3.7 树型结构的应用（Cont.）

哈夫曼（Huffman）树

Huffman编码（最优编码）

问题的提出：

哈 2594
尔 2291
滨 1785
工 2504
业 5024
大 2083
学 4907

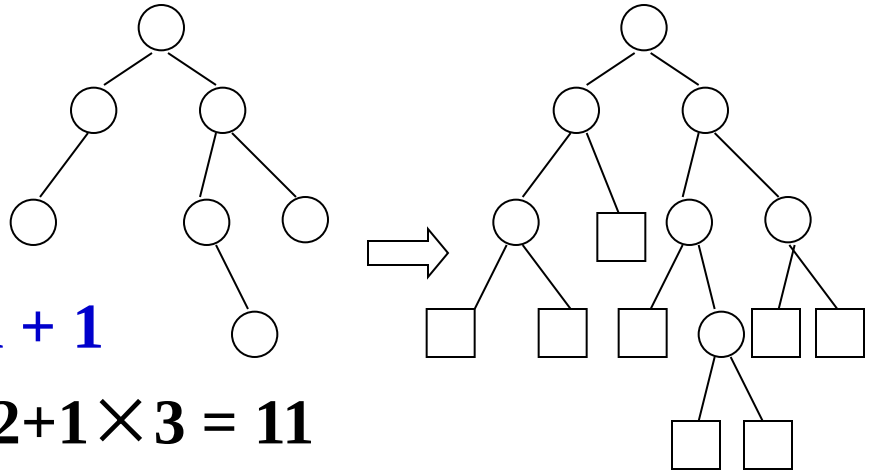
啊1601阿1602吖6325嗄6436腌7571锕7925埃1603挨1604哎1605唉1606哀1607皑1608癌1609蔼1610矮1611
6441嫒7040瑗7208暖7451碍7733镍7945霭8616鞍1616氨1617安1618俺1619按1620暗1621岸1622胺1623案
7281铵7907鹌8038黯8786肮1625昂1626盎1627凹1628敖1629熬1630翱1631袄1632傲1633奥1634懊1635澳
6959媪7033鳌7081螯7365鳌8190鳌8292鳌8643鳌8701鏖8773芭1637捌1638扒1639叭1640吧1641笆1642八
1649耙1650坝1651霸1652罢1653爸1654茇6056菝6135芭6517灞6917钯7857耙8446鲅8649魃8741白1655柏
1662摆6267呗6334斑7494斑1663班1664搬1665扳1666般1667颁1668板1669版1670扮1671拌1672伴1673瓣
7851瘢8103癍8113舨8418邦1678帮1679梆1680榜1681膀1682绑1683棒1684磅1685蚌1686镑1687傍1688谤
1693剥1694薄1701雹1702保1703堡1704饱1705宝1706抱1707报1708暴1709豹1710鲍1711爆1712葆6165孢
1713碑1714悲1715卑1716北1717辈1718背1719贝1720钡1721倍1722狈1723备1724鐾1725焙1726被1727孛
6703碚7753鹌8039褙8156鐾8645鞴8725奔1728苯1729本1730笨1731崩5946绷5948甭7458镚7928崩1732绷
7420逼1738鼻1739比1740鄙1741笔1742彼1743碧1744蓖1745蔽1746毕1747毙1748毖1749币1750庇1751痹
1758臂1759避1760陛1761匕5616俾5734萆6074荸6109薛6221吡6333哔6357狴6589庳6656愎6725滗6868潟
7815铋7873秕7985裨8152筚8357箅8375篦8387舭8416襞8437跸8547髀8734鞭1762边1763编1764贬1765扁
1772遍1773匾5650弁5945苄6048忭6677汴6774缏7134飚7614煸7652砭7730碥7760窆8125褊8159蝙8289笾
7027骠7084钑7228飘7609螵7613镖7958镳7980瘭8106裱8149鳔8707彬8752濒1778傧1779别1780瘪1781瘪
1787傧5747豳6557缤7145玢7167槟7336殡7375膑7587镔7957髌8738鬓8762兵1788冰1789柄1790丙1791秉

编码（如电报码）{ 等长编码 / 不等长编码

特点：{ 编码长度 / 译码速度 / 传输速度

3.7.1 哈夫曼树及其应用

增长树 {内结点 ○ / 外结点 □



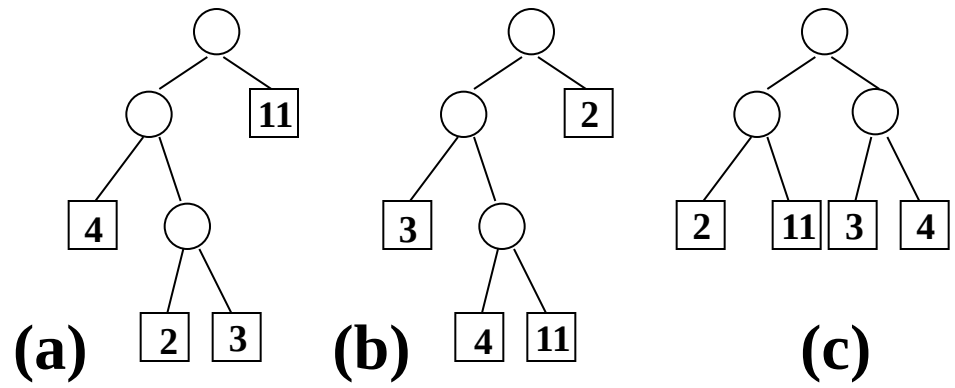
路径长度

如内结点数为 n，则外结点 S = n + 1

内结点路径长度 I = 2×1+3×2+1×3 = 11

外结点路径长度 E = 1×2+5×3+2×4 = 25

如内结点路径长度为I，则外结点路径长度 E = I＋2×n



设： $w_i = \{2, 3, 4, 11\}$

求： $\sum w_j \cdot l_j$（加权路长）

（a）11×1＋4×2＋2×3＋3×3=34

（b）2×1＋3×2＋4×3＋11×3=53

（c）2×2＋11×2＋3×2＋4×2=40

哈夫曼树（最优二叉树）：在给定权值为w_1， w_2… w_n的 n 个叶结点所构成的所有扩充二叉树中，WPL = $\sum w_j \cdot l_j$最小的称为huffman树。

3.7 树型结构的应用（Cont.）

优化(分类统计的)判定过程

例：输入一批学生成绩，将百分制转换成五分制。并且已知：

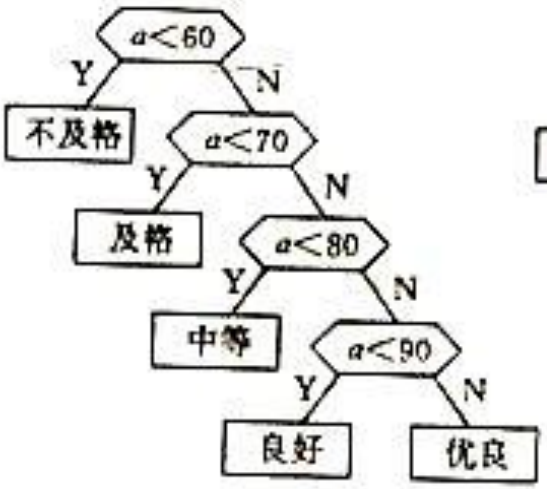
分数	0-59	60-69	70-79	80-89	90-100
比例数	0.05	0.15	0.40	0.30	0.10

```
if (a<60) b="fail"
else if (a<70)  b="pass"
  else if (a<80) b="general"
    else if(a<90) b="good"
      else b="excellent"
```

