

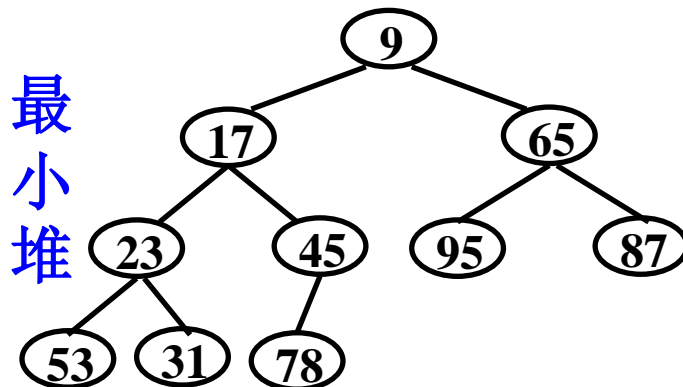
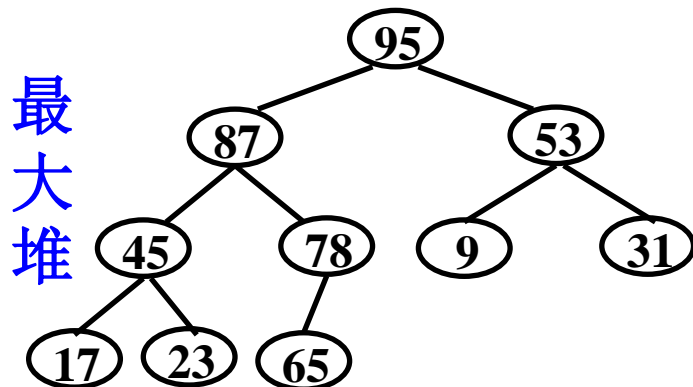


3.3 堆 (Heap)

一、ADT堆

堆的定义

- 如果一棵**完全二叉树**的任意一个非终端结点的元素都**不小于**其左儿子结点和右儿子结点（如果有的话）的元素，则称此完全二叉树为**最大堆**（**大顶堆**、**大根堆**）。
- 如果一棵**完全二叉树**的任意一个非终端结点的元素都**不大于**其左儿子结点和右儿子结点（如果有的话）的元素，则称此完全二叉树为**最小堆**（**小顶堆**、**小根堆**）。
- 特点：**根结点的元素是最大（小）的。。。。。





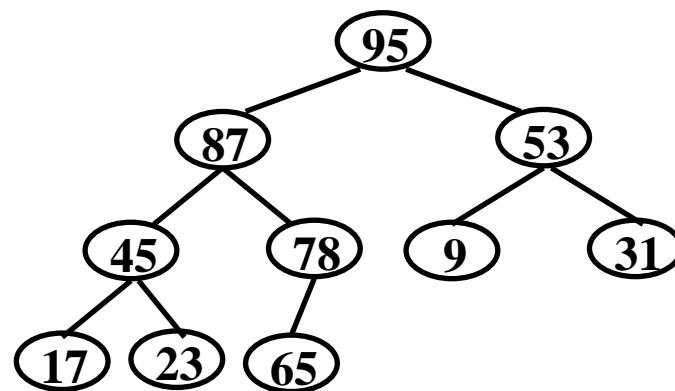
- (最大堆) 操作:**
- 1、MaxHeap(maxsize) 创建一个空堆
 - 2、HeapFull(heap, n) 判断堆是否为满
 - 3、Insert(heap,item) 插入一个元素
 - 4、HeapEmpty(heap) 判断堆是否为空
 - 5、DeleteMax(heap) 删除最大元素

```
#define Maxsize 200
```

(最大堆) 类型定义

```
Typedef struct {
    int key;
    /* other fields */
} Elementtype;
```

```
Typedef struct {
    Elementtype elements[ MaxSize ];
    int n; /*当前元素个数计数器*/
} HEAP;
```



	95	87	53	45	78	09	31	17	23	65									
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19





```
Void MaxHeap ( Heap heap )
```

```
{  
    heap.n = 0 ;  
}
```

堆操作

```
Bool HeapEmpty( HEAP heap )
```

```
{  
    return( !heap.n );  
}
```

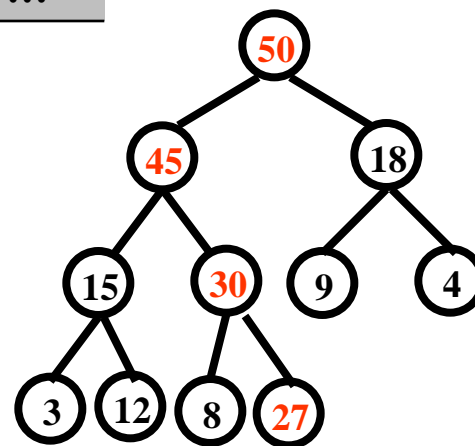
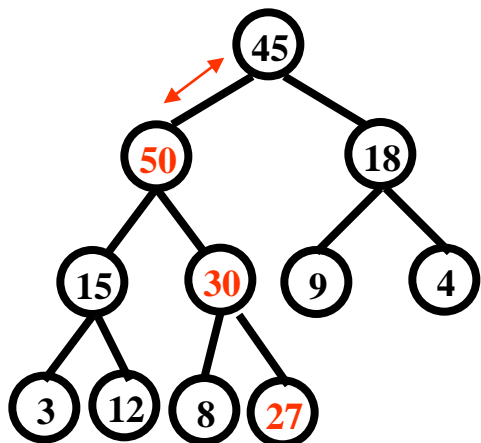
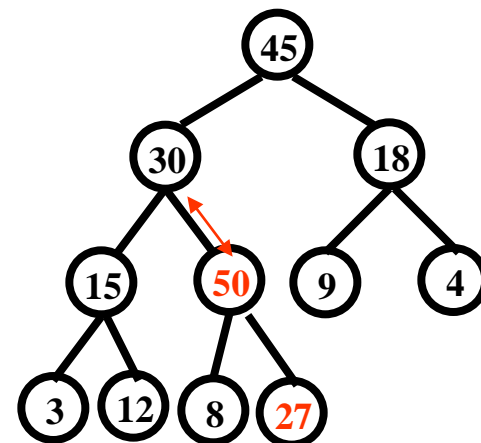
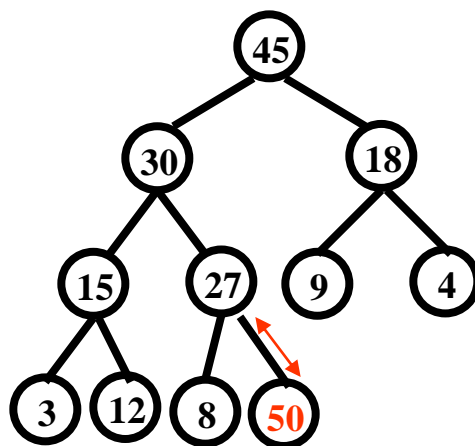
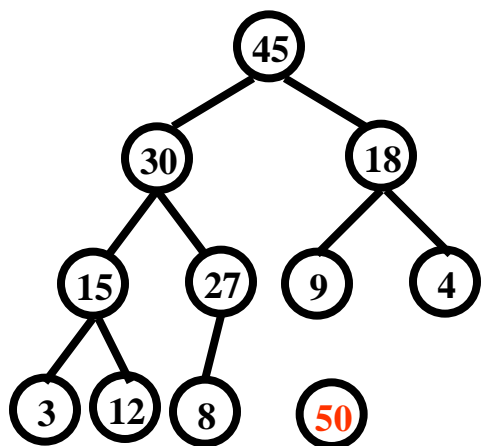
```
Bool HeapFull ( HEAP heap )
```

```
{  
    return( heap.n == MaxSize -1 );  
}
```





45	30	18	15	27	9	4	3	12	8	50	...
----	----	----	----	----	---	---	---	----	---	----	-----



50	45	18	15	30	9	4	3	12	8	27	...
----	----	----	----	----	---	---	---	----	---	----	-----



3.3 堆 (Heap)

二、ADT堆的实现—最大堆的实现

堆的基本操作的实现

④插入

```
void Insert(HEAP& heap, ElemType elem)
{
```

```
    int i;
```

```
    if (!HeapFull(heap)){
```

```
        i=heap.n+1;
```

```
        while((i!=1)&&(elem > heap.data[i/2])){
```

```
            heap.data[i]=heap.data[i/2]; // 下推
```

```
            i/=2;
```

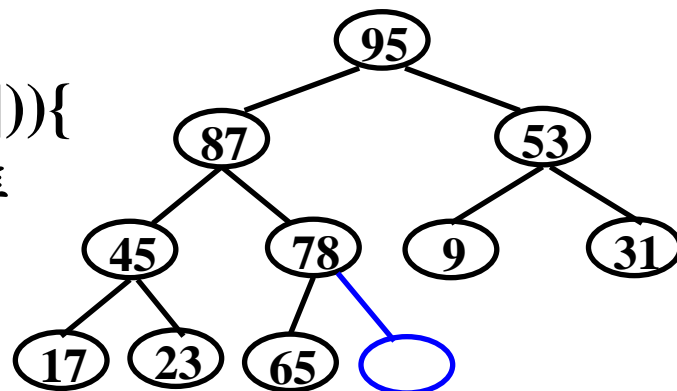
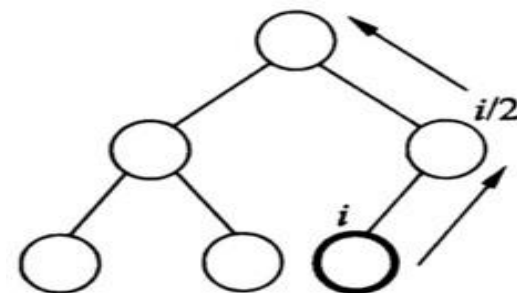
```
        }
```

```
    }
```

```
    heap.data[i]= elem;
```

```
    heap.n++
```

```
    } // 时间复杂度  $O(\log n)$ 
```

