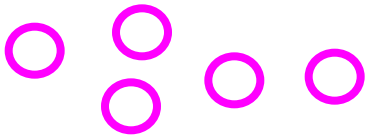
第五章 树和二叉树

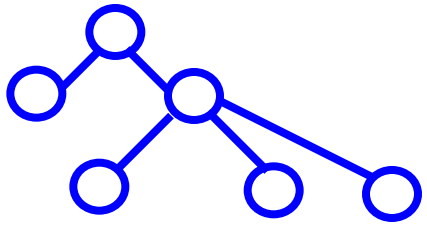
逻辑结构

集合——数据元素间除“同属于一个集合”外，无其它关系

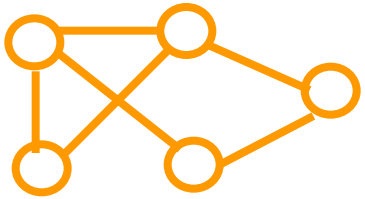


线性结构——一个对一个，如线性表、栈、队列

树形结构——一个对多个，如树



图形结构——多个对多个，如图



第5章　树和二叉树

5.1 树和二叉树的定义

5.2 案例引入

5.3 二叉树的抽象数据类型定义

5.4 二叉树的性质和存储结构

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

教学目标

1. 掌握二叉树的基本概念、性质和存储结构
2. 熟练掌握二叉树的前、中、后序遍历方法
3. 了解线索化二叉树的思想
4. 熟练掌握：哈夫曼树的实现方法、构造哈夫曼编码的方法
5. 了解：森林与二叉树的转换，树的遍历方法

5.1 树和二叉树的定义

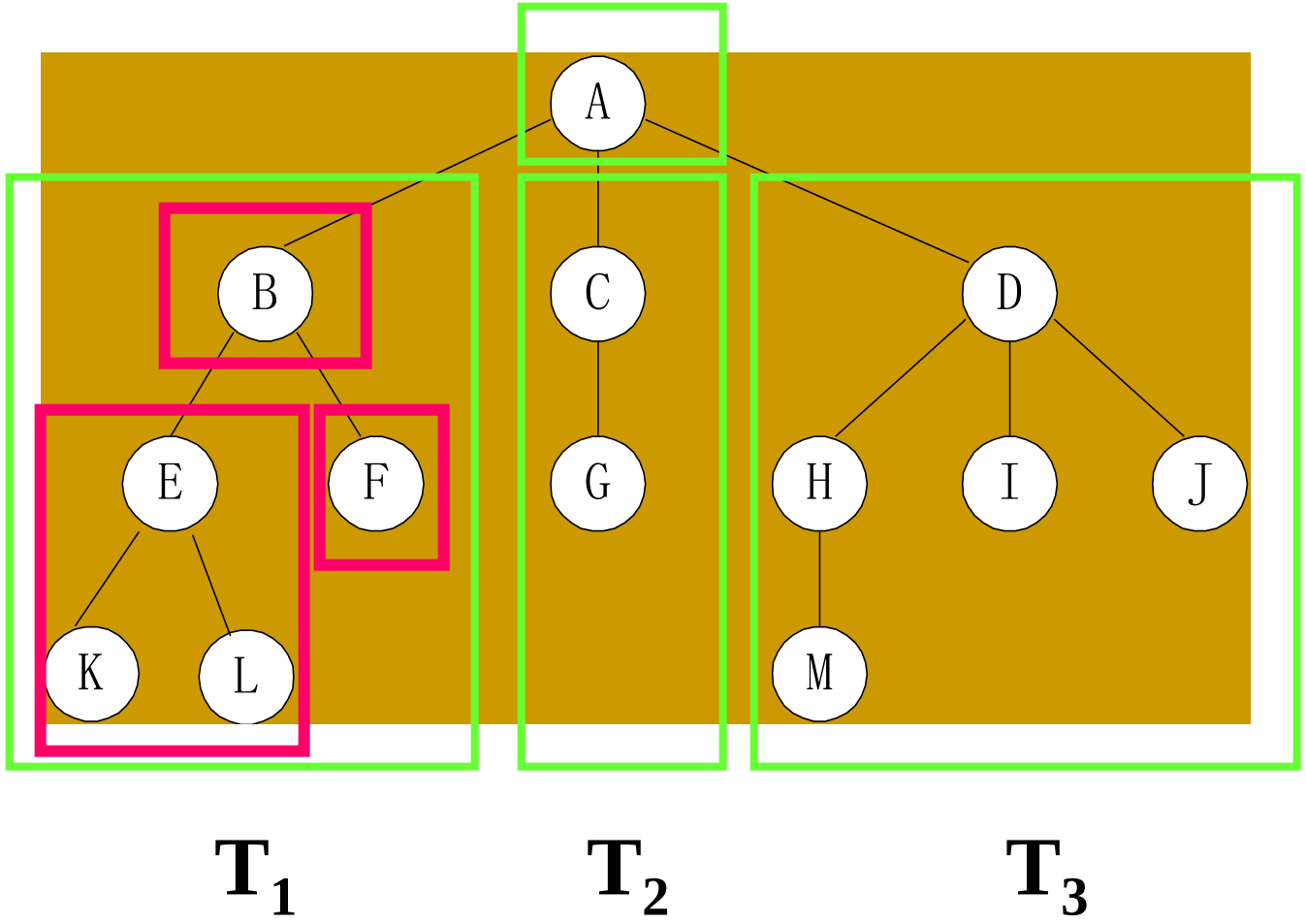
树的定义

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

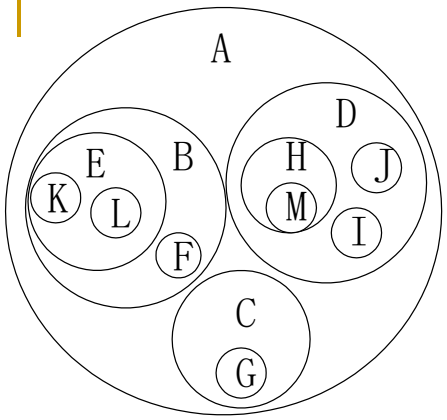
（1）有且仅有一个称之为根的结点；

（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$T_1, T_2, \cdots, T_m$，其中每一个集合本身又是一棵树，并且称为根的子树（SubTree）。

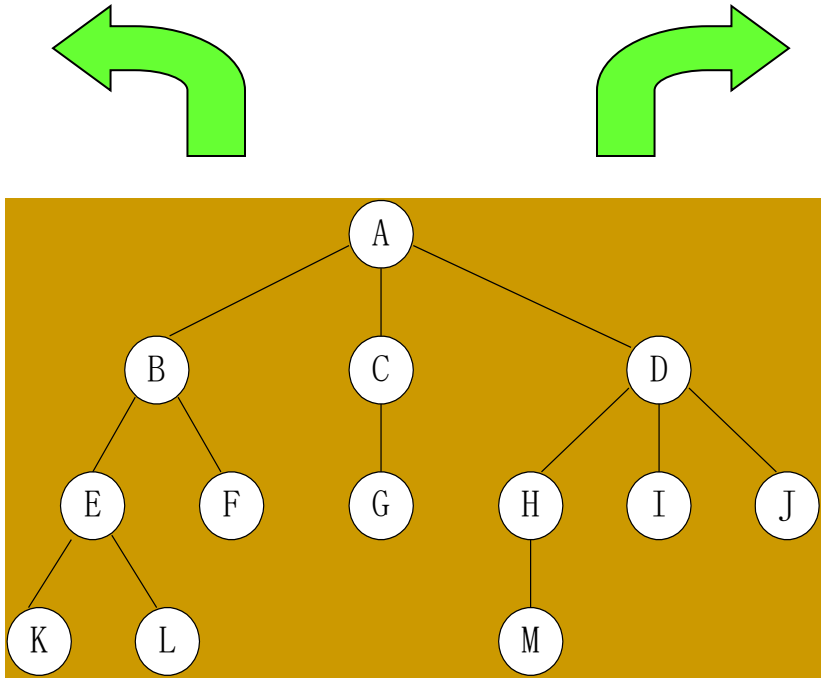
树是n个结点的有限集

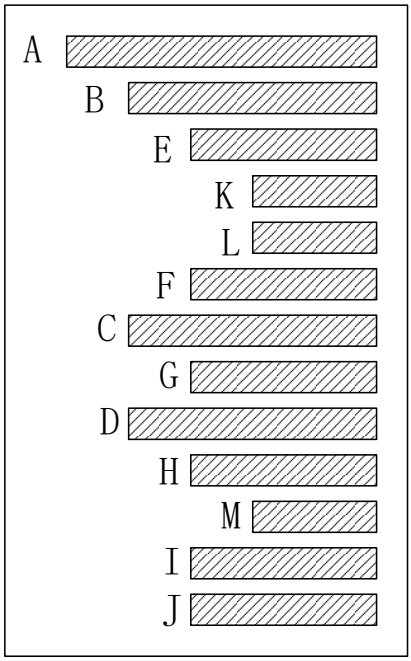


树的其它表示方式



嵌套集合





凹入表示

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

广义表

基本术语

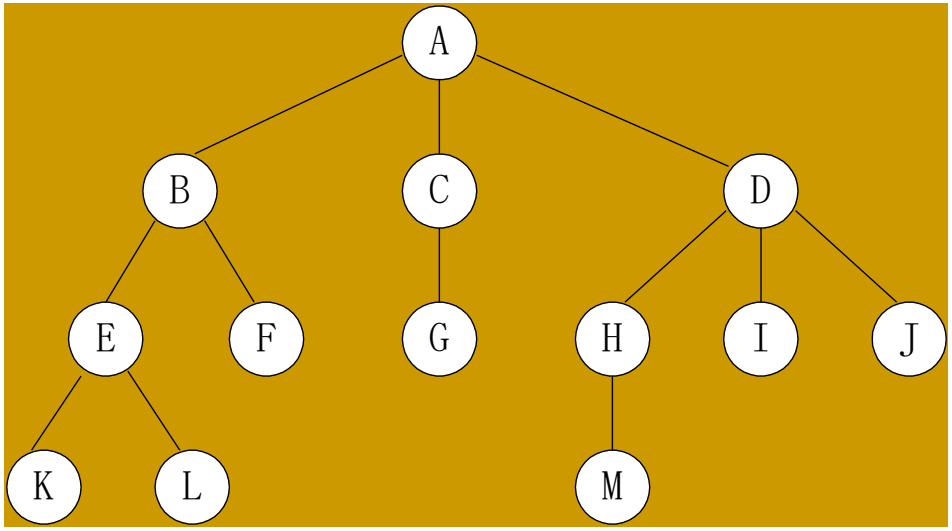
根 ——即根结点(没有前驱)

叶子 ——即终端结点(没有后继)

森林 ——指m棵不相交的树的集合(例如删除A后的子树个数)

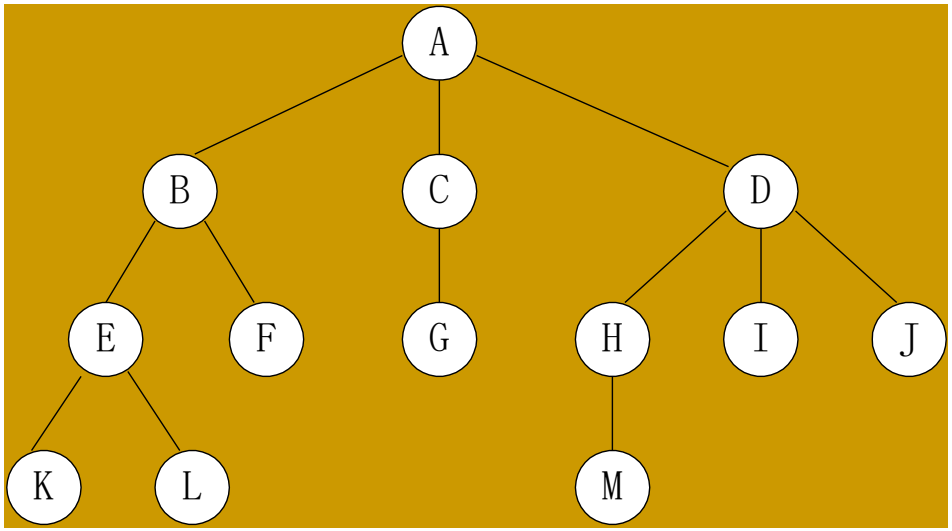
有序树 ——结点各子树从左至右有序，不能互换（左为第一）

无序树 ——结点各子树可互换位置。



基本术语

双亲 ——即上层的那个结点(直接前驱)
孩子 ——即下层结点的子树的根(直接后继)
兄弟 ——同一双亲下的同层结点（孩子之间互称兄弟）
堂兄弟 ——即双亲位于同一层的结点（但并非同一双亲）
祖先 ——即从根到该结点所经分支的所有结点
子孙 ——即该结点下层子树中的任一结点



基本术语

结点 ——即树的数据元素

结点的度——结点挂接的子树数

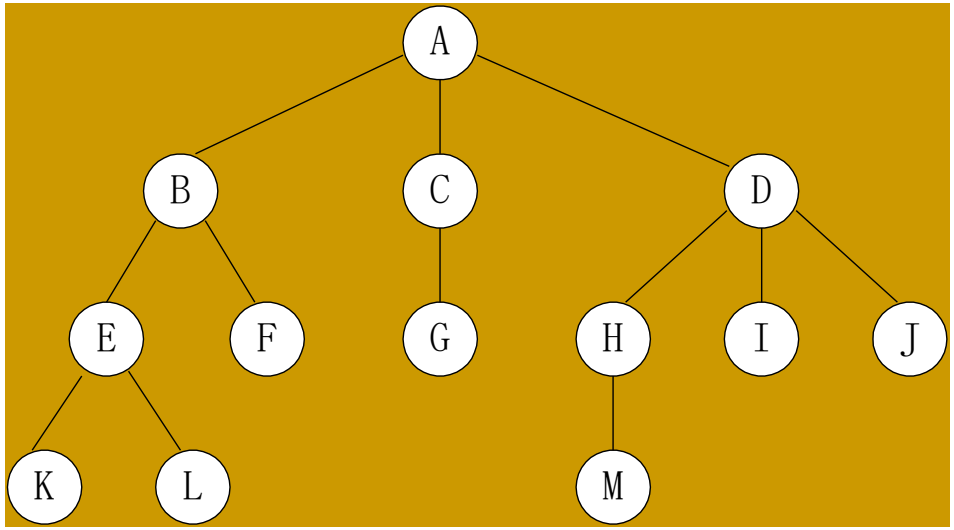
结点的层次——从根到该结点的层数（根结点算第一层）

终端结点 ——即度为0的结点，即叶子

分支结点 ——即度不为0的结点（也称为内部结点）

树的度 ——所有结点度中的最大值

树的深度(或高度)——指所有结点中最大的层数



层次

1

2

3

4

二叉树的定义

二叉树（Binary Tree）是n（$n \geqslant 0$）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T:

（1）有且仅有一个称之为根的结点；

（2）除根结点以外的其余结点分为两个互不相交的子集T_1和T_2，分别称为T的左子树和右子树，且T_1和T_2本身又都是二叉树。

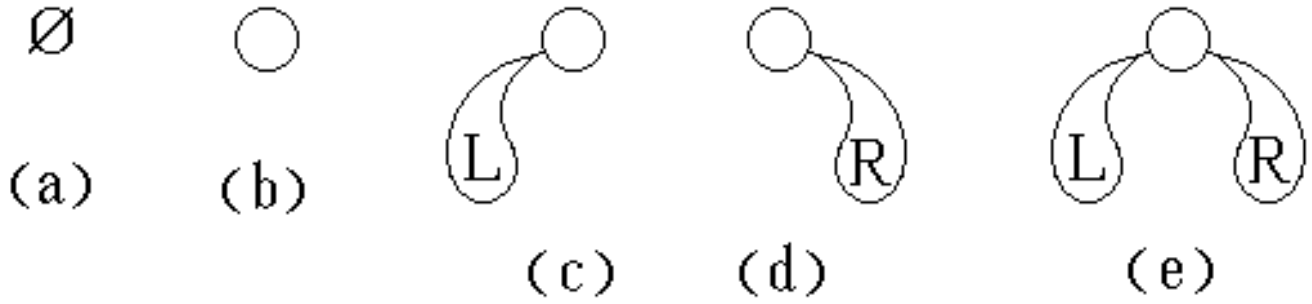
普通树（多叉树）若不转化为二叉树，则运算很难实现

为何要重点研究每结点最多只有两个 “叉” 的树？

- ✓ 二叉树的结构最简单，规律性最强；
- ✓ 可以证明，所有树都能转为唯一对应的二叉树，不失一般性。

二叉树基本特点：

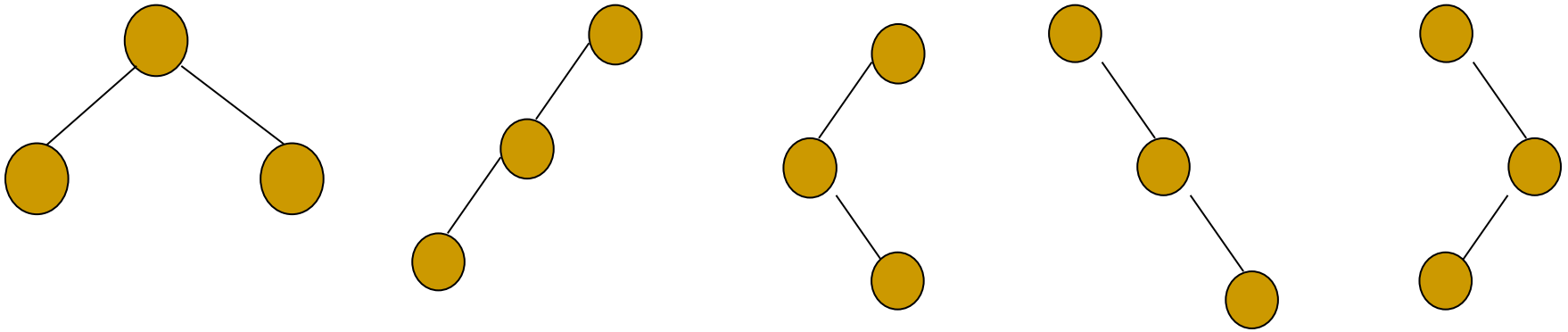
- 结点的度小于等于2
- 有序树（子树有序，不能颠倒）



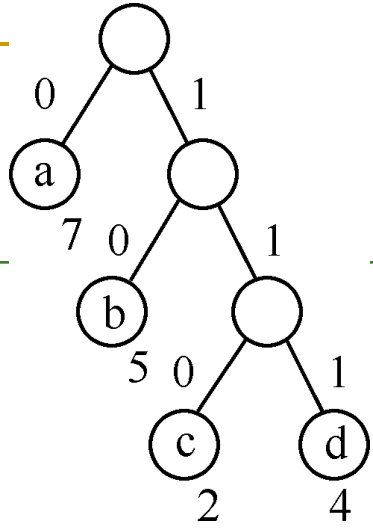
二叉树的五种不同形态

练习

具有3个结点的二叉树可能有几种不同形态？ 5种



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

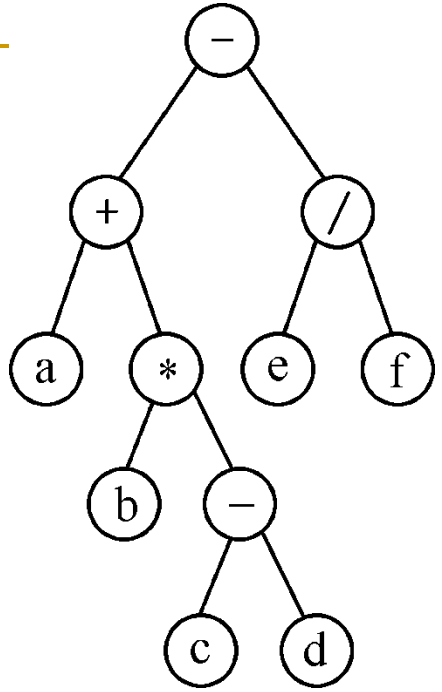
字符	编码
a	0
b	01
c	010
d	111

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 二叉树的抽象数据类型定义

二叉树的抽象数据类型定义

ADT BinaryTree{

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=Φ，则R= Φ ；**

若D≠Φ，则R= {H}；存在二元关系：

① root 唯一　　//关于根的说明

② $D_l \cap D_r = \Phi$　　//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

}ADT BinaryTree

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

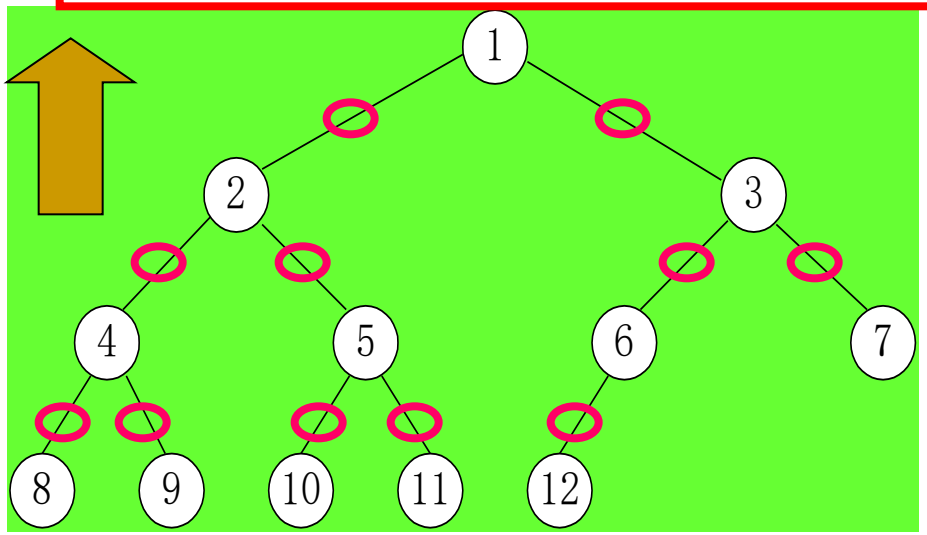
性质1： 在二叉树的第i层上至多有2^{i-1}个结点

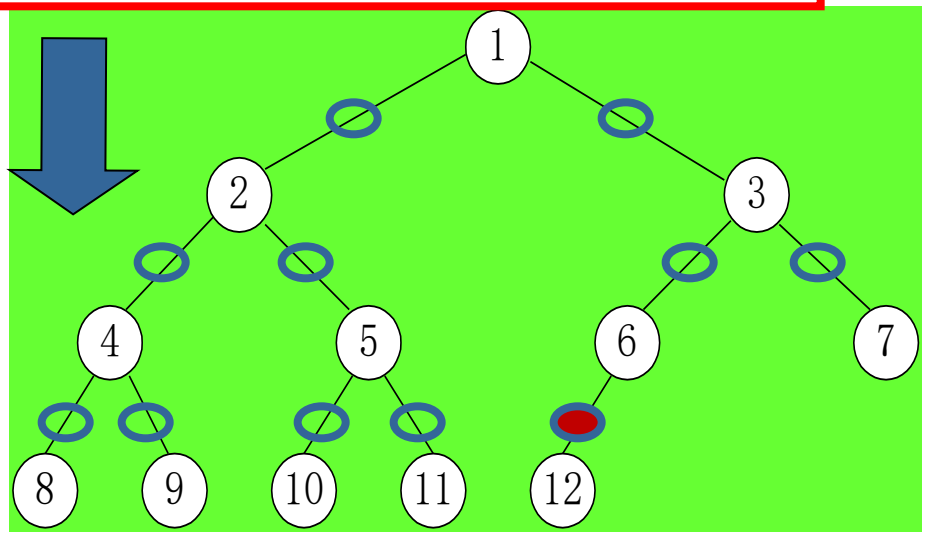
提问：第i层上至少有__1__个结点？

性质2： 深度为k的二叉树至多有2^k-1个结点

提问：深度为k时至少有__k__个结点？

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$n_0=n_2+1$）



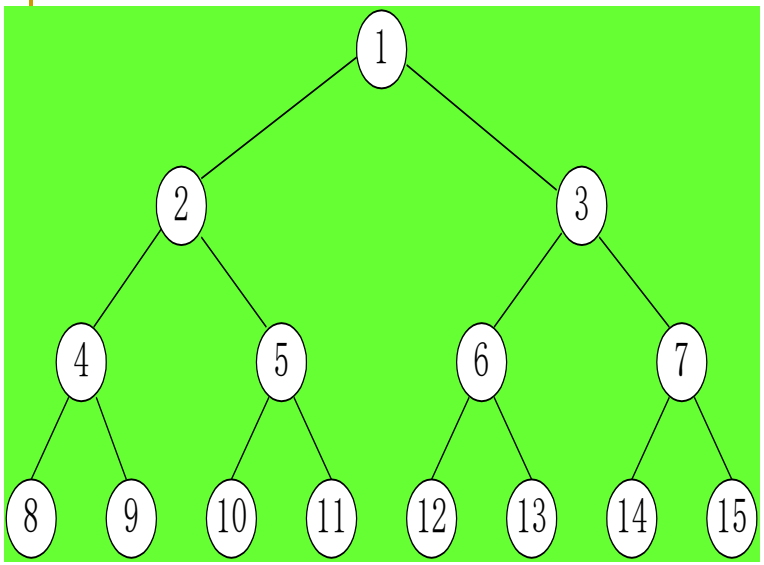


$$B=n-1$$

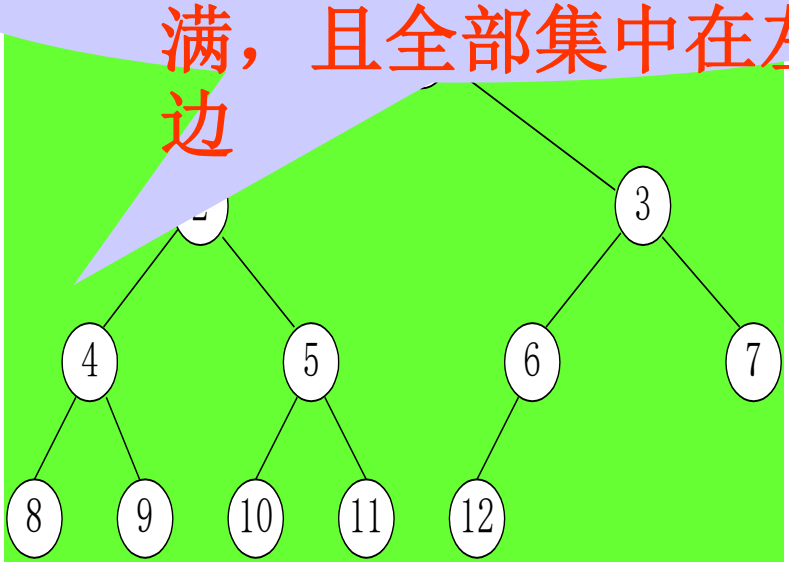
$$B=n_2\times 2+n_1\times 1$$

$$n=n_2\times 2+n_1\times 1+1=n_2+n_1+n_0$$

特殊形态的二叉树



只有最后一层叶子不满，且全部集中在左边

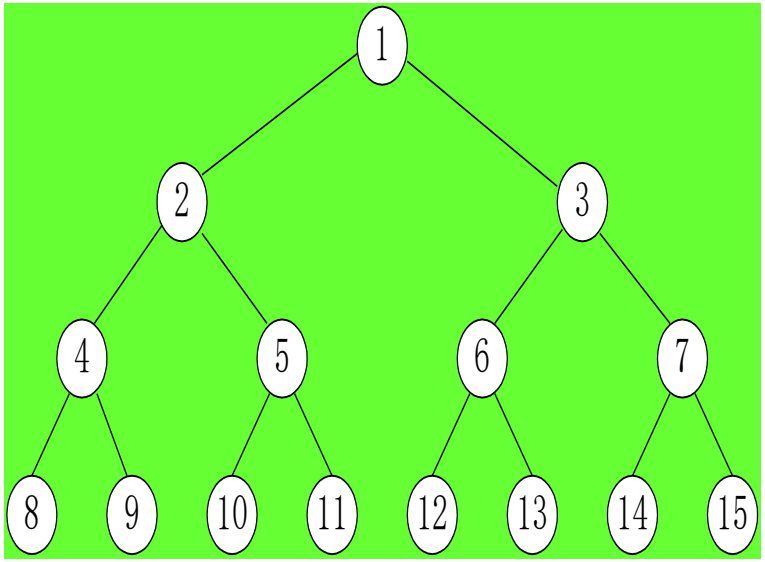


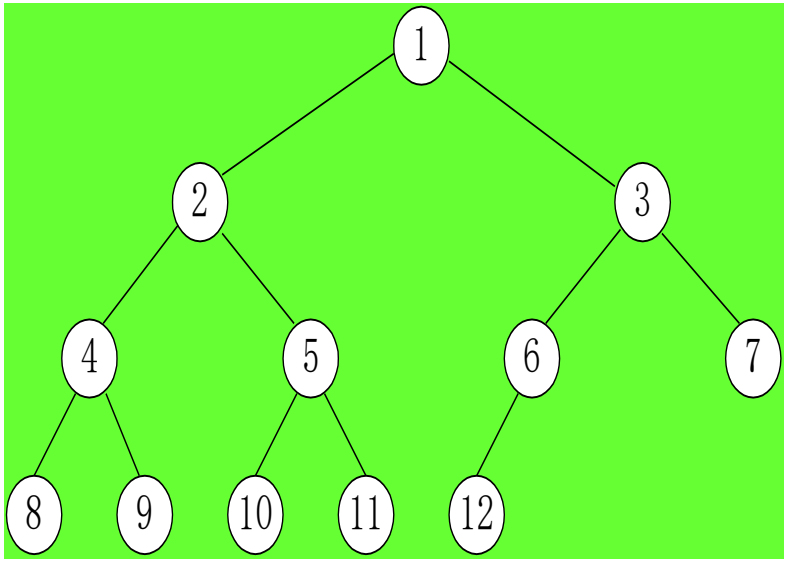
满二叉树：一棵深度为k 且有2^k -1个结点的二叉树。（特点：每层都“充满”了结点）

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

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n-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。