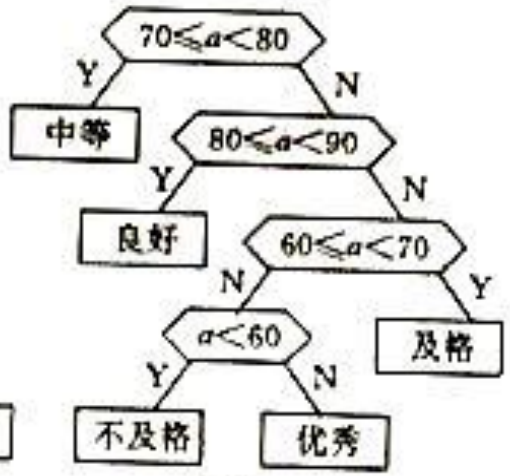
如图（a）所示

3.7 树型结构的应用（Cont.）

以5，15，40，30，10为权构造一株扩充二叉树如图(b)所示，将判定框中的条件分开，可得到(c)，从而实现判定过程的最优化。

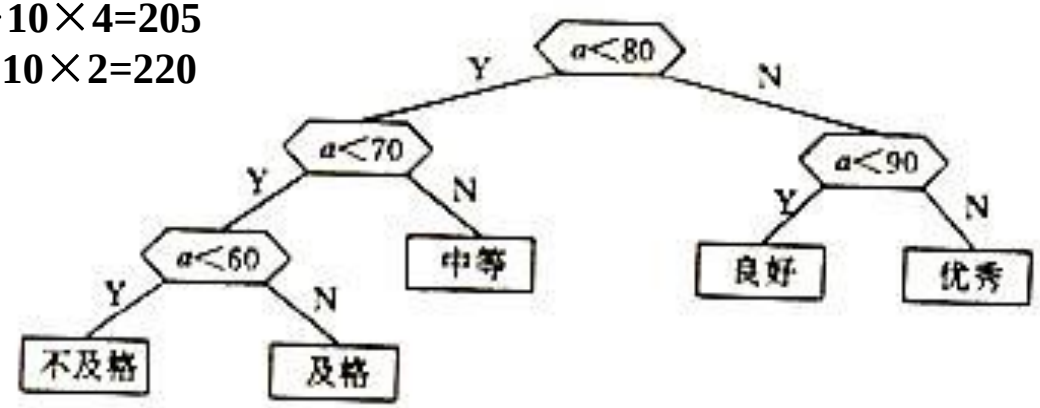


(a)



(b)

（a）5×1＋15×2＋40×3＋30×3＋10×4=285
（b）40×1＋30×2＋15×3＋5×4＋10×4=205
（c）5×3＋15×3＋40×2＋30×2＋10×2=220



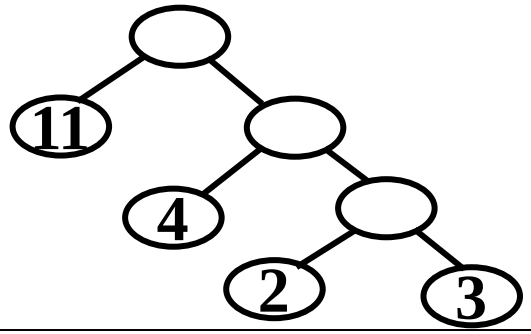
(c)

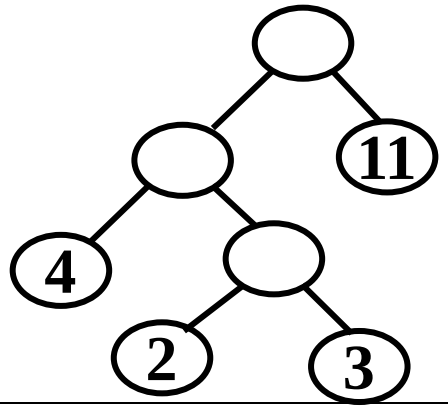
3.7 树型结构的应用（Cont.）

哈夫曼树（最优二叉树）

在给定权值为w_1， w_2... w_n的 n 个叶结点所构成的所有扩充二叉树中，$WPL=\sum w_j \cdot l_j$最小的称为huffman树。

- 哈夫曼树的特点：
  - 权值越大的叶子结点越靠近根结点，而权值越小的叶子结点越远离根结点。（构造哈夫曼树的核心思想）
  - 只有度为0（叶子结点）和度为2（分支结点）的结点，不存在度为1的结点。
  - n个叶结点的哈夫曼树的结点总数为2n-1个。
  - 哈夫曼树不唯一，但WPL唯一。



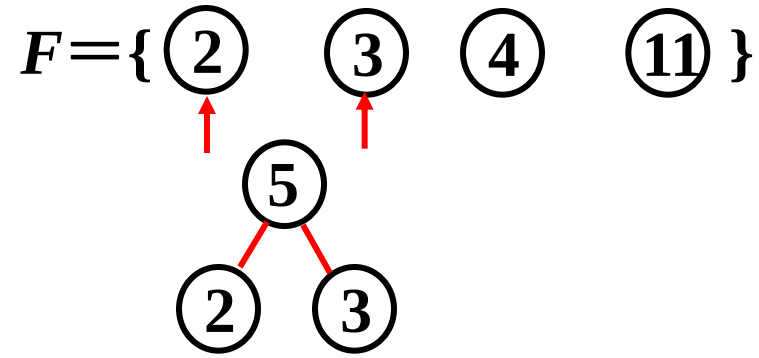


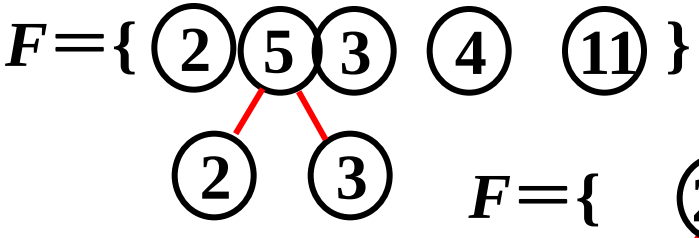
3.7 树型结构的应用（Cont.）

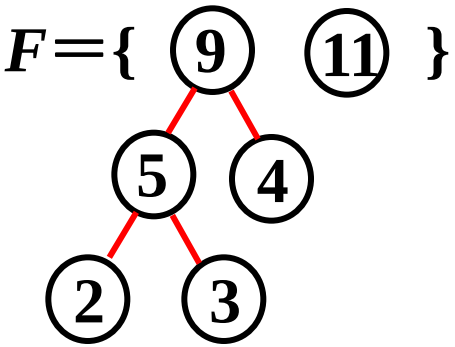
- 哈夫曼树的构造方法：
  - (1) 初始化：由给定的n个权值$\{w_1, w_2, \ldots, w_n\}$构造$n$棵只有一个根结点、左右子树均空的二叉树，从而得到一个二叉树集合$F=\{T_1, T_2, \ldots, T_n\}$；
  - (2) 选取与合并：在F中选取根结点的权值最小的两棵二叉树分别作为左、右子树构造一棵新的二叉树，这棵新二叉树的根结点的权值为其左、右子树根结点的权值之和；
  - (3) 删除与加入：在F中删除作为左、右子树的两棵二叉树，并将新建立的二叉树加入到F中；
  - (4) 重复(2)、(3)两步，当集合F中只剩下一棵二叉树时，这棵二叉树便是哈夫曼树。

