软件技术基础

第九讲

上讲主要内容

- 二叉树的存储结构
  - 顺序存储
  - 二叉树
- 二叉树的遍历

树

本章主要内容

二叉树的恢复
树的存储结构
森林与二叉树的转化
树的应用 哈夫曼树

从遍历序列恢复二叉树

[序列的特点]

- 先序遍历序列**第一个结点**必定是二叉树的**根结点**。
- 中序遍历序列已知的**根结点**将中序序列分割成两个子序列，根结点**左面的子序列**是**左子树的中序序列**，而在根结点**右边的子序列**是**右子树的中序序列**。

[基本过程]

- 利用先序序列确定根节点，在中序序列确定左子树的中序序列和左子树的中序序列。
- 根据左子树的结点的数目在先序序列中确定左子树的先序序列；同样得到右子树的先序序列。
- 如此反复，直到取尽先序序列中的结点时，便得到一棵二叉树。

从遍历序列恢复二叉树实例

先序序列为A, B, D, G, C, E, F, H，

中序序列为D, G, B, A, E, C, H, F

- A是二叉树的根结点，再根据A在中序序列中的位置，可知结点DGB在A的左子树上和ECHF在A的右子树上；
- 根据先序序列确定B和C分别是A的左子树和右子树的根，再根据B和C在DGB和ECHF中的位置，可知DG在B的左子树上和B的右子树为空以及C的左子树仅有结点E和C的右子树包含结点HF；
- 根据先序序列可知D是B的左子树的根和G是D的右子树的根，E是C的左子树的根和F是C的右子树的根；
- 最后确定H是F的左子树的根。