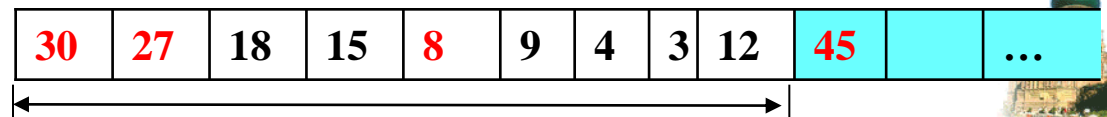
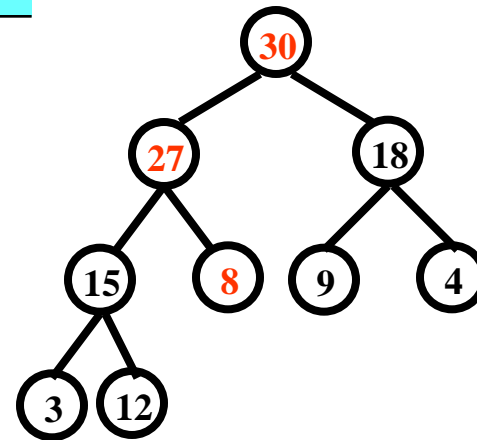
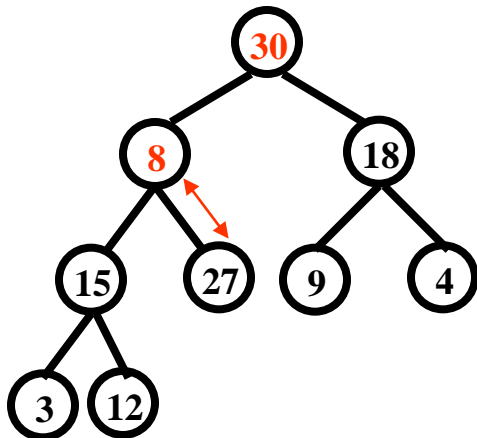
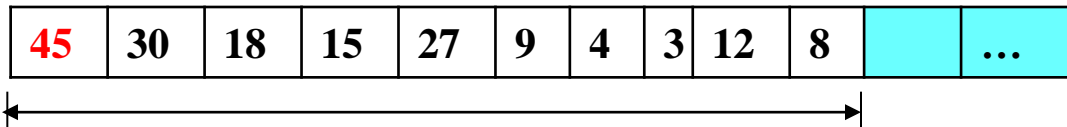
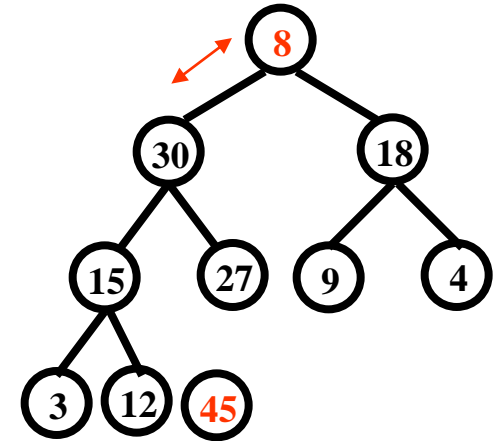
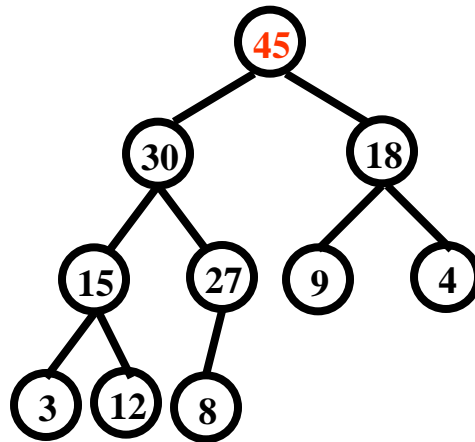


	95	87	53	45	78	09	31	17	23	65				
0	1	2	3	4	5	6	7	8	9	10				





DeleteMax(HEAP heap)





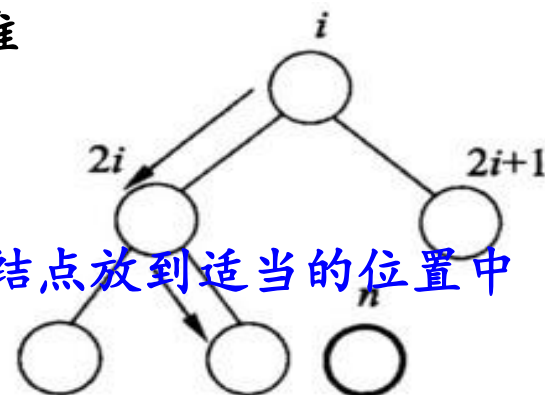
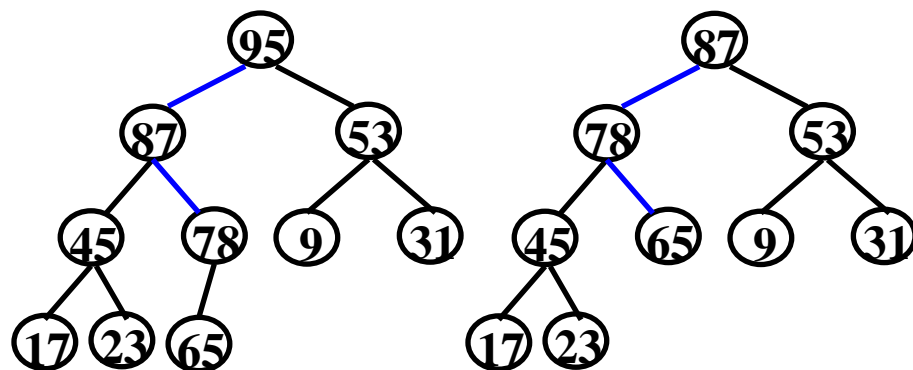
3.3 堆 (Heap)

⑤ 删除最大元素

ElemType DeleteMax(HEAP &heap)

```
{  int parent=1, child=2;
    ElemType elem, tmp;
    if (!HeapEmpty(heap)){
        elem=heap.data[1];
        tmp=heap.data[heap.n--];
        while (child<=heap.n){
            if ((child< heap.n)&&
                (heap.data [child]<heap.data [child+1]))
                child++; //找最大子结点 (左右儿子的大者)
            if (tmp>= heap.data[child]) break;
            heap.data[parent]= heap.data[child];//上推
            parent=child;
            child*=2;
        }
        heap[parent]=tmp; //腾出位置后, 把最后一个结点放到适当的位置中
        return elem;
    }
}
```

//时间复杂度 $O(\log n)$





经常使用堆来实现优先级队列（priority queue）。与第二章所讨论的队列不同的是，优先级队列只对最高（或最低）优先级的元素进行删除。但是在任何时候，都可以把任意优先级的元素插入到优先级队列。

操作系统中的进程管理是优先级队列的一个应用实例，系统中使用一个优先队列来管理进程。

每个进程有进程任务号和优先级两部分组成。当有多个进程排队时，优先级高的先操作。





练习题：设计一个程序模仿操作系统的进程管理问题，进程服务按优先级高的先服务，同优先级的先到先服务的管理原则。设文件task.dat中存放了仿真进程服务请求，其中第一列是进程任务号，第二列是进程的优先级。

1 30

2 20

3 40

4 20

5 0

算法：1) 建立队列
2) 建堆
3) 循环出队，输出。

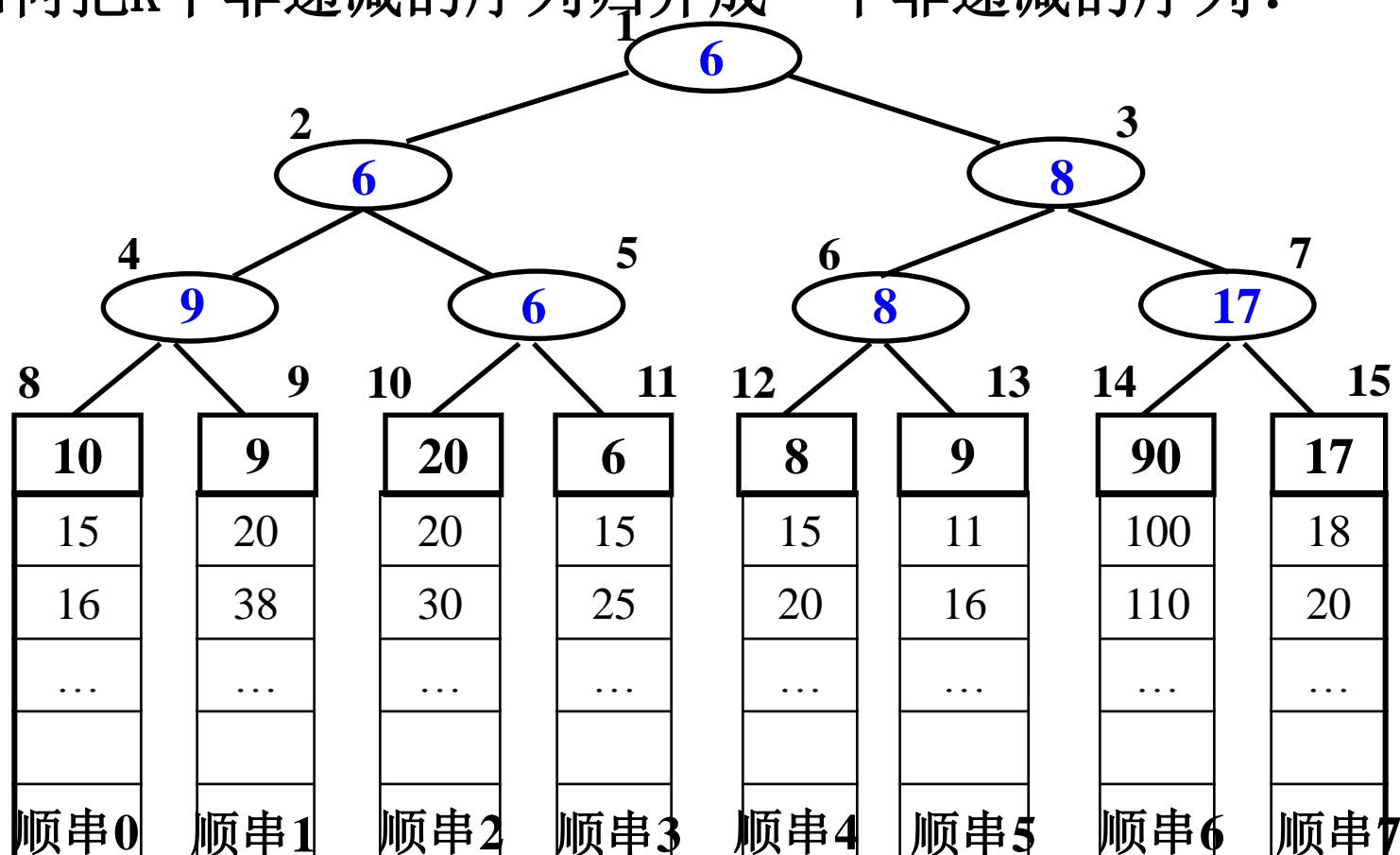




3.4 选择树 (Selection Tree)