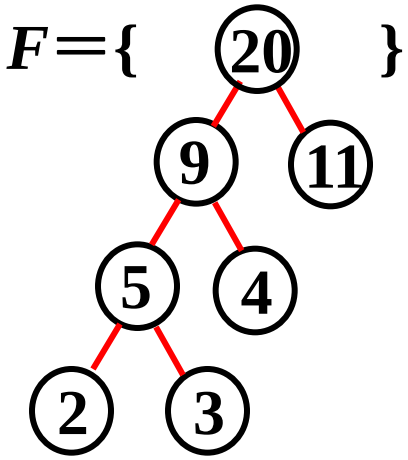
3.7 树型结构的应用（Cont.）

- 哈夫曼树的构造示例：$W=\{2, 3, 4, 11\}$
  - 初始化：
  - 选取与合并：
  - 删除与加入：
  - 重复：





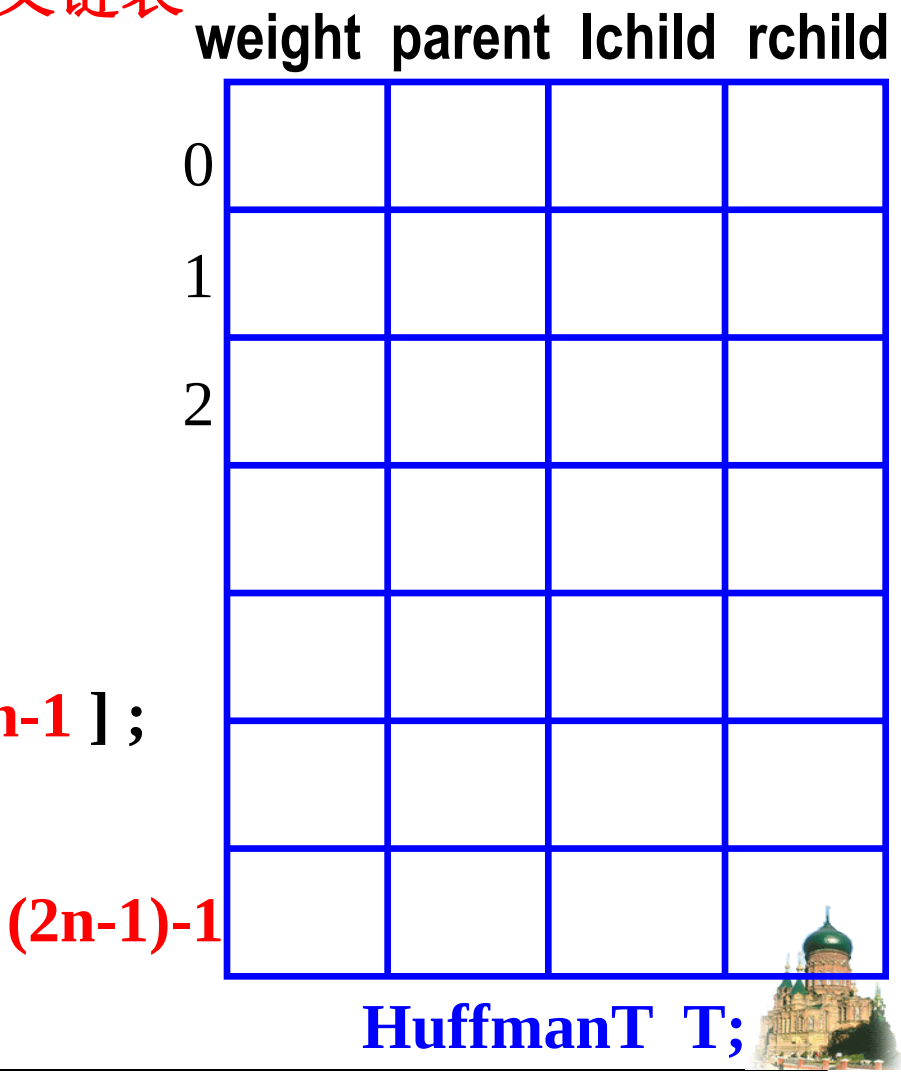




3.7 树型结构的应用（Cont.）

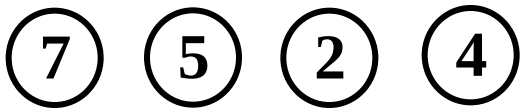
- 哈夫曼树的存储结构----静态三叉链表

```
typedef  struct { // 结点型
      double  weight ; // 权值
       int    lchild ;  // 左孩子链
       int    rchild ; // 右孩子链
       int    parent; // 双亲链
}HTNODE ;
typedef   HTNODE  HuffmanT[ 2n-1 ] ;
```



3.7 树型结构的应用（Cont.）

- 哈夫曼树构造算法的实现示例：

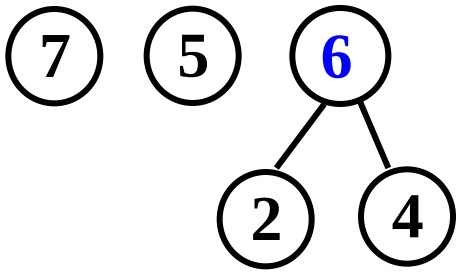


	weight	parent	lchild	rchild
0	7	-1	-1	-1
1	5	-1	-1	-1
2	2	-1	-1	-1
3	4	-1	-1	-1
4		-1	-1	-1
5		-1	-1	-1
6		-1	-1	-1

初始化

3.7 树型结构的应用（Cont.）

- 哈夫曼树构造算法的实现示例：



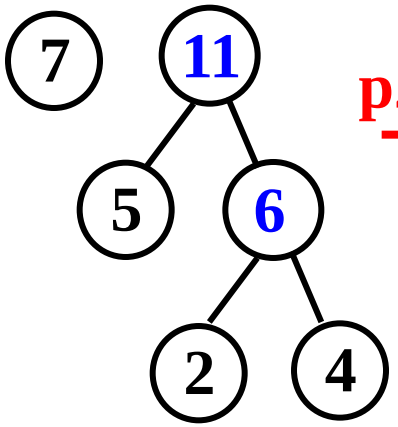
p1 →
p2 →
i →

	weight	parent	lchild	rchild
0	7	-1	-1	-1
1	5	-1	-1	-1
2	2	4 ~~-1~~	-1	-1
3	4	4 ~~-1~~	-1	-1
4	6	-1	2 ~~-1~~	3 ~~-1~~
5		-1	-1	-1
6		-1	-1	-1

过程

3.7 树型结构的应用（Cont.）

- 哈夫曼树构造算法的实现示例：



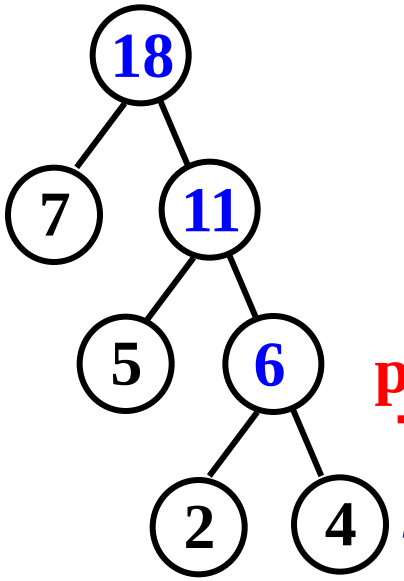
p1 →
p2 →
i →

	weight	parent	lchild	rchild
0	7	-1	-1	-1
1	5	5 ~~-1~~	-1	-1
2	2	4	-1	-1
3	4	4	-1	-1
4	6	5 ~~-1~~	2	3
5	11	-1	1 ~~-1~~	4 ~~-1~~
6		-1	-1	-1