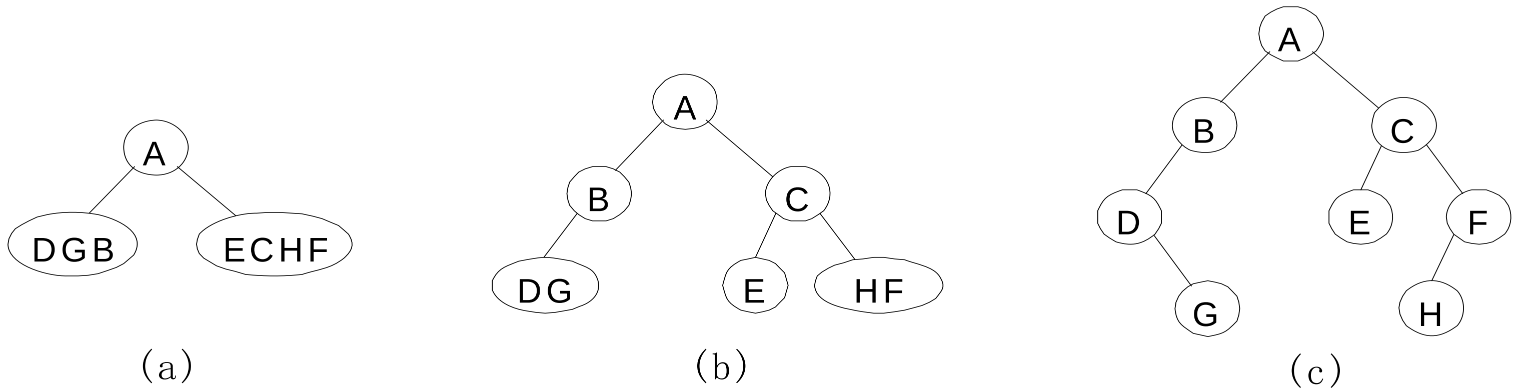


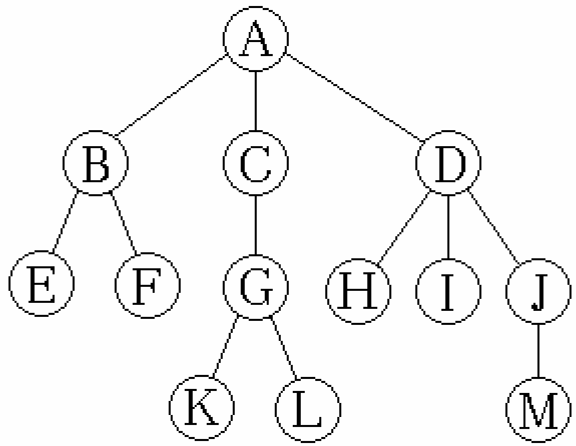
图 10.12　由先序和中序序列构造二叉树的过程

树的存储结构

- ☼双亲表示法
- ☼ 孩子表示法
- ☼孩子兄弟表示法

双亲表示法

用结构数组——树的顺序存储方式



(a) 树

1	A	0
2	B	1
3	C	1
4	D	1
5	E	2
6	F	2
7	G	3
8	H	4
9	I	4
10	J	4
11	K	7
12	L	7
13	M	10

双亲表示法

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

找双亲方便，找孩子难

孩子表示法

顺序和链式结合的表示方法

- 定长结构 （n为树的度）指针利用率不高

data	***$child_1$***	***$child_2$***	***$child_3$***	▪▪▪▪▪▪▪▪	***$child_n$***

- 不定长结构 d为结点的度，节省空间，但算法复杂

data	***d***	***$child_2$***	***$child_3$***	▪▪▪▪▪▪▪▪	***$child_d$***

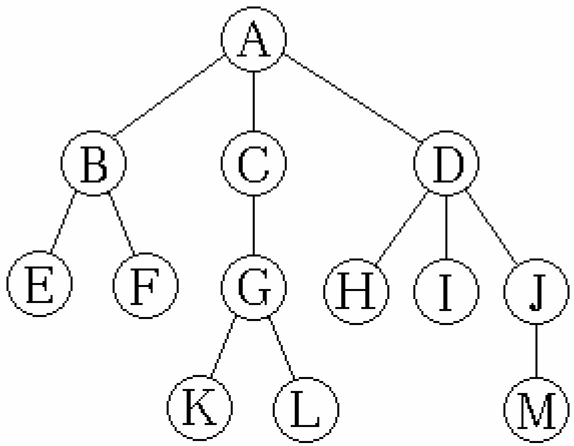
- 一般采用定长结构
  - 如有n个结点，树的度为k，则共有n*k个指针域，只有n-1个指针域被利用，而未利用的指针域为：n*k-(n-1) =n(k-1)+1，未利用率为：(n(k-1)+1)/nk ＞ n(k-1)/nk=(k-1)/k
    二次树：1/2 ； 三次树：2/3；四次树：3/4
  - 树的度越高，未利用率越高，由于二叉树的利用率较其他树高，因此用二叉树。

孩子表示法

- 把每个结点的孩子结点排列起来，看成一个线性表，用单链表作存储结构。
- n个结点有n个孩子链表
- n个头指针构成线性表，采用顺序的存储结构

孩子表示法

1	A		2	3	4 ^
2	B		5	6 ^	
3	C		7 ^		
4	D		8	9	10 ^
5	E	^			
6	F	^			
7	G		11	12 ^	
8	H	^			
9	I	^			
10	J		13 ^		
11	K	^			
12	L	^			
13	M	^			



（a）树

孩子表示法

```
typedef struct CTNode {
int child;
struct  CTNode *next;
} *ChildPtr;
typedef struct {
TElemType data;
ChildPtr firstchild;
}CTBox;
Typedef struct {
    CTBox nodes[MAX_TREE_SIZE];
    int r,n;
}CTree;
```

/*孩子结点*/

/* 孩子链表头指针*/

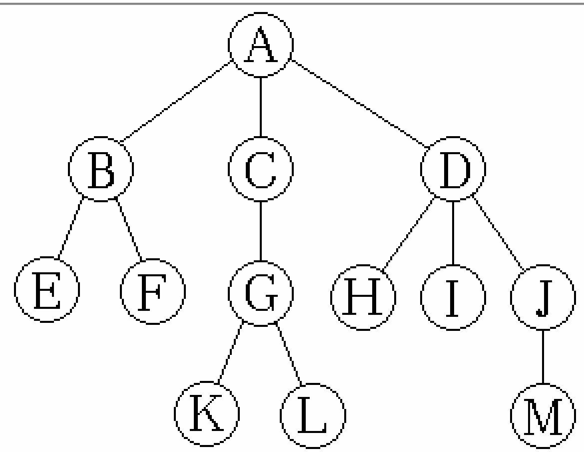
/*根的位置和结点数*/

✓找孩子方便，找双亲难

✓将双亲链表和孩子链表结合

孩子表示法

1	0	A		1		2		3	^
2	1	B		4		5	^		
3	1	C		6	^				
4	1	D		7		8		9	^
5	2	E	^						
6	2	F	^						
7	3	G		K		L	^		
8	4	H	^						
9	4	I	^						
10	4	J		12	^				
11	7	K	^						
12	7	L	^						
13	10	M	^						

