[s2]的parent值： parent=i
③置HT[i]：weight=HT[s1].weight + HT[s2].weight ，lchild=s1，rchild=s2

例:设n=8, w={5,29,7,8,14,23,3,11}

算法5.10 构造哈夫曼树

```
void CreatHuffmanTree
(HuffmanTree HT,int n)
{  if(n<=1)return;
  m=2*n-1;
  HT=new HTNode[m+1];
//0号单元未用，HT[m]表示根结点
 for(i=1;i<=m;++i)
  {HT[i].parent=0;
   HT[i].lchild=0;
   HT[i].rchild=0;
 }
for(i=1;i<=n;++i)
cin>>HT[i].weight;
```

	weight	parent	lchild	rchild
1				
2				
3				
4				
5				
6				
7				
8				
9				
10				
11				
12				
13				
14				
15				

例:设n=8, w={5,29,7,8,14,23,3,11}

算法5.10 构造哈夫曼树

```
void CreatHuffmanTree
(HuffmanTree HT,int n)
{  if(n<=1)return;
  m=2*n-1;
  HT=new HTNode[m+1];
//0号单元未用，HT[m]表示根结点
 for(i=1;i<=m;++i)
  {HT[i].parent=0;
   HT[i].lchild=0;
   HT[i].rchild=0;
 }
for(i=1;i<=n;++i)
cin>>HT[i].weight;
```

	weight	parent	lchild	rchild
1		0	0	0
2				
3				
4				
5				
6				
7				
8				
9				
10				
11				
12				
13				
14				
15				

例:设n=8, w={5,29,7,8,14,23,3,11}

算法5.10 构造哈夫曼树

```
void CreatHuffmanTree
(HuffmanTree HT,int n)
{  if(n<=1)return;
  m=2*n-1;
  HT=new HTNode[m+1];
//0号单元未用，HT[m]表示根结点
 for(i=1;i<=m;++i)
  {HT[i].parent=0;
   HT[i].lchild=0;
   HT[i].rchild=0;
 }
for(i=1;i<=n;++i)
cin>>HT[i].weight;
```

	weight	parent	lchild	rchild
1		0	0	0
2		0	0	0
3		0	0	0
4		0	0	0
5		0	0	0
6		0	0	0
7		0	0	0
8		0	0	0
9				
10				
11				
12				
13				
14				
15				

例:设n=8, w={5,29,7,8,14,23,3,11}

算法5.10 构造哈夫曼树

```
void CreatHuffmanTree
(HuffmanTree HT,int n)
{  if(n<=1)return;
  m=2*n-1;
  HT=new HTNode[m+1];
//0号单元未用，HT[m]表示根结点
 for(i=1;i<=m;++i)
  {HT[i].parent=0;
   HT[i].lchild=0;
   HT[i].rchild=0;
 }
for(i=1;i<=n;++i)
cin>>HT[i].weight;
```

	weight	parent	lchild	rchild
1	5	0	0	0
2	29	0	0	0
3	7	0	0	0
4	8	0	0	0
5	14	0	0	0
6	23	0	0	0
7	3	0	0	0
8	11	0	0	0
9				
10				
11				
12				
13				
14				
15				

```
for( i=n+1;i<=m;++i)
   //构造 Huffman树
 { Select(HT,i-1, s1, s2);
 //在HT[k](1≤k≤i-1)中选择两个其
 //双亲域为0, 且权值最小的结点,
 // 并返回它们在HT中的序号s1和s2
 HT[s1].parent=i; HT[s2] .parent=i;
     //表示从F中删除s1,s2
 HT[i].lchild=s1;    HT[i].rchild=s2;
     //s1,s2分别作为i的左右孩子
 HT[i].weight=HT[s1].weight +
HT[s2] .weight;
    //i 的权值为左右孩子权值之和
    }
}
```

	weight	parent	lchild	rchild
1	5	0	0	0
2	29	0	0	0
3	7	0	0	0
4	8	0	0	0
5	14	0	0	0
6	23	0	0	0
7	3	0	0	0
8	11	0	0	0
9				
10				
11				
12				
13				
14				
15				