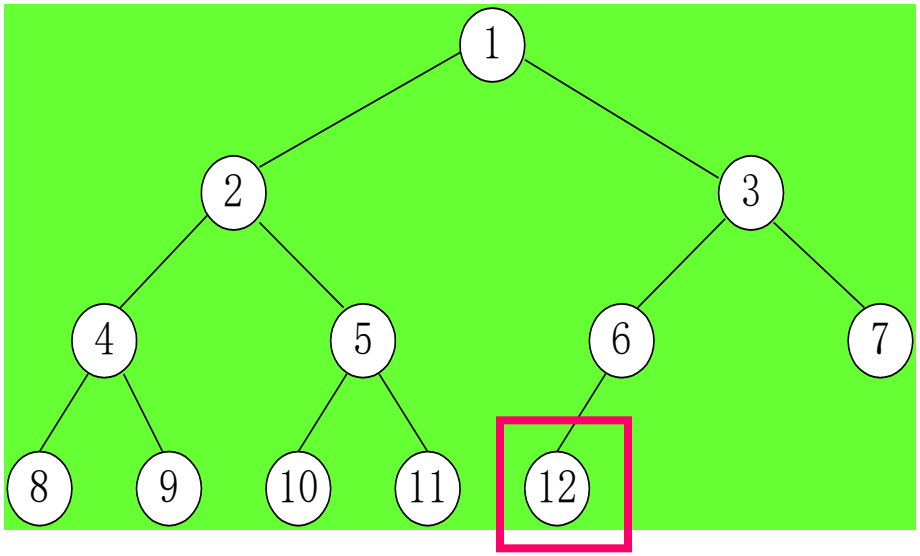




练习

一棵完全二叉树有5000个结点，可以计算出其叶结点的个数是（ 2500 ）。

性质4： 具有n个结点的完全二叉树的深度必为$\lfloor \log_2 n \rfloor + 1$



k-1层 $2^{k-1}-1$

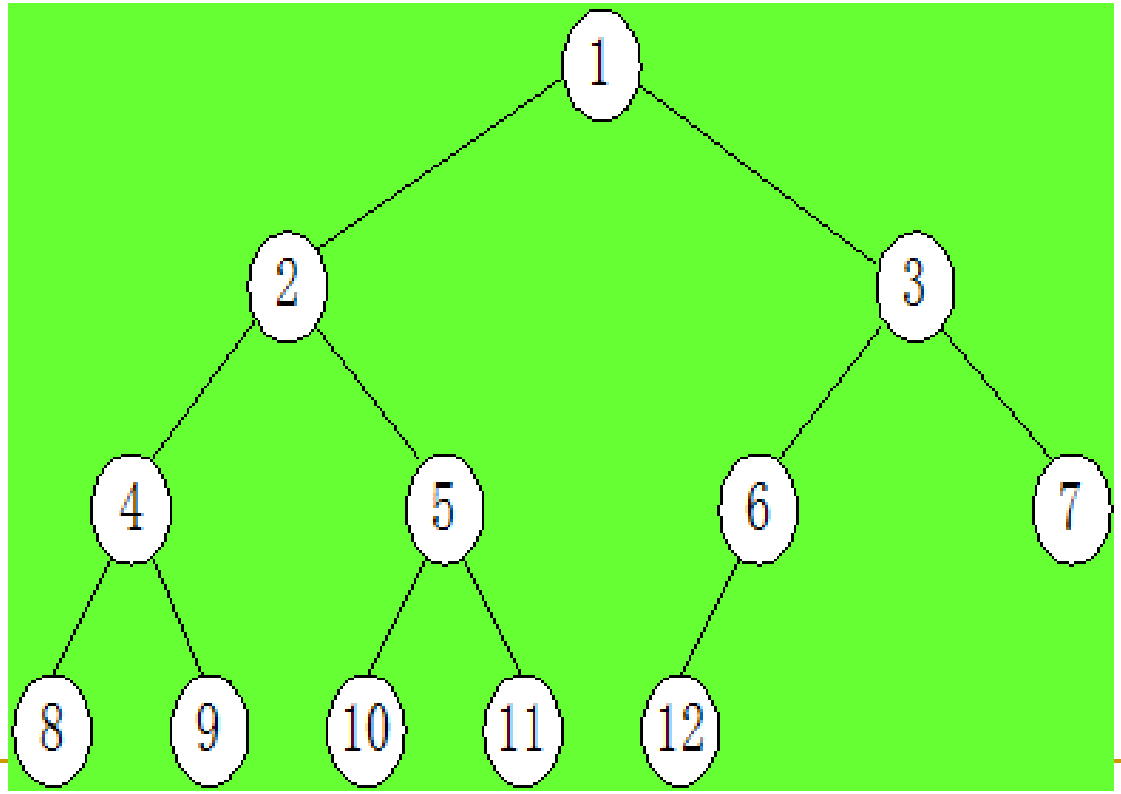
k层 $2^{k}-1$

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

$k-1 \leq \log_2 n < k$，因为k是整数

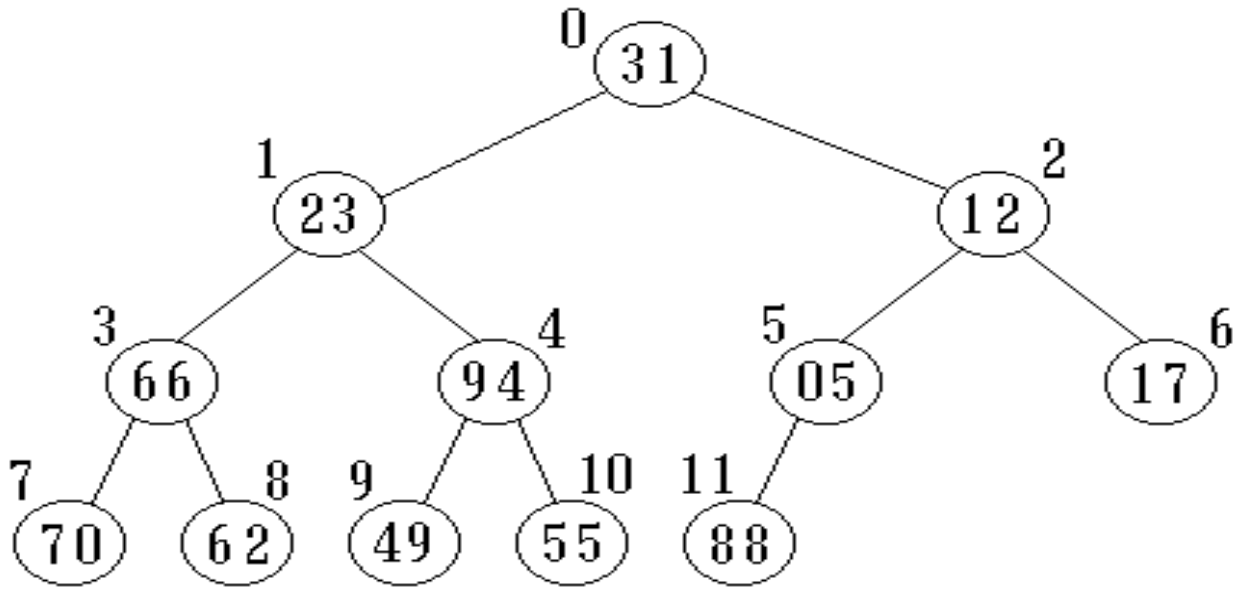
所以$k = \lfloor \log_2 n \rfloor + 1$

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，其右孩子编号必为2i＋1；其双亲的编号必为i/2。



二叉树的顺序存储

实现：按满二叉树的结点层次编号，依次存放二叉树中的数据元素。

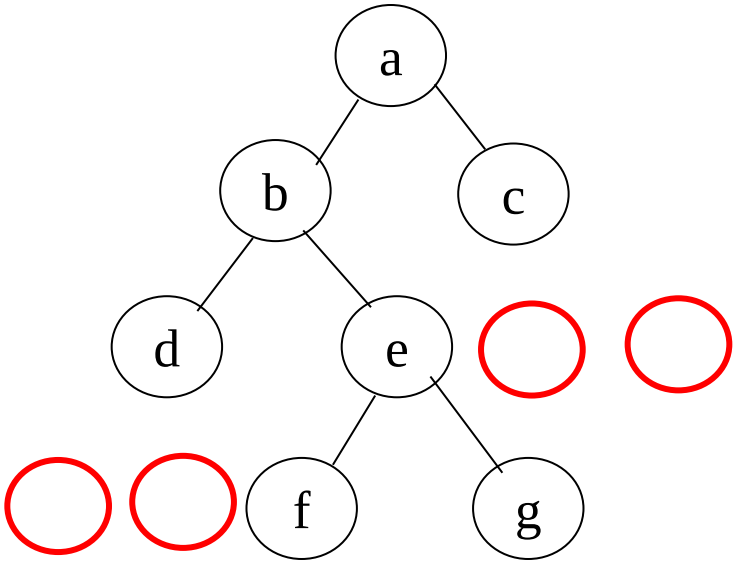


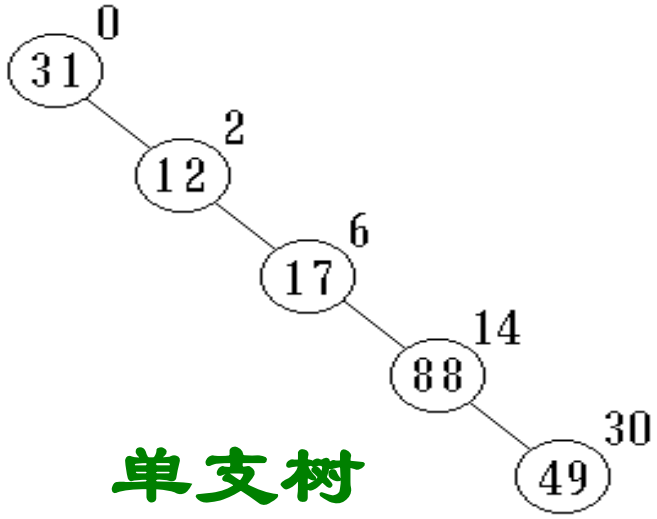
（a）

0	1	2	3	4	5	6	7	8	9	10	11
31	23	12	66	94	05	17	70	62	49	55	88

二叉树的顺序存储

0	1	2	3	4	5	6	7	8	9	10
a	b	c	d	e	0	0	0	0	f	g





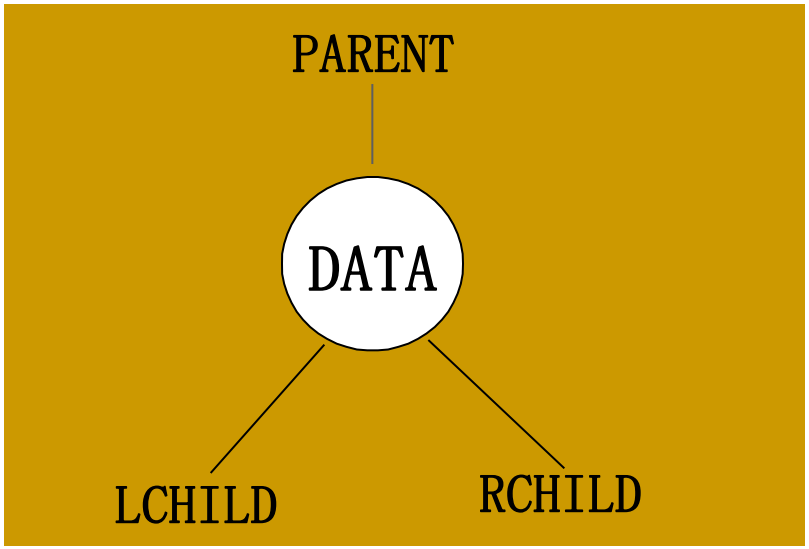
单支树

特点：

结点间关系蕴含在其存储位置中

浪费空间，适于存满二叉树和完全二叉树

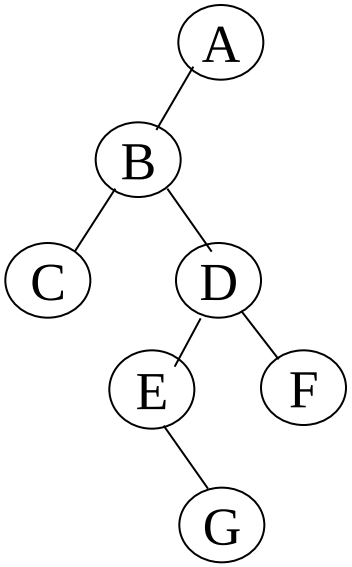
二叉树的链式存储

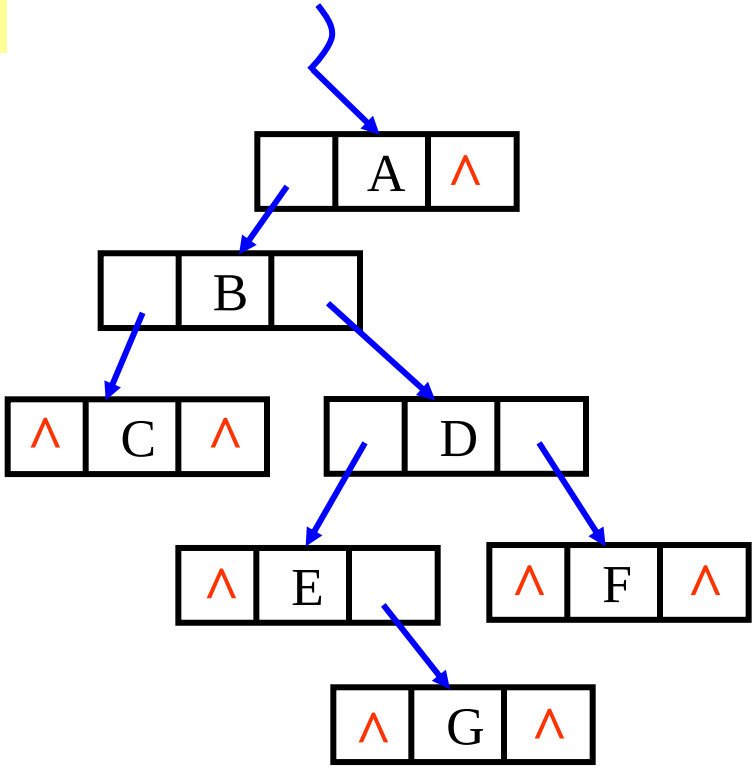


lchild	data	rchild

lchild	data	parent	rchild

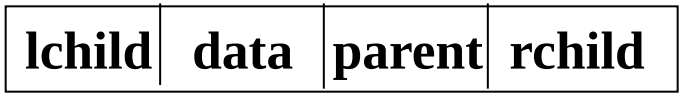
二叉链表

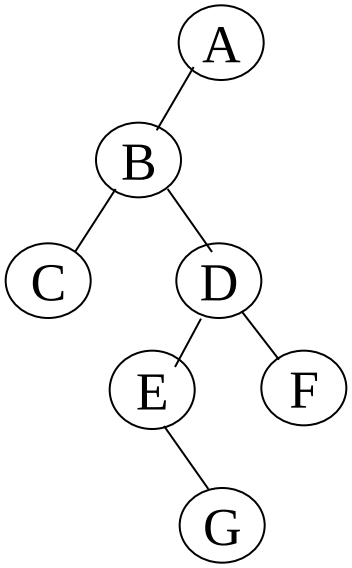


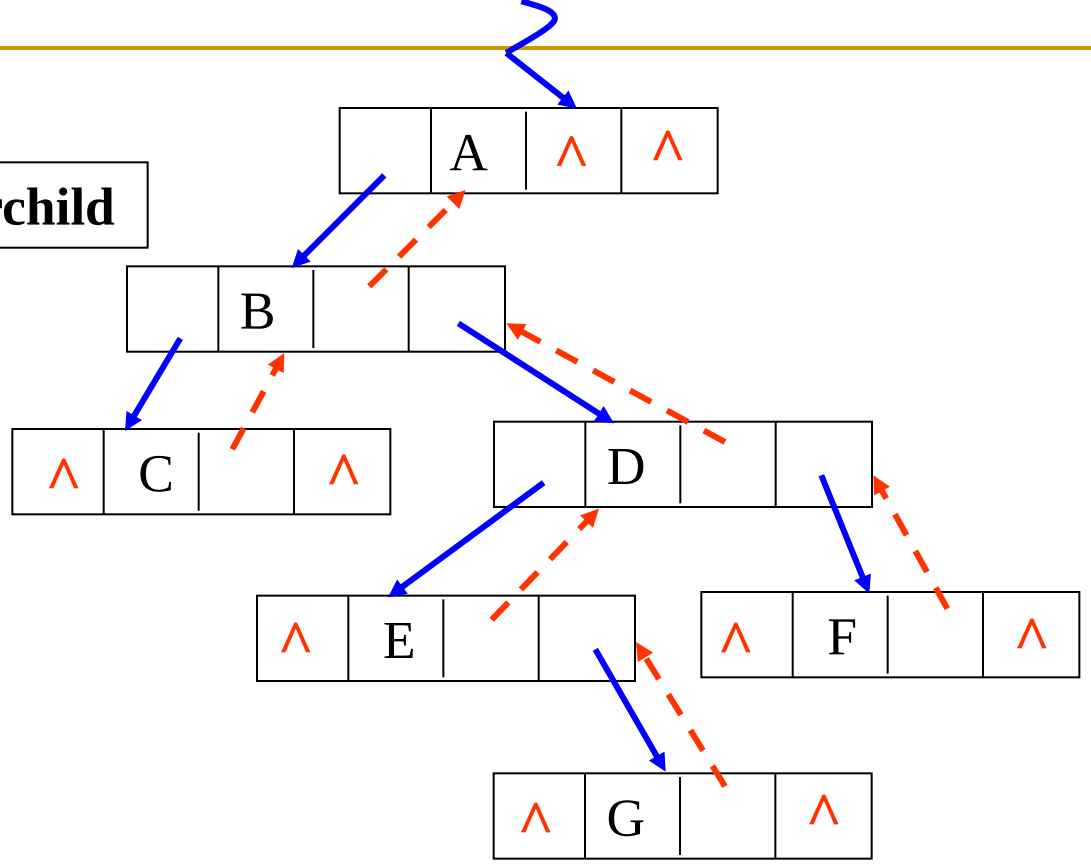



```
typedef struct BiNode{
   TElemType   data;
   struct  BiNode   *lchild,*rchild; //左右孩子指针
}BiNode,*BiTree;
```