过程

3.7 树型结构的应用（Cont.）

- 哈夫曼树构造算法的实现示例：



		weight	parent	lchild	rchild
p1 →	0	7	~~-1~~ 6	-1	-1
	1	5	5	-1	-1
	2	2	4	-1	-1
	3	4	4	-1	-1
	4	6	5	2	3
p2 →	5	11	~~-1~~ 6	1	4
i →	6	18	-1	~~-1~~ 0	~~-1~~ 5

过程

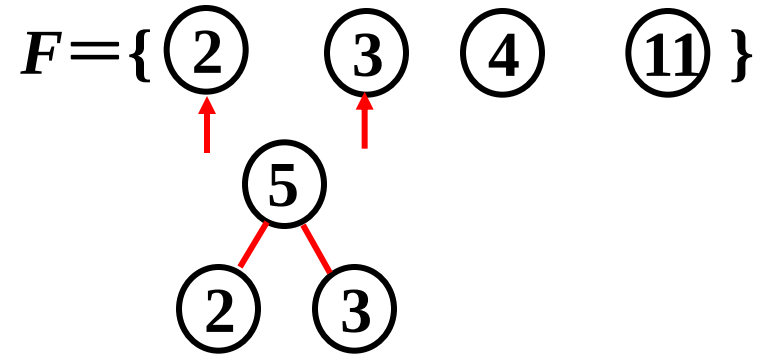
3.7 树型结构的应用（Cont.）

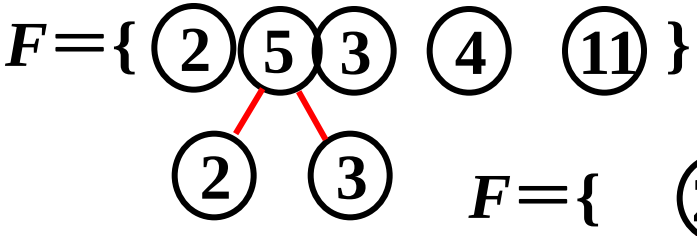
- 哈夫曼树构造算法的实现

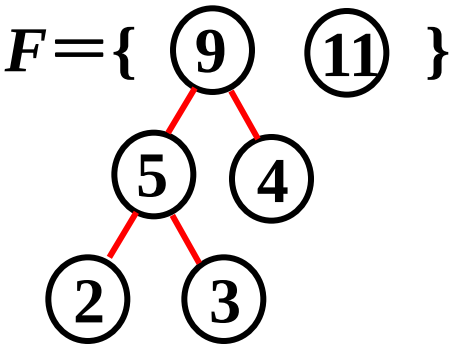
```
void  CreartHT(HuffmanT T)//构造huffam树,T[2n-2]为其根
{  int i ,p1 ,p2;
   InitHT(T);                                  //1.初始化
   InputW(T);                                  //2.输入权值
   for（i = n; i < 2n-1; i++）{                //3. n-1次合并*/
      SelectMin(T, i-1, &p1, &p2);             //3.1
      T[p1].parent =T[p2].parent = i; //3.2
      T[i].lchild= p1;
      T[i].rchild= p2;
      T[i].weight =T[p1].weight + T[p2].weight;
    }
}
```

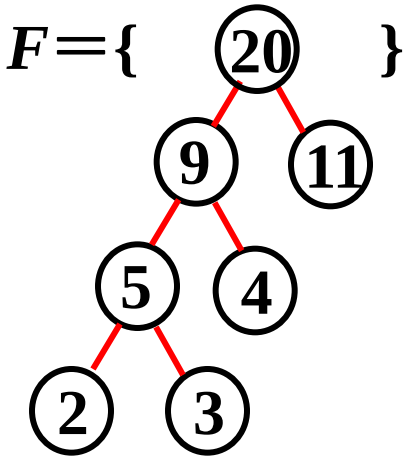
3.7 树型结构的应用（Cont.）

- 哈夫曼树的构造示例：$W=\{2, 3, 4, 11\}$
  - 初始化：
  - 选取与合并：
  - 删除与加入：
  - 重复：









3.7 树型结构的应用（Cont.）

哈夫曼树的应用----哈夫曼编码

■编码：是指将文件（字符集）中的每个字符转换为一个唯一的(二)进制串。

■译码（解码）：是指将(二)进制串转换为对应的字符。

§ 对于给定的字符集，可能存在多种编码方案，但应选择最优的

3.编码的前缀性：

■对字符集进行编码时，如果任意一个字符的编码都不是其它任何字符编码的前缀，则称这种编码具有前缀性或前缀编码。

■注意

✓等长编码具有前缀性；

✓变长编码可能使译码产生二义性，即不具有前缀性。

如, E(00), T(01), W(0001), 则译码时无法确定信息串是ET还是W。

3.7 树型结构的应用（Cont.）

哈夫曼树的应用----哈夫曼编码

- 相关术语
  - 平均编码长度：
    - 对于给定的字符集（一组对象），可能存在多种编码方案，但应选择最优的。
    - 平均编码长度：设每个（对象）字符c_j的出现的概率为p_j，其二进制位串长度（码长）为l_j，则$\sum p_j \cdot l_j$表示该组对象（字符）的平均编码长度。
    - 最优前缀码：使得平均编码长度$\sum p_j \cdot l_j$最小的前缀编码称为最优的前缀码。

字符	a	b	c	d	e	f	平均码长
概率	0.45	0.13	0.12	0.16	0.09	0.05	
等长	000	001	010	011	100	101	3
变长	0	101	100	111	1101	1100	2.24

$=\lceil \log_2 |C| \rceil$

$=\sum p_j \cdot l_j$

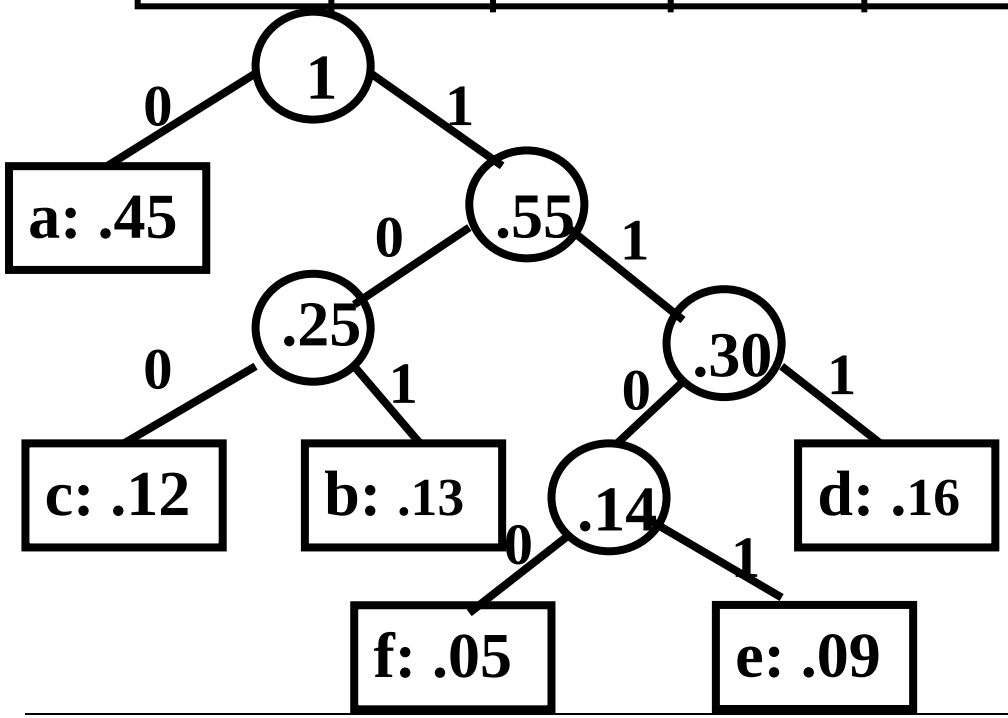
3.7 树型结构的应用（Cont.）

- 哈夫曼编码示例

字符	a	b	c	d	e	f	平均码长
概率	0.45	0.13	0.12	0.16	0.09	0.05	
等长	000	001	010	011	100	101	3
变长	0	101	100	111	1101	1100	2.24