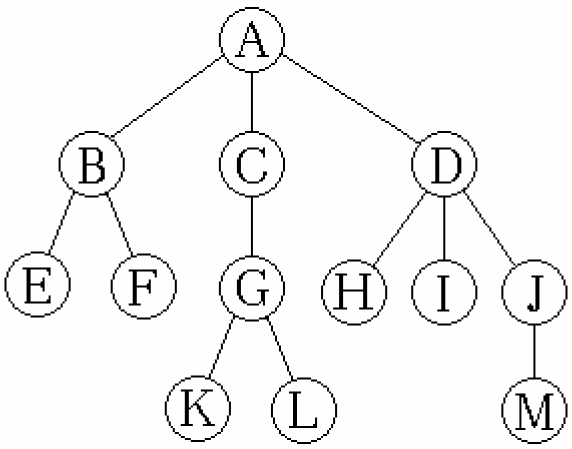


（a）树

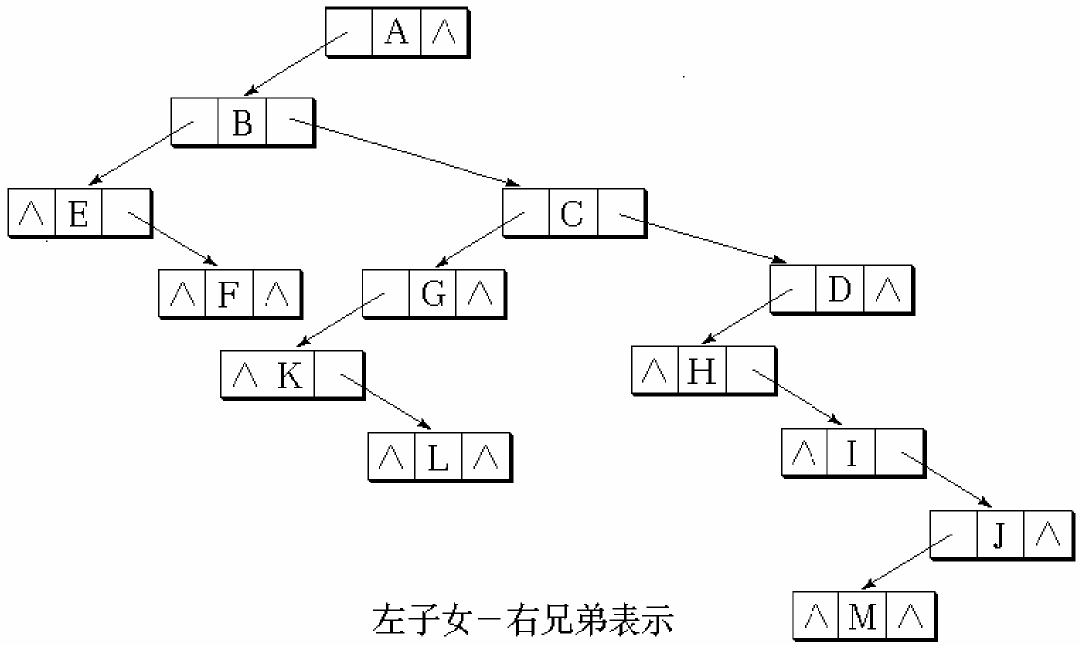
孩子兄弟表示法

二叉链表作为树的存储结构

- 该结点的第一个孩子
- 下一个兄弟



（a）树



左子女－右兄弟表示

孩子表示法

data	***firstChild***	***nextSibling***

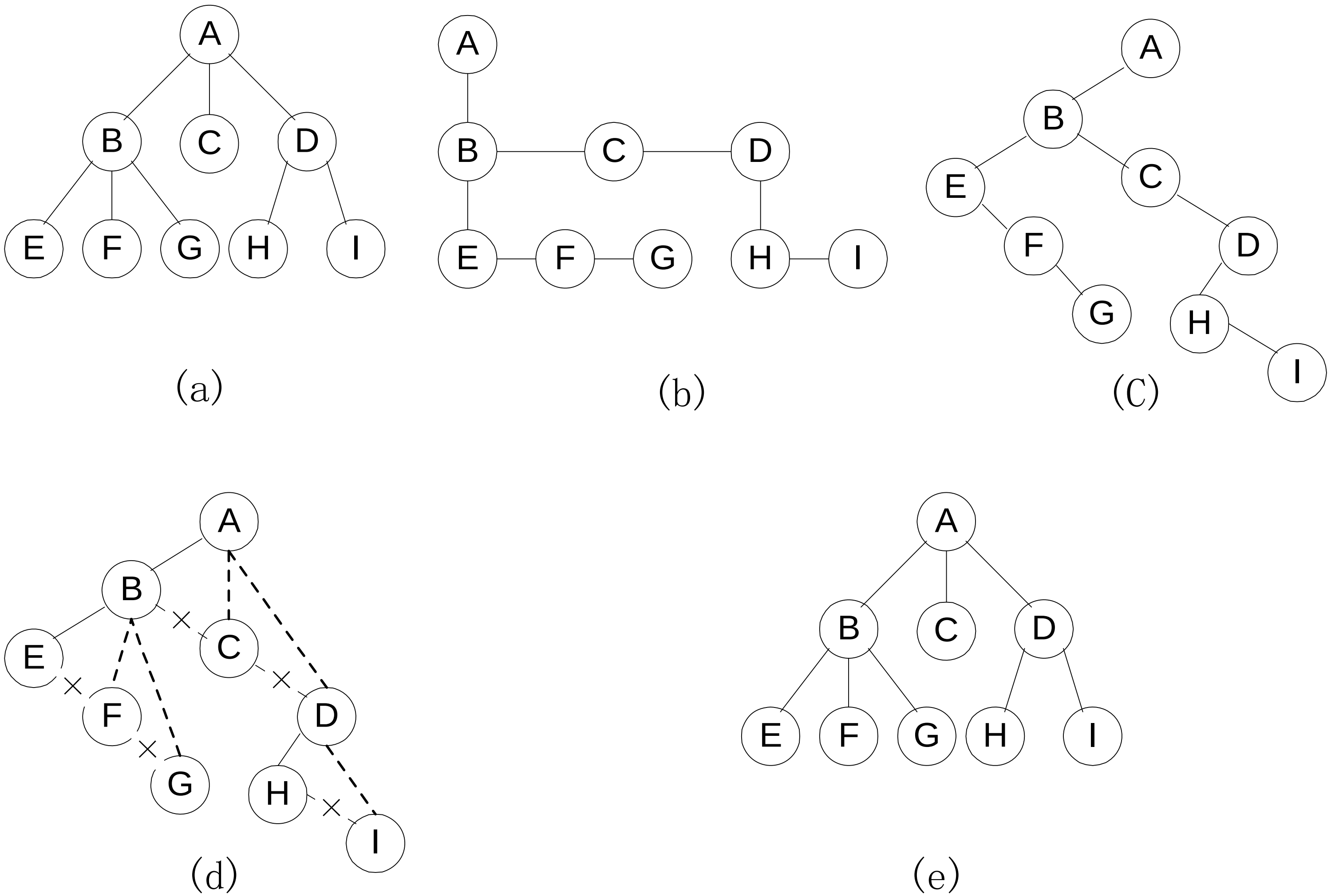
```
typedef struct CSNode{
ElemType  data;
struct CSNode  *firstchild, *nextsibling;
}CSNode,*CSTree;
```