三叉链表







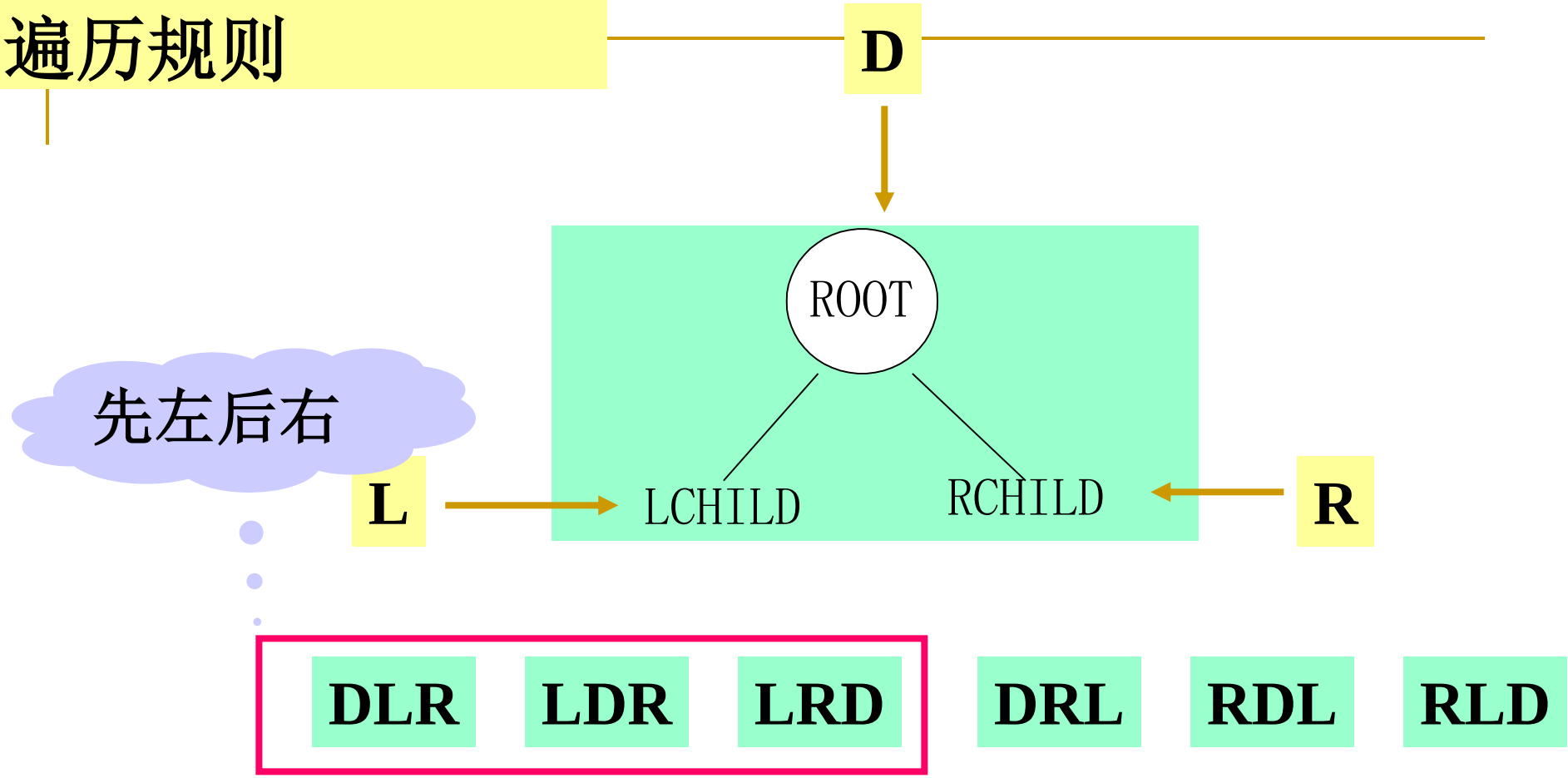

```
typedef struct TriTNode
{  TelemType data;
   struct TriTNode *lchild,*parent,*rchild;
 }TriTNode,*TriTree;
```

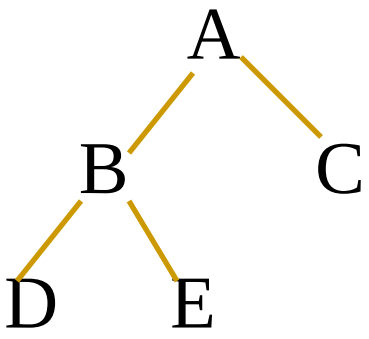
5.5 遍历二叉树和线索二叉树

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





先序遍历：**A B D E C**

中序遍历：**D B E A C**

后序遍历：**D E B C A**

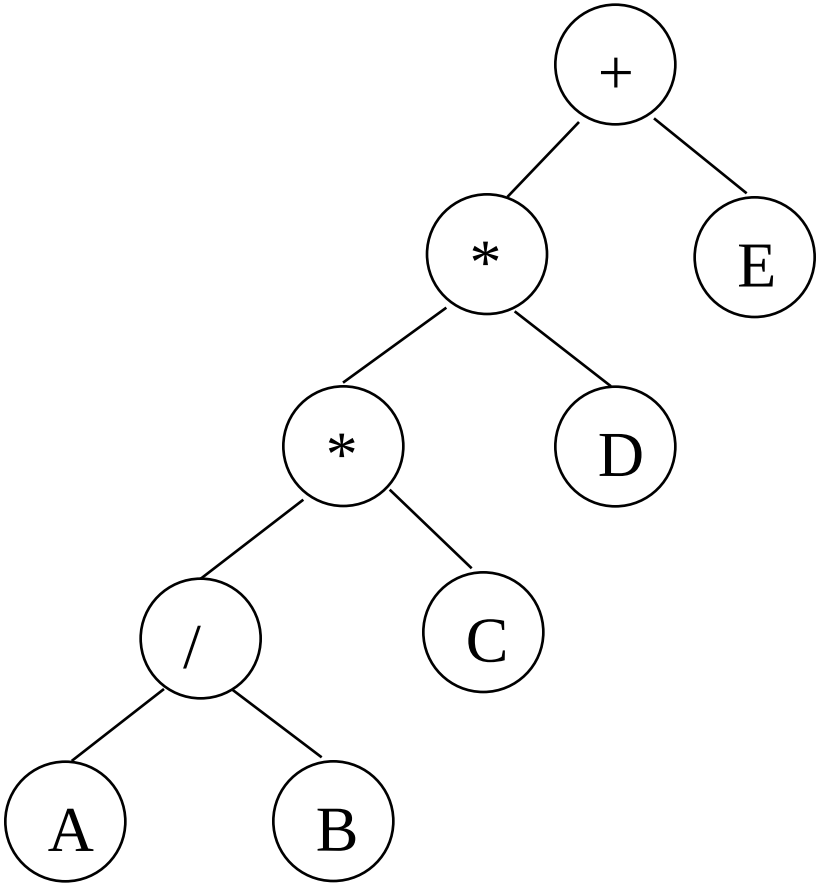
口诀：

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

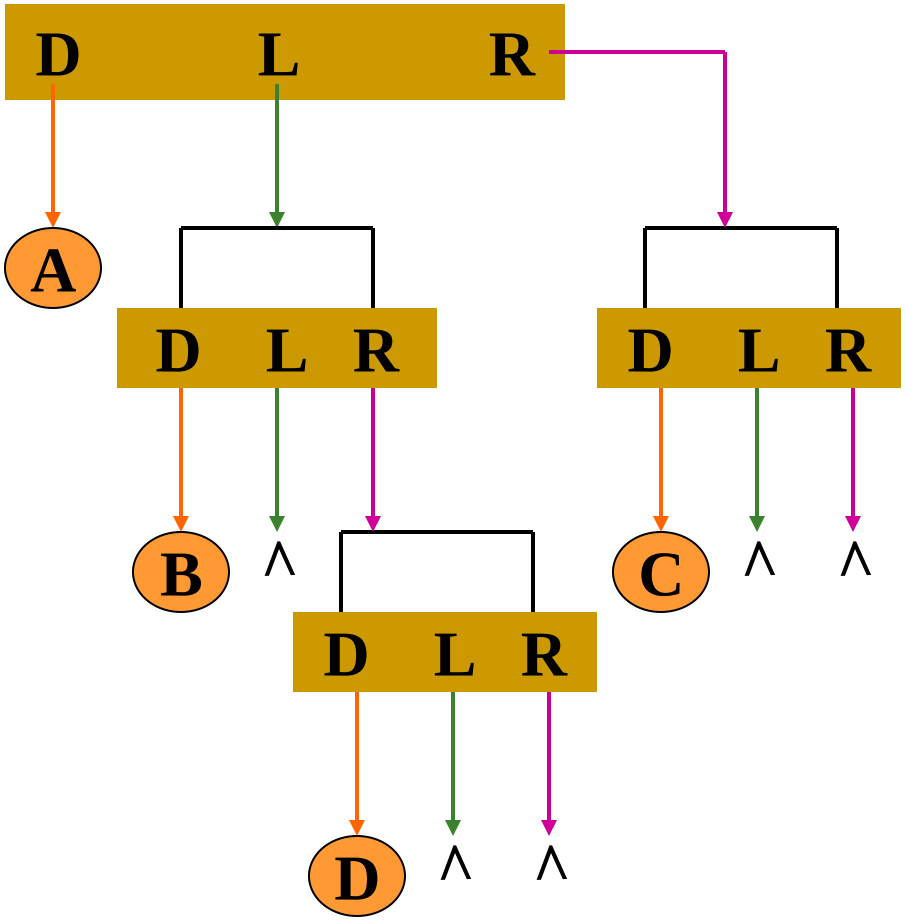
若二叉树为空，则空操作

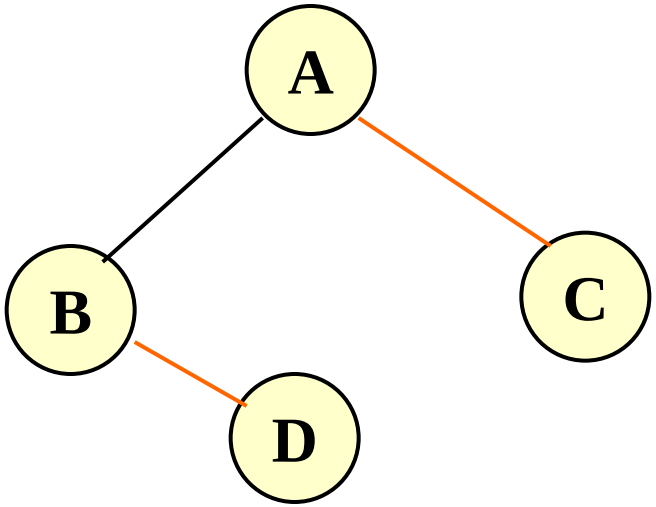
否则

访问根结点 (D)

前序遍历左子树 (L)

前序遍历右子树 (R)





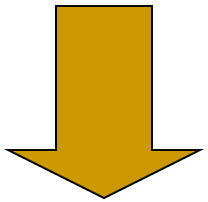
先序遍历序列：A B D C

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：

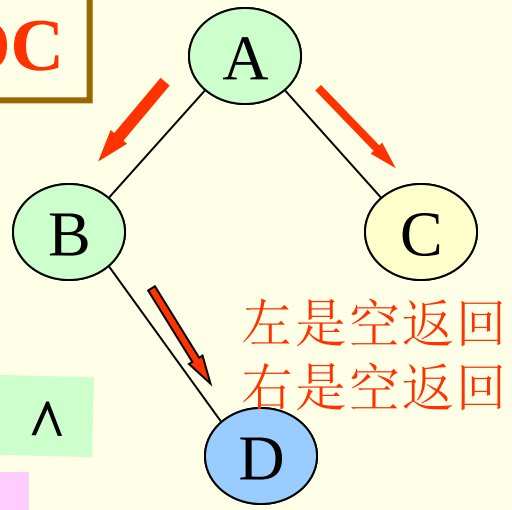


先序遍历算法

```
Status PreOrderTraverse(BiTree T){
  if(T==NULL) return OK; //空二叉树
  else{
     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
    }
}
```

```
Status PreOrderTraverse(BiTree T){
  if(T==NULL) return OK; else{
    cout<<T->data;
    PreOrderTraverse(T->lchild);
    PreOrderTraverse(T->rchild); }
}
```

先序序列：**ABDC**



左是空返回

左是空返回
右是空返回

左是空返回
右是空返回

主程序

Pre(T)

T → A
printf(A);
pre(T→L);
pre(T→R);

T → B
printf(B);
pre(T→L);
pre(T→R);

T → C
printf(C);
pre(T→L);
pre(T→R);

T → D
printf(D);
pre(T→L);
pre(T→R);

T → ∧
返回

T → ∧
返回

T → ∧
返回

T → ∧
返回

T → ∧
返回

遍历的算法实现—中序遍历

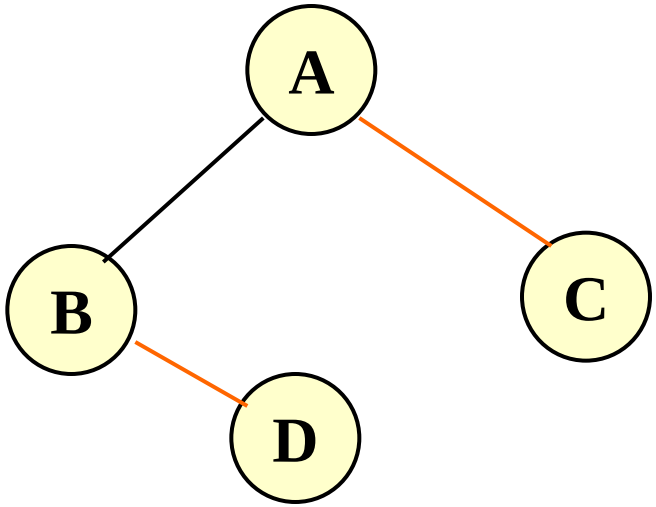
若二叉树为空，则空操作

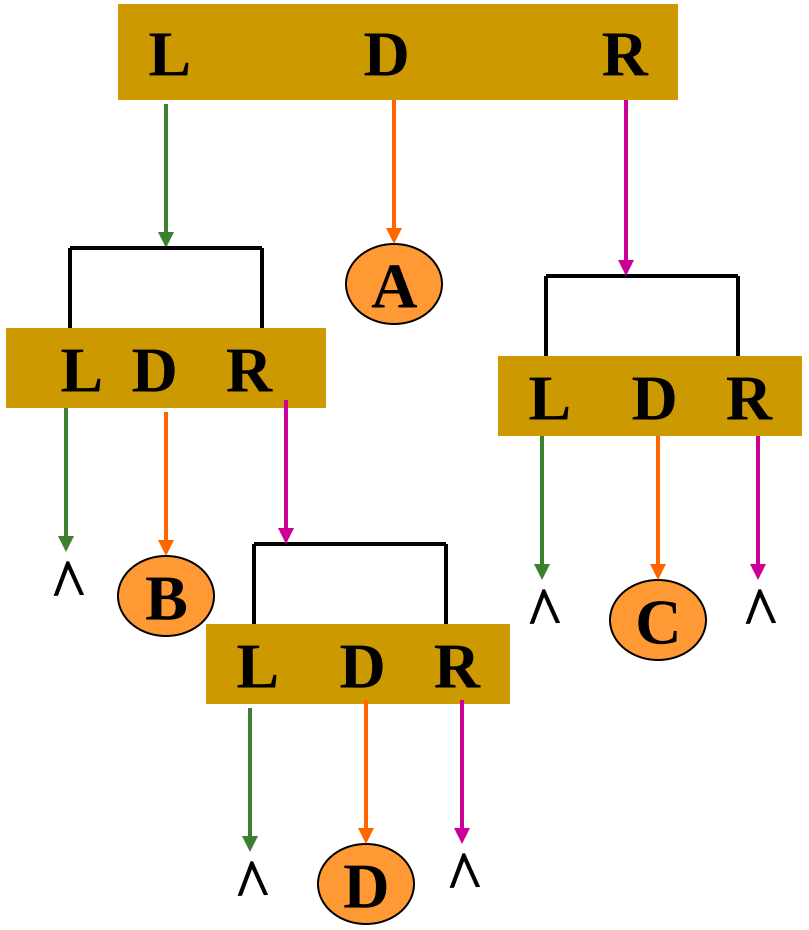
否则：

中序遍历左子树（L）

访问根结点（D）

中序遍历右子树（R）





中序遍历序列：B D A C

中序遍历算法

```
Status InOrderTraverse(BiTree T){
  if(T==NULL) return OK; //空二叉树
  else{
     InOrderTraverse(T->lchild); //递归遍历左子树
    cout<<T->data; //访问根结点
     InOrderTraverse(T->rchild); //递归遍历右子树
    }
}
```