$=\lceil \log_2 |C| \rceil$

$=\sum p_j \cdot l_j$



	ch	bits
0	a	0
1	b	101
2	c	100
3	d	111
4	e	1101
5	f	1100
6		

编码表 H

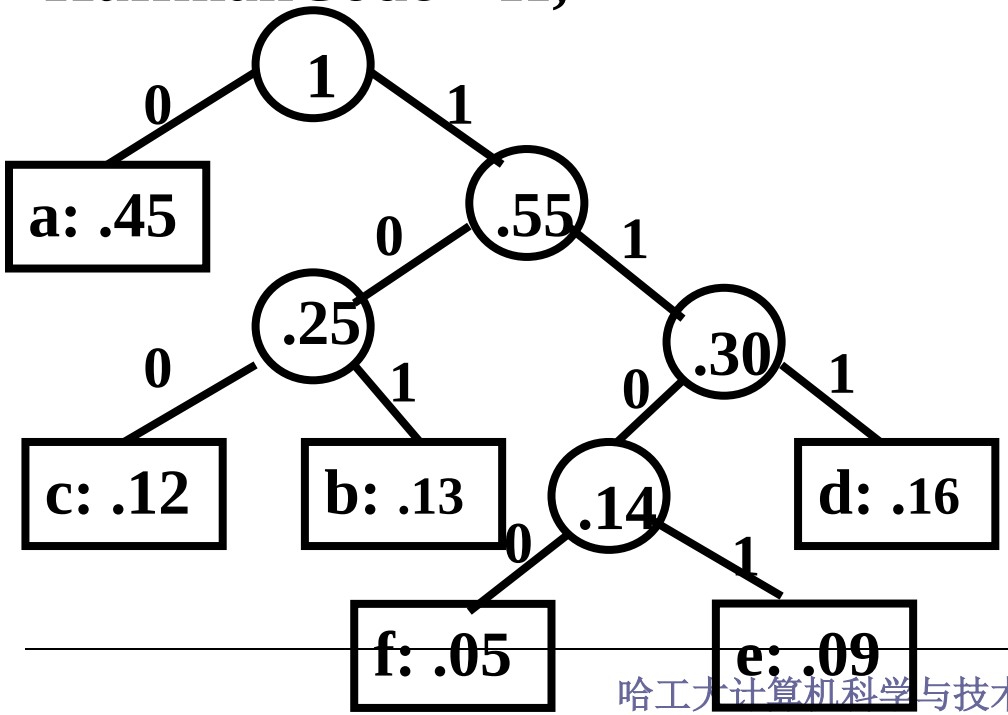
3.7 树型结构的应用（Cont.）

哈夫曼编码表的存储结构

```
typedef struct{
  char  ch;  //存储被编码的字符
  char  bits[n+1]; //字符编码位串
}CodeNode;
typedef CodeNode HuffmanCode[n];
HuffmanCode   H;
```

	ch	bits
0	a	0\0
1	b	101\0
2	c	100\0
3	d	111\0
4	e	1101\0
5	f	1100\0

编码表H



	ch	weight	parent	lchild	rchild
0	a	0.45	10	-1	-1
1	b	0.13	7	-1	-1
2	c	0.12	7	-1	-1
3	d	0.16	8	-1	-1
4	e	0.09	6	-1	-1
5	f	0.05	6	-1	-1
6		0.14	8	5	4
7		0.25	9	2	1
8		0.30	9	6	3
9		0.55	10	7	8
10		1.00	-1	0	9

哈夫曼树T

3.7 树型结构的应用（Cont.）

- 哈夫曼编码算法的实现

```
void CharSetHuffmanEncoding( HuffmanT T, HuffmanCode H)
{  //根据Huffman树T 求Huffman编码表 H
   int c, p, i;          // c 和p 分别指示T 中孩子和双亲的位置
   char cd[n+1];         // 临时存放编码
   int start;            // 指示编码在cd 中的位置
   cd[n]='\0';           // 编码结束符
   for( i =0; i <n;  i++){  // 依次求叶子T[i]的编码
      H[i].ch=getchar();  // 读入叶子T[i]对应的字符
      start=n;            // 编码起始位置的初值
      c =i;               // 从叶子T[i]开始上溯
      while( (p=T[c].parent)>=0){ // 直到上溯到T[c]是树根位置
         cd[--start]=(T[p].lchild==c)? '0' : '1';
         // 若T[c]是T[p]的左孩子，则生成代码0，否则生成代码1
         c=p;     // 继续上溯
      }
      strcpy(H[i].bits,&cd[start]); //复制编码为串于编码表H
   }
}
```

	ch	bits
0	a	0 \0
1	b	1 0 1 \0
2	c	1 0 0 \0
3	d	1 1 1 \0
4	e	1 1 0 1 \0
5	f	1 1 0 0 \0
6		

编码表 H

3.7 树型结构的应用（Cont.）

- 编码：依次读入文件的字符c，在huffman编码表H 中找到此字符，若H[i].ch==c，则将c 转换为H[i].bits中的编码串
- 译码：依次读入文件的二进制码，在huffman树中从根结点T[m-1]出发，若读入0，则走左支，否则，走右支，一旦到达某叶结点T[i]时便译出相应的字符H[i].ch。然后重新从根出发继续译码，直到文件结束。

- 哈夫曼编码一定具有前缀性；
- 哈夫曼编码是最小冗余码；
- 哈夫曼编码方法，使出现概率大的字符对应的码长较短；
- 哈夫曼编码不唯一，可以用于加密；
- 哈夫曼编码译码简单唯一，没有二义性.

- 国际流行两种图像压缩编码标准：在多媒体技术如视频信号的压缩技术中用到了哈夫曼编码。 JPEG、MPEG
- 哈夫曼编码是一种无失真编码，即对源数据压缩后形成的编码，进行恢复时，可完全恢复源数据，但它对静态的数据是可行的。
- 自适应哈夫曼编码