一、背景

- 如何从 n 个元素中选择最小的，进而对 n 个元素排序？
- 如何把 K 个非递减的序列归并成一个非递减的序列？

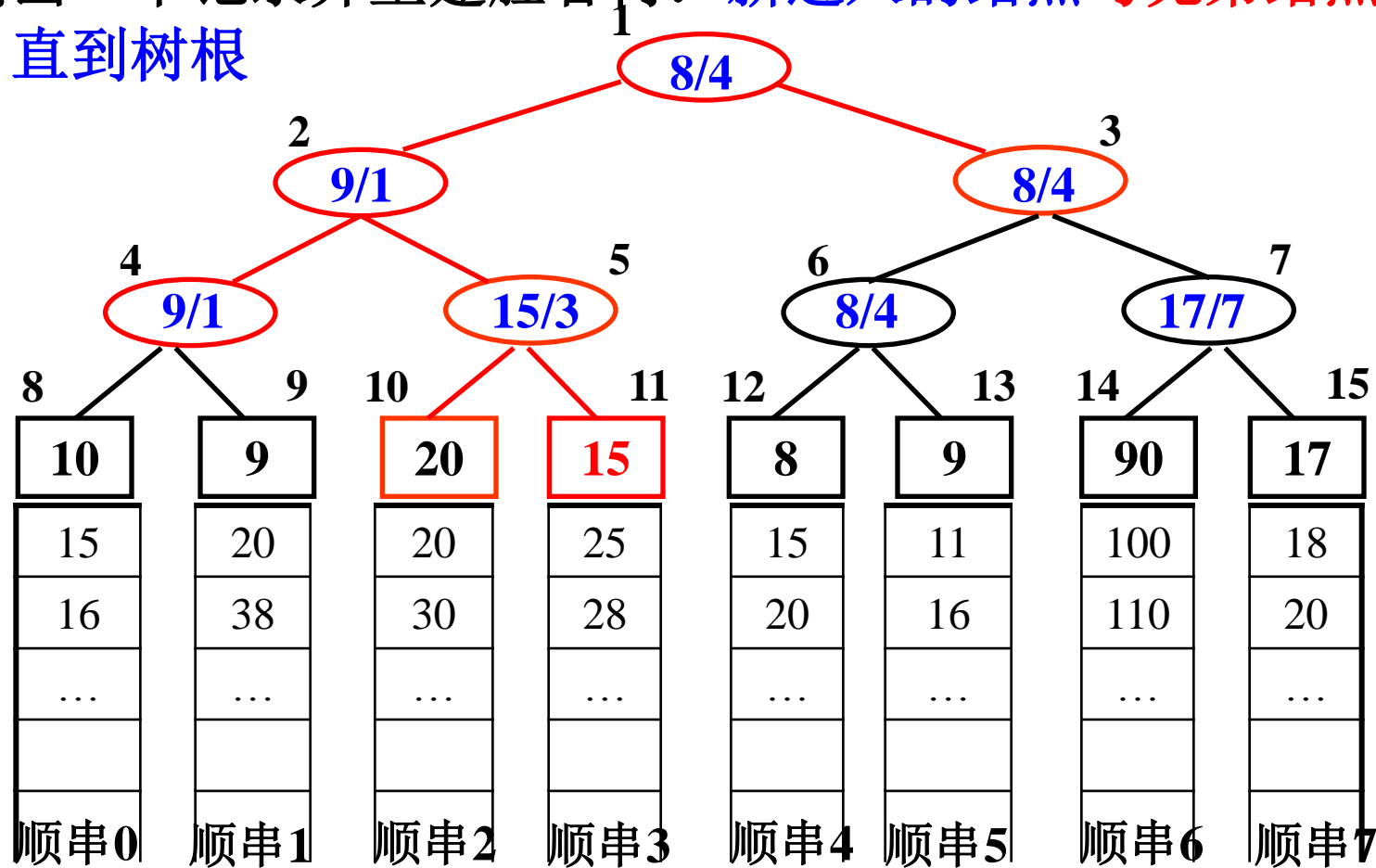




3.4 选择树 (Selection Tree)

二、胜者树(Winner Tree)

➤ 输出一个记录并重建胜者树：新进入的结点与兄弟结点比较，直到树根



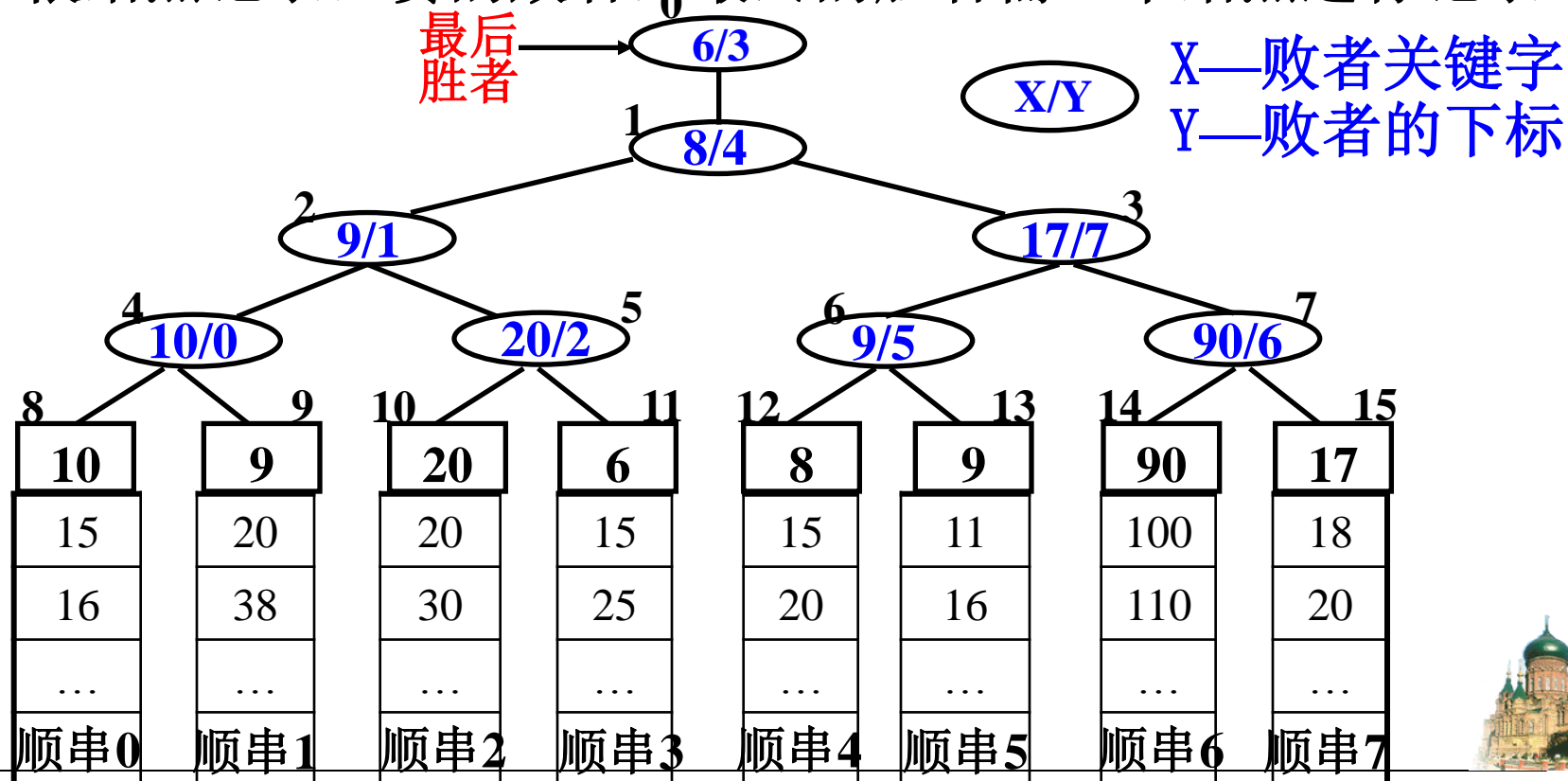


3.4 选择树 (Selection Tree)

三、败者树(Loser Tree)

败者树的构建

- 内部结点保存**败者**，胜者参加下一轮比赛
- 根结点记录比赛的败者，最终的胜者需一个结点进行记录

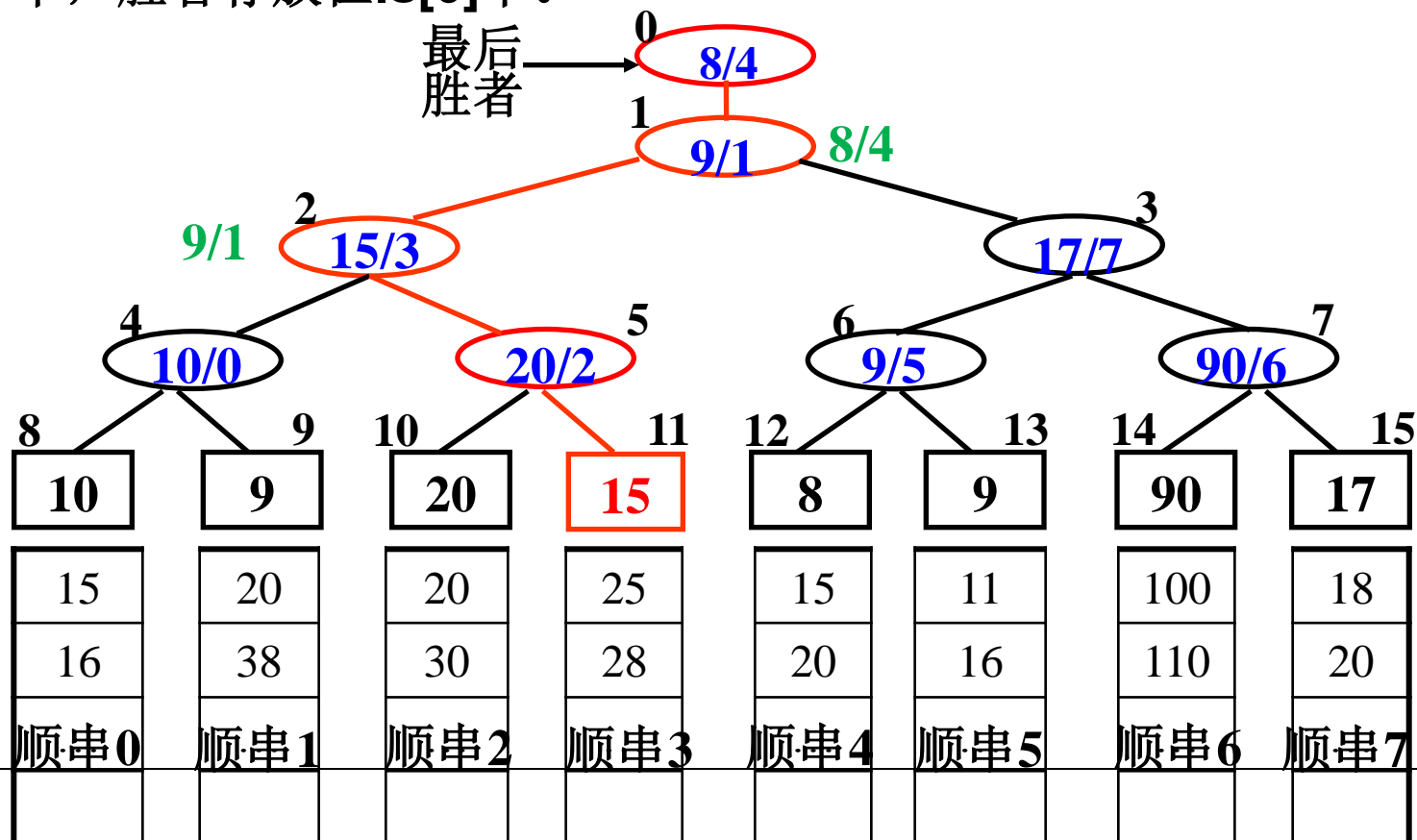




3.4 选择树 (Selection Tree)

败者树的重构

- 将新进入的结点与其父结点进行比赛：将败者存入父结点中，胜者再与上一级的父结点比较。
- 比赛沿着到根的路径不断进行，直到 $ls[1]$ 处。把败者存放在结点 $ls[1]$ 中，胜者存放在 $ls[0]$ 中。





练习题：

1. 已知顺串 $R1[10, 15, 16]$, $R2[9, 20, 38]$, $R3[20, 20, 30]$, $R4[6, 15, 25]$, $R5[8, 15, 20]$, $R6[9, 11, 16]$, $R7[90, 100, 110]$, $R8[17, 18, 20]$ 建立败者树。
2. 简答题：分别利用堆和败者树，给出求 n 个数中的前 k 个最小数的方法描述。并分析时间复杂度和空间占用情况。（*）





3.5 树

树的基本操作

- **Parent(n , T)** 求结点 **n** 的父节点
- **LeftMostChild(n , T)** 返回结点 **n** 的最左儿子
- **RightSibling(n , T)** 返回结点 **n** 的右兄弟
- **Data(n , T)** 返回结点 **n** 的信息
- **CreateK k (v , T1 , T2 , , Tk) , k = 1 , 2 ,**
 - 建立data域值为v的根结点r，有k株子树T1 , T2 , , Tk ， 且自左至右排列；返回r。
- **Root(T)** 返回树T的根结点
- **树的遍历操作**
 - 从根结点出发，按照某种次序访问树中所有结点，使得每个结点被访问一次且仅被访问一次。



树的四种遍历

■ 先根顺序

访问根结点；

先根顺序遍历 T_1 ；

先根顺序遍历 T_2 ；

...