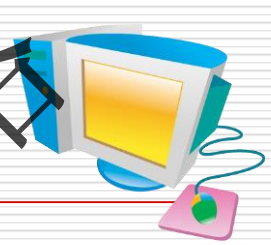
```
for( i=n+1;i<=m;++i)
   //构造 Huffman树
 { Select(HT,i-1, s1, s2);
 //在HT[k](1≤k≤i-1)中选择两个其
 //双亲域为0, 且权值最小的结点,
 // 并返回它们在HT中的序号s1和s2
 HT[s1].parent=i; HT[s2] .parent=i;
     //表示从F中删除s1,s2
 HT[i].lchild=s1;    HT[i].rchild=s2;
     //s1,s2分别作为i的左右孩子
 HT[i].weight=HT[s1].weight +
HT[s2] .weight;
    //i 的权值为左右孩子权值之和
    }
}
```

	weight	parent	lchild	rchild
1	5	0	0	0
2	29	0	0	0
3	7	0	0	0
4	8	0	0	0
5	14	0	0	0
6	23	0	0	0
7	3	0	0	0
8	11	0	0	0
9				
10				
11				
12				
13				
14				
15				

```
for( i=n+1;i<=m;++i)
   //构造 Huffman树
 { Select(HT,i-1, s1, s2);
 //在HT[k](1≤k≤i-1)中选择两个其
 //双亲域为0, 且权值最小的结点,
 // 并返回它们在HT中的序号s1和s2
 HT[s1].parent=i; HT[s2] .parent=i;
     //表示从F中删除s1,s2
 HT[i].lchild=s1;    HT[i].rchild=s2;
     //s1,s2分别作为i的左右孩子
 HT[i].weight=HT[s1].weight +
HT[s2] .weight;
    //i 的权值为左右孩子权值之和
    }
}
```

	weight	parent	lchild	rchild
1	5	9	0	0
2	29	14	0	0
3	7	10	0	0
4	8	10	0	0
5	14	12	0	0
6	23	13	0	0
7	3	9	0	0
8	11	11	0	0
9	8	11	7	1
10	15	12	3	4
11	19	13	9	8
12	29	14	5	10
13	42	15	11	6
14	58	15	2	12
15	100	0	13	14

3. 哈夫曼编码



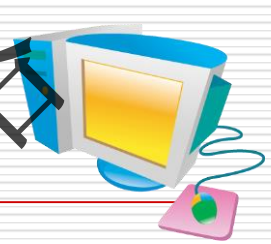
算法5.11 根据哈夫曼树求哈夫曼编码

```
void CreatHuffmanCode(HuffmanTree HT, HuffmanCode &HC, int n){
//从叶子到根逆向求每个字符的赫夫曼编码，存储在编码表HC中
HC=new char *[n+1];
//分配n个字符编码的头指针矢量
cd=new char [n];
//分配临时存放编码的动态数组空间
cd[n-1]='\0';          //编码结束符
```

(续前)再求出n个字符的Huffman编码HC

```
for(i=1; i<=n; ++i){ //逐个字符求赫夫曼编码
    start=n-1; c=i; f=HT[i].parent;
       while(f!=0)   //从叶子结点开始向上回溯，直到根结点
      {  --start;      //回溯一次start向前指一个位置
         if (HT[f].lchild= =c)  cd[start]=’0’;
        //结点c是f的左孩子，则生成代码0
         else cd[start]=’1’;
        //结点c是f的右孩子，则生成代码1
         c=f; f=HT[f].parent;                    //继续向上回溯
        }                                  //求出第i个字符的编码
    HC[i]= new char [n-start];      // 为第i 个字符编码分配空间
    strcpy(HC[i], &cd[start]);
          //将求得的编码从临时空间cd复制到HC的当前行中
  }
  delete cd;                                   //释放临时空间
} // CreatHuffanCode
```

Huffman编码举例



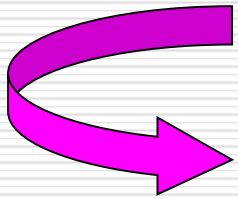
例1【严题集6.26③】：假设用于通信的电文仅由8个字母 {a, b, c, d, e, f, g, h} 构成，它们在电文中出现的概率分别为{ 0.07, 0.19, 0.02, 0.06, 0.32, 0.03, 0.21, 0.10 }，试为这8个字母设计哈夫曼编码。如果用0～7的二进制编码方案又如何？