遍历的算法实现—后序遍历

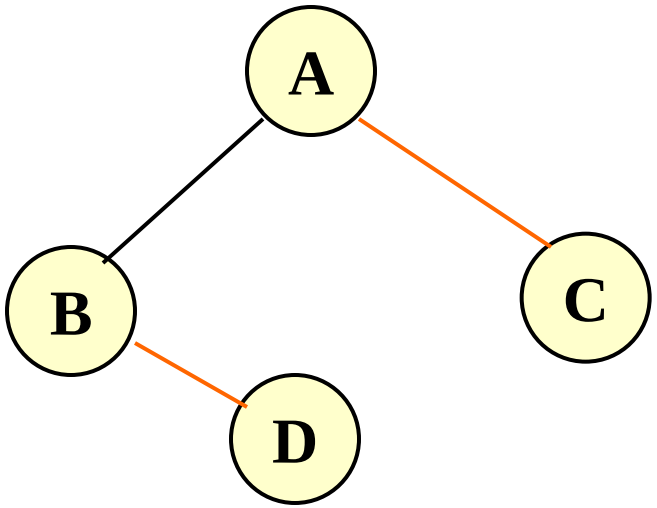
若二叉树为空，则空操作

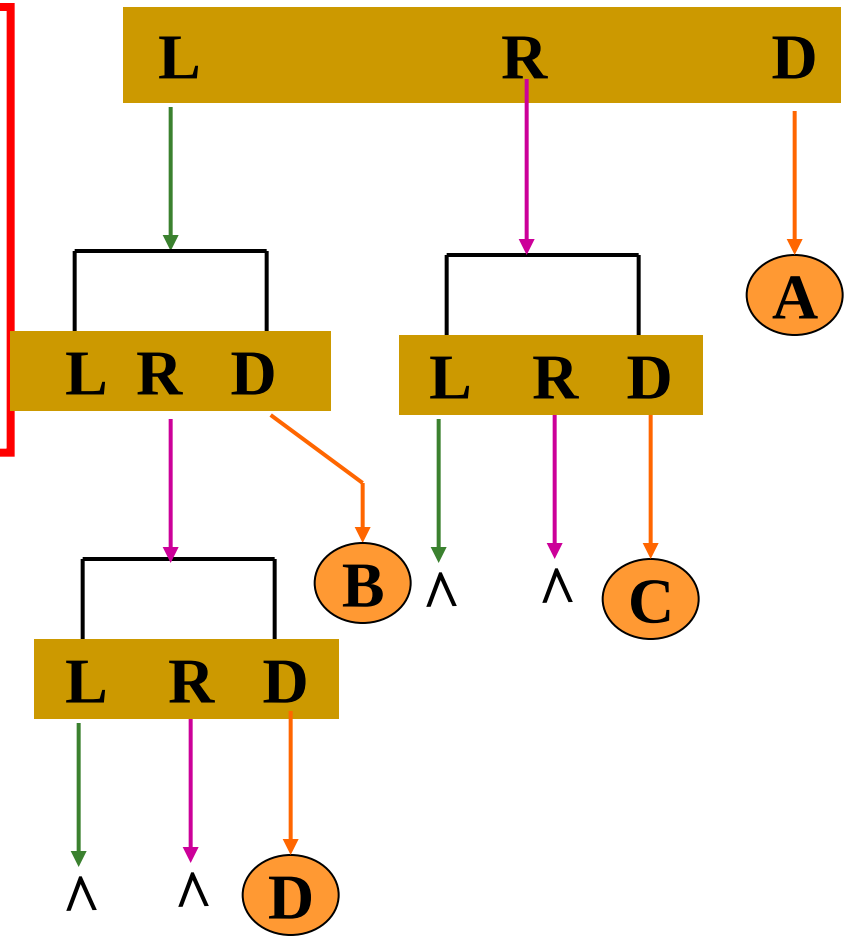
否则

后序遍历左子树 (L)

后序遍历右子树 (R)

访问根结点 (D)





后序遍历序列： D B C A

后序遍历算法

```
Status PostOrderTraverse(BiTree T){
  if(T==NULL) return OK; //空二叉树
  else{
     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
    }
}
```

遍历算法的分析

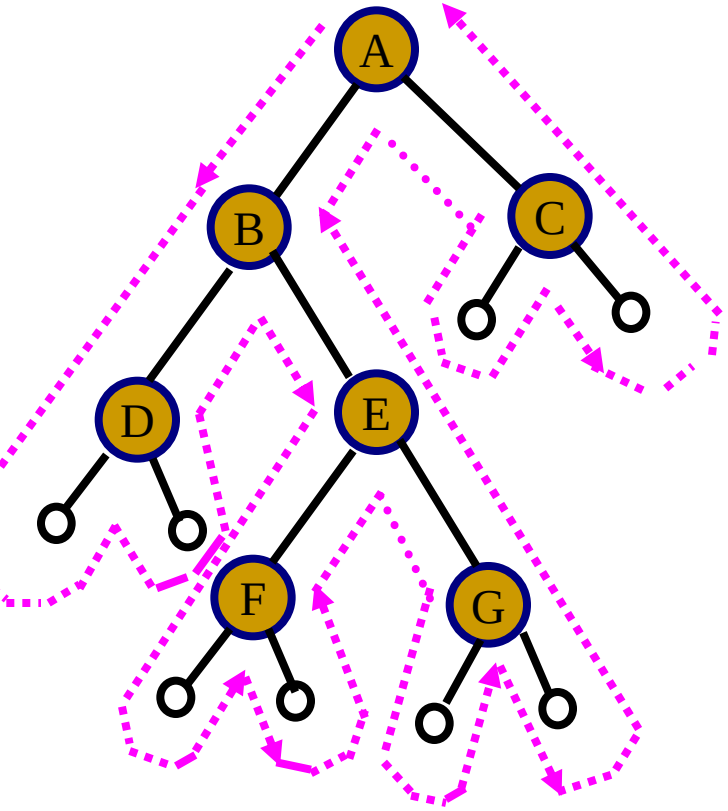
```
Status PreOrderTraverse(BiTree T){
  if(T==NULL) return OK;
  else{
     cout<<T->data;
     PreOrderTraverse(T->lchild);
     PreOrderTraverse(T->rchild);
    }
}
```

```
Status InOrderTraverse(BiTree T){
  if(T==NULL) return OK;
  else{
     InOrderTraverse(T->lchild);
     cout<<T->data;
     InOrderTraverse(T->rchild);
    }
}
```

```
Status PostOrderTraverse(BiTree T){
  if(T==NULL) return OK;
  else{
     PostOrderTraverse(T->lchild);
     PostOrderTraverse(T->rchild);
     cout<<T->data;
    }
}
```

遍历算法的分析

如果去掉输出语句，从递归的角度看，三种算法是完全相同的，或说这三种算法的访问路径是相同的，只是访问结点的时机不同。



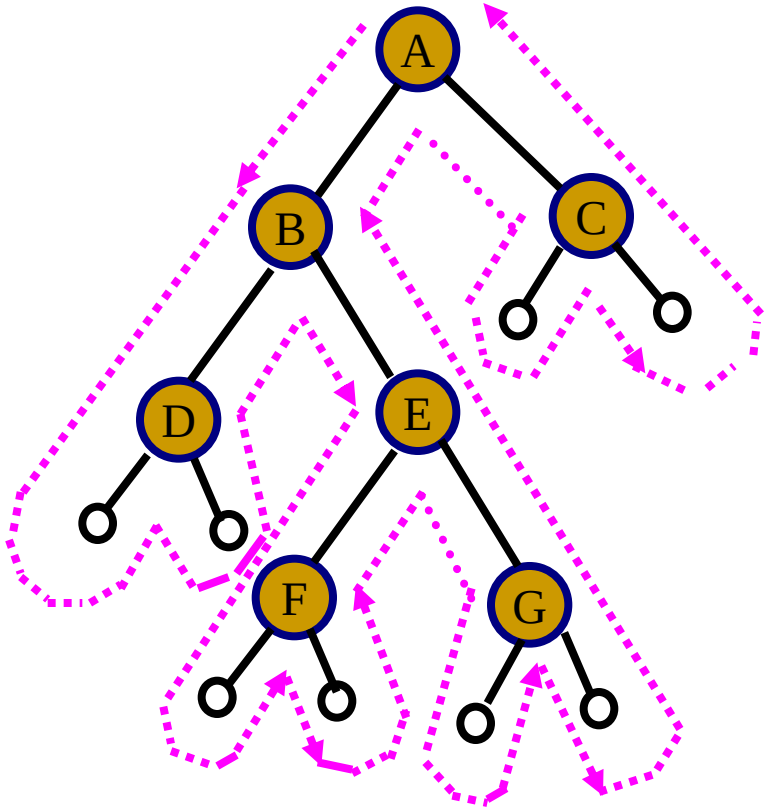
从虚线的出发点到终点的路径上，每个结点经过3次。

第1次经过时访问＝先序遍历
第2次经过时访问＝中序遍历
第3次经过时访问＝后序遍历

遍历算法的分析

时间效率:O(n) //对n个结点进行访问

空间效率:O(n) //栈占用的最大辅助空间

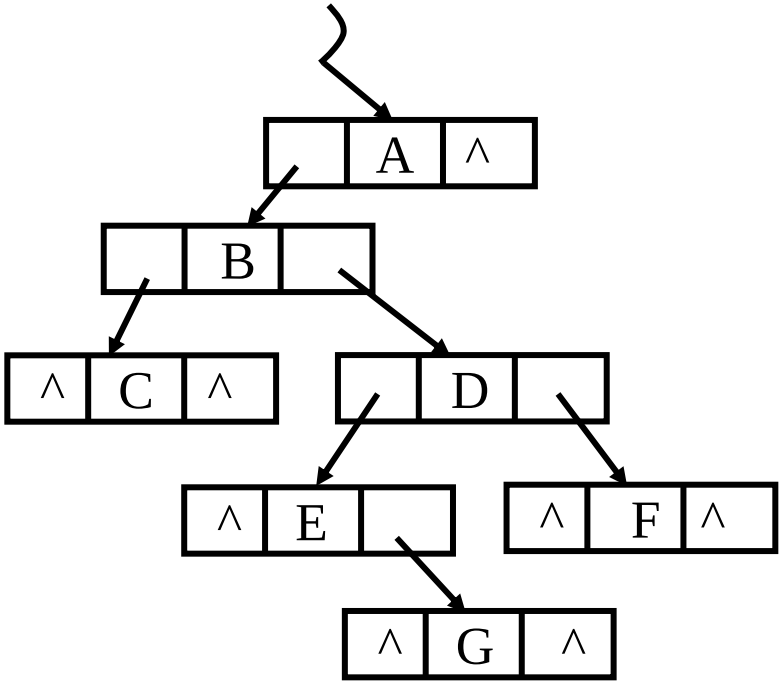


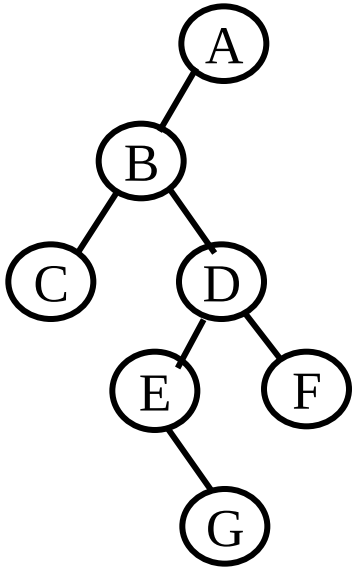
二叉树的建立（算法5.3）

按先序遍历序列建立二叉树的二叉链表

例：已知先序序列为：

A B C Φ Φ D E Φ G Φ Φ F Φ Φ Φ （ Φ 表示空格字符）





二叉树的建立（算法5.3）

```
void CreateBiTree(BiTree &T) {
cin>>ch;
if (ch==' ')    T=NULL;     //递归结束
else{
    T=new BiTNode;     T->data=ch;
     //生成根结点
    CreateBiTree(T->lchild);  //递归创建左子树
    CreateBiTree(T->rchild); //递归创建右子树
  }

}
```

二叉树遍历算法的应用

✓计算二叉树结点总数

- 如果是空树，则结点个数为0；
- 否则，结点个数为左子树的结点个数+右子树的结点个数再+1。

算法5.6

```
int NodeCount(BiTree T){
   if(T == NULL ) return 0;
   else return NodeCount(T-
   >lchild)+NodeCount(T->rchild)+1;
}
```

二叉树遍历算法的应用

✓计算二叉树叶子结点总数

- 如果是空树，则叶子结点个数为0；
- 如果是叶子结点返回1；
- 否则，为左子树的叶子结点个数+右子树的叶子结点个数。

```
int LeadCount(BiTree T){
   if(T==NULL)          //如果是空树返回0
        return 0;
   if (T->lchild == NULL && T->rchild == NULL)
        return 1; //如果是叶子结点返回1
   else return LeafCount(T->lchild) + LeafCount(T->rchild);
```

二叉树遍历算法的应用

✓计算二叉树深度

- 如果是空树，则深度为0；
- 否则，递归计算左子树的深度记为m，递归计算右子树的深度记为n，二叉树的深度则为m与n的较大者加1。

重要结论

若二叉树中各结点的值均不相同，则：
由二叉树的前序序列和中序序列，或由其后序序列和中序序列均能唯一地确定一棵二叉树，
但由前序序列和后序序列却不一定能唯一地确定一棵二叉树。

练习

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。

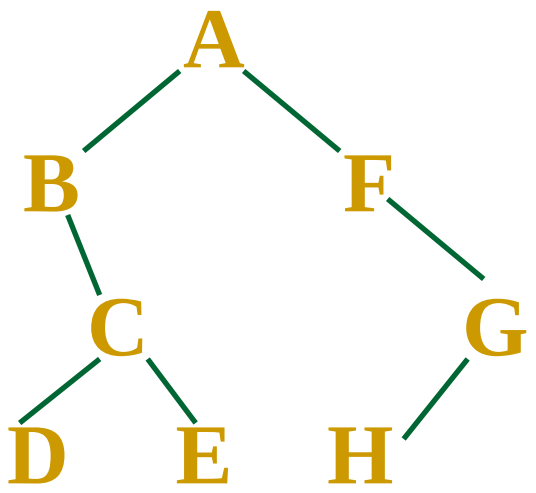
①由后序遍历特征，根结点必在后序序列尾部（A）；

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子孙（BDCE），其右部必全部是右子树子孙（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：B D C E A F H G

后序遍历：D E C B H G F A



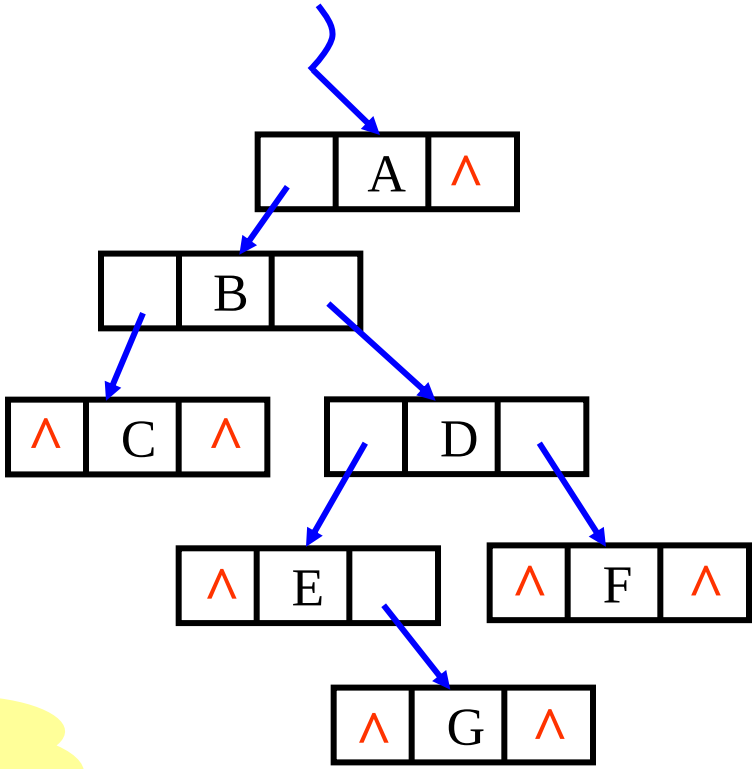
（D C E）　　（H G）

思考

在n个结点的二叉链表中，有n+1个空指针域

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？

——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。



线索化二叉树

线索化二叉树

普通二叉树只能找到结点的左右孩子信息，而该结点的直接前驱和直接后继只能在遍历过程中获得

若将遍历后对应的有关前驱和后继预存起来，则从第一个结点开始就能很快“顺藤摸瓜”而遍历整个树

可能是根、或最左（右）叶子

例如中序遍历结果：B D C E A F H G，实际上已将二叉树转为线性排列，显然具有唯一前驱和唯一后继！

线索化二叉树

如何保存这类信息？

两种解决方法：

- 增加两个域：fwd和bwd；
- 利用空链域（n+1个空链域）

线索化二叉树

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)** 。

为了避免混淆，增加两个标志域

lchild	LTag	data	RTag	rchild

线索化二叉树

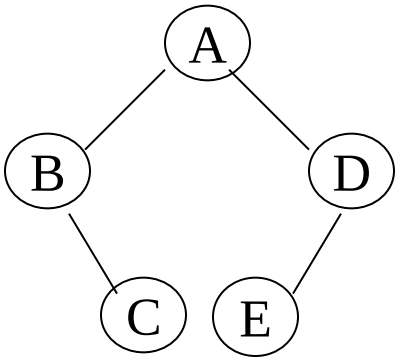
LTag :若 LTag=0, lchild域指向左孩子;
若 LTag=1, lchild域指向其前驱。