先根顺序遍历 T_k ；

■ 中根顺序

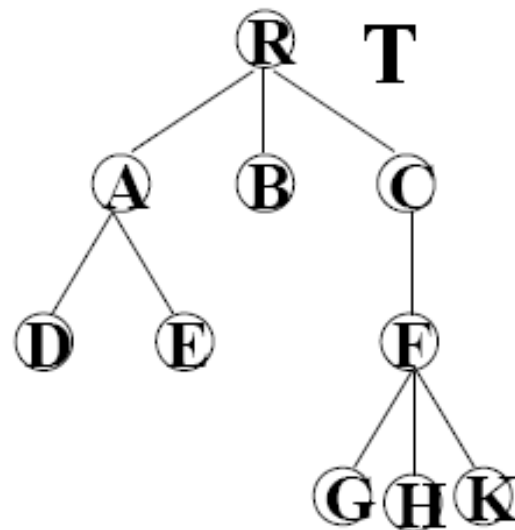
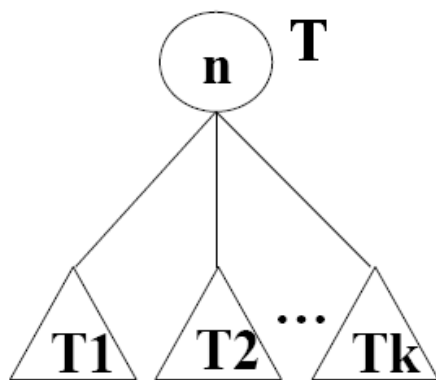
中根顺序遍历 T_1 ；

访问根结点；

中根顺序遍历 T_2 ；

...

中根顺序遍历 T_k ；



先根遍历序列: **RADEBCFGHK**

中根遍历序列: **DAERBGFHKC**

后根遍历序列: **DEABGHKFCR**

■ 后根顺序

后根顺序遍历 T_1 ；

后根顺序遍历 T_2 ；

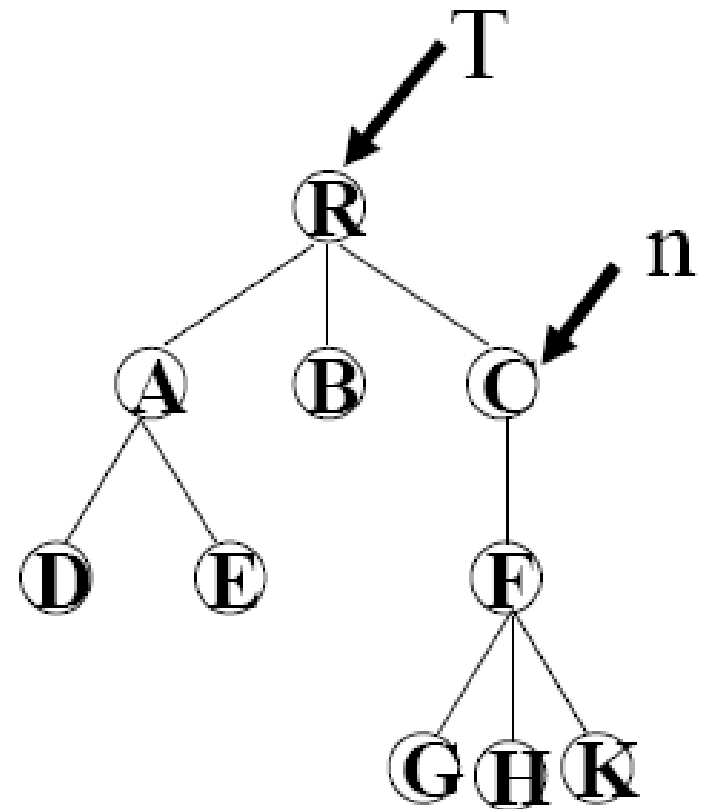
...

后根顺序遍历 T_k ；

访问根结点；

例：假设树的类型为**TREE**，结点的类型为**node**，数据项的类型为**elementtype**，用递归方法给出树的先根遍历如下：

```
void PreOrder(node n , TREE T )
{
    node c ;
    If(n)
    {
        visit( DATA( n,T ) ) ;
        c = LeftMostChild( n , T ) ;
        while ( c != NULL ) {
            PreOrder( c , T ) ;
            c = RightSibling( c , T ) ;
        }
    }
}
```



先根遍历整株树：**PreOrder(ROOT(T), T)**

思考题：写出树的中根遍历和后根遍历算法



3.5 树 (Cont.)

树的存储结构

双亲表示法 (单链表示、父链表示)

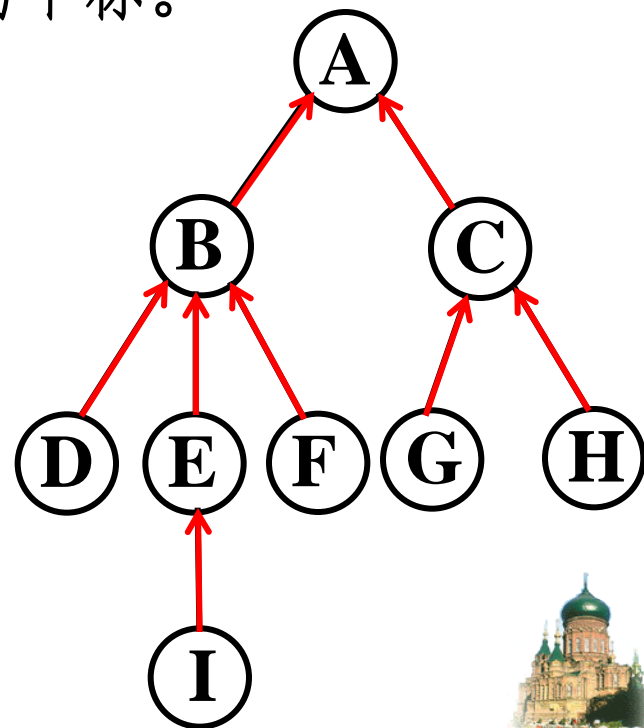
- 每个结点 (根结点除外) 都只有**唯一**的**双亲**结点
- 因此, 可以把各个结点 (一般按**层序**) 存储一维数组中, 同时记录其唯一双亲结点在数组中的下标。

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

结点结构定义

```
Struct node {
    char data ;
    int parent ; } ;
```

```
Typdef node TREE[9];
```





3.5 树 (Cont.)

双亲表示法 (单链表示、父链表示)

存储特点:

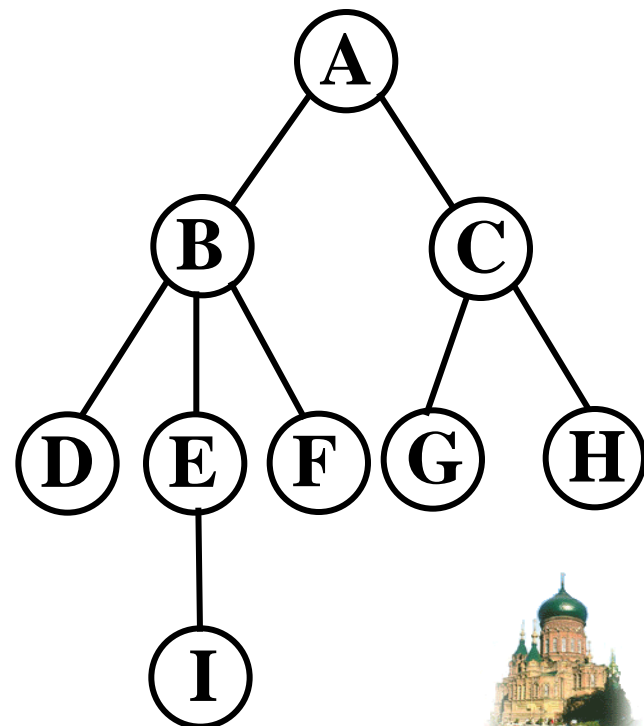
- 每个结点均保存父结点所在的数组单元下标
- 兄弟结点的编号连续。

如何查找双亲结点和祖先? 时间性能?

如何查找孩子结点? 时间性能?

如何查找兄弟结点? 时间性能?

	1	2	3	4	5	6	7	8	9
data	A	B	C	D	E	F	G	H	I
parent	0	1	1	2	2	2	3	3	5
firstchild	2	4	7	0	9	0	0	0	0
rightsib	0	3	0	5	6	0	8	0	0

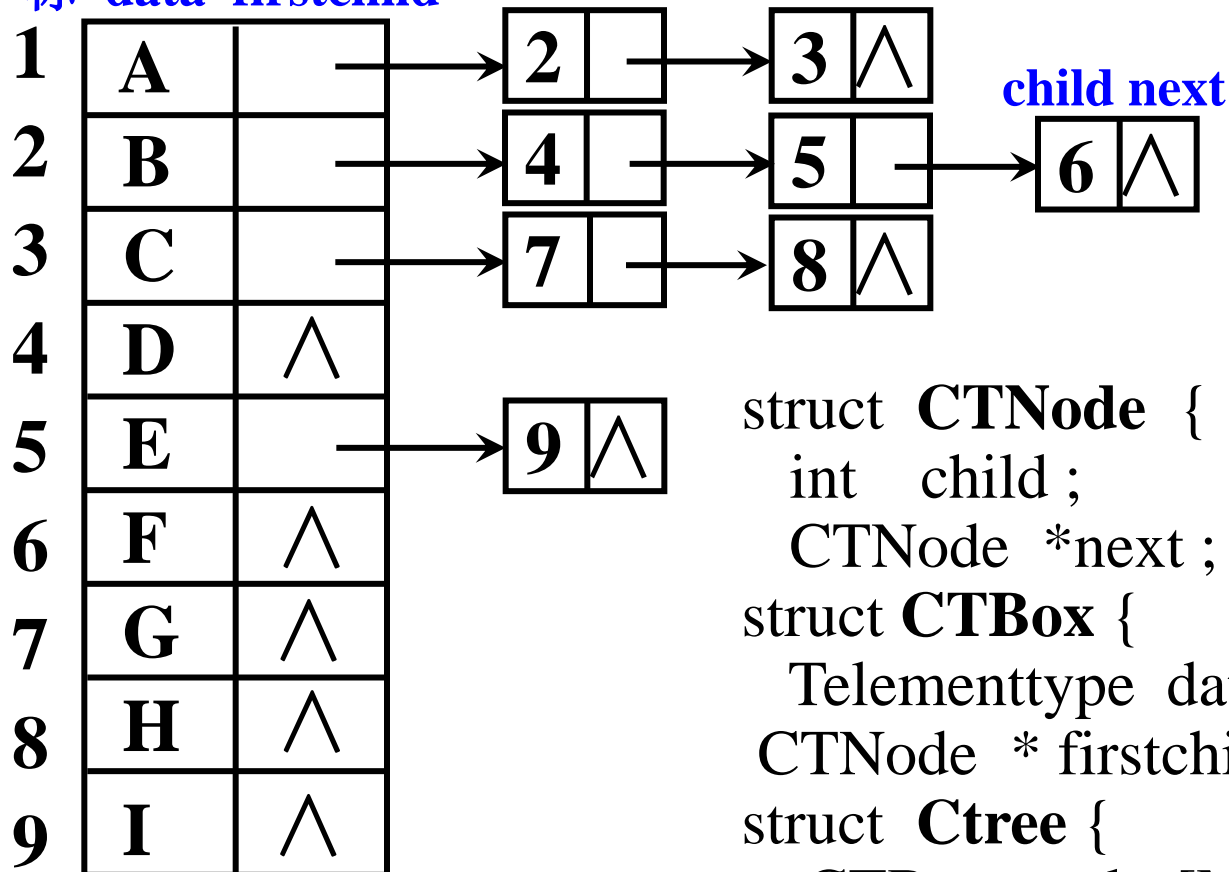




3.5 树 (Cont.)