【类同P148例2】

解：先将概率放大100倍，以方便构造哈夫曼树。

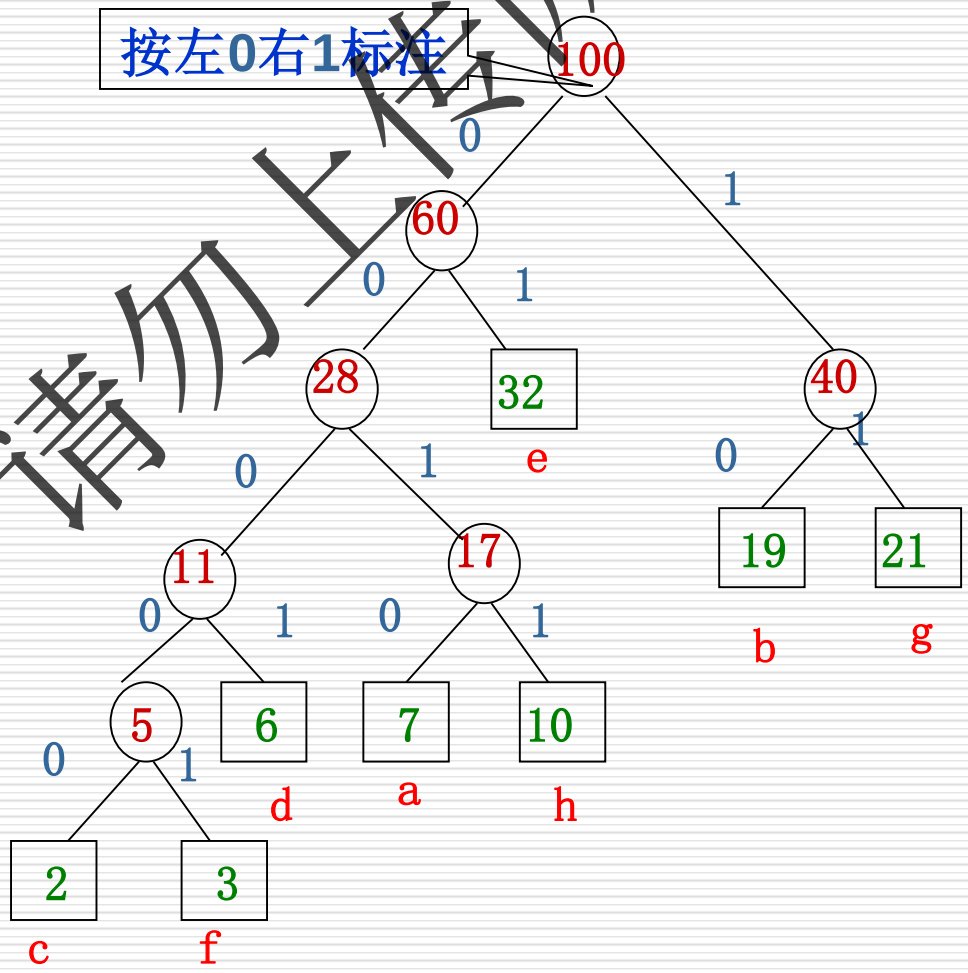
放大后的权值集合 w={ 7, 19, 2, 6, 32, 3, 21, 10 },