3.7 树型结构的应用（Cont.）

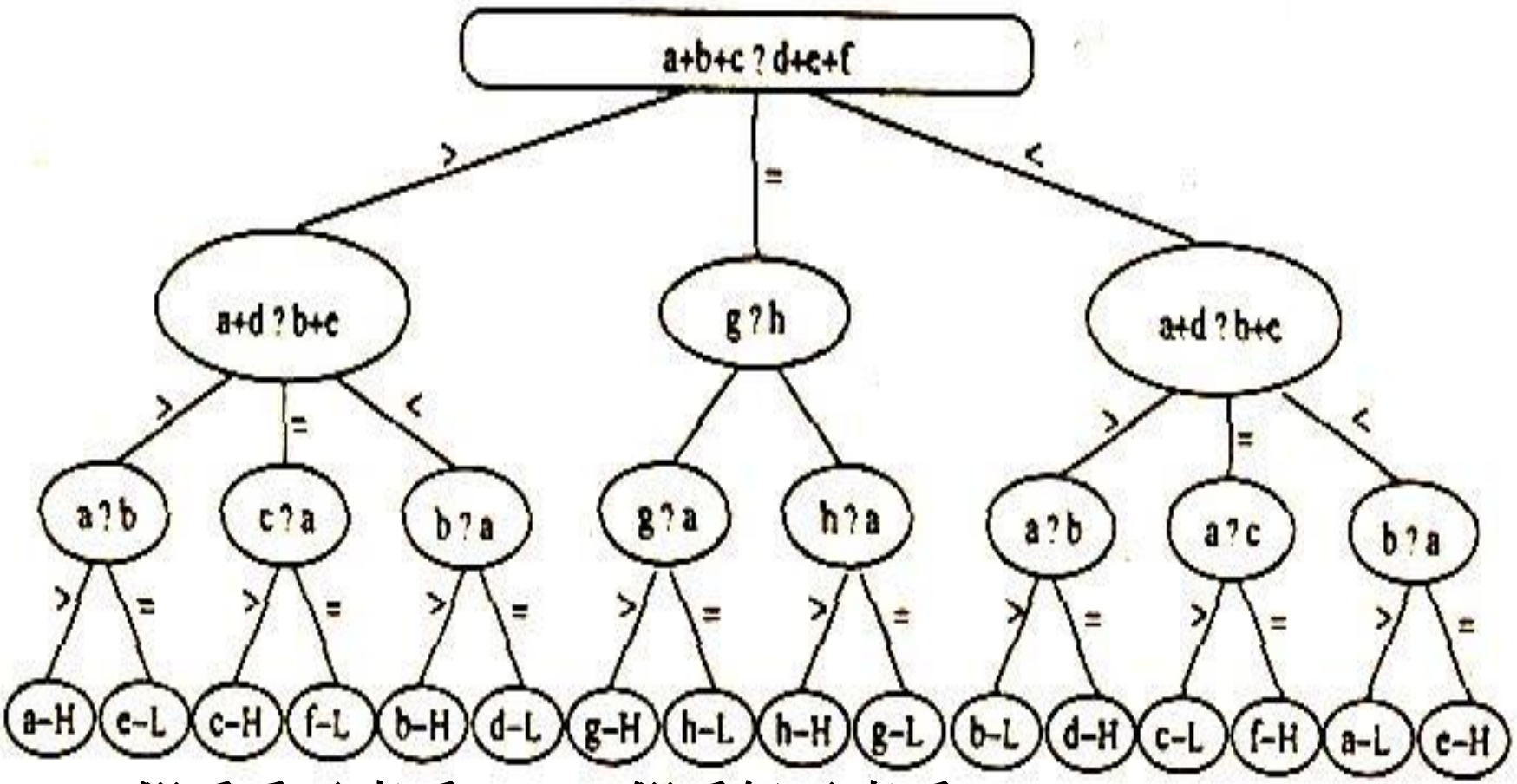
判定树

- 八枚硬币问题：
  - 假定有八枚硬币a、b、c、d、e、f、g、h，已知其中1枚是伪造的假币，假币的重量与真币不同，或重或轻。要求以天平为工具，用最少的比较次数挑出假币。
- 八枚硬币问题的判定树

3.7 树型结构的应用（Cont.）

判定树

- 八枚硬币问题的判定树



H—假币重于真币；L —假币轻于真币

3.7 树型结构的应用（Cont.）

判定树

- 判定树的特点：
  - 一个判定树是一个算法的描述；
  - 每个内部结点对应一个部分解；
  - 每个叶子对应一个解；
  - 每个内部结点连接与一个获得新信息的测试；
  - 从每个结点出发的分支标记着不同的测试结果；
  - 一个解决过程的执行对应于通过根到叶的一条路
  - 一个判定树是所有可能的解的集合

3.7 树型结构的应用

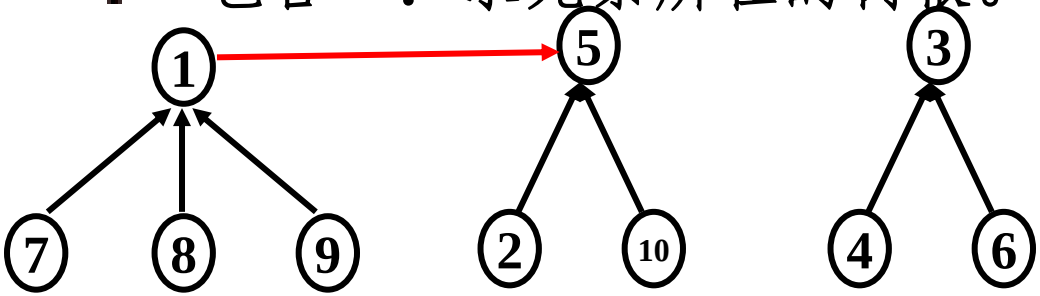
用树结构表示集合

- ADT集合MFSET
  - 集合：
    - 性质相同的元素所组成的整体（有限且互不相交）
  - 集合上的基本操作
    - Union(S_i, S_j, S)： If $S_i \cap S_j == \Phi$， S = $S_i \cup S_j$;
    - Find(i, S)： 求包含i 的集合;
    - Initial(A, x)： 建立集合A，使之只包含x。
  - 例如，S_1={1, 7, 8, 9}，S_2={2, 5, 10 }，S_3={3, 4, 6}，则$S_1 \cup S_2$={1, 2, 5, 7, 8, 9, 10}

3.7 树型结构的应用（Cont.）

用树结构表示集合

- ADT集合MFSET的实现
  - 集合的树结构表示（父链表示）
    - 令集合的元素对应于数组的下标，
    - 而相应的元素值表示其父结点所对应的数组单元下标。
    - “并”：把其中之一当成另一棵树的子树即可。
    - “包含”：求元素所在的树根。



parent		6	5	0	3	0	3	1	1	1	5
	0	1	2	3	4	5	6	7	8	9	10

数组下标：代表元素名

根结点的下标：集合名

3.7 树型结构的应用（Cont.）

用树结构表示集合

- 集合的存储结构

#define n 元素的个数

typedef int MFSET[n+1];

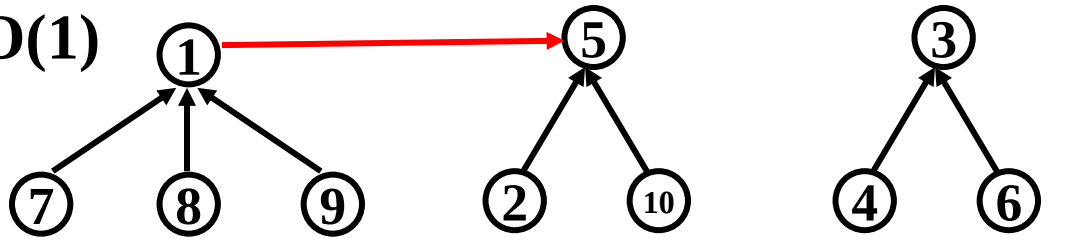
/* 集合的“型”为MFSET,元素的“型”为int */

- 基本操作的实现

void Union(int i, int j ,MFSET parent)

{ parent[i]=j;/* 归并，结果树之根为j */

}//O(1)