孩子链表表示法 (邻接表表示)

下标 data firstchild



```

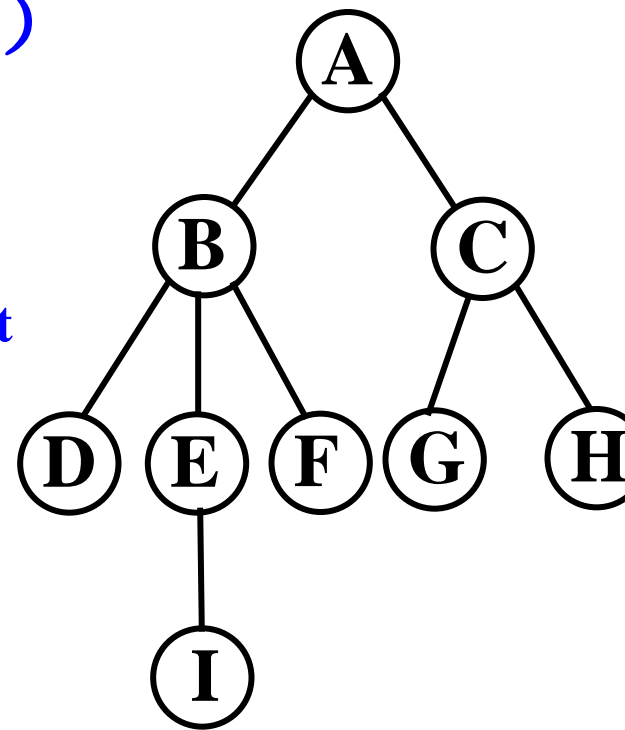
struct CTNode {
    int child;
    CTNode *next;
};
struct CTBox {
    Telementype data;
    CTNode *firstchild;
};
struct Ctree {

```

```

    CTBox nodes[MAX_TREE_SIZE];
    int n, r; // 结点个数、根节点位置
};

```



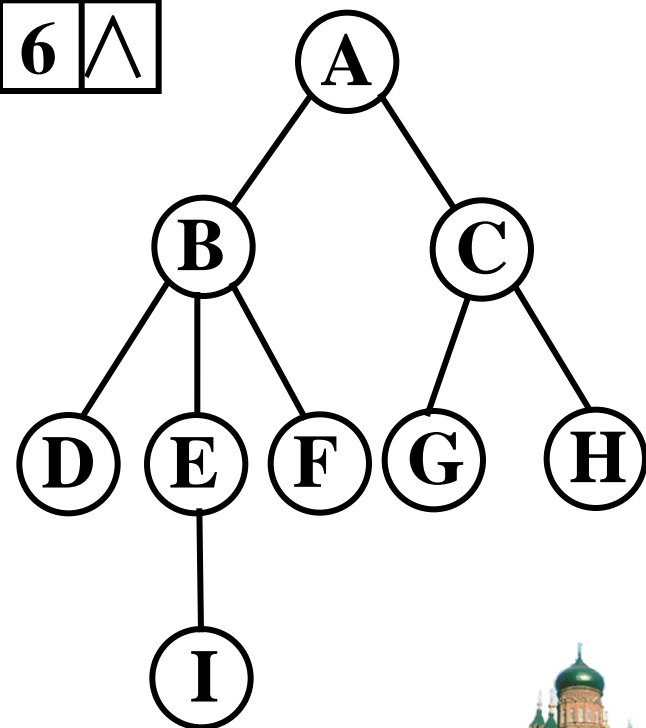


3.5 树 (Cont.)

双亲孩子表示法

data parent firstchild

1	A	0	—	2	—	3	^
2	B	1	—	4	—	5	—
3	C	1	—	7	—	8	^
4	D	2	^				
5	E	2	—	9	^		
6	F	2	^				
7	G	3	^				
8	H	3	^				
9	I	5	^				



➡ 二叉链表表示法（（左）孩子—（右）兄弟链表表示）

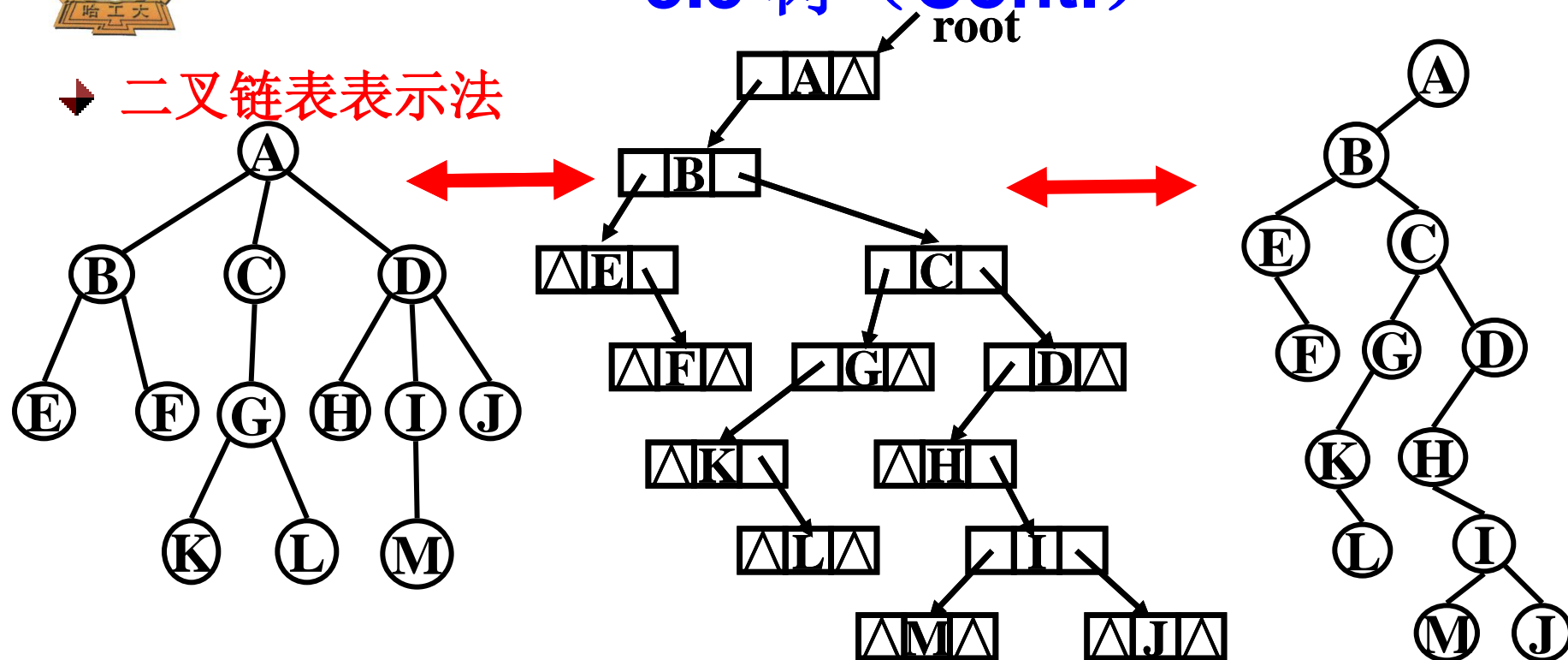
- 某结点的**右兄弟**是唯一的
- 设置两个分别指向该结点的**第一个孩子**和**右兄弟**的指针





3.5 树 (Cont.)

➡ 二叉链表表示法



遍历	树	二叉树
先序	ABEFCGKLDHIMJ	ABEFCGKLDHIMJ
中序	EBFAKGLCHDMIJ	EFBKLGCHMIJDA
后序	EFBKLGCHMIJDA	FELKGMJIHDCBA





3.5 树（Cont.）

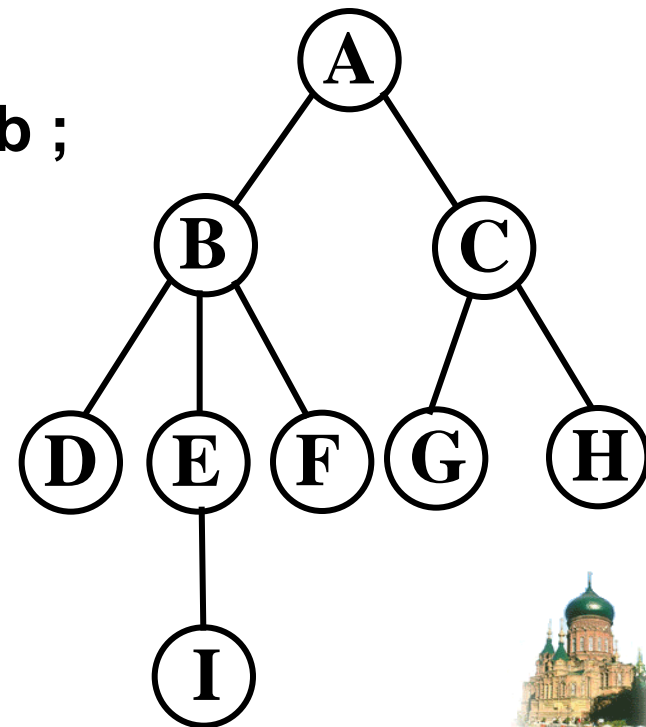
二叉链表表示法（（左）孩子—（右）兄弟链表表示）

■ 结点结构:



firstchild	data	rightsib
------------	------	----------

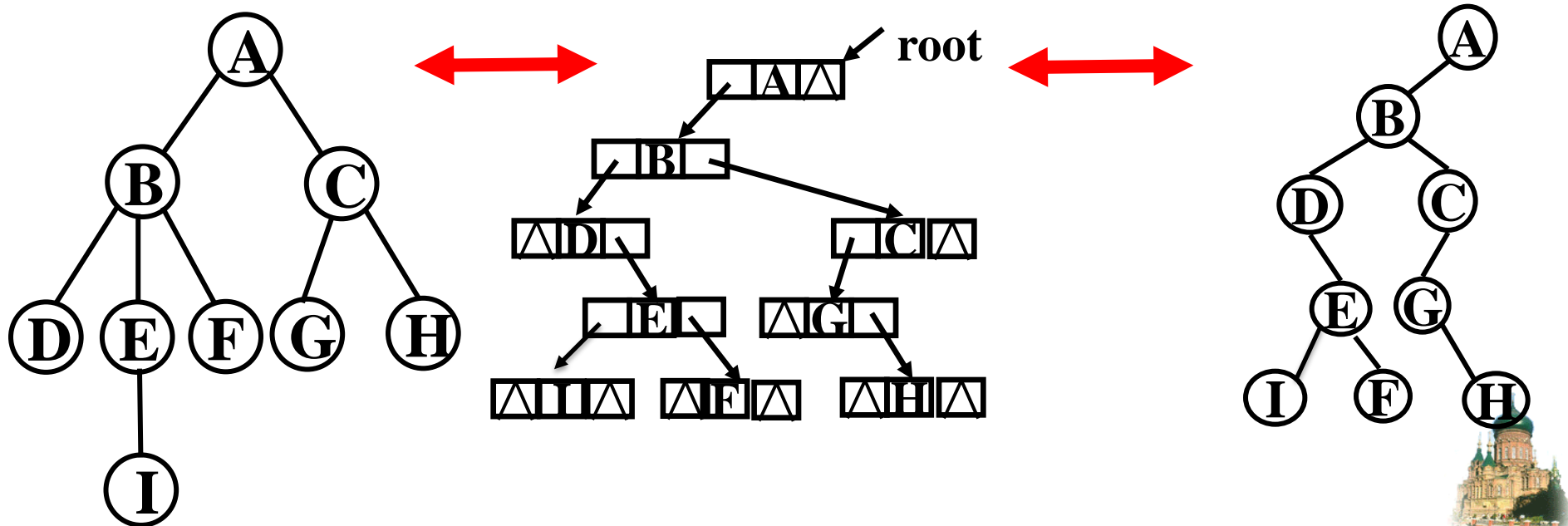
■ 类型定义:

```
struct CSNode { //动态存储结构
    DataType    data;
    CSNode  *firstchild , *rightsib ;
};
typedef struct CSNode  *CSTree ;
```



3.6 森林(树)与二叉树间的转换

	树	二叉树
结点关系	兄弟关系 	双亲和右孩子
	双亲和长子 	双亲和左孩子

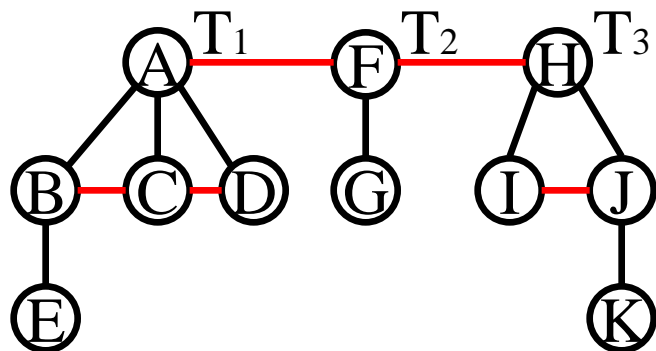




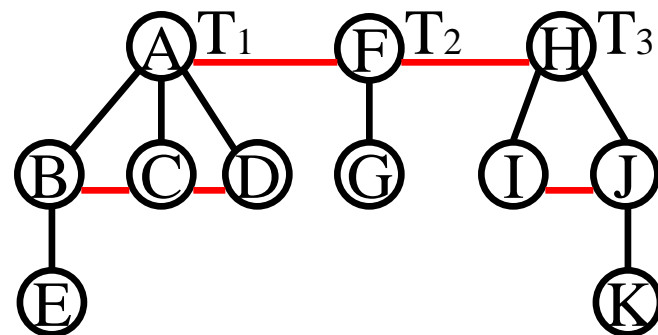
3.6 森林(树)与二叉树间的转换 (Cont.)

森林(树)转换成二叉树

连线:



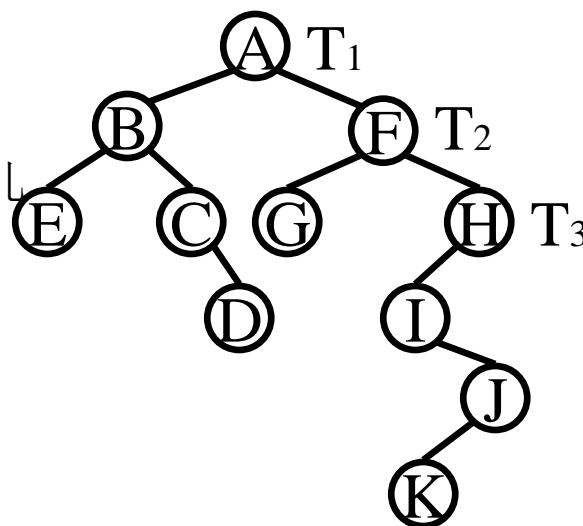
抹线:



§ **连线**: 把每株树的各兄弟结点连起来;
把各株树的根结点连起来 (视为兄弟)

§ **抹线**: 对于每个结点, 只保留与其最左儿子的连线, 抹去该结点与其它结点之间的连线

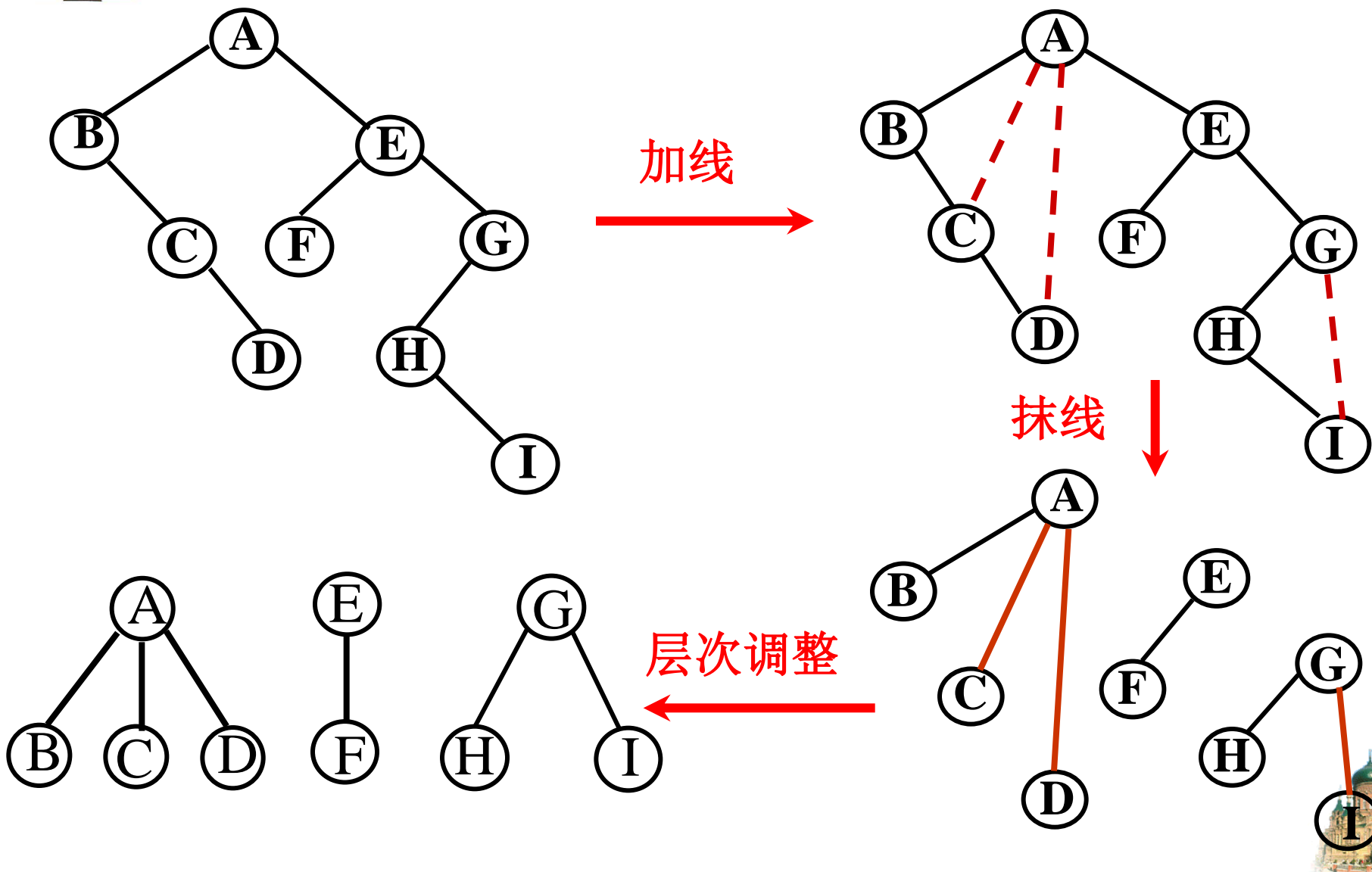
§ **旋转**: 按顺时针旋转45度角 (左链竖画, 右链横画)



旋转:



3.6 森林(树)与二叉树间的转换 (Cont.)





3.6 森林(树)与二叉树间的转换 (Cont.)

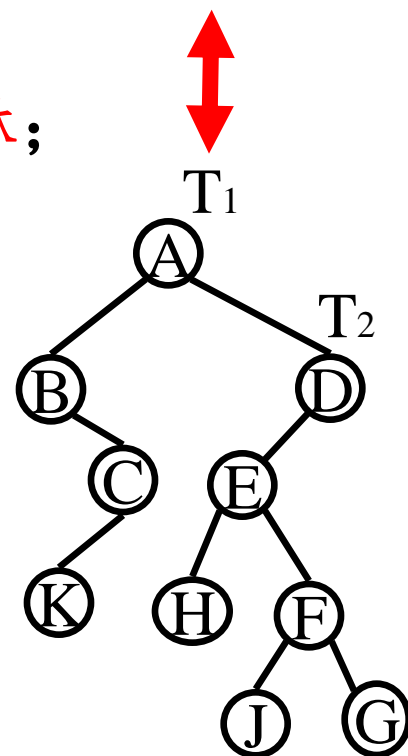
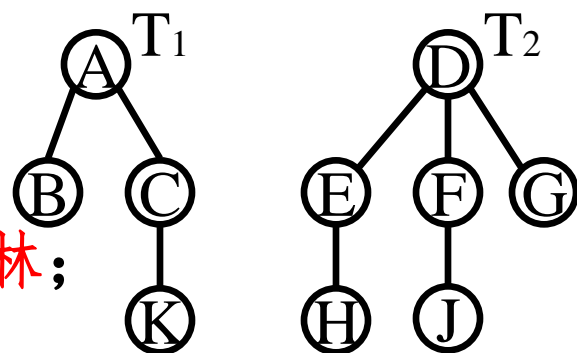
非空森林的基本遍历