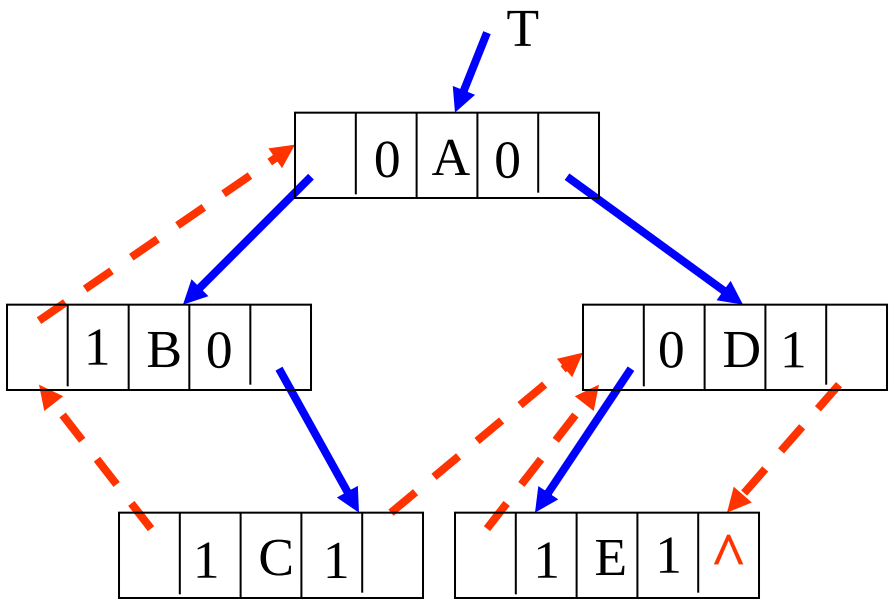
RTag :若 RTag=0, rchild域指向右孩子;
若 RTag=1, rchild域指向其后继。

先序线索二叉树

LTag=0, lchild域指向左孩子
LTag=1, lchild域指向其前驱
RTag=0, rchild域指向右孩子
RTag=1, rchild域指向其后继