	0	1	2	3	4	5	6	7	8	9	10
parent		6 5	5	0	3	0	3	1	1	1	5

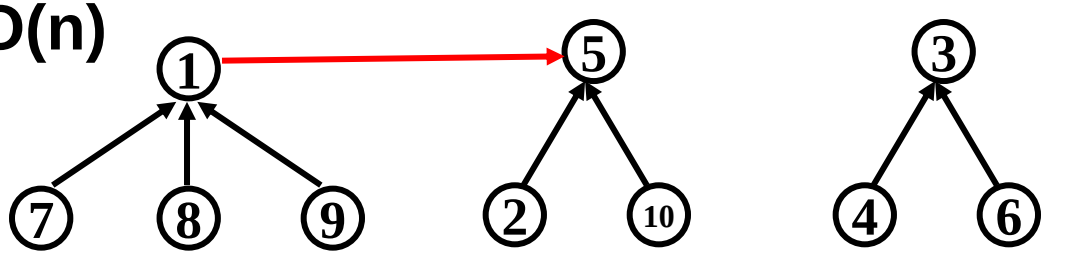
3.7 树型结构的应用（Cont.）

用树结构表示集合

- 基本操作的实现

```
int Find(int i, MFSET parent)
{   int tmp=i;
    while(parent[tmp]!=0)/* >0,未到根 */
      tmp=parent[tmp]; /* 上溯 */
    return tmp;
}//O(n)
```

```
void Initial(int x ,MFSET parent)
{    parent[x]=0;
} //O(1)
```



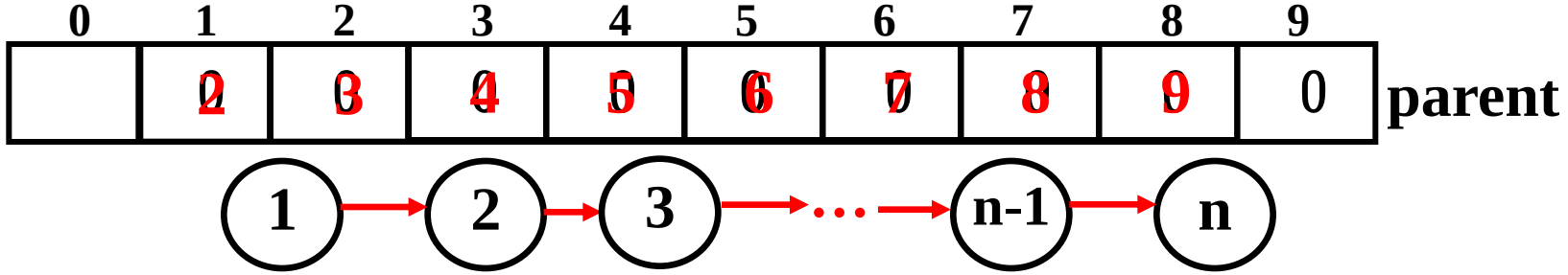
parent		5	5	0	3	0	3	1	1	1	5
	0	1	2	3	4	5	6	7	8	9	10

3.7 树型结构的应用（Cont.）

用树结构表示集合

- 性能分析：看下列操作序列
  - Union(1，2，parent)， Find(1，parent)
  - Union(2，3，parent)， Find(1，parent)
  - Union(3，4，parent)， Find(1，parent)
  - ……
  - Union(n-1,n，parent)，Find(1，parent)

原因：在“并”操作时，将结点多的并入结点少的，从而形成单链树。



每次执行Union的时间都是O(1) ，共n-1次，所需时间O(n)；而每个Find(1, parent)，需要从1开始找到根，当1位于第i层时，Find(1, parent) 所需时间为O(i)，共n-2次，所需时间为O(∑i) = O(n^2)。

3.7 树型结构的应用（Cont.）

用树结构表示集合

- 改进的ADT MFSET的实现
  - 基本想法：
    - 改进“并”操作的原则，即将结点少的并入结点多的；另外，相应的存储结构也要提供支持—以加权规则压缩高度。
  - 存储结构：

```
typedef struct{
        int father;
        int count; // 加权
} MFSET[ n+1 ];
```

  - 基本操作的实现：

3.7 树型结构的应用（Cont.）

用树结构表示集合

- 改进的ADT MFSET的实现

```
void Union(int A,int B,MFSET C)
{   if(C[A].count > C[B].count) { // |B|<|A|
        C[B].father = A; // 并入A
        C[A].count += C[B].count;
    }
    else {   //|A|<|B|
        C[A].father = B;  //并入B
        C[B].count += C[A].count;
    }
}
```

3.7 树型结构的应用（Cont.）

用树结构表示集合

- 改进的ADT MFSET的实现

```
int Find(int x, MFSET C)
{     int tmp=x;
      while(C[tmp].father!=0)//>0,未到根
           tmp=C[tmp].father; // 上溯
      return tmp;
}
void Initial(int A ,MFSET C)
{     C[x].father=0;
       C[x].count=1;
}
```

3.7 树型结构的应用（Cont.）

用树结构表示集合

- 集合的等价分类
  - 等价关系：集合S上具有自反性、对称性和传递性的二元关系R.
  - 等价类：x∈S，y∈S，x≡y ⇔ (x，y)∈R或 xRy。
  - 集合S上的一个等价关系唯一确定一个等价类的集合S/R(商集)。
  - 等价分类：把一个集合分成若干个等价类的过程(分清、分净)
  - 等价分类算法：
    - 例如集合S = {1,2,3,4,5,6,7}的等价对分别是：1≡2，5≡6，3≡4，1≡4

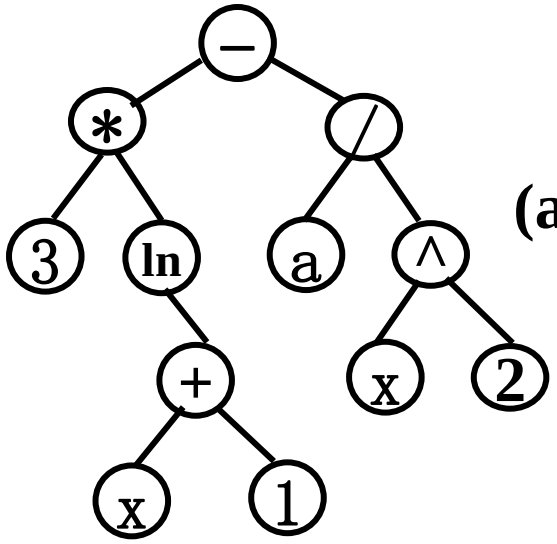
3.7 树型结构的应用（Cont.）

```
void Equivalence (MFSET S)   //等价分类算法
{    int i ,j , k ,m;
     for(i=1; i<=n+1;i++)
         Initial(i,S);              //使集合S只包含元素i
     cin>>i>>j;                     // 读入等价对
     while(!(i==0&&j==0){ // 等价对未读完
          k=Find(i,S);              //求i的根
          m=Find(j,S);              // 求j的根
          if(k!=m)                  //if k==m,i,j已在一个树中，不需合并
              Union(i,j,S);         //合并
          cin<<i<<j;
     }
}
```

3.7 树型结构的应用（Cont.）

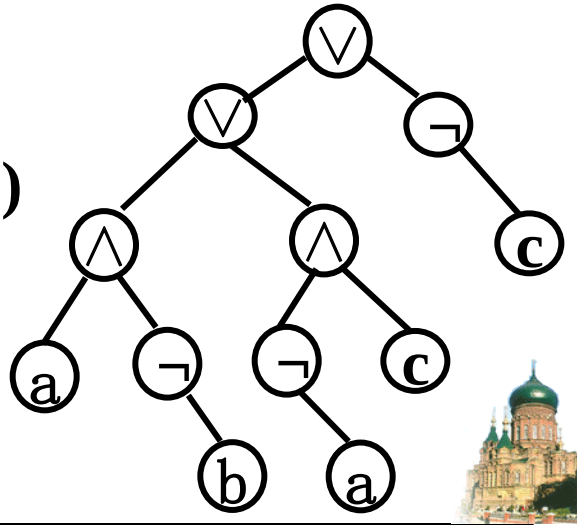
树的应用—表达式求值

- 用树结构表示表达式----表达式树
  - 叶结点表示操作数；
  - 非叶结点表示运算符：
    - 二元运算符有两棵子树对应于它的操作数；
    - 一元运算符有一棵子树对应于它的操作数。