先根遍历

- 访问第一株树的根结点;
- 按先根顺序遍历第一棵树的子树森林;
- 按先根顺序遍历其余子树森林。

后根遍历

- 按后根顺序遍历第一株树的子树森林;
- 访问第一株树的根结点;
- 按后根顺序遍历其余子树森林。



遍历	森林	树	二叉树
先序	√	√	√
中序	×	×	√
后序	√	√	√





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吶6325嘎6436腌7571钢7925埃1603挨1604哎1605唉1606哀1607皑1608癌1609谄1610矮1611
6441媛7040瑗7208暖7451砒7733琅7945霭8616鞍1616氨1617安1618俺1619按1620暗1621岸1622胺1623案
7281铵7907鹁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罄7081姦7365馨8190罄8292鳌8643鳌8701麇8773芭1637捌1638扒1639叭1640吧1641芭1642八
1649耙1650坝1651霸1652罢1653爸1654菱6056菰6135芭6517灞6917钹7857耙8446鲛8649魑8741白1655柏
1662捭6267呗6334翯7494斑1663班1664搬1665扳1666般1667颁1668板1669版1670扮1671拌1672伴1673鞭
7851瘢8103瘢8113版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16165孢
1713碑1714悲1715卑1716北1717辈1718背1719贝1720钡1721倍1722狈1723备1724惫1725焙1726被1727李
6703砒7753鹁8039梢8156璧8645鞣8725奔1728苯1729本1730笨1731奋5946盆5948贡7458铤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恣1749币1750庇1751痹
1758臂1759避1760陛1761匕5616俾5734萆6074萆6109薛6221吡6333哝6357甞6589庠6656愰6725滢6868渚
7815秘7873批7985裨8152筵8357算8375篁8387舩8416龔8437踔8547髀8734鞭1762边1763编1764贬1765扁
1772遍1773匾5650弁5945苳6048忤6677汴6774纒7134飏7614煊7652砭7730编7760窠8125褊8159蝙8289筵
7027骠7084杓7228咆7609飏7613鏖7958镗7980瘰8106裱8149鏖8707髡8752鳌1778愰1779别1780瘰1781璜
1787倌5747鹵6557缤7145玢7167缤7336缤7375缤7587缤7957髀8738髻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





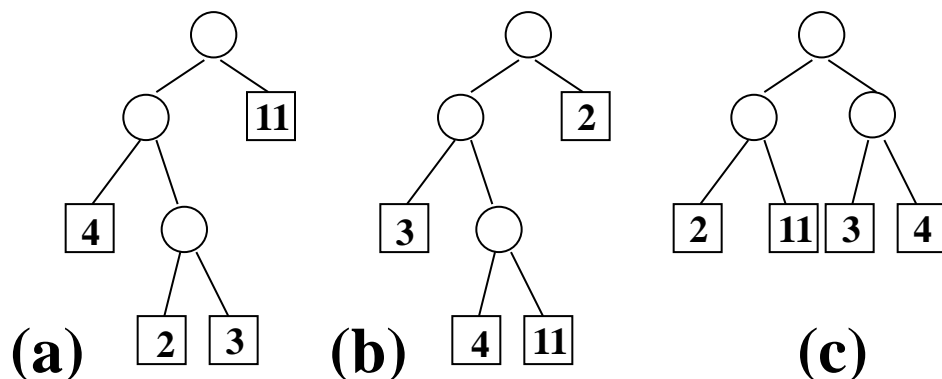
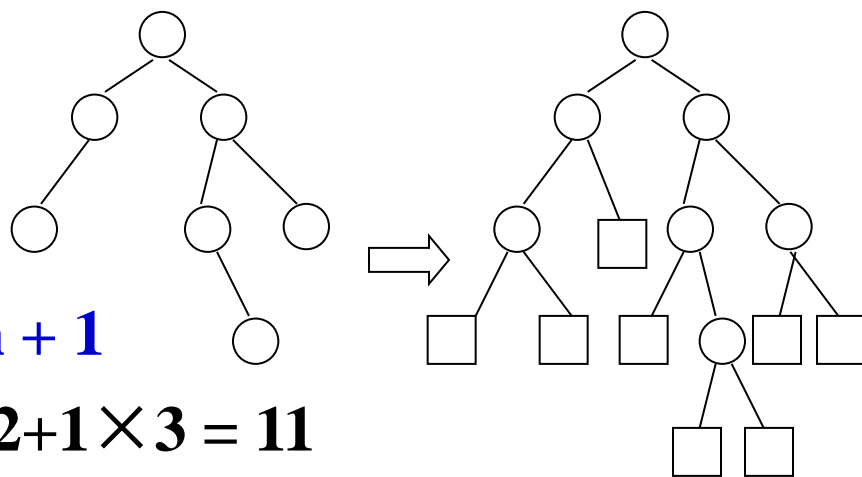
3.7.1 哈夫曼树及其应用

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 $I = 2 \times 1 + 3 \times 2 + 1 \times 3 = 11$ 外结点路径长度 $E = 1 \times 2 + 5 \times 3 + 2 \times 4 = 25$ 如内结点路径长度为 I , 则外结点路径长度 $E = I + 2 \times n$ 设: $w_i = \{ 2, 3, 4, 11 \}$ 求: $\sum w_j \cdot l_j$ (加权路长)(a) $11 \times 1 + 4 \times 2 + 2 \times 3 + 3 \times 3 = 34$ (b) $2 \times 1 + 3 \times 2 + 4 \times 3 + 11 \times 3 = 53$ (c) $2 \times 2 + 11 \times 2 + 3 \times 2 + 4 \times 2 = 40$

哈夫曼树 (最优二叉树): 在给定权值为 w_1, w_2, \dots, 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"
```

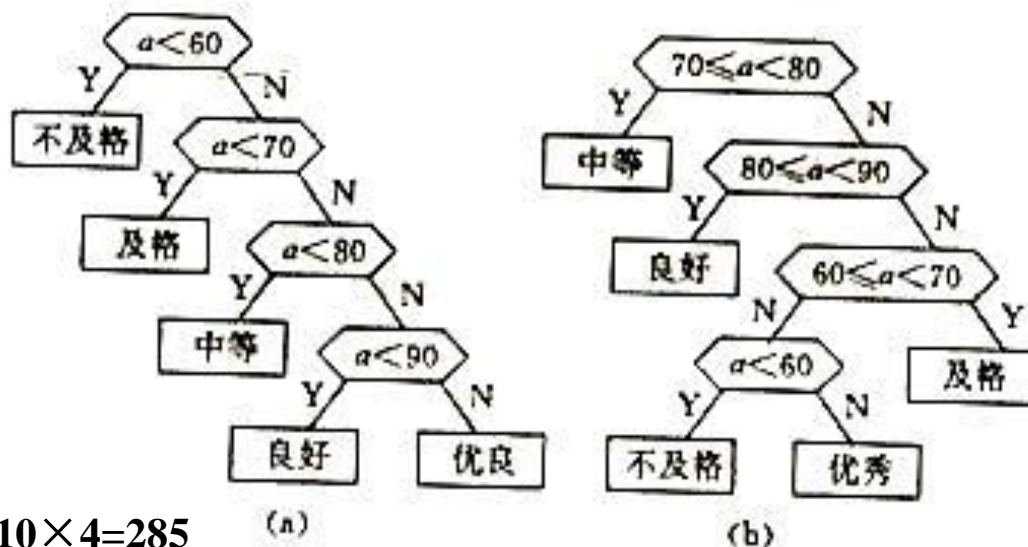
如图 (a) 所示





3.7 树型结构的应用 (Cont.)

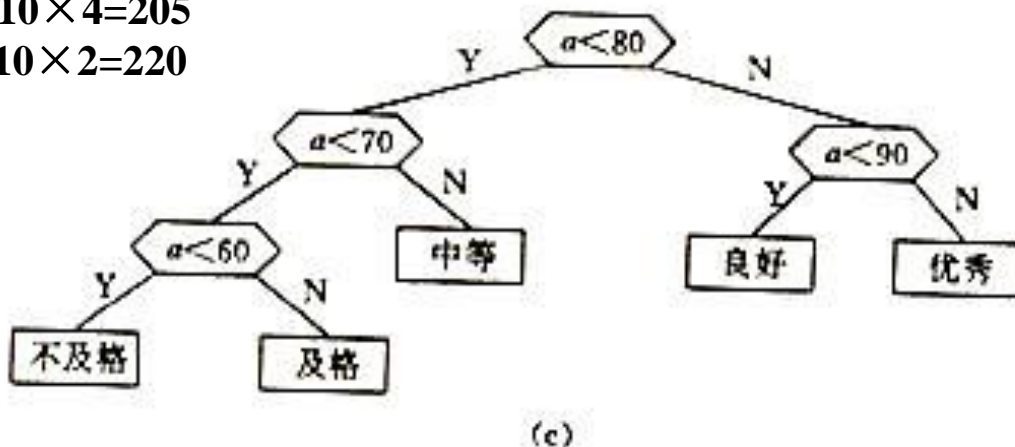
以5, 15, 40, 30, 10为权构造一株扩充二叉树如图(b)所示, 将判定框中的条件分开, 可得到(c), 从而实现判定过程的最优化。



(a) $5 \times 1 + 15 \times 2 + 40 \times 3 + 30 \times 3 + 10 \times 4 = 285$

(b) $40 \times 1 + 30 \times 2 + 15 \times 3 + 5 \times 4 + 10 \times 4 = 205$

(c) $5 \times 3 + 15 \times 3 + 40 \times 2 + 30 \times 2 + 10 \times 2 = 220$





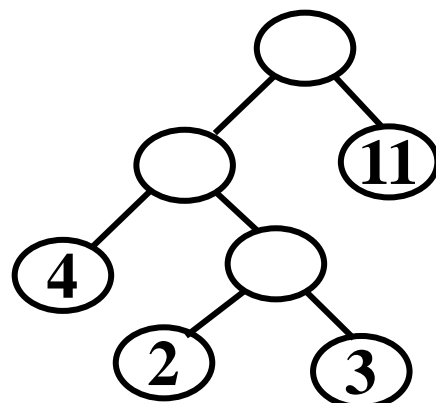
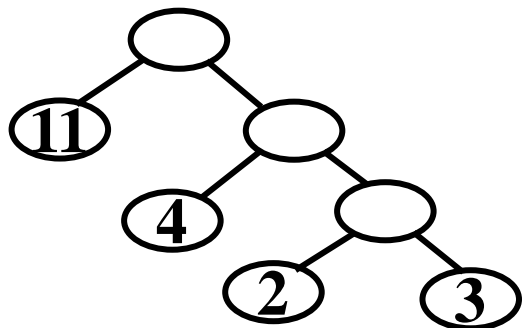
3.7 树型结构的应用 (Cont.)

哈夫曼树 (最优二叉树)

在给定权值为 $w_1, w_2 \dots w_n$ 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 $\{w_1, w_2, \dots, w_n\}$ 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 $F=\{T_1, T_2, \dots, T_n\}$;
- (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**。



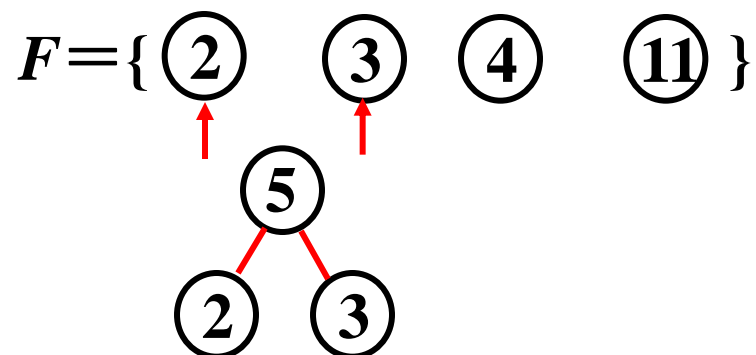


3.7 树型结构的应用 (Cont.)

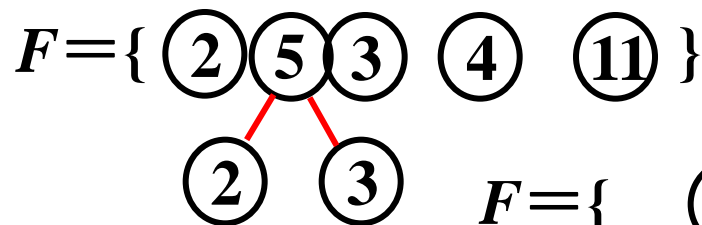
➤ 哈夫曼树的构造示例: $W=\{2, 3, 4, 11\}$

■ 初始化:

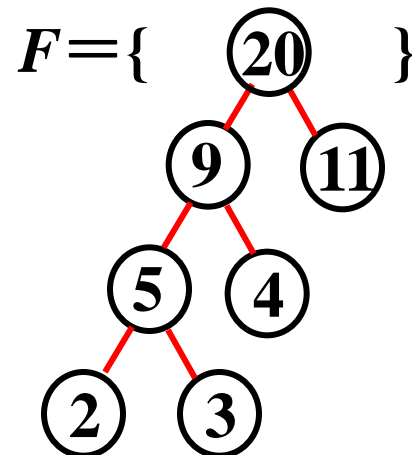
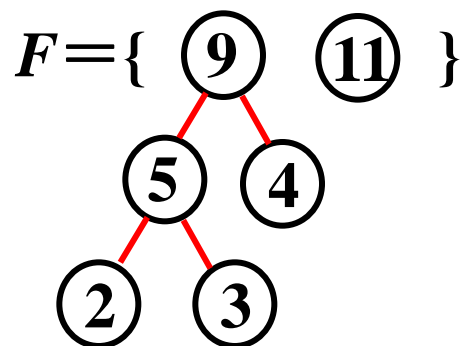
■ 选取与合并:



■ 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 ];
```