✓找孩子容易，若增加parent域，则找双亲也较方便。

树和二叉树间的转换

- 任何一棵树都可以转换为一棵二叉树，同时一棵二叉树也可转换成一棵树，而且这种转换是具有惟一性的。
- 将一棵树转换为二叉树的方法是：
  - (1)在兄弟之间增加一条连线；
  - (2)对每个结点，除了保留与其左孩子的连线外，除去与其它孩子之间的连线；
  - (3)以树的根结点为轴心，将整个树顺时针旋转45度。
- 从一棵二叉树到树的转换规则是：
  - (1)若结点X是双亲Y的左孩子，则把X的右孩子，右孩子的右孩子…都与Y用连线相连；
  - (2)去掉原有的双亲到右孩子的连线。

树和二叉树间的转换实例



树与二叉树相互转换示例

树和森林的遍历

树的遍历（先根遍历、后根遍历、层次遍历）

（1）先根遍历：先根访问树的根结点，然后依次先根遍历根的每棵子树

（2）后根遍历：先依次后根遍历每棵子树，然后访问根结点

树的先根遍历 ⟷ 转换后的二叉树的先序遍历

树的后根遍历 ⟷ 转换后的二叉树的中序遍历

树和森林的遍历

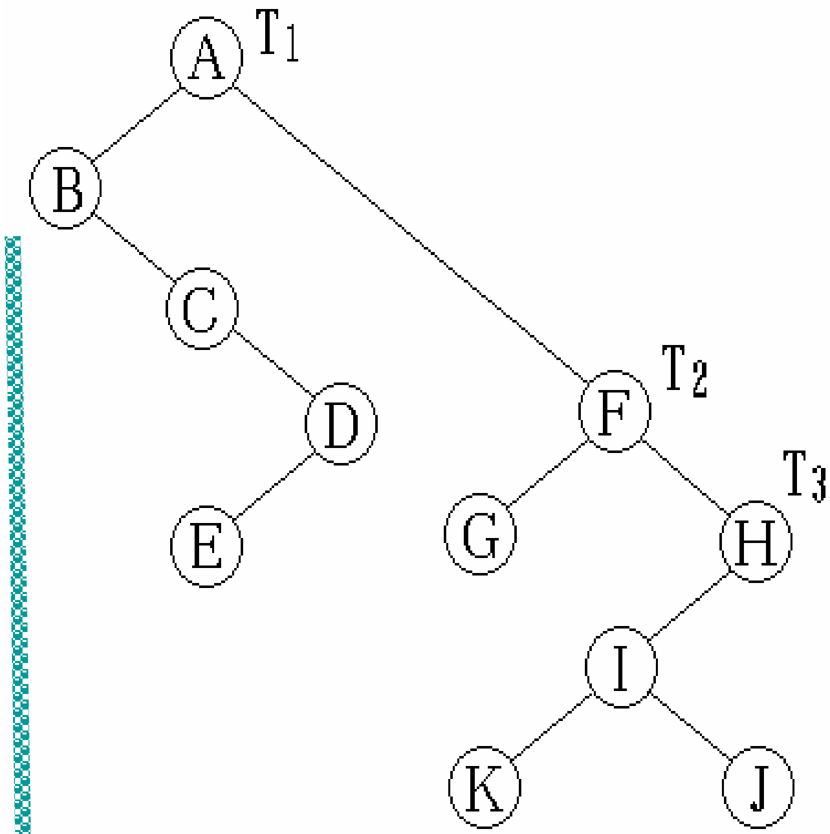
森林的遍历

- 先序：对应二叉树的先序遍历
- 后序：对应二叉树的后序遍历

森林的遍历

(1) 先根次序遍历的规则:

- 若森林F为空, 返回; 否则
  - ☞ 访问F的第一棵树的根结点;
  - ☞ 先根次序遍历第一棵树的子树森林;
  - ☞ 先根次序遍历其它树组成的森林。



森林的二叉树表示

森林的遍历

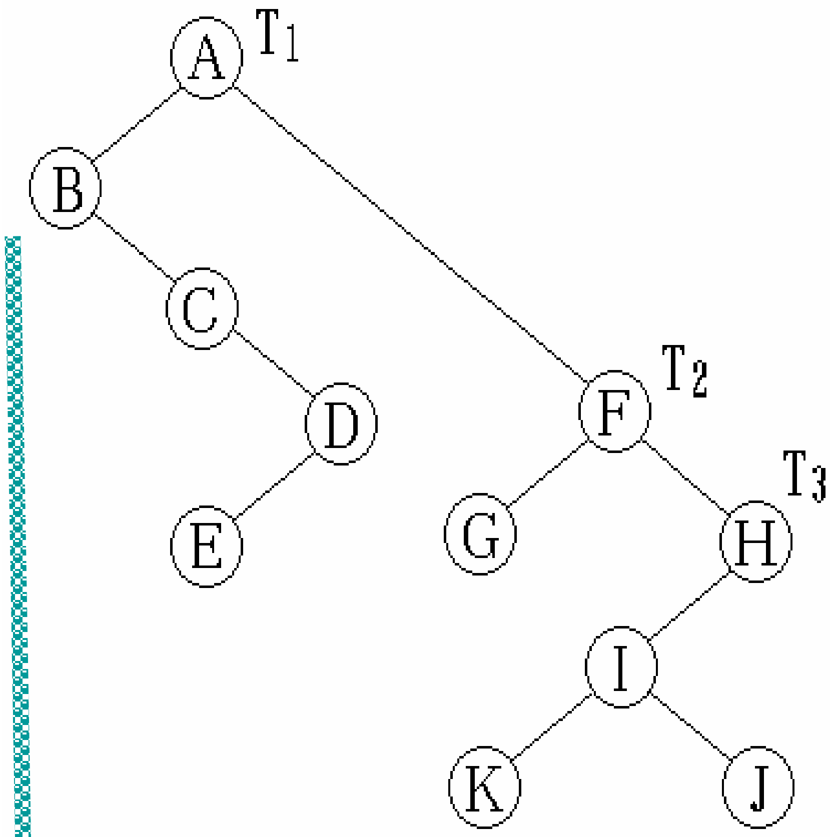
(2) 中根次序遍历的规则:

❒ 若森林F为空，返回；否则

☞ 中根次序遍历第一棵树的子树森林;

☞ 访问F的第一棵树的根结点;

☞ 中根次序序遍历其它树组成的森林。



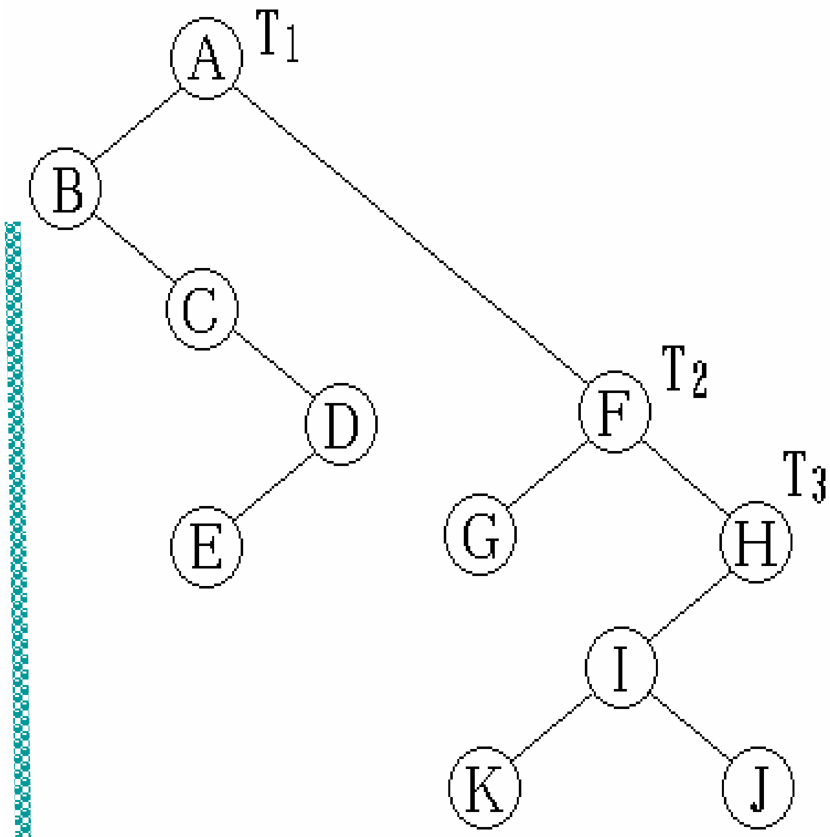
森林的二叉树表示

森林的遍历

(3) 后根次序遍历的规则:

- 若森林F为空，返回；否则
  - ☞ 后根次序遍历第一棵树的子树森林;
  - ☞ 后根次序遍历其它树组成的森林;
  - ☞ 访问F的第一棵树的根结点。

。



森林的二叉树表示

哈夫曼树

[概念]

- 若树中的两个结点之间存一条路径，则路径的长度是指路径所经过的边(即连接两个结点的线段)的数目。
- 树的路径长度是树根到树中每一结点的路径长度之和。
- 树的带权路径长度为树中所有叶子结点的带权路径长度之和，记作：

$$WPL = \sum_{i=1}^{n} w_i l_i$$

其中n为树中叶子结点的数目，w_i为叶子结点i的权值，l_i为叶子结点i到根结点之间的路径长度。

哈夫曼树的定义

[定义]

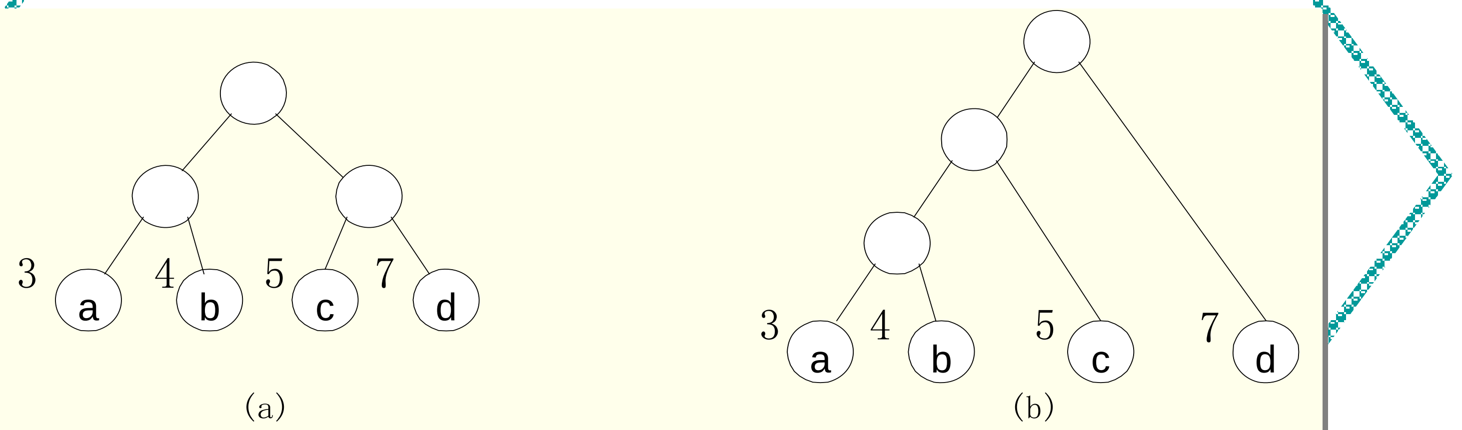
在有n个带权叶子结点的所有二叉树中，带权路径长度**WPL最小**的二叉树被称为**最优二叉树或哈夫曼树**。