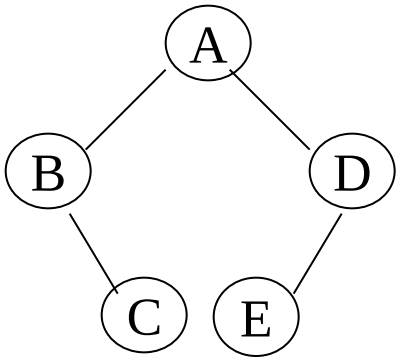
lchild	LTag	data	RTag	rchild

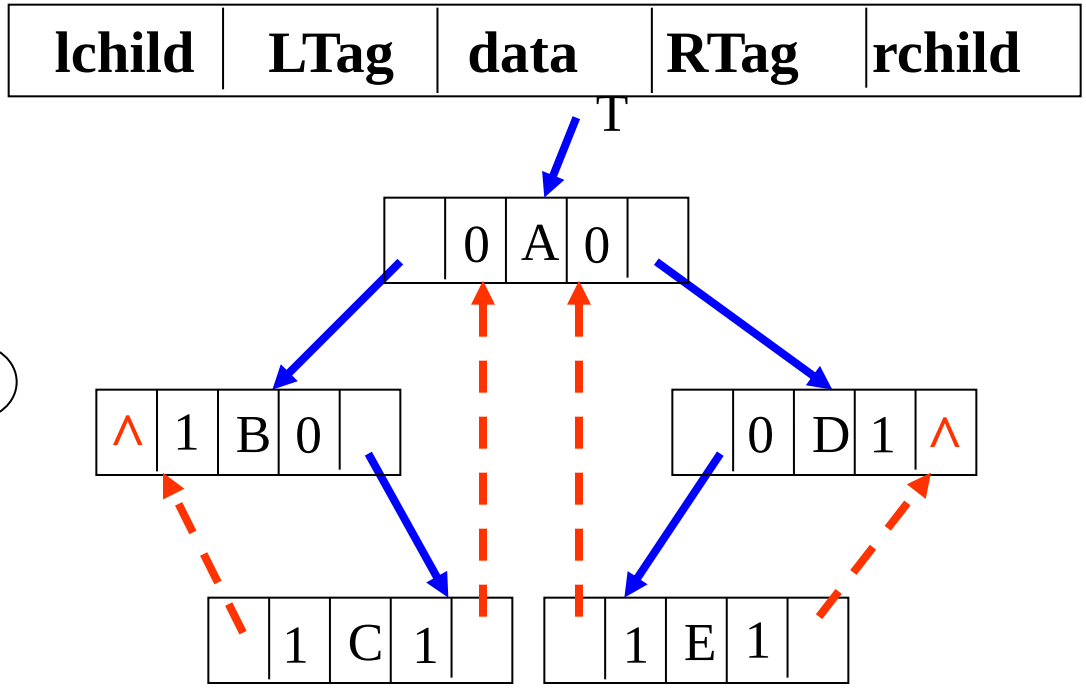




先序序列：ABCDE

中序线索二叉树

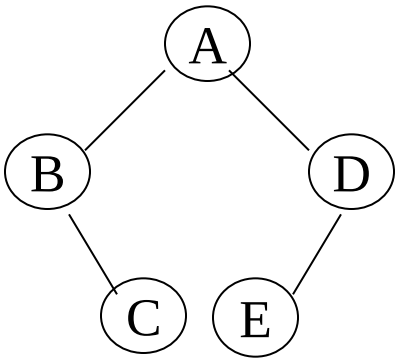


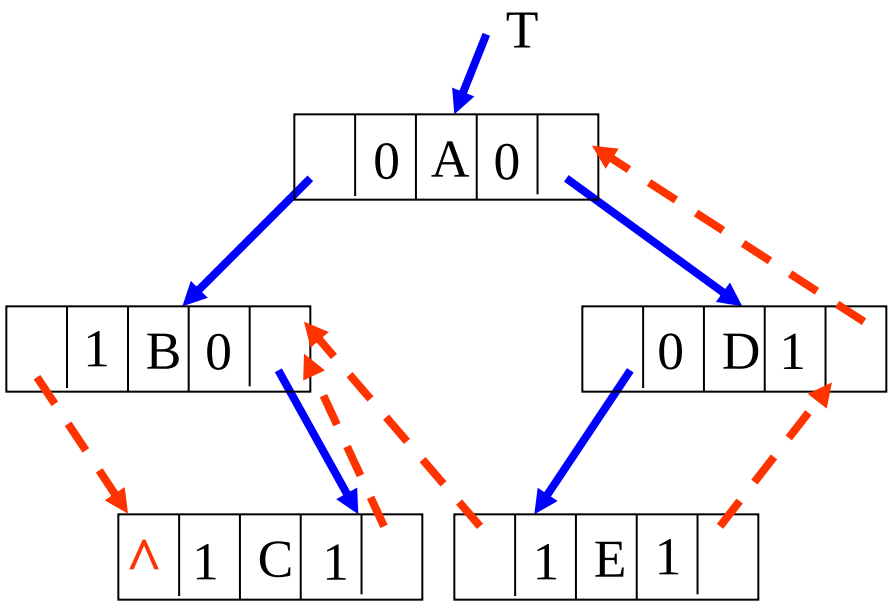


中序序列：BCAED

后序线索二叉树

lchild	LTag	data	RTag	rchild





后序序列：CBEDA

线索化二叉树的几个术语

线索：指向结点前驱和后继的指针

线索链表：加上线索二叉链表

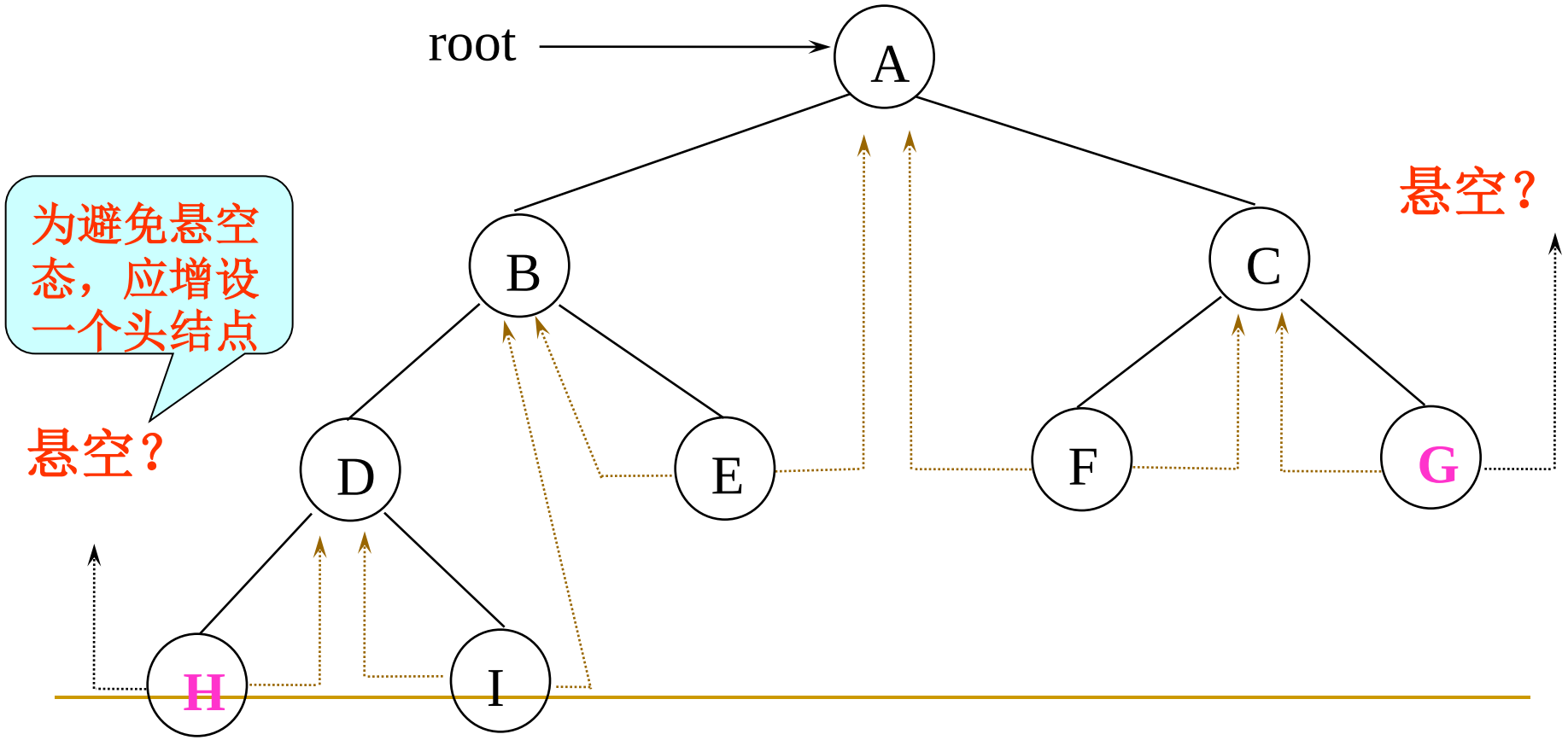
线索二叉树：加上线索的二叉树（图形式样）

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

练习

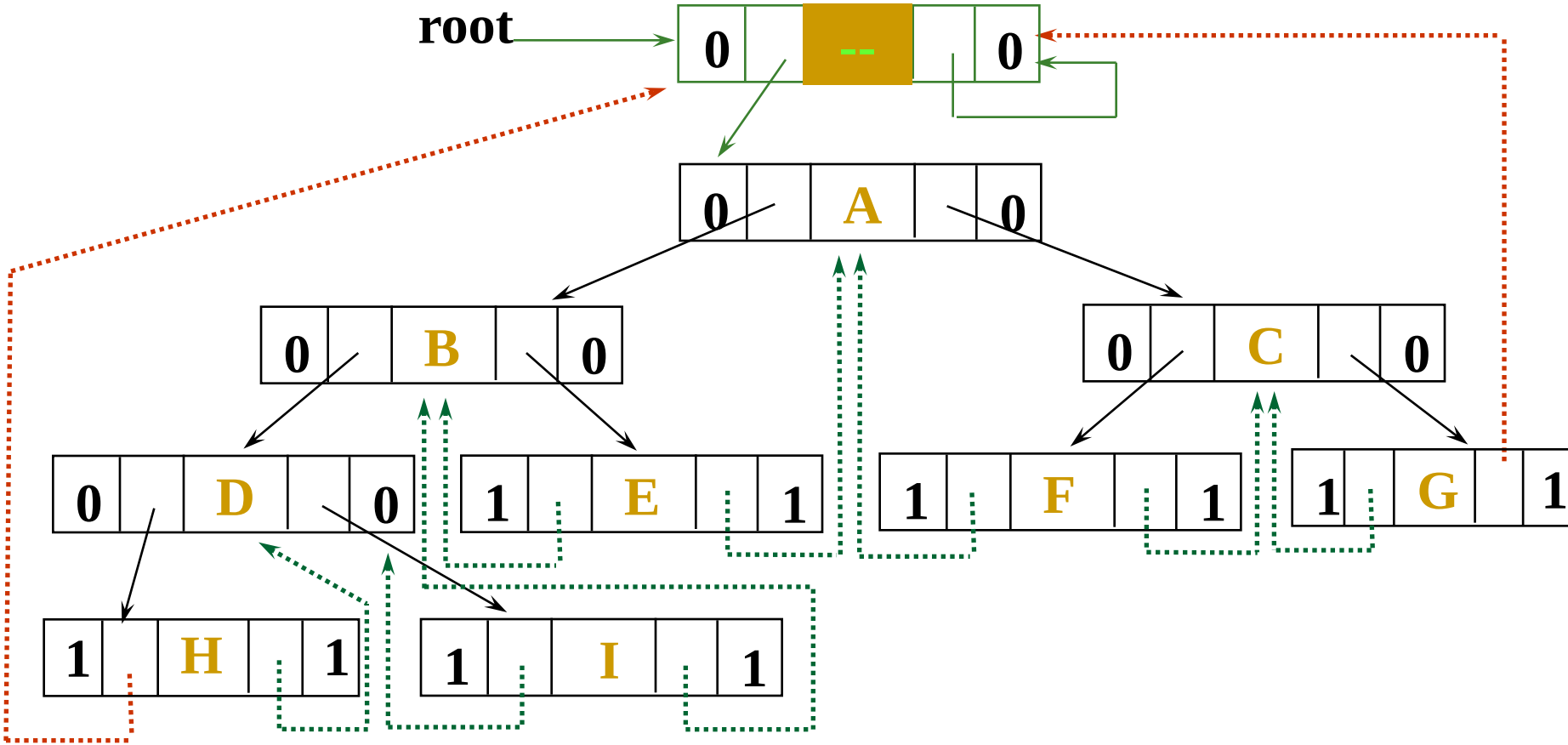
画出以下二叉树对应的中序线索二叉树。

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



对应的中序线索二叉树存储结构如图所示：

注：此图中序遍历结果为： H, D, I, B, E, A, F, C, G

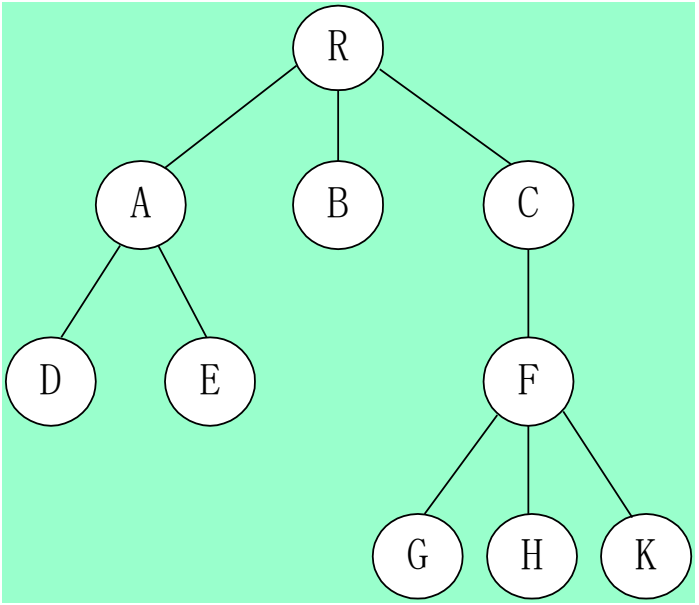


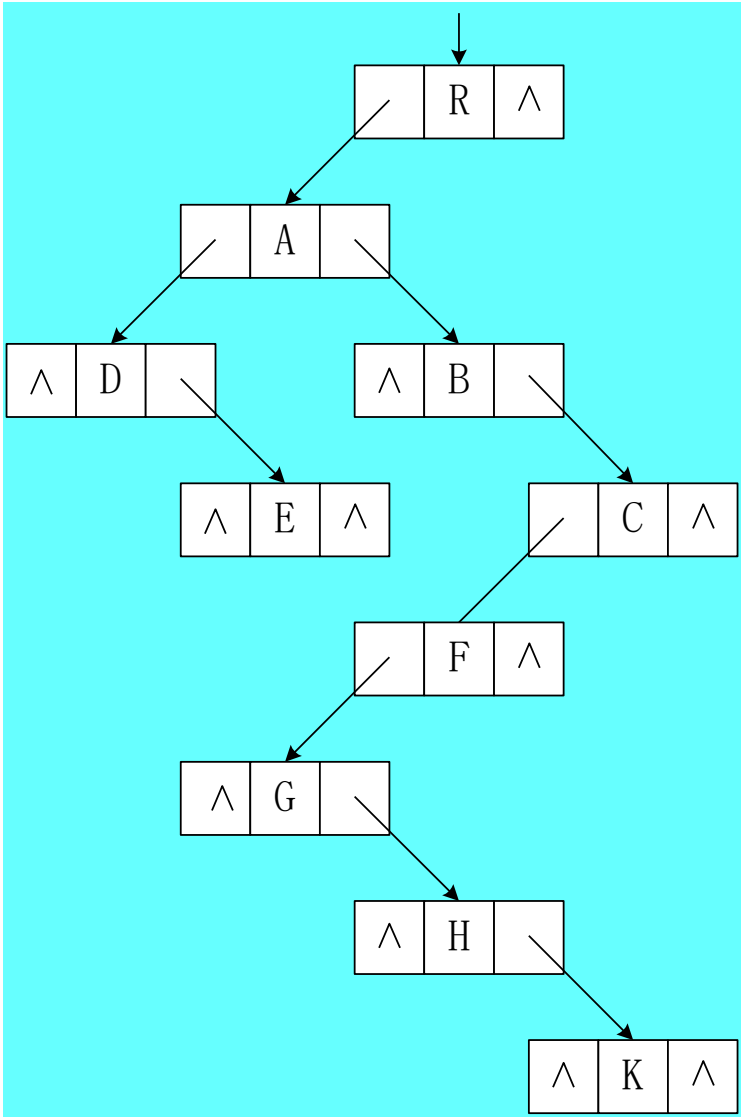
5.6 树和森林

树的存储结构——二叉链表表示法

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

树的存储结构——二叉链表表示法

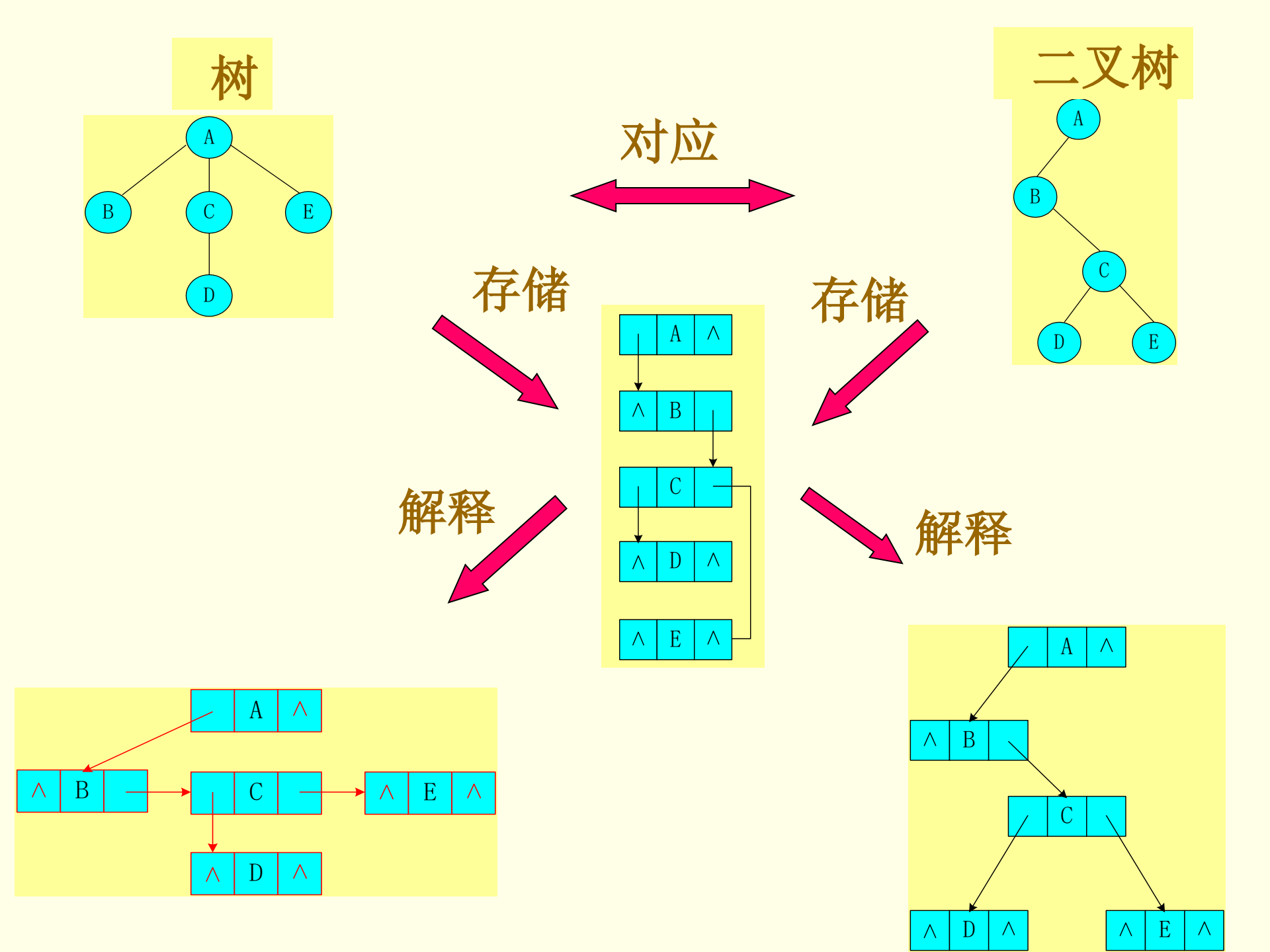




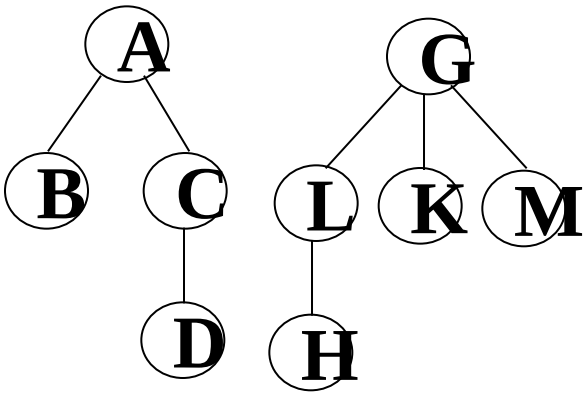
普通树（多叉树）若不转化为二叉树，则运算很难实现

为何要重点研究每结点最多只有两个 “叉” 的树？

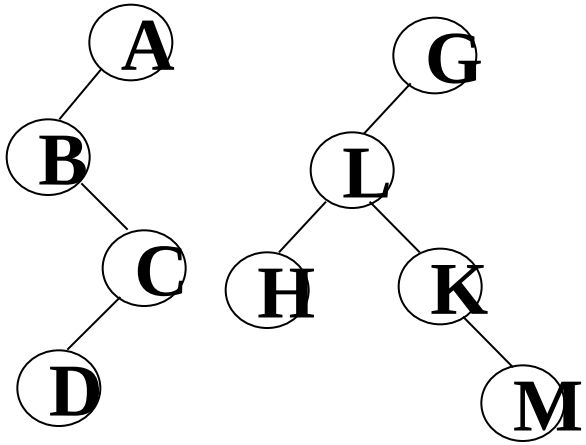
- ✓ 二叉树的结构最简单，规律性最强；
- ✓ 可以证明，所有树都能转为唯一对应的二叉树，不失一般性。


树
A
B
C
D
E
对应
二叉树
A
B
C
D
E
存储
存储
解释
解释
A ^
^ B
C
^ D ^
^ E ^
A ^
^ B
C
^ E ^
^ D ^
A ^
^ B
C
^ D ^
^ E ^

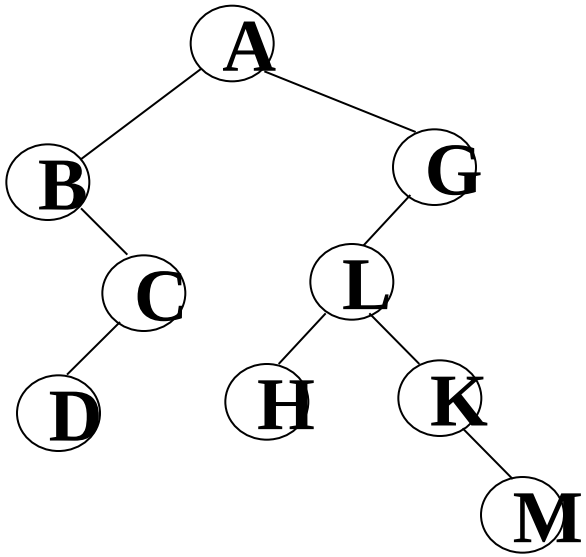
森林与二叉树的对应关系



(a) 森林



(b) 森林中每棵树对应的二叉树



(c) 森林对应的二叉树

5.7 哈夫曼树及其应用

哈夫曼树的相关概念

路　　径：由一结点到另一结点间的分支所构成

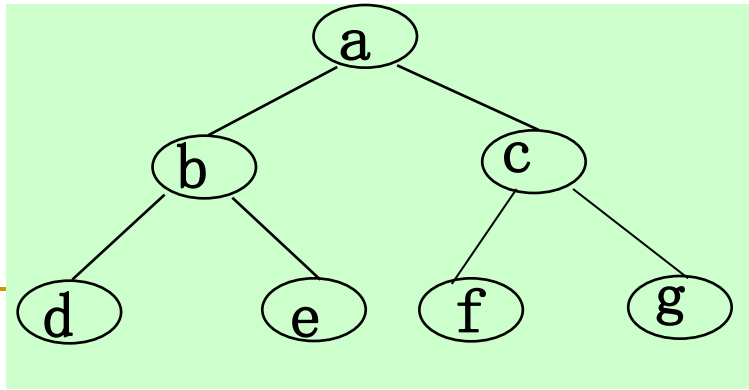
路径长度：路径上的分支数目　a→e的路径长度＝2

带权路径长度：结点到根的路径长度与结点上权的乘积

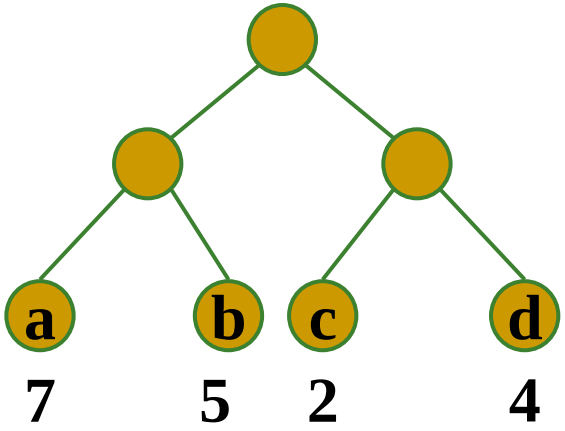
树的带权路径长度：树中所有叶子结点的带权路径长度之和

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

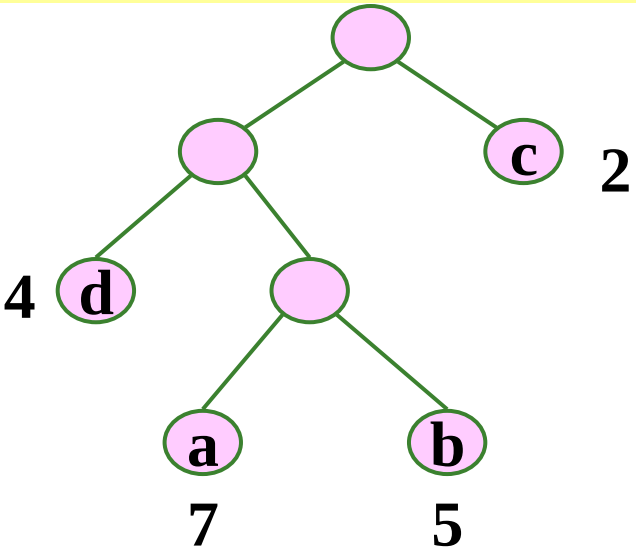
哈 夫 曼 树：带权路径长度最小的树



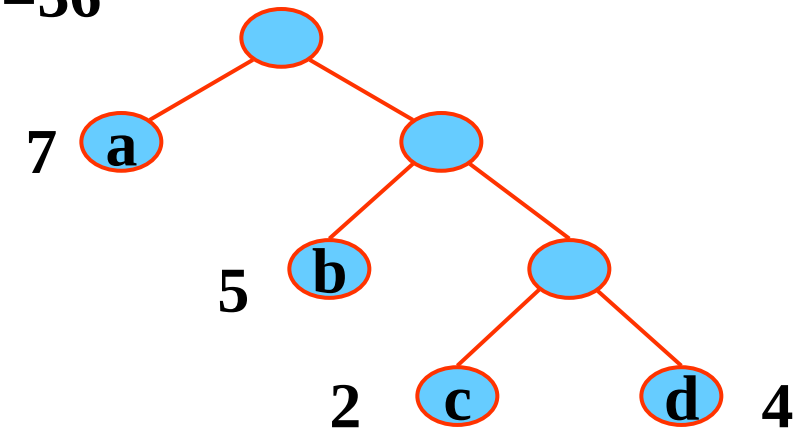
权值分别为7，5，2，4，构造有4个叶子结点的二叉树



WPL=7*2+5*2+2*2+4*2=36

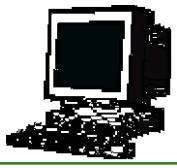


WPL=7*3+5*3+2*1+4*2=46



WPL=7*1+5*2+2*3+4*3=35

5.7 哈夫曼树及其应用



- 游戏中主角的生命值d，有这样的条件判定：当怪物碰到主角后，怪物的反应遵从下规则：

条件：	d<100	100<=d<200	200<=d<300	300<=d<500	d>500
反应：	嘲笑，单挑	单挑	嗜血魔法	呼唤同伴	逃跑

条件：	d<100	100<=d<200	200<=d<300	300<=d<500	d>500
反应：	嘲笑，单挑	单挑	嗜血魔法	呼唤同伴	逃跑

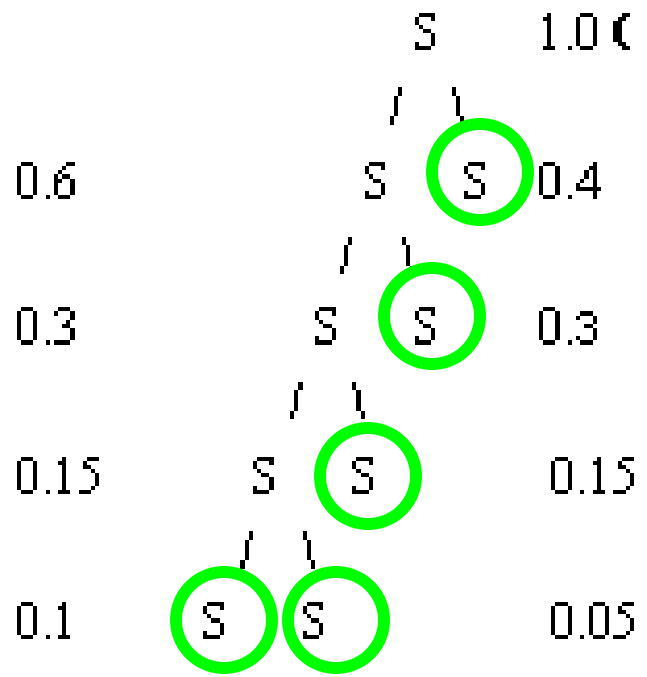
```
if(d<100) state=嘲笑，单挑;
    else if(d<200) state=单挑;
        else if(d<300) state=嗜血魔法;
            else if(d<500) state=呼唤同伴;
                else state=逃跑;
```