$3 * \ln(x + 1) - a / x^2$

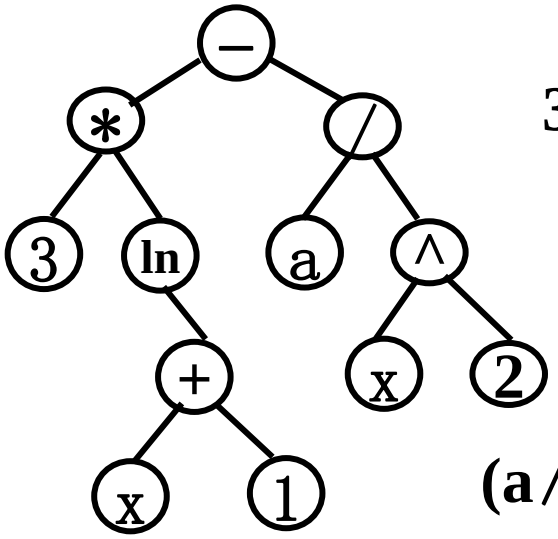
$(a \wedge \neg b) \vee (\neg a \wedge c) \vee (\neg c)$



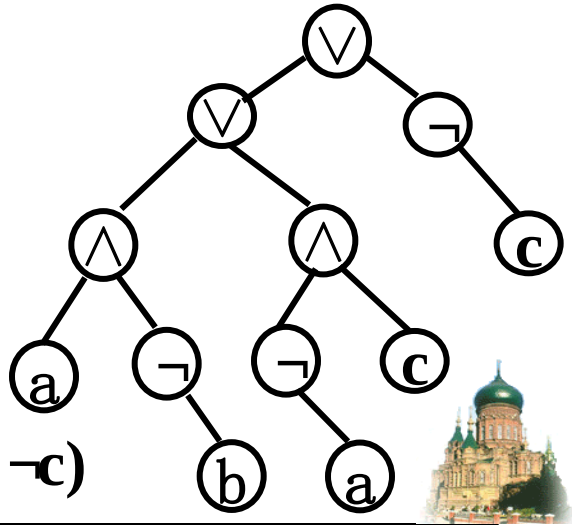
3.7 树型结构的应用（Cont.）

树的应用—表达式求值

- 表达式求值方法
  - 把中缀表达式转换为后缀表达式（栈结构、树结构）；根据后缀表达式计算表达式的值
  - 利用后序遍历算法，先计算左子树的值，然后再计算右子树的值。当到达某结点时，该结点的左右操作数都以求出。



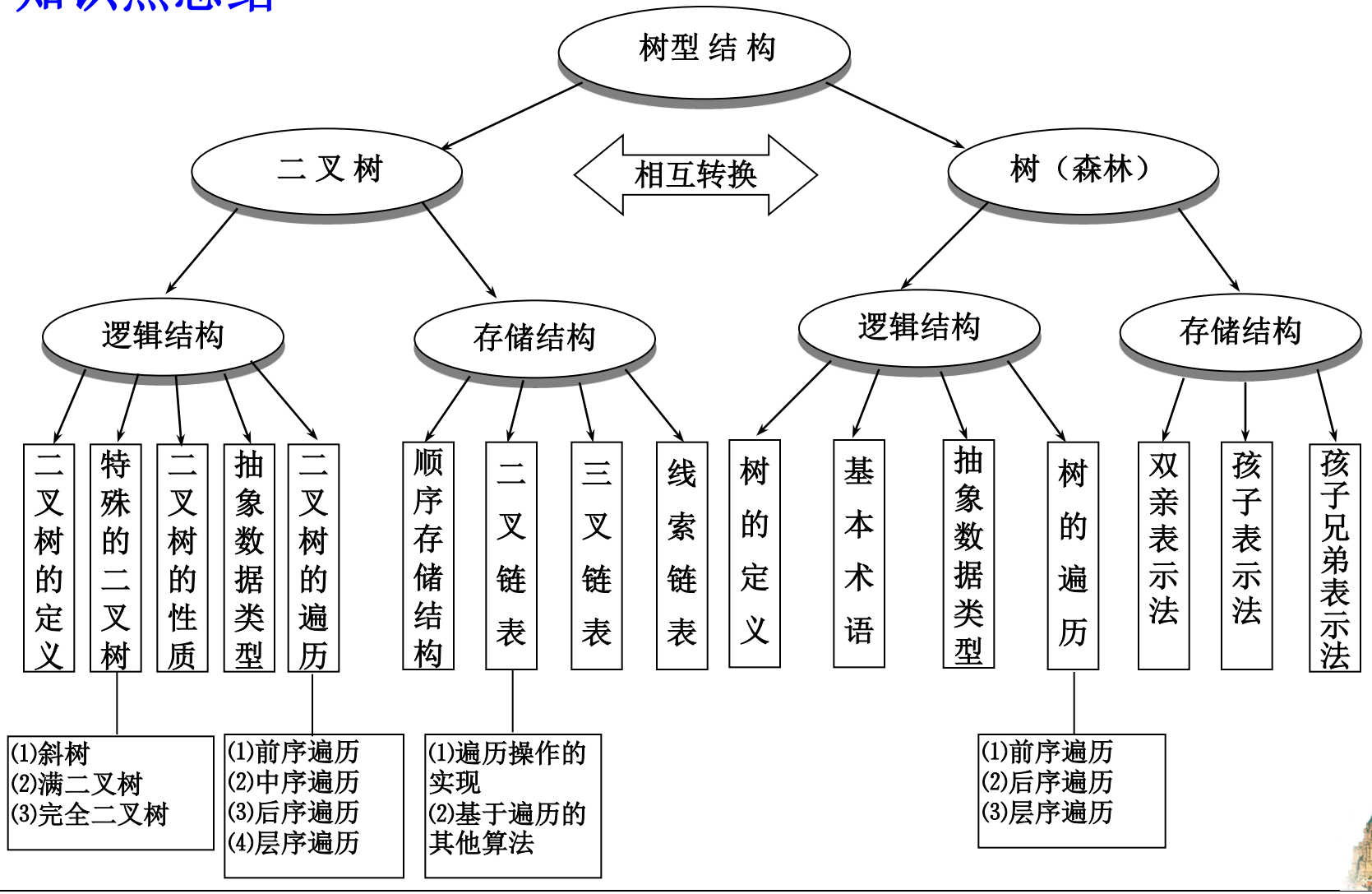
$3 * \ln(x + 1) - a / x^2$



$(a \wedge \neg b) \vee (\neg a \wedge c) \vee (\neg c)$

本章小结

知识点总结


树型结构
二叉树
相互转换
树（森林）
逻辑结构
存储结构
逻辑结构
存储结构
二叉树的定义
特殊的二叉树
二叉树的性质
抽象数据类型
二叉树的遍历
顺序存储结构
二叉链表
三叉链表
线索链表
树的定义
基本术语
抽象数据类型
树的遍历
双亲表示法
孩子表示法
孩子兄弟表示法
(1)斜树
(2)满二叉树
(3)完全二叉树
(1)前序遍历
(2)中序遍历
(3)后序遍历
(4)层序遍历
(1)遍历操作的实现
(2)基于遍历的其他算法
(1)前序遍历
(2)后序遍历
(3)层序遍历