[两种遍历方式]

在结点数目相同的二叉树中，完全二叉树的路径长度最短。如果对于一般二叉树，以带权路径长度为度量，如何构造二叉树才能使其路径长度最短？

哈夫曼树的不惟一性

权值为$w_1, w_2, \ldots, w_n$ 的n个叶子结点形成的二叉树，可以具有多种形态，其中能被称为哈夫曼树的二叉树并不是惟一的。如下图。

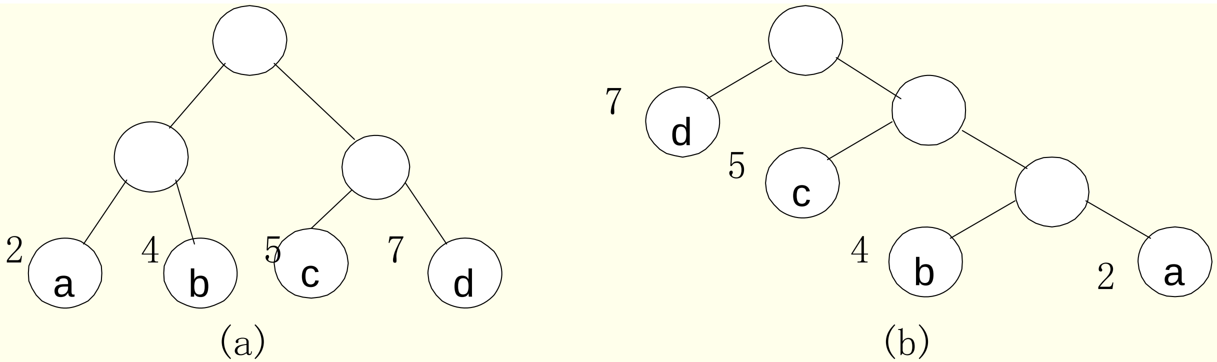


不同形态的哈夫曼树

(a) WPL =3×2+4×2+5×2+7×2=38；
(b) WPL =3×3+4×3+5×2+7=38。

完全二叉树不一定是哈夫曼树

在叶子数和权值相同的二叉树中，完全二叉树不一定是最优二叉树。如下图。



完全二叉树与哈夫曼树

(a) WPL＝2×2＋4×2＋5×2＋7×2＝36；

(b) WPL＝2×3＋4×3＋5×2＋7×1＝35。

哈夫曼树的构造算法

(1)根据给定的n个权值$\{w_1, w_2,\ldots,w_n\}$构成n棵二叉树的集合$F=\{T_1,T_2,\ldots, T_n\}$，其中$T_i$中只有一个权值为$w_i$的根结点，左、右子树均为空。