- 分析主角生命值d的特点，即预测出每种条件占总条件的百分比，将这些比值作为权值来构造最优二叉树（哈夫曼树），作为判定树来设定算法。

提高效率

条件：	d<100	100<=d<200	200<=d<300	300<=d<500	d>500
比例：	5%	15%	40%	30%	10%

条件：	d<100	100<=d<200	200<=d<300	300<=d<500	d>500
比例：	5%	15%	40%	30%	10%




```
if(d>=200)&&(d<300) state=嗜血魔法;
  else if(d>=300)&&(d<500) state=呼唤同伴;
    else if(d>=100)&&(d<200) state=单挑;
      else if(d<100) state=嘲笑，单挑;
        else state=逃跑;
```

```
if(d<100) state=嘲笑，单挑;
  else if(d<200) state=单挑;
    else if(d<300) state=嗜血魔法;
      else if(d<500) state=呼唤同伴;
        else state=逃跑;
```

哈夫曼树应用实例——哈夫曼编码

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

ABACCDA

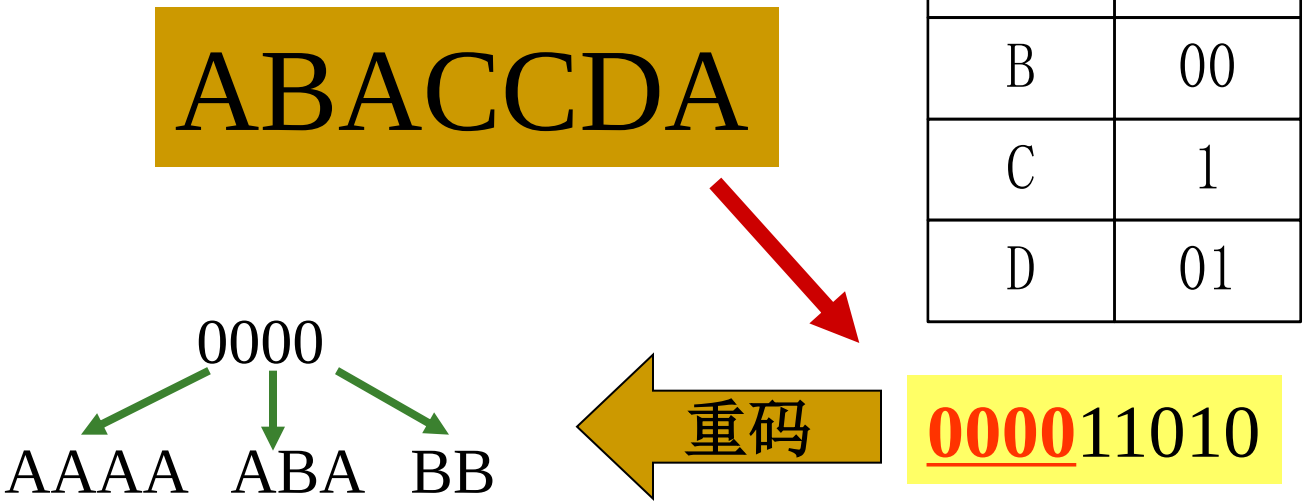
A	0
B	00
C	1
D	01

000110010101100

000011010

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



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

ABACCDA

A—0

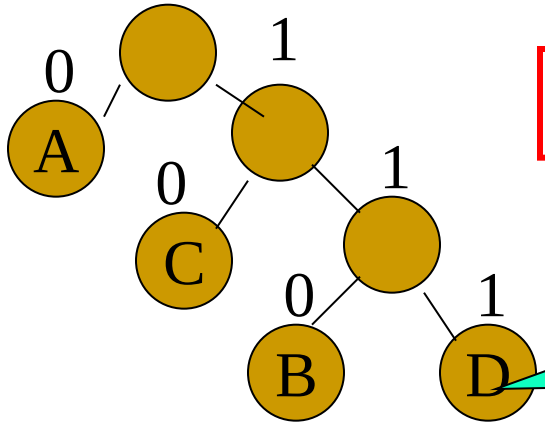
B—110

C—10

D—111

0110010101110

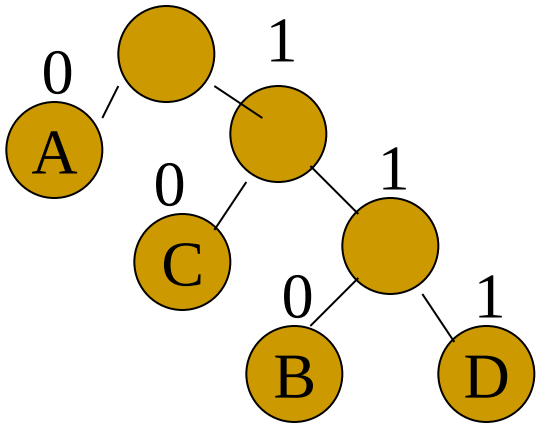
采用二叉树设计前缀编码



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。



0110010101110

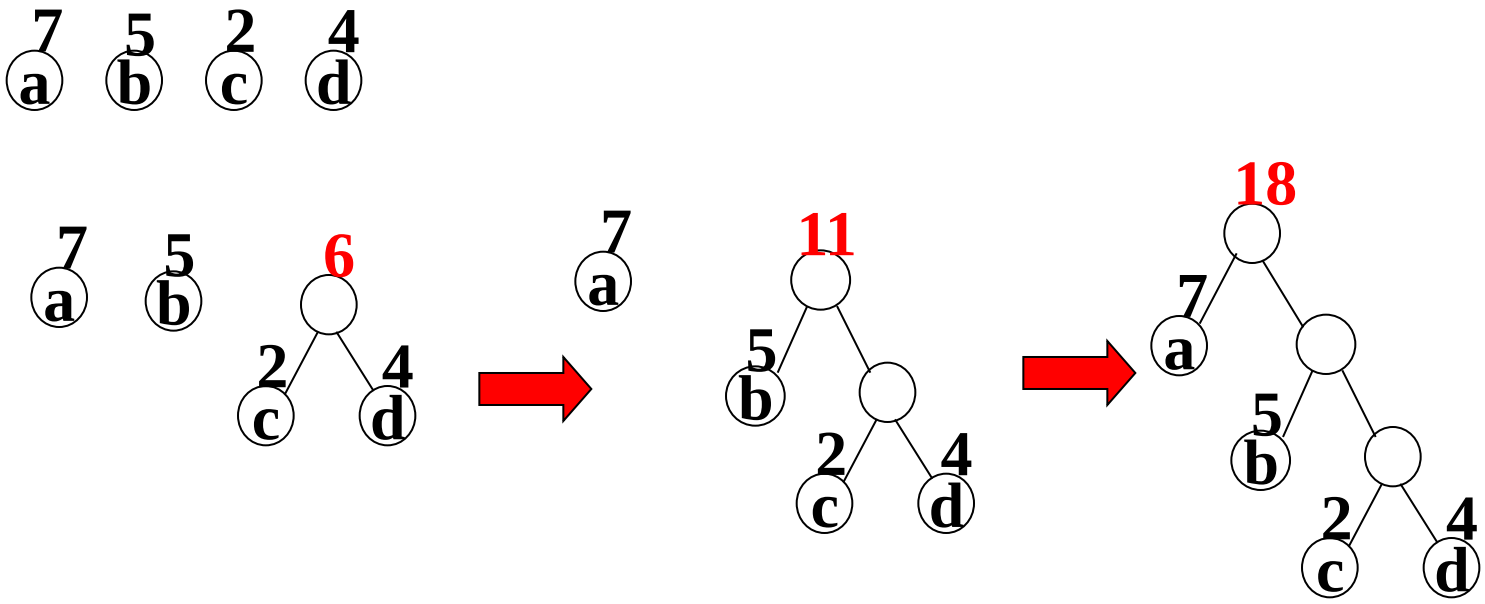
ABACCDA

特点：每一码都不是另一码的前缀，绝不会错译！称为前缀码

哈夫曼树的构造过程

基本思想：使权大的结点靠近根

操作要点：对权值的合并、删除与替换，总是合并当前值最小的两个

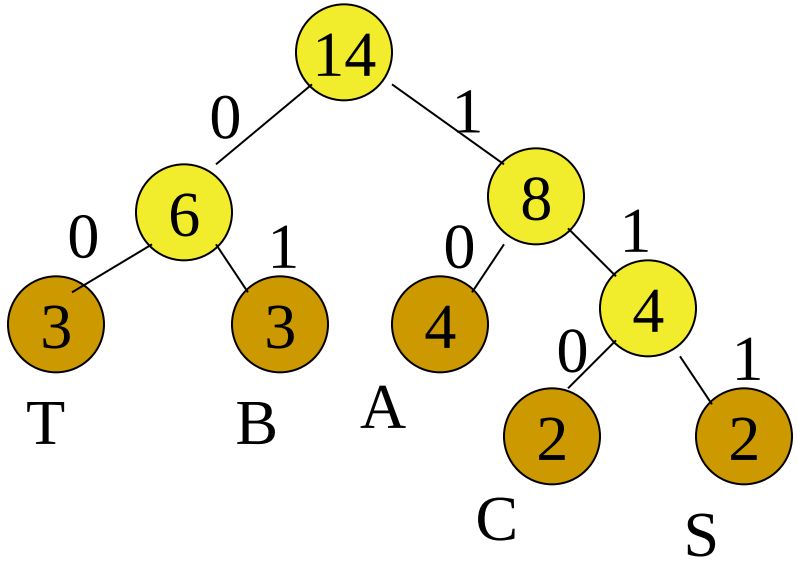


哈夫曼编码的构造

基本思想：概率大的字符用短码，小的用长码，构造哈夫曼树

例：某系统在通讯时，只出现C，A，S，T，B五种字符，其出现频率依次为2，4，2，3，3，试设计Huffman编码。

T	**00**
B	**01**
A	**10**
C	**110**
S	**111**



哈夫曼树的构造过程

✓根据给定的n个权值$\{w_1, w_2, \ldots\ldots w_n\}$，构造n棵只有根结点的二叉树。

✓在森林中选取两棵根结点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。

✓在森林中删除这两棵树，同时将新得到的二叉树加入森林中。

✓重复上述两步，直到只含一棵树为止，这棵树即哈夫曼树。

哈夫曼树构造算法的实现（算法5.10）

一棵有n个叶子结点的Huffman树有 **2n-1** 个结点

- ✓ 采用顺序存储结构——一维结构数组
- ✓ 结点类型定义

```
typedef  struct
{  int weght;
   int parent,lch,rch;
}*HuffmanTree;
```

哈夫曼树构造算法的实现

1)初始化HT[1..2n-1]：lch=rch=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：

3.1)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)

3.2)修改HT[s1]和HT[s2]的parent值： parent=i

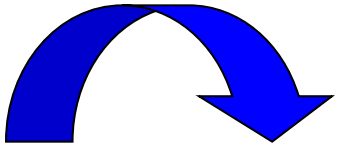
3.3)置HT[i]：weight=HT[s1].weight + HT[s2].weight ，

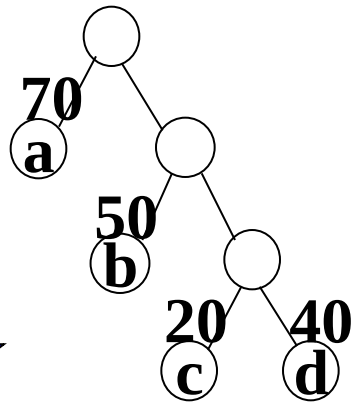
lch=s1, rch=s2

例:设n=4,　w={70, 50, 20, 40}

试设计 huffman code (m=2*4-1=7)

	weight	parent	lch	rch
1	70	0	0	0
2	50	0	0	0
3	20	0	0	0
4	40	0	0	0
5				
6				
7				





	weight	parent	lch	rch
1	70	7	0	0
2	50	6	0	0
3	20	5	0	0
4	**40**	5	**0**	**0**
5	**60**	**6**	**3**	**4**
6	**110**	**7**	**2**	**5**
7	**180**	**0**	**1**	**6**

算法

```
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].lch=0;HT[i].rch=0;
   HT[i].parent=0;}
for(i=1;i<=n;++i)
cin>>HT[i].weight;
```

例:设n=8，w={5,29,7,8,14,23,3,11}

试设计 huffman code (m=2*8-1=15)

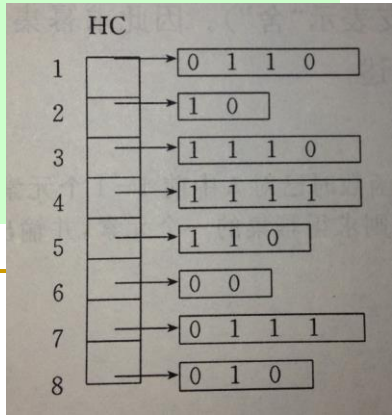
	weight	parent	lch	rch
1 . . . 8	5 29 7 8 14 23 3 11	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0
9 . . 15		0 0 0 0 0 0 0	0 0 0 0 0 0 0	0 0 0 0 0 0 0

```
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].lch=s1;    HT[i].rch=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
    }
}
```

构造Huffman tree后,HT为:

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

```
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
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
```



哈夫曼编码的几点结论

- 哈夫曼编码是不等长编码
- 哈夫曼编码是前缀编码，即任一字符的编码都不是另一字符编码的前缀
- 哈夫曼编码树中没有度为1的结点。若叶子结点的个数为n，则哈夫曼编码树的结点总数为 2n-1
- 发送过程：根据由哈夫曼树得到的编码表送出字符数据
- 接收过程：按左0、右1的规定，从根结点走到一个叶结点，完成一个字符的译码。反复此过程，直到接收数据结束

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。

表达式树的创建---【算法步骤】

① 初始化OPTR栈和EXPT栈，将表达式起始符“#”压入OPTR栈。

② 扫描表达式，读入第一个字符ch，如果表达式没有扫描完毕至“#”或OPTR的栈顶元素不为“#”时，则循环执行以下操作：

- 若ch不是运算符，则以ch为根创建一棵只有根结点的二叉树，且将该树根结点压入EXPT栈，读入下一字符ch；
- 若ch是运算符，则根据OPTR的栈顶元素和ch的优先级比较结果，做不同的处理：
  - 若是小于，则ch压入OPTR栈，读入下一字符ch；
  - 若是大于，则弹出OPTR栈顶的运算符，从EXPT栈弹出两个表达式子树的根结点，以该运算符为根结点，以EXPT栈中弹出的第二个子树作为左子树，以EXPT栈中弹出的第一个子树作为右子树，创建一棵新二叉树，并将该树根结点压入EXPT栈；
  - 若是等于，则OPTR的栈顶元素是“(”且ch是“)”，这时弹出OPTR栈顶的“(”，相当于括号匹配成功，然后读入下一字符ch。

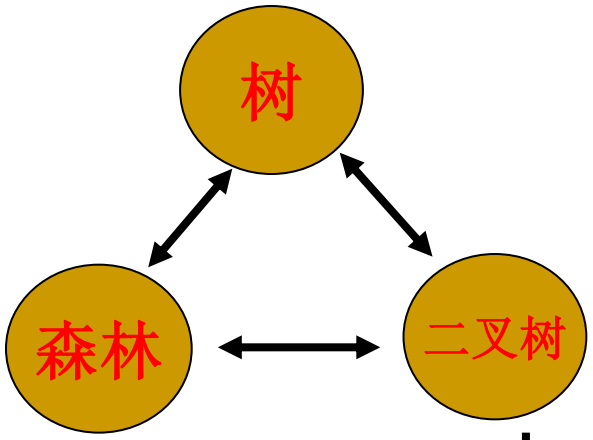
表达式树的求值---【算法步骤】

① 设变量lvalue和rvalue分别用以记录表达式树中左子树和右子树的值，初始均为0。

② 如果当前结点为叶子（结点为操作数），则返回该结点的数值，否则（结点为运算符）执行以下操作：

- 递归计算左子树的值记为lvalue;
- 递归计算右子树的值记为rvalue;
- 根据当前结点运算符的类型，将lvalue和rvalue进行相应运算并返回。

小结



1、定义和性质

2、存储结构
- 顺序结构
- 链式结构
  - 二叉链表
  - 三叉链表

3、遍历
- 先序遍历
- 中序遍历
- 后序遍历

4、线索化：线索树

哈夫曼树

哈夫曼编码

小结

1. 掌握二叉树的基本概念、性质和存储结构
2. 熟练掌握二叉树的前、中、后序遍历方法
3. 了解线索化二叉树的思想
4. 熟练掌握：哈夫曼树的实现方法、构造哈夫曼编码的方法
5. 了解：森林与二叉树的转换，树的遍历方法