按哈夫曼树构造规则（合并、删除、替换），可得到哈夫曼树。



对应的哈夫曼编码：

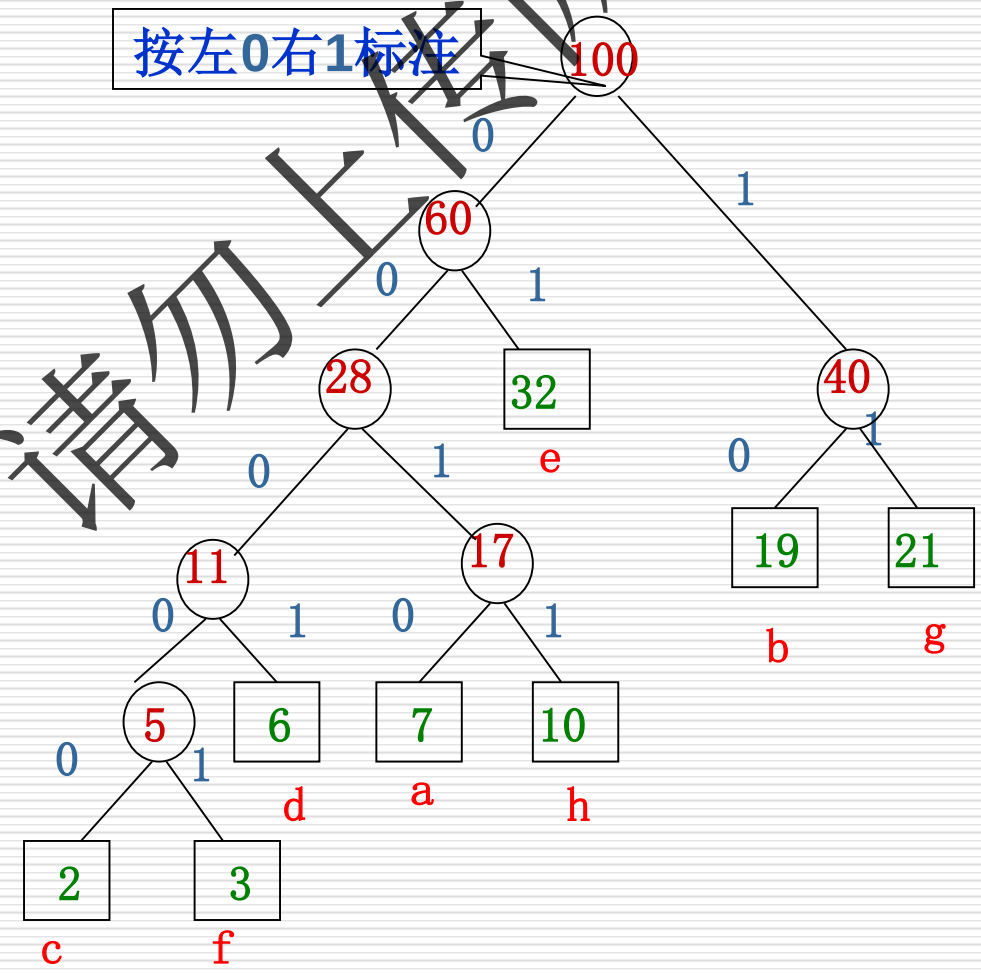
符	编码	频率
a	0010	0.07
b	10	0.19
c	00000	0.02
d	0001	0.06
e	01	0.32
f	00001	0.03
g	11	0.21
h	0011	0.10



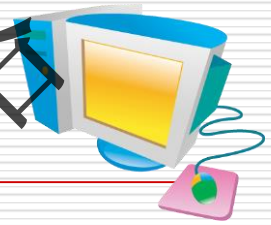
对应的哈夫曼编码：

Huffman码的WPL＝
2(0.19+0.32+0.21) +
4(0.07+0.06+0.10)
+5(0.02+0.03)
=1.44+0.92+0.25=2.61