(2)在F中选取两棵根结点的权值为最小的树作为左、右子树构造一棵新的二叉树，且置新的二叉树的根结点的权值为左、右子树上根结点的权值之和。

(3)在F中删除这两个棵权值为最小的树，同时将新得到的二叉树加入F中。

(4) 重复(2)、(3)直到F中仅剩一棵树为止。这棵树就是哈夫曼树。

哈夫曼树的构造实例

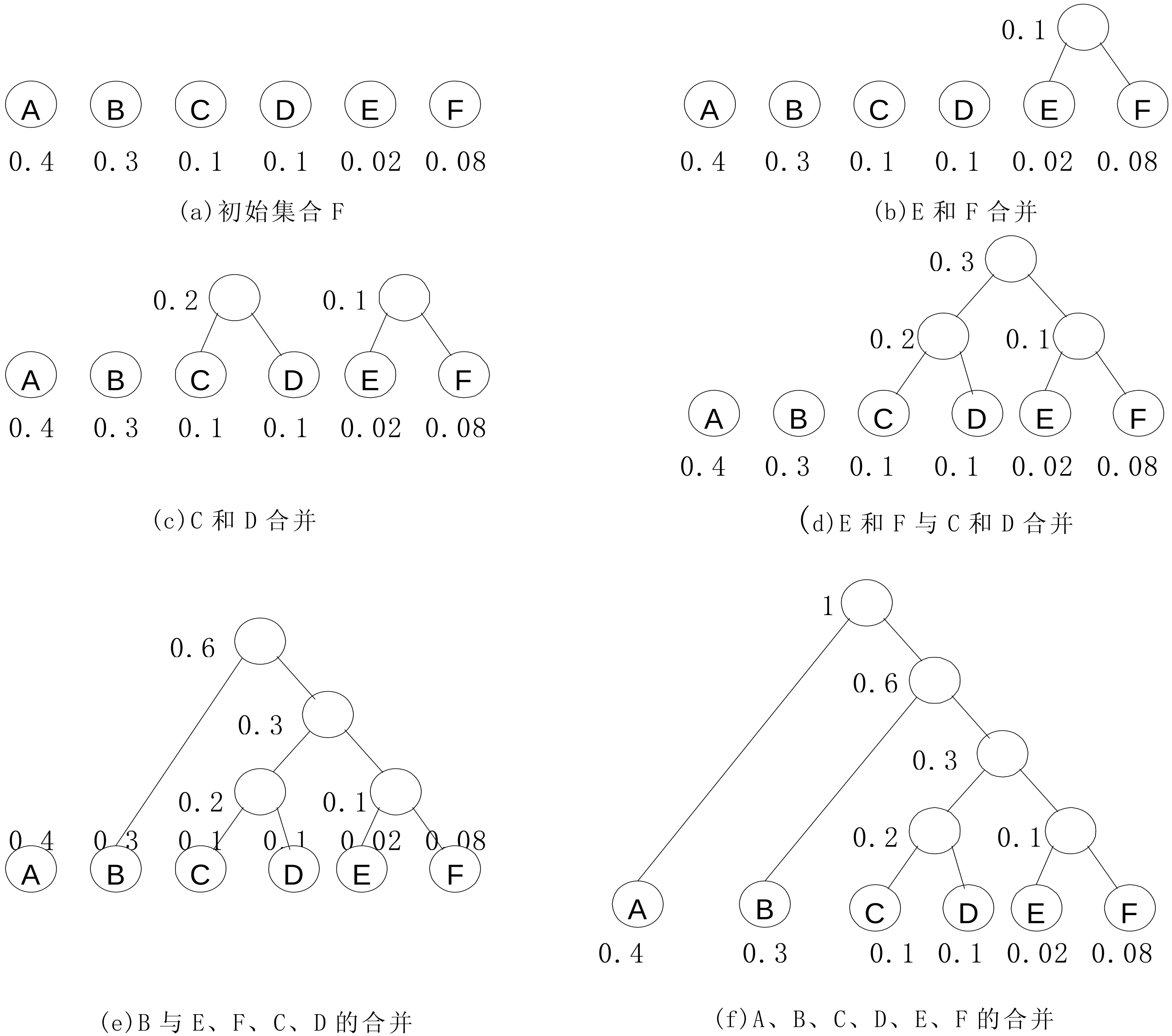
- 权值为0.4、0.3、0.1、0.1、0.02、0.08的6个叶子结点A、B、C、D、E、F构造哈夫曼树来观察哈夫曼树的构造过程，如下图所示。

注：

– 一个有**n**个叶子结点的初始集合，要生成哈夫曼树共要进行**n-1**次合并，产生**n-1**个新结点。

– 最终求得的哈夫曼树共有**2n-1**个结点，并且哈夫曼树中没有度为1的分支结点。

- 我们常称没有度为1的结点的二叉树为严格二叉树。


A B C D E F
0.4 0.3 0.1 0.1 0.02 0.08
(a)初始集合 F
0.1
A B C D E F
0.4 0.3 0.1 0.1 0.02 0.08
(b)E 和 F 合并
0.2 0.1
A B C D E F
0.4 0.3 0.1 0.1 0.02 0.08
(c)C 和 D 合并
0.3
0.2 0.1
A B C D E F
0.4 0.3 0.1 0.1 0.02 0.08
(d)E 和 F 与 C 和 D 合并
0.6
0.3
0.2 0.1
0.4 0.3 0.1 0.1 0.02 0.08
A B C D E F
(e)B 与 E、F、C、D 的合并
1
0.6
0.3
0.2 0.1
A B C D E F
0.4 0.3 0.1 0.1 0.02 0.08
(f)A、B、C、D、E、F 的合并

哈夫曼树的构造过程

构造哈夫曼树算法中的数据结构

```
#define n                          /*叶子数目*/
#define m  2*n-1                   /* 结点总数 */

typedef  char datatype;
typedef  struct
    { float      weight;
      datatype  data;
      int          lchild, rchild, parent;
    } hufmtree;
hufmtree  tree[m];
```

构造哈夫曼树算法实现

```
HUFFMAN (hufmtree tree[ ])
{    int  i, j, p;
     char  ch;
     float small1, small2, f;
     for ( i=0; i<m; i++)                          /* 初始化 */
        {   tree[i].parent=0;
            tree[i].lchild=0;
            tree[i].rchild=0;
            tree[i].weight=0.0;
             tree[i].data= '0';
        }
      for ( i=0; i<n; i++)                         /* 输入前n个结点的权值 */
        {    scanf(" %f ", &f);
             tree[i].weight=f;  scanf(" %c ",&ch);
             tree[i].data=ch;
        }
```

构造哈夫曼树算法实现

```
for ( i=n; i<m ;i++ )                          /* 进行n-1次合并，产生n-1个新结点 */
 {   p1=p2=0;
     small1=small2=Maxval;                     /* Maxval是float类型的最大值 */
     for ( j=0; j<=i-1; j++ )
        if  ( tree[j].parent= =0)
            if ( tree[j].weight<small1 )       /* 改变最小权，次最小权及对应位置 */
               {     small2=small1;
                     small1=tree[j].weight;
                      p2=p1;
                      p1=j;
               }
            else
               if ( tree[j].weight<small2 )    /*改变次小权及位置 */
                  {   small2=tree[j].weight;
                      p2=j;
                  }
```