weight parent lchild rchild

0				
1				
2				
(2n-1)-1				

HuffmanT T;





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
⑦	0	7	-1	-1	-1
⑤	1	5	-1	-1	-1
②	2	2	-1	-1	-1
④	3	4	-1	-1	-1
	4		-1	-1	-1
	5		-1	-1	-1
	6		-1	-1	-1

初始化

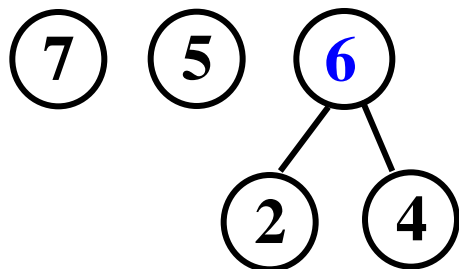




3.7 树型结构的应用 (Cont.)

➡ 哈夫曼树构造算法的实现示例:

	weight	parent	lchild	rchild
0	7	-1	-1	-1
1	5	-1	-1	-1
$p1 \rightarrow$ 2	2	4 -1	-1	-1
$p2 \rightarrow$ 3	4	4 -1	-1	-1
$i \rightarrow$ 4	6	-1	2 -1	3 -1
5		-1	-1	-1
6		-1	-1	-1



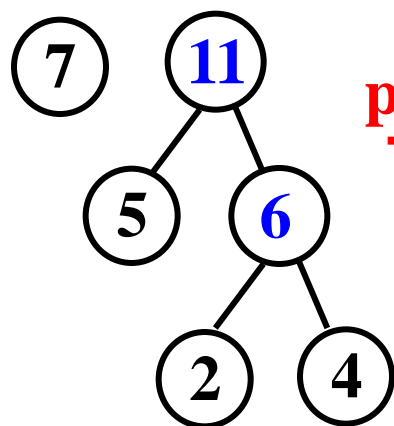
过程





3.7 树型结构的应用 (Cont.)

哈夫曼树构造算法的实现示例:



	weight	parent	lchild	rchild
0	7	-1	-1	-1
$p1 \rightarrow 1$	5	5 -1	-1	-1
2	2	4	-1	-1
3	4	4	-1	-1
$p2 \rightarrow 4$	6	5 -1	2	3
$i \rightarrow 5$	11	-1	1 -1	4 -1
6		-1	-1	-1

过程

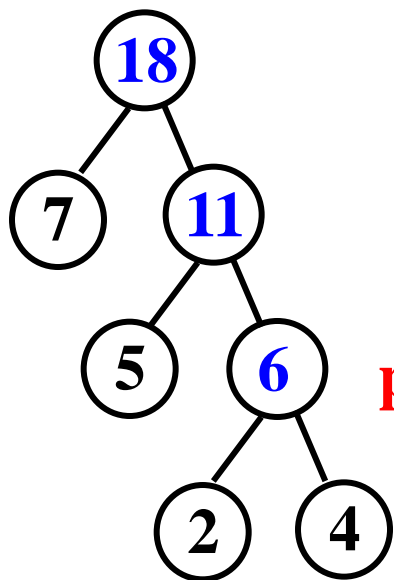




3.7 树型结构的应用 (Cont.)

哈夫曼树构造算法的实现示例:

	weight	parent	lchild	rchild
p1 → 0	7	6 -1	-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	6 -1	1	4
i → 6	18	-1	0 -1	5 -1



过程





3.7 树型结构的应用 (Cont.)

哈夫曼树构造算法的实现

void CreatHT(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
概率	0.45	0.13	0.12	0.16	0.09	0.05	码长
等长	000	001	010	011	100	101	3
变长	0	101	100	111	1101	1100	2.24

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

$$= \sum p_j \cdot l_j$$

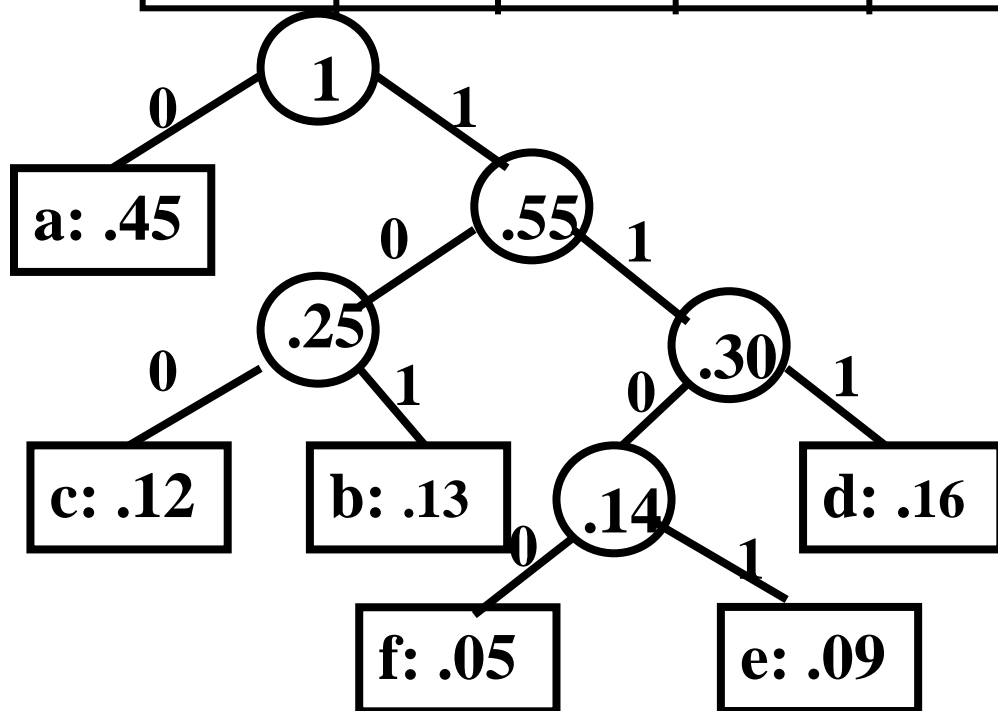




3.7 树型结构的应用 (Cont.)

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 = $\lceil \log_2 C \rceil$
变长	0	101	100	111	1101	1100	2.24 = $\sum p_j \cdot l_j$



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

编码表 H

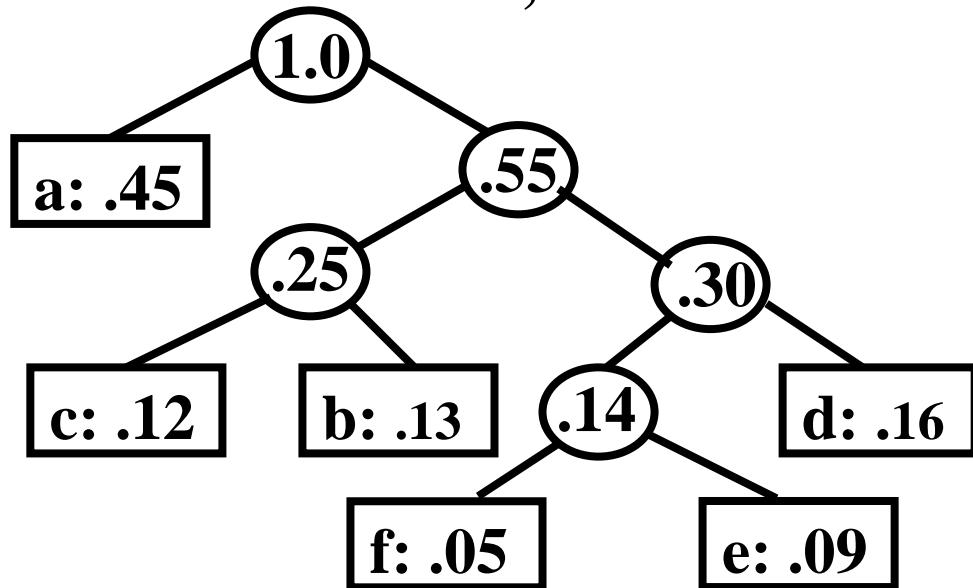




3.7 树型结构的应用 (Cont.)

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。





作业2 树型结构及其应用

➤ 作业题目：哈夫曼编码与译码方法

哈夫曼编码是一种以哈夫曼树（最优二叉树，带权路径长度最小的二叉树）为基础变长编码方式。其基本思想是：将使用次数多的代码转换成长度较短的编码，而使用次数少的采用较长的编码，并且保持编码的唯一可解性。在计算机信息处理中，经常应用于数据压缩。是一种一致性编码法（又称“熵编码法”），用于数据的无损耗压缩。要求实现一个完整的哈夫曼编码与译码系统。

➤ 作业要求：

- 从文件中读入任意一篇英文文本文件，分别统计英文文本文件中各字符（包括标点符号和空格）使用频率；
- 根据已统计的字符使用频率构造哈夫曼编码树，并给出每个字符的哈夫曼编码（字符集的哈夫曼编码表）；
- 将文本文件利用哈夫曼树进行编码，存储成压缩文件（哈夫曼编码文件）；
- 计算哈夫曼编码文件的压缩率；
- 将哈夫曼编码文件译码为文本文件，并与原文件进行比较。

以下可以不做，供思考，做了可以适当加分

- 能否利用堆结构，优化的哈夫曼编码算法。
- 上述1-5的编码和译码是基于字符的压缩，考虑基于单词的压缩，完成上述工作，讨论并比较压缩效果。
- 上述1-5的编码是二进制的编码，可以采用K叉的哈夫曼树完成上述工作，实现“K进制”的编码和译码，并与二进制的编码和译码进行比较。