二进制等长码的WPL＝
3(0.19+0.32+0.21+0
.07+0.06+0.10+0.02
+0.03)=3



小结



1. Huffman算法的思路：

——权值大的结点用短路径，权值小的结点用长路径。

2. 构造Huffman树的步骤：

——对权值的合并、删除与替换

3. Huffman编码规则： 左“0”右“1”

——又称为前缀码、最小冗余编码、紧致码等等，它是数据压缩学的基础。

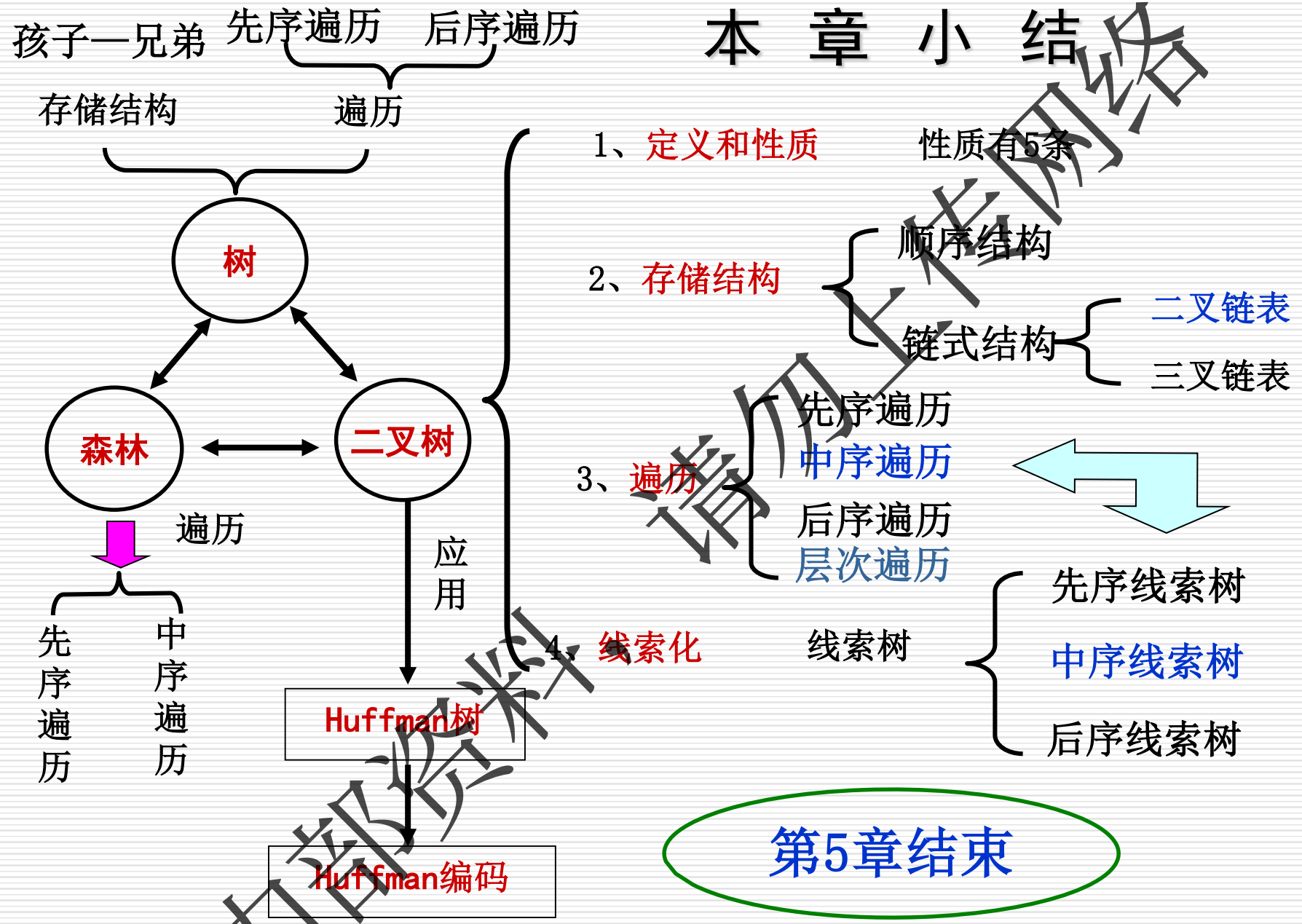
5.8 案例分析与实现

案例5.2 ：利用二叉树求解表达式的值

【案例实现】

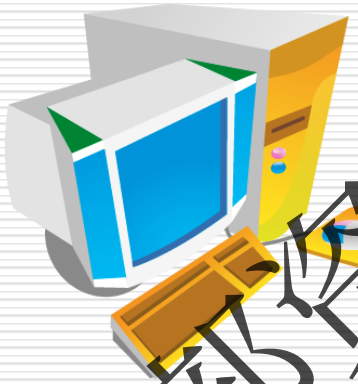
- 假设运算符均为双目运算符，则表达式对应的表达式树中叶子结点均为操作数，分支结点均为运算符。
- 由于创建的表达式树需要准确的表达运算次序，因此在扫描表达式创建表达式树的过程中，当遇到运算符时不能直接创建结点，而应将其与前面的运算符进行优先级比较，根据比较的结果再进行处理。
- 借助一个运算符栈OPTR，来暂存已经扫描到的还未处理的运算符。
- 每两个操作数和一个运算符就可以建立一棵表达式二叉树，而该二叉树又可以作为另一个运算符结点的一棵子树。
- 另外借助一个表达式树栈EXPT，来暂存已建立好的表达式树的根结点，以便其作为另一个运算符结点的子树而被引用。

本 章 小 结



第5章结束

第五章　结束



Thank you!