构造哈夫曼树算法实现

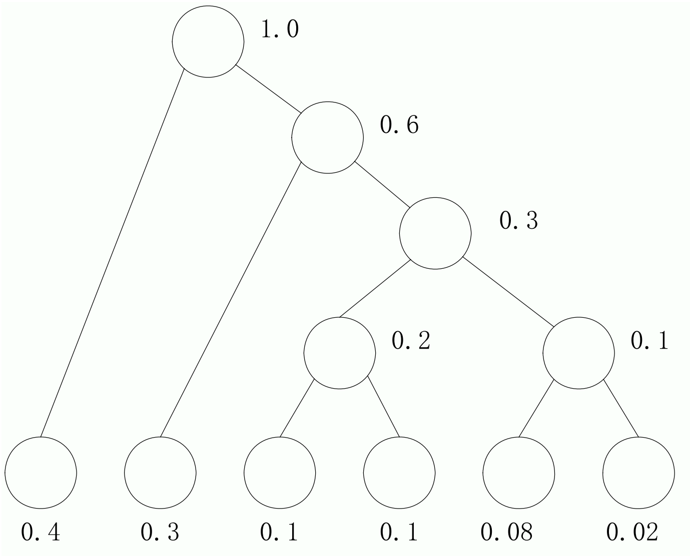
```
        tree[p1].parent=i;
        tree[p2].parent=i;
        tree[i].lchild=p1;
        tree[i].rchild=p2;
        tree[i].weight = tree[p1].weight+tree[p2].weight
    }
}
```

/* 给合并的两个结点的双亲域赋值 */

/* HUFFMAN */

构造哈夫曼树算法tree[]的变化过程

下图中的叶子结点集合构造哈夫曼树的初始状态如左图(a)所示，第一次合并状态如左图(b)所示，结果状态如左图(c)所示。



数组下标	lchild	data	weight	rchild	parent
0	0	A	0.4	0	0
1	0	B	0.3	0	0
2	0	C	0.1	0	0
3	0	D	0.1	0	0
4	0	E	0.02	0	0
5	0	F	0.08	0	0
6	0	‘0’	0	0	0
7	0	‘0’	0	0	0
8	0	‘0’	0	0	0
9	0	‘0’	0	0	0
10	0	‘0’	0	0	0

(a)

数组下标	lchild	data	weight	rchild	parent
0	0	A	0.4	0	0
1	0	B	0.3	0	0
2	0	C	0.1	0	0
3	0	D	0.1	0	0
4	0	E	0.02	0	6
5	0	F	0.08	0	6
6	4	‘0’	0.1	5	0
7	0	‘0’	0	0	0
8	0	‘0’	0	0	0
9	0	‘0’	0	0	0
10	0	‘0’	0	0	0

(b)

数组下标	lchild	data	weight	rchild	parent
0	0	A	0.4	0	10
1	0	B	0.3	0	9
2	0	C	0.1	0	7
3	0	D	0.1	0	7
4	0	E	0.02	0	6
5	0	F	0.08	0	6
6	4	‘0’	0.1	5	8
7	2	‘0’	0.2	3	8
8	6	‘0’	0.3	7	9
9	1	‘0’	0.6	8	10
10	0	‘0’	1	9	0

(c)

图 10.18　哈夫曼树的初始，第一次合并和结果状态

建立赫夫曼树及求赫夫曼编码的算法

```
typedef struct {
          unsigned int weight;
          unsigned int parent,lchild,rchild;
          }  HTNode, *HuffmanTree;
typedef char  **HuffmanCode;

void HuffmanCoding(HuffmanTree &HT,HuffmanCode &HC,int *w, int n)
 {  HuffmanTree p;   char *cd;       int i,s1,s2,start;
    unsigned int c,f;
    if (n<=1) return;      // n为字符树木，m为结点树木
    int m=2*n-1;
    HT = (HuffmanTree)malloc((m+1)*sizeof(HTNode));
                                     //  0号单元未用
```

```
 for (p=HT, i=1; i<=n; ++i,++p,++w)
    { p->weight = *w; p->parent=0; p->lchild=0;p->rchild=0; }
                // *p = { *w,0,0,0 };
for (; i<=m;++i,++p)
   {  p->weight = 0; p->parent=0;  p->lchild=0; p->rchild=0; }
               //*p={ 0,0,0,0 };
for (i=n+1; i<=m;++i)    //  建赫夫曼树
   {    Select(HT,i-1,s1,s2);
        HT[s1].parent=i;  HT[s2].parent = i;
        HT[i].lchild = s1;  HT[i].rchild = s2;
        HT[i].weight = HT[s1].weight + HT[s2].weight;
    }
```

```
        //从叶子到根逆向求赫夫曼编码
  HC=  (HuffmanCode)malloc((n+1)*sizeof(char *));
  cd = (char*)malloc(n*sizeof(char));
  cd[n-1]='\0';
  for (i=1;i<=n;++i)
     {  start = n-1;
        for (c=i,f=HT[c].parent; f!=0; c=f,f=HT[f].parent)
        if (HT[f].lchild ==c) cd[--start]='0';
        else cd[--start]='1';
        HC[i]=(char *)malloc((n-start)*sizeof(char));
        strcpy(HC[i],&cd[start]);
        printf("%s\n",HC[i]);
      }
  free(cd);
}
```

小结

二叉树的遍历

树的存储
- 双亲
- 孩子
- 孩子兄弟

树和二叉树转换

树的遍历

作业

6.2 ,**6.13，6.20，6.21，**
6.26，6.27，6.42，6.45